

Rational Application Developer



EGL リファレンス・ガイド

バージョン 6 リリース 001

Rational Application Developer



EGL リファレンス・ガイド

バージョン 6 リリース 001

ご注意

本書および本書で紹介する製品をご使用になる前に、1151 ページの『特記事項』に記載されている情報をお読みください。

このエディションは Rational Web Developer バージョン 6 リリース 0 モディフィケーション 0 および Rational Application Developer に適用され、新しいエディションで明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

本マニュアルに関するご意見や感想は、次の URL からお送りください。今後の参考にさせていただきます。

<http://www.ibm.com/jp/manuals/main/mail.html>

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

<http://www.ibm.com/jp/manuals/> の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原 典： SC31-6839-02
Rational Application Developer
EGL Reference Guide
Version 6 Release 001

発 行： 日本アイ・ビー・エム株式会社

担 当： ナショナル・ランゲージ・サポート

第1刷 2005.4

この文書では、平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財)日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W7、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

© Copyright International Business Machines Corporation 1996, 2005. All rights reserved.

© Copyright IBM Japan 2005

目次

概説	1
EGL の紹介	1
EGL 6.0.0.1 の新機能	2
EGL 6.0 iFix の新機能	3
EGL バージョン 6.0 の新機能	5
開発過程	9
ランタイム構成	10
Java ラッパーの使用	10
有効な呼び出し	11
有効な転送	12
EGL に関する追加情報のソース	13
EGL 言語の概説	15
EGL プロジェクト、パッケージ、およびファイル	15
EGL プロジェクト	15
パッケージ	16
EGL ファイル	17
推奨	18
パーツ	19
パーツの参照	24
固定構造体	28
Typedef	30
インポート	36
背景	36
import ステートメントの形式	36
プリミティブ型	37
宣言時におけるプリミティブ型	39
さまざまな数値型の相対的な効率	39
ANY	41
文字型	41
日時型	44
LOB の型	51
数値型	53
EGL での変数と定数の宣言	57
動的アクセスと静的アクセス	58
有効範囲指定の規則と EGL での「this」	60
EGL での変数の参照	62
動的アクセス用の大括弧構文	64
固定構造体を参照するための簡略構文	66
EGL プロパティの概要	68
フィールド表示プロパティ	70
フォーマット設定プロパティ	71
SQL 項目のプロパティ	71
検証プロパティ	72
値の設定ブロック	72
基本的な状態での値の設定ブロック	72
フィールドのフィールド用の値の設定ブロック	74
「this」の使用	75
値の設定ブロック、配列、および配列エレメント	76
追加の例	77
配列	78

動的配列	78
構造体フィールド配列	82
辞書	87
辞書のプロパティ	88
辞書関数	90
ArrayDictionary	91
EGL ステートメント	93
キーワードのアルファベット順リスト	96
プログラム間での制御の移動	98
例外処理	100
try ブロック	100
EGL システム例外	100
try ブロックの制限	101
エラー関連システム変数	102
I/O ステートメント	104
エラーの識別	105

EGL 6.0 iFix への EGL コードのマイ グレーション	107
EGL から EGL へのマイグレーション	108
EGL から EGL へのマイグレーション時における プロパティの変更	112
EGL 間マイグレーション設定の変更	117

環境のセットアップ	119
EGL 設定の変更	119
テキストの設定の変更	119
EGL デバッガーの設定の変更	120
デフォルトのビルド記述子の設定	122
EGL エディターの設定の変更	123
ソース・スタイルの設定の変更	124
テンプレートの設定の変更	124
SQL データベース接続設定の変更	126
SQL 検索設定の変更	128
EGL 機能の使用可能化	129

コード開発の開始	131
プロジェクトの作成	131
EGL プロジェクトの作成	131
EGL Web プロジェクトの作成	132
プロジェクト作成時のデータベース・オプション の指定	133
EGL ソース・フォルダーの作成	134
EGL パッケージの作成	134
EGL ソース・ファイルの作成	135
コンテンツ・アシストを使用した EGL テンプレ ートの使用	135
EGL のキーボード・ショートカット	136

基本 EGL ソース・コードの開発	139
EGL dataItem パーツの作成	139

DataItem パーツ	139
EGL レコード・パーツの作成	140
レコード・パーツ	141
固定レコード・パーツ	142
レコード・タイプとプロパティ	143
EGL プログラム・パーツの作成	146
プログラム・パーツ	147
EGL 関数パーツの作成	149
関数パーツ	149
EGL ライブラリー・パーツの作成	150
basicLibrary タイプのライブラリー・パーツ	151
nativeLibrary タイプのライブラリー・パーツ	152
EGL dataTable パーツの作成	154
DataTable	155

EGL および JSP ファイルへのコードの断片の挿入 157

書式フィールドへのフォーカスの設定	158
セッション変数の有無についてのブラウザのテスト	158
データ・テーブル内のクリックされた行の値の検索	159
リレーショナル・テーブル内の行の更新	160

テキスト書式と印刷書式による作業 . . . 163

EGL formGroup パーツの作成	163
FormGroup パーツ	163
書式パーツ	164
EGL 印刷書式の作成	165
EGL テキスト書式の作成	168
EGL 書式エディターの概要	174
EGL 書式エディターを使用した書式グループの編集	175
フィルターの作成	176
EGL 書式エディターを使用した書式の作成	177
定数フィールドの作成	178
印刷書式またはテキスト書式での変数フィールドの作成	179
EGL 書式エディターのパレット・エントリーの設定	181
EGL 書式エディターの書式テンプレート	181
EGL 書式エディターの表示オプション	186
EGL 書式エディターの設定	186
EGL 書式エディターの書式フィルター	187

コンソール・ユーザー・インターフェースの作成 189

コンソール・ユーザー・インターフェース	189
consoleUI を使用したインターフェースの作成	190
ConsoleUI パーツおよび関連変数	192
Window	192
Prompt	193
ConsoleField	193
ConsoleForm	193
ConsoleUI での new の使用	195
UNIX 用の ConsoleUI 画面オプション	196

EGL Web アプリケーションの作成 . . 199

Web サポート	199
単一テーブルの EGL Web アプリケーションの作成	199
「EGL データ・パーツおよびページ」ウィザード	199
単一テーブルの EGL Web アプリケーションの作成	201
「EGL データ・パーツおよびページ」ウィザードでの Web ページの定義	203
EGL ページ・ハンドラーパーツの作成	204
EGL の Page Designer サポート	205
ページ・ハンドラー	207
JavaServer Faces のコントロールと EGL	211
EGL フィールドの作成と Faces JSP との関連付け	212
EGL レコードと Faces JSP との関連付け	214
EGL ページ・ハンドラーへの JavaServer Faces コマンド・コンポーネントのバインディング	215
ページ・ハンドラー・コード用の「クイック編集」ビューの使用	216
EGL ページ・ハンドラーへの JavaServer Faces 入出力コンポーネントのバインディング	217
EGL ページ・ハンドラーへの JavaServer Faces チェック・ボックス・コンポーネントのバインディング	218
EGL ページ・ハンドラーへの JavaServer Faces 単一選択コンポーネントのバインディング	219
EGL ページ・ハンドラーへの JavaServer Faces 複数選択コンポーネントのバインディング	221

EGL レポートの作成 225

EGL レポートの概要	225
EGL レポート作成プロセスの概要	226
データ・ソース	228
ライブラリー内のデータ・レコード	229
EGL レポート・ハンドラー	230
事前定義されたレポート・ハンドラー関数	231
追加の EGL レポート・ハンドラー関数	232
XML 設計文書内のデータ・タイプ	233
EGL レポート・ドライバー関数用のサンプル・コード	234
パッケージへの設計文書の追加	236
レポート・テンプレートの使用	237
EGL レポート・ハンドラーの作成	237
EGL レポート・ハンドラーの手操作による作成	238
レポートをドライブするコードの作成	242
レポート用のファイルの生成およびレポートの実行	244
レポートのエクスポート	245

ファイルとデータベースによる作業 . . . 247

SQL サポート	247
EGL ステートメントと SQL	247
結果セットの処理	252
SQL レコードおよびその値	254
宣言時のデータベース・アクセス	258
動的 SQL	259

SQL 例	260
デフォルト・データベース	270
Informix および EGL	271
SQL 特定のタスク	272
SQL テーブル・データの検索	272
SQL レコード・パーツからのデータ項目パーツ の作成 (概説)	273
リレーショナル・データベース・テーブルからの EGL データ・パーツの作成	274
SQL レコードの SQL SELECT ステートメント の表示	278
SQL レコードの SQL SELECT ステートメント の検証	279
EGL prepare ステートメントの構成	279
暗黙的なステートメントからの明示的な SQL ス テートメントの 構成	280
明示的な SQL ステートメントのリセット	282
SQL 関連 EGL ステートメントからの SQL ス テートメントの除去	282
暗黙的 SQL ステートメントを表示するための参 照の解決	283
標準 JDBC 接続の作成方法について	283
VSAM サポート	285
アクセスの前提条件	285
システム名	285
MQSeries のサポート	285
接続	286
トランザクションにメッセージを組み込む	287
カスタマイズ	288
MQSeries 関連の EGL キーワード	289
直接 MQSeries 呼び出し	292
EGL コードの保守 297	
EGL ソース・コード行のコメント化	297
パーツの検索	297
パーツ参照の表示	298
.egl ファイル内のパーツのオープン	299
プロジェクト・エクスプローラーでの EGL ソー ス・ファイルの 位置決め	300
プロジェクト・エクスプローラーでの EGL ファイ ルの削除	300
EGL コードのデバッグ 303	
EGL デバッガー	303
デバッガー・コマンド	304
ビルド記述子の使用	306
SQL データベース・アクセス	307
call ステートメント	307
デバッグ時に使用されるシステム・タイプ	308
EGL デバッガー・ポート	308
推奨	308
J2EE 以外のアプリケーションのデバッグ	310
EGL デバッガーでの 非 J2EE アプリケーシ ョンの開始	310
EGL デバッガーでの起動構成の作成	310
EGL リスナー起動構成の作成	311

J2EE アプリケーションのデバッグ	312
EGL Web デバッグのためのサーバーの準備	312
EGL Web デバッグのためのサーバーの開始	313
EGL Web デバッグ・セッションの開始	313
EGL デバッガーでのブレークポイントの使用	314
EGL デバッガーでのアプリケーションのステップ スルー	315
EGL デバッガーでの変数の表示	316
EGL ビルド・パーツによる作業 319	
ビルド・ファイルの作成	319
汎用ビルド・オプションの設定	319
外部ファイル、プリンター、およびキューの関連 付けの設定	331
呼び出しおよび転送オプションの設定	338
他の EGL ビルド・ファイルの参照の設定	347
EGL ビルド・パスの編集	348
EGL 出力の生成、準備、および実行 351	
生成	351
プロジェクトへの Java コードの生成	351
ビルド	354
EGL 出力のビルド	355
ビルド計画	356
Java プログラム、ページ・ハンドラー、および ライブラリー	357
結果ファイル	357
ワークベンチでの生成方法	358
ワークベンチでの生成	359
ワークベンチのバッチ・インターフェースからの生 成	360
ワークベンチ・バッチ・インターフェースからの 生成	361
EGL ソフトウェア開発キット (SDK) からの生成	362
EGL ソフトウェア開発キット (SDK) からの生 成	362
生成後のビルド計画の呼び出し	363
Java の生成、その他トピック	364
ディレクトリーへ生成される Java コードの処理	364
EJB プロジェクトのデプロイメント・コードの 生成	368
変数 EGL_GENERATORS_PLUGINDIR の設定	368
ローカル・マシンでの EGL 生成 Java コードの実 行	369
ローカル・マシンでの基本またはテキスト・ユー ザー・インターフェース Java アプリケーシ ョンの開始	369
ローカル・マシンでの Web アプリケーシ ョンの開始	370
ビルド・スクリプト	372
Java ビルド・スクリプト	372
ビルド・サーバー	373
AIX、Linux、または Windows 2000/NT/XP 上 でのビルド・サーバーの開始	373
EGL 生成 Java 出力のデプロイ 377	

Java ランタイム・プロパティー	377
J2EE 環境では	377
非 J2EE Java 環境では	377
ビルド記述子およびプログラム・プロパティー	379
追加情報について	379
EGL 生成コード用の非 J2EE ランタイム環境の設定	380
プログラム・プロパティー・ファイル	380
J2EE の外側での Java アプリケーションのデプロイ	381
Java 用 EGL ランタイム・コードのインストール	381
ターゲット・マシンの CLASSPATH への JAR ファイルの組み込み	382
EGL ランタイムの UNIX curses ライブラリーの設定	383
呼び出し先となる非 J2EE アプリケーションの TCP/IP リスナーのセットアップ	383
EGL 生成コード用の J2EE ランタイム環境のセットアップ	384
重複する JAR ファイルの除去	385
デプロイメント記述子値の設定	386
J2EE 環境ファイルの更新	387
デプロイメント記述子の手操作による更新	388
EJB プロジェクトの JNDI 名の設定	389
CICSJ2C 呼び出し用 J2EE サーバーのセットアップ	389
J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用の TCP/IP リスナーのセットアップ	390
J2EE JDBC 接続のセットアップ	394
リンケージ・プロパティー・ファイルのデプロイ	395
非 EGL JAR ファイルへのアクセスの提供	397

EGL リファレンス 401

EGL での代入の互換性	401
各種の数値型への代入	402
その他の異なる型への代入	402
文字型に関する埋め込みと切り捨て	404
タイム・スタンプ間の代入	404
固定構造内の副構造化フィールドへの代入、または副構造化フィールドからの代入	405
固定レコードの代入	406
代入	407
関連エレメント	407
commit	408
conversionTable	408
fileType	408
fileName	408
formFeedOnClose	408
replace	409
system	409
systemName	409
text	410
asynchLink エレメント	410

リモート呼び出し用 csouidpwd.properties ファイル	411
asynchLink エレメントの package	412
asynchLink エレメントの recordName	412
EGL ソース形式の基本レコード・パーツ	413
ビルド・パーツ	414
EGL ビルド・ファイル形式	414
ビルド記述子オプション	415
ビルド・スクリプト	442
EGL ビルド・スクリプトで必要なオプション	442
callLink エレメント	442
callLink type が localCall の場合 (デフォルト)	442
callLink type が remoteCall の場合	443
callLink type が ejbCall の場合	443
callLink エレメントの alias	444
callLink エレメントの conversionTable	445
callLink エレメントの ctgKeyStore	446
callLink エレメントの ctgKeyStorePassword	446
callLink エレメントの ctgLocation	446
callLink エレメントの ctgPort	447
callLink エレメントの JavaWrapper	447
callLink エレメントの linkType	448
callLink エレメントの library	448
callLink エレメントの location	449
callLink エレメントの luwControl	450
callLink エレメントの package	451
callLink エレメントの parmForm	452
callLink エレメントの pgmName	453
callLink エレメントの providerURL	453
callLink エレメントの refreshScreen	454
callLink エレメントの remoteBind	455
callLink エレメントの remoteComType	456
callLink エレメントの remotePgmType	458
callLink エレメントの serverID	459
callLink エレメントの type	460
C 関数と EGL との併用	461
C の BIGINT 関数	465
C データ型と EGL プリミティブ型	466
C の DATE 関数	467
C の DATETIME および INTERVAL 関数	467
C の DECIMAL 関数	468
EGL プログラムからの C 関数の呼び出し	470
C の Stack 関数	471
C の Return 関数	474
コメント	476
VisualAge Generator との互換性	477
ConsoleUI	478
ConsoleField プロパティーとフィールド	478
EGL consoleUI の ConsoleForm プロパティー	492
EGL consoleUI の Menu フィールド	494
EGL consoleUI の MenuItem フィールド	495
EGL consoleUI の PresentationAttributes フィールド	497
EGL consoleUI の Prompt フィールド	499
EGL consoleUI の Window フィールド	500
containerContextDependent	505

データベースの権限とテーブル名	506	format	581
データ変換	507	noRecordFound	582
起動側が Java コードの場合のデータ変換	507	unique	582
変換アルゴリズム	509	isa 演算子	583
双方向言語テキスト	510	Java ランタイム・プロパティ (詳細)	584
データの初期化	511	Java ラッパー・クラス	595
EGL ソース形式の DataItem パーツ	512	ラッパー・クラスの使用法の概要	596
EGL ソース形式の DataTable パーツ	513	プログラム・ラッパー・クラス	598
EGL ビルド・パスおよび eglpath	517	パラメーター・ラッパー・クラスのセット	599
EGLCMD	518	副構造体項目付き配列ラッパー・クラスのセット	600
構文	518	動的配列ラッパー・クラス	601
例	520	Java ラッパー・クラスの命名規則	603
EGL コマンド・ファイル	521	データ型の相互参照	603
コマンド・ファイルの例	522	EGL での JDBC ドライバーの要件	604
EGL エディター	523	キーワード	605
EGL でのコンテンツ・アシスト	524	add	605
EGL での列挙型	524	call	608
EGL 予約語	527	case	611
SQL ステートメント以外の予約語	527	close	613
EGLSDK	529	continue	615
構文	529	converse	616
例	530	delete	617
eglmaster.properties ファイルの形式	531	display	619
EGL ソース形式	532	execute	619
EGL システム例外	533	exit	624
EGL システム制限	535	for	626
式	536	forEach	627
日時式	537	forward	629
論理式	538	freeSQL	631
数式	546	get	631
テキスト式	547	get absolute	638
マスター・ビルド記述子 plugin.xml ファイルの形		get current	640
式	548	get first	641
EGL ソース形式の FormGroup パーツ	550	get last	643
画面の浮動域のプロパティ	552	get next	644
印刷浮動域のプロパティ	552	get previous	650
EGL ソース形式の書式パーツ	553	get relative	654
テキスト書式のプロパティ	554	goTo	656
印刷書式のプロパティ	555	if, else	656
書式フィールド	556	move	657
テキスト書式フィールドのプロパティ	557	open	664
関数呼び出し	560	openUI	669
関数変数	562	prepare	679
関数パラメーター	564	print	681
inOut とそれに関連する修飾子の影響	568	replace	681
EGL ソース形式の関数パーツ	570	return	684
生成される出力	572	set	685
生成される出力 (参照)	573	show	696
「Generation Results (生成結果)」ビュー	575	transfer	697
in 演算子	575	try	698
1 次元配列の場合の例	576	while	699
多次元配列の場合の例	577	ライブラリー (生成された出力)	700
EGL ソース形式の索引付きレコード・パーツ	578	EGL ソース形式のライブラリー・パーツ	700
入出力エラー値	579	like 演算子	707
duplicate	580	リンケージ・プロパティ・ファイル (詳細)	708
endOfFile	581		

実行時のリンケージ・プロパティ・ファイルの 識別	709	modified	761
リンケージ・プロパティ・ファイルの形式	709	needsSOSI	761
matches 演算子	712	newWindow	762
EGL Java ランタイム用のメッセージのカスタマイ ズ	713	numElementsItem	763
EGL ソース形式の MQ レコード・パーツ	714	numericSeparator	763
MQ レコードのプロパティ	717	outline	763
キュー名	717	pattern	764
トランザクションにメッセージを組み込む	717	persistent	764
入力キューを排他使用で開く	717	protect	765
MQ レコード用のオプション・レコード	718	selectFromListItem	766
別名の割り当て	720	selectType	767
JSP ファイル内の EGL ID の変更と Java Bean の生成	720	sign	767
Java 名の別名の割り当て方法	721	sqlDataCode	768
Java ラッパーの別名の割り当て方法	722	sqlVariableLen	769
命名規則	725	timeFormat	770
演算子と優先順位	726	timeStampFormat	771
Java プログラム生成の出力	727	typeChkMsgKey	771
Java ラッパー生成の出力	729	upperCase	772
例	730	validationOrder	772
EGL ソース形式のページ・ハンドラー・パーツ	732	validatorDataTable	773
ページ・ハンドラー・パーツ・プロパティ	735	validatorDataTableMsgKey	774
ページ・ハンドラー・フィールドのプロパティ	738	validatorFunction	774
pfKeyEquate	739	validatorFunctionMsgKey	775
プリミティブ・フィールド・レベル・プロパティ	739	validValues	776
action	743	validValuesMsgKey	777
align	744	value	778
byPassValidation	745	zeroFormat	778
color	745	パラメーター以外のプログラム・データ	779
column	746	プログラム・パラメーター	781
currency	747	EGL ソース形式のプログラム・パーツ	783
currencySymbol	748	EGL ソース形式の基本プログラム	784
dateFormat	748	EGL ソース形式のテキスト UI プログラム	786
displayName	751	プログラム・パーツ・プロパティ	789
displayUse	751	入力書式	792
fieldLen	752	入力レコード	792
fill	753	レコードとファイル・タイプの相互参照	793
fillCharacter	753	可変長レコードをサポートするプロパティ	793
help	753	lengthItem プロパティを持つ可変長レコード	793
highlight	754	numElementsItem プロパティを持つ可変長レ コード	794
inputRequired	754	lengthItem プロパティと numElementsItem プ ロパティの両方を持つ可変長レコード	795
inputRequiredMsgKey	754	呼び出しまたは転送で渡される可変長レコード	795
intensity	755	EGL での参照の互換性	795
isBoolean	755	EGL ソース形式の相対レコード・パーツ	796
isDecimalDigit	756	実行単位	798
isHexDigit	756	resultSetID	799
isNullable	756	EGL ソース形式のシリアル・レコード・パーツ	799
isReadOnly	757	SQL データ・コードおよび EGL ホスト変数	801
lineWrap	758	変数と固定長の列	801
lowerCase	759	SQL データ型と EGL プリミティブ型との互換 性	801
masked	759	VARCHAR、VARGRAPHIC、および関連する LONG データ型	803
maxLen	759	DATE、TIME、および TIMESTAMP	803
minimumInput	760	SQL レコードの内部レイアウト	803
minimumInputMsgKey	760		

EGL ソース形式の SQL レコード・パーツ	804
EGL ソース形式の構造体フィールド	808
サブストリング	810
EGL 関数の構文図	811
EGL ステートメントおよびコマンドの構文図	812
システム・ライブラリー	813
EGL ライブラリー ConsoleLib	813
EGL ライブラリー ConverseLib	846
EGL ライブラリー DateTimeLib	849
EGL ライブラリー J2EELib	860
EGL ライブラリー JavaLib	864
EGL ライブラリー LobLib	891
EGL ライブラリー MathLib	899
recordName.resourceAssociation	920
EGL ライブラリー ReportLib	922
EGL ライブラリー StrLib	928
EGL ライブラリー SysLib	949
EGL ライブラリー VGLib	980
EGL ライブラリー外部のシステム変数	985
ConverseVar	986
SysVar	991
VGVar	1006
transferToTransaction エレメント	1019
transfer 関連リンク要素の別名	1019
transferToTransaction エレメントの externallyDefined	1020
使用宣言	1020
背景	1020
プログラムまたはライブラリー・パーツ内	1021
formGroup パーツ内	1024
ページ・ハンドラーパーツ内	1024

EGL Java ランタイム・エラー・コー

ド 1027

EGL Java ランタイム・エラー・コード CSO7000E	1028
EGL Java ランタイム・エラー・コード CSO7015E	1029
EGL Java ランタイム・エラー・コード CSO7016E	1029
EGL Java ランタイム・エラー・コード CSO7020E	1030
EGL Java ランタイム・エラー・コード CSO7021E	1030
EGL Java ランタイム・エラー・コード CSO7022E	1030
EGL Java ランタイム・エラー・コード CSO7023E	1030
EGL Java ランタイム・エラー・コード CSO7024E	1031
EGL Java ランタイム・エラー・コード CSO7026E	1031
EGL Java ランタイム・エラー・コード CSO7045E	1031
EGL Java ランタイム・エラー・コード CSO7050E	1032
EGL Java ランタイム・エラー・コード CSO7060E	1032
EGL Java ランタイム・エラー・コード CSO7080E	1032
EGL Java ランタイム・エラー・コード CSO7160E	1032
EGL Java ランタイム・エラー・コード CSO7161E	1033
EGL Java ランタイム・エラー・コード CSO7162E	1033
EGL Java ランタイム・エラー・コード CSO7163E	1033
EGL Java ランタイム・エラー・コード CSO7164E	1034
EGL Java ランタイム・エラー・コード CSO7165E	1034
EGL Java ランタイム・エラー・コード CSO7166E	1034
EGL Java ランタイム・エラー・コード CSO7360E	1034
EGL Java ランタイム・エラー・コード CSO7361E	1035

EGL Java ランタイム・エラー・コード CSO7488E	1035
EGL Java ランタイム・エラー・コード CSO7489E	1036
EGL Java ランタイム・エラー・コード CSO7610E	1036
EGL Java ランタイム・エラー・コード CSO7620E	1036
EGL Java ランタイム・エラー・コード CSO7630E	1037
EGL Java ランタイム・エラー・コード CSO7640E	1037
EGL Java ランタイム・エラー・コード CSO7650E	1037
EGL Java ランタイム・エラー・コード CSO7651E	1038
EGL Java ランタイム・エラー・コード CSO7652E	1038
EGL Java ランタイム・エラー・コード CSO7653E	1039
EGL Java ランタイム・エラー・コード CSO7654E	1039
EGL Java ランタイム・エラー・コード CSO7655E	1040
EGL Java ランタイム・エラー・コード CSO7656E	1041
EGL Java ランタイム・エラー・コード CSO7657E	1041
EGL Java ランタイム・エラー・コード CSO7658E	1042
EGL Java ランタイム・エラー・コード CSO7659E	1042
EGL Java ランタイム・エラー・コード CSO7669E	1043
EGL Java ランタイム・エラー・コード CSO7670E	1043
EGL Java ランタイム・エラー・コード CSO7671E	1043
EGL Java ランタイム・エラー・コード CSO7816E	1043
EGL Java ランタイム・エラー・コード CSO7819E	1044
EGL Java ランタイム・エラー・コード CSO7831E	1044
EGL Java ランタイム・エラー・コード CSO7836E	1044
EGL Java ランタイム・エラー・コード CSO7840E	1045
EGL Java ランタイム・エラー・コード CSO7885E	1045
EGL Java ランタイム・エラー・コード CSO7886E	1046
EGL Java ランタイム・エラー・コード CSO7955E	1046
EGL Java ランタイム・エラー・コード CSO7957E	1046
EGL Java ランタイム・エラー・コード CSO7958E	1047
EGL Java ランタイム・エラー・コード CSO7966E	1047
EGL Java ランタイム・エラー・コード CSO7968E	1047
EGL Java ランタイム・エラー・コード CSO7970E	1048
EGL Java ランタイム・エラー・コード CSO7975E	1048
EGL Java ランタイム・エラー・コード CSO7976E	1048
EGL Java ランタイム・エラー・コード CSO7977E	1049
EGL Java ランタイム・エラー・コード CSO7978E	1049
EGL Java ランタイム・エラー・コード CSO7979E	1049
EGL Java ランタイム・エラー・コード CSO8000E	1049
EGL Java ランタイム・エラー・コード CSO8001E	1050
EGL Java ランタイム・エラー・コード CSO8002E	1050
EGL Java ランタイム・エラー・コード CSO8003E	1050
EGL Java ランタイム・エラー・コード CSO8004E	1050
EGL Java ランタイム・エラー・コード CSO8005E	1051
EGL Java ランタイム・エラー・コード CSO8100E	1051
EGL Java ランタイム・エラー・コード CSO8101E	1051
EGL Java ランタイム・エラー・コード CSO8102E	1052
EGL Java ランタイム・エラー・コード CSO8103E	1052
EGL Java ランタイム・エラー・コード CSO8104E	1052
EGL Java ランタイム・エラー・コード CSO8105E	1053
EGL Java ランタイム・エラー・コード CSO8106E	1053
EGL Java ランタイム・エラー・コード CSO8107E	1054
EGL Java ランタイム・エラー・コード CSO8108E	1054
EGL Java ランタイム・エラー・コード CSO8109E	1054
EGL Java ランタイム・エラー・コード CSO8110E	1054
EGL Java ランタイム・エラー・コード CSO8180E	1055
EGL Java ランタイム・エラー・コード CSO8181E	1055

概説

EGL の紹介

エンタープライズ開発言語 (EGL) は、完全な機能を持つアプリケーションを素早く作成するための開発環境およびプログラミング言語で、ユーザーはソフトウェア・テクノロジーよりもコードが表すビジネス上の問題に集中することができます。類似した I/O ステートメントを使用して各種の外部データ・ストア (例えば、これらデータ・ストアがファイル、¥、またはメッセージ・キューの場合など) にアクセスすることができます。Java™ および J2EE に関する詳細はユーザーに対して隠蔽され、Web テクノロジーに関する経験が少ないユーザーでも、企業データをブラウザーに送信できます。

EGL プログラムをコーディングしたら、これを生成して Java ソースを作成します。これで、EGL により、実行可能オブジェクトを作成するための出力が用意されます。EGL は、以下のサービスを提供することもできます。

- 開発プラットフォームの外部のデプロイメント・プラットフォームにソースを配置する
- デプロイメント・プラットフォームでソースを準備する
- 結果をチェックできるよう、デプロイメント・プラットフォームから開発プラットフォームに状況情報を送信する

EGL では、実行可能オブジェクトの最終的なデプロイメントを行うための出力も作成します。

あるターゲット・プラットフォーム用に作成された EGL プログラムは、別のプラットフォーム用に簡単に変換できます。この利点として、現在のプラットフォームの要件に応じてコーディングすることができ、将来的なマイグレーションに関する多くの詳細が処理されることにあります。EGL では、同じソースから、1 つのアプリケーション・システムの複数のパーツを生成することもできます。

関連する概念

9 ページの『開発過程』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

572 ページの『生成される出力』

19 ページの『パーツ』

10 ページの『ランタイム構成』

関連するタスク

132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

523 ページの『EGL エディター』

532 ページの『EGL ソース形式』

EGL 6.0.0.1 の新機能

バージョン 6.0.0.1 での変更点は次のとおりです。

- EGL 書式エディターにより、テキスト書式と印刷書式を作成するためのグラフィカル・ユーザー・インターフェースが提供されます。
- ターゲット環境は HP-UX と Solaris です。EGL では、それらのプラットフォームの 32 ビットと 64 ビット・サポートを提供しており、AIX の 64 ビット・サポートも追加されました。
- EGL デバッガーの変更点は次のとおりです。
 - ユーザーによる consoleUI ベースのアプリケーションのデバッグを許可します
 - デバッグ・セッション中、文字データと数値データを表現する EBCDIC コード・ページの使用を許可します
- 言語がさらに柔軟になりました。
 - システム変数 **SysVar.sqlCode** と **SysVar.sqlState** を変更できます
 - 関数を含まない数値表現であれば、配列添え字とサブストリング・インデックスに組み込むことができます。
 - 値を戻すすべての関数は、その戻り値のタイプが式内で有効であれば、数値、テキスト、または論理式内部から呼び出すことができます
 - 値を戻すすべての関数は、その戻り値およびパラメーター・タイプに割り当て互換性があれば、修飾子 **in** を持つ関数仮パラメーターへの引数として使用できます
 - すべての EGL システム変数は、その引数とパラメーター・タイプに割り当て互換性があれば、修飾子 **in** を持つ関数仮パラメーターへの引数として受け渡すことができます
 - すべての変更可能な EGL システム変数は、修飾子 **out** (その引数とパラメーター・タイプに割り当て互換性がある場合) または修飾子 **inOut** (その引数とパラメーター・タイプに参照互換性がある場合) を持つ関数仮パラメーターへの引数として受け渡すことができます
- ドキュメンテーションで、すべての EGL システム関数のすべてのパラメーターに関するアクセス修飾子 (**in**、**out**、または **inOut**) が示され、参照互換性と割り当て互換性が説明されるようになりました
- 次のような新しいシステム関数が使用できます。
 - **MathLib.stringAsDecimal** は、文字値 ("98.6" など) を受け入れ、DECIMAL タイプの同等値を戻します。
 - **MathLib.stringAsFloat** は、文字値 ("98.6" など) を受け入れ、FLOAT タイプの同等値を戻します。
 - **MathLib.stringAsInt** は、文字値 ("98" など) を受け入れ、BIGINT タイプの同等値を戻します。
 - **SysLib.conditionAsInt** は、論理式 (*myVar* == 6 など) を受け入れ、式が真であれば 1 を、式が偽であれば 0 を戻します。
 - **SysLib.startLog** は、エラー・ログを開きます。テキストは、プログラムが **SysLib.errorLog** を呼び出すたびにそのログに書き込まれます。

- **SysLib.errorLog** は、テキストをシステム関数 **SysLib.startLog** が開始したエラー・ログにコピーします
- 新規関数サポート consoleUI --
 - **ConsoleLib.currentArrayCount** は、現在のアクティブ・フォームに関連付けられた動的配列のエレメント数を戻します
 - **ConsoleLib.setCurrentArrayCount** は、スクリーン内の arrayDictionary にバウンドされた動的配列に存在する行数を指定します
 - **ConsoleLib.hideAllMenuItems** は、現在表示されているメニューのメニュー項目をすべて隠します
 - **ConsoleLib.showAllMenuItems** は、現在表示されているメニューのメニュー項目をすべて表示します
- この製品には、Informix 4GL 変換ツールが同梱されています
- VAGen マイグレーション・ツールに対して、より効率的なマイグレーションを実現させるため変更が行われました

関連する概念

13 ページの『EGL に関する追加情報のソース』

EGL 6.0 iFix の新機能

注: EGL は、古いコードを EGL 6.0 iFix で機能するコードに変換するのに役立つサービスを提供します。

- JavaServer Faces に基づいた Web アプリケーションの作成にバージョン 6.0 より前の EGL を使用していた場合は、ワークベンチで次のようにしてください。
 1. 「ヘルプ」 > 「Rational ヘルプ (Rational Help)」をクリックします。
 2. ヘルプ・システムの「検索 (Search)」テキスト・ボックスに、『Web プロジェクト内の JavaServer Faces リソースのマイグレーション (Migrating JavaServer Faces resources in a Web project)』というストリングの少なくとも最初の文字を入力します。
 3. 「実行 (GO)」をクリックします。
 4. 「Web プロジェクト内の JavaServer Faces リソースのマイグレーション (Migrating JavaServer Faces resources in a Web project)」をクリックし、そのトピック内の指示に従います。
- EGL 6.0 から、またはそれより前のバージョンからコードをマイグレーションする方法のそれ以外の詳細については、『EGL 6.0 iFix への EGL コードのマイグレーション (Migrating EGL code to the EGL 6.0 iFix)』を参照してください。
- Informix® 4GL または VisualAge® Generator からコードをマイグレーションする場合は、『EGL に関する追加情報のソース』を参照してください。

バージョン 6.0 の iFix は、次のように、EGL 言語を大幅にアップグレードしたものです。

- EGL レポート・ハンドラーが導入されました。これにはカスタマイズされた関数が入っており、それらの関数は、JasperReport の設計ファイルの実行中にさまざ

まな時点で呼び出されます。それぞれの関数から戻されたデータは、出力レポートに組み込まれます。出力レポートは、PDF、XML、テキスト、または HTML フォーマットで提供されます。このテクノロジーは、Informix 4GL にあったレポート作成機能を改良したものです。

- EGL コンソール UI が導入されました。これは、文字ベースのインターフェースを作成するためのテクノロジーで、このインターフェースを使用すると、ユーザーと EGL 生成 Java プログラムの間で、即時の、キー・ストローク駆動型の対話を行うことができます。このテクノロジーは、Informix 4GL にあった動的ユーザー・インターフェースを改良したものです。
- コード開発に新たな柔軟性を提供します。
 - 新しいタイプの変数を宣言できます。
 - ビジネス・データを含んでいるのではなく、そのようなデータを指し示す参照変数。
 - 大量のデータ、特にバイナリー・ラージ・オブジェクト (BLOB) または文字ラージ・オブジェクト (CLOB) を格納または参照する変数。
 - 実行時に長さが変化するユニコード・ストリングを参照するストリング変数。
 - あらゆるプリミティブ型のビジネス・データを格納できる ANY 型変数。
 - 式の中に関数呼び出しを組み込むことができます。
 - レコードまたはレコード内のフィールドのサイズやその他の特性が開発時に分からなくても、レコードを参照できます。各フィールドは、それ自体がレコードを参照できます。
 - 動的配列のサポートが拡張され、動的配列に複数の次元を持たせることができるようになりました。
 - 次のような 2 つの新しい種類のデータ・コレクションが導入されました。
 - 辞書。これは、一連の「キーおよび値」エントリーから構成されます。実行時にエントリーの追加、削除、および取り出しを行うことができ、そのエントリー内の値は、どのようなタイプであってもかまいません。
 - arrayDictionary。これは一連の 1 次元配列から構成され、それぞれの配列は、どのようなタイプであってもかまいません。arrayDictionary の内容にアクセスするには、すべての配列から同じ番号が付いているエレメントを取り出します。
 - 次のように、さまざまな目的でシステム関数の数が増えました。
 - 日時処理、ランタイム・メッセージ処理、ユーザー定義 Java ランタイム・プロパティーの取り出しなどの機能を向上させるため。
 - レポート、コンソール UI、BLOB、および CLOB に関連した新機能をサポートするため。
 - 例外の処理、データの初期化、および DLL アクセスのためのサポートが改良されました。
- EGL レポート・ハンドラーを作成するための新しいウィザードを提供します。
- データ・パーツおよびページ・ウィザードと一緒に使用する Web ページ・テンプレートをカスタマイズできます。これにより、単一のリレーショナル・データベースにアクセスするための Web アプリケーションが迅速に提供されます。

- NULL の処理とデータベースのコミットとの関連で、Informix 4GL の実行時の振る舞いを反映したコードを作成できます。

関連する概念

108 ページの『EGL から EGL へのマイグレーション』

13 ページの『EGL に関する追加情報のソース』

『EGL バージョン 6.0 の新機能』

EGL バージョン 6.0 の新機能

注: JavaServer Faces に基づいた Web アプリケーションの作成に旧バージョンの EGL を使用していた場合は、ワークベンチで次のようにしてください。

1. 「ヘルプ」 > 「Rational ヘルプ (Rational Help)」 をクリックします。
2. ヘルプ・システムの「検索 (Search)」テキスト・ボックスに、『Web プロジェクト内の JavaServer Faces リソースのマイグレーション (Migrating JavaServer Faces resources in a Web project)』というストリングの少なくとも最初の文字を入力します。
3. 「実行 (GO)」をクリックします。
4. 「Web プロジェクト内の JavaServer Faces リソースのマイグレーション (Migrating JavaServer Faces resources in a Web project)」をクリックし、そのトピック内の指示に従います。

バージョン 6.0 では、次のように EGL 言語の能力が向上しています。

- リレーショナル・データベースの処理が改良されました。
 - 新規ウィザードにより、次の作業を迅速に行うことができます。
 - リレーショナル・データベース表からデータ・パーツを直接作成する
 - そのような表から表の行の作成、読み取り、更新、および削除を行う Web アプリケーションを作成する
 - 次のような新しいシステム関数が使用できます。
 - **sysLib.loadTable** はファイルから情報をロードし、リレーショナル・データベース表に挿入します。
 - **sysLib.unloadTable** はリレーショナル・データベース表から情報をアンロードし、ファイルに挿入します。
 - Java コードを生成する場合、カーソル内の SQL データベース行にアクセスする方法として、(常にそうであったように) 次の行ヘナビゲートするか、最初の行、最後の行、前の行、現在の行ヘナビゲートするか、カーソル内の絶対位置または相対位置を指定することができます。
 - **forEach** ステートメントを使用して、SQL 結果セットの行を容易にループさせることができます。
 - **freeSQL** ステートメントは、動的に準備された SQL ステートメントに関連したリソースを解放し、その SQL ステートメントに関連したすべてのオープン・カーソルをクローズします。
- ストリングの処理が改良されました。
 - 次の例のように、テキスト式の中にサブストリングを指定できます。

```
myItem01 = "1234567890";
```

```
// myItem02 = "567"  
myItem02 = myItem01[5:7];
```

- テキスト・リテラルの中にバックスペース、用紙送り、またはタブを指定できます。
- スtringの比較に、次の 2 つのパターン・タイプを使用できます。

- **LIKE** キーワードを組み込んだ **SQL** タイプのパターン。以下に例を示します。

```
// variable myVar01 is the string expression  
// whose contents will be compared to a like criterion  
myVar01 = "abcdef";
```

```
// the next logical expression evaluates to "true"  
if (myVar01 like "a_c%")  
;  
end
```

- 正規表現パターン。以下に例を示します。

```
// variable myVar01 is the string expression  
// whose contents will be compared to a match criterion  
myVar01 = "abcdef";
```

```
// the next logical expression evaluates to "true"
```

```
if (myVar01 matches "a?c*")  
;  
end
```

- 以下のテキスト・フォーマット設定システム関数を使用できます。

strLib.characterAsInt

文字Stringを整数Stringに変換します。

strLib.clip

戻された文字Stringの末尾から、ブランク・スペースと **NULL** を削除します。

strLib.formatNumber

数値をフォーマット設定されたStringとして戻します。

strLib.integerAsChar

整数Stringを文字Stringに変換します。

strLib.lowercase

文字String内のすべての大文字値を小文字値に変換します。

strLib.spaces

指定された長さのStringを戻します。

strLib.upperCase

文字String内のすべての小文字値を大文字値に変換します。

- 新しい型の変数と構造体項目を宣言できます。

新しい数値型は以下のとおりです。

FLOAT

有効数字が 16 桁の倍精度浮動小数点数を格納する 8 バイト領域

MONEY

有効数字が最大 32 桁までの固定小数点 10 進数として格納される通貨金額

SMALLFLOAT

有効数字が 8 桁の単精度浮動小数点数を格納する 4 バイト領域

新しい日時型は以下のとおりです。

DATE

8 つの単一バイトの数字で表される特定のカレンダー日付

INTERVAL

1 から 21 個の単一バイトの数字で表され、時分秒を表す「hhmmss」などのマスクへ関連付けられている時間幅

TIME

6 つの単一バイトの数字で表される時刻のインスタンス

TIMESTAMP

1 から 20 個の単一バイトの数字で表され、年月日と時を表す「yyyyMMddhh」などのマスクへ関連付けられている時刻のインスタンス。

- 構文に以下のオプションが追加されました。
 - 構造体項目の配列の要素は、次のようにして常に参照できました。しかし、iFix の変更を考慮して、この構文を回避する必要があります。

```
mySuperItem.mySubItem.mySubmostItem[4,3,1]
```

次の構文が強く推奨されています。

```
mySuperItem[4].mySubItem[3].mySubmostItem[1]
```

- パラメーター、use ステートメントエントリ、set ステートメントエントリ、または変数を宣言するときに、次の例のように、ID のコンマ区切りリストを使用できます。

```
myVariable01, myVariable02 myPart;
```

- 数式の中で、指数を指定するのに数値の前に二重のアスタリスク (**) を使用できるようになり、例えば、8 の 3 乗は 8**3 と指定できます。
- 日付、時刻、タイム・スタンプ、または間隔へと解決されるそれぞれの式を指定できるようになり、日付の演算を使用して、2 つの日付の間の分数を計算するなど、さまざまなタスクを実行できます。
- 日付と時刻の処理に、以下の追加機能も使用できます。
 - **DateTimeLib.currentTime** および **DateTimeLib.currentTimeStamp** は、現在時刻を反映するシステム変数です。
 - 新しいフォーマット設定関数を、日付 **StrLib.formatDate**、時刻 (**StrLib.formatTime**)、およびタイム・スタンプ (**sysLib.TimeStamp**) に使用できます。
 - 以下の各関数により、一連の文字を日時型の項目に変換でき、その項目を日時の式に使用できます。
 - **DateTimeLib.dateValue** は、日付を戻します。
 - **DateTimeLib.timeValue** は、時刻を戻します。
 - **DateTimeLib.timeStampValue** は、「yyyyMMdd」などの特定のマスクへ関連付けられているタイム・スタンプを戻します。

- **DateTimeLib.intervalValue** は、「yyyyMMdd」などの特定のマスクへ関連付けられている間隔を戻します。
 - **DateTimeLib.extendDateTimeValue** は、日付、時刻、またはタイム・スタンプを受け入れ、それを「yyyyMMddmmss」などの特定のマスクへ関連付けられている項目へ拡張します。
- 以下の新しい汎用文を使用できます。
 - **for** ステートメントは、ループ内でテストによって真と評価された回数のみ実行される文ブロックを組み込みます。このテストはループの開始時に実行され、カウンターの値が指定された範囲内であるかどうかを示します。
 - **continue** ステートメントは、**for**、**forEach**、**while** のいずれかのステートメント（これらの文自体も **continue** ステートメントを含んでいる）の末尾に制御を渡します。収容文の実行が続行されるか終了するかは、収容文の開始時に実行される論理テストによって決まります。
 - システム・コマンドは、(**sysLib.callCmd** 関数を発行することによって) 同期式に実行するか、(**sysLib.startCmd** 関数を発行することによって) 非同期式に実行できます。
 - 次の 2 つの新しい関数を使用して、ループ内のコマンド行引数にアクセスできます。
 - **sysLib.callCmdLineArgCount** は、引数の数を戻します。
 - **sysLib.callCmdLineArg** は、引数リスト内の指定した位置にある引数を戻します。
 - それぞれの文節が異なる論理式へ関連付けられている **case** ステートメントを指定できるようになりました。この新しい構文を使用した場合、EGL ランタイムは最初に真と評価された式へ関連付けられている文を実行します。


```
case
  when (myVar01 == myVar02)
    conclusion = "okay";
  when (myVar01 == myVar03)
    conclusion = "need to investigate";
  otherwise
    conclusion = "not okay";
end
```
 - 関数パラメーターを入力だけに使用するか、出力だけに使用するか、それとも両方に使用するかを制御できます。また、無制限に「両方に使用」するデフォルト設定を受け入れることにより、選択を回避できます。
 - 次のような場合には、単一の項目や定数より複雑な日時式、テキスト式、または数式を指定できるようになりました。
 - **return** ステートメントによってオペレーティング・システムへ提供される値を指定する場合
 - 関数呼び出しまたはプログラム呼び出しで渡される引数を指定する場合。ただし、受け取り側パラメーターの特性は、生成時に既知であることが必要です。
 - プログラムを終了するときに、複雑な数式を指定できるようになりました。

開発環境も、次のように改良されました。

- 次の 2 つの新規フィーチャーにより、コードの複雑さが増した場合でもパーツに迅速にアクセスできます。

- 「パーツ参照」ビューにより、プログラム、ライブラリー、またはページ・ハンドラーが参照する EGL パーツの階層リストを表示できます。また、そのリストから、任意の参照先パーツにアクセスできます。
- EGL 検索メカニズムにより、ワークスペース内またはプロジェクトのサブセット内にある一連のパーツまたは変数にアクセスするための検索基準を指定できます。
- 最後に、広く使用されている Web パースペクティブに配慮して、EGL Web パースペクティブが除去されました。

関連する概念

108 ページの『EGL から EGL へのマイグレーション』

13 ページの『EGL に関する追加情報のソース』

3 ページの『EGL 6.0 iFix の新機能』

開発過程

EGL の作業には以下のステップがあります。

セットアップ

作業環境をセットアップします。例えば、設定を行ったりプロジェクトを作成したりします。

EGL ファイルを作成して開く

ソース・コードの作成を開始します。

宣言

コードの詳細情報を作成して指定します。

検証

さまざまな場合（ファイルを保存するときなど）に、EGL により宣言が検討され、ある程度まで文法が正しいか、内部の統一が取れているかどうかが表示されます。

デバッグ

組み込みデバッガーとの対話形式で、作成したコードが要件を満たしていることを確認できます。

生成

EGL により宣言が検証され、ソース・コードを含む出力が生成されます。

準備

EGL はソース・コードから実行可能オブジェクトを作成できるように準備します。多くの場合、このステップによりソース・コードが開発プラットフォーム外部のデプロイメント・プラットフォームに配置され、デプロイメント・プラットフォームでソース・コードの準備が行われます。次に、デプロイメント・プラットフォームから開発プラットフォームに結果ファイルが送信されます。

即時送信

場合によっては、Java 出力を右クリックし「実行」 > 「Java アプリケーション」をクリックすることにより、ワークベンチのコードを即時に実行することができます。

デプロイメント

EGL は実行可能オブジェクトのデプロイメントを容易にするため出力を生成します。

関連する概念

303 ページの『EGL デバッガー』

351 ページの『プロジェクトへの Java コードの生成』

1 ページの『EGL の紹介』

関連するタスク

364 ページの『ディレクトリーへ生成される Java コードの処理』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

523 ページの『EGL エディター』

532 ページの『EGL ソース形式』

ランタイム構成

EGL では、複数のサポートされるプラットフォームいずれについても、Java プログラムを生成することができます。このプログラムは、J2EE の外側、または以下の J2EE コンテナのコンテキストのいずれかにデプロイすることができます。

- J2EE アプリケーション・クライアント
- J2EE Web アプリケーション
- EJB コンテナ (この場合、EJB セッション Bean も生成します)

また、EGL では、以下の特性を持つ Web アプリケーションを定義する方法が提供されています。

- Web ブラウザーに図形ページを配信する
- 多数になる可能性があるユーザーのデータを保管および検索する
- JavaServer Face という Java ベースのフレームワークに組み込む

この特殊な Web アプリケーションのサポートについて詳しくは、『ページ・ハンドラー・パーツ』を参照してください。

これで、次の節で説明するように、EGL を使用して、Java ラッパーを作成できます。

Java ラッパーの使用

EGL 生成 Java ラッパーは一連のクラスであり、これを使用すると、非 EGL 生成 Java コード (例えば Struts または JSF ベースの J2EE Web アプリケーションのアクション・クラス、または非 J2EE Java プログラム) から EGL 生成プログラムを呼び出すことができます。Java と EGL の統合は以下のように行われます。

1. EGL で生成された特定のプログラムに固有の Java ラッパー・クラスを生成する
2. これらのラッパー・クラスを、EGL 以外で生成された Java コードに取り込む
3. EGL 以外で生成された Java コードからラッパー・クラス・メソッドを呼び出すことによって、実際の呼び出しを行い、これら 2 つの形式間でデータを変換する

- Java で使用されるデータ型の形式
- EGL 生成プログラムとの間でデータをやり取りするには、プリミティブ型の形式が必要です。

有効な呼び出し

EGL 生成コードとの間で使用できる有効な呼び出しは、以下の表のとおりです。

呼び出し側オブジェクト	呼び出されるオブジェクト
J2EE の外部の Java クラス内 EGL 生成 Java ラッパー	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS® COBOL プログラム
J2EE アプリケーション・クライアント内 EGL 生成 Java ラッパー	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS COBOL プログラム
J2EE Web アプリケーション内 EGL 生成 Java ラッパー	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	同一の J2EE Web アプリケーション内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS COBOL プログラム
J2EE の外部の EGL 生成 Java プログラム	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS COBOL プログラム
	非 EGL 生成プログラム (C または C++)
	非生成プログラム (CICS で実行される任意の言語で作成されたプログラム)

呼び出し側オブジェクト	呼び出されるオブジェクト
J2EE アプリケーション・クライアント内の EGL 生成 Java プログラム	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	EGL 生成 CICS COBOL プログラム
	非生成プログラム (CICS で実行される任意 の言語で作成されたプログラム)
	C または C++ で作成された非生成プログラ ム
J2EE Web アプリケーション内 EGL 生成 Java プログラム	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	同一の J2EE Web アプリケーション内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS COBOL プログラム
	C または C++ で作成された非生成プログラ ム
EGL 生成 EJB セッション Bean	EGL 生成 Java プログラム (J2EE 以外)
	J2EE アプリケーション・クライアント内 EGL 生成 Java プログラム
	EGL 生成 EJB セッション Bean
	VisualAge Generator で生成された CICS COBOL プログラム
	C または C++ で作成された非生成プログラ ム

有効な転送

EGL 生成コードとの間の有効な転送は、以下の表のとおりです。

転送側オブジェクト	オブジェクトの受信
J2EE の外部の EGL 生成 Java プログラム	EGL 生成 Java プログラム (J2EE 以外)
J2EE アプリケーション・クライアント内の EGL 生成 Java プログラム	同一の J2EE アプリケーション・クライアン ト内 EGL 生成 Java プログラム
J2EE Web アプリケーション内 EGL 生成 Java プログラム	同一の J2EE Web アプリケーション内 EGL 生成 Java プログラム

関連する概念

572 ページの『生成される出力』

1 ページの『EGL の紹介』

357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』

327 ページの『Java ラッパー』

207 ページの『ページ・ハンドラー』

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

EGL に関する追加情報のソース

最新版の資料は、以下の Web サイトから入手できます。

<http://www.ibm.com/developerworks/rational/library/egldoc.html>

VisualAge Generator で作成されたソース・コードのマイグレーションの詳細については、「*VisualAge Generator から EGL へのマイグレーション・ガイド*」(vagenmig.pdf ファイル) を参照してください。この資料は、上記 Web サイトおよび『*Installing and migrating*』というヘルプ・システム・セクションにあります。

ただし、関連 site を含むプロジェクト内にページ・ハンドラーを生成することをお勧めします。

ただし、Web サイトにアクセスすることをお勧めします。

関連する概念

1 ページの『EGL の紹介』

EGL 言語の概説

EGL プロジェクト、パッケージ、およびファイル

EGL プロジェクトには、ソース・フォルダーがゼロから多数まで含まれています。それぞれのフォルダーには、パッケージがゼロから多数まで含まれています。それぞれのパッケージには、ファイルがゼロから多数まで含まれています。それぞれのファイルには、パーツがゼロから多数まで含まれています。

EGL プロジェクト

EGL プロジェクトは、後述する一連のプロパティによって特徴付けられています。EGL プロジェクトのコンテキストでは、特定のタスク実行時 (例えば、EGL ファイル、または EGL ビルド・ファイルの保管時) に、EGL は自動的に検証を実行し、パーツ参照を解決します。また、ページ・ハンドラー・パーツ (この出力が、Websphere テスト環境の Web アプリケーションのデバッグで使用されます) の操作を行う場合、EGL は、以下の場合にのみ自動的に出力を生成します。

- 次のオプション「ウィンドウ」>「設定」>「ワークベンチ」>「リソース変更時にビルドを自動的に実行」の選択後に、自動ビルド・プロセスを設定した場合
- デフォルトのビルド記述子が設定またはプロパティとして設定されている場合

EGL プロジェクトを構成するには、新規プロジェクトの作成時にプロジェクト・タイプとして **EGL** または **EGL Web** を選択します。プロジェクト作成ステップを実行中に、プロパティを割り当てます。これらのステップの完了後に選択を変更するには、プロジェクト名を右マウス・ボタン・クリックし、表示されたコンテキスト・メニューで、「**プロパティ**」をクリックします。

EGL プロパティには、次のものがあります。

EGL ソース・フォルダー

プロジェクトのパッケージのルートである 1 つ以上のプロジェクト・フォルダー。各フォルダーは、サブディレクトリーの集合です。ソース・フォルダーは、Java ファイルとは区別して EGL ソースを保持したり、Web デプロイメント・ディレクトリー以外の場所に EGL ソース・ファイルを保持する場合に役立ちます。どのような場合でも、EGL ソース・フォルダーを指定することをお勧めします。ただし、ソース・フォルダーが指定されていない場合には、唯一のソース・フォルダーはプロジェクト・ディレクトリーになります。

このプロパティの値は、プロジェクト・ディレクトリーの `.eglp` という名前のファイルに格納され、EGL ファイルの保管に使用するリポジトリー (存在する場合) に保管されます。

それぞれの EGL プロジェクト・ウィザードは、**EGLSource** という名前の 1 つのソース・フォルダーを作成します。

EGL ビルド・パス

現行プロジェクトに存在しない任意のパーツで検索されるプロジェクト・リスト。

このプロパティの値は、プロジェクト・ディレクトリーの `.eglp` という名前のファイルに格納され、EGL ファイルの保管に使用するリポジトリ (存在する場合) に保管されます。

次の `.eglp` ファイルの例では、EGLSource は現行プロジェクト内のソース・フォルダーで、AnotherProject は EGL パス内のプロジェクトです。

```
<?xml version="1.0" encoding="UTF-8"?>
<eglp>
  <eglpentry kind="src" path="EGLSource"/>
  <eglpentry kind="src" path="¥AnotherProject"/>
</eglp>
```

AnotherProject のソース・フォルダーは、その プロジェクトの `.eglp` ファイルによって判別されます。

デフォルトのビルド記述子

出力を迅速に生成できるビルド記述子 (『ワークベンチでの生成』を参照)

パッケージ

パッケージは、関連するソース・パーツの名前付きコレクションです。ビルド・パーツを作成するときに、パッケージは使用されません。

規則に従って、パッケージ名の最初の部分を組織のインターネット・ドメイン名を語順反転したものにすることによって、パッケージ名の一意性を実現します。例えば、IBM® ドメイン名は、`ibm.com` で、EGL パッケージは「`com.ibm`」で始まります。この規則を使用することによって、組織が開発した Web プログラム名は、他の組織が開発したプログラム名と重複しないということが保証され、名前が衝突することなく同一サーバーにインストールできます。

個々のパッケージのフォルダーは、パッケージ名によって識別されます。これは、ID をピリオド (.) で区切ったシーケンスです。例を次に示します。

`com.mycom.mypack`

それぞれの ID は、EGL ソース・フォルダー内のサブフォルダーに対応しています。例えば、`com.mycom.mypack` のディレクトリー構造は `¥com¥mycom¥mypack` で、ソース・ファイルは最下位のフォルダー (この場合には `mypack`) に格納されます。ワークスペースが `c:¥myWorkspace`、プロジェクトが `new.project`、ソース・フォルダーが `EGLSource` の場合には、このパッケージのパスは次のようになります。

`c:¥myWorkspace¥new.project¥EGLSource¥com¥mycom¥mypack`

EGL ファイル内のすべてのパーツは同一のパッケージに属します。ファイルの `package` ステートメントがある場合には、その文はそのパッケージ名を指定します。`package` ステートメントを指定しない場合には、パーツはソース・フォルダーに直接格納されます。これで、デフォルト・パッケージに格納されたことになります。`package` ステートメントは、常に指定することをお勧めします。これは、デフォルト・パッケージ内のファイルは、他のパッケージまたはプロジェクトのパーツによって共有できないためです。

同一の ID を持つ 2 つのパーツは、同じパッケージに定義することはできません。同じパッケージ名を異なるプロジェクトまたは異なるフォルダーで使用しないことを特にお勧めします。

生成された Java 出力のパッケージは、EGL ファイル・パッケージと同じです。

EGL ファイル

すべての EGL ファイルは、以下のいずれかのカテゴリーに含まれます。

ソース・ファイル

EGL ソース・ファイル (拡張子 .egl) には、ロジック、データ、およびユーザー・インターフェース・パーツが含まれ、EGL ソース形式で作成されています。

以下の生成可能パーツは、コンパイル可能単位に変換できます。

- DataTable
- FormGroup
- Handler (レポート・ハンドラーの基礎)
- Library
- ページ・ハンドラー
- Program

EGL ソース・ファイルには、非生成可能パーツをゼロ個以上いくつでも組み込むことができますが、生成可能パーツは 1 つしか組み込むことができません。生成可能パーツ (存在する場合) は、ファイルの最上位にある必要があります。そのファイルと同じ名前になっている必要があります。

ビルド・ファイル

EGL ビルド・ファイル (拡張子 .eglbld) には、ビルド・パーツをいくつでも組み込むことができ、Extensible Markup Language (XML)、EGL ビルド・ファイル形式で作成されています。関連する DTD (以下のディレクトリーにあります) を検討することができます。

```
installationDir%egl%eclipse%plugins%  
com.ibm.etools.egl_version
```

installationDir

製品のインストール・ディレクトリー。例えば、C:\Program Files\IBM\Rational\SPD\6.0 など。これから使用しようとしている製品をインストールする前に Rational® Developer 製品をインストールし、保持していた場合は、以前のインストール時に使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

ファイル名 (egl_wssd_6_0.dtd など) は、文字 *egl* およびアンダースコアで開始します。*wssd* の文字は Rational Web デベロッパーおよび Rational Application Developer を指します。*wsed* の文字は、Rational Application Developer for z/OS® を指し、*wdsc* の文字は Rational Application Developer for iSeries™ を指します。

パーツをファイルに追加した後に、リポジトリを使用して変更の履歴を管理することができます。

推奨

このセクションでは、ユーザーの開発プロジェクトの設定に関する推奨事項について説明します。

ビルド記述子の場合

チーム・プロジェクトでは、ビルド記述子開発者として 1 人を指名する必要があります。開発者の作業は以下のとおりです。

- ソース・コード開発者用のビルド記述子を作成する
- これらのビルド記述子をソース・コード・プロジェクトとは別のプロジェクトに配置する。さらに、リポジトリまたはその他の方法で、その別のプロジェクトを利用可能にします。
- ソース・コード開発者に、プロジェクト内にプロパティーデフォルトのビルド記述子を設定するように依頼する。これによって、このプロパティーは、適切なビルド記述子を参照します。
- ビルド記述子オプションの小さなサブセット (ユーザー ID およびパスワードなど) が、1 人のソース・コード開発者と次のソース・コード開発者で異なる場合には、それぞれのソース・コード開発者に以下を行うよう依頼します。
 - **nextBuildDescriptor** オプションを使用してグループ・ビルド記述子を指す個人用ビルド記述子をコード化する。
 - ソース・コード開発者に、ファイル、フォルダー、またはパッケージ内にプロパティーデフォルトのビルド記述子を設定するように依頼する。これにより、このプロパティーは、個人用ビルド記述子を参照します。開発者は、プロパティーをプロジェクト・レベルでは設定しません。これは、プロジェクト・レベルのプロパティーが、他のプロジェクト情報とともに、リポジトリによって制御されるためです。

追加情報については、『ビルド記述子パーツ』を参照してください。

パッケージの場合

パッケージに関する推奨事項は、次のとおりです。

- 異なるプロジェクトまたは異なるソース・ディレクトリーで、同じパッケージ名は使用しない
- デフォルト・パッケージは使用しない

パーツ割り当て

パーツの場合には、推奨事項の多くは役立つ例であり、困難な要件ではありません。別の方法で行う正当な理由がない場合には、以下のオプションの推奨事項も実行してください。

- 要件 は、関連付けされているページ・ハンドラーと同じプロジェクトに JSP を配置することです。
- 非生成可能パーツ (レコード・パーツなど) が、1 つのプログラム、ライブラリー、またはページ・ハンドラーのみによって使用される場合は、使用する側のパーツと同じファイル内にその非生成可能パーツを配置してください。
- パーツが同じパッケージ内の異なるファイルから参照される場合は、そのパッケージ内の別のファイルにそのパーツを配置します。

- パーツが単一プロジェクト内のパッケージ間で共用される場合は、そのプロジェクト内の別のパッケージにそのパーツを配置します。
- 完全に無関係なアプリケーションのコードは、異なるプロジェクトに配置します。プロジェクトは、ローカル・ディレクトリ構造とリポジトリ間でコード転送を行う単位です。開発者が開発システムにロードする必要があるコードの量を最小化できるようにプロジェクト構造を設計します。
- プロジェクト、パッケージ、およびファイルに、それらに含まれるパーツの使用法を反映した名前を付けます。
- プロセスで開発者のコード所有権が主張されている場合は、所有者が異なるパーツを同じファイルに割り当てないでください。
- パッケージの目的を明確に理解してから、パッケージにパーツを割り当てます。また、パーツ間の関係の緊密度に応じてパーツをグループ化します。

以下の特徴は重要です。

- 同一のパッケージ内でファイルからファイルへパーツを移動するために、他のファイルの `import` ステートメントを変更する必要はありません。
- 1 つのパッケージから他のパッケージへパーツを移動する場合は、移動するパーツを参照するすべてのファイルに、`import` ステートメントを追加するか、または変更する必要がある場合があります。

関連する概念

319 ページの『ビルド記述子パーツ』

359 ページの『ワークベンチでの生成』

24 ページの『パーツの参照』

36 ページの『インポート』

1 ページの『EGL の紹介』

『パーツ』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

532 ページの『EGL ソース形式』

93 ページの『EGL ステートメント』

パーツ

EGL ファイルには、一連のパーツが含まれています。各パーツは、個別の名前付きの単位です。いくつかのパーツ（プログラムなど）は、生成可能パーツであり、これらはそれぞれ、コンパイル可能単位の基礎となります。生成可能パーツは、そのパーツを含んでいる EGL ソース・ファイルと同じ名前を持っている必要があります。

EGL ソース・ファイル（拡張 `.egl`）には、ゼロ個または 1 個の生成可能パーツと、ゼロ個以上多数のその他のパーツを組み込むことができます。

パーツは、次のように分類することもできます。

- ロジック・パーツ は、EGL プロシージャ型言語で作成したランタイム・シーケンスを定義します。
 - 非生成可能パーツである関数 は、ロジックの基本単位です。その他の種類のロジック・パーツすべてに、関数を組み込むことができます。
 - 以下に示すような 2 つのタイプのプログラム を定義でき、これらはインターフェース・タイプによって異なります。それぞれが 1 つの生成可能パーツです。
 - 基本プログラムは、ユーザーとの対話と回避するか、その対話を特定の種類の文字ベースのインターフェースに制限します。その場合、インターフェース・テクノロジーは、次のように機能します。
 - 出力をコマンド・ウィンドウに表示します。また、
 - ユーザーが個々のキー・ストロークごとにプログラムと即時に対話できるようにします。それらのキー・ストロークは、プログラムが処理する個々のイベントを潜在的に定義しています。

この種類のインターフェースの詳細については、『コンソール・ユーザー・インターフェース』を参照してください。

- *textUI* プログラムは、次のような方法でユーザーと対話します。
 - 一連のフィールドをコマンド・ウィンドウに表示します。また、
 - ユーザーが実行依頼キーを押したときにのみ、ユーザーのフィールド入力を受け入れます。

どちらのタイプのプログラムをメインプログラムとして定義してもかまいません。メインプログラムは、以下の方法によって開始されます。

- ユーザーによって開始される
- 呼び出し以外のプログラム転送によって開始される
- オペレーティング・システムのプロセスによって直接開始される

どちらの基本プログラムも、呼び出しによって起動できる呼び出し先プログラム として宣言できます。

メインプログラムおよび呼び出し先プログラムのランタイム・デプロイメントについて詳しくは、『ランタイム構成』を参照してください。

- ページ・ハンドラーは、生成可能パーツの 1 つであり、ユーザーと Web ページの間の対話を制御します。
- JasperReport タイプのハンドラーは、カスタマイズされた関数が入っている生成可能パーツであり、それらの関数は、JasperReport の設計ファイルの実行中にさまざまな時点で呼び出されます。それぞれの関数から戻されたデータは、出力レポートに組み込まれます。出力レポートは、PDF、XML、テキスト、または HTML フォーマットで提供されます。
- ライブラリーも生成可能パーツです。これは、プログラム、ページ・ハンドラー、および他のライブラリーから使用可能にすることができる、共用関数および変数のコレクションです。
- データ・パーツ は、ユーザー・プログラムが利用可能なデータ構造を定義します。

以下の種類のデータ・パーツが変数宣言内でタイプとして使用されます。

- *DataItem* パーツ には、最も基本的な種類のデータに関する情報が入っています。これらのパーツは、システム全体のデータ・ディクショナリー内のエンタリーによく似ており、それぞれのパーツには、データのサイズ、タイプ、フォーマット設定規則、入力妥当性検査規則、および表示の提案に関する詳細が組み込まれています。ある *DataItem* パーツを 1 回定義しておけば、そのパーツを任意の数のプリミティブ変数またはレコード・フィールドの基礎として使用できます。

DataItem パーツは、プリミティブ型から変数を作成するのに便利です。例えば、次のような *myStringPart* の定義について考えてみます。これは、*String* 型の *DataItem* パーツです。

```
DataItem
  MyStringPart String { validValues = ["abc", "xyz"] }
end
```

ある関数を開発する場合は、次のように *MyStringPart* タイプの変数を宣言できます。

```
myString MyStringPart;
```

次の宣言は、上記の宣言と同じ効果があります。

```
myString STRING { validValues = ["abc", "xyz"] };
```

このように、*DataItem* パーツの名前は、特定のプロパティー設定値が与えられたプリミティブ型の別名です。

- レコード・パーツ は、複合データの基礎となります。タイプがレコード・パーツである変数は、フィールドを含んでいます。各フィールドは、以下のものを基礎とすることができます。
 - *STRING* などのプリミティブ型
 - *DataItem* パーツ
 - 固定レコード・パーツ (後述)
 - 別のレコード・パーツ
 - 上記のいずれかの種類の配列

それぞれのフィールドは、辞書または *ArrayDictionary* (後述) にすることも、辞書または *ArrayDictionary* の配列にすることもできます。

レコード・パーツに基づいた変数はレコードと呼ばれ、レコード内のデータの長さは、実行時に変化する場合があります。

レコード・パーツを使用して、汎用処理のための変数を作成したり、リレーショナル・データベースにアクセスしたりすることができます。

- 固定レコード・パーツ は、固定長の複合データの基礎となります。タイプが固定レコード・パーツである変数は、フィールドを含んでおり、各フィールドは、以下のいずれかを型として持つことができます。
 - *CHAR* などのプリミティブ型
 - *dataItem* パーツ

各フィールドは、副構造を持つことができます。例えば、電話番号を指定するフィールドは、次のように定義できます。

```
10 phoneNumber    CHAR(10);
20 areaCode       CHAR(3);
20 localNumber    CHAR(7);
```

固定レコード・パーツはあらゆる種類の処理に使用できますが、最良の用途は、VSAM ファイル、MQSeries® メッセージ・キュー、およびその他の順次ファイルに対する入出力操作です。

EGL は、ある程度、VisualAge Generator などの以前のプロダクトとの互換性が保たれるよう、固定レコード・パーツをサポートしています。固定レコードは、リレーショナル・データベースへのアクセスや汎用処理に使用できますが、なるべくそのような目的に固定レコードを使用しないことをお勧めします。

- 辞書パーツ は、常時使用可能であり、定義を必要としません。辞書パーツに基づいた変数には一連のキーとそれらに関連した値を組み込むことができ、実行時に「キーおよび値」エントリーを追加したり除去したりできます。
- *ArrayDictionary* パーツ は、常時使用可能であり、定義を必要としません。*ArrayDictionary* パーツに基づいた変数を使用すると、一連の配列にアクセスし、すべての配列から同じ番号が付いたエレメントを取り出すことができます。この方法で取り出されたエレメント・セットは、それ自体が 1 つの辞書であり、それぞれの配列名は、配列エレメント内の値と対になったキーとして扱われます。

ArrayDictionary は、『コンソール・ユーザー・インターフェース』で説明されている表示テクノロジーとの関係で特に便利です。

もう 1 つのデータ・パーツとして *DataTable* があり、これは、変数の型ではなく、変数として扱われます。*DataTable* は、複数のプログラムで共用できる生成可能パーツです。これには一連の行と列が入っており、それぞれのセルにはプリミティブ値が組み込まれています。このパーツは、実行単位に対して (ほとんどの場合) グローバルな変数として扱われます。

- *UI* (ユーザー・インターフェース) パーツ は、固定フォント画面および印刷書式でユーザーに表示されるデータのレイアウトを記述します。*UI* パーツは、さまざまなコンテキストで使用され、以下のタイプがあります。
 - サブタイプ *ConsoleForm* のレコード・パーツは、*consoleUI* テクノロジーのコンテキストでユーザーに対して提示されるデータ編成です。他のレコード・パーツと同様に、それぞれが 1 つ以上の変数の型として使用されます。ただし、その場合、それぞれの変数は、レコードでなくコンソール書式 と呼ばれます。*ConsoleUI* テクノロジーには、ユーザーのために定義され、変数の基礎として使用できるその他のパーツも組み込まれています。詳しくは、『コンソール・ユーザー・インターフェース』を参照してください。
 - 書式も、ユーザーに対して提示されるデータ編成です。書式には、*textUI* プログラム内で画面へ送信されるデータを編成するものと、任意の種類のプログラム内でプリンターへ送信されるデータを編成するものがあります。

書式は、固定レコードの場合のような、固定された内部構造を含んでいますが、副構造を含めることはできません。

書式は、次に述べるように、FormGroup に含まれるか、または FormGroup によって参照されている場合にのみ、プログラム、ページ・ハンドラー、またはライブラリーで使えるようになります。

- *FormGroup* パーツ は、テキスト書式と印刷書式のコレクションであり、生成可能パーツです。ほとんどの場合、プログラムには、ヘルプ関連出力用の 1 つの *FormGroup* のほかに、1 つの *FormGroup* のみを組み込むことができます。同じ書式を複数の *FormGroup* に組み込むことができます。

FormGroup 内の各書式は、プログラム全体からアクセスできます。ただし、プログラム固有の *use* ステートメントの中でアクセスを指定する必要があります。書式は、変数として参照されます。

Web ユーザー・インターフェースは、Page Designer で作成します。Page Designer は JSP ファイルをビルドし、それを EGL ページ・ハンドラーへ関連付けます。JSP ファイルは、Web 経由でユーザーと対話するアプリケーション用に、UI パーツの役割を置き換えます。

- *ビルド・パーツ* は、EGL ビルド・ファイル (拡張 *.eglbld*) で定義されており、さまざまな処理特性を定義します。
 - *ビルド記述子パーツ* は生成プロセスを制御し、プロセス中に読み取られた他の制御パーツを示します。
 - 『リンケージ・オプション・パーツ』では、生成されたプログラムの、他のプログラムへの転送方法について、詳しく説明されています。このパーツ内の情報は、生成時、テスト時、および実行時に使用されます。
 - *リソース関連パーツ* は、EGL レコードを、特定のターゲット・プラットフォーム上のファイルにアクセスするために必要な情報に関連付けます。このパーツ内の情報は、生成時、テスト時、および実行時に使用されます。

固定レコード、DataTable、または書式 (テキスト書式であるか印刷書式であるかを問いません) は、*固定構造体* を含んでいます。この構造体は一連のフィールドから構成されており、各フィールドのサイズと型は生成時に既知となり、DataTable または固定レコードの場合は、フィールドに副構造を持たせることができます。

関連する概念

91 ページの『ArrayDictionary』

319 ページの『ビルド記述子パーツ』

477 ページの『VisualAge Generator との互換性』

189 ページの『コンソール・ユーザー・インターフェース』

139 ページの『DataItem パーツ』

87 ページの『辞書』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

142 ページの『固定レコード・パーツ』

149 ページの『関数パーツ』

36 ページの『インポート』

1 ページの『EGL の紹介』

338 ページの『リンケージ・オプション・パーツ』

147 ページの『プログラム・パーツ』

141 ページの『レコード・パーツ』
『パーツの参照』
62 ページの『EGL での変数の参照』
331 ページの『リソース関連とファイル・タイプ』
10 ページの『ランタイム構成』
28 ページの『固定構造体』
30 ページの『Typedef』
199 ページの『Web サポート』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』
523 ページの『EGL エディター』
532 ページの『EGL ソース形式』
93 ページの『EGL ステートメント』
37 ページの『プリミティブ型』

パーツの参照

このセクションでは、名前が参照するパーツを EGL が識別する方法を決定する一連の規則について説明します。これらの規則は、以下の状況で重要です。

- ある関数が他の関数を呼び出す場合
- 非関数パーツ (例えば `dataItem` パーツ) が、バリデーター関数を参照する場合
- パーツが、構造体項目または変数の宣言で `typedef` (形式のモデル) として使用されている場合
- あるパーツが使用宣言で他のパーツを参照する場合
- あるビルド・パーツが他のビルド・パーツを参照する場合

2 番目の規則のセットは、EGL が変数参照を解決する方法を決定します。詳細については、『変数および定数の参照』を参照してください。

基本的可視性規則

最も単純なケースでは、それぞれのパーツを別々のパッケージで宣言せずに、単一パッケージ内に順次定義します。以下のリストは同じ階層レベルにある一連のパーツを示しています (詳細は省略されています)。

```
Function: Function01
Function: Function02
Function: Function03
Record:   Record01
```

同じレベルにあるパーツは相互に使用可能です。例えば、`Function01` は他の関数のいずれかまたは両方を呼び出すことができます。また、`Record01` は 3 つのそれぞれの関数で変数に対する `typedef` として使用できます。

ほとんどの場合、パーツは他のパーツにネストできません。以下のような例外があります。

- プログラム、ライブラリー、またはページ・ハンドラーは、関数をネストできます。ただし、包含関係は直接的である必要があります。関数は、他の関数をネストできません。
- 書式グループは書式をネストできます。

次の例は、ネストされたパーツを使用した例です。

```
Program: Program01
Function: Function01
Function: Function02
Function: Function03
Record: Record01
```

最上位のパーツは、パッケージ内の他のすべてのパーツが利用できます。ただし、ネストされたパーツ (Function01 および Function02) は、そのパッケージ内のパーツのサブセットのみが以下のように利用できます。

- 相互に利用可能
- ネストしているパーツ、およびネストしているパーツが実行時に使用する関数で利用可能例えば Function01 が Function03 を起動する場合には、Function03 は Function02 を起動できます。これは、Function03 が Program01 で使用されるためです。

最後に、ユーザーのコードにテキスト書式または印刷書式が含まれている場合には、これらの書式を含む書式グループにアクセスするには、使用宣言が必要です。使用宣言は、データ・テーブルまたはライブラリーをアクセスする場合にも推奨されます。詳しくは、『使用宣言』を参照してください。

追加可視性規則

ほとんどの開発では、複数のパッケージで共有するパーツが使用されています。これらの規則が有効になります。

- ファイル内の任意のパーツが他のパッケージのパーツを参照できるのは、アクセスされたパーツが以下の特性を持っている場合です。
 - そのパーツが最上位のパーツである
 - そのパーツが *private* として宣言されていない
 - そのパーツが、参照元のパーツと同じプロジェクト内にあるか、または参照元のプロジェクトの EGL ビルド・パス内にリストされたプロジェクト内にある

以下の方法でアクセスできます。

- パーツ名は、パッケージ名で修飾できます。この場合には、ソース・ファイルに `import` ステートメントは必要ありません。例えばパッケージ名が *my.package* で、パーツ名が *myPart* の場合には、次のようにそのパーツを参照できます。

```
my.package.myPart
```

- `import` ステートメントを使用できます。このステートメントの利用には以下の利点があります。
 - パッケージ名を使用してあいまい参照を避ける必要がある場合を除き、`import` ステートメントを使用すると、インポートされたパーツ名を修飾する必要がなくなります。
 - `import` ステートメントによって、ソース・コードで使用されているパッケージを文書化する方法が提供されます。

パーツ名の解決

パーツ参照を解決するために、EGL は 1 つのステップから多数のステップまで含む検索を実施します。以下の文は、各ステップに適用されます。

- 検索は、一意の名前を持つパーツが検索された場合には正常終了する
- 検索は、2 つの同じ名前を持つパーツが検索された場合には異常終了する

これらの状況が考えられます。

- パーツ参照は、パッケージ名を使用して修飾されている。この場合には、検索に含まれるステップは常に 1 つのステップです。
- パーツ参照は、パッケージ名では修飾されておらず、関数呼び出しではない。
- パーツ参照は、パッケージ名では修飾されておらず、関数呼び出しである。

次の文が重要となるのは多くありませんが、最後の 2 つの状況のいずれかに適用される場合があります。

- 参照元のパーツ内のプロパティ **containerContextDependent** は、yes に設定することができます。『*containerContextDependent*』の説明のように、このプロパティを設定して、参照の解決で使用するネーム・スペースを拡張します。
- ユーザー関数の 1 つが、プログラムまたはページ・ハンドラーで可視になっている、EGL システム関数名と同じ名前になっている場合には、システム関数ではなく、そのユーザー関数が参照されます。

パッケージ名が指定されている場合のパーツ名の解決: 前述のとおり、例 `my.package.myPart` のような形式で、パーツの参照時にパッケージ名を指定できます。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。

その参照が、同じパッケージ内にあるパーツからの参照の場合には、以下の文が適用されます。

- パッケージ名は有効 (ただし必須ではない)
- パーツ名は、パーツが `private` として宣言されている場合でも解決される

パッケージ名が指定されていない場合のパーツ名解決 (関数呼び出しを除く): パーツが関数ではなくパーツを参照していて、パッケージ名が指定されていない場合には、検索順序のステップは次のようになります。

1. 参照元のパーツがネストされているコンテナーと同じコンテナーにネストされているパーツを検索する。
2. 参照しているパーツが存在するファイルに明示的にインポートされたパーツを検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。

この場合の各 `import` ステートメントは、特定のパッケージ内の特定のパーツを明示的に参照します。このような明示的 `import` ステートメント で指定されたパーツは、現行パッケージの同じ名前のパーツをオーバーライドするよう動作します。

2 つの異なるプロジェクト内に同じ名前のパッケージが存在する場合、指定された明示型の `import` ステートメントは、EGL ビルド・パスを使用して最初の検索を行い、必要なパーツが見つかるまで検索を停止します。(パーツは、あるプロジェクト内のパッケージに対して固有である必要があります。) 2 つの異なるプロジェクト内で同じ名前のパッケージが存在してもエラーではありませんが、混乱を招くことになりお勧めできません。

同じパーツ名を指定した 2 つの明示型 `import` ステートメントがある場合、エラーが発生します。

3. 参照元のパーツと同じパッケージにある最上位パーツを検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。同じ名前の 2 つのパーツを検索すると、エラーが発生します。
4. 他のインポートされたパーツを検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。

この場合の各 `import` ステートメントは、指定パッケージ内のすべてのパーツを参照し、ワイルドカード `import` ステートメント と呼ばれます。

2 つの異なるプロジェクト内に同じ名前のパッケージが存在する場合、指定されたワイルドカード `import` ステートメントは、EGL ビルド・パスを使用して最初の検索を行い、必要なパーツが見つかりと検索を停止します。(パーツは、あるプロジェクト内のパッケージに対して固有である必要があります。)

複数のワイルドカード `import` ステートメントが同じ名前のパーツを検索すると、エラーが発生します。

パッケージ名が指定されていない場合の関数呼び出し： パーツが関数を呼び出し、パッケージ名を指定していない場合には、検索順序のステップは次のようになります。

1. 起動側がネストされているコンテナと同じコンテナにネストされている関数を検索する。
2. コンテナの使用宣言で指定されているライブラリー内に常駐する関数を検索する。
3. 生成時にコンテナに組み込まれた関数のみを使用して検索を継続する。(同一のコンテナにネストされている、またはライブラリーに常駐している関数以外の関数を組み込むには、コンテナ・プロパティ **`includeReferencedFunctions`** を `yes` に設定します。)

組み込み関数の検索は、次のように行われます。

- a. コンテナが存在するファイルに明示的にインポートされたパーツを検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。

この場合の各 `import` ステートメントは、特定のパッケージ内の特定のパーツを明示的に参照します。このような明示的 `import` ステートメント で指定されたパーツは、現行パッケージの同じ名前のパーツをオーバーライドするよう動作します。

2 つの異なるプロジェクト内に同じ名前のパッケージが存在する場合、指定された明示型の `import` ステートメントは、EGL ビルド・パスを使用して最初の検索を行い、必要な関数が見つかりと検索を停止します。(関数は、あるプロジェクト内のパッケージに対して固有である必要があります。) 2 つの異なるプロジェクト内で同じ名前のパッケージが存在してもエラーではありませんが、混乱を招くことになりお勧めできません。

同じパーツ名を指定した 2 つの明示型 `import` ステートメントがある場合、エラーが発生します。

- b. コンテナと同じパッケージ内の最上位関数を検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。同じ名前の 2 つのパーツが検出されると、エラーが出ます。
- c. 他のインポートされたパーツを検索する。現行プロジェクトは、EGL ビルド・パスにリストされているプロジェクトのいずれかとして扱われます。

この場合の各 `import` ステートメントは、指定パッケージ内のすべてのパーツを参照し、ワイルドカード `import` ステートメント と呼ばれます。

2 つの異なるプロジェクト内に同じ名前のパッケージが存在する場合、指定されたワイルドカード `import` ステートメントは、EGL ビルド・パスを使用して最初の検索を行い、必要なパーツが見つかりと検索を停止します。(パーツは、あるプロジェクト内のパッケージに対して固有である必要があります。)

複数のワイルドカード `import` ステートメントが同じ名前のパーツを検索すると、エラーが発生します。

プログラムの呼び出し

`call` または `transfer` ステートメントでプログラムを呼び出す場合、呼び出し側の引数リストは、呼び出し先プログラムのパラメーター・リストと一致している必要があります。引数およびパラメーターのミスマッチがある場合には、エラーになります。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 36 ページの『インポート』
- 1 ページの『EGL の紹介』
- 19 ページの『パーツ』
- 62 ページの『EGL での変数の参照』

関連する参照項目

- 505 ページの『`containerContextDependent`』
- 517 ページの『EGL ビルド・パスおよび `eglpath`』
- 523 ページの『EGL エディター』
- 532 ページの『EGL ソース形式』
- 1020 ページの『使用宣言』

固定構造体

固定構造体 は、テキスト書式、印刷書式、`dataTable`、または固定レコード・パーツのフォーマットを設定し、一連のフィールドから構成されています。各フィールドは、次の例に示すように、エレメントのメモリー・ロケーションまたはメモリー・ロケーションのコレクションを記述します。

```
10 workAddress;  
20 streetAddress1 CHAR(20);  
30 Line1 CHAR(10);  
30 Line2 CHAR(10);
```

```

20 streetAddress2 CHAR(20);
   30 Line1 CHAR(10);
   30 Line2 CHAR(10);
20 city CHAR(20);

```

上記の例のように、定義の中ですべてのフィールドを直接定義することができます。あるいは、その構造体の全体またはサブセットが、別の固定レコード・パーツ内にある構造体と同じものであることを示す方法もあります。詳細については、『*Typedef*』を参照してください。

フィールドへのアクセスは、まず変数名が基礎となり、その後、ドット構文を持つ一連のフィールド名に基づいて行われます。レコード *myRecord* に前の例で示された構造体が含まれていることを宣言すると、次の各 ID は、メモリー領域を参照します。

```

myRecord.workAddress
myRecord.workAddress.streetAddress1
myRecord.workAddress.streetAddress1.Line1

```

基本構造体フィールド は、従属構造体フィールドを持たず、以下のいずれかの方法によって、メモリーの 1 つの領域を記述します。

- 前述の例のような長さおよびプリミティブ型の仕様
- 『*Typedef*』の説明のように、*dataItem* パーツの宣言をポインティング

前述のように、固定構造体内のフィールドは、従属フィールドを持つことができます。次の例を考えてください。

```

10 topMost;
   20 next01 HEX(4);
   20 next02 HEX(4);

```

上位の構造体フィールド (*topMost* など) を定義する場合には、以下のようなオプションがあります。

- 長さまたはプリミティブ型を割り当てなかった場合、上位の構造体フィールドは CHAR 型となり、EGL が長さを計算します。例えば、*topMost* のプリミティブ型は CHAR であり、長さは 4 です。
- プリミティブ型を割り当てて、長さを割り当てない場合には、EGL は従属構造体項目の特性に基づいて長さを計算します。
- 長さとプリミティブ型の両方を割り当てる場合、その長さは、従属構造体フィールド用に提供されたスペースを反映している必要があります。そうでない場合はエラーが発生します。

注: 固定構造体フィールドのプリミティブ型によって、長さの各単位のバイト数が決まります。詳細については、『*プリミティブ型*』を参照してください。

基本的な各構造体フィールドには、デフォルトで備えているか、またはその構造体フィールドの中で指定されている一連のプロパティーがあります。(構造体フィールドは、それ自体がプロパティーを持つ *dataItem* パーツを参照している場合があります。) 詳細については、『*EGL プロパティーとオーバーライドの概要*』を参照してください。

関連する概念

139 ページの『*DataItem* パーツ』

142 ページの『固定レコード・パーツ』
68 ページの『EGL プロパティの概要』
19 ページの『パーツ』
62 ページの『EGL での変数の参照』
『Typedef』

関連する参照項目

511 ページの『データの初期化』
532 ページの『EGL ソース形式』
37 ページの『プリミティブ型』
71 ページの『SQL 項目のプロパティ』

Typedef

型定義 (typedef) は形式のモデルとして使用されるパーツです。typedef メカニズムは以下のような理由で使います。

- 変数の特性を識別する
- パーツ宣言を再利用する
- フォーマット設定の規則を制定する
- データの意味を明確にする

多くの場合、typedef は抽象グループ化を示します。例えば、*address* という名前のレコード・パーツを宣言して、情報を *streetAddress1*、*streetAddress2*、および *city* に分割できます。従業員レコードに構造体項目 *workAddress* と *homeAddress* が含まれている場合、これらの構造体項目はいずれも *address* という名前のレコード・パーツの形式を指すことができます。このように typedef を使用することにより、アドレス形式が同じになります。

このページに記載された一連の規則の範囲内で、他のパーツを宣言するときや変数を宣言するときにパーツの形式を指すことができます。

パーツを宣言するときはパーツを typedef として使用する必要はありませんが、使用しても構いません。使用例を後述します。また、データ項目の特性を持つ変数を宣言するときに typedef を使用する必要はありません。代わりに変数のすべての特性を指定することができます。パーツを参照する必要はありません。

typedef は、データ項目よりも複雑な変数を宣言する場合も常に 有効です。例えば、**myRecord** という名前の変数を宣言し、**myRecordPart** という名前のパーツの形式を指した場合、EGL では宣言された変数はそのパーツをモデルとして参照します。**myRecordPart02** という名前のパーツの形式を指した場合は、変数の名前は **myRecord** になりますが、この変数は **myRecordPart02** という名前のパーツの特性をすべて保持します。

次の表およびセクションに、さまざまなコンテキストにおける typedefs の詳細を示します。

typedef を指すエントリ	typedef が参照できるパーツのタイプ
関数仮パラメーターまたはその他の関数変数	レコード・パーツまたは dataItem パーツ

typedef を指すエンタリー	typedef が参照できるパーツのタイプ
プログラム・パラメーター	データ項目パーツ、書式パーツ、レコード・パーツ
program variable (non-parameter)	データ項目パーツ、レコード・パーツ
構造体項目	データ項目パーツ、レコード・パーツ

typedef としてのデータ項目パーツ

以下のような場合にデータ項目パーツを typedef として使用できます。

- 変数またはパラメーターを宣言するとき
- レコード・パーツ、書式パーツ、または dataTable パーツのサブユニットである構造体項目を宣言するとき

以下の規則が適用されます。

- 構造体項目が同じ宣言内にリストされた他の構造体項目の親である場合、その構造体項目は、以下の例に示すようにデータ項目パーツの形式のみを指すことができます。

```
DataItem myPart CHAR(20) end

Record myRecordPart type basicRecord
  10 mySI myPart; // myPart が typedef として機能します。
  20 a CHAR(10);
  20 b CHAR(10);
end
```

前のレコード・パーツは以下の宣言と同等です。

```
Record myRecordPart type basicRecord
  10 mySI CHAR(20);
  20 a CHAR(10);
  20 b CHAR(10);
end
```

- データ項目パーツを typedef として使用することはできません。また、typedef を指すエンティティーの長さやプリミティブ型を指定することもできません。

```
DataItem myPart HEX(20) end

// mySI がプリミティブ型を持ち、パーツの形式
// (この場合は myPart) を指すため無効です。
Record myRecordPart type basicRecord
  10 mySI CHAR(20) myPart;
end
```

- レコード・パーツを参照しない変数宣言は、データ項目パーツの形式を指すか、または基本特性を持ちます。(プログラム・パラメーターは、書式パーツも参照できます。) ただし、データ項目パーツでは別のデータ項目パーツの形式や他のパーツを指すことはできません。
- SQL レコード・パーツは、typedef として以下のパーツ・タイプのみ使用できます。
 - 別の SQL レコード・パーツ
 - dataItem パーツ

typedef としてのレコード・パーツ

以下のような場合にレコード・パーツを typedef として使用できます。

- 構造体項目を宣言するとき
- 変数 (パラメーターを含む) を宣言するとき。この場合には、変数は以下に関して typedef が反映されています。
 - フォーマット
 - レコード・タイプ (例えば、indexedRecord または serialRecord)
 - プロパティ値 (例えば、file プロパティの値)

他のパーツの形式を指す構造体項目を宣言するときは、後述するように、typedef によって階層のレベルを追加するかどうかを指定します。

以下の規則が適用されます。

- 再利用を容易にするために構造体項目を使用する場合、レコード・パーツを typedef として使用できます。

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
  10 city CHAR(20);
end

Record record1 type serialRecord
{
  fileName = "myFile"
}
  10 person CHAR(30);
  10 homeAddress address;
end
```

2 番目のレコード・パーツは以下の宣言と同等です。

```
Record record1 type serialRecord
{ fileName = "myFile" }
  10 person CHAR(30);
  10 homeAddress;
  20 streetAddress1 CHAR(30);
  20 streetAddress2 CHAR(30);
  20 city CHAR(20);
end
```

構造体項目が構造体パーツの形式を指すために前の構文を使用すると、EGL ではその構造体項目を含む構造体パーツに階層レベルが追加されます。このため、前述の例の内部構造体には構造体と項目の間に階層が生じ、*person* が *streetAddress1* と異なるレベルになります。

- 場合によっては、単層の構造の方が良い場合もあります。また、リレーショナル・データベースにアクセスする I/O オブジェクトである SQL レコードは、単層構造でなければなりません。
 - 前出の例では、レコード・パーツの構造体項目名 (この場合 *homeAddress*) をワード **embed** と置換して、そのワードの後に typedef として機能するレコード・パーツ名 (この場合 *address*) を続ける場合、パーツ宣言は次のようになります。

```
Record address type basicRecord
  10 streetAddress1 CHAR(30);
  10 streetAddress2 CHAR(30);
```

```

    10 city CHAR(20);
end

Record record1 type serialRecord
{
    fileName = "myFile"
}
    10 person CHAR(30);
    10 embed address;
end

```

これで、レコード・パーツの内部構造体は以下のように単層になりました。

```

Record record1 type serialRecord
{
    fileName = "myFile"
}
    10 person CHAR(30);
    10 streetAddress1 CHAR(30);
    10 streetAddress2 CHAR(30);
    10 city CHAR(20);
end

```

構造体項目名の位置にワード **embed** を使用するの、階層レベルが追加されるのを避けるためです。ワード **embed** で識別される構造体項目には、以下の制限事項があります。

- レコード・パーツの形式を指すことはできるが、データ項目パーツを指すことはできない
- 配列を指定したりプリミティブ型の指定を組み込むことはできない
- 次に、2 つのレコードに同一構造を宣言する際に、レコード・パーツが **typedef** になっている場合を想定します。

```

Record common type serialRecord
{
    fileName = "mySerialFile"
}
    10 a BIN(10);
    10 b CHAR(10);
end

Record recordA type indexedRecord
{
    fileName = "myFile",
    keyItem = "a"
}
    embed common; // accepts the structure of common,
                  // not the properties
end

Record recordB type relativeRecord
{
    fileName = "myOtherFile",
    keyItem = "a"
}
    embed common;
end

```

最後の 2 つのレコード・パーツは、以下の宣言と同等です。

```

Record recordA type indexedRecord
{
    fileName = "myFile",
    keyItem = "a"
}

```



```

    }
    10 a BIN(10);
    10 b CHAR(10);
end

Record recordB type relativeRecord
{
    fileName = "myOtherFile",
    keyItem = "a"
}
    10 a BIN(10);
    10 b CHAR(10);
end

```

- 構造体項目を連続して宣言するときに、レコード・パーツを `typedef` として複数回使用できます。このような再利用は、自宅の住所と職場の住所が含まれた従業員レコード・パーツを宣言する場合などに有用です。基本レコードを使用して、以下のように構造内の 2 つのロケーションに同じ形式を記述できます。

```

Record address type basicRecord
    10 streetAddress1 CHAR(30);
    10 streetAddress2 CHAR(30);
    10 city CHAR(20);
end

Record record1 type serialRecord
{
    fileName = "myFile"
}
    10 person CHAR(30);
    10 homeAddress address;
    10 workAddress address;
end

```

レコード・パーツは以下の宣言と同等です。

```

Record record1 type serialRecord
{
    fileName = "myFile"
}
    10 person CHAR(30);
    10 homeAddress;
        20 streetAddress1 CHAR(30);
        20 streetAddress2 CHAR(30);
        20 city CHAR(20);
    10 workAddress;
        20 streetAddress1 CHAR(30);
        20 streetAddress2 CHAR(30);
        20 city CHAR(20);
end

```

- レコード・パーツを `typedef` として使用することはできません。また、`typedef` を指すエンティティの長さやプリミティブ型を指定することもできません。以下に例を示します。

```

Record myTypedef type basicRecord
    10 next01 HEX(20);
    10 next02 HEX(20);
end

// myFirst がプリミティブ型を持ち、パーツの形式
// を指すため無効です。
Record myStruct02 type serialRecord
{

```



```

        fileName = "myFile"
    }
    10 myFirst HEX(40) myTypedef;
end

```

ただし、以下の例を参照してください。

```

Record myTypedef type basicRecord
    10 next01 HEX(20);
    10 next02 HEX(20);
end

Record myStruct02 type basicRecord
    10 myFirst myTypedef;
end

```

2 番目の構造体は以下の宣言と同等です。

```

Record myStruct02 type basicRecord
    10 myFirst;
    20 next01 HEX(20);
    20 next02 HEX(20);
end

```

従属構造体項目を持つすべての構造体項目のプリミティブ型は、従属構造体項目のプリミティブ型に関係なくデフォルトで **CHAR** であり、構造体項目の長さは従属構造体項目によって表されるバイト数です。詳細については、『構造体』を参照してください。

- SQL レコードに関連して、以下の制限事項があります。
 - SQL レコード・パーツが typedef として他の SQL レコード・パーツを使用している場合には、typedef が提供するそれぞれの項目には、4 バイトの接頭部が含まれます。非 SQL レコードが SQL レコード・パーツを typedef として使用している場合には、接頭部は含まれません。背景情報については、『SQL レコードの詳細』を参照してください。
 - SQL レコード・パーツは、typedef として以下のパーツ・タイプのみ使用できます。
 - 別の SQL レコード・パーツ
 - dataItem パーツ
- つまり、構造および構造体項目のいずれも typedef として指定することはできません。

typedef としての書式

プログラム・パラメーターを宣言する場合に限り、書式パーツを typedef として使用できます。

関連する概念

139 ページの『DataItem パーツ』
 164 ページの『書式パーツ』
 1 ページの『EGL の紹介』
 141 ページの『レコード・パーツ』
 28 ページの『固定構造体』

関連するタスク

146 ページの『EGL プログラム・パーツの作成』

関連する参照項目

93 ページの『EGL ステートメント』

803 ページの『SQL レコードの内部レイアウト』

インポート

`import` ステートメントは、指定したパッケージ内のパーツの集合 (EGL ソース・ファイルの場合)、または指定した一連のファイル内のパーツの集合 (EGL ビルド・ファイルの場合) を識別します。 `import` ステートメントを保持するファイルは、インポートされたパーツがそのファイルと同じパッケージに存在する場合と同様にそれらのパーツを参照できます。

背景

`public` パーツが現行のパッケージ以外のパッケージにあり、`import` ステートメントで識別されていない場合、コードは、次の例のように、パッケージ名 (例えば `my.pkg`) を使用してそのパーツ名を修飾する (例えば `myPart`) 必要があります。

```
my.pkg.myPart
```

パーツが `import` ステートメントで識別されている場合では、コードはそのパッケージ名を除去することができます。この場合には、非修飾パーツ名 (例えば `myPart`) で十分です。

`import` ステートメントを使用してパーツ名を解決する場合の説明については、『[パーツ参照](#)』を参照してください。

import ステートメントの形式

EGL ファイルにおける `import` ステートメントの構文は、以下のとおりです。

```
import packageName.partSelection;
```

packageName

検索するパッケージ名を示します。名前は完全な名前である必要があります。

partSelection

パーツ名またはアスタリスク (*) です。アスタリスクは、パッケージ内のすべてのパーツが選択されることを示します。

ビルド・ファイル内の `import` ステートメントは、インポート・ファイル内のパーツが参照可能な他のビルド・ファイルのパーツを識別します。 `import` ステートメントは、ビルド・ファイル内で `<EGL>` タグの次に続き、各ステートメントは以下の構文になっています。

```
<import file=filePath.egl<bld>
```

filePath

インポートするファイルのパスおよび名前を示します。パスを指定する場合は、次のことが該当します。

- ファイル・パスは、同じプロジェクト内の任意のソース・ディレクトリー、または EGL パス内の他のプロジェクト内の任意のソース・ディレクトリーです。
- 各修飾子は、斜線 (/) で隣の修飾子と分離されます。

ファイル名としてアスタリスク (*) を指定するか、ファイル名の最後の文字にアスタリスクを指定できます。アスタリスクが使用された場合には、EGL は以下の特性を持つすべての .eglbuild ファイルをインポートします。

- 指定したファイル・パス内のファイル。
- アスタリスクの前の文字で始まる名前を持つファイル。(アスタリスクの前に文字がない場合には、そのディレクトリー・パス内のすべてのビルド・ファイルが選択されます。)

ファイル拡張子 .eglbuild はオプションです。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

1 ページの『EGL の紹介』

19 ページの『パーツ』

24 ページの『パーツの参照』

関連するタスク

348 ページの『EGL ビルド・パスの編集』

プリミティブ型

各 EGL プリミティブ型は、メモリーの一領域を表します。プリミティブ型には、文字、数値、および日時の 3 種類があります。

- 文字型は以下のとおりです。
 - *CHAR* は、1 バイト文字を参照します。
 - *DBCHAR* は、2 バイト文字を参照します。dbchar は、EGL V5 のプリミティブ型である DBCS に取って代わります。
 - *MBCHAR* は、単一バイト文字と 2 バイト文字の組み合わせであるマルチバイト文字を参照します。mbchar は、EGL V5 のプリミティブ型である MIX に取って代わります。
 - *STRING* は可変長のフィールドです。このフィールドの 2 バイト文字は、Unicode Consortium によって開発された UTF-16 エンコード標準に準拠します。
 - *UNICODE* は固定フィールドです。このフィールドの 2 バイト文字は、Unicode Consortium によって開発された UTF-16 エンコード標準に準拠します。
 - *HEX* は、16 進文字です。
- 日時型は以下のとおりです。
 - *DATE* は、単一バイト 8 桁の固定長を持つ固有のカレンダー日付を参照します。
 - *INTERVAL* は、単一バイト 2 桁から 27 桁の範囲の長さを持つ時間幅を参照します。
 - *TIME* は、単一バイト 6 桁の固定長を持つ時刻インスタンスを参照します。
 - *TIMESTAMP* は現在時刻を参照し、単一バイト 2 桁から 20 桁の範囲の長さを持ちます。
- ラージ・オブジェクト型は、以下のとおりです。

- *BLOB* は、1 バイトから 2 ギガバイトの範囲の長さを持つバイナリー・ラージ・オブジェクトです。
- *CLOB* は、1 バイトから 2 ギガバイトの範囲の長さを持つ文字ラージ・オブジェクトです。
- 数値型は以下のとおりです。
 - *BIGINT* は、18 桁の整数を格納する 8 バイト領域を参照します。この型は、長さが 8 で、小数部を持たない *BIN* 型と同じです。
 - *BIN* は 2 進数を参照します。
 - *DECIMAL* は、右端のバイトの右半分にある 16 進数 *C* (正数) または 16 進数 *D* (負数) によって符号が表されるパック 10 進文字を参照します。
DECIMAL は、EGL バージョン 5.0 のプリミティブ型である *PACK* に取って代わります。
 - *FLOAT* は、有効数字が 16 桁までの倍精度浮動小数点数を格納する 8 バイト領域を参照します。
 - *INT* は、9 桁の整数を格納する 4 バイト領域を参照します。この型は、長さが 4 で、小数部を持たない *BIN* 型と同じです。
 - *MONEY* は、*DECIMAL* 値として格納される通貨金額を参照します。
 - *NUM* は、右端のバイトの左半分にある符号固有の 16 進値によって符号が表される数字を参照します。ASCII の場合、この値は 3 (正数) または 7 (負数) です。EBCDIC の場合、この値は F (正数) または D (負数) です。
 - *NUMC* は、右端のバイトの左半分にある符号固有の 16 進値によって符号が表される数字を参照します。ASCII の場合、この値は 3 (正数) または 7 (負数) です。EBCDIC の場合、この値は F (正数) または C (負数) です。
 - *PACF* は、右端のバイトの右半分にある 16 進数 *F* (正数) または 16 進数 *D* (負数) によって符号が表されるパック 10 進文字を参照します。
 - *SMALLFLOAT* は、有効数字が 8 桁の単精度浮動小数点数を格納する 4 バイト領域を参照します。
 - *SMALLINT* は、4 桁の整数を格納する 2 バイト領域を参照します。この型は、長さが 2 で、小数部を持たない *BIN* 型と同じです。

固定小数点数値型のフィールドの内部表記は、小数点を指定した場合であっても、整数表記と同じです。例えば、12.34 の表記は 1234 の表記と同じです。同様に、通貨記号は *MONEY* 型のフィールドと一緒に格納されません。

DB2[®] と (直接または JDBC によって) 対話する場合、固定小数点数の最大桁数は 31 桁です。

ANY 型の変数は、その変数に割り当てられた値の型を受け取ります。詳細については、『*ANY*』のトピックを参照してください。

宣言時には、以下の値のそれぞれを表すプリミティブ型を指定します。

- 関数から戻される値
- 名前順で参照され、単一の値を含むメモリーの一領域であるフィールドの値

他のエンティティーにもプリミティブ型があります。

- システム変数には、フィールドに固有のプリミティブ型 (通常は NUM) があります。
- 文字リテラルは以下の型のいずれかです。
 - CHAR: リテラルに 1 バイト文字しか含まれていない場合
 - DBCHAR: リテラルに 2 バイト文字セットの 2 バイト文字しか含まれていない場合
 - MBCHAR: リテラルに 1 バイト文字と 2 バイト文字の組み合わせが含まれている場合
- UNICODE 型の文字リテラルはサポートされていません。

それぞれのプリミティブ型については別個のページで説明します。代入、論理式、関数呼び出し、および call ステートメントについて扱っているページには、追加情報があります。

以降のセクションでは以下の主題について扱います。

- 宣言時におけるプリミティブ型
- さまざまな数値型の相対的な効率

宣言時におけるプリミティブ型

以下の宣言を見てください。

```
DataItem
  myItem CHAR(4)
end
Record mySerialRecordPart type serialRecord
{
  fileName="myFile"
}
10 name CHAR(20);
10 address;
20 street01 CHAR(20);
20 street02 CHAR(20);
end
```

示されているように、以下のエンティティを宣言するときは、プリミティブ型を指定する必要があります。

- プリミティブ変数
- 副構造のない構造体フィールド

副構造のある構造体フィールド (例えば *address*) のプリミティブ型を指定することもできます。このような構造体フィールドのプリミティブ型を指定せずに、コードでこの構造体フィールドを参照する場合、プロダクトでは、以下のことが想定されます。

- 従属構造体フィールドが別の型であっても、プリミティブ型は CHAR と想定される。
- 長さは、従属構造体フィールドのバイト数であると想定される。

さまざまな数値型の相対的な効率

EGL では DECIMAL、NUM、NUMC および PACF 型がサポートされているので、レガシー・アプリケーションによって使用されるファイルやデータベースをよ

り簡単に処理できます。BIN 型のフィールドの計算が最も速いので、新しい開発作業では BIN 型のフィールドを使用するか、同等な整数型 (BIGINT、INT、または SMALLINT) を使用することをお勧めします。効率が最大になるのは、長さ 2 で小数部のない BIN 型 (SMALLINT 型と同等) のフィールドを使用したときです。

計算、代入、および比較においては、小数部のない NUM 型のフィールドの方が、小数部を持つ NUM 型のフィールドよりも効率的です。

DECIMAL、NUM、NUMC、および PACF 型のフィールドのそれぞれの計算効率は同じです。

関連する概念

- 139 ページの『DataItem パーツ』
- 141 ページの『レコード・パーツ』
- 62 ページの『EGL での変数の参照』
- 28 ページの『固定構造体』

関連する参照項目

- 41 ページの『ANY』
- 407 ページの『代入』
- 53 ページの『BIN および整数型』
- 608 ページの『call』
- 41 ページの『CHAR』
- 44 ページの『DATE』
- 42 ページの『DBCHAR』
- 54 ページの『DECIMAL』
- 100 ページの『例外処理』
- 54 ページの『FLOAT』
- 560 ページの『関数呼び出し』
- 42 ページの『HEX』
- 45 ページの『INTERVAL』
- 538 ページの『論理式』
- 43 ページの『MBCHAR』
- 55 ページの『MONEY』
- 55 ページの『NUM』
- 55 ページの『NUMC』
- 546 ページの『数式』
- 726 ページの『演算子と優先順位』
- 56 ページの『PACF』
- 56 ページの『SMALLFLOAT』
- 71 ページの『SQL 項目のプロパティ』
- 43 ページの『STRING』
- 547 ページの『テキスト式』
- 47 ページの『TIME』
- 47 ページの『TIMESTAMP』
- 44 ページの『UNICODE』

ANY

ANY 型の変数は、その変数に割り当てられた値のタイプを受け取ります。値は、INT などのプリミティブ型か、タイプとして使用されたデータ・パーツに基づいた変数とすることができます。書式または dataTable を値とすることはできません。

以下の例を考えてください。

```
myInt INT = 1;
myString STRING = "EGL";

myAny01, myAny02 any;

// myAny01 は値 1 と INT の型を受け取ります。
myAny01 = myInt;

// myAny02 は値「EGL」と STRING の型を受け取ります。
myAny02 = myString;

// 次の文は無効です。なぜなら、
// INT 型の変数を STRING 型の変数へ
// 代入しようとしているからです。
myAny02 = myAny01;
```

無効な方法でタイプを結合しようとするアクションは、実行時にのみ検出され、プログラムの終了の原因となります。それらのアクションには、値を互換性のないタイプのフィールドに代入すること、引数値を互換性のないタイプのパラメーターに渡すこと、式の内部で互換性のない値同士を結合することが含まれます。

リテラルの型は、そのリテラルの値によって暗黙に指定されます。

- 引用符付きストリングは STRING 型
- 4 桁以下の整数は SMALLINT 型
- 5 桁から 8 桁までの整数は INT 型
- 9 桁から 18 桁までの整数は BIGINT 型
- 小数点を含んでいる数値は NUM 型

ANY 型の変数を参照する場合、アクセスは常に動的です。固定構造体 (dataTable、印刷書式、テキスト書式、または固定レコード) の中に ANY 型のフィールドを組み込むことはできません。

関連する参照項目

37 ページの『プリミティブ型』

文字型

CHAR

CHAR 型の項目は一連の 1 バイト文字として解釈されます。長さは文字数とバイト数の両方を反映し、1 から 32767 の範囲です。

Windows® 2000 などのワークステーション・プラットフォームでは、ASCII 文字セットが使用されます。z/OS UNIX® システム・サービスなどのメインフレーム・プラットフォームでは、EBCDIC 文字セットが使用されます。これら 2 種類の環境では照合シーケンスが異なるため、一般に「より大」および「より小」の比較で結果が異なるものになります。

関連する参照項目

37 ページの『プリミティブ型』

DBCHAR

DBCHAR 型の項目は一連の 2 バイト文字として解釈されます。長さは文字数を反映し、1 から 16383 の範囲です。バイト数を判別するには、長さ値を 2 倍してください。

Windows 2000 などのワークステーション・プラットフォームでは、ASCII 文字セットが使用されます。z/OS UNIX システム・サービスなどのメインフレーム・プラットフォームでは、EBCDIC 文字セットが使用されます。これら 2 種類の環境では照合シーケンスが異なるため、一般に「より大」および「より小」の比較で結果が異なるものになります。

DBCS データは表意文字であり、中国語、日本語、および韓国語などの表示に必要です。このデータの表示には、DBCS に対応した端末が必要です。

関連する参照項目

37 ページの『プリミティブ型』

HEX

HEX 型の項目は一連の 16 進数文字 (0 - 9、a - f、および A - F) として解釈され、文字として扱われます。長さは桁数を反映し、1 から 65534 の範囲です。バイト数を判別するには、2 で除算してください。

長さ 4 の項目の場合、値の例の内部ビット表記は以下のようになります。

```
// 16 進値 04 D2
00000100 11010010
```

```
// 16 進値 FB 2E
11111011 00101110
```

HEX 型の項目は、データ型が他の EGL プリミティブ型と一致しないファイルやデータベース・フィールドへのアクセスでよく使用されます。

16 進値を代入するには、以下の例のように、16 進数の範囲の文字のみを含む CHAR 型のリテラルを使用します。

```
myHex01 = "ab02";

myHex02 = "123E";
```

以下の例のように、16 進項目は、論理式のオペランドとして含めることができます。

```
if (myHex01 = "aBCd")
  myFunction01();
else
  if (myHex > myHex02)
    myFunction02();
  end
end
```

16 進項目を算術式に含めることはできません。

関連する参照項目

37 ページの『プリミティブ型』

MBCHAR

MBCHAR 型の項目は、単一バイト文字と 2 バイト文字の組み合わせとして解釈されます。長さは、項目に含めることのできる 1 バイト文字の数とバイト数を反映します。長さは 1 から 32767 です。

Windows 2000 などのワークステーション・プラットフォームでは、ASCII 文字セットが使用されます。z/OS UNIX システム・サービスなどのメインフレーム・プラットフォームでは、EBCDIC 文字セットが使用されます。これら 2 種類の環境では照合シーケンスが異なるため、一般に「より大」および「より小」の比較で結果が異なるものになります。

メインフレーム環境では、項目で 2 バイト文字を使用できる場合、以下のようなシフトアウト文字とシフトイン文字のスペースを含める必要があります。

- 一連の 2 バイト文字の開始を示す、単一バイトのシフトアウト文字 (16 進値 0E)
- この一連の 2 バイト文字の終了を示す、単一バイトのシフトイン文字 (16 進値 0F)

これらのシフトアウト文字とシフトイン文字は、EBCDIC から ASCII へのデータ変換時に削除され、ASCII から EBCDIC へのデータ変換時に挿入されます。可変長レコードを変換する場合に、現行レコードの終わり (レコード長で示される) が MBCHAR 型の構造体項目内にあると、シフトアウト文字とシフトイン文字の挿入や削除を反映するようにレコード長が調整されます。

2 バイト文字データは表意文字であり、中国語、日本語、および韓国語などの表示に必要です。このデータの表示には、2 バイト文字セットに対応した端末が必要です。

関連する参照項目

37 ページの『プリミティブ型』

STRING

プリミティブ型 STRING は、2 バイトの UNICODE 文字で構成されます。

このフィールドの値はファイルやデータベースに保管できます。コードが DB2 UDB と対話する場合は、GRAPHIC データのコード・ページが UNICODE であり、このデータ項目値が保管される列が SQL データ型の GRAPHIC または VARGRAPHIC でなければなりません。

Unicode の詳細については、Unicode Consortium の Web サイト (www.unicode.org) を参照してください。

関連する参照項目

37 ページの『プリミティブ型』

UNICODE

プリミティブ型 UNICODE を使用すれば、一部の人間言語のテキストを処理したり保管したりできます。ただし、テキストはコードの外部から提供される必要があります。UNICODE 型のリテラルはサポートされていません。

UNICODE 型の項目には、以下のことが当てはまります。

- 長さは文字数を反映し、1 から 16383 の範囲です。この項目のために予約されるバイト数は、長さに指定された値の 2 倍です。
- この項目は、UNICODE 型の項目に対してのみ代入や比較を行えます。
- 比較時には、UTF-16 エンコード標準での文字の順序でビット値が比較されます。
- 必要な場合、EGL ではこの項目にユニコードのブランクが埋め込まれます。
- システム・ストリング関数では、この項目は、個々のバイト (ユニコードのブランクが追加された場合はそのブランクを含む) のストリングとして扱われます。これらの関数で指定する長さは、文字数でなくバイト数でなければなりません。
- この項目の値はファイルやデータベースに保管できます。コードが DB2 UDB と対話する場合は、GRAPHIC データのコード・ページが UNICODE であり、このデータ項目値が保管される列が SQL データ型の GRAPHIC または VARGRAPHIC でなければなりません。

Unicode の詳細については、Unicode Consortium の Web サイト (www.unicode.org) を参照してください。

関連する参照項目

37 ページの『プリミティブ型』

日時型

DATE

DATE 型の項目は、特定のカレンダー日付を反映する 8 桁の連続した単一バイトの数字です。

DATE 型の形式は、yyyyMMdd です。

yyyy

年を表す 4 桁の数字。範囲は 0000 から 9999 までです。

MM

月を表す 2 桁の数字。範囲は 01 から 12 までです。

dd

日を表す 2 桁の数字。範囲は 01 から 31 までで、20050230 などの無効な日付を割り当てるコードはエラーになります。

EBCDIC を使用するホスト環境では、値の内部 16 進表記は、例えば次のようになります。

```
// 2005 年 3 月 15 日  
F2 F0 F0 F5 F0 F3 F1 F5
```

Windows 2000 のような ASCII を使用するワークステーション環境では、値の内部 16 進表記は、例えば次のようになります。

```
// 2005 年 3 月 15 日  
32 30 30 35 30 33 31 35
```

DATE 型の項目は、リレーショナル・データベースとの間でデータの受け渡しができます。

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

536 ページの『日時式』

37 ページの『プリミティブ型』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

INTERVAL

INTERVAL 型の項目は、ある間隔を反映する 1 桁から 21 桁までの連続した単一バイトの数字で、この間隔は、2 つの時点の差を数値で表したものです。各桁の意味は、項目を宣言したときに指定したマスクによって決定されます。

間隔は、(2005 から 1980 を減算したときのように) 正の場合と、(1980 から 2005 を減算したときのように) 負の場合があり、項目の先頭には、マスクに反映されない余分なバイトがあります。レコード内に INTERVAL 型の項目がある場合は、レコードの長さを計算するときに、上位項目 (もしあれば) の長さだけでなく、その余分なバイトも考慮する必要があります。

次のどちらの形式のマスクでも指定できます。

- 月幅。これには、年数と月数を含めることができます。
- 秒幅。これには、日数、時間数、分数、秒数、および秒の小数部を含めることができます。

いずれの場合でも、マスク内の 1 文字が 1 桁を表します。例えば、月幅の形式では、一連の *y* が項目内の年数を示します。年数を表すのに 3 桁だけ必要であれば、マスク内に *yyy* を指定します。年数を表すのに最大桁数 (9 桁) が必要であれば、*yyyyyyyyy* を指定します。

指定されたマスク内で、最初の文字を最大 9 回まで (別の指定がない場合) 使用できますが、後続の文字種の数値は、それだけ制限されます。

月幅形式のマスクには、以下の文字を順に使用できます。

y 間隔の年数を表す 0 桁から 9 桁までの数字。

M 間隔の月数を表す 0 桁から 9 桁までの数字。*M* がマスク内の最初の文字でない場合、指定できるのは最大 2 桁までです。

デフォルトのマスクは *yyyyMM* です。

秒幅形式のマスクには、以下の文字を順に使用できます。

d 間隔の日数を表す 0 桁から 9 桁までの数字。

- H* 間隔の時間数を表す 0 桁から 9 桁までの数字。*H* がマスク内の最初の文字でない場合、指定できるのは最大 2 桁までです。
- m* 間隔の分数を表す 0 桁から 9 桁までの数字。*m* がマスク内の最初の文字でない場合、指定できるのは最大 2 桁までです。
- s* 間隔の秒数を表す 0 桁から 9 桁までの数字。*s* がマスク内の最初の文字でない場合、指定できるのは最大 2 桁までです。
- f* それぞれが秒の小数部を表す 0 から 6 桁の数字。最初の数字は 10 分の 1 を表し、2 番目は 100 分の 1 を表し、以下同様です。*f* がマスク内の最初の文字でない場合でも、指定できるのは最大 6 桁までです。

マスクの先頭または末尾で所定の種類の文字を使用しなくてもかまいませんが、中間の文字をスキップすることはできません。以下は有効なマスクです。

```
yyyyyyMM
yyyyyy
MM

ddHHmmssffffff
HHmmssff
mmss
HHmm
```

しかし、以下のマスクは、中間の文字が欠落しているので無効です。

```
// 無効
ddmmssffffff
HHssff
```

デフォルトのマスク (*yyyyMM*) が有効で、項目が、EBCDIC を使用するホスト環境にある場合、値の内部 16 進表記は、例えば次のようになります。

```
// 100 年、2 か月。4E は値が正であることを意味する
4E F0 F1 F0 F0 F0 F2
```

```
// 100 年、2 か月。60 は値が負であることを意味する
60 F0 F1 F0 F0 F0 F2
```

デフォルトのマスク (*yyyyMM*) が有効で、項目が、Windows 2000 のような ASCII を使用するワークステーション環境にある場合、値の内部 16 進表記は、例えば次のようになります。

```
// 100 年、2 か月。2B は値が正であることを意味する
2B 30 31 30 30 30 32
```

```
// 100 年、2 か月。2D は値が負であることを意味する
2D 30 31 30 30 30 32
```

INTERVAL 型の項目は、強く型定義されているので、この型の項目をその他の型の項目と比較したり、その他の型の項目とこの型の項目の間に代入を行ったりすることはできません。

最後に、INTERVAL 型の項目は、リレーショナル・データベースとの間でデータの受け渡しができません。

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

536 ページの『日時式』

37 ページの『プリミティブ型』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

TIME

TIME 型の項目は、特定の瞬間を反映する 6 桁の連続した単一バイトの数字です。

TIME 型の形式は、*HHmmss* です。

HH

時を表す 2 桁の数字。範囲は 00 から 24 までです。

mm

1 時間の中の分を表す 2 桁の数字。範囲は 00 から 59 までです。

ss 1 分の中の秒を表す 2 桁の数字。範囲は 00 から 59 までです。

EBCDIC を使用するホスト環境では、値の内部 16 進表記は、例えば次のようになります。

```
// 8 時 40 分 20 秒  
F0 F8 F4 F0 F2 F0
```

Windows 2000 のような ASCII を使用するワークステーション環境では、値の内部 16 進表記は、例えば次のようになります。

```
// 8 時 40 分 20 秒  
30 38 34 30 32 30
```

TIME 型の項目は、リレーショナル・データベースとの間でデータの受け渡しができます。

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

536 ページの『日時式』

37 ページの『プリミティブ型』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

TIMESTAMP

TIMESTAMP 型の項目は、特定の瞬間を反映する 1 桁から 20 桁までの連続した単一バイトの数字です。各桁の意味は、項目を宣言したときに指定したマスクによって決定されます。

マスクを指定するときは、以下の文字を順に使用できます。

yyyy

年を表す 4 桁の数字。範囲は 0000 から 9999 までです。

MM

月を表す 2 桁の数字。範囲は 01 から 12 までです。

dd 日を表す 2 桁の数字。範囲は 01 から 31 までです。

HH

時を表す 2 桁の数字。範囲は 00 から 23 までです。

mm

分を表す 2 桁の数字。範囲は 00 から 59 までです。

ss 秒を表す 2 桁の数字。範囲は 00 から 59 までです。

f それぞれが秒の小数部を表す 0 から 6 桁の数字。最初の数字は 10 分の 1 を表し、2 番目は 100 分の 1 を表し、以下同様です。

デフォルトのマスクは `yyyyMMddHHmmss` です。

DB2 と (直接または JDBC によって) 対話する場合は、年 (*yyyy*) から秒 (*ss*) まですべてのコンポーネントを指定する必要があります。その他のコンテキストでは、次のことが当てはまります。

- マスクの先頭または末尾で所定の種類の文字を使用しなくてもかまいませんが、中間の文字をスキップすることはできません。
- 以下は有効なマスクです。

```
yyyyMMddHHmmss
yyyy MMss
```

- 以下のマスクは、中間の文字が欠落しているので無効です。

```
// 無効
ddMMssffffff
HHssff
```

デフォルトのマスク (`yyyyMMddHHmmss`) が有効で、項目が、EBCDIC を使用するホスト環境にある場合、値の内部 16 進表記は、例えば次のようになります。

```
// 8:05:10 o'clock on 12 January 2005
F2 F0 F0 F5 F0 F1 F1 F2 F0 F8 F0 F5 F1 F0
```

デフォルトのマスク (`yyyyMMddHHmmss`) が有効で、項目が Windows 2000 のような ASCII を使用するワークステーション環境にある場合、値の内部 16 進表記は、例えば次のようになります。

```
// 8:05:10 o'clock on 12 January 2005
32 30 30 35 30 31 31 32 30 38 30 35 31 30
```

TIMESTAMP 型の項目は、TIMESTAMP 型の項目または DATE 型、TIME 型、NUM 型、CHAR 型の項目と比較する (または相互に代入する) ことができます。しかし、無効な値を代入した場合は、開発時にエラーが発生します。以下に例を示します。

```
// 2 月 30 日は有効な日付でないので無効
myTS timestamp("yyyymmdd");
myTS = "20050230";
```

フル・マスクの先頭の文字が欠落している場合 (例えば、マスクが「dd」の場合)、EGL は、上位レベルの文字 (この例では「yyyyMM」) が、マシン・クロックに従って現在の瞬間を表すものと見なします。次の文を使用すると、2 月にランタイム・エラーが発生します。

```
// 2 月 30 日 という日付はないので無効
myTS timestamp("dd");
myTS = "30";
```

最後に、TIMESTAMP 型の項目は、リレーショナル・データベースとの間でデータの受け渡しができます。

関連する参照項目

407 ページの『代入』

『日付、時刻、およびタイム・スタンプのフォーマット指定子』

536 ページの『日時式』

849 ページの『EGL ライブラリー DateTimeLib』

538 ページの『論理式』

37 ページの『プリミティブ型』

日付、時刻、およびタイム・スタンプのフォーマット指定子

日付、時刻、およびタイム・スタンプのフォーマットは、文字のパターンによって指定され、それぞれの文字は日付または時刻のコンポーネントを表します。これらの文字には大/小文字の区別があり、a から z および A から Z のすべての文字は、日付または時刻のコンポーネントに構文解析されます。

日付または時刻のコンポーネントとしてテキストを構文解析することなく日付、時刻、またはタイム・スタンプ内で文字を表示するには、その文字 (単数または複数) を単一引用符で囲んでください。日付、時刻、またはタイム・スタンプで単一引用符を表示する場合は、単一引用符を 2 個使用します。

次の表では、日付、時刻、またはタイム・スタンプのパターン内の文字およびその値をリストしています。

文字	日付または時刻コンポーネント	型	例
G	紀元指定子	テキスト	AD
y	年	年	1996; 96
M	1 年のうちの月	月	July; Jul; 07
w	1 年のうちの週	数値	27
W	1 か月のうちの週	数値	2
D	1 年のうちの日	数値	189
d	1 か月のうちの日	数値	10
F	1 か月のうちの曜日	数値	2
E	週の曜日	テキスト	Tuesday; Tue
a	AM/PM マーカー	テキスト	PM
H	1 日のうちの時間 (0 から 23)	数値	0
k	1 日のうちの時間 (0 から 24)	数値	24
K	AM/PM での時間 (0 から 11)	数値	0
h	AM/PM での時間 (1 から 12)	数値	12
m	1 時間のうちの分数	数値	30
s	1 分のうちの秒数	数値	55
S	ミリ秒	数値	978
z	タイム・ゾーン	一般タイム・ゾーン	太平洋標準時; PST; GMT-08:00
Z	タイム・ゾーン	RFC 822 タイム・ゾーン	-800
C	世紀	世紀	20; 21

パターン内で連続して使用される各文字の数により、文字のそのグループがどのように解釈および構文解析されるかが決まります。解釈は、文字のタイプによって異なります。また解釈は、パターンがフォーマット設定に使用されるか、構文解析に使用されるかによっても異なります。次のリストでは、文字の型、およびこれらの文字の数の違いによってどのように解釈に影響するかを示しています。

テキスト

フォーマット設定の場合、文字数が 4 文字未満であれば、フル書式が使用されます。4 文字以上のときは、可能な場合、略語が使用されます。構文解析では、パターン文字の数に関係なく、両方の書式が受け入れられます。

数値 フォーマット設定の場合、パターン文字の数は、最小桁数を表します。数字を指定の長さにするために短くする場合は、ゼロが追加されます。構文解析では、2 つの隣接するフィールドを分離するのに必要でないかぎり、パターン文字の数は無視されます。

年 フォーマット設定の場合、パターン文字の数が 2 であれば、年が 2 桁に切り捨てられます。その他の場合は、年は数値型として解釈されます。

構文解析では、パターン文字の数が 2 でない場合、桁数に関係なく、年は文字どおりに解釈されます。例えば、パターン MM/dd/yyyy に値 01/11/12 を割り当てると、January 11, 12 A.D. と解析されます。同じパターンに値 01/02/3 または 01/02/0003 を割り当てると、January 2, 3 A.D. と解析されます。同様に、同じパターンに値 01/02/-3 を割り当てると、January 2, 4 B.C. と解析されます。

構文解析では、パターンが yy の場合、パーサーにより、現在の年を基準としてフル年が判別されます。パーサーは、2 桁で表された年を、処理時点の 80 年前から 20 年後以内と想定します。例えば、現在の年が 2004 年の場合、パターン MM/dd/yy に値 01/11/12 を割り当てると、2012 年 1 月 11 日と解析し、また同じパターンに値 05/04/64 に割り当てると、1964 年 5 月 4 日と解析します。

月 文字パターンの数が 3 以上の場合、月はテキスト型として解釈されます。その他の場合は、年は数値型として解釈されます。

一般タイム・ゾーン

一般タイム・ゾーンは、名前が付けられている場合、テキスト型として解釈されます。GMT オフセット値を表すタイム・ゾーンの場合、次の構文規則が使用されます。

GMTOffsetTimeZone = GMT 符号 時間 : 分

符号 + または -

時間 0 から 23 までの 1 桁または 2 桁の数値。書式はロケールに依存せず、ユニコード規格の基本ローマ字ブロックから取得する必要があります。

分 00 から 59 までの 2 桁の数値。書式はロケールに依存せず、ユニコード規格の基本ローマ字ブロックから取得する必要があります。

構文解析では、RFC 822 タイム・ゾーンも許容されます。

RFC 822 タイム・ゾーン

フォーマット設定には、RFC 822 4 桁タイム・ゾーン形式が使用されます。

RFC822TimeZone = 符号 *TwoDigitHour* : 分

TwoDigitHours は、00 から 23 までの 2 桁の数値でなければなりません。その他の定義は、一般タイム・ゾーンと同じです。

構文解析では、一般タイム・ゾーンも許容されます。

世紀 100 までのフル年 MOD を取る数値型として表示されます。

次の表では、米国のロケールで解釈される日付と時刻パターンのサンプルの一部を記載しています。

日付と時刻のパターン	結果
yyyy.MM.dd G 'at' HH:mm:ss z	2001.07.04 AD at 12:08:56 PDT
EEE, MMM d, 'yy	Wed, Jul 4, '01
h:mm a	12:08 PM
hh 'o'clock' a, zzzz	12 o'clock PM, Pacific Daylight Time
K:mm a, z	0:08 PM, PDT
yyyyy.MMMMM.dd GGG hh:mm aaa	02001.July.04 AD 12:08 PM
EEE, d MMM yyyy HH:mm:ss Z	Wed, 4 Jul 2001 12:08:56 -0700
yyMMddHHmmssZ	010704120856-0700

LOB の型

CLOB

CLOB 型の項目は、1 バイトから 2 ギガバイトの範囲の長さを持つ文字ラージ・オブジェクトを表します。

CLOB 型の項目には、以下のことが当てはまります。

- 個々の項目としてのみ宣言でき、BasicRecords 内ではサポートされません。
- ローカル関数およびプログラム呼び出しに渡すことができます。ラージ・オブジェクトのパラメーターおよび対応する引数は、両方とも同じ型のラージ・オブジェクトとして宣言する必要があります。
- 別の CLOB 変数にのみ割り当てが可能です。
- 別の CLOB 変数に移動可能です。これは、CLOB 変数に割り当てられるのと同じ結果になります。
- BLOB の参照変数を作成できます。
- SQLlocator (CLOB) を使用します。つまり、CLOB には、データそのものではなく、SQL CLOB データへの論理ポインターが含まれています。
- SQLRecord と一緒に使用される場合は次のとおりです。

- CLOB は、文字ラージ・オブジェクト (Character Large Object) をデータベース内の列として表します。
- CLOB は、作成時の変換期間中にのみ有効です。
- リモート・プログラムまたは EGL 以外のプログラムの呼び出しに渡すことはできません。
- assignment ステートメントまたは式でのオペランドとして参照できません。

以下の関数は、CLOB と一緒に使用できます。

- attachClobToFile
- freeClob
- getClobLen
- getStrFromClob
- getSubStrFromClob
- loadClobFromFile
- setClobFromString
- setClobFromStringAtPosition
- truncateClob
- updateClobToFile

関連する参照項目

『BLOB』

891 ページの『EGL ライブラリー LobLib』

892 ページの『attachClobToFile()』

893 ページの『freeClob()』

894 ページの『getClobLen()』

894 ページの『getStrFromClob()』

895 ページの『getSubStrFromClob()』

896 ページの『loadClobFromFile()』

896 ページの『setClobFromString()』

897 ページの『setClobFromStringAtPosition()』

897 ページの『truncateClob()』

898 ページの『updateClobToFile()』

37 ページの『プリミティブ型』

BLOB

BLOB 型の項目は、1 バイトから 2 ギガバイトの範囲の長さを持つバイナリー・ラージ・オブジェクトを表します。

BLOB 型の項目には、以下のことが当てはまります。

- 個々の項目としてのみ宣言でき、BasicRecords 内ではサポートされません。
- ローカル関数およびプログラム呼び出しに渡すことができます。ラージ・オブジェクトのパラメーターおよび対応する引数は、両方とも同じ型のラージ・オブジェクトとして宣言する必要があります。
- 別の BLOB 変数にのみ割り当てが可能です。

- 別の BLOB 変数に移動可能です。これは、BLOB 変数に割り当てられるのと同じ結果になります。
- BLOB の参照変数を作成できます。
- SQLlocator (BLOB) を使用します。つまり、BLOB には、データそのものではなく、SQL BLOB データへの論理ポインターが含まれています。
- SQLRecord と一緒に使用される場合は次のとおりです。
 - BLOB は、バイナリー・ラージ・オブジェクト (Binary Large Object) をデータベース内の列として表します。
 - BLOB は、作成時の変換期間中にのみ有効です。
- リモート・プログラムまたは EGL 以外のプログラムの呼び出しに渡すことはできません。
- assignment ステートメントまたは式でのオペランドとして参照できません。

以下の関数は、BLOB と一緒に使用できます。

- attachBlobToFile
- freeBlob
- getBlobLen
- loadBlobFromFile
- truncateBlob
- updateBlobToFile

関連する参照項目

51 ページの『CLOB』
 891 ページの『EGL ライブラリー LobLib』
 892 ページの『attachBlobToFile()』
 893 ページの『freeBlob()』
 894 ページの『getBlobLen()』
 895 ページの『loadBlobFromFile()』
 897 ページの『truncateBlob()』
 898 ページの『updateClobToFile()』
 37 ページの『プリミティブ型』

数値型

BIN および整数型

BIN 型の項目は 2 進値として解釈されます。長さは 4、9、または 18 で、10 進数形式における符号を除いた桁数 (小数部を含む) を反映します。例えば値 -12.34 は、長さ 4 の項目に収まります。4 桁の数には 2 バイトが必要であり、9 桁の数には 4 バイト、18 桁の数には 8 バイトが必要です。

長さ 4 の項目の場合、値の例の内部ビット表記は以下のようになります。

```
// 10 進数 1234 に対する 16 進値は 04 D2 です
00000100 11010010

// 10 進数 -1234 に対する値は、2 の補数 (FB 2E) です
11111011 00101110
```

算術オペランドや結果、配列の添え字、相対レコードのキー項目などでは、可能な限り、他の数値型でなく **BIN** 型の項目を使用することをお勧めします。

以下の型は、**BIN** 型と同等です。

- **BIGINT** は長さ 18、小数部なし
- **INT** は長さ 9、小数部なし
- **SMALLINT** は長さ 4、小数部なし

関連する参照項目

37 ページの『プリミティブ型』

DECIMAL

DECIMAL 型の項目は、各ハーフバイトが 16 進文字である数値であり、右端のバイトの右半分にある 16 進数 **C** (正数) または 16 進数 **D** (負数) によって符号が表されます。

長さは桁数を反映し、1 から 32 までの範囲です。

バイト数を判別するには、長さ値に 2 を加え、その合計を 2 で割り、結果から小数部を切り捨ててください。

長さ 4 の項目の場合、値の例の内部 16 進表記は以下のようになります。

```
// 10 進数 123 の場合
00 12 3C
```

```
// 10 進数 -123 の場合
00 12 3D
```

```
// 10 進数 1234 の場合
01 23 4C
```

```
// 10 進数 -1234 の場合
01 23 4D
```

ファイルやデータベースから **DECIMAL** 型のフィールドに読み込まれた負の値は、16 進数 **D** が **B** に置き換えられる場合があります。この値は **EGL** で受け入れられますが、**B** が **D** に変換されます。

DB2 UDB での **DECIMAL** 型の列の形式は、**DECIMAL** 型のホスト変数の形式と同じです。

関連する参照項目

37 ページの『プリミティブ型』

FLOAT

FLOAT 型の項目は、有効数字が最大 16 桁までの倍精度浮動小数点数の 2 進値として解釈されます。長さは、8 バイトに固定されます。**EGL** により生成された Java プログラムでは、値の範囲は 4.9e-324 から 1.7976931348623157e308 までです。

FLOAT は、以下の各定義に対応します。

- リレーショナル・データベース管理システムでの **FLOAT** データ型。

- C、C++、または Java の **double** データ型。

浮動小数点数値については、Java 形式とホスト COBOL 形式間の形式変換が DB2 でサポートされていますが、ホスト・プログラムの呼び出しではサポートされていません。

関連する参照項目

37 ページの『プリミティブ型』

MONEY

MONEY 型の項目は、ほとんどの場合、DECIMAL 型の項目と同じ数値です。MONEY の場合、長さのデフォルトは 16、小数点以下の桁数のデフォルトは 2、最小長は 2 で、出力フィールドには通貨記号が表示されます。MONEY は、IBM Informix 4GL の MONEY データ型に対応します。

形式は、変数 `defaultMoneyFormat` に基づいています。

関連する参照項目

54 ページの『DECIMAL』

71 ページの『フォーマット設定プロパティ』

37 ページの『プリミティブ型』

NUM

NUM 型の項目は、各バイトが文字形式の数字になっている数値であり、右端のバイトの左半分にある符号固有の 16 進値によって符号が表されます。長さは桁数とバイト数の両方を反映します。長さは 1 から 32 です。

長さ 4 の項目の場合、EBCDIC を使用するホスト環境であれば、値の例の内部 16 進表記は以下のようになります。

```
// 10 進数 1234 の場合  
F1 F2 F3 F4
```

```
// 10 進数 -1234 の場合  
F1 F2 F3 D4
```

Windows 2000 などの ASCII を使用するワークステーション環境であれば、値の例の内部 16 進表記は以下のようになります。

```
// 10 進数 1234 の場合  
31 32 33 34
```

```
// 10 進数 -1234 の場合  
31 32 33 74
```

関連する参照項目

37 ページの『プリミティブ型』

NUMC

NUMC 型のフィールドの値は、各バイトが文字形式の数字になっている数値であり、右端のバイトの左半分にある符号固有の 16 進値によって符号が表されます。長さは桁数とバイト数の両方を反映し、1 から 18 までの範囲です。

長さ 4 のフィールドの場合、CICS for EBCDIC を使用するホスト環境であれば、サンプル値の内部 16 進表記は以下のようになります。

```
// 10 進数 1234 の場合  
F1 F2 F3 C4
```

```
// 10 進数 -1234 の場合  
F1 F2 F3 D4
```

このフィールドが Windows 2000 などの ASCII を使用するワークステーション環境にある場合、サンプル値の内部 16 進表記は以下のようになります。

```
// 10 進数 1234 の場合  
31 32 33 34
```

```
// 10 進数 -1234 の場合  
31 32 33 74
```

関連する参照項目

37 ページの『プリミティブ型』

PACF

PACF 型のフィールドは、各ハーフバイトが 16 進文字である数値であり、右端のバイトの右半分にある 16 進数 F (正数) または 16 進数 D (負数) によって符号が表されます。長さは桁数を反映し、1 から 18 の範囲です。バイト数を判別するには、長さ値に 2 を加え、その合計を 2 で割り、結果から小数部を切り捨ててください。

長さ 4 のフィールドの場合、サンプル値の内部 16 進表記は以下のようになります。

```
// 10 進数 123 の場合  
00 12 3F
```

```
// 10 進数 -123 の場合  
00 12 3D
```

```
// 10 進数 1234 の場合  
01 23 4F
```

```
// 10 進数 -1234 の場合  
01 23 4D
```

ファイルやデータベースから PACF 型のフィールドに読み込まれた負の値は、16 進数 D が B に置き換えられる場合があります。この値は EGL で受け入れられますが、B が D に変換されます。

関連する参照項目

37 ページの『プリミティブ型』

SMALLFLOAT

SMALLFLOAT 型の項目は、有効数字が最大 8 桁までの単精度浮動小数点数の 2 進値として解釈されます。長さは、4 バイトのメモリー・ストレージに固定されています。

EGL により生成された Java プログラムでは、値の範囲は 3.40282347e+38 から 1.40239846e-45 までです。

SMALLFLOAT は、以下の各定義に対応します。

- リレーショナル・データベース管理システムでの SMALLFLOAT データ型。
- C、C++、または Java の **float** データ型。

浮動小数点数値については、Java 形式とホスト COBOL 形式間の形式変換が DB2 でサポートされていますが、ホスト・プログラムの呼び出しではサポートされていません。

関連する参照項目

37 ページの『プリミティブ型』

EGL での変数と定数の宣言

以下の方法で変数を宣言できます。

- 次の例のように、複数のプリミティブ型のうちの 1 つを変数のベースにすることができます。

```
myItem CHAR(10);
```

- 次の例のように、dataItem パーツ、レコード・パーツ、または固定レコード・パーツを変数のベースにすることができます。

```
myRecord myRecordPart;
```

- 次の例のように、辞書または arrayDictionary の特定の構成を変数のベースにすることができます。

```
myVariable Dictionary
{
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
    birthday="021460"
};
```

- プログラムまたはその他の生成可能パーツは、プログラムまたは実行単位に対してグローバルな変数として扱われる、dataTable のフィールドにアクセスできます。プログラムの使用宣言の 1 つに dataTable がリストされている場合、より単純な構文を使用してそれらのフィールドにアクセスできます。
- プログラムは、プログラムに対してグローバルな変数として扱われる、テキストまたは印刷書式のフィールドにアクセスできます。プログラムは、関連した formGroup を使用宣言に含めていなければなりません。
- プログラムまたはその他の生成可能なロジック・パーツは、ライブラリー関数の外側で宣言されるライブラリー変数にアクセスできます。それらの変数は、実行単位に対してグローバルです。プログラムの使用宣言の 1 つにライブラリーがリストされている場合、より単純な構文を使用してそれらのフィールドにアクセスできます。

定数を宣言するには、シンボル CONST の後に、定数名、タイプ、等号、および値を続けて指定します。指定された値は、実行時に変更できません。以下に例を示します。

```
const myString String = "Great software!";
const myArray BIN[] = [36, 49, 64];
const myArray02 BIN[] [] = [[1,2,3],[5,6,7]];
```

定数は、レコードまたはその他の複雑な構造に入れることはできません。

最後に、単一の文内で複数の変数または定数を宣言するには、次の例のように、ID 同士をコンマで区切ります。

```
const myString01, myString02 STRING = "INITIAL";  
myItem01, myItem02, myItem03 CHAR(5);  
myRecord01, myRecord02 myRecordPart;
```

関連する概念

24 ページの『パーツの参照』

19 ページの『パーツ』

30 ページの『Typedef』

関連する参照項目

37 ページの『プリミティブ型』

1020 ページの『使用宣言』

動的アクセスと静的アクセス

EGL は、変数参照を静的アクセスまたは動的アクセスによって次のように解決します。

- **動的アクセス** が使用されている場合、フィールド名とタイプは実行時にのみ既知になります。コードでは、コード内の値から、またはランタイム入力から名前の判別する必要があります。

動的アクセスは、コードが次のいずれかを参照しているときに使用されます。

- プリミティブ型が ANY である変数。
- 辞書内の値フィールド。このフィールドは ANY 型です。
- レコード内のフィールド。ただし、そのフィールドへ導いた (レコードからサブフィールドへのフィールドへの) 関係チェーンが、前の参照で動的アクセスを使用したものである場合。
- EGL 大括弧構文によって参照されているフィールド。この場合、フィールド名は識別子の規則に従っている必要はなく、EGL 予約語にすることができ、他の場合に無効となるスペースやその他の文字を含んでいてもかまいません。

詳細については、『動的アクセス用の大括弧構文 (*Bracket syntax for dynamic access*)』を参照してください。

- **静的アクセス** が使用されている場合、フィールド名とタイプは生成時に既知であり、名前は常に EGL 識別子の命名規則に従っています。その名前は、実行時には使用されません。

静的アクセスは、コードが次のいずれかを参照しているときに使用されます。

- コンテナの外部にあり、型が ANY 以外である変数。
- 固定レコード内のフィールド。
- 非固定レコード内のフィールド。ただし、そのフィールドへ導いた (変数からサブフィールドへのフィールドへの) 関係チェーンが、すべての参照で静的アクセスを使用したものである場合。

辞書内の値が、1 つの固定レコードと 1 つの非固定レコードを含んでいる例を考えてみます。

```
// 固定レコード・パーツ
Record myFixedRecordPart type=serialRecord
{
    fileName = "myFile"
}
10 ID INT;
10 Job CHAR(10);
end

// レコード・パーツ (非固定)
Record myDynamicRecordPart type=basicRecord
ID INT;
Job CHAR(10);
end

Program myProgram

dynamicPerson myDynamicRecordPart;
myFlexID INT;

fixedPerson myFixedRecordPart;
myFixedID INT;

Function main()

dynamicPerson.ID = 123;
dynamicPerson.Job = "Student";

fixedPerson.ID = 456;
fixedPerson.Job = "Teacher";

relationship Dictionary
{
    dynamicRecord=dynamicPerson,
    staticRecord=fixedPerson
};
end
end
end
```

以下の規則が適用されます。

- 辞書値への参照は動的参照であり、それに従属するすべての参照は動的参照です。コードに以下の文が含まれていた場合を考えてみます。

```
myDynamicID INT;
myDynamicID = relationship.dynamicRecord.ID;
```

dynamicRecord への参照は動的参照となり、ID への参照は動的参照となり、識別子 ID は実行時に可視になります。

- 固定構造体で始まる参照は、その構造体の内部にあるメモリーのみを参照できます。この例では、fixedPerson で始まる参照は、固定レコード内のフィールド ID および JOB にアクセスできますが、それ以外のフィールドにはアクセスできません。
- コードでは、固定構造体に動的にアクセスできますが、同じ参照文でそのフィールドのフィールドにアクセスすることはできません。この例では、次の参照は、識別子 ID が実行時に使用可能でないので、無効です。


```
myFixedID INT;  
  
// 無効  
myFixedID = relationship.fixedRecord.ID;
```

この問題は、次のように別の固定レコードを宣言し、それに辞書内の固定レコードからの値を代入することによって処理できます。

```
myFixedID INT;  
myOtherRecord myFixedRecordPart;  
myOtherRecord = relationship.staticRecord;  
myFixedID = myOtherRecord.ID;
```

動的アクセスは、代入 (左辺または右辺)、論理式、および **set**、**for**、**openUI** の各文の中で有効です。

関連する概念

64 ページの『動的アクセス用の大括弧構文』
87 ページの『辞書』
147 ページの『プログラム・パーツ』
62 ページの『EGL での変数の参照』
30 ページの『Typedef』

関連するタスク

57 ページの『EGL での変数と定数の宣言』

関連する参照項目

407 ページの『代入』
538 ページの『論理式』
37 ページの『プリミティブ型』
685 ページの『set』

有効範囲指定の規則と EGL での「this」

EGL パーツで変数または定数を宣言した場合、宣言の中で使用された識別子は、そのパーツ全体で有効範囲内 (使用可能) となります。

- 宣言が 1 つの関数の中で行われた場合、識別子はその関数のローカルな有効範囲内に置かれます。例えば、関数 `Function01` で変数 `Var01` を宣言した場合、`Function01` 内のすべてのコードで `Var01` を参照できます。この識別子は、宣言より前に置かれた関数コード内でも使用可能です。

この変数を、引数として別の関数に渡すことができますが、元の識別子をその関数の中で使用することはできません。このパラメーター名が、受け取り側の関数内で使用可能なのは、そのパラメーター名がそこで宣言されていたからです。

- 宣言がプログラムなどの生成可能パーツ内にあり、どの関数内にもない場合、識別子はプログラム・グローバルな有効範囲内にあります。つまり、そのパーツによって呼び出されるすべての関数で、その識別子を参照できます。たとえば、プログラムが `Var01` を宣言して `Function01` を呼び出し、`Function01` が `Function02` を呼び出す場合でも、`Var01` を両方の関数のどこからでも使用できます。

テキスト書式または印刷書式の中に置かれた識別子は、その書式を参照する生成可能パーツに対してグローバルです。それらの識別子は、その書式を表示する関数より前に置かれた関数の中でも使用できます。

- 宣言がライブラリー内にあり、どの関数内にもない場合、識別子は**実行単位の有効範囲**内にあります。つまり、その実行単位内のすべてのコードに対してグローバルです。
- `dataTable` の名前とそのフィールドは、`dataTable` のプロパティーの設定と、その `dataTable` が存在する環境に応じて、プログラム・グローバルな有効範囲、実行単位の有効範囲、またはそれより大きい有効範囲の中に置くこともできます。

同一の識別子を同じ有効範囲の中に置くことはできません。しかし、ほとんどの識別子は、論理的にコンテナ（レコードなど）の内部にあるメモリー領域を参照します。また、そのような場合、コードでは、その識別子を格納しているコンテナの名前で識別子を修飾します。例えば、関数変数 `myString` が `myRecord01` というレコード内にある場合、コードでは、次のように、その変数をレコードのフィールドとして参照します。

```
myRecord01.myString
```

同じ識別子が 2 つの有効範囲内にある場合、その識別子への参照は、すべて最もローカルな有効範囲への参照となりますが、次のように修飾子を使用して、その振る舞いをオーバーライドできます。

- あるプログラムが変数 `Var01` を宣言して関数を呼び出しており、その関数自体が同じ名前の変数を宣言している場合を考えてみます。その関数内での `Var01` への非修飾参照は、ローカルに宣言された変数にアクセスします。

ローカル識別子が優先されるときでもプログラム・グローバルな識別子にアクセスするには、次の例のように、識別子をキーワード `this` で修飾します。

```
this.Var01
```

まれな例ですが、キーワード `this` は、`assignment` ステートメント内で値の設定ブロックの振る舞いをオーバーライドするためにも使用されます。詳細については、『*値の設定ブロック (Set value blocks)*』を参照してください。

- 次のような事例を考えてみます。
 - プログラムには、ライブラリーにアクセスするための使用宣言があります。また、
 - このプログラムとライブラリーは、それぞれが、`Var01` という名前の変数を宣言しています。

プログラム内の関数に `Var01` への非修飾参照が含まれている場合、その関数はプログラム変数にアクセスします。

実行単位の有効範囲内の識別子に、たとえ別の識別子によって防止されてもアクセスしたい場合には、次の例のように、識別子をパーツ名で修飾します（この例では `myLib` がライブラリーの名前です）。

```
myLib.Var01
```

ライブラリーまたは `dataTable` が別のパッケージ内にあり、そのパーツを `import` ステートメントの中で参照していない場合は、次の例のように、パーツ名の前にパッケージ名を付けます (この例では、`myPkg` がパッケージ名です)。

```
myPkg.myLib.Var01
```

パッケージ名は常にパーツ名を修飾し、変数や定数の識別子の直前に置くことはできません。

最後に、ローカル識別子が `dataTable` またはライブラリーの存在する場所と異なるパッケージ内にある場合は、ローカル識別子を `dataTable` 名またはライブラリー名と同じものにすることができます。その `dataTable` 名またはライブラリー名を参照するには、パッケージ名を組み込みます。

関連する概念

- 149 ページの『関数パーツ』
- 151 ページの『`basicLibrary` タイプのライブラリー・パーツ』
- 151 ページの『`basicLibrary` タイプのライブラリー・パーツ』
- 207 ページの『ページ・ハンドラー』
- 19 ページの『パーツ』
- 147 ページの『プログラム・パーツ』
- 24 ページの『パーツの参照』
- 『EGL での変数の参照』
- 68 ページの『EGL プロパティの概要』
- 28 ページの『固定構造体』
- 30 ページの『`Typedef`』

関連するタスク

- 57 ページの『EGL での変数と定数の宣言』

関連する参照項目

- 560 ページの『関数呼び出し』
- 570 ページの『EGL ソース形式の関数パーツ』

EGL での変数の参照

2 種類のメモリー・アクセスの違いについて、詳しくは、『動的アクセスと静的アクセス』を参照してください。

どの種類のアクセスが使用されているかに関係なく、通常では EGL のドット構文で十分です。例えば、次のパーツ定義を考えてみます。

```
Record myRecordPart01 type basicRecord
  myString      STRING;
  myRecordVar02 myRecordPart02;
end
```

```
Record myRecordPart02 type basicRecord
  myString02     STRING;
  myRecordVar03  myRecordPart03;
  myDictionary   Dictionary
  {
    empnum=0005,
    lastName="Twain",
    firstName="Mark",
  }
```

```

        birthday="021460"
    };
end

Record myRecordPart03 type basicRecord
    myInt INT;
    myDictionary Dictionary
    {
        customerNum=0005,
        lastName="Clemens"
    };
end

```

ある関数で、*myRecordVar01* という名前の変数を宣言するときに、レコード・パーツ *myRecordPart01* をタイプとして使用するとします。

フィールド *myInt* を参照するには、以下のシンボルをこの順序でリストします。

- 変数の名前 (この例では、*myRecordVar01*)
- ピリオド (.)
- 求めるフィールドに至るフィールドのリストで、ピリオドを指定して識別子同士を分離する (例えば、*myRecordVar02.myRecordVar03*)
- 求めるフィールド名の前にピリオドを付けたもの (この例では、*.myInt*)

配列が存在する場合は、これと同じ構文を単に拡張します。例えば、*myRecordVar03* が 3 つのレコードの配列として宣言されている場合は、次のシンボルを使用して、その配列の第 3 エレメント内にある *myInt* フィールドにアクセスできます。

```
myRecordVar01.myRecordVar02.myRecordVar03[3].myInt
```

ドット構文は、この例で辞書フィールドを参照するときにも有効です。値「Twain」にアクセスするには、*assignment* ステートメントの右辺に次の文字を指定します。

```
myRecordVar01.myRecordVar02.myDictionary.lastName
```

myDictionary という名前のフィールドが 2 つの異なるレコード・パーツ内に存在しても、問題になりません。なぜなら、それぞれの同名フィールドは、それを格納している独自のレコードとの関係で参照されるからです。

ドット構文を使用して、次のように、ライブラリー (*myLib* など) 内の定数 (*myConst* など) を参照することもできます。

```
myLib.myConstant
```

それ以外に、以下の 2 つの構文があります。

- 動的アクセスを使用する場合は、フィールド名を引用符付きのストリングとして指定するか、**STRING** 型の識別子として指定できます。この機能は、主に次のような事例で辞書エントリ (「キーおよび値」ペア) の追加または取り出しを行うときに使用されます。
 - キーが、EGL 予約語であるか、識別子内では無効な文字 (ピリオドやスペースなど) を含んでいる場合。または、
 - ストリング定数を使用してキーの代入または参照を行いたい場合。

この構文では、変数、定数、またはリテラルを 1 対の大括弧 ([]) の中に入れる必要があります。中身が入っている大括弧は、ドットの直後に有効な識別子を

置いたものと等価であり、その 2 つの構文を混在させてもかまいません。ただし、参照の先頭は識別子であることが必要です。

例については、『動的アクセス用の大括弧構文』を参照してください。

- 固定構造体 (dataTable、テキスト書式、印刷書式、または固定レコード) 内のフィールドを参照するために、簡略構文を使用することもできます。しかし、なるべくこの構文を避け、前述の完全修飾を使用することをお勧めします。

簡略構文が固定構造体との関係で有効となるのは、プロパティ **allowUnqualifiedItemReferences** を *yes* に設定した場合のみです。このプロパティは、プログラムやライブラリー、ページ・ハンドラーなどの生成可能ロジック・パーツの特性の 1 つであり、デフォルト値は *no* です。

詳細については、『静的アクセス用の簡略構文 (Abbreviated syntax for static access)』を参照してください。

関連する概念

66 ページの『固定構造体を参照するための簡略構文』
『動的アクセス用の大括弧構文』
58 ページの『動的アクセスと静的アクセス』
524 ページの『EGL での列挙型』
149 ページの『関数パーツ』
19 ページの『パーツ』
147 ページの『プログラム・パーツ』
24 ページの『パーツの参照』
60 ページの『有効範囲指定の規則と EGL での「this」』
28 ページの『固定構造体』
30 ページの『Typedef』

関連するタスク

57 ページの『EGL での変数と定数の宣言』

関連する参照項目

78 ページの『配列』
560 ページの『関数呼び出し』
570 ページの『EGL ソース形式の関数パーツ』
37 ページの『プリミティブ型』
1020 ページの『使用宣言』

動的アクセス用の大括弧構文

動的アクセスが有効であるすべての場所で、ストリング変数、定数、またはリテラルを大括弧で囲んで使用することにより、フィールドを参照できます。中身が入っている 1 対の大括弧は、ドットの直後に有効な識別子を置いたものと等価です。

辞書宣言の中で指定されたキーは、EGL 識別子の規則を満たしている必要がありますが、EGL の `assignment` ステートメントの大括弧構文を使用することにより、より広範囲のキーを指定できます。次の例では、大括弧構文が必要です。この例では、2 つのエントリーが辞書に追加され、それらの各エントリーに入っている値が取り出されます。

```

row Dictionary { lastname = "Smith" };
category, motto STRING;

row["Record"] = "Reserved word";
row["ibm.com"] = "Think!";

category = row["Record"];
motto    = row["ibm.com"]

```

ドット構文で識別子を使用して値を参照する場合は、その識別子と等価であるストリングを使用することにより、大括弧構文で同じ値を参照できます。以下の代入には、同じ効果があります。

```

row.age = 20;
row["age"] = 20;

```

myRecordVar01 というレコードを宣言し、このレコードに myRecordVar02 というフィールドが含まれており、myRecordVar02 は、それ自体が前の辞書を含んでいるレコードであるとしてします。次の参照は有効です。

```
myRecordVar01.myRecordVar02.row.lastName
```

この参照では、ほとんどの場合、アクセスは静的です。動的アクセスは、辞書内のフィールドにアクセスしたときに始まります。ただし、以下の定数が有効範囲内にあるとします。

```

const SECOND STRING = "myRecordVar02";
const GROUP  STRING = "row";
const LAST   STRING = "lastName";

```

前の参照は、次のようにコーディングできます。

```
myRecordVar01[SECOND][GROUP][LAST]
```

参照内の最初のシンボルは、常に有効な識別子であることが必要ですが、この例の場合、動的アクセスはその識別子の後で使用されます。

ドット構文と大括弧構文を混在させることもできます。例えば、次の参照は、前の参照と等価です。

```
myRecordVar01[SECOND].row[LAST]
```

最後の例として、次のような配列指標を使用した参照を考えてみます。

```
myRecordVar01.myRecordVar02.myRecordVar03[3][2].myInt
```

以下の定数が有効範囲内にあるとします。

```

const SECOND STRING = "myRecordVar02";
const THIRD  STRING = "myRecordVar03";
const CONTENT STRING = "myInt";

```

前の参照は、次のような方法でコーディングできます。

```
myRecordVar01[SECOND][THIRD][3][2][CONTENT]
```

```
myRecordVar01[SECOND][THIRD][3][2].myInt
```

```
myRecordVar01.myRecordVar02.THIRD[3][2][CONTENT]
```

関連する概念

66 ページの『固定構造体を参照するための簡略構文』

58 ページの『動的アクセスと静的アクセス』
149 ページの『関数パーツ』
19 ページの『パーツ』
147 ページの『プログラム・パーツ』
24 ページの『パーツの参照』
62 ページの『EGL での変数の参照』
60 ページの『有効範囲指定の規則と EGL での「this」』
28 ページの『固定構造体』
30 ページの『Typedef』

関連するタスク

57 ページの『EGL での変数と定数の宣言』

関連する参照項目

78 ページの『配列』
560 ページの『関数呼び出し』
570 ページの『EGL ソース形式の関数パーツ』
718 ページの『MQ レコード用のオプション・レコード』
37 ページの『プリミティブ型』
1020 ページの『使用宣言』

固定構造体を参照するための簡略構文

以下の規則は、dataTable、テキスト書式、印刷書式、または固定レコード内のフィールドを参照する場合に適用されます。

- 固定レコードなどのコンテナに入っているフィールドを参照する場合は、通常のドット構文を使用して、参照先のメモリー領域を明確に示すことができます。例えば、次のパーツ宣言について考えましょう。

```
Record myRecordPart type serialRecord
{
    fileName = "myFile"
}
10 myTop;
20 myNext;
30 myAlmost;
40 myChar CHAR(10);
40 myChar02 CHAR(10);
end
```

ある関数で、*myRecordVar* という名前の変数を宣言するときに、レコード・パーツ *myRecordPart* をタイプとして使用するとします。

myRecordVar の中の *myChar* を有効に参照するには、以下のようにします。

```
myRecordVar.myTop.myNext.myAlmost.myChar
```

上記の参照は、完全修飾 されていると見なされます。

- あるフィールドを参照したい場合、そのフィールドの名前が構造体内で固有ならば、変数名の後にピリオドを付け、その後にフィールド名を指定することができます。先程の例の場合は、以下のようなシンボルを使った参照が有効です。

```
myRecordVar.myChar
```

上記の参照は、部分修飾 されていると見なされます。

フィールド名を部分修飾する方法は、これ以外にありません。例えば、変数名とフィールド名の間に存在するフィールド名を一部のみ組み込んだり、フィールドの上位にある構造体フィールド名をいくつか残したまま変数名を除去したりすることはできません。前述の例の場合、以下の参照は無効です。

```
// 無効
myRecordVar.myNext.myChar
myRecordVar.myAlmost.myChar
myNext.myChar
myAlmost.myChar
```

- 名前の前に修飾子を付けずにフィールドを参照することもできます。先程の例の場合は、以下のようなシンボルを使った参照が有効です。

```
myChar
myChar02
```

これらの参照は、*非修飾* と見なされます。

- 構造体フィールドへの参照はすべて、あいまいさをなくするために必要な程度まで修飾する必要があります。
- 関連したメモリー領域が充てん文字、つまり名前が重要でない領域である場合は、構造体フィールドの名前をアスタリスク (*) にすることができます。参照にアスタリスクを含めることはできません。以下の例を考えてください。

```
record myRecordPart type serialRecord
{
  fileName = "myFile"
}
10 person;
20 *;
30 streetAddress1 CHAR(30);
30 streetAddress2 CHAR(30);
30 nation CHAR(20);
end
```

変数 *myRecordVar* を宣言するときに、このパーツをタイプとして使用する場合は、*myRecordVar.nation* または *nation* を参照できますが、以下の参照は無効です。

```
// 無効
myRecordVar.*.streetAddress1
myRecordVar.*.streetAddress2
myRecordVar.*.nation
```

- EGL が参照を解決しようとするときは、ローカル変数の名前が最初に検索され、次に同じ関数の入出力に使用されるレコード内の構造体フィールドの名前、次にローカル構造体フィールドの名前、次にプログラム・グローバルな名前という順序で検索されます。

ある関数で、*nation* というプリミティブ変数と、次のような基本レコードを指す変数の両方を宣言する場合を考えてみます。

```
record myRecordPart
10 myTop;
20 myNext;
30 nation CHAR(20);
end
```

nation への非修飾参照は、構造体フィールドでなくプリミティブ変数を参照します。

- 名前の検索では、プログラム・グローバル構造体フィールドよりもプログラム・グローバル・プリミティブ変数が優先されることはありません。あるプログラムで、*nation* というプリミティブ変数と、次のような基本レコードの形式を指す変数の両方を宣言する場合を考えてみます。

```
record myRecordPart
  10 myTop;
  20 myNext;
  30 nation CHAR(20);
end
```

nation への非修飾参照は、*nation* がプリミティブ変数と構造体フィールドのどちらでも参照できるために失敗します。構造体フィールドを参照するには、参照を修飾する以外に方法がありません。

追加の規則については、『配列』および『使用宣言』を参照してください。

関連する概念

- 149 ページの『関数パーツ』
- 19 ページの『パーツ』
- 147 ページの『プログラム・パーツ』
- 24 ページの『パーツの参照』
- 62 ページの『EGL での変数の参照』
- 60 ページの『有効範囲指定の規則と EGL での「this」』
- 28 ページの『固定構造体』
- 30 ページの『Typedef』

関連するタスク

- 57 ページの『EGL での変数と定数の宣言』

関連する参照項目

- 78 ページの『配列』
- 560 ページの『関数呼び出し』
- 570 ページの『EGL ソース形式の関数パーツ』
- 718 ページの『MQ レコード用のオプション・レコード』
- 37 ページの『プリミティブ型』
- 1020 ページの『使用宣言』

EGL プロパティの概要

ほとんどの EGL パーツは、生成時に適切な出力を作成するための一連のプロパティを備えています。有効なプロパティ・セットは、コンテキストによって変化します。

- それぞれのパーツ型ごとにプロパティ・セットが定義され、プロパティ・セット全体がパーツ型用に使用されます。例えば、個々のプログラム・パーツには、**alias** というプロパティがあり、これはコンパイル可能単位の名前を識別します。

パーツ自体がサブタイプである場合は、追加のプロパティを使用できます。

textUI タイプのプログラムには、**alias** というプロパティのほかに、**inputForm**

というプロパティがあります。後者は、プログラム・ロジックが実行される前にユーザーに提示されるテキスト書式を識別します。

- 多くのパーツ型では、そのパーツ型のコンポーネントである任意のプリミティブ・フィールドで使用するプロパティのセットも定義します。例えば、SQL レコード・タイプのレコード・パーツには、一連のプリミティブ・フィールドが組み込まれており、それぞれに、フィールドがアクセスする SQL テーブル列を識別する **column** プロパティがあります。

DataItem パーツで使用可能なプロパティには、任意のコンテキストにおいて有効なプリミティブ・フィールド・レベルのすべての プロパティが含まれています。例えば、9 桁の (そして 9 桁のみの) ID を表す **DataItem** パーツを考えてみます。この事例では、ID は *SSN* というリレーショナル・データベース列に関連付けられている場合があります。

```
DataItem IDPart CHAR(9)
{
  minInput = 9,      // 9 文字の入力が必要
  isDigits = yes,    // 数字でなければならない
  columnName = "SSN" // 列へ関連付けられている
}
```

次のようにして、**IDPart** タイプの変数を宣言できます。

```
myVariable IDPart;
```

この変数は、レコード・パーツなどのコンポジット・パーツの中で宣言するか、プログラムなどのロジック・パーツの中で直接宣言することができます。いずれの場合でも、指定プロパティが使用されるかどうかは、パーツ型によって決定されます。

この例では、プロパティ **columnName** が使用されるのは、変数が **SQLRecord** タイプのレコード内で宣言されている場合のみです。2 つの妥当性検査プロパティは、変数がページ・ハンドラーなどのユーザー・インターフェース・パーツの中で宣言されている場合にのみ使用されます。

- 一部の変数宣言では、関連するパーツ定義の中で指定されていたプロパティをオーバーライドできますが、それは、変数が宣言されているコンテキストの中で、そのプロパティを使用できる場合に限られます。
 - コンテキスト内のオーバーライドは、**DataItem** パーツに基づいた変数を宣言するときに可能です。次の文は、(前に定義された) *SSN* タイプのページ・ハンドラー・フィールドを宣言しますが、この文はユーザーによる数字の入力を必要としません。

```
myVariable IDPart { isDigits = no };
```

この例では、プロパティ **minInput** はオーバーライドによって影響を受けず、プロパティ **columnName** は無視されます。

- ほとんどの場合、オーバーライドは、レコード・パーツなどのコンポジット・パーツのプロパティについては不可能です。
- 固定構造体を定義した場合、基本的な構造体フィールドにプロパティを割り当て、関連する変数を宣言するときに、それらのプロパティをオーバーライドできます。また、従属的な構造体フィールドを持つ構造体フィールドにプロパティ

ーを割り当てることもできますが、その場合、割り当てられたプロパティは無視されます。ただし、プロパティのドキュメンテーションで別の説明がある場合は除きます。

- プリミティブ型の変数を宣言するときに、変数宣言のコンテキスト内で使用できる任意のプリミティブ・フィールド・レベル・プロパティを設定できます。

実行時にプロパティにアクセスすることはできません。例えば、SQL レコード・パーツに基づいた変数を作成する場合、**tableNames** プロパティ（これは、そのレコードのアクセス先となる SQL テーブルを識別します）に割り当てられた名前の取り出しや変更を行うロジックを作成することはできません。変数宣言の中でプロパティ値をオーバーライドしたとしても、開発時に指定した値をロジックで変更することはできません。

プロパティ値へのランタイム・アクセスの欠如は、変数の内容を代入したり、その変数を引数として使用したときに、プロパティ値が内容と一緒に転送されないことを意味します。例えば、SQL レコード間でデータをコピーした場合、コピー先レコードのアクセス先となる SQL テーブルの指定には、変更が加えられません。同様に、SQL レコードを EGL 関数に渡す場合、パラメーターはフィールド内容を受け取りますが、開発時に割り当てられていた SQL テーブルの指定を保存します。

ConsoleField などの事前定義 EGL パーツには、プロパティとフィールドの両方を組み込むこともできます。フィールドは、プロパティと異なり、実行時に使用可能です。フィールド値を読み取るロジックを作成でき、多くの場合、フィールド値を変更するロジックも作成できます。

値の設定ブロックは、プロパティとフィールドの両方の値を設定できるコード領域です。詳細については、『値の設定ブロック』を参照してください。

関連する概念

- 62 ページの『EGL での変数の参照』
- 72 ページの『値の設定ブロック』

関連する参照項目

- 553 ページの『EGL ソース形式の書式パーツ』
- 71 ページの『SQL 項目のプロパティ』

フィールド表示プロパティ

EGL フィールド表現プロパティは、フィールドを画面内の出力に表示した場合に有効な特性を指定します。これは、宛先が コマンド・ウィンドウの場合で、Web ブラウザーの場合は該当しません。

プロパティには、次のものがあります。

- 745 ページの『color』
- 754 ページの『highlight』
- 755 ページの『intensity』
- 763 ページの『outline』

また、宛先がプリンターまたは印刷ファイルの場合、フィールドを印刷可能出力に表示したときに以下のプロパティーが有効になります。

- **highlight** プロパティー (ただし、*underline* および *noHighlight*)
- **outline** プロパティー。2 バイト文字をサポートする装置の場合にのみ有効。

フィールド表示プロパティーは、テキスト書式からプログラムに戻されるデータに対しては有効ではなく、完全に出力用です。

フォーマット設定プロパティー

フォーマット設定プロパティーは、データが書式または Web ブラウザーで表示された場合に有効になる特性を指定します。

- 744 ページの『align』
- 747 ページの『currency』
- 748 ページの『currencySymbol』
- 748 ページの『dateFormat』
- 753 ページの『fillCharacter』
- 755 ページの『isBoolean』
- 758 ページの『lineWrap』
- 759 ページの『lowerCase』
- 759 ページの『masked』
- 763 ページの『numericSeparator』
- 763 ページの『outline』
- 767 ページの『sign』
- 770 ページの『timeFormat』
- 771 ページの『timeStampFormat』
- 772 ページの『upperCase』
- 778 ページの『zeroFormat』

関連する概念

68 ページの『EGL プロパティーの概要』

SQL 項目のプロパティー

SQL 項目のプロパティーは、項目が `SQLRecord` 型のレコード内で使用するときの意味のある特性を指定します。ただし、デフォルト値が用意されているため、SQL 項目のプロパティーを指定する必要はありません。

プロパティーには、次のものがあります。

- 746 ページの『column』
- 756 ページの『isNullable』
- 757 ページの『isReadOnly』
- 759 ページの『maxLen』
- 764 ページの『persistent』
- 768 ページの『sqlDataCode』

- 769 ページの『sqlVariableLen』

検証プロパティ

検証プロパティは、ユーザーがテキスト書式にデータを入力する場合に受け入れられる内容を制限します。

プロパティには、次のものがあります。

- 753 ページの『fill』
- 754 ページの『inputRequired』
- 754 ページの『inputRequiredMsgKey』
- 756 ページの『isDecimalDigit』
- 756 ページの『isHexDigit』
- 760 ページの『minimumInput』
- 760 ページの『minimumInputMsgKey』
- 761 ページの『needsSOSI』
- 771 ページの『typeChkMsgKey』
- 773 ページの『validatorDataTable』
- 774 ページの『validatorDataTableMsgKey』
- 774 ページの『validatorFunction』
- 775 ページの『validatorFunctionMsgKey』
- 776 ページの『validValues』
- 777 ページの『validValuesMsgKey』

値の設定ブロック

値の設定ブロックは、プロパティとフィールドの両方の値を設定できるコード領域です。背景情報については、『EGL プロパティの概要』を参照してください。

値の設定ブロックは、以下のアクションを行うときに使用できます。

- パーツの定義
- 変数の宣言
- 特殊な書式の `assignment` ステートメントのコーディング
- **openUI** ステートメントのコーディング (『*openUI*』の説明を参照)

最後の 2 つの事例では、フィールドにのみ、値を代入できます。

注: 固定構造体内のフィールドには、1 つの制限事項が適用されます。値の設定ブロックを使用して、プリミティブ・フィールド・レベル・プロパティに値を代入できますが、フィールド自体の値を設定することはできません。

基本的な状態での値の設定ブロック

基本的な事例で、ほとんどの場合に適用される規則について考えてみます。

- それぞれの値の設定ブロックは、左中括弧 ({) で始まり、複数のエントリーをコンマで区切ったリストか単一のエントリーを含んでおり、右中括弧 (}) で終わります。
- すべてのエントリーは、次の 2 つの形式のどちらかになっています。
 - それぞれのエントリーが識別子と値のペア (**inputRequired = yes** など) で構成される。または、
 - 配列の連続したエレメントへ連続した値が代入される場合、それぞれのエントリーには、位置によって代入される値が入っている。

いずれの場合でも、値の設定ブロックは、変更されようとしているパーツ、変数、またはフィールドの有効範囲内にあります。分かりやすいように、さまざまな構文の例を以下に示します。

最初の例として、2 つのプロパティ (**inputRequired** と **align**) を持つ **dataItem** パーツを次に示します。

```
// 値の設定ブロックの有効範囲は myPart です。
DataItem myPart INT
{
  inputRequired = yes,
  align = left
}
end
```

次の例は、プリミティブ型の変数を示しています。

```
// 有効範囲は myVariable です。
myVariable INT
{
  inputRequired = yes,
  align = left
};
```

次の例は、2 つのレコード・プロパティ (**tableNames** と **keyItems**) を含んでいる SQL レコード・パーツ宣言を示しています。

```
// 有効範囲は myRecordPart です。
Record myRecordPart type SQLRecord
{ tableNames = ["myTable"],
  keyItems = ["myKey"] }
myKey CHAR(10);
myOtherKey CHAR(10);
myContent01 CHAR(60);
myContent02 CHAR(60);
end
```

次の例は、直前のパーツをタイプとして使用し、2 つのレコード・プロパティの 1 つをオーバーライドし、レコード内の 2 つのフィールドを設定する変数宣言を示しています。

```
// 有効範囲は myRecord です。
myRecord myRecordPart
{
  keyItems = ["myOtherKey"],
  myContent01 = "abc",
  myContent02 = "xyz"
};
```

追加の例として、いくつかの変数宣言と **assignment** ステートメントを以下に示します。


```
// この例は、変数宣言の中で
// レコード・プロパティをオーバーライド
// できる唯一の事例を示しています。// 有効範囲は myRecord です。
myRecord myRecordPart {keyItems = ["myOtherKey"]};

// 有効範囲は myInteger で、これは配列です。
myInteger INT[5] {1,2,3,4,5};

// 以下の assignment ステートメントには、
// 値の設定ブロックがありません。
myRecord02.myContent01 = "abc";
myRecord02.myContent02 = "xyz";

// 次の簡略 assignment ステートメントは、
// 上記の 2 つと等価であり、
// 有効範囲は myRecord02 です。
myRecord02
{
    myContent01="abc",
    myContent02="xyz"
};

// 次の簡略 assignment ステートメントは、
// 前に宣言された配列の
// 最初の 4 エlementをリセットします。
myInteger{6,7,8,9};
```

簡略 assignment ステートメントを固定構造体内のフィールドに使用することはできません。

フィールドのフィールド用の値の設定ブロック

フィールドのフィールドに値を代入するときは、使用する構文の中で、各エントリがそのフィールドのみを変更するような有効範囲内に値の設定ブロックを入れるようにします。

次のパーツ定義を考えてみます。

```
record myBasicRecPart03 type basicRecord
    myInt04 INT;
end

record myBasicRecPart02 type basicRecord
    myInt03 INT;
    myRec03 myBasicRecPart03;
end

record myBasicRecPart type basicRecord
    myInt01 INT;
    myInt02 INT;
    myRec02 myBasicRecPart02;
end
```

以下のようにして、任意のフィールドにプロパティ値を代入できます。

- レコードの値の設定ブロックを作成します。
- 有効範囲を絞り込むために、一連のフィールド名を埋め込みます。
- フィールド固有の値の設定ブロックを作成します。

プロパティ値を代入する構文には、次の例に示す 3 つの書式のどれでも使用できます。これらは、フィールド myInt04 に適用されます。

```
// ドット構文
// (『EGL での変数の参照』を参照)。
myRecB myBasicRecPart
{
    myRec02.myRec03.myInt04{ align = left }
};

// 大括弧構文
// (『動的アクセス用の大括弧構文』を参照)。
// この構文を使用して、固定構造体内のフィールドに
// 影響を及ぼすことはできません。
myRecC myBasicRecPart
{
    myRec02["myRec03"]["myInt04">{ align = left }
};

// 中括弧構文
myRecA myBasicRecPart
{
    myRec02 {myRec03 { myInt04 { align = left }}}
};
```

複雑な事例でも、値の設定ブロック内の個々のエントリーをコンマで区切ります。ただし、そのブロックがネストされているレベルを考慮する必要があります。

```
// ドット構文
myRecB myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02.myRec03.myInt04{ align = left },
    myRec02.myInt03 = 6
};

// 大括弧構文
myRecC myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02["myRec03"]["myInt04">{ align = left },
    myRec02["myInt03"] = 6
};

// 中括弧構文。
// ただし、これを使用すると保守がずっと難しくなります。
myRecA myBasicRecPart
{
    myInt01 = 4,
    myInt02 = 5,
    myRec02
    {
        myRec03
        { myInt04
          { action = label5 }},
        myInt03 = 6
    }
};
```

「this」の使用

変数宣言または `assignment` ステートメントの中で、レコード・プロパティーと同じ名前が付いたフィールド (*keyItems* など) を含んでいるコンテナ (SQL レコード など) を使用できます。そのプロパティーでなくフィールドを参照するには、キー

ワード **this** を使用します。これは、値の設定ブロック、または値の設定ブロック内にあるエントリーの正しい有効範囲を設定します。

次のレコード宣言を考えてみます。

```
Record myRecordPart type SQLRecord
{
  tableNames = ["myTable"],
  keyItems = ["myKey"] }
myKey CHAR(10);
myOtherKey CHAR(10);
keyItems CHAR(60);
end
```

次のレコード宣言は、最初にプロパティ `keyItems` の値を設定し、次に同名のフィールドの値を設定します。

```
myRecord myRecordPart
{
  keyItems = ["myOtherKey"],
  this.keyItems = "abc"
};
```

次のセクションに、配列宣言での追加の例を示します。

値の設定ブロック、配列、および配列エレメント

動的配列を宣言する場合、初期のエレメント数を次の例のように指定できます。

```
col1 ConsoleField[5];
```

値の設定ブロックでの代入は、タイプ `ConsoleField` の各初期エレメントのプロパティと事前定義フィールドを参照します。ただし、後で追加されたエレメントは参照しません。

```
col1 ConsoleField[5]
{
  position = [1,1],
  color = red
};
```

変数宣言内の特定のエレメントに値を代入するには、そのエレメントを有効範囲とする埋め込みの値の設定ブロックを作成します。次の例に示すように、有効範囲を指定するには、キーワード **this** と大括弧で囲んだ指標を使用します。

```
// 第 2、第 4 エレメントに値を代入します。
col1 ConsoleField[5]
{
  this[2] { color = blue },
  this[4] { color = blue }
};
```

キーワード **this** の別の使用法の詳細については、『[有効範囲指定の規則と EGL](#)』での「*this*」を参照してください。

値の設定ブロックの中で定位置エントリーを使用して、以下のタイプの配列に入っている連続したエレメントに値を代入できます (レポートの処理、またはコンソール書式の作成を行う場合に限りです)。

- `ConsoleField`
- `Menu`
- `MenuItem`

- Prompt
- Report
- ReportData

次の例を OpenUI ステートメントに入れることもできます。埋め込みの値の設定ブロックの有効範囲は、いずれも特定の配列エレメントです。

```
new Menu
{
  labelText = "Universe",
  MenuItems =

  // プロパティ値は動的配列です。
  [
    new MenuItem
    { name = "Expand",
      labelText = "Expand" },
    new MenuItem
    { name = "Collapse",
      labelText = "Collapse" }
  ]
}
```

追加の例

以下のパーツを考えてみます。

```
Record Point
  x, y INT;
end

Record Rectangle
  topLeft, bottomRight Point;
end
```

次のコードは有効です。

```
Function test()
  screen Rectangle
  {
    topLeft{x=1, y=1},
    bottomRight{x=80, y=24}
  };

  // 次のコードと等価のステートメントを使用して、
  // コード内の x、y を変更します。
  //   screen.topLeft.x = 1;
  //   screen.topLeft.y = 2;
  screen.topLeft{x=1, y=2};
end
```

次に、Point タイプのエレメントのからなる動的配列を、同じ関数内で初期化します。

```
pts Point[2]
{
  this[1]{x=1, y=2},
  this[2]{x=2, y=3}
};
```

この時点で配列に入っている各エレメントの値を設定した後、次のように最初のエレメントを別の値に設定します。

```
pts{ x=1, y=1 };  
pts[1]{x=10, y=20};
```

上記の例で、`this[1]` でなく `pts[1]` が使用される理由は、配列名があいまいだからです。

次に、`Point` タイプの別の動的配列を考えてみます。

```
points Point[];
```

次の `assignment` ステートメントは、エレメントが存在しないので何も効果がありません。

```
points{x=1, y=1};
```

一方、次の `assignment` ステートメントは、存在しない特定のエレメントを参照しているので、範囲外による例外を起こします。

```
points[1]{x=10, y=20};
```

次のようにすると、配列にエレメントを追加し、その後、単一の文を使用してすべてのエレメントに値を設定できます。

```
points.resize(2);  
points{x=1, y=1};
```

関連する概念

- 64 ページの『動的アクセス用の大括弧構文』
- 68 ページの『EGL プロパティの概要』
- 62 ページの『EGL での変数の参照』
- 60 ページの『有効範囲指定の規則と EGL での「this」』

関連する参照項目

- 『配列』
- 511 ページの『データの初期化』
- 669 ページの『openUI』

配列

EGL では、以下の種類の配列がサポートされます。

- 『動的配列』
- 82 ページの『構造体フィールド配列』

いずれの場合でも、サポートされる次元の最大数は 7 です。

動的配列

レコード、固定レコード、またはプリミティブ変数の配列を宣言すると、その配列は、次のように、配列内のエレメントに依存しない独自性を持ちます。

- その配列に固有の関数セットがあり、それらの関数を使用して、エレメントの数を実行時に増減できます。
- 配列固有プロパティ **maxSize** は、その配列内で有効なエレメントの数を示します。デフォルト値は無制限で、エレメントの数はターゲット環境の要件によってのみ制限を受けます。

宣言の中でエレメントの数を指定する必要はありませんが、宣言した場合は、その数が初期のエレメント数を示します。また、宣言の中で、一連の配列定数をリストすることによって初期のエレメント数を指定することもできます。ただし、これはプリミティブ変数でのみ可能であり、レコードでは不可能です。

動的配列を宣言するための構文を以下の例で示します。

```
// エレメントの数が 5 つ以下の配列
myDataItem01 CHAR(30)[] { maxSize=5 };

// エレメントの数が 6 つ以下の配列で、
// 初期のエレメントは 4 つ
myDataItem02 myDataItemPart[4] { maxSize=6 };

// エレメントなしの配列
// ただし、最大サイズは可能な限り最大となります。
myRecord myRecordPart[];

// 3 つのエレメントの配列で、エレメントには
// 値 1、3、5 が代入されます。
position int[] = [1,3,5];
```

リテラル整数を使用してエレメントの数を初期設定できますが、変数と定数は、どちらも無効です。

配列の配列を宣言する場合、初期のエレメント数が有効となるのは、指定された最も左側の次元と、それ以降、初期数が指定されない次元までの各次元です。以下のような宣言が有効です。

```
// 有効。maxsize で最初の次元の
// 最大値を指定
myInt01 INT[3][];
myInt02 INT[4][2][] {maxsize = 12};
myInt03 INT[7][3][1];

// 次の例では、配列定数により、外側の配列が
// 初期に 3 つのエレメントを持つことを示しています。
// 外側の配列の第 1 エレメントは、2 つの
// エレメント (値は 1 と 2) からなる配列です。
// 外側の配列の第 2 エレメントは、3 つのエレメント
// (値は 3、4、5) からなる配列です。
// 外側の配列の第 3 エレメントは、2 つの
// エレメント (値は 6 と 7) からなる配列です。
myInt04 INT[][] = [[1,2],[3,4,5],[6,7]];
```

次の例に示す構文は無効です。その理由は、(例えば) 配列 myInt04 がエレメントなしの配列として宣言されているにもかかわらず、それぞれのエレメントに 3 つのエレメントが代入されているからです。

```
// 無効
myInt04 INT[][3];
myInt05 INT[5][][2];
```

プログラムまたは関数のパラメーターとして指定された配列が、エレメントの数を指定することはできません。

コードが配列または配列エレメントを参照する際には、次の規則が適用されます。

- エレメント添え字には、整数に変換される任意の数式を使用できますが、数式に関数呼び出しを含めることはできません。

- コードで動的配列を参照する場合、添え字を指定しないと、配列全体が参照されます。

メモリー不足状態は破局的エラーとして扱われ、プログラムが終了します。

動的配列関数

動的配列ごとに、一連の関数および読み取り専用変数が使用可能です。次の例では、配列は *series* と呼ばれます。

```
series.resize(100);
```

配列の名前に一連の大括弧を組み込み、それぞれの大括弧に整数を入れることもできます。以下に例を示します。

```
series INT[] [];

// 配列の配列である series の
// 第 2 エレメントのサイズを変更
series[2].resize(100);
```

以下のセクションの中の *arrayName* を配列名に置き換えてください。また、名前はパッケージ名またはライブラリー名、あるいはその両方のいずれでも修飾できることに注意してください。

appendAll():

```
arrayName.appendAll(appendArray Array in)
```

この関数は以下のことを実行します。

- *appendArray* によって参照される配列のコピーを追加して、*arrayName* によって参照される配列に付加する
- 追加したエレメントの数だけ配列サイズを増分する
- 付加したエレメントごとに適切な索引値を割り当てる

appendArray のエレメントの型は、*arrayName* のエレメントの型と同じでなければなりません。

appendElement()

```
arrayName.appendElement(content ArrayElement in)
```

この関数は、指定された配列の末尾にエレメントを置き、サイズを 1 ずつ増分します。 *content* は、適切な型の変数に置き換えることができます。または、操作中に作成されたエレメントに割り当てられるリテラルを指定できます。このプロセスではデータがコピーされるため、ある変数を割り当てても、その変数を比較または他の目的に使用できます。

リテラルを割り当てるための規則は、『代入』で説明したとおりです。

getMaxSize()

```
arrayName.getMaxSize ( ) returns (INT)
```

この関数は、その配列内で許容されるエレメントの最大値を示す整数を戻します。

getSize()

`arrayName.getSize ()` returns (INT)

この関数は、その配列内のエレメントの数を示す整数を戻します。動的配列を処理するときは、この関数を `SysLib.size` の代わりに使用することをお勧めします。

次に示すもう 1 つの関数は、`arrayName.getSize` に相当する機能を提供します。

SysLib.size() returns (INT)

ただし、動的配列を処理するときは、`arrayName.getSize ()` を使用することをお勧めします。

insertElement()

`arrayName.insertElement (content ArrayElement in, index INT in)`

この関数は以下のことを実行します。

- あるエレメントを、現在配列内の指定されたロケーションにあるエレメントの前に置く。
- 配列サイズを 1 つずつを増分する
- 挿入されたエレメントの後ろにある各エレメントの指標を増分する

`content` は新しい内容 (配列の適切な型の定数または変数) で、`index` は新規エレメントのロケーションを示す整数リテラルまたは数値変数です。

`index` が配列内のエレメント数より大きい場合は、関数は新規エレメントを配列の末尾に作成し、配列サイズを 1 つずつ増分します。

removeAll()

`arrayName.removeAll ()`

この関数は、メモリーから配列のエレメントを除去します。配列を使用することはできますが、配列のサイズはゼロです。

removeElement()

`arrayName.removeElement(index INT in)`

この関数は、指定されたロケーションのエレメントを除去し、配列サイズを 1 つずつ減分し、除去されたエレメントの後ろにある各エレメントの指標を減分します。

`index` は、除去するエレメントのロケーションを示す整数リテラルまたは数値変数です。

resize()

`arrayName.resize(size INT in)`

この関数は、配列の現行サイズを `size` (整数リテラル、定数、変数のいずれか) で指定されたサイズまで増減します。 `size` の値が、その配列に許容される最大サイズより大きい場合は、実行単位が終了します。

reSizeAll()

```
arrayName.reSizeAll(sizes INT[  
] in)
```

この関数は、多次元配列のすべての次元を増減します。パラメーター *sizes* は整数の配列で、連続したそれぞれのエレメントが、逐次 1 つの次元のサイズを指定します。サイズ変更された次元数が *arrayName* 内の次元数より大きい場合、または *sizes* 内のエレメントの値が *arrayName* の相応の次元内で許容される最大サイズより大きい場合は、実行単位が終了します。

setMaxSize()

```
arrayName.setMaxSize (size INT in)
```

その配列内で許容されるエレメントの最大値を設定します。この最大サイズをその配列の現行サイズより小さく設定した場合は、実行単位が終了します。

setMaxSizes()

```
arrayName.setMaxSizes(sizes INT[  
] in)
```

この関数は、多次元配列のすべての次元を設定します。パラメーター *sizes* は整数の配列で、連続したそれぞれのエレメントが、逐次 1 つの次元の最大サイズを指定します。指定された次元数が *arrayName* 内の次元数より大きい場合、または *sizes* 内のエレメントの値が *arrayName* の相応の次元内で現行エレメント数より小さい場合は、実行単位が終了します。

引数およびパラメーターとしての動的配列の使用

動的配列は、EGL 関数に引数として渡すことができます。関連するパラメーターは、引数と同じ型の動的配列として定義する必要があり、データ項目については、型の長さおよび小数点以下の桁数は同じでなければなりません (存在する場合)。

動的配列は、別のプログラムに引数として渡すことはできません。

動的配列をパラメーターとして使用する関数の例は、以下のとおりです。

```
Function getAll (employees Employee[])  
;  
end
```

実行時に、パラメーターの最大サイズは、対応する引数に宣言された最大サイズになります。関数または呼び出し先プログラムは、配列のサイズを変更することができ、その変更は呼び出し側コードで有効になります。

SQL 処理と動的配列

EGL では、動的配列を使用してリレーショナル・データベースの行にアクセスできます。複数行を読み取る方法の詳細については、『*get*』を参照してください。複数行を追加する方法の詳細については、『*add*』を参照してください。

構造体フィールド配列

構造体フィールド配列を宣言するのは、次の例に示すように、固定構造体内のフィールドの *occurs* 値が 1 より大きいときです。

```
Record myFixedRecordPart
  10 mySi CHAR(1)[3];
end
```

myRecord という固定レコードが、このパーツに基づいている場合、シンボル *myRecord.mySi* は、3 つの要素 (それぞれが 1 文字) からなる 1 次元配列を参照します。

構造体フィールド配列の使用法

以下のようなコンテキストでは、構造体フィールドの配列全体 (例えば、*myRecord.mySi*) を参照できます。

- *in* 演算子によって使用される第 2 オペランドとして。この演算子は、指定された値が配列に含まれているかどうかをテストします。
- 関数 **sysLib.size** のパラメーターとして。この関数は、構造体フィールドの *occurs* 値を返します。

それ自体が配列ではない配列要素は、他の要素と同様に 1 つのフィールドであり、さまざまな方法で参照できます。例えば、*assignment* ステートメントの中で参照したり、関数呼び出しの引数として参照したりできます。

要素添え字には、整数に変換される任意の数式を使用できますが、数式に関数呼び出しを含めることはできません。

1 次元の構造体フィールド配列

myRecord.mySi のような 1 次元配列の要素を参照するときは、配列の名前を指定し、その後ろに大括弧で囲った添え字を指定します。添え字には、整数か、整数へと解決されるフィールドを指定します。例えば、*myStruct.mySi[2]* のようにすると、例に示した配列の 2 番目の要素を参照できます。添え字は、1 から構造体フィールドの *occurs* 値の間で変化します。この範囲の外で添え字が指定されると、ランタイム・エラーが発生します。

フィールドを必要とするコンテキスト内で構造体フィールド配列の名前を使用し、大括弧で囲った添え字を指定しなかった場合、EGL では、VisualAge Generator との互換性モードであるときに限り、配列の最初の要素を参照していると見なされます。各要素を明示的に識別することをお勧めします。VisualAge Generator との互換性モードでない場合は、各要素を明示的に識別する必要があります。

次の例は、1 次元配列内の要素を参照する方法を示しています。これらの例では、*valueOne* が 1 に解決され、*valueTwo* が 2 に解決されます。

```
// 以下は、3 つのうち最初の要素を参照します
myRecord.mySi[valueOne]

// お勧めしません。これは
// VisualAge Generator との互換性が
// 有効な場合にのみ、有効です
myRecord.mySi

// 以下は、2 つ目の要素を参照します
myRecord.mySi[valueTwo]
```

次の例に示すとおり、1 次元配列は副構造を持つことができます。

```

record myRecord01Part
  10 name[3];
    20 firstOne CHAR(20);
    20 midOne CHAR(20);
    20 lastOne CHAR(20);
end

```

myRecord01 というレコードが前のパーツに基づいている場合、シンボル *myRecord01.name* は、それぞれが 60 文字の長さを持つ 3 つのエレメントの 1 次元配列を参照し、*myRecord01* の長さは 180 になります。

副構造を参照しなくても、*myRecord01.name* 内の各エレメントを参照することは可能です。例えば、*myRecord01.name[2]* は 2 番目のエレメントを参照します。また、エレメント内の副構造を参照することもできます。例えば、固有性の規則が満たされているなら、以下のいずれかの方法で、2 番目のエレメントの最後の 20 文字を参照できます。

```

myRecord01.name.lastOne[2]
myRecord01.lastOne[2]
lastOne[2]

```

最後の 2 つは、生成可能パーツ・プロパティー **allowUnqualifiedItemReferences** が *yes* に設定されている場合にのみ有効です。

各種の参照については、『変数および定数の参照』を参照してください。

多次元の構造体フィールド配列

1 より大きい *occurs* 値を持つ構造体項目に副構造があり、従属の構造体項目で *occurs* 値が 1 より大きい場合、その従属の構造体項目は、追加の次元を持つ配列を宣言します。

別のレコード・パーツについて考えて見ましょう。

```

record myRecord02Part
  10 siTop[3];
    20 siNext CHAR(20)[2];
end

```

myRecord02 というレコードがそのパーツに基づいている場合、1 次元配列 *myRecord02.siTop* の各エレメントは、それ自体が 1 次元配列になっています。例えば、3 つの従属 1 次元配列の 2 番目は *myRecord02.siTop[2]* として参照できます。構造体項目 *siNext* は 2 次元配列を宣言します。この配列のエレメントは、以下のいずれかの構文によって参照できます。

```

// 行 1、列 2 とします。
// 次の構文を強くお勧めします。
// なぜなら、動的配列に対しても機能するからです。
myRecord02.siTop[1].siNext[2]

// 次の構文は、VisualAge Generator との
// 互換性のためにサポートされています。
myRecord02.siTop.siNext[1,2]

```

参照されるメモリーの領域を明確にするため、多次元配列ではどのようにデータが保管されるのかを考えましょう。現在の例では、*myRecord02* は 120 バイトです。参照される領域は、それぞれが 40 バイトの長さを持つ、3 つのエレメントの 1 次元配列に分けられます。

```
siTop[1]      siTop[2]      siTop[3]
```

さらに、この 1 次元配列の各エレメントは、同じメモリーの領域の中で、それぞれが 20 バイトの長さを持つ 2 つのエレメントの配列に分けることができます。

```
siNext[1,1] siNext[1,2] siNext[2,1] siNext[2,2] siNext[3,1] siNext[3,2]
```

2 次元配列は、行が大きい順に保管されます。これには、配列をループで 2 重に初期設定するとき、1 つの行ごとに列を処理してゆくことによってパフォーマンスを高める意味があります。

```
// i、j、myTopOccurs、および myNextOccurs は、データ項目です。
// myRecord02 はレコードです。
// sysLib.size() は、構造体項目の occurs 値を戻します。
i = 1;
j = 1;
myTopOccurs = sysLib.size(myRecord02.siTop);
myNextOccurs = sysLib.size(myRecord02.siTop.siNext);
while (i <= myTopOccurs)
  while (j <= myNextOccurs)
    myRecord02.siTop.siNext[i,j] = "abc";
    j = j + 1;
  end
  i = i + 1;
end
```

多次元配列の各次元に応じた値を指定する必要があります。例えば、*myRecord02.siTop.siNext[1]* という参照は、2 次元配列では無効です。

3 次元配列の宣言の例を以下に示します。

```
record myRecord03Part
  10 siTop[3];
  20 siNext[2];
  30 siLast CHAR(20)[5];
end
```

myRecord03 というレコードがそのパーツに基づいており、固有性の規則が満たされている場合は、以下のいずれかの方法で、配列の最後のエレメントを参照できます。

```
// 推奨されているように、それぞれのレベルを示し、
// 各レベルごとに添え字を指定します。
myRecord03.siTop[3].siNext[2].siLast[5]

// それぞれのレベルを示し、下位レベルで添え字を指定する
myRecord03.siTop.siNext[3,2].siLast[5]
myRecord03.siTop.siNext[3][2].siLast[5]

// それぞれのレベルを示し、最下位レベルで添え字を指定する
myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]

// コンテナと最後のレベルを添え字付きで示す
myRecord03.siLast[3,2,5]
myRecord03.siLast[3,2][5]
myRecord03.siLast[3][2,5]
myRecord03.siLast[3][2][5]

// 最後のレベルのみを添え字付きで示す
```

```

siLast[3,2,5]
siLast[3,2][5]
siLast[3][2,5]
siLast[3][2][5]

```

前の例で示したように、多次元配列のエレメントは、さまざまな方法で大括弧に入れた一連の添え字を追加することによって参照します。いずれの場合でも、最初の添え字は第 1 の次元を参照し、2 番目の添え字は第 2 の次元を参照し、その後も同様です。各添え字の値は、1 から関連する構造体項目の `occurs` 値の間で変化します。添え字がこの範囲の外の数値に解決されると、ランタイム・エラーが発生します。

最初に、添え字が組み込まれなかった状態を考慮してください。

- 変数名を先頭に置いて、段階的に従属する構造体項目の名前をピリオドで区切ってリストしていったものを指定します。例えば、

```
myRecord03.siTop.siNext.siLast
```

- 変数名の直後にピリオドを付け、その後に、求める最下位レベル項目の名前を付けて指定できます。次に例を示します。

```
myRecord03.siLast
```

- 求める最下位レベル項目が所定の名前空間内で固有な場合は、次の例のように、その項目だけを指定できます。

```
siLast
```

次に、配列添え字を配置するための規則を考慮します。

- 複数のエレメントのいずれかが有効な場合は、次の例のように、それぞれのレベルごとに添え字を指定できます。

```
myRecord03.siTop[3].siNext[2].siLast[5]
```

- 複数のエレメントの 1 つが有効なレベルでは、次の例のように、一連の添え字を指定できます。

```
myRecord03.siTop.siNext[3,2].siLast[5]
```

- 複数のエレメントの 1 つが有効なレベル、またはそれに従属するレベルでは、次の例のように、一連の添え字を指定できます。

```
myRecord03.siTop.siNext.siLast[3,2,5]
```

- 所定のレベルで適切な添え字の数を超えて添え字を代入すると、エラーが発生します。次に例を示します。

```
// 無効
myRecord03.siTop[3,2,5].siNext.siLast
```

- 添え字を個々に大括弧に入れて分離するか、一連の添え字を表示するか、1 つずつコンマで区切ることができ、あるいは、2 つの使用法を結合してもかまいません。以下の例は有効です。

```

myRecord03.siTop.siNext.siLast[3,2,5]
myRecord03.siTop.siNext.siLast[3,2][5]
myRecord03.siTop.siNext.siLast[3][2,5]
myRecord03.siTop.siNext.siLast[3][2][5]

```

関連する概念

477 ページの『VisualAge Generator との互換性』

62 ページの『EGL での変数の参照』

関連する参照項目

605 ページの『add』
407 ページの『代入』
535 ページの『EGL システム制限』
631 ページの『get』
575 ページの『in 演算子』
973 ページの『size()』

辞書

辞書パーツ は、常時使用可能なパーツであり、定義を必要としません。辞書パーツに基づいた変数には一連のキーとそれらに関連した値を組み込むことができ、実行時に「キーおよび値」エントリーを追加したり除去したりできます。それらのエントリーは、レコード内のフィールドのように扱われます。

辞書宣言の例を次に示します。

```
row Dictionary
{
    ID          = 5,
    lastName    = "Twain",
    firstName    = "Mark",
};
```

宣言の中にエントリーを組み込むとき、各キー名は EGL 識別子であり、EGL の命名規則に従っている必要があります。実行時にエントリーを追加するときは、さらに柔軟性が増し、文字列リテラル、定数、または変数を指定できます。また、その場合、内容は EGL の予約語でもよく、識別子の中では無効となる文字を含めることもできます。詳細については、『動的アクセス用の大括弧構文 (*Bracket syntax for dynamic access*)』を参照してください。

代入の例を以下に示します。

```
row.age = 30;
row["Credit"] = 700;
row["Initial rating"] = 500
```

すでに存在するキーを代入しようとする、既存の「キーおよび値」エントリーがオーバーライドされます。次の代入は有効であり、「Twain」が「Clemens」に置き換えられます。

```
row.lastname = "Clemens";
```

代入をデータの取り出しに使用することもできます。

```
lastname String
age, credit, firstCredit int;

lastname = row.lastname;
age = row.age;
credit = row.credit;
credit = row["Credit"];
firstCredit = row["Initial rating"];
```

「キーおよび値」エントリーの値は、ANY 型です。したがって、さまざまな種類の情報を単一の辞書に入れることができます。それぞれの値は、以下のいずれかにすることができます。

- 前もって宣言されているレコードまたはその他の変数
- 定数またはリテラル

変数を辞書に入れると、その変数のコピーが代入されます。次のレコード・パーツを考えてみます。

```
Record myRecordPart
  x int;
end
```

次のコードは、myRecordPart タイプの変数を辞書に入れ、その後、元の変数内の値を変更します。

```
testValue int;

myRecord myRecordPart;

// 変数の値を設定し、
// その変数のコピーを辞書に入れる。
myRecord.x = 4;
row Dictionary
{
  theRecord myRecord;
}

// 元のレコードに新規の値を入れる。
myRecord.x = 700;

// 辞書のレコードのコピーにアクセスし、
// testValue に 4 を代入する。
testValue = row.theRecord.x;
```

1 つの辞書を別の辞書に代入すると、代入先の内容が代入元の内容で置き換えられ、代入先辞書のプロパティ値がオーバーライドされます。これについては、後で説明します。例えば、次のコード内の条件文は、真です。

```
row Dictionary { age = 30 };

newRow Dictionary { };
newRow = row

// 結果は真
if (newRow.age == 30)
;
end
```

宣言の中のプロパティ・セットは、その辞書が処理される方法に影響を及ぼします。辞書固有の一連の関数は、コードにデータとサービスを提供します。

辞書のプロパティ

次の例に示すように、それぞれの「プロパティおよび値」エントリーは、構文上、1 つの「キーおよび値」エントリーと等価であり、これらのエントリーは任意の順序で配置できます。

```
row Dictionary
{
  // プロパティ
  caseSensitive = no,
  ordering = none,

  // フィールド
  ID = 5,
```

```

        lastName = "Twain",
        firstName = "Mark"
        age = 30;
};

```

コードでプロパティやその値を追加したり、取り出したりすることはできません。ありそうにないことですが、プロパティ名をキーとして使用したいという場合は、キーを指定するか参照するときに、次の例のように変数名を修飾子として使用します。

```

row Dictionary
{
    // プロパティ
    caseSensitive = no,
    ordering = none,

    // フィールド
    row.caseSensitive = "yes"
    row.ordering = 50,
    age = 30
};

```

プロパティは、以下のとおりです。

caseSensitive

値の格納に使用された大/小文字が、キーまたは関連する値の取り出しに影響を及ぼすかどうかを示します。以下のオプションがあります。

No (デフォルト)

キー・アクセスは、キーの大/小文字に影響を受けず、以下の文は同じ意味になります。

```

age = row.age;
age = row.AGE;
age = row["aGe"];

```

Yes

EGL は本質的に大/小文字を区別しない言語ですが、以下の文は異なる結果になる可能性があります。

```

age = row.age;
age = row.AGE;
age = row["aGe"];

```

プロパティ **caseSensitive** の値は、この後のセクションで説明するいくつかの関数の振る舞いに影響を及ぼします。

ordering

「キーおよび値」エントリーが、取り出しの目的でどのように順序付けられるかを示します。このプロパティの値は、この後のセクションで述べるように、関数 **getKeys** と **getValues** の振る舞いに影響を及ぼします。

以下のオプションがあります。

None (デフォルト)

コードで、「キーおよび値」エントリーの順序を当てにすることはできません。

プロパティ **ordering** の値が **None** の場合、(関数 **getKeys** が呼び出されたときの) キーの順序は、(関数 **getValues** が呼び出されたときの) 値の順序と異なる場合があります。

ByInsertion

「キーおよび値」ペアを、それらが挿入された順序で入手できます。宣言内のエントリーは、左から右への順序で挿入されるものと見なされます。

ByKey

「キーおよび値」ペアを、キーの順序で入手できます。

辞書関数

以下の関数を呼び出すには、必ず、次の例のように関数名を辞書名で修飾してください。この例では、辞書名は `row` です。

```
if (row.containsKey(age))  
;  
end
```

containsKey()

`dictionaryName.containsKey(key String in)` returns (Boolean)

この関数は、入力ストリング (*key*) が辞書内のキーであるかどうかに応じて、真または偽に解決されます。辞書プロパティ `caseSensitive` が `no` に設定されている場合、大/小文字の区別は考慮されません。それ以外の場合、関数は大/小文字の一致も含め、完全に一致するものを探します。

`containsKey` は、論理式でのみ使用されます。

getKeys()

`dictionaryName.getKeys ()` returns (String[])

この関数は、ストリングの配列を返し、各ストリングは辞書内のキーです。

辞書プロパティ `caseSensitive` が `no` に設定されている場合、戻される各キーは小文字になります。それ以外の場合、戻される各キーは、そのキーが格納されたときと同じ大/小文字になります。

辞書プロパティ `ordering` が `no` に設定されている場合、戻されるキーの順序を当てにすることはできません。それ以外の場合、順序は、このプロパティの説明の中で指定されたとおりになります。

getValues()

`dictionaryName.getValues ()` returns (ANY[])

この関数は、任意型の値の配列を返します。それぞれの値は、辞書内のキーに関連付けられています。

insertAll()

`dictionaryName.insertAll(sourceDictionary Dictionary in)`

この機能は、一連の `assignment` ステートメントが、ソース辞書 (*sourceDictionary*) 内の「キーおよび値」エントリーを *target* (これは関数名を修飾している名前を持つ辞書です) にコピーするかのように動作します。

キーがソース内に存在してターゲット内に存在しない場合、その「キーおよび値」エントリーは、ターゲットへコピーされます。キーがソースとターゲットの両方に存在する場合、ソース・エントリーの値がターゲット内のエントリーをオーバーライドします。ターゲット内のキーがソース内のキーに一致するかどうかの判別は、各辞書の **caseSensitive** プロパティの値によって影響を受けます。

この関数は、1 つの辞書を別の辞書へ代入するわけではありません。なぜなら、関数 **insertAll** では、以下のエントリーが保存されるからです。

- ターゲット内の「プロパティおよび値」エントリー、および
- ターゲット内に存在し、ソース内に存在しない「キーおよび値」エントリー

removeElement()

`dictionaryName.removeElement(key String in)`

この関数は、入力ストリング (*key*) が辞書内のキーであるエントリーを除去します。辞書プロパティ **caseSensitive** が **no** に設定されている場合、大/小文字の区別は考慮されません。それ以外の場合、関数は大/小文字の一致も含め、完全に一致するものを探します。

removeAll()

`dictionaryName.removeAll()`

この関数は、辞書内のすべての「キーおよび値」エントリーを除去しますが、辞書のプロパティには影響を及ぼしません。

size()

`dictionaryName.size()` returns (INT)

辞書内の「キーおよび値」エントリーの数を示す整数を戻します。

関連する概念

19 ページの『パーツ』

62 ページの『EGL での変数の参照』

関連する参照項目

538 ページの『論理式』

ArrayDictionary

arrayDictionary パーツ は、常時使用可能なパーツであり、定義を必要としません。*arrayDictionary* パーツに基づいた変数を使用すると、一連の配列にアクセスし、すべての配列から同じ番号が付いたエレメントを取り出すことができます。この方法で取り出されたエレメント・セットは、それ自体が 1 つの辞書であり、それぞれのオリジナル配列名は、配列エレメントに格納された値と対になったキーとして扱われます。

arrayDictionary は、『コンソール・ユーザー・インターフェース』で説明されている表示テクノロジーとの関係で特に便利です。

次の図は、宣言に *ID*、*lastName*、*firstName*、および *age* という名前の配列を含んでいる `arrayDictionary` の例を示したものです。楕円で囲まれている辞書には、以下の「キーおよび値」エントリーが含まれています。

```
ID = 5,
lastName = "Twain",
firstName = "Mark",
age = 30
```

ID	LastName	FirstName	Age
5	Twain	Mark	30

この配列は辞書の配列であり、図では各辞書が横でなく上下に並べて示されています。しかし、`arrayDictionary` の宣言では配列の初期リストが必要となり、図では配列が横並びに示されています。

次のコードは、配列のリストの宣言を示しており、その後に、それらの配列を使用する `arrayDictionary` の宣言が続いています。

```
ID          INT[4];
lastName     STRING[4];
firstName    STRING[4];
age          INT[4];

myRows ArrayDictionary
{
    col1 = ID,
    col2 = lastName,
    col3 = firstName,
    col4 = age
};
```

値を取り出すために、コードでは、特定の辞書を分離してから、その辞書内にある特定のフィールド（「キーおよび値」エントリー）を分離する構文が使用されます。その `arrayDictionary` 構文を使用して、`arrayDictionary` 自体の値の更新や特性の変更を行うことはできません。

最初に、辞書を宣言し、次の例のように、その辞書へ `arrayDictionary` の行を代入します。

```
row Dictionary = myRows[2];
```

次に、適当な型の変数を宣言し、次のどちらかの例を使用して、その変数にエレメントを代入します。

```
cell INT = row["ID"];

cell INT = row.ID;
```

代わりに、次のどちらかの例のように、値を 1 ステップで取り出す構文もあります。

```
cell int = myRows[2]["ID"];

cell int = myRows[2].ID;
```

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

87 ページの『辞書』

62 ページの『EGL での変数の参照』

EGL ステートメント

各 EGL 関数は、以下の種類のゼロから複数の EGL ステートメントで構成されています。

- **変数宣言または定数宣言** は、メモリーの名前付き領域へのアクセスを可能にします。実行時に変数の値は変更できますが、定数の値は変更できません。どちらの種類の宣言も、ブロック以外であれば、関数内のどこにでも置くことができますが、これについては後述します。
- **関数呼び出し** は、次の例のように処理を関数に向けます。

```
myFunction(myInput);
```

再帰呼び出しも有効です。

- **assignment ステートメント** は、以下の値を変数にコピーできます。
 - 定数または変数からのデータ
 - リテラル
 - 関数呼び出しから戻された値
 - 算術計算の結果
 - スtring連結の結果

assignment ステートメントの例を以下に示します。

```
myItem = 15;
myItem = readFile(myKeyValue);
myItem = bigValue - 32;
record1.message = "Operation " + "successful!";
```

- **キーワード文** は、ファイル・アクセスなどの追加の機能を提供します。これらの各文は、文を開始するキーワードごとに名前が付けられています。例えば、次のとおりです。

```
add record1;    // an add statement
return (0);     // a return statement
```

- **null ステートメント** は、次の例に示すとおりセミコロンのことです。これは影響を与えませんが、プレースホルダーとして役立ちます。

```
if (myItem == 5)
;           // a null statement
else
  myFunction(myItem);
end
```

NULL 以外の EGL ステートメントには、以下のような特性があります。

- 文では、以下の種類の名前付きメモリー領域を参照できます。
 - 書式
 - ページ・ハンドラー
 - レコード
 - DataTable
 - 項目 (このカテゴリーには、データ項目に加えて、レコード、書式、および表に含まれる構造体項目も含まれます)
 - 配列 (1 より大きい occurs 値を持つ構造体項目に基づいたメモリー領域)
- 文に含めることが可能な式を以下に示します。
 - 日時式 は、日付、整数、間隔、またはタイム・スタンプへ解決されます。
 - 論理式 は、真または偽に解決されます。
 - 数式 は数値に解決されます。符号が付いたり、小数点以下の値を伴う場合があります。
 - スtring式 は、1 バイト文字、2 バイト文字、またはその両方を含む一連の文字に解決されます。
- セミコロンかブロック (1 つの単位として働く、0 個以上の一連の従属文) のいずれかで終わる文。ブロックを含む文は、以下の例が示しているように、end 区切り文字で終わります。

```
if (record2.status= "Y")
  record1.total = record1.total + 1;
  record1.message = "Operation successful!";
else
  record1.message = "Operation failed!";
end
```

終了区切り文字の後ろのセミコロンは、エラーではありませんが、null ステートメントとして扱われます。

文や EGL 全体では、名前の大文字小文字が区別されません。例えば、*record1* は *RECORD1* と等しく、*add* と *ADD* は同じキーワードを参照します。

注: Page Designer でソース・タブを使用した場合は、JSP ファイル (特に、JavaServer Faces ファイル) 内のコンポーネントをページ・ハンドラー内のデータ域に手動でバインドできます。EGL に大/小文字の区別はありませんが、JSP ファイル内で参照される EGL 変数名は EGL 変数宣言と大/小文字が同じである必要があります。完全に一致しなかった場合は JavaServer Faces エラーが発生します。EGL 変数を JSP フィールドにバインドした後は、EGL 変数の大/小文字を変更しないようにすることをお勧めします。

システム・ワード とは、特殊な機能を備えた一連のワードのことを言います。

- システム関数 は、コードを実行し、値を戻します。例えば、以下のものがあります。
 - **sysLib.minimum(arg1, arg2)** は、2 つの数値のうちの小さい方を戻します。
 - **strLib.strLen(arg1)** は、文字ストリングの長さを戻します。

プログラムに同じ名前の関数がある場合にのみ修飾子 (**mathLib**、**strLib** または **sysLib**) が必要です。

- システム変数 は、関数を呼び出さずに値を提供します。例えば、以下のものがあります。
 - **sysVar.errorCode** には、プログラムがファイルにアクセスして別の状態になった後の状況コードが含まれます。
 - **sysVar.sqlcode** には、プログラムがリレーショナル・データベースにアクセスした後の状況コードが含まれます。

修飾子 **sysVar** は、プログラムに同じ名前の変数がある場合にのみ必要です。

関数の 1 行に複数の文を含めることができます。しかし、1 行に含める文の数は 1 つにすることをお勧めします。EGL Debugger でブレークポイントを設定できるのが、その行の先頭の文のみになるからです。

『コメント』も参照してください。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

149 ページの『関数パーツ』

19 ページの『パーツ』

関連する参照項目

605 ページの『add』

407 ページの『代入』

608 ページの『call』

611 ページの『case』

613 ページの『close』

476 ページの『コメント』

511 ページの『データの初期化』

617 ページの『delete』

527 ページの『EGL 予約語』

619 ページの『execute』

560 ページの『関数呼び出し』

631 ページの『get』

644 ページの『get next』

650 ページの『get previous』

656 ページの『if, else』

96 ページの『キーワードのアルファベット順リスト』

538 ページの『論理式』

546 ページの『数式』

664 ページの『open』

679 ページの『prepare』

681 ページの『replace』

685 ページの『set』

547 ページの『テキスト式』

1004 ページの『terminalID』

699 ページの『while』

キーワードのアルファベット順リスト

キーワード	用途
605 ページの『add』	ファイル、メッセージ・キュー、またはデータベースの中にレコードを書き込みます。またはデータベースにレコード・セットを書き込みます。
608 ページの『call』	制御を別のプログラムに移動し、オプションで一連の値を渡します。呼び出し先プログラムが終了すると、制御は呼び出し側に戻ります。変数により渡されたデータが呼び出し先プログラムで変更された場合、呼び出し側が利用できるストレージ域も変更されます。
611 ページの『case』	文の複数のセットの始まりを示します。この複数のセットの内、実行されるセットは 1 つだけです。 case ステートメントは、各 case 文節の最後に break がある C または Java 言語の switch ステートメントと等価です。
613 ページの『close』	プリンターを切断したり、または指定のレコードに関連付けられたファイルやメッセージ・キューを閉じます。また、SQL レコードの場合は、EGL の open または get ステートメントによってオープンされたカーソルを閉じます。
615 ページの『continue』	テキスト・アプリケーションでテキスト書式を表示します。
616 ページの『converse』	テキスト・アプリケーションでテキスト書式を表示します。
617 ページの『delete』	ファイルからレコードを除去するか、データベースから行を除去します。
619 ページの『display』	テキスト書式をランタイム・バッファに追加しますが、画面にはデータを表示しません。
619 ページの『execute』	SQL ステートメント、特に SQL データ定義ステートメント (CREATE TABLE 型など) やデータ操作ステートメント (INSERT 型、UPDATE 型など) を 1 つ以上作成できます。
624 ページの『exit』	指定したブロックを出ます。このブロックは、デフォルトでは、 exit ステートメントを直接含むブロックです。
626 ページの『for』	ループ内でテストによって真と評価された回数だけ実行される文ブロックを開始します。
627 ページの『forEach』	ループで実行される一連の文の始まりを示します。最初の反復が発生するのは、指定された結果セットが使用可能であり、(多くの場合) その結果セット内の最後の行が処理されるまでの間、継続している場合だけです。
629 ページの『forward』	可変情報を含む Web ページを表示します。この文はページ・ハンドラーから呼び出されます。
631 ページの『freeSQL』	動的に準備された SQL ステートメントに関連したリソースを解放し、その SQL ステートメントに関連したすべてのオープン・カーソルをクローズする。
631 ページの『get』	単一ファイル・レコードまたはデータベース行を検索し、コード内で後で格納されたデータを置換または削除することができるオプションを提供します。また、このステートメントを使用してデータベース行の集合を検索し、それぞれの後続の行を動的配列の次の SQL レコードに入れることができます。 get ステートメントは、 get by key value として識別されることがあり、 get next のような get by position ステートメントとは区別されます。

キーワード	用途
638 ページの『get absolute』	open ステートメントによって選択されたりレシヨナル・データベースの結果セット内の、番号で指定された行を読み取ります。
640 ページの『get current』	open ステートメントによって選択されたデータベース結果セット内の、現在すでにカーソルが位置付けられている行を読み取ります。
641 ページの『get first』	open ステートメントによって選択されたデータベース結果セット内の最初の行を読み取ります。
643 ページの『get last』	open ステートメントによって選択されたデータベース結果セット内の最後の行を読み取ります。
644 ページの『get next』	ファイルまたはメッセージ・キューから次のレコードを読み取るか、データベース結果セットから次の行を読み取ります。
650 ページの『get previous』	指定の EGL 索引付きレコードに関連付けられたファイル内の直前のレコードを読み取るか、 open ステートメントによって選択されたデータベース結果セット内の直前の行を読み取ります。
654 ページの『get relative』	open ステートメントによって選択されたデータベース結果セット内の、番号で指定された行を読み取ります。この行は、結果セット内のカーソル位置との相対関係で識別されます。
656 ページの『goTo』	指定されたラベルから処理を続行します。このラベルは、この文と同じ関数内にあり、かつブロック外にある必要があります。
656 ページの『if, else』	論理式が真に解決される場合にのみ実行される一連の文 (ある場合) の始まりを示します。オプションのキーワード else は、論理式が偽に解決される場合にのみ実行される一連の代替文 (ある場合) の始まりを示します。予約語 end は、 if ステートメントの終わりを示します。
657 ページの『move』	データをバイトごと、または名前順にコピーします。後者の操作では、ある構造体内の指定の項目から、別の構造体内の同じ名前の項目にデータをコピーします。
664 ページの『open』	後で get next のような get by position ステートメントを使用して行のセットを取り出すために、リレシヨナル・データベースからその行セットを選択します。 open ステートメントは、カーソルまたは呼び出されたプロシージャ上で作動できます。
679 ページの『prepare』	SQL PREPARE ステートメントを指定します。オプションで、実行時にのみ既知になる詳細が含まれます。EGL execute ステートメントを実行するか、または、SQL ステートメントが結果セットを 戻す場合に、EGL open または get ステートメントを実行することで、準備済み SQL ステートメントが実行されます。
681 ページの『print』	印刷書式をランタイム・バッファに追加します。
681 ページの『replace』	変更したレコードをファイルまたはデータベースに書き込みます。
684 ページの『return』	関数を終了し、オプションで呼び出し側に値を戻します。
685 ページの『set』	レコード、テキスト書式、および項目にさまざまな影響を与えます。

キーワード	用途
696 ページの『show』	<code>display</code> ステートメントを使用してバッファに入れられた他の書式と一緒にテキスト書式をメインプログラムから表示します。また、現行プログラムを終了し、オプションでユーザーからの入力データおよび現行プログラムからの状態データを、ユーザーからの入力を処理するプログラムに転送します。
697 ページの『transfer』	メインプログラムから別のプログラムに制御を与えて、転送側のプログラムを終了します。さらにオプションで、受け取り側プログラムの入力レコードが受け取るデータを持っているレコードを渡します。呼び出し先プログラムでは、 transfer ステートメントは使用することができません。
698 ページの『try』	入出力 (I/O) ステートメント、システム関数呼び出し、または call ステートメントがエラーになり、 try ステートメント内にある場合にプログラムの実行が継続されていることを示します。例外が発生すると、処理は、 onException ブロック (ある場合) の最初の文または try ステートメントの直後の文で再開されます。ただし、入出力ハード・エラーは、システム変数 VGVar.handleHardIOErrors が 1 に設定されている場合にのみ処理されます。その他の場合、プログラムは、メッセージを表示 (可能な場合) して終了します。
699 ページの『while』	ループで実行される一連の文の始まりを示します。1 回目のステートメントが実行されるのは、論理式が真になる場合のみであり、2 回目以降が繰り返されるかどうかと同じ条件に左右されます。予約語 end は、 while ステートメントの終わりを示します。

関連する参照項目

93 ページの『EGL ステートメント』

プログラム間での制御の移動

EGL には、以下のように、制御をプログラムから別のプログラムに切り替える方法がいくつかあります。

- **call** ステートメントは、別のプログラムに制御を与えます。オプションにより、一連の値を渡すこともできます。呼び出し先プログラムが終了すると、制御は呼び出し側に戻ります。呼び出し先プログラムが、変数として渡された任意のデータを変更すると、その変数の内容は、呼び出し側でも変更されます。

呼び出しは、データベースやその他のリカバリー可能リソースをコミットしませんが、サーバー・サイドの自動コミットは発生する場合があります。

リンケージ・オプション・パーツの `callLink` エlementを設定することで、その呼び出しの特性を指定することができます。詳細については、『*call*』および『*callLink* エlement』を参照してください。サーバー・サイドの自動コミットの詳細については、『*callLink* エlementの *luwControl*』を参照してください。

- **transfer** ステートメントは、制御をメインプログラムから別のメインプログラムに移動し、移動元のプログラムを終了し、オプションで、受け取り先プログラムの入力レコードによって受領されるデータを持つレコードを渡します。呼び出し先プログラムでは、**transfer** ステートメントは使用することができません。

プログラムは、トランザクションへの移動またはプログラムへの移動という書式の文によって、制御を移動することができます。

- トランザクションへの転送は、以下のようにして行われます。
 - Java メイン・テキストまたはメイン・バッチ・プログラムとして実行されるプログラムでは、その振る舞いは、ビルド記述子オプション `synchOnTrxTransfer` の設定によって、以下のように異なります。
 - `synchOnTrxTransfer` の値が YES の場合、`transfer` ステートメントは回復可能リソースをコミットし、ファイルをクローズし、カーソルをクローズし、同じ実行単位内でプログラムを開始します。
 - `synchOnTrxTransfer` の値が NO (デフォルト) の場合も、`transfer` ステートメントは同じ実行単位内でプログラムを開始しますが、呼び出されたプログラムが使用できるリソースのクローズまたはコミットは行いません。
 - プログラムへの移動では、リカバリー可能リソースのコミットまたはロールバックは行いませんが、ファイルをクローズし、ロックを解除し、同じ実行単位内でプログラムを開始します。

リンケージ・オプション・パーツは、どの種類の移動の特性にも影響しません。

ページ・ハンドラー内では、移動は無効です。

詳細については、『*transfer*』を参照してください。

- システム関数 `sysLib.startTransaction` は、非同期的に実行単位を開始します。その操作は、移動元のプログラムを終了せず、移動元のプログラム内のデータベース、ファイル、およびロックに影響しません。受け取り側のプログラム内の領域である入力コードに、データを渡すオプションを選択できます。

プログラムが `sysLib.startTransaction` を呼び出す場合は、リンケージ・オプション・パーツ `asynchLink` エレメントを持つプログラムを生成する必要があります。詳細については、『*sysLib.startTransaction*』および『*asynchLink* エレメント』を参照してください。

- EGL `show` ステートメントは、テキスト・アプリケーション内の現行メインプログラムを終了し、書式に従ってユーザーにデータを表示します。ユーザーが書式を処理依頼すると、`show` ステートメントはオプションで制御権を 2 番目のメインプログラムに移動します。そのメインプログラムは、ユーザーから受け取ったデータと、元のプログラムから変更なしで渡されたデータを受信します。

`show` ステートメントは、リンケージ・オプション・パーツ `transferLink` エレメントの設定に影響されます。

詳細については、『*show*』を参照してください。

- 最後に、`forward` ステートメントがページ・ハンドラーまたはプログラムから呼び出されます。この文の動作は、以下のとおりです。
 1. 回復可能リソースのコミット、ファイルのクローズ、ロックの解除
 2. 制御権の移動
 3. コードの終了

この場合の宛先は、別のプログラムまたは Web ページです。詳細については、『*forward*』を参照してください。

関連する参照項目

410 ページの『asynchLink エlement』
608 ページの『call』
442 ページの『callLink エlement』
629 ページの『forward』
450 ページの『callLink エlementの luwControl』
696 ページの『show』
975 ページの『startTransaction()』
697 ページの『transfer』

例外処理

EGL 生成のプログラムが次のような動作を行うと、エラーが発生することがあります。

- ファイル、キュー、またはデータベースにアクセスする
- 別のプログラムを呼び出す
- 関数を呼び出す
- 代入、比較、または計算を実行する

try ブロック

EGL *try* ブロック は、**try** と **end** の区切り文字に挟まれた多くの EGL ステートメントに対するゼロの系列です。以下に例を示します。

```
if (userRequest = "A")
  try
    add record1;
  onException
    myErrorHandler(12);
  end
end
```

一般に、try ブロックによって、エラーが生じた場合でもプログラムで処理を継続することができます。

try ブロックには、以前に示した *onException* 文節 が含まれていることがあります。この文節は、try ブロック内の以前の文のいずれかが失敗した場合に呼び出されますが、onException 文節がない場合は、try ブロック内のエラーによって、その try ブロックの直後の最初の文が呼び出されます。

EGL システム例外

EGL は、実行時の問題について特定の性質を示すために、一連のシステム例外を備えています。これらの例外は、それぞれが 1 つの辞書になっており、そこから情報を取り出すことができますが、取り出しには、常にシステム変数

SysLib.currentException (これも辞書) を使用します。この変数により、実行単位内でスローされた最新の例外にアクセスできます。

どの例外にも、**code** という 1 つのフィールドがあり、これは、その例外を識別する文字列です。このフィールドを次のようにロジック内でテストすることにより、現行の例外を判別できます。


```

if (userRequest = "A")
  try
    add record1;
  onException
    case (SysLib.currentException.code)
      when (FileIOException)
        myErrorHandler(12);
      otherwise
        myErrorHandler(15);
    end
  end
end
end

```

この場合、FileIOException は「com.ibm.egl.FileIOException」という文字列値に相当する定数です。EGL 例外定数は常に、「com.ibm.egl」で始まる文字列内の最後の修飾子と等価です。

例外フィールドへのアクセスは、onException ブロック内でのみ行うことを強くお勧めします。例外が発生しなかったときに、コードが **SysLib.currentException** にアクセスすると、実行単位は終了します。

次の例は、例外 SQLException 内の sqlcode フィールドにアクセスします。

```

if (userRequest = "A")
  try
    add record01;
  onException
    case (SysLib.currentException.code)
      when ("com.ibm.egl.SQLException")
        if (SysLib.currentException.sqlcode == -270)
          myErrorHandler(16);
        else
          myErrorHandler(20);
        end
      otherwise
        myErrorHandler(15);
    end
  end
end
end

```

システム例外の詳細については、『EGL システム例外』を参照してください。

try ブロックの制限

以前の try ブロックについての詳細は、限定される必要があります。まず、try ブロックは、以下の種類の EGL ステートメントでのエラー処理にのみ影響します。

- I/O ステートメント
- システム関数
- call ステートメント

try ブロックの存在によって、数値オーバーフローの処理が影響を受けることはありません。この種のエラーの詳細については、『VGVar.handleOverflow』を参照してください。

第 2 に、try ブロックは、try ブロック内部から呼び出されたユーザー関数 (またはプログラム) 内のエラーに影響を与えません。次の例では、文が関数 myABC で失敗すると、関数 myABC が自らエラーを処理する場合を除いて、プログラムはエラー・メッセージとともに即時に終了します。


```

if (userRequest = "B")
  try
    myVariable = myABC();
  onException
    myErrorHandler(12);
  end
end

```

第 3 に、プログラムは即時に終了し、以下のような場合にエラー・メッセージが表示されます。

- try ブロックによって明確にカバーされた種類のエラーが、try ブロックの外部で生じる
- 以下の場合のいずれかが当てはまる場合 (try ブロック内も含む)
 - ユーザー作成関数の呼び出しまたは戻り時に、障害が生じる
 - 数値変数に非数値文字が割り当てられている
 - ファイル I/O ステートメントがハード・エラー (後に説明) で終了したときに、システム変数 **VGVar.handleHardIOErrors** に 1 でなく 0 が設定されている

以下の場合も該当します。

- 値がゼロで割られると、Java プログラムではこの状況が数値オーバーフローとして処理されます。
- 数値変数に非数値文字が代入されると、Java プログラムは終了します。

注: VisualAge Generator および EGL 5.0 で作成されたプログラムのマイグレーションをサポートするために、変数 **VGVar.handleSysLibraryErrors** (旧名 **ezereply**) は、try ブロックの外側で生じるいくつかのエラーを処理することができます。この変数は使用しないでください。この変数が機能するのは VisualAge Generator との互換性モードの場合のみです。

エラー関連システム変数

EGL は、正常なイベントの応答または終了しないエラーの応答として、try ブロックに設定されたエラー関連システム変数を提供します。これらの変数の値は、try ブロック内およびその try ブロックに後続して実行されるコード内で使用でき、大抵の場合、値は変換後にリストアされます。

EGL ランタイムは、文が try ブロックの外部で実行されるときに、エラー関連変数の値を変更しません。ただし、ユーザーのプログラムは、try ブロック外部のエラー関連変数に値を割り当てることができます。

システム変数 **sysVar.exceptionCode** には、さまざまな状態の値が与えられます。そして、それらのすべての状態において、ランタイム環境とのプログラムの相互作用の性質に応じて、1 つ以上の追加の変数も設定されます。

- システム変数 **sysVar.exceptionCode** および **sysVar.errorCode** には、両方ともに、以下の種類の任意の文を try ブロックで実行した後に、値が与えられます。
 - **call** ステートメント
 - 索引ファイル、MQ ファイル、相対ファイル、またはシリアル・ファイルに対する I/O ステートメント
 - ほとんどすべてのシステム関数の呼び出し

- システム変数 **sysVar.exceptionCode**、**sysVar.errorCode**、**VGVar.mqConditionCode**、および **sysVar.mqReasonCode** のすべてには、try ブロック内の I/O ステートメントが MQ レコード上で作動した後に値が与えられます。
- システム変数 **sysVar.exceptionCode** には、try ブロック内の文からリレーショナル・データベースにアクセスした後に、値が与えられます。SQL 通信域 (SQLCA) の 変数にも値が割り当てられます (詳細については *sysVar.sqlca* を参照)。

try ブロック内で無限のエラーが生じた場合、**sysVar.exceptionCode** の値は、エラーが try ブロックの外部で生じた場合にユーザーに提示される EGL エラー・メッセージの数値コンポーネントと同一の値です。ただし、**sysVar.errorCode** や **VGVar.mqConditionCode** など、状態固有の変数の値は、ランタイム・システムによって提供されます。エラーがない場合、**sysVar.exceptionCode** の値と、少なくとも 1 つの状態特定変数とは、同じ 8 つのゼロのストリングです。

エラー・コードは、『*VGVar.handleOverflow*』の説明にあるように、無限の数値オーバーフローの場合、**sysVar.exceptionCode** および **sysVar.errorCode** に割り当てられます。しかし、正常な算術計算がエラー関連システム変数に影響を与えることはありません。

また、エラー関連システム変数は、システム関数以外の関数の呼び出しによって影響を受けません。そして、**sysVar.errorCode** (大抵のシステム関数によって影響を受ける変数) は、以下の変数でのエラーによって影響を受けません。

- **sysLib.calculateChkDigitMod10**
- **sysLib.calculateChkDigitMod11**
- **strLib.concatenate**
- **strLib.concatenateWithSeparator**
- **VGLib.connectionService**
- **sysLib.connect**
- **sysLib.convert**
- **sysLib.disconnect**
- **sysLib.disconnectAll**
- **sysLib.purge**
- **sysLib.queryCurrentDatabase**
- **strLib.setBlankTerminator**
- **sysLib.setCurrentDatabase**
- **strLib.strLen**
- **sysLib.verifyChkDigitMod10**
- **sysLib.verifyChkDigitMod11**
- **sysLib.wait**

エラー値が **sysVar.exceptionCode** に割り当てられた場合、システム変数 **sysVar.exceptionMsg** には関連した EGL エラー・メッセージのテキストが割り当てられ、システム変数 **sysVar.exceptionMsgCount** にはエラー・メッセージ内のバイト数 (末尾ブランクおよび NULL を除く) が割り当てられます。8 つのゼロのストリングが **sysVar.exceptionCode** に割り当てられた場合、**sysVar.exceptionMsg** はブランクが割り当てられ、**sysVar.exceptionMsgCount** はゼロに設定されます。

I/O ステートメント

I/O ステートメントに関して、エラーはハードかソフトのいずれかです。

- ソフト・エラーは、以下のいずれかです。
 - SQL データベース表または に対する入出力 (I/O) 操作時にレコードが見付からない
 - 索引付きファイル、相対ファイル、またはシリアル・ファイルの入出力 (I/O) 操作で、 以下のいずれかの問題が発生する
 - 重複レコード (外部データ・ストアが重複の追加を許可する場合)
 - レコードが見付からない
 - ファイル終わり
- ハード・エラー。その他の問題であり、例えば次のようなエラーです。
 - 重複レコード (外部データ・ストアが重複の追加を禁止する場合)
 - ファイルが見つからない
 - データ・セットのリモート・アクセス時に通信リンクが使用できない

ソフト・エラーの原因となった文が `try` ブロック内にある場合は、以下のことが当てはまります。

- デフォルトでは、EGL は `onException` ブロックに制御を渡さず、実行を続行します。
- `onException` ブロックに制御を渡したい場合は、プログラム、ページ・ハンドラー、またはライブラリーの中で、**`throwNrfEofExceptions`** プロパティを `yes` に設定します。

ハード入出力エラーが `try` ブロックで生じた場合は、結果は、エラー関連システム変数の値によって決まります。

- ファイル、リレーショナル・データベース、または `MQSeries` メッセージ・キューへのアクセス時には、以下の規則が適用されます。
 - **`VGVar.handleHardIOErrors`** が 1 に設定されていれば、プログラムは実行を続行する
 - **`VGVar.handleHardIOErrors`** が 0 に設定されていれば、プログラムは (可能な場合) エラー・メッセージを表示して終了する

この変数のデフォルト設定は、**`handleHardIOErrors`** プロパティの値によって異なり、このプロパティは、プログラム、ライブラリー、ページ・ハンドラーなどの生成可能ロジック・パーツの中で使用可能です。このプロパティのデフォルト値は `yes` で、これは、変数 **`VGVar.handleHardIOErrors`** の初期値を 1 に設定します。

ハードまたはソフトのいずれかの入出力エラーが `try` ブロック外部で生じた場合、生成したプログラムは可能な場合はエラー・メッセージを提示してから、終了します。

DB2 に直接アクセスしている場合 (JDBC 経由でなく)、ハード・エラーの `sqlcode` は 304、802、または 0 より小さい値です。

エラーの識別

try ブロックの内部または外部に **case** または **if** ステートメントを組み込むことによって、try ブロック内で生じたエラーの種類を判別できます。その文内では、さまざまなシステム変数の値をテストすることができます。ただし、入出力エラーに回答している場合、および文で EGL レコードを使用している場合は、基本論理式の使用が推奨されます。式の形式には、次の 2 種類があります。

recordName is IOErrorValue

recordName not IOErrorValue

recordName

入出力操作で使用されたレコードの名前。

IOErrorValue

データベース管理システム全体で一定の入出力エラー値の 1 つ。

入出力エラー値が指定された論理式を使用しないでデータベース管理システムを変更する場合は、プログラムの変更と再生成が必要になることがあります。特に、**sysVar.sqlcode** または **sysVar.sqlState** 内の同等の値ではなく、入出力エラー値を使用して、エラーの有無をテストすることをお勧めします。、それらの値は、基盤となるデータベース実装によって異なります。

関連する概念

477 ページの『VisualAge Generator との互換性』

87 ページの『辞書』

関連する参照項目

1027 ページの『EGL Java ランタイム・エラー・コード』

93 ページの『EGL ステートメント』

579 ページの『入出力エラー値』

538 ページの『論理式』

996 ページの『errorCode』

997 ページの『overflowIndicator』

1000 ページの『sqlca』

1001 ページの『sqlcode』

1002 ページの『sqlState』

1013 ページの『handleSysLibraryErrors』

1012 ページの『handleHardIOErrors』

1012 ページの『handleOverflow』

1014 ページの『mqConditionCode』

EGL 6.0 iFix への EGL コードのマイグレーション

EGL V6.0 マイグレーション・ツールは、EGL V6.0 iFix に準拠するために、V5.1.2 および V6.0 から EGL ソースを変換します。このツールは、プロジェクト全体、単一ファイル、または選択された複数ファイルに対して使用できます。このツールをパッケージまたはフォルダーに対して実行すると、そのパッケージまたはフォルダー内のすべての EGL ソース・ファイルが変換されます。このマイグレーション・ツールによって変更されるコードに関する詳細については、『EGL から EGL へのマイグレーション』を参照してください。

注:すでに EGL V6.0 iFix に更新されているコードに対して、マイグレーション・ツールを使用しないでください。これを行うと、コードでエラーが発生する可能性があります。

EGL コードを EGL V6.0 iFix にマイグレーションするには、次のようにします。

1. ワークベンチで「ウィンドウ」>「設定」をクリックする。
2. 「設定」ウィンドウの左側で、「ワークベンチ」を展開し、「機能」をクリックする。
3. 機能のリストで、「EGL デベロッパー」を展開する。
4. 「EGL V6.0 Migration」という名前の機能のチェック・ボックスを選択する。
5. 「OK」をクリックする。
6. もう一度、「ウィンドウ」>「設定」をクリックする。
7. 「設定」ウィンドウの左側で、「EGL」を展開し、「EGL V6.0 マイグレーション設定」をクリックする。
8. EGL V6.0 マイグレーション・ツールの設定を変更する。このウィンドウの設定に関する詳細については、『EGL 間マイグレーション設定の変更』を参照してください。
9. 「プロジェクト・エクスプローラー」ビューまたはナビゲーター・ビューで、マイグレーションしたい EGL プロジェクト、パッケージ、フォルダー、またはファイルを選択する。マイグレーションする任意の数の EGL リソースを選択できます。一度に複数のリソースを選択するには、CTRL を押したままリソースをクリックしてください。
10. 選択されたリソースを右クリックし、ポップアップ・メニューから「EGL V6.0 マイグレーション (EGL V6.0 Migration)」>「マイグレーション」をクリックする。
11. EGL V6.0 iFix に準拠しない箇所がないかどうか、コードを調べる。

マイグレーション・ツールは、EGL V6.0 iFix に準拠するように、選択された EGL ソース・ファイルを変換します。ツールがソース・コードに加えた変更を検討するには、次のようにします。

1. 「プロジェクト・エクスプローラー」ビューまたはナビゲーター・ビューで、マイグレーションされた EGL ソース・ファイルを右クリックし、ポップアップ・メニューから「次と比較」>「ローカル・ヒストリー」をクリックする。

2. ワークスペース内のファイルと旧バージョンとの違いを調べる。
3. 変更の検討が終了したら、「OK」をクリックする。

関連する概念

『EGL から EGL へのマイグレーション』

117 ページの『EGL 間マイグレーション設定の変更』

関連するタスク

129 ページの『EGL 機能の使用可能化』

EGL から EGL へのマイグレーション

EGL V6.0 マイグレーション・ツールは、EGL の V5.1.2 と V6.0 のソースを EGL V6.0 iFix に準拠するように変換します。このツールは、プロジェクト全体、単一のファイル、または選択した複数のファイルに対して使用できます。このツールをパッケージまたはフォルダーに対して実行すると、そのパッケージまたはフォルダー内にあるすべての EGL ソース・ファイルが変換されます。このマイグレーション・ツールの使用方法については、『EGL 6.0 iFix への EGL コードのマイグレーション (Migrating EGL code to the EGL 6.0 iFix)』を参照してください。

マイグレーション・ツールでは、変更する各ファイルにコメントを追加でき、プロジェクトのログ・ファイルにもコメントを追加できます。これらのオプションを変更するには、『EGL から EGL へのマイグレーションの設定』を参照してください。

マイグレーション・ツールは、EGL コードを次のような方法で変更し、EGL V6.0 iFix に準拠させます。

- マイグレーション・ツールは、プロパティの指定方法に変更を加えます。プロパティの変更方法についての情報は、『EGL から EGL へのマイグレーション時におけるプロパティの変更』を参照してください。
- マイグレーション・ツールは、予約語と競合する変数およびパーツ名を検索します。マイグレーション・ツールは、それらの変数およびパーツ名を変更するために、『EGL から EGL へのマイグレーションの設定』で定義されている接頭部または接尾部を追加します。デフォルトでは、このツールが、現時点で予約語になっているすべての名前に `_EGL` という接尾部を追加します。マイグレーション・ツールは、CALL ステートメントのオブジェクトの名前変更は行わず、EGL ビルド・パーツ・ファイル内の参照を更新しません。『EGL 予約語』を参照してください。マイグレーション・ツールを使用する前と後のコードの例を次に示します。

マイグレーション前:

```
Library Handler
  boolean Bin(4);
End
```

マイグレーション後:

```
Library Handler_EGL
  boolean_EGL Bin(4);
End
```


- マイグレーション・ツールは、比較演算子として使用されている単一の等号 (=) を二重の等号 (==) に置き換えます。代入演算子として使用されている単一の等号は変更されません。

マイグレーション前:

```
Function test(param int)
  a int;
  If(param = 3)
    a = 0;
  End
End
```

マイグレーション後:

```
Function test(param int)
  a int;
  If(param == 3)
    a = 0;
  End
End
```

- マイグレーション・ツールは、レベル番号のないレコードにレベル番号を追加します。

マイグレーション前:

```
Record MyRecord
  item1 int;
  item2 int;
End
```

マイグレーション後:

```
Record MyRecord
  10 item1 int;
  10 item2 int;
End
```

- マイグレーション・ツールは、定数の宣言の構文を変更します。

マイグレーション前:

```
intConst 3;
```

マイグレーション後:

```
const intConst int = 3;
```

- マイグレーション・ツールは、別のライブラリーへ移動するか名前が変更された変数および関数名を変更します。この変更により、SysLib および SysVar ライブラリーに属する変数と関数が影響を受けます。

マイグレーション前:

```
SysLib.java();
clearRequestAttr();
```

マイグレーション後:

```
JavaLib.invoke();
J2EELib.clearRequestAttr();
```

SysLib および SysVar ライブラリーの変更される変数と関数名のリストを以下に示します。

表 1. SysLib および SysVar ライブラリーの変更される変数と関数名

マイグレーション前	マイグレーション後
SysLib.dateValue	DateTimeLib.dateValue
SysLib.extendTimestampValue	DateTimeLib.extend
SysLib.formatDate	StrLib.formatDate
SysLib.formatTime	StrLib.formatTime
SysLib.formatTimestamp	StrLib.formatTimestamp
SysLib.intervalValue	DateTimeLib.intervalValue
SysLib.timeValue	DateTimeLib.timeValue
SysLib.timeStampValue	DateTimeLib.timestampValue
SysLib.java	JavaLib.invoke
SysLib.javaGetField	JavaLib.getField
SysLib.javaIsNull	JavaLib.isNull
SysLib.javaIsObjID	JavaLib.isObjID
SysLib.javaRemove	JavaLib.remove
SysLib.javaRemoveAll	JavaLib.removeAll
SysLib.javaSetField	JavaLib.setField
SysLib.javaStore	JavaLib.store
SysLib.javaStoreCopy	JavaLib.storeCopy
SysLib.javaStoreField	JavaLib.storeField
SysLib.javaStoreNew	JavaLib.storeNew
SysLib.javaType	JavaLib.qualifiedTypeName
SysLib.clearRequestAttr	J2EELib.clearRequestAttr
SysLib.clearSessionAttr	J2EELib.clearSessionAttr
SysLib.getRequestAttr	J2EELib.getRequestAttr
SysLib.getSessionAttr	J2EELib.getSessionAttr
SysLib.setRequestAttr	J2EELib.setRequestAttr
SysLib.setSessionAttr	J2EELib.setSessionAttr
SysLib.displayMsgNum	ConverseLib.displayMsgNum
SysLib.clearScreen	ConverseLib.clearScreen
SysLib.fieldInputLength	ConverseLib.fieldInputLength
SysLib.pageEject	ConverseLib.pageEject
SysLib.validationFailed	ConverseLib.validationFailed
SysLib.getVAGSysType	VGLib.getVAGSysType
SysLib.connectionService	VGLib.connectionService
SysVar.systemGregorianCalendarFormat	VGVar.systemGregorianCalendarFormat
SysVar.systemJulianDateFormat	VGVar.systemJulianDateFormat
SysVar.currentDate	VGVar.currentGregorianCalendar
SysVar.currentFormattedDate	VGVar.currentFormattedGregorianCalendar
SysVar.currentFormattedJulianDate	VGVar.currentFormattedJulianDate
SysVar.currentFormattedTime	VGVar.currentFormattedTime
SysVar.currentJulianDate	VGVar.currentJulianDate

表 1. SysLib および SysVar ライブラリーの変更される変数と関数名 (続き)

マイグレーション前	マイグレーション後
SysVar.currentShortDate	VGVar.currentShortGregorianDate
SysVar.currentShortJulianDate	VGVar.currentShortJulianDate
SysVar.currentTime	DateTimeLib.currentTime
SysVar.currentTimeStamp	DateTimeLib.currentTimeStamp
SysVar.handleHardIOErrors	VGVar.handleHardIOErrors
SysVar.handleSysLibErrors	VGVar.handleSysLibraryErrors
SysVar.handleOverflow	VGVar.handleOverflow
SysVar.mqConditionCode	VGVar.mqConditionCode
SysVar.sqlerrd	VGVar.sqlerrd
SysVar.sqlerrmc	VGVar.sqlerrmc
SysVar.sqlIsolationLevel	VGVar.sqlIsolationLevel
SysVar.sqlWarn	VGVar.sqlWarn
SysVar.commitOnConverse	ConverseVar.commitOnConverse
SysVar.eventKey	ConverseVar.eventKey
SysVar.printerAssociation	ConverseVar.printerAssociation
SysVar.segmentedMode	ConverseVar.segmentedMode
SysVar.validationMsgNum	ConverseVar.validationMsgNum

- ・ マイグレーション・ツールは、日付、時刻、およびタイム・スタンプの指定方法を変更します。以下に、いくつかの例を示します。

表 2. 日付、時刻、およびタイム・スタンプの変更

マイグレーション前	マイグレーション後
dateFormat = "yy/mm/dd"	dateFormat = "yy/MM/dd"
dateFormat = "YYYY/MM/DD"	dateFormat = "yyyy/MM/dd"
dateFormat = "YYYY/DDDD"	dateFormat = "yyyy/DDDD"
timeFormat = "hh:mm:ss"	timeFormat = "HH:mm:ss"

- ・ マイグレーション・ツールは、マイグレーションされたライブラリー、プログラム、およびページ・ハンドラーのプロパティー HandleHardIOErrors を、(このプロパティーが指定されていなければ) すべて no に設定します。

関連するタスク

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

関連する概念

117 ページの『EGL 間マイグレーション設定の変更』

112 ページの『EGL からEGL へのマイグレーション時におけるプロパティーの変更』

関連する参照項目

527 ページの『EGL 予約語』

EGL から EGL へのマイグレーション時におけるプロパティの変更

マイグレーション・ツールは、プロパティの指定方法に大幅な変更を加えます。以下は、それらの変更の要約です。

- マイグレーション・ツールは、EGL V6.0 iFix で名前が変更されたプロパティを名前変更します。名前変更されるプロパティのリストを以下に示します。

表 3. 名前変更されるプロパティ

マイグレーション前	マイグレーション後
action	actionFunction
boolean	isBoolean
getOptions	getOptionsRecord
msgDescriptor	msgDescriptorRecord
onPageLoad	onPageLoadFunction
openOptions	openOptionsRecord
putOptions	putOptionsRecord
queueDescriptor	queueDescriptorRecord
range	validValues
rangeMsgKey	validValuesMsgKey
selectFromList	selectFromListItem
sqlVar	sqlVariableLen
validator	validatorFunction
validatorMsgKey	validatorFunctionMsgKey
validatorTable	validatorDataTable
validatorTableMsgKey	validatorDataTableMsgKey

- マイグレーション・ツールは、文字列リテラルとして使用されているプロパティ一値に二重引用符を追加します。

マイグレーション前:

```
{ alias = prog }
```

マイグレーション後:

```
{ alias = "prog" }
```

以下のプロパティが影響を受けます。

- alias
- column
- currency
- displayName
- fileName
- fillCharacter
- help
- helpKey
- inputRequiredMsgKey

- minimumInputMsgKey
- msgResource
- msgTablePrefix
- pattern
- queueName
- rangeMsgKey
- tableNames
- title
- typeChkMsgKey
- validatorMsgKey
- validatorTableMsgKey
- value
- view
- マイグレーション・ツールは、プロパティの値として配列リテラルを指定するときに、小括弧を大括弧に変更します。
 - formSize
 - keyItems
 - outline
 - pageSize
 - position
 - range
 - screenSize
 - screenSizes
 - tableNames
 - tableNameVariables
 - validationBypassFunctions
 - validationBypassKeys
- 配列リテラルをとるプロパティの場合、マイグレーション・ツールは単一エレメント配列リテラルを大括弧で囲み、エレメントが 1 つだけの配列が、依然として配列であることを指定します。マイグレーション・ツールは、配列の配列をとるプロパティに、二重の大括弧セットを使用します。

マイグレーション前:

```
{ keyItems = var, screenSizes = (24, 80), range = (1, 9) }
```

マイグレーション後:

```
{ keyItems = ["var"], screenSizes = [[24, 80]], range = [[1, 9]] }
```

- マイグレーション・ツールは、配列内の特定エレメントのプロパティをオーバーライドするときに、変数名の代わりにキーワード **this** を使用します。

マイグレーション前:

```
Form myForm type TextForm
  fieldArray char(10)[5] { fieldArray[1] {color = red } };
end
```

マイグレーション後:

```
Form myForm type TextForm
  fieldArray char(10)[5] { this[1] {color = red } };
end
```

- マイグレーション・ツールは、必要であればパーツ、関数、およびフィールドへの参照を変更し、引用符と大括弧を追加します。

マイグレーション前:

```
{ keyItems = (item1, item2) }
```

マイグレーション後:

```
{ keyItems = ["item1", "item2"] }
```

この方法でマイグレーション・ツールの影響を受けるプロパティは、以下のとおりです。

- action
- commandValueItem
- getOptions
- helpForm
- inputForm
- inputPageRecord
- inputRecord
- keyItem
- keyItems
- lengthItem
- msgDescriptorRecord
- msgField
- numElementsItem
- onPageLoadFunction
- openOptionsRecord
- putOptionsRecord
- queueDescriptorRecord
- redefines
- selectFromListItem
- tableNameVariables
- validationBypassFunctions
- validatorFunction
- validatorDataTable
- マイグレーション・ツールは、指定されていて値が代入されていない boolean プロパティにデフォルト値の yes を代入します。

マイグレーション前:

```
{ isReadOnly }
```

マイグレーション後:

```
{ isReadOnly = yes }
```

この方法でマイグレーション・ツールの影響を受けるプロパティは、以下のとおりです。

- addSpaceForSOSI
- allowUnqualifiedItemReferences
- boolean
- bypassValidation
- containerContextDependent
- currency
- cursor
- deleteAfterUse
- detectable
- fill
- helpGroup
- includeMsgInTransaction
- includeReferencedFunctions
- initialized
- inputRequired
- isDecimalDigit
- isHexDigit
- isNullable
- isReadOnly
- lowerCase
- masked
- modified
- needsSOSI
- newWindow
- numericSeparator
- openQueueExclusive
- pfKeyEquate
- resident
- runValidatorFromProgram
- segmented
- shared
- sqlVar
- upperCase

- wordWrap
- zeroFormat
- マイグレーション・ツールは、**currency** プロパティを **currency** と **currencySymbol** の 2 つのプロパティに分割します。次の表に、マイグレーション・ツールによる **currency** プロパティの変更の例を示します。

表 4. **currency** プロパティへの変更

マイグレーション前	マイグレーション後
{ currency = yes }	{ currency = yes }
{ currency = no }	{ currency = no }
{ currency = "usd" }	{ currency = yes, currencySymbol = "usd" }

- マイグレーション・ツールは、**dateFormat** および **timeFormat** プロパティの値を変更し、大/小文字が区別されるようにします。詳細については、『日付、時刻、およびタイム・スタンプ・フォーマット指定子 (*Date, time, and timestamp format specifiers*)』を参照してください。
- 設定メニューで「**列挙プロパティ値に修飾子を追加する (Add qualifiers to enumeration property values)**」チェック・ボックスにチェックマークを付けた場合、マイグレーション・ツールはプロパティの値に値のタイプを追加します。

マイグレーション前:

```
color = red
outline = box
```

マイグレーション後:

```
color = ColorKind.red
outline = OutlineKind.box
```

この変更によって、以下のプロパティが影響を受けます。

- align
- color
- deviceType
- displayUse
- highlight
- indexOrientation
- intensity
- outline
- protect
- selectType
- sign
- マイグレーション・ツールは、**tableNames** プロパティの値を、ストリングの配列の配列になるように変更します。それぞれのストリング配列は 1 つか 2 つの要素を持つ必要があります。最初の要素はテーブル名であり、2 番目の要素 (ある場合) はテーブル・ラベルです。次の表に、マイグレーション・ツールによる **tableNames** プロパティの変更の例を示します。

表 5. **tableNames** プロパティへの変更

マイグレーション前	マイグレーション後
<code>{ tableNames = (table1, table2) }</code>	<code>{ tableNames = ["table1", "table2"] }</code>
<code>{ tableNames = (table1 t1, table2) }</code>	<code>{ tableNames = ["table1", "t1", "table2"] }</code>
<code>{ tableNames = (table1 t1, table2 t2) }</code>	<code>{ tableNames = ["table1", "t1", "table2", "t2"] }</code>

- マイグレーション・ツールは、**tableNameVariables** プロパティの値を、**tableNames** プロパティの値の変更と同じ方法で変更します。
- マイグレーション・ツールは、**defaultSelectCondition** プロパティの値を、**sqlCondition** タイプとなるように変更します。

マイグレーション前:

```
{ defaultSelectCondition =
  #sql{
    hostVar02 = 4
  }
}
```

マイグレーション後:

```
{ defaultSelectCondition =
  #sqlCondition{ // #sqlCondition と中括弧の間にスペースを入れない
    hostVar02 = 4
  }
}
```

- マイグレーション・ツールは、**fillCharacter** の NULL 値を空字符串値の "" に置き換えます。

関連するタスク

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

関連する概念

108 ページの『EGL から EGL へのマイグレーション』

『EGL 間マイグレーション設定の変更』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

EGL 間マイグレーション設定の変更

EGL V6.0 マイグレーション・ツールが EGL ソース・コードを変換する方法を制御する設定を変更できます。このマイグレーション・ツールに関する詳細については、『EGL から EGL へのマイグレーション』を参照してください。マイグレーション・ツールの設定を変更する手順は、次のとおりです。

1. 「ウィンドウ」>「設定」をクリックする。
2. **EGL** を展開する。
3. 「**EGL V6.0 マイグレーション設定**」をクリックする。

注: 「**EGL V6.0 マイグレーション設定**」を検出できない場合は、EGL V6.0 マイグレーション機能を使用可能にしてください。『EGL 機能の使用可能化』を参照してください。

4. ラジオ・ボタンをクリックして、新しい予約語との名前の競合を解決する方法を選択する。
 - ・「**接頭部の追加 (Add prefix)**」は、ソース・コード内で予約語になった語に接頭部を追加するように、マイグレーション・ツールを設定します。このラジオ・ボタンのテキスト・ボックスに、変更された語にマイグレーション・ツールが追加する接頭部を入力してください。
 - ・「**接尾部の追加 (Add suffix)**」は、ソース・コード内で予約語になった語に接尾部を追加するように、マイグレーション・ツールを設定します。このラジオ・ボタンのテキスト・ボックスに、変更された語にマイグレーション・ツールが追加する接尾部を入力してください。
5. 可能な値の有限リストがあるプロパティの値に修飾子を追加するには、「**列挙プロパティ値に修飾子を追加する (Add qualifiers to enumeration property values)**」チェック・ボックスを選択する。このボックスにチェック・マークが付いている場合、マイグレーション・ツールは、値のタイプを値の名前に追加します。
6. マイグレーション・ツールによって変更されたファイルにコメントを追加するには、「**ロギング・レベル**」の下でオプションを選択する。
 - ・「**マイグレーション・ツールが処理するすべてのファイルにコメントを追加する (Add comments to all files processed by the migration tool)**」は、ファイルに変更を加えていない場合であっても、処理する各ファイルにコメントを追加するように、マイグレーション・ツールを設定します。
 - ・「**マイグレーション・ツールが変更するすべてのファイルにコメントを追加する (Add comments to all files that are changed by the migration tool)**」は、変更を加えるファイルのみにコメントを追加するように、マイグレーション・ツールを設定します。
 - ・「**ファイルにコメントを追加しない (Do not add comments to files)**」は、処理するファイルにコメントを追加しないように、マイグレーション・ツールを設定します。
 - ・「**ファイルの先頭に追加する (Add to beginning of file)**」は、ファイルの先頭にコメントを追加するように、マイグレーション・ツールを設定します。
 - ・「**ファイルの終わりに追加する (Add to end of file)**」は、ファイルの終わりにコメントを追加するように、マイグレーション・ツールを設定します。
7. 処理されたファイルのリストを V60MigrationLog.txt という名前のファイルに書き込むには、「**プロジェクトごとにマイグレーション結果をログ・ファイルに追加する (Append migration results to log file per project)**」チェック・ボックスを選択する。
8. 「**適用**」をクリックする。
9. 「**OK**」をクリックする。

関連するタスク

129 ページの『EGL 機能の使用可能化』

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

関連する概念

108 ページの『EGL から EGL へのマイグレーション』

環境のセットアップ

EGL 設定の変更

以下のような基本 EGL 設定を変更します。

1. 「ウィンドウ」>「設定」をクリックする。
2. リストが表示された後、「EGL」をクリックして EGL 画面を表示する。
3. 「VisualAge Generator との互換性」チェック・ボックスを選択またはクリアする。『VisualAge Generator との互換性』で説明されているように、この選択を行うと、開発時に使用可能なオプションに影響を与えます。
4. 「Encoding (エンコード)」リスト・ボックスで、新規 EGL ビルド (.eglbld) ファイルの作成時に使用する文字エンコード・セットを選択する。この設定は、既存のビルド・ファイルには影響しません。デフォルト値は UTF-8 です。
5. 「ユーザー ID」テキスト・ボックスで、リモート・ビルド・マシン (ある場合) にアクセスするためのユーザー ID を指定する。ビルド記述子オプション「destUserID」が優先し、このオプションと設定値は、マスター・ビルド記述子オプション「destUserID」よりも優先します。
6. 「パスワード」テキスト・ボックスで、リモート・ビルド・マシン (ある場合) にアクセスするためのパスワードを指定する。ビルド記述子オプション「destPassword」が優先し、このオプションと設定値は、マスター・ビルド記述子オプション「destPassword」よりも優先します。
7. 「適用」をクリックする。

その他の EGL 設定を変更するには、このページの下部にある関連タスクのリストを参照してください。設定の変更が終了した後、「OK」をクリックします。

関連する概念

354 ページの『ビルド』

477 ページの『VisualAge Generator との互換性』

関連するタスク

122 ページの『デフォルトのビルド記述子の設定』

120 ページの『EGL デバッガーの設定の変更』

124 ページの『ソース・スタイルの設定の変更』

126 ページの『SQL データベース接続設定の変更』

128 ページの『SQL 検索設定の変更』

テキストの設定の変更

EGL エディターでのテキストの表示方法を変更するには、次のようにします。

1. 「ウィンドウ」>「設定」をクリックする。
2. 設定のリストが表示されたならば、「ワークベンチ」を展開し、「色とフォント」をクリックする。「色とフォント」ペインが表示されます。
3. 「EGL」および「エディター (Editor)」を展開し、「EGL エディター・テキスト・フォント」をクリックする。

4. フォントと色のリストから選択を行うには、「**変更 (Change)**」ボタンをクリックしてから、次のようにする。
 - a. フォント設定を変更するには、スクロール可能リストからフォント、フォント・スタイル、およびサイズを選択する。
 - b. 色設定を変更するには、ドロップダウン・リストから色を選択する。
 - c. テキストの中央に線を引きたい場合は、「**取り消し線**」チェック・ボックスを選択する。
 - d. テキストの下に線を引きたい場合は、「**アンダーライン**」チェック・ボックスを選択する。
 - e. 選択したもののプレビューを、「**サンプル**」ボックスに表示できる。選択が終了したならば、「**OK**」をクリックします。
5. デフォルトのオペレーティング・システムのフォントを使用するには、「**システム・フォントの使用**」ボタンをクリックする。
6. デフォルトのワークベンチのフォントを使用するには、「**リセット**」ボタンをクリックする。
7. EGL エディターだけでなく、すべてのエディターのフォントをデフォルトのワークベンチのフォントに設定するには、「**デフォルトの復元**」ボタンをクリックする。
8. 変更を保管するには、「**適用**」または (設定の変更が完了した場合は)「**OK**」をクリックする。

関連するタスク

119 ページの『EGL 設定の変更』

EGL デバッガーの設定の変更

EGL デバッガーの設定を変更するには、以下の手順を実行します。

1. 「**ウィンドウ**」>「**設定**」をクリックする。
2. リストが表示された後、「**EGL**」を展開して「**デバッグ**」をクリックする。
3. 「**必要な場合は SQL ユーザー ID およびパスワードを要求**」というラベルのチェック・ボックスをクリアまたは選択する。

選択に関する詳細については、『*EGL デバッガー*』を参照してください。

4. 「**systemType を DEBUG に設定**」というラベルのチェック・ボックスをクリアまたは選択する。

選択に関する詳細については、『*EGL デバッガー*』を参照してください。

5. sysVar.terminalID、sysVar.sessionID、および sysVar.userID の初期値を設定する。値を指定しない場合は、Windows 2000/NT/XP または Linux™ のデフォルトのユーザー ID が使用されます。
6. EGL デバッガー・ポートの値を設定する。デフォルトは 8345 です。
7. デバッグ・セッション中のデータ処理時に使用する文字エンコードの種類を選択する。デフォルトはローカル・システムのファイル・エンコードです。選択に関する詳細については、『*EGL デバッガーの文字エンコード・オプション*』を参照してください。

8. デバッガーの実行時にユーザーに対して外部 Java クラスを指定するには、クラスパスを変更する。たとえば、MQSeries、JDBC ドライバー、または Java アクセス関数をサポートするには、追加のクラスが必要となる場合があります。

追加したクラスパスは、WebSphere® Application Server テスト環境では可視ではありません。ただし、サーバー構成の「Environment (環境)」タブを使用すると、この環境のクラスパスに追加できます。

「クラスパスの順序」ボックスの右側のボタンを使用します。

- プロジェクト、JAR ファイル、ディレクトリー、または変数を追加するには、「プロジェクトの追加」、「JAR の追加」、「ディレクトリーの追加」、または「変数の追加」のいずれかのボタンをクリックする。
 - エントリーを除去するには、エントリーを選択して、「除去」をクリックする。
 - 複数のエントリーのリストにエントリーを移動するには、エントリーを選択し、「上に移動」または「下に移動」をクリックする。
9. デフォルト設定を復元するには、「デフォルトの復元」をクリックする。
 10. 変更を保管するには、「適用」または (設定の変更が完了した場合は) 「OK」をクリックする。

関連する概念

477 ページの『VisualAge Generator との互換性』

303 ページの『EGL デバッガー』

『EGL デバッガーの文字エンコード・オプション』

関連するタスク

126 ページの『SQL データベース接続設定の変更』

関連する参照項目

999 ページの『sessionID』

1004 ページの『terminalID』

1005 ページの『userID』

EGL デバッガーの文字エンコード・オプション

EGL デバッガーでは、デバッグ中に使用する文字エンコードのタイプを指定することができます。文字エンコードは、デバッガーが文字データと数値データを内部で表示する方法、文字データを比較する方法、およびリモート・プログラム、ファイル、およびデータベースにパラメーターを送信する方法をコントロールします。これらのオプションを変更するには、『EGL デバッガーの設定の変更』を参照してください。

EGL デバッガーは、ローカル・システムと EBCDIC におけるデフォルト・エンコードという、2 つの異なる文字エンコード・タイプをサポートしています。EGL デバッガーのデフォルトの文字エンコードは、ご使用のローカル・システムのデフォルト・エンコードと同じになります。

- デフォルトの文字エンコードを選択した場合、デバッガーは、デフォルト形式 (通常は ASCII) で、CHAR、DBCHAR、MBCHAR、DATE、TIME、INTERVAL、NUM、および

NUMC 変数を表示します。文字変数の比較では、ASCII 照合シーケンスが使用されます。リモート・プログラムを呼び出す場合やリモート・ファイルおよびデータベースにアクセスする場合には、データをホストの形式に変換する必要があります。

この設定を選択し、変換表を指定しない場合、デバッガーは、リモート・プログラムを呼び出す際やリモート・ファイルとデータベースにアクセスする際に、適切な変換表を選択します。変換表については、『データ変換』を参照してください。

- EBCDIC 文字エンコードを使用する場合、デバッガーは、EBCDIC エンコードを使用して CHAR、DBCHAR、MBCHAR、DATE、TIME、および INTERVAL 変数を表示します。NUM および NUMC 変数は、ホストの数値形式で表示されます。文字変数の比較では、EBCDIC 照合シーケンスが使用されます。リモート・プログラムの呼び出し時やリモート・ファイルとデータベースのアクセス時にデータをホスト形式に変換する必要はありませんが、SQL 呼び出しやローカル C++ ルーチンへの呼び出しを行ったときに、データは適切な Java または ASCII 形式に変換されます。EBCDIC エンコードは、幾つかの言語で使用可能です。

EBCDIC 文字エンコードを選択し、変換表を指定しない場合、デバッガーは、リモート・プログラムを呼び出す際やリモート・ファイルとデータベースにアクセスする際に、変換表を使用しません。プログラム名、ライブラリー名、および任意のパス済みパラメーターは、EBCDIC 文字エンコードに従ってエンコードされます。

選択した文字エンコードがご使用の Java ランタイム環境でサポートされない場合、デバッガーの開始時に警告メッセージが表示されます。デバッグの続行を選択すると、デバッガーはデフォルト・エンコードのタイプに戻ります。

デバッグ・セッション中は文字エンコードを変更できません。文字エンコードの変更を有効にするには、デバッガーを再開する必要があります。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

120 ページの『EGL デバッガーの設定の変更』

関連する参照項目

507 ページの『データ変換』

デフォルトのビルド記述子の設定

デフォルトのビルド記述子とビルド記述子優先順位規則の概要については、『ワークベンチでの生成』を参照してください。

ワークベンチ・レベルでビルド記述子の設定を指定するには、次のようにします。

1. 「ウィンドウ」>「設定」をクリックする。
2. リストが表示された後、「EGL」を展開し、「デフォルトのビルド記述子」をクリックする。

3. 「デバッグ・ビルド記述子」および「ターゲット・システムのビルド記述子」を選択する。
4. 「Apply (適用)」をクリックして「OK」をクリックする。

ファイル、フォルダー、パッケージ、またはプロジェクト・レベルでビルド記述子の設定を指定するには、次のようにします。

1. 必要なレベルを右クリックし (ファイル名やフォルダー名など)、コンテキスト・メニューで「プロパティー」をクリックする。
2. 「EGL デフォルト・ビルド記述子」を選択する。
3. 「デバッグ・ビルド記述子」および「ターゲット・システムのビルド記述子」を選択する。
4. 「OK」をクリックする。

関連する概念

359 ページの『ワークベンチでの生成』

EGL エディターの設定の変更

EGL エディター設定を指定するには、次のようにします。

1. 「ウィンドウ」>「設定」をクリックする。
2. リストが表示された後、「EGL」を展開して「エディター」をクリックする。
3. EGL ファイルを検討するときに行番号を表示するには、「行番号の表示」チェック・ボックスを選択する。行番号をクリアするには、チェック・ボックスをクリアする。ファイル自体は影響を受けません。
4. ソース・コード内でエラーが見つかったときに赤色のアンダーラインを表示するには、「テキスト内のエラーに注釈を付ける」チェック・ボックスを選択する。これらのアンダーラインをクリアするには、チェック・ボックスをクリアします。ファイル自体は影響を受けません。
5. ソース・コード内でエラーが見つかったときにエディターの右マージン (概要罫線) に赤色のエラー標識を表示するには、「概要表示域のエラーに注釈を付ける」チェック・ボックスを選択する。エラー標識をクリックすると、ソース・コード内のエラーのロケーションに移動します。エラー標識をクリアするには、チェック・ボックスをクリアします。ファイル自体は影響を受けません。
6. ソース・スタイルを指定するには、『ソース・スタイル設定の変更』で説明されている手順を実行する。
7. コンテンツ・アシストで使用するためにテンプレートを追加、除去、またはカスタマイズするには、『テンプレートの設定の変更』で説明されているプロセスを行う。
8. テキストの表示方法を変更するには、『テキストの設定の変更』で説明されているプロセスを実行する。

関連するタスク

124 ページの『ソース・スタイルの設定の変更』

124 ページの『テンプレートの設定の変更』

119 ページの『テキストの設定の変更』

関連する参照項目

524 ページの『EGL でのコンテンツ・アシスト』

ソース・スタイルの設定の変更

EGL エディターでの EGL コードの表示方法を変更できます。

1. 「ウィンドウ」>「設定」をクリックする。
2. 設定リストが表示された後、「EGL」と「エディター」を展開し、「ソース・スタイル」をクリックする。
3. ソース・タイプの背景にしたい色を選択するには、「背景色」ボックスの「Custom (カスタム)」ラジオ・ボタンをクリックする。カスタム・ラベルの横のボタンをクリックします。カラー・パレットが表示されます。色を選択し、「OK」をクリックします。
4. 「前景色」ボックスでテキストのタイプを選択し、「Color (色)」ボタンをクリックする。カラー・パレットが表示されます。色を選択し、「OK」をクリックします。
5. タイプを太字に設定する場合は、「太字」チェック・ボックスをチェックする。
6. 変更を保管するには、「適用」または (設定の変更が完了した場合は) 「OK」をクリックする。

関連するタスク

119 ページの『EGL 設定の変更』

テンプレートの設定の変更

EGL エディターでコンテンツ・アシストを要求した場合に表示されるテンプレートを追加、除去、またはカスタマイズするには、次のようにします。

1. 「ウィンドウ」>「設定」をクリックする。
2. 設定リストが表示された後、「EGL」と「エディター」を展開し、「テンプレート」をクリックする。テンプレートのリストが表示されます。

注: Windows 2000/NT/XP の他のアプリケーションと同様、エントリーはクリックして選択できます。 **Ctrl** クリックを使用すると、他の選択に影響を与えずに選択または選択解除を行うことができます。 **Shift** クリックを使用すると、最後にクリックしたエントリーに隣接するエントリーのセットを選択できます。

3. EGL エディターでテンプレートを使用可能にするには、テンプレート名の左側のチェック・ボックスを選択する。リストされているすべてのテンプレートを使用可能にするには、「Enable All (すべて使用可能にする)」をクリックします。同様に、テンプレートを使用不可にするには、関連チェック・ボックスをクリアします。リストされているすべてのテンプレートを使用不可にするには、「Disable All (すべて使用不可にする)」をクリックします。
4. 新規テンプレートを作成するには、次のようにする。
 - a. 「新規」をクリックする。

- b. 名前と説明の組み合わせがすべてのテンプレートで固有である場合にのみ、テンプレートはコンテンツ・アシスト・リストに表示されることが保障されるため、「New Template (新規テンプレート)」ダイアログを表示するには、名前と説明の両方を指定する。

注: テンプレートで使用される最初のワードが EGL キーワード (Function など) である場合は、EGL エディターでコンテンツ・アシストを要求した場合にのみテンプレートを使用できます。ただし、表示中のカーソルが、ワードが有効である個所に配置されている場合に限られます。同様に、接頭部を入力してコンテンツ・アシストを要求する場合は、その接頭部で始まるすべてのテンプレートを使用できます。ただし、表示中のカーソルが、そのテンプレートで構文上許可されている位置にある場合に限ります。たとえば、「fun」を入力して関数テンプレートを要求します。接頭部または最初の完全なワードのいずれも入力しない場合、コンテンツ・アシストを要求したときにテンプレートは表示されません。

- c. 「パターン」フィールドで、テンプレート自体を入力する。

- 表示したい任意のテキストを入力する。
- 表示中のカーソル位置に既存の変数を配置するには、「**Insert Variable (変数の挿入)**」をクリックして変数をダブルクリックする。EGL エディターにテンプレートを挿入すると、これらの変数は適切な値に解決されます。
- カスタム変数を作成するには、ドル記号 (\$) を入力し、その後左中括弧 ({)、ストリング、および右中括弧 (}) を入力する。たとえば、以下のようになります。

```
${variable}
```

既存の変数を挿入し、必要に応じて名前を変更の方が簡単な場合があります。

EGL エディターにカスタム・テンプレートを挿入すると、各変数にアンダーラインが表示され、値が必要であることが示されます。

- タスクを完了するには、「**OK**」をクリックし、テンプレート画面で「**Apply (適用)**」をクリックする。
5. 既存のテンプレートを確認するには、リストされたエントリーをクリックし、「プレビュー」ボックスを検討する。
6. 既存のテンプレートを編集するには、リストされたエントリーをクリックし、「**編集**」をクリックする。「New Template (新規テンプレート)」ダイアログと同様、「Edit Template (テンプレートの編集)」ダイアログと対話します。
7. 既存のテンプレートを除去するには、リストされたエントリーをクリックし、「**除去**」をクリックする。複数のテンプレートを除去する場合は、Windows 2000/NT/XP の規則に従って複数のリスト・エントリーを選択し、「**除去**」をクリックします。
8. XML ファイルからテンプレートをインポートするには、テンプレート・リストの右側で「**Import (インポート)**」をクリックし、参照メカニズムに従ってファイルのロケーションを指定する。
9. XML ファイルにテンプレートをエクスポートするには、テンプレート・リストの右側で「**Export (エクスポート)**」をクリックし、参照メカニズムに従って新

規ファイルのロケーションを指定する。複数のテンプレートをエクスポートする場合は、Windows 2000/NT/XP のメカニズムに従って複数のリスト・エントリーを選択し、「**Export (エクスポート)**」をクリックします。

10. リストされているすべてのテンプレートを XML ファイルにエクスポートするには、「**Export All (すべてエクスポート)**」をクリックし、参照メカニズムに従ってファイルのロケーションを指定する。
11. 変更を保管するために、「**適用**」をクリックする。インストール時に有効になるテンプレート・リストに戻るには、「**デフォルトの復元**」をクリックします。

関連するタスク

135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

SQL データベース接続設定の変更

SQL データベース接続のページは、以下の理由で使用します。

- J2EE の外側でアクセスするデータベースへの宣言時アクセスとデバッグ時アクセスを使用可能にすることができる。
- ビルド記述子オプション `sqlJNDIName` に値を設定する。このオプションは、JNDI レジストリーでデフォルト・データ・ソースをバインドする名前を指定します (例: `java:comp/env/jdbc/MyDB`)。このオプションは、以下の状況で作成されるビルド記述子に含まれています。
 - 『EGL を処理するプロジェクトの作成』で説明されているように、EGL Web プロジェクト・ウィザードを使用する。
 - このウィザードで作業を行うときに、ビルド記述子を作成するよう要求する。

以下のようにします。

1. 「**ウィンドウ**」>「**設定**」をクリックする。
2. 設定リストが表示された後、「**EGL**」を展開し、「**SQL データベース接続**」をクリックする。
3. 「**接続 URL**」フィールドに JDBC を介したデータベースへの接続に使用している URL を入力する。
 - Windows 用の IBM DB2 APP DRIVER の場合、URL は `jdbc:db2:dbName` となります (ここで、`dbName` はデータベース名です)。
 - Oracle JDBC シン・クライアント・サイド・ドライバの場合、URL は、データベース・ロケーションごとに異なります。データベースがご使用のマシンに対してローカルな場合は、URL は `jdbc:oracle:thin:dbName` になります (ここで、`dbName` はデータベース名です)。データベースがリモート・サーバーにある場合は、URL は `jdbc:oracle:thin:@host:port:dbName` になります (ここで、`host` はデータベース・サーバーのホスト名、`port` はポート番号、`dbName` はデータベース名です)。
 - Informix JDBC NET ドライバの場合、URL は次のようになります (実際には 1 行に結合されます)。

```
jdbc:informix-sqli://host:port
/dbName:informixserver=servername;
user=userName;password=password
```

host

データベース・サーバーが配置されているマシンの名前。

port

ポート番号

dbName

データベース名

serverName

データベース・サーバーの名前

userName

Informix ユーザー ID

passWord

ユーザー ID に関連したパスワード

4. 「データベース」フィールドにデータベースの名前を入力する。
5. 「ユーザー ID」フィールドに接続用のユーザー ID を入力する。
6. 「パスワード」フィールドにユーザー ID のパスワードを入力する。
7. 「Database vendor type (データベース・ベンダー型)」フィールドで、JDBC 接続で使用しているデータベース製品とバージョンを選択する。
8. 「JDBC ドライバー」フィールドで、JDBC 接続で使用している JDBC ドライバーを選択する。
9. 「JDBC ドライバー・クラス」フィールドに、選択したドライバーのドライバー・クラスを入力する。Windows 用 IBM DB2 APP DRIVER の場合は、ドライバー・クラスは COM.ibm.db2.jdbc.app.DB2Driver になります。Oracle JDBC シン・クライアント・サイド・ドライバーの場合は、ドライバー・クラスは oracle.jdbc.driver.OracleDriver となります。Informix JDBC NET ドライバーの場合は、ドライバー・クラスは com.informix.jdbc.IfxDriver となります。その他のドライバー・クラスについては、各ドライバーの資料を参照してください。
10. 「class location (クラス・ロケーション)」には、ドライバー・クラスが含まれている *.jar ファイルまたは *.zip ファイルの完全修飾ファイル名を入力する。Windows 用 IBM DB2 APP DRIVER の場合は、db2java.zip ファイルに対して完全修飾ファイル名を入力する (例: d:\sql\lib\java\db2java.zip)。Oracle THIN JDBC DRIVER の場合は、classes12.zip ファイルに対して完全修飾ファイル名を入力する (例: d:\0ra81\jdbc\lib\classes12.zip)。その他のドライバー・クラスについては、各ドライバーの資料を参照してください。
11. 「接続 JNDI 名」フィールドで、J2EE で使用するデータベースを指定する。この値は、JNDI レジストリーでデータ・ソースをバインドする名前です (例: java:comp/env/jdbc/MyDB)。上記で説明されているように、この値は、特定の EGL Web プロジェクトに対して自動的に構成されるビルド記述子内のオプション「sqlJNDIName」に割り当てられています。
12. DB2 UDB を受け入れ、「2 次認証 ID」フィールドに値を指定する場合、この値は、検証時に EGL で使用する SET CURRENT SQLID ステートメントで使用されます。値は大/小文字を区別します。

設定値をクリアまたは適用できます。

- デフォルト値を復元するには、「**デフォルトの復元**」をクリックする。
- 「設定」ダイアログを終了せずに設定を適用するには、「**適用**」をクリックする。
- 設定を変更し終わったならば、「**OK**」をクリックする。

関連するタスク

132 ページの『EGL Web プロジェクトの作成』

119 ページの『EGL 設定の変更』

関連する参照項目

437 ページの『sqlJNDIName』

SQL 検索設定の変更

EGL 宣言時には、SQL 検索機能を使用して SQL テーブルの列から SQL レコードを作成できます。概要については『SQL サポート』を参照してください。

SQL 検索機能の設定を変更するには、次のようにします。

1. 「**ウィンドウ**」>「**設定**」をクリックし、「**EGL**」を展開し、「**SQL 検索**」をクリックする。
2. SQL 検索機能によって作成される各構造体項目の作成規則を指定する。
 - a. SQL 文字データ・タイプから構造体項目を作成する場合に使用する EGL タイプを指定するには、次のラジオ・ボタンの 1 つをクリックする。
 - 「**EGL 型ストリングを使用 (Use EGL type string)**」(デフォルト)は、SQL 文字データ・タイプを EGL ストリング・データ・タイプにマップします。
 - 「**EGL 型文字を使用 (Use EGL type char)**」は、SQL 文字データ・タイプを EGL 文字データ・タイプにマップします。
 - 「**EGL 型 mbChar を使用 (Use EGL type mbChar)**」は、SQL 文字データ・タイプを EGL mbChar データ・タイプにマップします。
 - 「**EGL 型ユニコードを使用 (Use EGL type Unicode)**」は、SQL 文字データ・タイプを EGL ユニコード・データ・タイプにマップします。
 - b. 構造体項目名における大文字と小文字の扱いを指定するために、以下のいずれかのラジオ・ボタンをクリックする。
 - 「**大/小文字を変更しない**」(デフォルト)は、構造体項目名の大文字と小文字の区別が関連するテーブル列名の大文字と小文字の区別と同じ状態であることを指定します。
 - 「**小文字に変更**」は、構造体項目名がテーブル列名を小文字にした名前であることを指定します。
 - 「**小文字に変更してアンダースコアの後の最初の文字を大文字に変更**」も構造体項目名がテーブル列名を小文字にした名前であることを指定しますが、テーブル列名で文字がアンダースコアの直後にある場合は構造体項目名を大文字に変更します。
 - c. テーブル列名のアンダースコアを構造体項目名に反映させる方法を指定するために、以下のいずれかラジオ・ボタンをクリックする。

- 「アンダースコアを変更しない」 (デフォルト) は、テーブル列名のアンダースコアを構造体項目名に含めることを指定します。
 - 「アンダースコアを除去 (Remove underscores)」は、テーブル列名のアンダースコアを構造体項目名に含めないことを指定します。
 - 「アンダースコアをハイフンに変更」は、テーブル列名のアンダースコアを構造体項目名ではハイフンに変更することを指定します。
3. Informix システム・スキーマのパーツであるテーブルからデータを検索する場合は、「システム・スキーマの除外」 チェック・ボックスのチェック・マークを外します。(この場合、「Informix」はテーブル所有者です。) その他の場合は、このチェック・ボックスを選択し、SQL 検索機能のパフォーマンスを向上させます。

このチェック・ボックスはデフォルトで選択されています。

関連する概念

247 ページの『SQL サポート』

関連するタスク

272 ページの『SQL テーブル・データの検索』

119 ページの『EGL 設定の変更』

126 ページの『SQL データベース接続設定の変更』

関連する参照項目

271 ページの『Informix および EGL』

EGL 機能の使用可能化

EGL 機能にアクセスするには、EGL 機能を使用可能にする必要があります。以下の EGL 機能を使用することができます。

EGL 開発

EGL アプリケーションの開発とデバッグに関連するすべての機能から構成されます。

EGL V6.0 マイグレーション

EGL V6.0 iFix に準拠するために、EGL 5.1.2 および 6.0 ソース・コードの変換に関連したすべての機能から構成されます。

VisualAge Generator から EGL へのマイグレーション

既存の VisualAge Generator コードを EGL コードへマイグレーションすることに関連したすべての機能から構成されます。

EGL 機能を使用可能にするには、次のようにします。

1. 「ウィンドウ」>「設定」をクリックする。
2. 設定のリストが表示されたならば、「ワークベンチ」を展開し、「機能 (Capabilities)」をクリックする。「機能」ペインが表示されます。
3. 使用可能にされた機能を必要とするフィーチャーが最初に使用されたときに、プロンプトを受け取りたい場合は、「機能を使用可能にする際にプロンプトを出す」のチェック・ボックスを選択する。
4. 「EGL デベロッパー機能」フォルダーを展開する。

5. 目的の EGL 機能のチェック・ボックスを選択する。あるいは、「**EGL デベロッパー機能**」フォルダーを選択して、そのフォルダーに含まれるすべての機能を有効にすることもできます。
6. 使用可能にされた機能のリストを、製品のインストール時の状態に戻すには、「**デフォルトを復元**」ボタンをクリックする。
7. 変更を保管するには、「**適用 (Apply)**」をクリックし、「**OK**」をクリックする。

注: EGL 機能を使用可能にすると、EGL アプリケーションの開発とデバッグに必要なその他の機能も、自動的に使用可能になります。

関連するタスク

119 ページの『EGL 設定の変更』

コード開発の開始

プロジェクトの作成

EGL プロジェクトの作成

作業を編成する方法に関する概要については、『*EGL* プロジェクト、パッケージ、およびファイル』を参照してください。

新規 EGL プロジェクトを設定するには、次のようにします。

1. ワークベンチで、以下の手順のいずれかを行う。
 - 「ファイル」>「新規」>「プロジェクト」を選択する。または
 - 「新規」>「プロジェクト」を右クリックしてからクリックする。

「新規プロジェクト」ウィザードが開きます。
2. 「EGL」を展開してから「EGL プロジェクト (EGL Project)」をクリックする。「次へ」をクリックする。「新規 EGL プロジェクト」ウィザードが表示されます。

注: EGL プロジェクトが使用不可である場合は、「すべてのウィザードを表示」チェック・ボックスにチェック・マークを付けてください。

3. 「プロジェクト名」フィールドにプロジェクトの名前を入力する。デフォルトでは、プロジェクトはワークスペースに配置されます。ただし、「参照」をクリックして異なるロケーションを選択できます。
4. ビルド記述子を指定する方法を選択する。ビルド記述子は、生成時間に処理を行うよう指示するパーツです。
 - 「新規プロジェクト・ビルド記述子を自動的に作成する」は、EGL がビルド記述子を提供し、プロジェクトと同じ名前を持つビルド・ファイル (拡張子 .eglbld) にこのビルド記述子を書き込むことを意味する。

これらのビルド記述子に値を指定する場合は、「オプション」をクリックします。これらの値を後で変更するには、作成されたビルド・ファイルを変更します。

詳細については、『プロジェクト作成時のデータベース・オプションの指定』を参照してください。

- 「EGL 設定で指定されているビルド記述子を使用する」は、EGL が、EGL 設定として作成および識別されるビルド記述子を指すことを意味する。
 - 「既存のビルド記述子を選択する」によって、ワークスペースで使用可能なビルド記述子からビルド記述子を指定することが可能になる。
5. 多くの場合、ここで「完了」をクリックする。ただし、「次へ」を選択した場合は、その他のソース・フォルダーおよびプロジェクトを指定し、作成するプロジェクトから参照することができます。その他のソース・フォルダーおよびプロジェクトを選択し終わった後、「完了」を選択します。

関連する概念

319 ページの『ビルド記述子パーツ』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連するタスク

133 ページの『プロジェクト作成時のデータベース・オプションの指定』

126 ページの『SQL データベース接続設定の変更』

EGL Web プロジェクトの作成

作業を編成する方法に関する概要については、『EGL プロジェクト、パッケージ、およびファイル』を参照してください。

新規 EGL Web プロジェクトを設定するには、次のようにします。

1. ワークベンチで、以下の手順のいずれかを行う。

- 「ファイル」>「新規」>「プロジェクト」を選択する。または
- 「新規」>「プロジェクト」を右クリックしてからクリックする。

「新規プロジェクト」ウィザードが開きます。

2. 「EGL」を展開してから「EGL Web プロジェクト」をクリックする。「次へ」をクリックする。「新規 EGL Web プロジェクト」ウィザードが表示されます。

3. 「プロジェクト名」フィールドにプロジェクトの名前を入力する。デフォルトでは、プロジェクトはワークスペースに配置されます。ただし、「参照」をクリックして異なるロケーションを選択することができます。

4. ビルド記述子を指定する方法を選択する。ビルド記述子は、生成時間に処理を行うよう指示するパーツです。

- 「新規プロジェクト・ビルド記述子を自動的に作成する」は、EGL がビルド記述子を提供し、プロジェクトと同じ名前を持つビルド・ファイル (拡張子 .eglbld) にこのビルド記述子を書き込むことを意味する。

これらのビルド記述子に値を指定する場合は、「オプション」をクリックします。これらの値を後で変更するには、作成されたビルド・ファイルを変更します。

詳細については、『プロジェクト作成時のデータベース・オプションの指定』を参照してください。

- 「EGL 設定で指定されているビルド記述子を使用する」は、EGL が、EGL 設定として作成および識別されるビルド記述子を指すことを意味する。
- 「既存のビルド記述子を選択する」によって、ワークスペースで使用可能なビルド記述子からビルド記述子を指定することが可能になる。

5. ビルド記述子を自動的に作成するよう要求する場合は、「SQL 接続の JNDI 名」フィールドに値を入れることができる。この結果、デバッグ時または生成時に JNDI レジストリーでデフォルト・データ・ソースにバインドされた名前が割り当てられます。(たとえば、java:comp/env/jdbc/MyDB。) 選択を行うと、ビルド記述子オプション sqlJNDIName に値が割り当てられます。「SQL 接続の

JNDI 名 フィールドにすでに値が入力されている場合は、『SQL データベース接続設定の変更』で説明されているように、値はワークベンチ設定から取得されます。

6. 多くの場合、ここで「完了」をクリックする。追加のカスタマイズ (任意の Web プロジェクトに実行可能) を行うには、ダイアログの下部で J2EE の設定を構成します。オプションとして、「拡張を隠す」をクリックして J2EE の設定を隠すことができます。「次へ」をクリックする。機能設定を選択し、「次へ」を選択します。ページ・テンプレートを選択し、「完了」をクリックします。

関連する概念

319 ページの『ビルド記述子パーツ』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連するタスク

『プロジェクト作成時のデータベース・オプションの指定』

126 ページの『SQL データベース接続設定の変更』

関連する参照項目

437 ページの『sqlJNDIName』

プロジェクト作成時のデータベース・オプションの指定

EGL で自動的に作成されるビルド記述子にオプション値を割り当てるには、「プロジェクト・ビルド・オプション」ダイアログで作業します。ダイアログを表示する方法の詳細については、『EGL を処理するプロジェクトの作成』を参照してください。

設定で指定したデータベース接続情報を受け入れるには、チェック・ボックスをクリックします。

次の表は、各スクリーン内のラベルと関連ビルド記述子オプションを示しています。

ラベル	ビルド記述子オプション
データベース・タイプ	dbms
データベース JDBC ドライバー	sqlJDBCDriverClass
データベース名	sqlJNDIName (J2EE 出力の場合) または sqlIDB (非 J2EE 出力の場合)

関連する概念

319 ページの『ビルド記述子パーツ』

関連するタスク

132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

415 ページの『ビルド記述子オプション』

EGL ソース・フォルダーの作成

いったんワークベンチ内にプロジェクトを作成すると、そのプロジェクト内に、EGL ファイルを入れる 1 つ以上のフォルダーを作成できます。

EGL ファイルのグループ化用にフォルダーを作成するには、次のようにします。

1. ワークベンチで「ファイル」 > 「新規」 > 「EGL ソース・フォルダー」をクリックする。
2. EGL フォルダーを含めるプロジェクトを選択する。「フォルダー名」フィールドに、EGL フォルダーの名前を入力する (例: myFolder)。
3. 「完了」 ボタンをクリックする。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
1 ページの『EGL の紹介』

関連するタスク

132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

135 ページの『EGL ソース・ファイルの作成』
725 ページの『命名規則』

EGL パッケージの作成

EGL パッケージは、関連するソース・パーツ名付きコレクションです。EGL パッケージを作成するには、次のようにします。

1. パッケージを含めるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「EGL パッケージ」をクリックする。
3. EGL パッケージを含めるプロジェクトまたはフォルダーを選択する。「ソース・フォルダー」フィールドには、プロジェクト・エクスプローラーでの現在の選択に応じて、事前に値が入力されている場合があります。
4. 「パッケージ名」フィールドに EGL パッケージの名前を入力する。パッケージの命名規則の詳細については、『EGL プロジェクト、パッケージ、およびファイル』を参照してください。
5. 「完了」 ボタンをクリックする。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
1 ページの『EGL の紹介』

関連するタスク

『EGL ソース・フォルダーの作成』
132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

『EGL ソース・ファイルの作成』

EGL ソース・ファイルの作成

EGL ソース・ファイルを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「EGL ソース・ファイル」をクリックする。
3. EGL ファイルを入れるプロジェクトまたはフォルダーを選択する。EGL ファイルを入れるパッケージを選択する。「EGL ソース・ファイル名」フィールドに、EGL ファイルの名前を入力する (例: myEGLFile)。
4. 「完了」をクリックしてファイルを作成する。自動的にファイル名の末尾に拡張子 (.egl) が付加されます。EGL ファイルが「プロジェクト・エクスプローラー」ビューに表示され、デフォルトの EGL エディターで自動的に開かれます。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

1 ページの『EGL の紹介』

関連するタスク

134 ページの『EGL ソース・フォルダーの作成』

132 ページの『EGL Web プロジェクトの作成』

コンテンツ・アシストを使用した EGL テンプレートの使用

コンテンツ・アシストを使用するには、次のようにします。

1. 新規 EGL ファイルを開く。
2. 選択可能な行に「P」（ページ・ハンドラーまたはプログラムを表します）と入力し、**Ctrl + Space** を押す。
3. ポップアップが表示された後、カスタマイズするパーツのアイコンをクリックする。以下の手順のいずれかを実行します。
 - **Enter** を押して、リスト内の最初のアイコンを選択する。
 - 矢印キーを使用して別の (プログラムの) アイコンを選択し、**Enter** を押す。

エディターは、ファイル内にパーツ・テンプレートを配置します。

4. パーツをカスタマイズする。

テンプレートが表示された後、エディターで、情報を入力する必要がある最初の領域を強調表示します。この場合はパーツ名を指定します。入力後に **Tab** を押し、入力が必要な次の領域を強調表示します。

Tab キーを繰り返し使用します。この操作は、ファイルの最後に達するか、または他の方法でファイル内の位置を変更するまで繰り返します。

5. プログラムまたはページ・ハンドラーに関数を挿入するには、「F」 (関数) を入力し、**Ctrl + Space** を押す。パーツ・テンプレートを再度選択できますが、次のようにします。

- 矢印キーまたはマウスを使用して、リストの最後にスクロールする。
- **Enter** を押すか、またはワード「*Function (関数)*」をクリックする。アイコンが表示されないのは、パーツ・テンプレートではなくストリングを選択したことを意味します。

ストリングを選択することは、変数名をすばやく入力する場合など、その他のコンテキストで役立ちます。

6. ワード「*Function (関数)*」の末尾にカーソルを置き、**Ctrl + Space** を押してリストでアイコンをクリックする。

エディターは、ファイル内に関数テンプレートを配置します。

7. パーツをカスタマイズする。
8. コードを開発する場合と同様、定期的に **Ctrl + Space** を押し、提供されているサービスの範囲について理解する。

関連するタスク

157 ページの『EGL および JSP ファイルへのコードの断片の挿入』

124 ページの『テンプレートの設定の変更』

関連する参照項目

570 ページの『EGL ソース形式の関数パーツ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

783 ページの『EGL ソース形式のプログラム・パーツ』

EGL のキーボード・ショートカット

次の表は、EGL エディターで使用可能なキーボード・ショートカットを示しています。

キーの組み合わせ	関数
Ctrl+/	コメント
Ctrl+¥	コメント解除
Ctrl+A	すべて選択
Ctrl+C	コピー
Ctrl+F	検索
Ctrl+H	検索
Ctrl+K	次を検索
Ctrl+S	保管
Ctrl+V	貼り付け
Ctrl+X	切り取り
Ctrl+G	生成
Ctrl+L	特定の行へジャンプ
Ctrl+Y	やり直し

キーの組み合わせ	関数
Ctrl+Z	元に戻す
Ctrl+Shift+A	明示的な SQL ステートメントを、暗黙の SQL ステートメントを持つ EGL I/O ステートメントに追加する。
Ctrl+Shift+K	前を検索
Ctrl+Shift+N	「パーツを開く」ダイアログにアクセスする
Ctrl+Shift+P	EGL の prepare ステートメントと、それに関連する get 、 execute 、または open ステートメントを構成する
Ctrl+Shift+R	取得機能を使用して SQL レコード・パーツ内の項目の作成または上書きを行う
Ctrl+Shift+S	プロジェクト・エクスプローラーで現在のファイルを表示する
Ctrl+Shift+V	EGL I/O ステートメントに関連した SQL ステートメントの表示および妥当性検査を行い、関連するアクションを実行する
Ctrl+Space	コンテンツ・アシストの取得
F3	名前が強調表示されているパーツを含んでいるファイルを開く
Tab	テキストを次のタブ停止位置までインデントする

基本 EGL ソース・コードの開発

EGL dataItem パーツの作成

EGL dataItem パーツは、分割不可能なメモリー領域を定義します。EGL dataItem パーツは、EGL ファイルに入っています。EGL データ項目パーツを作成するには、次のようにします。

1. データ項目パーツを入れる EGL ファイルを識別し、そのファイルを EGL エディターで開く。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
2. EGL 構文に従って dataItem パーツの特性を入力する（詳細については、『EGL ソース形式の DataItem パーツ』を参照してください）。コンテンツ・アシストを使用して、dataItem パーツの構文のアウトラインをファイルに入れることができます。
3. EGL ファイルを保管する。

関連する概念

『DataItem パーツ』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

関連する参照項目

524 ページの『EGL でのコンテンツ・アシスト』

512 ページの『EGL ソース形式の DataItem パーツ』

725 ページの『命名規則』

DatalItem パーツ

dataItem パーツ は、これ以上分けられないメモリー領域を定義します。*dataItem* パーツは、固定構造体内の構造体フィールドとは異なり、独立したパーツです。

primitive variable は、*dataItem* パーツ、または INT や CHAR(2) などのプリミティブ宣言に基づくメモリー領域です。プリミティブ変数は、以下のような方法で使用できます。

- 関数またはプログラムの中にデータを受け取るパラメーターとして
- EGL 関数の変数として (例えば、assignment ステートメントで、またはデータを他の関数やプログラムに渡す引数として)

それぞれのプリミティブ変数には、デフォルトで備えているか、または変数が *dataItem* パーツの中で指定されている一連のプロパティーがあります。詳細については、『EGL プロパティーとオーバーライドの概要』を参照してください。

関連する概念

142 ページの『固定レコード・パーツ』

28 ページの『固定構造体』
68 ページの『EGL プロパティの概要』
19 ページの『パーツ』
141 ページの『レコード・パーツ』
30 ページの『Typedef』

関連するタスク

124 ページの『テンプレートの設定の変更』

関連する参照項目

512 ページの『EGL ソース形式の DataItem パーツ』
532 ページの『EGL ソース形式』
511 ページの『データの初期化』
37 ページの『プリミティブ型』

EGL レコード・パーツの作成

レコード・パーツは、構造体 (ストレージ内の固定サイズのデータ・エレメントの階層レイアウト) と、オプションのバインディングを定義します。これは、レコードと外部データ・ソース (ファイル、データベース、またはメッセージ・キュー) との関係です。EGL レコード・パーツは、EGL ファイルに入っています。EGL レコード・パーツを作成するには、次のようにします。

1. レコード・パーツを入れる EGL ファイルを識別し、そのファイルを EGL エディターで開く。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
2. EGL 構文に従って、レコード・パーツの特性を入力する (詳細については、『EGL ソース形式の基本レコード・パーツ』、『EGL ソース形式の索引付きレコード・パーツ』、『EGL ソース形式の MQ レコード・パーツ』、『EGL ソース形式の相対レコード・パーツ』、『EGL ソース形式のシリアル・レコード・パーツ』、および『EGL ソース形式の SQL レコード・パーツ』を参照してください)。コンテンツ・アシストを使用して、レコード・パーツの構文のアウトラインをファイルに入れることができます。
3. EGL ファイルを保管する。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
141 ページの『レコード・パーツ』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』
135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

関連する参照項目

413 ページの『EGL ソース形式の基本レコード・パーツ』
524 ページの『EGL でのコンテンツ・アシスト』
578 ページの『EGL ソース形式の索引付きレコード・パーツ』
714 ページの『EGL ソース形式の MQ レコード・パーツ』
725 ページの『命名規則』
796 ページの『EGL ソース形式の相対レコード・パーツ』

799 ページの『EGL ソース形式のシリアル・レコード・パーツ』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

レコード・パーツ

レコード・パーツ とは、あるデータのシーケンスを定義したもので、その長さは生成時に既知でなくてもよく、その内容は、いくつかのフィールドから構成されます。EGL では、フィールドとはレコード・パーツに基づいたレコードの中の変数を定義したものです。

フィールドとしては、辞書、arrayDictionary、または辞書か arrayDictionary の配列、あるいは、以下のいずれかに基づくものなどがあります。

- STRING などのプリミティブ型
- DataItem パーツ
- 固定レコード・パーツ (後述)
- 別のレコード・パーツ
- 上記のいずれかの種類の配列

次の 2 つのタイプのレコード・パーツがあります。

- basicRecord。これは汎用処理に使用されますが、データ・ストアへのアクセスには使用されません。
- SQLRecord。これは、リレーショナル・データベースへのアクセスに使用されます。

レコード は、以下のコンテキストで使用できます。

- リレーショナル・データベースとの間でデータをコピーする文の中
- assignment または move ステートメントの中
- 他のプログラムまたは関数にデータを渡す引数として
- プログラムまたは関数の中にデータを受け取るパラメーターとして

レコード・パーツは、固定レコード・パーツとは異なります。固定レコード・パーツは、生成時に長さが既知であるデータのシーケンスを定義したものです。固定レコード・パーツは、基本的に、VSAM ファイル、MQSeries メッセージ・キュー、およびその他の順次ファイルにアクセスするために使用されます。

レベル番号を含んでいるレコード・パーツは、たとえそのレコード・パーツのタイプが basicRecord または SQLRecord であっても、固定レコード・パーツです。その他の詳細については、『固定レコード・パーツ』を参照してください。

関連する概念

139 ページの『DataItem パーツ』

142 ページの『固定レコード・パーツ』

19 ページの『パーツ』

143 ページの『レコード・タイプとプロパティ』

331 ページの『リソース関連とファイル・タイプ』

28 ページの『固定構造体』

30 ページの『Typedef』

関連するタスク

- 122 ページの『デフォルトのビルド記述子の設定』
- 123 ページの『EGL エディターの設定の変更』

関連する参照項目

- 532 ページの『EGL ソース形式』
- 511 ページの『データの初期化』
- 37 ページの『プリミティブ型』

固定レコード・パーツ

固定レコード・パーツとは、生成時に長さが既知であるデータのシーケンスを定義したものです。この種類のパーツは、必ず一連のプリミティブで固定長のフィールドから構成され、各フィールドには副構造を持たせることができます。例えば、電話番号を指定するフィールドは、次のように定義できます。

```
10 phoneNumber CHAR(10);
20 areaCode CHAR(3);
20 localNumber CHAR(7);
```

変数である固定レコードは、あらゆる種類の処理に使用できますが、最良の用途は、VSAM ファイル、MQSeries メッセージ・キュー、およびその他の順次ファイルに対する入出力操作です。固定レコードは、(VisualAge Generator などの以前のプロダクトのように) リレーショナル・データベースへのアクセスや汎用処理に使用できますが、新規の開発にあたっては、なるべくそのような目的に固定レコードを使用しないでください。

以下のいずれかのタイプのレコード・パーツは、固定レコード・パーツです。

- indexedRecord
- mqRecord
- relationalRecord
- serialRecord

さらに、以下のいずれかのタイプのレコード・パーツは、各フィールドの前にレベル番号が付いていれば固定レコード・パーツです。

- basicRecord
- SQLRecord

固定レコードは、以下のコンテキストで使用できます。

- データ・ソースとの間でデータをコピーする文の中
- assignment または move ステートメントの中
- 他のプログラムまたは関数にデータを渡す引数として
- プログラムまたは関数の中にデータを受け取るパラメーターとして

固定レコード・パーツと外部データ・ソースとの関係は、固定レコード・パーツのタイプと、タイプ固有の一連のプロパティ (fileName など) によって決まります。例えば、indexedRecord タイプのパーツに基づいたレコードは、VSAM キー・シーケンス・データ・セットへのアクセスに使用されます。レコード・パーツとデータ・ソースとの関係は、add などの EGL I/O ステートメントで固定レコードが使用されたときに生成される操作を決定します。

固定レコード・フィールドを別の固定レコード・パーツの基礎とすることもできます。また、assignment ステートメントでは、固定レコード・フィールドは、固定レコード・パーツ内のタイプに関係なく、CHAR 型のメモリー領域として扱われます。

関連する概念

- 139 ページの『DataItem パーツ』
- 141 ページの『レコード・パーツ』
- 『レコード・タイプとプロパティー』
- 331 ページの『リソース関連とファイル・タイプ』
- 28 ページの『固定構造体』
- 30 ページの『Typedef』

関連するタスク

- 122 ページの『デフォルトのビルド記述子の設定』
- 123 ページの『EGL エディターの設定の変更』

関連する参照項目

- 407 ページの『代入』
- 532 ページの『EGL ソース形式』
- 511 ページの『データの初期化』
- 37 ページの『プリミティブ型』

レコード・タイプとプロパティー

使用可能な EGL レコード・タイプには、以下のものがあります。

- basicRecord
- indexedRecord
- mqRecord
- relativeRecord
- serialRecord
- SQLRecord

どのターゲット・システムがどのレコード・タイプをサポートしているかについて詳しくは、『レコードとファイル・タイプの相互参照』を参照してください。レコード・パーツの初期化について詳しくは、『データの初期化』を参照してください。

basicRecord

基本レコードまたは固定基本レコードは、内部処理に使用され、データ・ストレージにはアクセスできません。

このパーツは、デフォルトではレコード・パーツですが、フィールド定義の前にレベル番号が置かれている場合は、固定レコード・パーツになります。

basicRecord タイプの固定レコード・パーツ内では、プロパティー **redefines** が使用可能です。このプロパティーは、設定された場合、宣言されたレコードを識別し、その固定レコード・パーツに基づいたすべてのレコードは、宣言されたレコードのランタイム・メモリーにアクセスします。

メインプログラムでは、プログラム・プロパティ **inputRecord** は、『データの初期化』で説明されているように、自動的に初期化されるレコード（または固定レコード）を識別します。

indexedRecord

索引付きレコードは、固定レコードの 1 つであり、これを使用すると、キー値によってアクセスされるファイルを操作できます。キー値は、ファイル内のレコードの論理的位置を示します。このファイルを読み取るには、**get**、**get next**、または **get previous** ステートメントを呼び出します。また、ファイルに書き込みを行うには、**add** ステートメント または **replace** ステートメントを呼び出します。ファイルからレコードを除去する場合は、**delete** ステートメントを呼び出します。

タイプ **indexedRecord** のパーツには、以下のプロパティがあります。

- **fileName** は必須です。入力の意味については、『リソース関連 (概説)』を参照してください。有効な文字の詳細については、『命名規則』を参照してください。
- **keyItem** は必須であり、そのレコード内で固有の構造体フィールドでなければなりません。キー・フィールドを指定するには、非修飾参照を使用する必要があります。例えば、*myRecord.myItem* ではなく、*myItem* を使用します。（ただし、EGL ステートメント内では、他のフィールドを参照するのと同じようにキー・フィールドを参照できます。）

『可変長レコードをサポートするプロパティ』も参照してください。

mqRecord

MQ レコードは固定レコードの 1 つであり、これを使用すると、MQSeries メッセージ・キューにアクセスできます。詳細については、『MQSeries のサポート』を参照してください。

relativeRecord

相対レコードは固定レコードの 1 つであり、これを使用すると、各レコードが以下のプロパティを持つデータ・セットを操作できます。

- 固定長
- ファイル内のレコードの順次位置を表す整数を使用してアクセス可能

タイプ **relativeRecord** のパーツには、以下のプロパティがあります。

- **fileName** は必須です。入力の意味については、『リソース関連 (概説)』を参照してください。有効な文字の詳細については、『命名規則』を参照してください。
- **keyItem** は必須です。キー・フィールドには、以下のメモリー領域を指定できます。
 - 同一レコード内の構造体フィールド
 - プログラムに対してグローバルであるか、またはそのレコードにアクセスする関数に対してローカルであるレコード内の構造体フィールド
 - プログラムに対してグローバルであるか、またはそのレコードにアクセスする関数に対してローカルであるプリミティブ変数

キー・フィールドを指定するには、非修飾参照を使用する必要があります。例えば、*myRecord.myItem* ではなく *myItem* を使用します。（EGL ステートメント内で

は、他のフィールドを参照するのと同じようにキー・フィールドを参照できます。) キー・フィールドは、そのレコードにアクセスする関数のローカルな有効範囲内で固有でなければなりません。または、ローカルな有効範囲には存在しないで、グローバルな有効範囲内で固有でなければなりません。

キー・フィールドには、以下の特性があります。

- BIN、DECIMAL、INT、または NUM のプリミティブ型を持つ
- 小数部を含まない
- 桁数は最大 9 桁まで

キー・フィールドを使用するのは **get** ステートメントと **add** ステートメントのみですが、ファイル・アクセス用にそのレコードを使用するすべての関数からキー・フィールドを使用することが必要です。

serialRecord

シリアル・レコードは固定レコードの 1 つであり、これを使用すると、順次にアクセスされるファイルまたはデータ・セットにアクセスできます。ファイルを読み取るには、**get** ステートメントを呼び出します。一連の **get next** では、最初のレコードから最後のレコードに向かって順次にファイル・レコードが読み取られます。**add** ステートメントを呼び出してファイルへの書き込みを実行することもできます。この文は新規レコードをファイルの末尾に追加します。

シリアル・レコードのプロパティに **fileName** が必須として含まれます。このプロパティに対する入力の意味については、『リソース関連 (概説)』を参照してください。有効な文字の詳細については、『命名規則』を参照してください。

『可変長レコードをサポートするプロパティ』も参照してください。

sqlRecord

SQL レコードは、リレーショナル・データベースにアクセスするときに、特殊なサービスを提供するレコード (または固定レコード) です。

このパーツは、デフォルトではレコード・パーツですが、フィールド定義の前にレベル番号が置かれている場合は、固定レコード・パーツになります。

各パーツには、以下のプロパティがあります。

- **tableNames** のエントリは、パーツに関連した SQL テーブルを識別します。複数のテーブルを結合参照することもできますが、単一の EGL ステートメントを使用して複数のテーブルに書き込みを行うことは制限事項により禁止されています。個々のテーブル名には、ラベル を関連付けることができます。ラベルはオプションの短い名前で、SQL ステートメント内の表を参照するために使用します。
- **defaultSelectCondition** はオプションです。この条件はデフォルト SQL ステートメントの WHERE 文節の一部になります。WHERE 文節は、SQL レコードを EGL の **open** または **get** ステートメントの中で使用するか、**get next** または **get previous** のようなステートメントの中で使用する場合に役立ちます。

たいていの場合、SQL デフォルト選択条件は第 2 条件を補足します。この第 2 条件は、SQL レコード内のキー・フィールド値と、SQL テーブルのキー列の関連付けが基礎となります。

- **tableNameVariables** はオプションです。『動的 SQL』で説明されているとおり、1 つ以上の変数を指定して、実行時に変数の内容によってアクセスするデータベース表を決定させることができます。
- **keyItems** はオプションです。各キー・フィールドは、同一レコード内で固有の構造体フィールドでなければなりません。これらの各フィールドを指定するには、非修飾参照を使用する必要があります。例えば、*myRecord.myItem* ではなく、*myItem* を使用します。(ただし、EGL ステートメント内では、他のフィールドを参照するのと同じようにキー・フィールドを参照できます。)

詳細については、『SQL サポート』を参照してください。

関連する概念

- 142 ページの『固定レコード・パーツ』
- 259 ページの『動的 SQL』
- 285 ページの『MQSeries のサポート』
- 141 ページの『レコード・パーツ』
- 331 ページの『リソース関連とファイル・タイプ』
- 247 ページの『SQL サポート』

関連する参照項目

- 605 ページの『add』
- 613 ページの『close』
- 511 ページの『データの初期化』
- 617 ページの『delete』
- 619 ページの『execute』
- 631 ページの『get』
- 644 ページの『get next』
- 650 ページの『get previous』
- 717 ページの『MQ レコードのプロパティ』
- 725 ページの『命名規則』
- 664 ページの『open』
- 679 ページの『prepare』
- 793 ページの『可変長レコードをサポートするプロパティ』
- 793 ページの『レコードとファイル・タイプの相互参照』
- 681 ページの『replace』
- 71 ページの『SQL 項目のプロパティ』
- 1004 ページの『terminalID』

EGL プログラム・パーツの作成

EGL プログラム・パーツは、Java プログラム、Java ラッパー、または Enterprise JavaBean セッション Bean の生成に使用される主な論理単位です。詳細については、『プログラム・パーツ』を参照してください。

プログラム・ファイルをワークベンチで作成すると、プログラム・パーツはこのファイルに自動的に追加され、適切な名前が付けられます。プログラム・ファイル指定は、1 つのファイルに対して 1 つのプログラム・パーツのみが可能であり、ファイル名と一致するプログラム名が必要です。

プログラム・パーツを含むプログラム・ファイルを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「プログラム」をクリックする。
3. EGL ファイルを入れるプロジェクトまたはフォルダーを選択し、パッケージを選択する。プログラム名をファイル名と一致させるため、EGL パーツ名規則に沿ったファイル名を選択する。「EGL ソース・ファイル名」フィールドに、EGL ファイルの名前を入力する (例: myEGLprg)。EGL プログラム・タイプを選択します (詳細については、『EGL ソース形式の基本プログラム』または『EGL ソース形式の TextUI プログラム』を参照)。プログラム・パーツがメインプログラムである場合は、クリックして、「呼び出し先プログラムとして作成」のチェック・マークを外します。
4. 「完了」ボタンをクリックする。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

1 ページの『EGL の紹介』

『プログラム・パーツ』

関連するタスク

134 ページの『EGL ソース・フォルダーの作成』

関連する参照項目

784 ページの『EGL ソース形式の基本プログラム』

135 ページの『EGL ソース・ファイルの作成』

725 ページの『命名規則』

786 ページの『EGL ソース形式のテキスト UI プログラム』

プログラム・パーツ

プログラム・パーツ は、ランタイム Java プログラム内の中央論理装置を定義します。メイン・プログラムと呼び出し先プログラム、およびプログラム・タイプ (基本、および textUI) の概要については、『パーツ』を参照してください。

すべての種類のプログラム・パーツは、*main* という関数を含みます。この関数は、プログラムの開始時に実行するロジックを表します。プログラムには、他の関数を含めることができ、プログラムの外部に関数をアクセスできます。関数 *main* は、このような他の関数、および他のプログラムに制御を渡すことができるすべての関数を呼び出すことができます。

最も重要なプログラム・プロパティは、次のとおりです。

- 各パラメーターは、呼び出し元から受信したデータを含むメモリ領域を参照します。パラメーターは、プログラムに対してグローバルで、呼び出し先プログラムでのみ有効です。
- 各変数は、プログラム内でグローバルに割り振られているメモリ領域を参照します。

- 書式グループは、以下のように、データをユーザーに対して表示する書式の集合です。
 - 基本プログラムは、印刷書式経由でプリンターでデータを表記できます。
 - テキスト・プログラムは、対話式で (テキスト書式経由で) データを表示するか、またはプリンターに表示します。

詳細については、『*FormGroup* パーツ』を参照してください。

- 入力レコードは、制御が他のプログラムから非同期で転送されたときにデータを受信するグローバル・メモリー領域です。入力レコードは、メイン・プログラムでのみ利用可能です。
- メイン・テキスト・プログラムでは、セグメント化されたプロパティは、プログラムがテキスト書式を表示するために対話を発行する前に、自動的に実行されるアクションを決定します。詳細については、『セグメンテーション』を参照してください。
- またテキスト・プログラムでは、プログラム開始時の入力書式に次の 2 つのうちいずれかの目的があります。
 - 書式は、モニターまたはターミナルからプログラムを起動するユーザーに送信されます。
 - または、ユーザーによって入力されたデータがその入力書式に受信され、プログラム自体の中でメモリー領域となる。この状況は、据え置きプログラム交換の場合にのみ適用されます。これは、**show** ステートメントのバリエーションによって発生するコントロールのツー・ステップ転送です。
 1. プログラムは、ユーザーにテキスト書式を処理依頼後に終了します
 2. ユーザーは、書式を処理依頼し、書式内の情報に基づいて、その処理依頼は自動的に 2 番目のプログラムを起動します。この 2 番目のプログラムには、入力書式が組み込まれています。

プログラム・プロパティの完全なリストについては、『プログラム・パーツ・プロパティ』を参照してください。

関連する概念

- 163 ページの『*FormGroup* パーツ』
- 149 ページの『関数パーツ』
- 19 ページの『パーツ』
- 62 ページの『EGL での変数の参照』
- 170 ページの『テキスト・アプリケーションのセグメンテーション』

関連するタスク

- 146 ページの『EGL プログラム・パーツの作成』

関連する参照項目

- 524 ページの『EGL でのコンテンツ・アシスト』
- 511 ページの『データの初期化』
- 532 ページの『EGL ソース形式』
- 93 ページの『EGL ステートメント』
- 783 ページの『EGL ソース形式のプログラム・パーツ』
- 789 ページの『プログラム・パーツ・プロパティ』

EGL 関数パーツの作成

関数パーツは、プログラムの最初のコードが含まれているか、別の関数から呼び出される論理単位です。EGL 関数パーツは、EGL ファイルに入っています。EGL 関数パーツを作成するには、次のようにします。

1. 関数パーツを入れる EGL ファイルを識別し、そのファイルを EGL エディターで開く。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
2. EGL 構文に従って、関数パーツの特性を入力する (詳細については、『EGL ソース形式の関数パーツ』を参照してください)。コンテンツ・アシストを使用して、関数パーツの構文のアウトラインをファイルに入れることができます。
3. EGL ファイルを保管する。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
『関数パーツ』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』
135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

関連する参照項目

524 ページの『EGL でのコンテンツ・アシスト』
560 ページの『関数呼び出し』
570 ページの『EGL ソース形式の関数パーツ』
725 ページの『命名規則』

関数パーツ

関数パーツ は、プログラムの最初のコードを含んでいるか、または他の関数から呼び出される論理単位です。プログラムの最初のコードを含んでいる関数は、*main* と呼ばれます。

関数パーツには、以下のプロパティを含むことができます。

- 戻り値。関数パーツが呼び出し元に戻すデータを表します。
- パラメーターのセット。それぞれのパラメーターにより、他のロジック・パーツによって割り振られて渡されるメモリーが参照されます。
- 他の変数のセット。それぞれの変数により、関数でローカルな他のメモリーが割り振られます。
- EGL ステートメント
- *containerContextDependent* で説明されているとおり、関数がプログラム・コンテキストを必要としているかどうかの指定

関数 *main* は、値を戻すことができない、パラメーターを含むことができない、そしてプログラム・パーツの内部で宣言しなければならないという点で独特の関数です。

関連する概念

19 ページの『パーツ』

147 ページの『プログラム・パーツ』
62 ページの『EGL での変数の参照』
247 ページの『SQL サポート』

関連する参照項目

505 ページの『containerContextDependent』
511 ページの『データの初期化』
532 ページの『EGL ソース形式』
560 ページの『関数呼び出し』
570 ページの『EGL ソース形式の関数パーツ』
93 ページの『EGL ステートメント』

EGL ライブラリー・パーツの作成

EGL ライブラリー・パーツには、プログラム、ページ・ハンドラー、またはその他のライブラリーで使用される関数、変数、および定数のセットが含まれています。

EGL ライブラリー・パーツを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「ライブラリー」をクリックする。
3. EGL ファイルを入れるプロジェクトまたはフォルダーを選択し、パッケージを選択する。ライブラリー名をファイル名と一致させるため、EGL パーツ名規則に沿ったファイル名を選択する。「EGL ソース・ファイル名」フィールドに、EGL ファイルの名前を入力する (例: myLibrary)。
4. 次のラジオ・ボタンのどちらかをクリックして、ライブラリーのタイプを選択する。
 - 基本 - 基本ライブラリーを作成する (Basic - Create a basic library)
 - ネイティブ - ネイティブ・ライブラリーを作成する (Native - Create a native library)
5. 「完了」 ボタンをクリックする。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
1 ページの『EGL の紹介』
151 ページの『basicLibrary タイプのライブラリー・パーツ』
151 ページの『basicLibrary タイプのライブラリー・パーツ』

関連するタスク

134 ページの『EGL ソース・フォルダーの作成』
132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

135 ページの『EGL ソース・ファイルの作成』
700 ページの『EGL ソース形式のライブラリー・パーツ』
725 ページの『命名規則』

basicLibrary タイプのライブラリー・パーツ

basicLibrary タイプのライブラリー・パーツには、プログラム、ページ・ハンドラー、または他のライブラリーから使用できる一連の関数、変数、および定数が格納されます。ライブラリーを使用して、共通コードおよび値を、最大限再利用することをお勧めします。

タイプ指定 *basicLibrary* は、そのパーツがコンパイル可能単位内に生成され、ローカル実行用の EGL の値とコードを含んでいることを示しています。このタイプは、キーワード **type** が指定されなかったときのデフォルトです。EGL で生成された Java プログラムからネイティブ DLL にアクセスするためのライブラリーを作成する方法の詳細については、『*nativeLibrary* タイプのライブラリー・パーツ』を参照してください。

basicLibrary タイプのライブラリーに関する規則は、以下のとおりです。

- プログラム固有の Use 宣言にライブラリーを組み込む場合のみ、ライブラリー名を指定せずに、ライブラリーの関数、変数、および定数を参照することができます。
- ライブラリー関数は、呼び出し元のプログラムまたはページ・ハンドラーと関連する任意のシステム変数にアクセスすることができます。以下の規則が適用されます。
 - ライブラリー内の関数がレコードを引数として受け取る場合、そのレコードを入力または出力 (I/O) に使用したり、endOfFile などの I/O 状態のテストに使用したりすることはできません。しかし、ライブラリーを呼び出すコードでは、その両方にレコードを使用できます。
 - ライブラリー内のレコードを宣言した場合、ライブラリー・ベースの関数でそのレコードを入力または出力 (I/O) に使用するか、I/O 状態のテスト (例えば、ファイルの終わりであるかどうかのテスト) に使用できます。しかし、ライブラリーを呼び出すコードでは、そのどちらにもレコードを使用できません。
- ライブラリー関数は、以下を除くすべての文を使用できます。
 - converse
 - forward
 - show
 - transfer
- ライブラリーはテキスト書式にアクセスできません。
- テキストまたは印刷書式にアクセスするライブラリーには、関連する書式グループの使用宣言を組み込む必要があります。
- 関数、変数、または定数宣言で修飾子 **private** を使用し、エレメントがライブラリー外部で使用されることのないようにできます。
- **public** として宣言された (デフォルトと同じ) ライブラリー関数は、ライブラリーの外から使用可能で、*loose* タイプのパラメーターを持つことはできません。これは、特別な種類のプリミティブ・タイプで、パラメーターが一定の範囲の引数の長さを受け入れる場合にのみ使用できます。*loose* タイプの詳細については、『EGL ソース形式の関数パーツ』を参照してください。

ライブラリーは、そのライブラリーを使用するパーツとは別に生成されます。EGL ランタイムは、ライブラリー・パーツにアクセスする場合、ライブラリー・プロパティ **alias** の設定値を使用します。このプロパティのデフォルト値は、EGL ライブラリー名です。

実行時には、ライブラリーは最初に使用されるときにロードされ、そのライブラリーにアクセスしたプログラムまたはページ・ハンドラーが (実行単位が終了したときのように) メモリーから退去したときにアンロードされます。

ページ・ハンドラーは、ロードされるたびにライブラリーの新しいコピーを取得します。また、別のライブラリーによって呼び出されたライブラリーは、呼び出し元のライブラリーが存続している間はメモリー内に存続します。

定数はそれを参照するプログラムおよびページ・ハンドラー内でリテラルとして生成されるため、定数専用のライブラリーは、実行時にロードされません。

関連する概念

- 629 ページの『forward』
- 570 ページの『EGL ソース形式の関数パーツ』
- 700 ページの『EGL ソース形式のライブラリー・パーツ』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 798 ページの『実行単位』
- 170 ページの『テキスト・アプリケーションのセグメンテーション』
- 696 ページの『show』
- 697 ページの『transfer』
- 1020 ページの『使用宣言』

関連する参照項目

- 616 ページの『converse』
- 629 ページの『forward』
- 570 ページの『EGL ソース形式の関数パーツ』
- 700 ページの『EGL ソース形式のライブラリー・パーツ』
- 798 ページの『実行単位』
- 170 ページの『テキスト・アプリケーションのセグメンテーション』
- 696 ページの『show』
- 697 ページの『transfer』
- 1020 ページの『使用宣言』

nativeLibrary タイプのライブラリー・パーツ

nativeLibrary タイプのライブラリーを使用すると、EGL 生成 Java コードで単一のローカル実行 DLL を呼び出すことができます。その DLL 用のコードは、EGL 言語では書かれていません。EGL 言語で書かれている 共用の関数と値が入っている基本ライブラリーの開発方法についての情報は、『basicLibrary タイプのライブラリー・パーツ』を参照してください。

nativeLibrary タイプのライブラリーでは、各関数の目的は DLL 関数へのインターフェースを提供することです。EGL 関数内で文を定義することはできず、ライブラリー内の任意の場所に変数や定数を宣言することもできません。

EGL ランタイムは、DLL ベースの関数にアクセスする場合、EGL 関数プロパティ **alias** の設定値を使用します。このプロパティのデフォルト値は、EGL 関数名です。DLL ベースの関数が『命名規則』で説明されている規則に準拠していない場合は、このプロパティを明示的に設定してください。

ライブラリー・プロパティ **callingConvention** は、EGL ランタイムが次の 2 種類のコード間でデータを受け渡す方法を指定します。

- ライブラリー関数を呼び出す EGL コード
- DLL 内の関数

現時点で **callingConvention** に使用可能な値は、*I4GL* のみです。つまり、

- データは、Informix スタック形式に従って渡されます。各入力パラメーターは、入力スタック上に置かれ、各出力パラメーターは出力スタック上に置かれます。
- レコードや辞書などの引数を渡すことはできません。以下の値のみが有効です。
 - プリミティブ変数 (ANY 型の変数を含む)
 - dataTable、印刷書式、テキスト書式、および固定レコード内のフィールド (ただし、そのフィールドに副構造がない場合に限る)

ライブラリー関数内のパラメーターは、プリミティブ変数でなければならず、ANY 型にすることができますが、ルーズ型にすることはできず、**field** 修飾子を組み込むこともできません。

ライブラリー・プロパティ **dllName** は、最終的な DLL 名を指定します。これをデプロイメント時にオーバーライドすることはできません。ライブラリー・プロパティ **dllName** の値を指定しない場合は、DLL 名を Java ランタイム・プロパティ `vgj.defaultI4GLNativeLibrary` の中で指定する必要があります。そのような Java ランタイム・プロパティは、1 つの実行単位に 1 つだけ使用できます。したがって、EGL ライブラリー内で識別される DLL のほかに、1 つだけ DLL を指定できます。

DLL 名を開発時に (**dllName** で) 指定する場合でも、デプロイメント時に (`vgj.defaultI4GLNativeLibrary` で) 指定する場合でも、その DLL は、ランタイム変数内で識別されるディレクトリー・パスに存在する必要があります。その変数は、PATH (Windows 2000/NT/XP の場合) または LIBPATH (UNIX プラットフォームの場合) です。

ライブラリー関数は、ライブラリーの外部からも使用できるよう、自動的に **public** として宣言されます。他の EGL コードの中では、ライブラリー名を指定せずに、関数の別名のみで関数を参照できますが、それは、そのライブラリーをプログラム固有の使用宣言に組み込んだ場合に限られます。

EGL ライブラリーは、そのライブラリーにアクセスするコードや DLL とは別個の Java クラスとして生成されます。EGL ランタイムは、そのクラスにアクセスする場合、ライブラリー・プロパティ **alias** の設定値を使用します。このプロパティのデフォルト値は、EGL ライブラリー名です。ライブラリー・パーツの名前が Java の規則に準拠していない場合は、このプロパティを明示的に設定してください。

実行時には、DLL が、最初に使用されるときにロードされ、その DLL にアクセスしたプログラムまたはページ・ハンドラーが (実行単位が終了したときのように) メモリーから退去したときに、アンロードされます。

ページ・ハンドラーは、ロードされるたびに DLL の新しいコピーを取得します。また、basicLibrary タイプの EGL ライブラリーによって呼び出された DLL は、呼び出し元のライブラリーが存続している間はメモリー内に存続します。

次のネイティブ・ライブラリーは、C で書かれた DLL へのアクセスを提供します。

```
Library myLibrary type nativeLibrary
{callingConvention="I4GL", dllname="mydll"}

Function entryPoint1( p1 int nullable in,
                      p2 date in, p3 time in,
                      p4 interval in, p5 any out)
end

Function entryPoint2( p1 float in,
                      p2 String in,
                      p3 smallint out)
end

Function entryPoint3( p1 any in,
                      p2 any in,
                      p3 any out,
                      p4 CLOB inout)
end
end
```

関連する概念

377 ページの『Java ランタイム・プロパティー』

151 ページの『basicLibrary タイプのライブラリー・パーツ』

関連する参照項目

570 ページの『EGL ソース形式の関数パーツ』

584 ページの『Java ランタイム・プロパティー (詳細)』

700 ページの『EGL ソース形式のライブラリー・パーツ』

725 ページの『命名規則』

798 ページの『実行単位』

1020 ページの『使用宣言』

EGL dataTable パーツの作成

EGL dataTable パーツは、データ構造と構造の初期値の配列を関連付けます。EGL dataTable パーツを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「データ・テーブル」をクリックする。
3. EGL ファイルを入れるプロジェクトまたはフォルダーを選択し、パッケージを選択する。dataTable 名をファイル名と一致させるため、EGL パーツ名規則に沿ったファイル名を選択する。「EGL ソース・ファイル名」フィールドに、

EGL ファイルの名前を 入力する (例: myDataTable)。 dataTable サブタイプを選択します (詳細については、『EGL ソース形式の DataTable パーツ』を参照)。

4. 「完了」ボタンをクリックする。

関連する概念

『DataTable』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

1 ページの『EGL の紹介』

関連するタスク

134 ページの『EGL ソース・フォルダーの作成』

132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

135 ページの『EGL ソース・ファイルの作成』

513 ページの『EGL ソース形式の DataTable パーツ』

725 ページの『命名規則』

DataTable

EGL *dataTable* は、主に以下のコンポーネントで構成されます。

- 列を定義する各最上位項目のある構造体
- それらの列と整合した値の配列。列の各エレメントは、行を定義します。

エラー・メッセージの *dataTable* など。以下のようなコンポーネントが含まれています。

- 数値フィールドおよび文字フィールドの宣言
- 次のようなペア値のリスト --

```
001   Error 1
002   Error 2
003   Error 3
```

レコードまたはデータ項目を宣言する場合と同じように、*dataTable* を宣言しないでください。その代わりに、*dataTable* にアクセスできるコードは、そのパーツを変数として扱うことができます。パーツ・アクセスについては、『パーツの参照』を参照してください。

dataTable にアクセスできるコードには、使用宣言内のパーツ名を参照するオプションがあります。

dataTable のタイプ

一部の *dataTable* はランタイム妥当性検査用のもので、特に書式入力との比較用にデータを保持します。(書式パーツの宣言時に、データ・テーブルと入力フィールドを関連付けます。) 以下に示す 3 つのタイプの妥当性検査 *dataTable* が使用可能です。

matchValidTable

ユーザーの入力は、*dataTable* の最初の列の値と一致している必要があります。

matchInvalidTable

ユーザーの入力は、dataTable の最初の列のどの値とも異なっている必要があります。

rangeChkTable

ユーザーの入力は、少なくとも 1 つの dataTable の行の 1 列目と 2 列目の値の間の値に一致している必要があります。(範囲は包括的です。ユーザーの入力が任意の行の最初の列または 2 番目の列の値と一致している場合は、有効です。)

その他のタイプの dataTable は、以下のとおりです。

msgTable

ランタイム・メッセージが含まれています。

basicTable

プログラム・ロジックで使用されるその他の情報を含んでいます。例えば、国と関連するコードのリストなどです。

dataTable の生成

dataTable の生成からの出力は、それぞれ dataTable の名前が付いた 1 対のファイルです。ファイルの拡張子は一方が .java、もう一方が .tab です。.tab ファイルは Java コンパイラーによって処理されませんが、パッケージを含むディレクトリ構造のルートに組み込まれます。例えば、パッケージが *my.product.package* の場合は、ディレクトリ構造は *my/product/package* で、.tab ファイルは、サブディレクトリ *my* が含まれているディレクトリにあります。

以前に同じ dataTable を生成済みのパッケージに生成する場合は、dataTable を生成する必要はありません。

dataTable を生成する必要がない場合は、生成時間を省略するため、ビルド記述子オプション **genTables** に NO を割り当てます。

dataTable のプロパティ

以下のプロパティを設定できます。

- **alias** は、生成される出力の名前に取り込まれます。別名を指定しなかった場合は、パーツ名が代わりに使用されます。
- **shared** プロパティは、複数のユーザーが dataTable にアクセスできるかどうかを指示します。デフォルトは *no* です。
- **resident** プロパティは、その dataTable を使用しているプログラムがない場合でも、dataTable をメモリー内に残すかどうかを指示します。(プログラムは最初のアクセスがあったときにメモリーに入ります。)デフォルトは *no* です。 *yes* は、共用の指定を *yes* の場合にのみ指定することもできます。

関連する概念

24 ページの『パーツの参照』

関連する参照項目

513 ページの『EGL ソース形式の DataTable パーツ』

1020 ページの『使用宣言』

EGL および JSP ファイルへのコードの断片の挿入

「断片」ビューでは、再使用可能なプログラミング・オブジェクトをコードに挿入できます。「断片」ビューには、EGL コードの複数の断片の他に、それ以外の多くのテクノロジーのコードも含まれています。用意されている断片を使用するか、独自の断片を「断片」ビューに追加することができます。「断片」ビューの使用の詳細については、『「断片」ビュー』を参照してください。

ご使用のコードに EGL コード断片を挿入するには、次のようにします。

1. 断片の追加先としたいファイルを開く。
2. 「断片」ビューを開く。
 - a. 「ウィンドウ」>「ビューの表示」>「その他」をクリックする。
 - b. 「基本」を展開し、「断片」をクリックする。
 - c. 「OK」をクリックする。
3. 「断片」ビューで、**EGL** ドロワーを展開する。このドロワーには、使用可能な EGL コードの断片が入っています。
4. 次のメソッドのどちらかを使用して、ファイルに断片を挿入する。
 - 断片をクリックして、ソース・コードにドラッグする。
 - 断片をダブルクリックして、現在のカーソル位置にその断片を挿入する。その断片内の変数と値を説明するウィンドウが表示される場合があります。その場合は、「挿入」をクリックしてください。

注: カーソルが、打ち消し線のある円 (そのポイントに断片を挿入できないことを示す) に変わる場合、断片を誤った位置に挿入しようとしている可能性があります。断片の詳細を検査して、断片をコードに挿入する位置を検出してください。

5. ご使用のコードに応じて適宜、断片内の関数、変数、およびデータ・パーツの事前定義名を変更する。大部分の断片には、変更の必要がある名前を説明するコメントが含まれています。

EGL で使用可能な断片は、次のとおりです。

表 6. EGL で使用可能な断片

断片の名前	説明
setCursorFocus	Web ページ上の指定された書式フィールドにカーソルのフォーカスを設定する JavaScript [™] 関数。
autoRedirect	セッション変数の有無についてテストする JavaScript 関数。セッション変数が存在しない場合は、ブラウザを別のページに転送します。
getClickedRowValue	データ・テーブル内のクリックされた行でハイパーリンクされた値を検索する EGL 関数。

表 6. EGL で使用可能な断片 (続き)

断片の名前	説明
databaseUpdate	ページ・ハンドラーからレコードを渡したときにリレーショナル・テーブルの単一行を更新する EGL 関数。

関連する概念

「断片」ビュー

関連するタスク

- 135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』
- 『書式フィールドへのフォーカスの設定』
- 『セッション変数の有無についてのブラウザーのテスト』
- 159 ページの『データ・テーブル内のクリックされた行の値の検索』
- 160 ページの『リレーショナル・テーブル内の行の更新』

関連する参照項目

書式フィールドへのフォーカスの設定

「断片」ビューの JSP ドロワー内の `setCursorFocus` 断片は、Web ページ上の指定された書式フィールドにカーソルのフォーカスを設定する JavaScript 関数です。JSP ページの `<script>` タグ内に置かれなければなりません。この断片を挿入し構成する手順は、次のとおりです。

1. 断片コードをページのソース・コードに挿入する。詳細については、『*EGL コード断片の挿入*』を参照してください。
2. `[n]` を、フォーカスを受け取る書式フィールドの番号で置き換える。たとえば、ページ上の 4 番目のフィールドにフォーカスを設定する場合には、`[3]` を使用します。
3. 書式名を `form1` に設定する。
4. JSP ページの `<body>` タグを `<body onload="setfocus();">` に変更する。

この断片によって挿入されるコードは、次のとおりです。

```
function setFocus(){
    document.getElementById('form1').elements[n].select();
    document.getElementById('form1').elements[n].focus();
}
```

関連するタスク

- 157 ページの『EGL および JSP ファイルへのコードの断片の挿入』

セッション変数の有無についてのブラウザーのテスト

「断片」ビューの JSP ドロワー内の `autoRedirect` 断片は、セッション変数の有無をテストします。セッション変数が存在しない場合は、ブラウザーを別のページに転送します。この断片は、`<pageEncoding>` タグの後で、JSP ページの `<head>` タグ内に置かれなければなりません。この断片を挿入し構成する手順は、次のとおりです。

1. <pageEncoding> タグの後でページの <head> タグに断片コードを挿入する。詳細については、『EGL コード断片の挿入』を参照してください。
2. {SessionAttribute} を、テスト対象のセッション変数の名前で置き換える。
3. {ApplicationName} を、プロジェクトまたはアプリケーションの名前で置き換える。
4. {PageName} を、セッション変数が存在しない場合にブラウザがリダイレクトされる先のページの名前で置き換える。

この断片によって挿入されるコードは、次のとおりです。

```
<%
if ((session.getAttribute("userID") == null ))
{
    String redirectURL =
    "http://localhost:9080/EGLWeb/faces/Login.jsp";
    response.sendRedirect(redirectURL);
}
%>
```

関連するタスク

157 ページの『EGL および JSP ファイルへのコードの断片の挿入』

データ・テーブル内のクリックされた行の値の検索

「断片」ビューの EGL ドロワー内の `getClickedRowValue` 断片は、データ・テーブル内のクリックされた行でハイパーリンクされた値を検索する関数です。この断片は、EGL ページ・ハンドラーに置かれなければなりません。この断片には、次の前提条件があります。

1. JSP ページにはデータ・テーブルがある。
2. JSP ID の名前は、デフォルトから変更されていない。
3. ページは、セッションではなく、`faces-config.xml` 内の有効範囲の要求として定義される。

この断片を挿入し構成する手順は、次のとおりです。

1. 断片コードをページ・ハンドラーに挿入する。詳細については、『EGL コード断片の挿入』を参照してください。
2. クリックされた値を受け取る文字またはストリング変数を定義する。
3. コマンド・ハイパーリンク (パレット・ビューの Faces コンポーネント・ドロワーから) をデータ・テーブル内のフィールドに追加する。
4. コマンド・ハイパーリンクのターゲットに、JSP ページの名前を指定する。このハイパーリンクは、独自のページにリンクします。
5. ハイパーリンクにパラメーターを追加して、クリックされた値を受け取るページ・ハンドラー内の変数と同じ名前をそのパラメーターに付ける。
6. `action` プロパティ (「プロパティ」ビューの「すべて」タブにある) を `getVal()` 関数に設定する。

この断片によって挿入されるコードは、次のとおりです。

```
function getVal()
    javaLib.store((objId)"context",
    "javax.faces.context.FacesContext",
```

```

    "getCurrentInstance");
javaLib.store((objId)"root",
    (objId)"context", "getViewRoot");
javaLib.store((objId)"parm",
    (objId)"root",
    "findComponent",
    "form1:table1:param1");
recVar = javaLib.invoke((objId)"parm",
    "getValue");
end

```

関連するタスク

157 ページの『EGL および JSP ファイルへのコードの断片の挿入』

リレーショナル・テーブル内の行の更新

「断片」ビューの EGL ドロワー内の databaseUpdate 断片は、ページ・ハンドラーからレコードを渡したときに、リレーショナル・テーブルの単一行を更新する関数です。この断片は、EGL ライブラリーに置かれるよう意図されたものです。この断片を挿入し構成する手順は、次のとおりです。

1. 断片コードをページ・ハンドラーに挿入する。詳細については、『EGL コード断片の挿入』を参照してください。
2. {tableName} および {keyColumn} を、表の名前とその主キー列で置き換える。

この断片によって挿入されるコードは、次のとおりです。

```

Function updateRec(${TableName}New ${TableName})

    // Function name - call this function
    // passing the ${TableName} Record as a parameter
    ${TableName}Old ${TableName};

    // A copy of the Record, used
    // to lock the table row and to obtain
    // the existing row values prior to update try
    ${TableName}Old.${KeyColumn} =
        ${TableName}New.${KeyColumn};
    get ${TableName}Old forUpdate;

    // Get the existing row.
    // Note that if you had custom processing to do,
    // you would insert your code after this call
    move ${TableName}New to ${TableName}Old byName;

    //Move the updated values to the copy-row
    replace ${TableName}Old;

    //And replace the row in the database.
    sysLib.commit();

    //Commit your changes to the Database
    onException
        //If the update fails...
        sysLib.rollback();

        // cancel all database updates
        // (assuming this is permitted
        // by your database) and call
        // a custom error handling routine
    end
end

```

関連するタスク

157 ページの『EGL および JSP ファイルへのコードの断片の挿入』

テキスト書式と印刷書式による作業

EGL formGroup パーツの作成

EGL formGroup パーツは、テキスト書式と印刷書式のコレクションを定義します。
EGL formGroup パーツを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「書式グループ」をクリックする。
3. EGL ファイルを入れるプロジェクトまたはフォルダーを選択し、パッケージを選択する。 formGroup 名 をファイル名と一致させるため、EGL パーツ名規則に沿ったファイル名を選択する。「EGL ソース・ファイル名」フィールドに、EGL ファイルの名前を入力する (例: myFormGroup)。
4. 「完了」ボタンをクリックする。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
174 ページの『EGL 書式エディターの概要』
175 ページの『EGL 書式エディターを使用した書式グループの編集』
『FormGroup パーツ』
1 ページの『EGL の紹介』

関連するタスク

134 ページの『EGL ソース・フォルダーの作成』
132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

135 ページの『EGL ソース・ファイルの作成』
553 ページの『EGL ソース形式の書式パーツ』
725 ページの『命名規則』

FormGroup パーツ

EGL の FormGroup パーツは、次の 2 つの目的に役立ちます。

- テキスト書式および印刷書式のコレクションを定義する。(パーツに固有の書式は、そのパーツ内に定義されるか、Use 宣言を介して組み込まれます。複数の FormGroup パーツに共通の書式は、Use 宣言を介して組み込まれます。)
- 『書式パーツ』で説明しているように、多くの浮動域にゼロを定義する

レコードまたはデータ項目を宣言する場合と同じように、書式グループを宣言しないでください。その代わりに、以下の記述が適用される場合に限り、プログラムは FormGroup パーツ (および関連する書式) にアクセスします。

- 『パーツへの参照』で説明しているように、プログラムから FormGroup パーツのロケーションにアクセスすることができる。

- プログラム内の Use 宣言が、FormGroup パーツを参照している。

プログラムには、最大で 2 つの formGroup パーツを組み込むことができます。2 つのパーツが指定されている場合は、その内の 1 つはヘルプ・グループである必要があります。ヘルプ・グループには、1 つ以上のヘルプ書式が含まれます。ヘルプ書式は、ユーザーのキー・ストロークに対する応答として情報を提供する読み取り専用書式です。

実行時に書式が使用可能になるのは、FormGroup を生成した場合だけです。生成された出力は、FormGroup パーツ用のクラス、および各 Form パーツ用のクラスです。

準備段階では、それらのエンティティはそれぞれ、独立したランタイム・ロード・モジュール内に生成されます。EGL のランタイムは、生成されたプログラムと書式固有のコードとの対話を処理します。

書式パーツを個別に生成することはできません。

関連する概念

『書式パーツ』

24 ページの『パーツの参照』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連する参照項目

1020 ページの『使用宣言』

書式パーツ

書式パーツ は、プレゼンテーションの単位です。フォーム・パーツは、ユーザーに一度に表示されるフィールドのセットのレイアウトおよび特性を記述するものです。

レコードまたはデータ項目を宣言する場合と同じように、書式を宣言しないでください。書式パーツにアクセスするには、関連する書式グループを参照する使用宣言をプログラムに組み込む必要があります。

書式パーツは、テキストまたは印刷という 2 つのタイプのいずれかになります。

- テキスト・タイプの書式は、コマンド・ウィンドウに表示されるレイアウトを定義します。例外が 1 つありますが、どのテキスト書式も、ユーザー入力を受け入れる変数フィールドを含む、定数フィールドおよび変数フィールドの両方を持つことができます。例外はヘルプ書式です。これは、定期情報を提供するためだけのものです。
- 印刷タイプの書式は、プリンターに送られるレイアウトを定義します。どの印刷書式も、定数フィールドおよび変数フィールドの両フィールドを持つことができます。

書式のプロパティにより、画面またはページにおける出力のサイズおよび位置が決まり、その出力のフォーマット設定特性が指定されます。

ある与えられた書式は、1 つ以上の装置に表示できます。それぞれの装置は、出力用周辺装置であるか、または動作上、出力用周辺装置と等価な装置です。

- 画面装置とは、端末、モニター、または端末エミュレーターです。この場合の出力面は画面です。
- 印刷装置とは、プリンターに送ることのできるファイル、またはプリンターそれ自体です。この場合の出力面はページです。

テキストまたは印刷 のいずれのタイプにせよ、書式はさらに以下のように分類されます。

- 固定書式は、装置の出力面に関する特定の開始行および開始列を持っています。固定の印刷書式は、例えば、あるページの第 10 行、第 1 列から始まるように割り当てることができます。
- 浮動書式は特定の開始行または開始列を持っていませんが、その代わりに、宣言した出力面のサブエリアの、次の未占有行に配置されます。宣言済みのサブエリアは、浮動域と呼ばれます。

浮動域は、第 10 行から始まって第 20 行まで広がり、出力装置の最大幅を持つ長方形である、と宣言することができます。同じ幅の 1 行の浮動書式がある場合は、20 回の間毎回、以下の振る舞いをするループを構成することができます。

1. データを浮動マップに配置する
2. 浮動マップを、浮動域の次の行に書き込む

FormGroup パーツでは 1 つ以上の浮動域が宣言されますが、特定の装置に対して浮動書式を受け入れることのできる浮動域は 1 つだけです。浮動域がないのに浮動書式を提供しようとする、出力面全体が浮動域として扱われます。

- 部分書式は、特定の装置の標準サイズの出力面よりも小さな書式です。部分書式は、複数の書式がそれぞれ異なる水平位置に表示されるよう宣言し、配置することができます。部分書式には、開始列および終了列を指定することができますが、互いに隣り合う書式を表示することはできません。

これ以上は、書式タイプに固有の詳細となります。

- 印刷書式
- テキスト書式

関連する概念

- 166 ページの『印刷書式』
- 168 ページの『テキスト書式』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 177 ページの『EGL 書式エディターを使用した書式の作成』

関連する参照項目

- 550 ページの『EGL ソース形式の FormGroup パーツ』
- 553 ページの『EGL ソース形式の書式パーツ』

EGL 印刷書式の作成

印刷書式は、プリンターへ送るレイアウトを定義する EGL 書式パーツです。EGL 印刷書式を作成するには、次のようにします。

1. 印刷書式を入れる EGL ファイルを識別し、そのファイルを EGL エディターで開く。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
2. EGL 構文に従って印刷書式の入力する (詳細については、『EGL ソース形式の書式パーツ』を参照してください)。コンテンツ・アシストを使用して、書式パーツの構文のアウトラインをファイルに入れることができます。
3. EGL ファイルを保管する。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

164 ページの『書式パーツ』

『印刷書式』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

177 ページの『EGL 書式エディターを使用した書式の作成』

関連する参照項目

524 ページの『EGL でのコンテンツ・アシスト』

553 ページの『EGL ソース形式の書式パーツ』

725 ページの『命名規則』

印刷書式

書式とそのタイプについては、『書式パーツ』で説明しています。このページでは、印刷書式を表示する方法について概説します。

印刷プロセス: 印刷のプロセスには、2 つのステップがあります。

- まず最初に、**print** ステートメントをコード化します。それぞれの文は、ランタイム・バッファーに書式を追加します。
- 次に、EGL ランタイムによって新規ページの開始に必要な記号が追加され、バッファーに入れられているすべての書式が印刷装置に送信され、バッファーの内容が消去されます。上記のサービスは、以下のいずれかの状況への対応として行われます。
 - プログラムが、同じ印刷装置を宛先とする印刷書式で **close** ステートメントを実行する
 - プログラムが、セグメント化モードで (『セグメンテーション』参照)、**converse** ステートメントを実行する
 - プログラムが、EGL 以外の (かつ VisualAge Generator 以外の) プログラムによって呼び出され、呼び出し先プログラムが終了する
 - 実行単位のメインプログラムが終了する

マルチ書式の出力の場合は、**print** ステートメントを、書式を表示する順序で呼び出す必要があります。以下の例を考えてください。

- 出力のトップで、固定書式によって、購入する会社とオーダー番号が識別されます。

- 後続の浮動域で、同じ形式でフォーマット設定された一連の浮動書式によって、その会社のオーダーの各項目が識別されます。
- 出力のボトムで、固定書式によって、項目のリスト全体でスクロールの必要な画面数またはページ数が識別されます。

そのような出力は、それぞれ印刷書式上で操作を行う一連の **print** ステートメントをサブミットすることで達成することができます。それらの文は、以下の順序で書式を参照します。

1. トップ書式
2. 浮動書式。ループの中で繰り返し呼び出される **print** ステートメントによって表示されます。
3. ボトム書式

新規ページを開始するのに必要な記号は、さまざまな状況で挿入されますが、**print** ステートメントを発行する前にシステム関数 `ConverseLib.pageEject` を呼び出すことによって、挿入することができます。

固定書式に関する考慮事項: 以下の文は、固定書式に適用されます。

- 開始行が現在の行より大きな固定書式の印刷文を発行すると、EGL は、印刷装置を指定された行まで進めるために必要な記号を挿入します。同様に、開始行が現在の行より小さな固定書式の印刷文を発行すると、EGL は、新規ページを開始するために必要な記号を挿入します。
- 固定書式が別の固定書式の全部ではなく一部の行をオーバーレイする場合、EGL は自動的に新規ページを開始するために必要な記号を挿入し、2 番目の固定書式を新規ページに配置します。
- 固定書式が別の固定書式のすべての行をオーバーレイする場合、EGL は、バッファからのその他の出力を消去することなく、既存の書式を置換します。既存の出力を保持し、新規書式を次ページに配置するには、新規書式の **print** ステートメントを発行する前に、システム関数 `ConverseLib.pageEject` を呼び出します。

浮動書式に関する考慮事項: 浮動書式を使用している場合、以下のような間違いが起こる可能性があります。

- **print** ステートメントを発行して、浮動域よりも下に浮動書式を配置する
- 少なくとも部分的に固定書式で浮動域をオーバーレイする **print** ステートメントを発行してから、**print** ステートメントを発行して浮動域に浮動書式を追加する

どちらの場合も、その結果として EGL が、新規ページを開始するために必要な記号を挿入し、浮動書式は新規ページの浮動域の最初の行に配置されます。例えば、そのページが前述のオーダーおよび項目の出力と類似する場合、新規ページには最上部の固定書式は組み込まれません。

印刷先: EGL プログラムが **close** ステートメントを処理して印刷ファイルを表示すると、その出力はプリンターまたはデータ・セットに送信されます。宛先は、以下の 3 つの時点のいずれかで指定することができます。

- テスト時 (EGL デバッガー で説明します)
- 生成時 (リソース関連およびファイル・タイプ で説明します)
- 実行時 (システム変数 `ConverseVar.printerAssociation` との関係で説明します)

関連する概念

- 303 ページの『EGL デバッガー』
- 550 ページの『EGL ソース形式の FormGroup パーツ』
- 553 ページの『EGL ソース形式の書式パーツ』
- 164 ページの『書式パーツ』
- 331 ページの『リソース関連とファイル・タイプ』
- 170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

- 848 ページの『pageEject()』
- 988 ページの『printerAssociation』

EGL テキスト書式の作成

テキスト書式は、コマンド・ウィンドウに表示するレイアウトを定義する EGL 書式パーツです。EGL テキスト書式を作成するには、次のようにします。

1. テキスト書式を入れる EGL ファイルを識別し、そのファイルを EGL エディターで開く。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
2. EGL 構文に従ってテキスト書式の入力する (詳細については、『EGL ソース形式の書式パーツ』を参照してください)。コンテンツ・アシストを使用して、書式パーツの構文のアウトラインをファイルに入れることができます。
3. EGL ファイルを保管する。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 164 ページの『書式パーツ』
- 『テキスト書式』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

- 135 ページの『EGL ソース・ファイルの作成』
- 177 ページの『EGL 書式エディターを使用した書式の作成』
- 135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

関連する参照項目

- 524 ページの『EGL でのコンテンツ・アシスト』
- 553 ページの『EGL ソース形式の書式パーツ』
- 725 ページの『命名規則』

テキスト書式

書式とそのタイプについては、『書式パーツ』で説明しています。このページでは、テキスト書式を表示する方法について概説します。

ユーザーが単一の固定テキスト書式にアクセスするには、**converse** ステートメントで十分です。プログラムの論理の流れは、表示された書式にユーザーが応答した後でのみ続行されます。また、以下の事例のように、複数の書式から出力を構成することもできます。

- 出力のトップで、固定書式によって、購入する会社とオーダー番号が識別されます。
- 後続の浮動域で、同じ形式でフォーマット設定された一連の浮動書式によって、その会社のオーダーの各項目が識別されます。
- 出力のボトムで、固定書式によって、項目のリスト全体でスクロールの必要な画面数が識別されます。

以下の 2 つのステップが必要です。

1. まず最初に、一連の **display** ステートメントをコード化することによって、オーダーと項目の出力を構成します。それぞれの文は、ランタイム・バッファに書式を追加しますが、データは画面に表示されません。各 **display** ステートメントは、以下の書式のいずれかで操作を行います。
 - トップ書式
 - 浮動書式。ループの中で繰り返し呼び出される **display** ステートメントによって表示されます。
 - ボトム書式
2. 次に、EGL ランタイムによって、以下のいずれかの状態への対応として、バッファに入れられているすべてのテキスト書式が、出力装置に表示されます。
 - プログラムによって **converse** ステートメントが実行されるか、または
 - プログラムは終了します。

多くの場合、**display** ステートメントよりも **converse** ステートメントをコード化して、画面に出力されている最後の書式を表示します。

固定書式はそれぞれ、スクリーン内の位置が決められているため、相互間や浮動書式の繰り返し表示との関連で指定する順序は、重要ではありません。バッファの内容は、画面に出力が送信された時点で、消去されます。

テキスト書式を別のテキスト書式でオーバーレイすると、エラーは発生しませんが、以下の文が表示されます。

- 部分的な書式によって別の固定書式の任意の行がオーバーレイされると、EGL は、バッファからのその他の出力を消去することなく、既存の書式を置換します。新規書式を表示する前に、既存の出力を消去する場合は、新規書式の **display** または **converse** ステートメントを発行する前に、システム関数 `ConverseLib.clearScreen` を呼び出します。
- 浮動域の下に浮動マップを配置するために **display** または **converse** ステートメントを使用すると、その浮動域内のすべての浮動書式が消去され、追加された書式が同じ浮動域の最初の行に配置されます。
- 浮動書式によって固定書式がオーバーレイされると、これらの文が適用されます。
 - 浮動域内の固定書式の行のみが、浮動書式によって上書きされる
 - 固定書式行が、変数フィールドを持つ浮動書式行によって上書きされた場合の結果は予測不能

表示する書式が 1 つでも多数でも、出力先は、ユーザーがその実行単位を開始した方のスクリーン装置です。

関連する概念

164 ページの『書式パーツ』

関連する参照項目

553 ページの『EGL ソース形式の書式パーツ』

550 ページの『EGL ソース形式の FormGroup パーツ』

847 ページの『clearScreen()』

テキスト・アプリケーションのセグメンテーション

セグメンテーションは、プログラムが、**converse** ステートメントを発行する前に、その環境とどのように相互作用するかに関係します。

デフォルトでは、テキスト書式を表示するプログラムはセグメント化されていないため、プログラムは常にメモリー内に存在して、単一のユーザーにのみサービスを提供しているかのように振る舞います。セグメント化されていないプログラムが **converse** ステートメントを発行する前に、以下の規則が有効です。

- データベースおよびその他のリカバリー可能リソースはコミットされない
- ロックは解放されない
- ファイルおよびデータベースの位置は保存される
- 単一ユーザーの EGL テーブルは更新されず、その値は **converse** ステートメントの前後で変わらない
- 同様に、システム変数も更新されない

呼び出し先プログラムは、常にセグメント化されません。

セグメント化されていないプログラムは、容易にコード化することができます。例えば、**converse** 後に、SQL 行のロックを再度獲得する必要はありません。欠点としては、SQL 行がユーザーの考慮時間中も保留されることなどがあり、この振る舞いによって、同じ SQL 行へのアクセスを必要とする他のユーザーに対するパフォーマンス上の問題が発生します。

セグメント化されていないプログラムで **converse** 前にリソースを解放または更新する手法は 2 つあります。

- システム変数 **ConverseVar.commitOnConverse** を 1 に設定すると、**converse** 前は以下ようになります。
 - データベースおよびその他のリカバリー可能リソースはコミットされる
 - ロックは解放される
 - ファイルおよびデータベースの位置は保存されない

ConverseVar.commitOnConverse の設定が、システム変数または EGL テーブルに影響することはありません。

- **converse** を取り扱う 2 番目の手法は、テキスト・プログラムの *segmented* プロパティーを、**yes** に設定することです。これを行うには、開発時にプログラム・プロパティーを変更するか、実行時にシステム変数 **ConverseVar.segmentedMode** を 1 に設定します。 **converse** 前のセグメンテーションの結果は、以下のようになります。
 - データベースおよびその他のリカバリー可能リソースはコミットされる

- ロックは解放される
- ファイルおよびデータベースの位置は保存されない
- 単一ユーザーの EGL テーブルは更新され、その値はプログラムの始動時と同じになる
- システム変数は更新され、その値はプログラムの始動時と同じになる (値が保管された *across segments* の、変数のサブセットを除く)

セグメント化されたプログラムの振る舞いは、システム変数 **ConverseVar.commitOnConverse** の値の影響を受けません。

関連する概念

147 ページの『プログラム・パーツ』

変更データ・タグおよびプロパティ

テキスト書式の各項目には、変更データ・タグがあります。変更データ・タグは、書式が最後に表示されたときに、ユーザーが書式項目を変更したと考えられるかどうかを示す、状況値です。

後で説明するように、項目の変更データ・タグは、項目の **modified** プロパティとははっきりと区別されます。このプロパティは、プログラム内で設定されるもので、変更データ・タグの値を事前設定します。

ユーザーとの対話: 多くの場合、変更データ・タグは、プログラムがユーザーに対して書式を表示する際に、*no* に事前設定されます。その後、ユーザーが書式項目のデータを変更すると、変更データ・タグが *yes* に設定され、プログラム・ロジックによって以下のことが行われます。

- データ・テーブルまたは関数を使用して変更されたデータを検証する (項目の変更データ・タグが *yes* の場合に自動的に行われます)
- ユーザーが項目を変更したことを検出する (例えば、タイプ *if item modified* の条件文を使用して行われます)

ユーザーは、項目内に文字を入力するか、文字を削除することによって、変更データ・タグを設定します。書式を処理依頼する前に、ユーザーがフィールドの内容を表示時の値に戻した場合でも、変更データ・タグは設定されたままとなります。

エラーのために書式が再表示された場合、その書式は依然として同じ **converse** ステートメントを処理しています。この結果、書式が再表示されたとき、**converse** で変更されたすべてのフィールドは変更データ・タグが *yes* に設定されています。例えば、バリデーター機能を持つフィールドにデータを入力した場合、この機能が **ConverseLib.validationFailed** 関数を呼び出し、エラー・メッセージを設定して書式の再表示を引き起こす場合があります。この場合、フィールドの変更データ・タグは *yes* に設定されたままなので、アクション・キーが押されると、バリデーター機能が再度実行されます。

変更プロパティの設定: ユーザーが特定のフィールドを変更したかどうかに関係なく、プログラムにタスクを実行させる場合があります。例えば以下のような場合です。

- ユーザーがフィールドにデータを入力していない場合でも、パスワード・フィールドの検証を強制する場合
- クリティカル・フィールド (保護フィールドを含む) の検証機能を指定し、プログラムが常に特定のフィールド間妥当性検査を行うようにする場合。つまり、プログラム・ロジックは、フィールドのグループを検証し、あるフィールドの値が他のフィールドの検証にどのように影響するかを考えます。

上記のケースを処理するには、プログラム・ロジックまたは書式宣言の中で、特定の項目の**変更**プロパティを設定します。

- 書式の表示以前の論理には、タイプ *set item modified* のステートメントが組み込まれます。その結果、書式が表示されると、その項目の変更データ・タグは *yes* に事前設定されます。
- 書式宣言の中で、項目の**変更** プロパティを *yes* に設定します。この場合、以下の規則が適用されます。
 - 書式の最初の表示時に、その項目の変更データ・タグは *yes* に事前設定されます。
 - 書式が表示される前に以下の状態のいずれかが発生した場合、書式の表示時に、変更データ・タグは *yes* に事前設定されます。
 - コードが、項目の元の内容およびプロパティ値を再割り当てする、タイプ *set item initial* のステートメントを実行した場合
 - コードが、書式の各項目の元のプロパティ値を再割り当てする (内容は再割り当てしない)、タイプ *set item initialAttributes* のステートメントを実行した場合
 - コードが、書式の各項目の元の内容およびプロパティ値を再割り当てする、タイプ *set form initial* のステートメントを実行した場合
 - コードが、書式の各項目の元のプロパティ値を再割り当てする (内容は再割り当てしない)、タイプ *set form initialAttributes* のステートメントを実行した場合

set ステートメントは、変更データ・タグの現在の設定ではなく、**modified** プロパティの値に影響します。タイプ *if item modified* のテストは、書式データが最後にプログラムに戻されたときに有効だった、変更データ・タグの値に基づきます。論理が書式を最初に表示する前に、項目の変更データ・タグをテストしようとする、実行時にエラーが発生します。

プログラムではなくユーザーが項目を変更したかどうかを検出する必要がある場合は、以下のようにして、項目の変更データ・タグの値が確実に *no* に事前設定されるようにしてください。

- 項目の **modified** プロパティが書式宣言で *no* に設定される場合は、タイプ *set item modified* のステートメントは使用しません。この文が使用されない場合、**modified** プロパティは各書式の表示前に、自動的に *no* に設定されます。
- 項目の **modified** プロパティが書式宣言で *yes* に設定される場合は、書式の表示より前の論理で、タイプ *set item normal* のステートメントを使用する。このステートメントは、**modified** プロパティを *no* に設定し、2 次的な結果として、項目を無保護として通常輝度で表示します。

書式が変更済みかどうかのテスト: 変数書式項目のいずれかの変更データ・タグが *yes* に設定されている場合、書式全体が変更済みとして認識されます。ユーザーに対してまだ表示されていない書式の変更状況をテストすると、そのテスト結果は *FALSE* になります。

例: 書式 *form01* で以下のように設定されていることを前提とします。

- フィールド *item01* の **modified** プロパティは *no* に設定されている
- フィールド *item02* の **modified** プロパティは *yes* に設定されている

以下の論理は、さまざまなテストの結果を示しています。

```
// tests false because a converse statement
// was not run for the form
if (form01 is modified)
;
end

// causes a run-time error because a converse
// statement was not run for the form
if (item01 is modified)
;
end

// assume that the user modifies both items
converse form01;

// tests true
if (item01 is modified)
;
end

// tests true
if (item02 is modified)
;
end

// sets the modified property to no
// at the next converse statement for the form
set item01 initialAttributes;

// sets the modified property to yes
// at the next converse statement for the form
set item02 initialAttributes;

// tests true
// (the previous set statement takes effect only
// at the next converse statement for the form
if (item01 is modified)
;
end

// assume that the user does not modify either item
converse form01;

// tests false because the program set the modified
// data tag to no, and the user entered no data
if (item01 is modified)
;
end

// tests true because the program set the modified
// data tag to yes
if (item02 is modified)
;
end
```

```

end

// assume that the user does not modify either item
converse form01;

// tests false
if (item01 is modified)
;
end

// tests false because the presentation was not
// the first, and the program did not reset the
// item properties to their initial values
if (item02 is modified)
;
end

```

EGL 書式エディターの概要

EGL 書式エディターでは、formGroup パーツをグラフィカルに編集できます。書式エディターは、formGroup パーツ、書式パーツ、およびそれらの書式パーツのフィールドを、他のグラフィカル・エディターが Web ページや Web ダイアグラムを処理するのと同様の方法で処理します。

書式エディターには次のパーツがあります。

- エディター自体。書式グループとその書式グループのソース・コードのグラフィカル表現を表示します。エディター下部の「**デザイン**」タブおよび「**ソース**」タブをクリックすることで、グラフィカル表現とソース・コードを切り替えることができます。「ソース」ビューまたは「デザイン」ビューに対して行われた変更は、即時に他のビューに反映されます。
- プロパティ・ビュー。エディターで現在選択されている書式またはフィールドの EGL プロパティを表示します。
- 「パレット」ビュー。エディターで作成できる書式とフィールドのタイプを表示します。
- 「アウトライン」ビュー。エディターで開いている書式グループの階層図を表示します。

書式エディターについては、『*EGL 書式エディターを使用した書式グループの編集*』を参照してください。

関連する概念

- 163 ページの『FormGroup パーツ』
- 164 ページの『書式パーツ』
- 186 ページの『EGL 書式エディターの表示オプション』
- 187 ページの『EGL 書式エディターの書式フィルター』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

- 177 ページの『EGL 書式エディターを使用した書式の作成』
- 186 ページの『EGL 書式エディターの設定』
- 181 ページの『EGL 書式エディターのパレット・エントリートの設定』

EGL 書式エディターを使用した書式グループの編集

EGL 書式エディターでは、formGroup パーツをグラフィカルに編集できます。書式エディターは、formGroup パーツ、書式パーツ、およびそれらの書式パーツのフィールドを、他のグラフィカル・エディターが Web ページや Web ダイアグラムを処理するのと同様の方法で処理します。いつでも、エディター下部の「ソース」タブをクリックして、エディターが生成している EGL ソース・コードを参照できます。書式エディターには次の機能があります。

- 書式エディターは、書式グループのサイズとプロパティを編集することができます。書式グループのプロパティを編集するには、書式エディターの書式グループを開き、プロパティ・ビューでプロパティを変更します。書式グループをサイズ変更するには、書式エディターで書式グループを開き、エディター上部のリストから文字のサイズを選択します。
- 書式エディターは、書式グループの書式を作成、編集、削除できます。書式を作成するには、「パレット」ビューで書式の適切なサイズをクリックし、エディター内での書式のサイズとロケーションを示す長方形を描画します。書式を編集するには、書式をクリックして選択し、その後、プロパティ・ビューを使用してプロパティを編集します。また、書式をドラッグして移動したり、選択した書式のボーダーに表示されるサイズ変更ハンドルを使用してサイズ変更を行ったりすることもできます。書式を右クリックしてポップアップ・メニューを開く際には、同じオプションを複数回使用可能です。『EGL 書式エディターを使用した書式の作成』を参照してください。
- 書式エディターはテンプレートを使用して、ポップアップ・フォームやポップアップ・メニューなどの、一般的に使用する書式のタイプを作成します。これらの書式には事前に作成されたボーダー、セクション、およびフィールドがあります。『EGL 書式エディターの書式テンプレート』を参照してください。
- 書式エディターは、書式のフィールドを作成、編集、削除できます。フィールドを作成するには、「パレット」ビューでフィールドの適切なサイズをクリックし、エディター内でのフィールドのサイズとロケーションを示す長方形を描画します。フィールドは既存の書式内にも追加できます。フィールドを編集するには、フィールドをクリックして選択し、その後、プロパティ・ビューを使用してプロパティを編集します。また、フィールドをドラッグして移動したり、選択した書式のボーダーに表示されるサイズ変更ハンドルを使用してサイズ変更を行ったりすることもできます。フィールドを右クリックしてポップアップ・メニューを開く際には、同じオプションを複数回使用することができます。『定数フィールドの作成』または『変数フィールドの作成』を参照してください。
- フィルターを使用すると、書式が書式エディターに表示されないようにすることができますので、実行時の書式グループの外観をまねることができます。フィルターの切り替え、作成、または編集を行うには、エディター上部の「フィルター」ボタンを使用してください。『EGL 書式エディターの書式フィルター』または『フィルターの作成』を参照してください。
- エディター上部の表示オプションを使用することにより、あるいは「設定」ウィンドウでエディターの設定を行うことにより、書式エディターの外観をカスタマイズすることができます。例えば、これらのオプションでは、書式グループ上にグリッドを表示したり、ズーム・レベルを増減させたり、フィールドにサンプル

値を表示または非表示にしたりすることができます。『EGL 書式エディターの表示オプション』または『EGL 書式エディターの設定』を参照してください。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 163 ページの『FormGroup パーツ』
- 164 ページの『書式パーツ』
- 186 ページの『EGL 書式エディターの表示オプション』
- 187 ページの『EGL 書式エディターの書式フィルター』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 『フィルターの作成』
- 182 ページの『ポップアップ書式の作成』
- 183 ページの『ポップアップ・メニューの作成』
- 184 ページの『テキスト書式または印刷書式によるレコードの表示』
- 186 ページの『EGL 書式エディターの設定』
- 181 ページの『EGL 書式エディターのパレット・エントリの設定』
- 177 ページの『EGL 書式エディターを使用した書式の作成』
- 178 ページの『定数フィールドの作成』
- 179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』

関連する参照項目

- 550 ページの『EGL ソース形式の FormGroup パーツ』
- 553 ページの『EGL ソース形式の書式パーツ』

フィルターの作成

EGL 書式エディターで新規フィルターを作成するには、以下の手順を実行します。

1. 書式エディターで書式グループを開く。
2. 書式エディターで、「**フィルター**」をクリックする。「フィルター」ウィンドウが開きます。
3. 「フィルター」ビューで、「**新規**」ボタンをクリックする。「新規フィルター」ダイアログが開きます。
4. 「新規フィルター」ダイアログで、フィルターの名前を入力し、「**OK**」をクリックする。
5. 以下の手順のうちの 1 つ以上を行って、フィルターをアクティブにすりときに表示する書式を選択する。
 - ・ フィルターで非表示にしたい書式の隣のチェック・ボックスをクリアする。
 - ・ フィルターで表示したい書式の隣のチェック・ボックスを選択する。
 - ・ すべての書式を表示する場合は、「**すべて選択**」ボタンをクリックする。
 - ・ すべての書式を非表示にする場合は「**選択をすべて解除**」ボタンをクリックする。
6. 「**OK**」をクリックする。

これで、新しいフィルターがアクティブになります。「**フィルター**」ボタンの隣のリストを使用するとフィルターを切り替えることができます。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 186 ページの『EGL 書式エディターの表示オプション』
- 187 ページの『EGL 書式エディターの書式フィルター』

関連するタスク

- 『EGL 書式エディターを使用した書式の作成』

EGL 書式エディターを使用した書式の作成

EGL 書式エディターで書式を作成するには、以下の手順を実行します。

1. 書式エディターで書式グループを開く。
2. 「パレット」ビューで、「テキスト書式」または「印刷書式」を開く。
3. エディターの書式グループ上で、書式のサイズと形状を示す長方形をクリックしてドラッグする。「書式パーツの作成」ウィンドウが開きます。
4. 「書式パーツの作成」ウィンドウで、「パーツ名の入力」フィールドに書式の名前を入力する。この名前が、EGL ソース・コードでの書式パーツの名前になります。
5. 「OK」をクリックする。
6. 書式をクリックし、プロパティを「プロパティ」ビューで編集する。
7. 書式に適宜フィールドを追加する。『定数フィールドの作成』および『変数フィールドの作成』を参照してください。

「パレット」ビューで、テンプレートに基づいた書式も作成することができます。これらのテンプレートでは、外観やフィールドが事前定義された書式が作成されます。『ポップアップ書式の作成』または『ポップアップ・メニューの作成』を参照してください。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 163 ページの『FormGroup パーツ』
- 164 ページの『書式パーツ』
- 186 ページの『EGL 書式エディターの表示オプション』
- 187 ページの『EGL 書式エディターの書式フィルター』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 176 ページの『フィルターの作成』
- 182 ページの『ポップアップ書式の作成』
- 183 ページの『ポップアップ・メニューの作成』
- 184 ページの『テキスト書式または印刷書式によるレコードの表示』
- 178 ページの『定数フィールドの作成』
- 179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』

関連する参照項目

- 550 ページの『EGL ソース形式の FormGroup パーツ』
- 553 ページの『EGL ソース形式の書式パーツ』

定数フィールドの作成

定数フィールドは、書式内の変更されないテキストのストリングを表示します。変数フィールドとは異なり、定数フィールドには EGL コードではアクセスできません。書式に定数フィールドを挿入するには、以下の手順で行ないます。

1. EGL 書式エディターで書式グループを開く。
2. 書式グループに書式がない場合は、書式を追加する。『書式の作成』を参照してください。
3. 「パレット」ビューで、追加する定数フィールドの種類をクリックする。デフォルトで使用可能な定数フィールドの種類は以下のとおりです。

表 7. 「パレット」ビューで使用可能な定数フィールド

フィールド名	デフォルトの色	デフォルトの輝度	デフォルトのハイライト	デフォルトの保護
タイトル	青	太字	なし	スキップ
列見出し	青	太字	なし	スキップ
ラベル	シアン	標準	なし	スキップ
手順	シアン	標準	なし	スキップ
ヘルプ	白	標準	なし	スキップ

これらのフィールドは、テキストベース・インターフェースで通常使用される定数テキスト・フィールドのサンプルです。これらのフィールドを書式に配置した後で、個々のフィールドをカスタマイズすることができます。また、フィールドのデフォルトの色、輝度、ハイライトを「パレット」ビューでカスタマイズすることもできます。『EGL 書式エディターのパレット・エントリーの設定』を参照してください。

4. エディターの書式内でマウスをクリックしたまま、フィールドのサイズおよび位置を示す長方形を描画する。マウスのカーソルの隣のプレビュー・ボックスに、フィールドのサイズと書式との相対的な位置が表示されます。

注: フィールドは既存書式の内部にのみ追加できます。

5. フィールドが適切なサイズになった後で、マウスをリリースする。新しいフィールドが作成されます。
6. このフィールドに表示したいテキストを入力する。
7. 「プロパティ」ビューで、新規フィールドのプロパティを設定する。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 164 ページの『書式パーツ』

関連するタスク

- 181 ページの『EGL 書式エディターのパレット・エントリーの設定』
- 179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』

関連する参照項目

- 553 ページの『EGL ソース形式の書式パーツ』

印刷書式またはテキスト書式での変数フィールドの作成

変数フィールドは、書式内で入力テキストまたは出力テキストとして使用できます。各変数フィールドは、EGL プリミティブまたは DataItem パーツに基づいています。定数フィールドとは異なり、変数フィールドは EGL コードでアクセスできます。書式に変数フィールドを挿入するには、以下の手順を実行します。

1. EGL 書式エディターで書式グループを開く。
2. 書式グループに書式がない場合は、書式を追加する。『書式の作成』を参照してください。
3. 「パレット」ビューで、追加する変数フィールドの種類をクリックする。デフォルトでは、以下の変数フィールドの種類が使用可能です。

表 8. 「パレット」ビューで使用可能な変数フィールド

フィールド名	デフォルトの色	デフォルトの輝度	デフォルトのハイライト	デフォルトの保護
入力	緑	標準	下線	なし
出力	緑	標準	なし	スキップ
メッセージ	赤	太字	なし	スキップ
パスワード	緑	不可視	なし	なし

これらのフィールドは、テキストベースのインターフェースで通常使用される変数テキスト・フィールドのサンプルです。これらのフィールドを書式に配置後、個々のフィールドをカスタマイズすることができます。また、フィールドのデフォルトの色、輝度、ハイライトを「パレット」ビューでカスタマイズすることもできます。『EGL 書式エディターのパレット・エントリーの設定』を参照してください。

4. エディターの書式内でマウスをクリック、保持しながらフィールドのサイズおよび位置を示す長方形を描画する。マウスのカーソルの隣のプレビュー・ボックスに、フィールドのサイズと書式との相対的な位置が表示されます。

注: フィールドは既存書式の内部にのみ追加できます。

5. フィールドが適切なサイズになった後で、マウスをリリースする。「新規 EGL フィールド」ウィンドウが開きます。
6. 「新規 EGL フィールド」ウィンドウで、「名前」フィールドに新しいフィールドの名前を入力する。
7. 以下のいずれかを実行して、フィールドの種類を選択します。
 - プリミティブ型を使用するには、「タイプ」リストからプリミティブ型をクリックする。
 - DataItem パーツを使用するには、以下の手順を実行する。
 - a. 「タイプ」リストから「dataItem」をクリックする。「DataItem パーツの選択」ウィンドウが開きます。
 - b. 「DataItem パーツの選択」ウィンドウで、リストから DataItem パーツを選択するか、またはパーツの名前を入力する。
 - c. 「OK」をクリックする。

8. 必要に応じて、1 つまたは複数の「**ディメンション**」フィールドに値を入力して、新規変数フィールドのディメンションを設定する。
9. フィールドで配列を使用したい場合は、「**配列**」チェック・ボックスを選択する。
10. 「**配列**」チェック・ボックスを選択した場合は、「**次へ**」をクリックして以下の手順に進みます。その他の場合は、「**完了**」をクリックして以下の手順を中断します。以下の残りの手順は、配列を作成しない限り適用されないため、配列を作成しない場合は、この時点で新規フィールドが作成され、以下の残りの手順を実行する必要はありません。
11. 「**新規 EGL フィールド**」ウィンドウの「**配列プロパティ**」ページで、「**配列のサイズ**」フィールドに配列のサイズを入力する。
12. 「**インデックスの向き**」ボタンから、「**縦**」または「**横**」を選択する。
13. 「**レイアウト**」で、垂直方向または水平方向のフィールドの数値を「**縦フィールド**」および「**横フィールド**」フィールドに入力する。
14. 「**スペース**」で、配列の行および列のスペースの量を、「**行間の線数**」フィールドおよび「**列間のスペース**」フィールドにそれぞれ入力する。
15. 「**完了**」をクリックする。書式グループに新規フィールドが作成されます。

新規フィールドを作成した後で、クリックしてそのフィールドを選択し、「**プロパティ**」ビューでフィールドのプロパティを設定します。

変数フィールドにはデフォルトの値がないため、ハイライトされていない場合は非表示です。各変数フィールドで適切なサンプル・テキストをマークするには、エディターの上にある「**サンプル値の切り替え**」ボタンをクリックしてください。

変数フィールドを作成した後で、エディター内でこのフィールドをダブルクリックすると「**タイプ・プロパティの編集**」ウィンドウを開くことができます。このウィンドウを使用して、以下の方法でフィールドを編集できます。

- ・「**フィールド名**」フィールドに新しい名前を入力してフィールドの名前を変更する。
- ・「**タイプ**」リストからフィールドの新しいタイプを選択する。
- ・「**精度**」フィールドに新しい数字を入力してフィールドの精度を変更する。

「**タイプ・プロパティの編集**」ウィンドウでフィールドのプロパティを編集し終わった後で、「**OK**」をクリックします。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 164 ページの『書式パーツ』

関連するタスク

- 181 ページの『EGL 書式エディターのパレット・エントリーの設定』
- 178 ページの『定数フィールドの作成』

関連する参照項目

- 553 ページの『EGL ソース形式の書式パーツ』

EGL 書式エディターのパレット・エントリーの設定

EGL 書式エディターのパレットを設定することで、パレット内の定数および変数のフィールドの種類のデフォルトの色、輝度、およびハイライトをコントロールできます。これらの設定を変更するには、以下の手順を実行します。

1. メニュー・バーの「ウィンドウ」 > 「設定」をクリックする。「設定」ウィンドウが開きます。
2. 「設定」ウィンドウの左ペインで、「EGL」 > 「EGL 書式エディター」を展開し、「EGL パレット・エントリー」をクリックする。
3. 「設定」ウィンドウの右ペインで、「パレット・エントリー」リストの定数および変数フィールドの各種類について、次のように選択する。
 - ・ 「色」リストで、当該フィールドの種類のデフォルトの色をクリックする。
 - ・ 「輝度」リストで、当該フィールドの種類のデフォルトの輝度をクリックする。
 - ・ 「ハイライト」ラジオ・ボタンで、当該フィールドのデフォルトのハイライト・スタイルをクリックする。
 - ・ 「保護」ラジオ・ボタンで、デフォルトでユーザー更新からフィールドを保護するかどうかを選択する。フィールド保護の詳細については、『*ConsoleField* プロパティおよびフィールド』を参照してください。

注: 「デフォルトの復元」をクリックすると、すべてのパレット・エントリーをデフォルト設定に復元できます。

4. パレット・エントリーの設定が終了した後で、「OK」をクリックする。

関連する概念

174 ページの『EGL 書式エディターの概要』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

186 ページの『EGL 書式エディターの設定』

178 ページの『定数フィールドの作成』

179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』

関連する参照項目

478 ページの『*ConsoleField* プロパティとフィールド』

EGL 書式エディターの書式テンプレート

EGL 書式エディターはテンプレートを使用して、書式とフィールドの一般的に使用するタイプを作成します。これらのテンプレートは、「パレット」ビューの「テンプレート」ドロワーにリストされています。

書式エディターは、テンプレートから書式を作成できます。これらの書式には事前に作成されたボーダー、セクション、およびフィールドがあります。テンプレートから書式を作成するには、『ポップアップ書式の作成』または『ポップアップ・メニューの作成』を参照してください。

書式エディターでは、EGL レコードをテンプレートとして使用してフィールドのグループを作成できます。EGL レコードからのデータを表示するフィールドを作成するには、『*Displaying a record in a form*』を参照してください。

関連する概念

174 ページの『EGL 書式エディターの概要』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

『ポップアップ書式の作成』

183 ページの『ポップアップ・メニューの作成』

184 ページの『テキスト書式または印刷書式によるレコードの表示』

186 ページの『EGL 書式エディターの設定』

181 ページの『EGL 書式エディターのパレット・エントリーの設定』

177 ページの『EGL 書式エディターを使用した書式の作成』

ポップアップ書式の作成

ポップアップ書式とは、書式グループに追加できる特殊な書式の一種です。基本的に、ポップアップ書式は、通常のテキスト書式と同じですが、境界線やセクションなどの機能をあらかじめ備えた状態で作成されます。EGL 書式エディターでポップアップ書式を作成するには、以下の手順を実行します。

1. 書式エディターで書式グループを開く。
2. 「パレット」ビューで、「**ポップアップ書式**」を開く。
3. エディターの書式グループ上で、ポップアップ書式のサイズと形状を示す長方形をクリックしてドラッグする。「書式パーツの作成」ウィンドウが開きます。
4. 「書式パーツの作成」ウィンドウで、「**パーツ名の入力**」フィールドに書式の名前を入力する。この名前が、EGL ソース・コードでの書式パーツの名前になります。
5. 「**OK**」をクリックする。「新規ポップアップ書式テンプレート」ウィンドウが開きます。
6. 「新規ポップアップ書式テンプレート」ウィンドウで、書式の境界線で使用する文字を「**垂直文字**」フィールドおよび「**水平文字**」フィールドに入力する。
7. 境界線に使用する色を「**色**」リストからクリックする。
8. 境界線に使用する輝度を「**輝度**」リストからクリックする。
9. 「**ハイライト**」ラジオ・ボタンからハイライト値をクリックする。
10. 書式を追加したいセクションで以下の手順を繰り返す。書式には最低 1 つのセクションを追加しなければなりません。
 - a. 「**ポップアップ・セクション**」で、「**追加**」ボタンをクリックする。「ポップアップ書式セクションの作成」ウィンドウが開きます。
 - b. 「ポップアップ書式セクション」ウィンドウで、「**セクションの名前**」フィールドにセクションの名前を入力する。
 - c. 「**行数**」フィールドに、セクションの行数を表す数を入力する。残りの有効行より大きい数字を入力しないでください。この行数は、「新規ポップアップ書式テンプレート」ウィンドウの下部に表示されています。
 - d. 「**OK**」をクリックする。

- e. 「上」ボタンおよび「下」ボタンを使用して、フィールドの表示順序を設定する。

注: ポップアップ・フィールドのセクションの合計行数は、ポップアップ・フィールドの合計行数を超えることはできません。セクションを追加する場合は、「残りの有効行」フィールドに留意し、セクション間の仕切りでは、新規フィールドごとに追加の行が必要になることを念頭に置くようにしてください。

11. ポップアップ・フィールドにセクションを追加し終わってから、「終了」をクリックする。新規のポップアップ書式がエディター内に作成されます。
12. 書式に適宜フィールドを追加する。『定数フィールドの作成』および『変数フィールドの作成』を参照してください。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 186 ページの『EGL 書式エディターの設定』
- 177 ページの『EGL 書式エディターを使用した書式の作成』
- 178 ページの『定数フィールドの作成』
- 179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』
- 『ポップアップ・メニューの作成』

ポップアップ・メニューの作成

ポップアップ・メニューとは、書式グループに追加できる特殊な書式の種類です。基本的に、ポップアップ・メニューは、通常のテキスト書式と同じですが、タイトル、ヘルプ・テキスト、指定の数のメニュー・オプションなどの機能をあらかじめ備えた状態で作成されます。EGL 書式エディターでポップアップ・メニューを作成するには、以下の手順を実行します。

1. 書式エディターで書式グループを開く。
2. 「パレット」ビューで、「ポップアップ・メニュー」を開く。
3. エディターの書式グループ上で、ポップアップ・メニューのサイズと形状を示す長方形をクリックしてドラッグする。「書式パーツの作成」ウィンドウが開きます。
4. 「書式パーツの作成」ウィンドウで、「パーツ名の入力」フィールドに書式の名前を入力する。この名前が、EGL ソース・コードでの書式パーツの名前になります。
5. 「OK」をクリックする。「新規ポップアップ・メニュー・テンプレート」ウィンドウが開きます。
6. 「新規ポップアップ・メニュー・テンプレート」で、ポップアップ・メニューのサイズを「幅」フィールドおよび「高さ」フィールドに入力する。デフォルトでは、これらのフィールドには、エディターで作成済み書式のサイズが入力されます。
7. 「メニュー・タイトル」フィールドに、メニューのタイトルを入力する。

8. 「メニュー・オプションの数」フィールドに、ポップアップ・メニューのメニュー・オプションの数を入力する。
9. 「メニュー・ヘルプ・テキスト」フィールドに、メニューに追加するヘルプ・テキストを入力する。
10. 「完了」をクリックする。新規のポップアップ・メニューがエディター内に作成されます。
11. 新規ポップアップ・メニューへのフィールドの追加および既存フィールドの編集を適宜行う。『定数フィールドの作成』および『変数フィールドの作成』を参照してください。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 186 ページの『EGL 書式エディターの設定』
- 177 ページの『EGL 書式エディターを使用した書式の作成』
- 178 ページの『定数フィールドの作成』
- 179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』
- 182 ページの『ポップアップ書式の作成』

テキスト書式または印刷書式によるレコードの表示

「パレット」ビューの「テンプレート」ドロワーにあるレコード・テンプレートを使用して、EGL レコード・パーツのフィールドに相当する書式フィールドのグループを作成します。書式フィールドを作成する手順は、次のとおりです。

1. EGL 書式エディターで書式グループを開く。
2. 書式を作成する。『EGL 書式エディターを使用した書式の作成』を参照してください。
3. 「パレット」ビューで、「レコード」をクリックする。
4. エディターの書式内でマウスをクリック、保持しながらフィールドのサイズおよび位置を示す長方形を描画する。マウスのカーソルの隣のプレビュー・ボックスに、レコード・フィールドのサイズとフィールドとの相対的な位置が表示されます。

注: レコードは、既存書式の内部にのみ追加できます。

5. レコードが適切なサイズになった後で、マウスをリリースする。「EGL レコードの配置」ウィンドウが開きます。
6. 「EGL レコードの配置」で、「参照」をクリックする。「レコード・パーツの選択」ダイアログが開きます。
7. 「レコード・パーツの選択」ダイアログで、使用したいレコード・パーツの名前をクリックするか、またはレコード・パーツの名前を入力する。
8. 「OK」をクリックする。これにより、「レコードの作成」ウィンドウに、レコード内のフィールドのリストが取り込まれます。
9. 以下の 1 つまたは複数の方法によって、書式にフィールドとして表示したいレコード・パーツ・フィールドを選択し、整理する。

- フィールドを除去するには、その名前をクリックしてから、「**除去**」をクリックする。
- フィールドを追加するには、以下の手順を実行します。
 - a. 「**追加**」ボタンをクリックする。「テーブル・エントリーの編集」ウィンドウが開きます。
 - b. 「テーブル・エントリーの編集」ウィンドウで、「**フィールド名**」ボックスにフィールドの名前を入力する。
 - c. 「**タイプ**」リストから、フィールドのタイプを選択する。
 - d. 選択したタイプで必要な場合は、「**精度**」フィールドにそのフィールドの精度を入力する。
 - e. 「**フィールドの幅**」フィールドにフィールドの幅を入力する。
 - f. フィールドを入力フィールドにしたい場合は、「**このフィールドを入力フィールドにする**」チェック・ボックスを選択する。それ以外の場合は、このチェック・ボックスをクリアしてください。
 - g. 「**OK**」をクリックする。
- フィールドを編集するには、以下の手順を実行します。
 - a. フィールド名をクリックする。
 - b. 「**編集**」ボタンをクリックする。「テーブル・エントリーの編集」ウィンドウが開きます。
 - c. 「テーブル・エントリーの編集」ウィンドウで、「**フィールド名**」ボックスにフィールドの名前を入力する。
 - d. 「**タイプ**」リストから、フィールドのタイプを選択する。
 - e. 選択したタイプで必要な場合は、「**精度**」フィールドにそのフィールドの精度を入力する。
 - f. 「**フィールドの幅**」フィールドにフィールドの幅を入力する。
 - g. フィールドを入力フィールドにしたい場合は、「**このフィールドを入力フィールドにする**」チェック・ボックスを選択する。それ以外の場合は、このチェック・ボックスをクリアしてください。
 - h. 「**OK**」をクリックする。
- リストを上下に移動するには、「**上**」ボタンと「**下**」ボタンを使用する。
- 10. 「**向き**」ラジオ・ボタンを使用して、フィールドを垂直方向と水平方向のどちらで配置するのか選択する。
- 11. 「**行数**」フィールドに、フィールドに含めたいグループの数を入力する。
- 12. フィールドのグループにヘッダー行を付けたい場合は、「**ヘッダー行の作成**」チェック・ボックスを選択する。
- 13. 「**完了**」をクリックする。

関連する概念

- 174 ページの『EGL 書式エディターの概要』
- 175 ページの『EGL 書式エディターを使用した書式グループの編集』
- 181 ページの『EGL 書式エディターの書式テンプレート』

関連するタスク

- 177 ページの『EGL 書式エディターを使用した書式の作成』

178 ページの『定数フィールドの作成』

179 ページの『印刷書式またはテキスト書式での変数フィールドの作成』

EGL 書式エディターの表示オプション

EGL 書式エディターには、設計時に書式グループがエディターでどのように表示されるかをコントロールできるようにする表示オプションがあります。これらのオプションは実行時に書式グループの外観を変更することはありません。以下に、エディター上部の左から右への順で、表示オプションをコントロールするボタンを示します。

Toggle Gridlines

このオプションは、書式グループ上にグリッドを表示し、書式のサイジングと配置を支援します。グリッドの色の変更については、『EGL 書式エディターの設定』を参照してください。

Toggle Sample Values

このオプションは、通常は不可視であるサンプル値を、変数フィールドに挿入します。

Toggle Black and White Mode

このオプションは、エディターの背景を黒から白に切り替えます。

ズーム・レベル

エディターの拡大率レベルを設定します。

エディター上部には、書式グループのサイズとエディターのフィルターをコントロールするその他のボタンがあります。『EGL 書式エディターを使用した書式グループの編集』または『EGL 書式エディターの書式フィルター』を参照してください。

関連する概念

174 ページの『EGL 書式エディターの概要』

187 ページの『EGL 書式エディターの書式フィルター』

関連するタスク

175 ページの『EGL 書式エディターを使用した書式グループの編集』

176 ページの『フィルターの作成』

『EGL 書式エディターの設定』

EGL 書式エディターの設定

EGL 書式エディターの設定によって、背景色やグリッドの色などの書式エディターの外観を変更することができます。書式エディターの設定を変更するには、以下の手順を実行します。

1. メニュー・バーの「ウィンドウ」 > 「設定」をクリックする。「設定」ウィンドウが開きます。
2. 「設定」ウィンドウの左ペインで **EGL** を展開し、「**EGL 書式エディター**」をクリックする。
3. 「設定」ウィンドウの右ペインで、書式エディターの設定を選択する。
 - ・ 「**背景色**」フィールドで、書式エディターの背景色を選択する。
 - ・ 「**グリッドの色**」フィールドで、書式エディターのグリッドの色を選択する。

- ・ フィールドの周囲に境界線を表示したい場合は、「フィールドのハイライト」チェック・ボックスおよび色を選択する。
- ・ 書式エディターの上部と左側にルーラーを表示したい場合は、「ルーラーの表示」チェック・ボックスを選択する。
- ・ 「フォント」リストで、フィールドで使用するフォントをクリックし、隣接するリストからサイズをクリックする。

注: モノスペース・フォントを選択して、フィールドが適切なサイズで書式エディターに表示されることを確認します。モノスペース・フォントとは、Courier New など、すべての文字幅が同じフォントのことです。

- ・ エディターに斜体で点滅フィールドを表示させたい場合は、「点滅フィールドを視覚的に表示する」チェック・ボックスを選択する。このオプションは、ランタイムにはフィールドの外観を変更しません。設計時に外観を変更するのみです。

注: 「デフォルトの復元」をクリックすると、EGL 書式エディター設定ウィンドウをデフォルトに復元できます。

4. EGL 書式エディターの設定が終了した後で、「OK」をクリックする。

関連する概念

174 ページの『EGL 書式エディターの概要』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

『EGL 書式エディターの書式フィルター』

関連するタスク

181 ページの『EGL 書式エディターのパレット・エントリーの設定』

EGL 書式エディターの書式フィルター

フィルターは、EGL 書式エディターに表示する書式を制限します。任意の数のフィルターを定義することができますが、一度にアクティブとなるフィルターは 1 つのみです。フィルターは、設計時の書式グループの表示のみに影響を与えます。EGL コードにはまったく影響を与えません。『フィルターの作成』を参照してください。

エディター上部の「フィルター」ボタンの隣にあるリストを使用して、アクティブなフィルターを切り替えることができます。フィルターを作成、編集、または、削除するには、「フィルター」ボタンをクリックしてください。

「フィルター」ウィンドウから次の関数を使用して、ご使用のフィルターを管理することができます。

- ・ リストからフィルターを選択します。
- ・ 「新規」をクリックして、新規フィルターを追加します。
- ・ リストからフィルターをクリックし「除去」をクリックしてフィルターを削除します。
- ・ フィルターがアクティブな場合に表示する書式を選択します。

関連する概念

174 ページの『EGL 書式エディターの概要』

175 ページの『EGL 書式エディターを使用した書式グループの編集』

関連するタスク

176 ページの『フィルターを作成』

177 ページの『EGL 書式エディターを使用した書式の作成』

コンソール・ユーザー・インターフェースの作成

コンソール・ユーザー・インターフェース

コンソール・ユーザー・インターフェース (ConsoleUI) とは、Windows または UNIX の画面上でデータをテキスト・ベースのフォーマットで表示するためのテクノロジーです。このテクノロジーは、EGL 生成済み Java プログラムでのみ使用でき、ページ・ハンドラーでは使用できません。

ConsoleUI で作成したインターフェースは、ローカルでもリモート端末セッションを経由しても Windows 2000/NT/XP または UNIX X Window システムで表示できます。

ConsoleUI は、テキスト・ユーザー・インターフェース (TextUI) とは性質が異なり、この両者は同一プログラム内で操作できません。

- TextUI が有効である場合、インターフェースのスタイルは、3270 ターミナルと対話しているメインフレーム・プログラムで使用されているインターフェースに似ています。プログラムはテキスト書式を提供していますが、ユーザーがあるフィールドから次のフィールドに移動しても、ユーザー入力は処理されません。ユーザーが (ほとんどの場合 **Enter** キーを押して) フォームを送信すると、フォームにあるすべてのデータがプログラムに戻され、その後のみプログラムがデータを検証します。妥当性検査が成功した場合、プログラムは次のコード化されたステートメントを実行します。
- ConsoleUI が有効である場合、インターフェースのスタイルは、文字ベースのターミナルと対話している UNIX ベースのプログラムで使用されているインターフェースに似ています。プログラムはコンソール・フォームを提供し、ユーザーが **Tab** キーを押してスクリーン内のカーソルを次のフィールドに移動すると、即座にユーザー・イベントに応答できます。妥当性検査はフィールドごとに行われ、ユーザーが現在のフィールドに有効なデータを入力するまで、そのフィールドにカーソルを制限できます。

ConsoleUI を使用するとき、通常は次のようにプログラムをコード化します。

1. ConsoleUI パーツに基づき、常に使用可能である変数セットを宣言します。
ConsoleUI に固有のパーツは定義しません。
2. 適切な EGL 関数を呼び出すときに引数として consoleUI 変数を組み込んで、フォームなどのビジュアル・エンティティを開きます。代わりに、実行時に既知の名前を受け入れる **displayFormByName** などの EGL 関数を呼び出してもビジュアル・エンティティを開けます。
3. EGL **openUI** ステートメントにあるビジュアル・エンティティを参照します。
これにより、特定のイベント (ユーザーによるキー・ストロークなど) を特定の論理に結び付けて、ユーザーの相互作用を行えるようになります。

consoleUI アプリケーションのユーザーは、キーを押すとスクリーン内の表示と対話できますが、マウスをクリックしても効果はありません。

ConsoleUI は、入力フィールドとプリミティブ型変数の間の対応であるバインディングを指定した場合のみ、フィールドへのユーザー入力を受け入れることができます。EGL ランタイムの動作は、以下のとおりです。

- 表示フィールドの初期内容として変数値を使用します。また、
- ユーザーがそのフィールドから移動すると即座に、ユーザーの入力をその変数に移動します。

また、ConsoleUI では、コードが 1 度に 1 行のみを読み取りまたは書き込みする処理モードである、ライン・モードでもユーザーと対話できます。ライン・モードの影響は以下のとおりです。

- Eclipse ワークベンチでは、ユーザーは「コンソール」ビューと対話します。
- コマンド・プロンプトで呼び出されたプログラムでは、ユーザーはコマンド・ウィンドウと対話します。
- UNIX の `Curse` の下で実行されるプログラムでは、ユーザーは UI が表示されているウィンドウと対話し、通常のウィンドウ・ベースの相互作用は中断されません。

ConsoleUI は、Informix 4GL 製品のユーザー・インターフェース・テクノロジーと同等です。

関連するタスク

『consoleUI を使用したインターフェースの作成』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

196 ページの『UNIX 用の ConsoleUI 画面オプション』

813 ページの『EGL ライブラリー ConsoleLib』

669 ページの『openUI』

195 ページの『ConsoleUI での `new` の使用』

consoleUI を使用したインターフェースの作成

コンソール・ユーザー・インターフェース (ConsoleUI) は、Windows または UNIX の画面にテキスト・ベースの書式でデータを表示する技術です。

consoleUI を使用してインターフェースを作成する手順は次のとおりです。

1. EGL ソース・ファイルを作成する
2. 『ConsoleUI パーツおよび関連の変数』で説明されているとおりに、言語エレメントをインクルードしたプログラムを書き込む
3. EGL ソース・ファイルから Java コードを生成する
4. 生成された Java ファイルをアプリケーションとして実行する

これらの各タスクについて、以下に詳しく説明します。

EGL ソース・ファイルの作成

1. ワークベンチで、EGL パースペクティブから「ファイル」>「新規」>「EGL ソース・ファイル」を選択する。または、任意のパーズペクティブから「ファイル」>「新規」>「その他」>「EGL ソース・ファイル」を選択してください。

2. ウィザード画面で、次の情報を入力する。
 - **ソース・フォルダー:** EGL ソース・ファイルが入るディレクトリー・ロケーション。
 - **パッケージ:** EGL ソース・ファイルが入るパッケージ・ロケーション。このフィールドはオプションです。
 - **EGL ソース・ファイル名:** コンソール UI ソース・ファイルのファイル名 (例: **myConsoleUI**)。
3. 「完了」を選択してファイルを作成する。自動的にファイル名の末尾に拡張子 (**.egl**) が付加されます。EGL ファイルが「プロジェクト・エクスプローラー」ビューに表示され、デフォルトの EGL エディターで自動的に開かれます。

ConsoleUI プログラムの作成

ソース・ファイルにデータを取り込み、ConsoleUI を作成するには、ConsoleUI 言語エレメントを使用する必要があります。この言語エレメントは、**egl.ui.console** 概説ヘルプ・トピックで紹介され、個々の ConsoleUI ライブラリー、**OpenUI** ステートメント、**レコード・タイプ**、および**列挙型ヘルプ・トピック**で詳細に定義されています。

ConsoleUI アプリケーションには、少なくとも次のエレメントが組み込まれている必要があります。

1. PROGRAM...END
2. Function main()
3. OpenUI ステートメント

注: **OpenUI** ステートメントは、ConsoleUI の基礎ですが、**OpenUI** ステートメントがなくても正常な ConsoleUI プログラムを作成できます。

EGL ソースからの Java コードの生成

Java ファイルを生成する手順は、次のとおりです。

1. EGL エディターで ConsoleUI ファイルを右クリックする。コンテキスト・メニューが表示されます。
2. 「生成」を選択する。

注: ConsoleUI **.egl** ソース・ファイルは、COBOL に対して生成することはできません。

生成された Java ファイルをアプリケーションとして実行します。

生成された Java ファイルを実行する手順は、次のとおりです。

1. プロジェクト・エクスプローラーで、生成された Java (**.java**) ファイルを右クリックする。コンテキスト・メニューが表示されます。
2. 「実行」>「次を実行」>「**Java アプリケーション**」を選択する。
3. または、エディターで Java ファイルが開いた状態で、メインメニューから「実行」>「次を実行」>「**Java アプリケーション**」を選択する。
4. ConsoleUI がウィンドウに表示されます。

ConsoleUI アプリケーションは、curses ベースの端末セッションか、Swing ベースのグラフィカル・ウィンドウで表示できます。UNIX ユーザーの場合、もっと柔軟に表示を選択できます。これについては、『UNIX 用の ConsoleUI 画面オプション (ConsoleUI screen options for UNIX)』ヘルプ・トピックを参照してください。

注: IBM は、同一プログラムにおける ConsoleUI と TextUI の両方の使用をサポートしません。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

『ConsoleUI パーツおよび関連変数』

196 ページの『UNIX 用の ConsoleUI 画面オプション』

669 ページの『openUI』

ConsoleUI パーツおよび関連変数

consoleUI を処理する際は、関連する consoleUI パーツに基づいて、以下の種類の変数を作成します。

- Window
- Prompt
- ConsoleField
- ConsoleForm
- Menu
- MenuItem

ライブラリー **ConsoleLib** には、PresentationAttributes タイプのシステム変数も含まれます。システム変数は表示される出力の視覚的な特徴を制御します。この特徴を変更するには、PresentationAttributes フィールド **color**、**highlight**、および **intensity** を設定して、システム変数を変更します。これらのフィールドの詳細については、『EGL consoleUI の PresentationAttributes フィールド』を参照してください。

Window

ウィンドウは長方形の領域で、ここに変数として表される他のビジュアル・エンティティーが配置されます。

新たにウィンドウが表示され、他のウィンドウが無効の場合、新規ウィンドウは画面ウィンドウ内に存在します。このウィンドウは、オペレーティング・システムのすべてのウィンドウが持つ基本的な特徴を備える長方形のウィンドウです。Curses ライブラリーが使用される際に UNIX で有効である点が、このウィンドウのみが持つ特徴です。この場合、consoleUI ウィンドウを表示すると、既存のターミナル・ウィンドウがウィンドウ操作モードになります。

追加で表示されるウィンドウはすべて、通常すでに開かれているウィンドウ上部の画面ウィンドウのコンテンツ部分に表示されます。ウィンドウは横並びで表示することもできます。

ウィンドウを宣言すると、さまざまなプロパティを設定できます。例えば、「**position**」はディスプレイの左上隅を基準とした相対位置を表します。「**size**」は、ウィンドウの高さおよび幅を文字数で表します。

ウィンドウ宣言の例を以下に示します。

```
myWindow WINDOW
{name="myWindow", position = [2,2],
 size = [18,75], color = red, hasborder=yes};
```

ConsoleLib.openWindow で始まる名前を持つ EGL 関数を使用して、ウィンドウを表示します。他のデータを表示する際にウィンドウが表示されていない場合は、EGL によってウィンドウが表示されます。

Prompt

プロンプトは、ユーザー入力を導出するオンライン・ステートメントです。プロンプトの宣言は、次のとおりです。

```
myPrompt Prompt { message = "Type your ID: "};
```

入力に対してのみプロンプトと `String` タイプの変数をバインドする **openUI** ステートメントに変数を含めて、プロンプトを表示します。プロンプトを構成して、単一文字またはストリングを使用できるように設定できます。

ConsoleField

`consoleField` はスクリーン内のフィールドで、コンソール・フォームのコンテキストで宣言されます (後述を参照)。次の例では、実行時にコンテンツが変更される可能性がある `consoleField` を宣言しています。

```
myField ConsoleField (
    name="myFieldName",
    position=[1,31],
    fieldLen=20,
    binding = "myVariable" );
```

定数テキストを指定するには、以下の例のように変数名の代わりにアスタリスク (*) を使用します。

```
* ConsoleField
{ position=[2,5], value="Title: "};
```

名前付き `consoleField` を宣言する際は、`consoleField`、および `consoleField` 内の名前属性の値に対して同じ名前を使用することを推奨します。ただし、それぞれに異なる名前を使用することもできます。`consoleField` に対するアクセスが生成時に解決されるときには、`consoleField` 名 (*myField* など) を参照します。`consoleField` が **openUI** ステートメントのイベントを定義するために使用される場合など、アクセスが実行時に解決されるときには、名前属性値 (*myFieldName* など) を参照します。

ConsoleForm

`consoleForm` は主に `consoleFields` のセットです。`consoleForm` をアクティブにするには、システム関数 **ConsoleLib.displayForm** を呼び出します。例えば、読み取り専用 `consoleForm` を表示するには、次の手順を実行します。

1. **ConsoleLib.displayForm** を呼び出します。

2. システム関数 **ConsoleLib.getKey** を呼び出し、キー・ストロークの間待機します。

ユーザーの `consoleField` の書き込みを可能にするには、代わりに次の手順を実行します。

1. **ConsoleLib.displayForm** を呼び出します。
2. 表示された `consoleForm` または `consoleForm` の特定の `consoleField` のいずれかを参照する **openUI** ステートメントを発行します。

`consoleForm` は `ConsoleForm` サブタイプのレコードで、`consoleField` だけでなく、すべての EGL レコードで有効なフィールドすべてが含まれます。

ユーザーと `consoleField` の表示中のテーブルとの相互作用を可能にするには、次の手順を実行します。

1. `consoleForm` で、`consoleForm` でも宣言されている `consoleField` 配列を順番に参照する `arrayDictionary` を宣言します。
2. **openUI** ステートメントでその `arrayDictionary` を使用します。

ユーザーと `consoleForm` の `consoleField` のサブセットのみの相互作用を可能にするには、明示的に、または辞書を参照して、**openUI** ステートメント に `consoleField` をリストします。`arrayDictionary` と同様に、辞書は `consoleForm` で宣言され、`consoleForm` でも宣言されている `consoleField` を参照します。

EGL では、`consoleForm` で宣言されたプリミティブ変数は表示されません。このような変数を使用すると、`consoleForm` 外で宣言された変数を使用できるため、`consoleField` をバインドできます。

一般に、次の 2 つの方法で `consoleForm` バインディングを作成します。

- `consoleForm` 宣言時にデフォルトのバインディングを設定する。
- **openUI** ステートメントのコード化時にバインディングを設定する。

openUI ステートメントで指定されたバインディングは、デフォルトのバインディング全体をオーバーライドします。`consoleForm` 宣言バインディングはすべて削除されます。

openUI ステートメントを使用して変数をバインドする場合は、ステートメント・プロパティ **isConstruct** を使用することができます。この場合、次の手順を実行します。

- ユーザー入力を SQL WHERE 文節に適切なストリングにフォーマットします。
- そのストリングを単一変数に配置します。EGL **prepare** ステートメントをコード化する際に関連データベースからユーザーが要求するデータを取得する SQL **SELECT** を簡単にコード化できます。

プロパティ **isConstruct** の詳細については、『*OpenUI ステートメント*』を参照します。

タブ順序 とは、ユーザーがある `consoleField` から別の `consoleField` に指定する順序のことです。デフォルトでは、タブ順序は `consoleForm` 宣言における `consoleField` の順序になっています。**openUI** ステートメントで `consoleField` のリストを使用する

際は、タブ順序はそのステートメントの `consoleField` の順序となります。同様に **openUI** ステートメントで辞書または `arrayDictionary` を使用する際は、タブ順序はその辞書または `arrayDictionary` の宣言における `consoleField` の順序となります。

デフォルトでは、**Esc** キーを押すと `consoleForm`-related **openUI** ステートメントが終了します。

Menu

メニューとは、水平に表示されるラベルのセットです。1 つ 1 つのラベルが統合されてメニューを表し、メニューの `menuItem` ごとにラベルが存在します。ユーザーが特定の `menuItem` を選択した際に応答が行われるようにするには、**openUI** ステートメントでメニュー全体を参照し、また、このステートメントの `OnEvent` 文節で `menuItem` を参照します。

MenuItem

`menuItem` はラベルを表示し、直前のセクションの説明に従って使用されます。

関連する概念

- 91 ページの『`ArrayDictionary`』
- 189 ページの『コンソール・ユーザー・インターフェース』
- 87 ページの『辞書』

関連する参照項目

- 478 ページの『`ConsoleField` プロパティとフィールド』
- 492 ページの『EGL `consoleUI` の `ConsoleForm` プロパティ』
- 813 ページの『EGL ライブラリー `ConsoleLib`』
- 196 ページの『UNIX 用の `ConsoleUI` 画面オプション』
- 494 ページの『EGL `consoleUI` の `Menu` フィールド』
- 495 ページの『EGL `consoleUI` の `MenuItem` フィールド』
- 669 ページの『`openUI`』
- 497 ページの『EGL `consoleUI` の `PresentationAttributes` フィールド』
- 499 ページの『EGL `consoleUI` の `Prompt` フィールド』
- 500 ページの『EGL `consoleUI` の `Window` フィールド』

関連するタスク

- 190 ページの『`consoleUI` を使用したインターフェースの作成』

ConsoleUI での new の使用

`consoleUI` を使用する EGL プログラムを作成するときには、`Menu`、`MenuItem`、`Prompt`、および `Window` タイプの変数はすべて参照変数 となります。この参照変数には、変数の外部に保管された値を参照するメモリー・アドレスが含まれます。

次の例では、他の変数を宣言する際に参照変数を宣言することができます。

```
myPrompt Prompt { message = "Type your ID: "};
```

また、次の例で示すように、参照変数を宣言し、この参照変数を予約語 **new** で初期化することができます。

```
myPrompt Prompt = new Prompt { message = "Type your ID: "};
```

変数の宣言時には、2 つのフォーマット間の差異による実際の影響はほとんどありません。ただし、`openUI` ステートメントのコーディング時、予約語 **new** によって `openUI` に示されるようなコーディングの便宜性が得られます。

new の一般構文は、以下のとおりです。

```
new partName
```

partName

以下のワードのいずれかによって特定の種類のパーツが参照されます。

- Menu
- MenuItem
- Prompt
- Window

参照変数のその他の作用については、『*EGL* における参照の互換性』を参照してください。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

669 ページの『openUI』

795 ページの『EGL での参照の互換性』

関連するタスク

190 ページの『consoleUI を使用したインターフェースの作成』

UNIX 用の ConsoleUI 画面オプション

サポートされている UNIX プラットフォームの EGL ユーザーは、グラフィカル表示モードまたは UNIX `curses` モードのいずれかを使用して、ConsoleUI アプリケーションを実行できます。

グラフィカル表示モード (Graphical display mode)

ConsoleUI アプリケーションをグラフィカル表示モードで実行するには、EGL `curses` ライブラリーが、実行中のシェルのライブラリー `PATH` 環境変数に置かれていないことを確認する必要があります。これがデフォルト・モードです。

UNIX `curses` モード

ConsoleUI アプリケーションを UNIX `curses` モードで実行するには、実行中のシェルのライブラリー `PATH` 環境変数内に、該当するプラットフォーム固有の EGL `curses` ライブラリーが必要です。EGL `curses` ライブラリーは、EGL Support Web サイトからダウンロードする必要があります。

EGL `curses` ライブラリーをダウンロードする手順

1. 該当する EGL Support Web サイトにアクセスする。
 - Rational Application Developer の URL は、次のとおりです。
`http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist`
 - Rational Web Developer の URL は、次のとおりです。
`http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rwd/60/redist`
2. **EGLRuntimesV60IFix001.zip** ファイルを適切なディレクトリーにダウンロードする。
3. **EGLRuntimesV60IFix001.zip** を unzip して、以下のファイルがあることを確認する。
 - AIX® : **EGLRuntimes/Aix/bin/libCursesCanvas6.so**
 - Linux : **EGLRuntimes/Linux/bin/libCursesCanvas6.so**
4. 該当する EGL curses ライブラリーをライブラリー PATH 環境変数に挿入する。

AIX: 次の Bourne シェルを使用して、**LIBPATH** ライブラリー PATH 環境変数を設定します。

```
"LIBPATH=$INSTDIR/aix; export LIBPATH"
```

Linux: 次の Bourne シェルを使用して、**LD_LIBRARY_PATH** ライブラリー PATH 環境変数を設定します。

```
"LD_LIBRARY_PATH=$INSTDIR/aix; export LD_LIBRARY_PATH"
```

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

192 ページの『ConsoleUI パーツおよび関連変数』

669 ページの『openUI』

関連するタスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL Web アプリケーションの作成

Web サポート

EGL は、以下のようにして Web ベースのアプリケーションをサポートします。

- ページ・ハンドラー という論理パーツを開発できます。この論理パーツの各関数は、Web ページでの特定のユーザー・アクションによって呼び出されます。この論理パーツを生成することによって、カスタマイズ可能な JavaServer Faces JSP を作成することもできます。
- アプリケーション内の各ページに 1 は、ユーザーがログアウトするためのボタンを表示する場合のように、いくつかの Web ページに共通する機能性を持たせることができます。この場合、ユーザーがボタンをクリックすることにより、共通のサブルーチンとして働く EGL プログラムも呼び出すことができます。
- 最後に、WebSphere Page Designer を使用すると、JavaServer Faces JSP をカスタマイズし、ページ・ハンドラーに影響を与えることができます (『EGL の Page Designer サポート』を参照)。

関連する概念

207 ページの『ページ・ハンドラー』

371 ページの『WebSphere Application Server と EGL』

関連するタスク

370 ページの『ローカル・マシンでの Web アプリケーションの開始』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

単一テーブルの EGL Web アプリケーションの作成

「EGL データ・パーツおよびページ」ウィザード

「EGL データ・パーツおよびページ」ウィザードを使用すると、リレーショナル・データベース内の特定のテーブルを保守するための Web ベースのユーティリティを簡単に作成できます。

このウィザードは、以下のエンティティを作成します。

- ページ・ハンドラーのセット。ユーザーは後で、これを Java Server Faces の下で稼働するパーツのセット内に生成します。
- SQL レコード・パーツ、および関連するデータ項目パーツとライブラリー・ベースの関数パーツ。
- 以下の Web ページを提供する一連の JSP ファイル。
 - 選択条件ページ。これは、ユーザーから選択基準を受け入れます。

- リスト・ページ。これは、ユーザーの基準に基づいて複数の行を表示します。
- 作成詳細ページ。これにより、ユーザーは 1 つの行を表示するか挿入することができます。
- 詳細ページ。これにより、ユーザーは 1 つの行を表示、更新、または削除できます。

ユーザーは最初に選択基準ページを検出しますが、そのページに必要な情報を指定しなかった場合、ユーザーは最初にリスト・ページを検出し、(この状態では) このページでテーブル内のすべての行にアクセスできます。

ウィザードを使用して作業している場合は、以下のことができます。

- 表示されるフィールドを変更したり、ページ間のリンクを組み込んだりすることにより、上記の各 Web ページをカスタマイズする。
- 所定のデータベース表またはビューから行の作成、読み取り、更新、削除を行うために使用する SQL レコードのキー・フィールドを指定する。
- 行の作成、読み取り、または更新を行うための明示的な SQL ステートメントをカスタマイズする。(行を削除する SQL ステートメントをカスタマイズすることはできません。)
- 所定のデータベース表またはビューから一連の行を選択するために使用する SQL レコードのキー・フィールドを指定する。
- 一連の行を選択する明示的な SQL ステートメントをカスタマイズする。
- 各 SQL ステートメントを検証し、実行する。

出力には、以下のファイルが含まれます。

- Web アプリケーションを呼び出す HTML ファイル (index.html)。
- 上記の各 Web ページを提供する一連の JSP ファイル。
- SQL レコード・パーツ内の構造体項目によって参照されるすべてのデータ項目が入っている EGL ソース・ファイル。
- また、このウィザードは、それぞれの SQL レコード・パーツごとに 2 つのファイルを生成します。1 つはレコード・パーツ自体用のファイルで、1 つは関連するライブラリー・ベースの関数用のファイルです。「**同じファイル内のレコードとライブラリー (Record and library in the same file)**」チェック・ボックスを選択した場合は、ファイルの数を減らすことができます。

ウィザードが Web ベースのユーティリティを作成した後、そのユーティリティをカスタマイズできます。

関連する概念

357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』

247 ページの『SQL サポート』

関連するタスク

201 ページの『単一テーブルの EGL Web アプリケーションの作成』

276 ページの『EGL ウィザード用のデータベース接続の作成、編集、または削除』

277 ページの『EGL ウィザードでの SQL ステートメントのカスタマイズ』

203 ページの『「EGL データ・パーツおよびページ」ウィザードでの Web ページの定義』

単一テーブルの EGL Web アプリケーションの作成

単一のリレーショナル・データベース・テーブルから EGL Web アプリケーションを作成するには、次のようにします。

1. 「ファイル」>「新規」>「その他...」を選択する。ウィザードを選択するためのダイアログが表示されます。
2. 「EGL」を展開し、「EGL データ・パーツおよびページ」をダブルクリックする。「EGL データ・パーツおよびページ」ダイアログが表示されます。
3. EGL Web プロジェクト名を入力するか、ドロップダウン・リストから既存プロジェクトを選択する。EGL パーツがそのプロジェクト内に生成されます。
4. 既存のデータベース接続をドロップダウン・リストから選択するか、次のようにして新規データベース接続を確立する。
 - 新規データベース接続を確立するには、「追加」をクリックし、ヘルプ・トピック『データベース接続ページ』の指示に従います。ヘルプ・トピックにアクセスするには、F1 を押します。
 - データベース接続の編集または削除の詳細については、『EGL ウィザード用のデータベース接続の作成、編集、または削除』を参照してください。

データベースへの接続が作成されると、データベース・テーブルのリストが表示されます。

5. デフォルトのデータ項目の EGL ファイル名を受け入れたくない場合は、新しいファイル名を入力する。
6. 「データの選択」フィールドで、必要なデータベース・テーブルの名前をクリックする。
7. 「レコード名」フィールドで、作成する EGL レコードの名前を指定するか、デフォルト名を受け入れる。
8. ライブラリー・パーツと SQL レコード・パーツを同じファイルに組み込みたい場合は、チェック・ボックスを選択する。
9. 追加フィールドをデフォルト以外の値に設定するには、「次へ (Next)」を選択し、そうでない場合は「完了」をクリックする。残りのステップは、「次へ」をクリックしたことを想定しています。
10. 個々の行の読み取り中、更新中、および削除中に使用するキー・フィールドを選択し、右矢印をクリックする。複数のキー・フィールドを選択するには、**Ctrl** キーを押したまま、さまざまなフィールド名をクリックします。右側のリストからキー・フィールドを除去するには、フィールド名を強調表示にして、左矢印をクリックします。
11. 行セットを選択するときに使用する選択条件フィールドを選択し、右矢印をクリックする。複数のフィールドを選択するには、**Ctrl** キーを押したまま、さまざまなフィールド名をクリックします。右側のリストからフィールドを除去するには、フィールド名を強調表示にして、左矢印をクリックします。
12. 暗黙の SQL ステートメントをカスタマイズするには、『EGL ウィザードでの SQL ステートメントのカスタマイズ』を参照する。このオプションは、EGL の **delete** ステートメントには使用できません。
13. 「次へ」をクリックする。

14. 新しい Web ページにテンプレートを適用したい場合は、次の手順を実行する。
 - a. 「ページ・テンプレートを選択」チェック・ボックスを選択する。
 - b. 「サンプル・ページ・テンプレート」または「ユーザー定義ページ・テンプレート」のどちらかをクリックして、ページ・テンプレートのタイプを選択する。
 - c. 使用したいページ・テンプレートをクリックする。サムネールを選択するか、「参照」ボタンをクリックしてテンプレートのロケーションまでブラウズすることができます。
15. 「次へ」をクリックする。
16. Web ページをカスタマイズするには、『「EGL データ・パーツおよびページ」ウィザードでの Web ページの定義』を参照する。
17. 「次へ」をクリックする。
18. 「Web アプリケーションの生成」画面が表示される。これには、生成されるファイルおよび Web ページのリスト (画面下部) も含まれます。
 - a. EGL パーツを受け取る EGL Web プロジェクトの名前を変更するには、「EGL Web プロジェクト名」フィールドにプロジェクト名を入力するか、関連するドロップダウン・リストからプロジェクトの選択を行います。
 - b. 特定のパーツ型 (ページ・ハンドラー、データ、またはライブラリー) 用の EGL および Java パッケージを指定するには、関連するフィールドにパッケージ名を入力するか、または関連するドロップダウン・リストから名前を選択します。
 - c. 特定の Web ページ用に生成された JSP ファイルおよび EGL ファイルの名前を変更するには、「Web ページ」から該当するエントリーをクリックし、新しい名前を入力します。それぞれのファイル名には、入力した文字や数字が含まれていますが、スペースおよびその他の文字は除外されます。

ページ・ハンドラーパーツ、データ・パーツ、およびライブラリー・パーツのパッケージを入力するか、または選択してください。
19. 「完了」をクリックする。

関連する概念

247 ページの『SQL サポート』

関連するタスク

276 ページの『EGL ウィザード用のデータベース接続の作成、編集、または削除』

274 ページの『リレーショナル・データベース・テーブルからの EGL データ・パーツの作成』

277 ページの『EGL ウィザードでの SQL ステートメントのカスタマイズ』

203 ページの『「EGL データ・パーツおよびページ」ウィザードでの Web ページの定義』

「EGL データ・パーツおよびページ」ウィザードでの Web ページの定義

「EGL データ・パーツおよびページ」ウィザードは、リレーショナル・データベース・テーブルから Web アプリケーションを作成します。このウィザードで作業する場合は、作成される各 Web ページ・タイプの以下のような外観を指定できます。

- ページ・タイトル
- スタイル・シート
- 表示するフィールド (それらの順序とプロパティーも含む)
- 他のページへのリンク

「アプリケーションの Web ページの定義」と呼ばれるウィザードのダイアログで作業している場合は、タブをクリックしてページ間をナビゲートすることができます。ページごとに、次の手順を実行します (可能な場合)。

1. 「ページ・タイトル」フィールドでページ・タイトルを設定する。
2. 「スタイル・シート」フィールドでドロップダウン・リストからスタイル・シートを選択する。
3. 現行ページ定義を受け入れた場合の効果を表示するには、「**プレビュー**」をクリックする。
4. ページに表示する行数を指定したい場合は、「**ページ編集**」を選択し、「**ページ・サイズ**」に行数 (正整数) を割り当てる。
5. ページに組み込むフィールドを次のようにして選択する。
 - a. フィールドを組み込むには、関連するチェック・ボックスを選択する。すべてのフィールドを選択するには、「**すべて (All)**」をクリックします。
 - b. フィールドを除外するには、関連するチェック・ボックスをクリアする。すべてのフィールドを除外するには、「**なし (None)**」をクリックします。
 - c. フィールドの表示ロケーションを変更するには、フィールドをクリックしてから、上下の矢印を使用してフィールドを別のロケーションへ移動します。
 - d. フィールドのプロパティーを設定するには、フィールドをクリックしてから、「**プロパティー**」ペインで「**値**」フィールドをダブルクリックします。値を入力するか、場合によっては、ドロップダウン・リストから選択します。
6. ユーザーがページ上で実行できるアクションを次のようにして選択する。
 - a. アクションを組み込むには、関連するチェック・ボックスを選択します。すべてのアクションを選択するには、「**すべて選択 (Select All)**」をクリックします。

個々のアクションには、次のように、「**作成**」、「**削除**」、「**抽出**」、「**リスト**」、および「**読み取り**」があります。

- 「**作成**」は、ユーザーが 1 行を表示または挿入できる作成詳細ページへリンクします。このオプションは作成詳細ページだけにあり、ユーザーがそのページから行を作成できることを示します。
- 「**削除**」は、詳細ページでのみ使用できます。このオプションは、ユーザーの指定したキーを持つレコードを削除します。

- 「抽出」は、ユーザーからの選択基準を受け入れる選択条件ページへリンクします。このオプションは選択条件ページだけにあり、ユーザーがそのページから結果セットを戻させることができることを示します。
 - 「リスト」は、ユーザーの基準に従って複数の行を表示するリスト・ページへリンクします。
 - 「読み取り」は、作成詳細ページでのみ使用できます。このオプションは、ユーザーの指定したキーを持つレコードを表示します。
 - 「更新」は、詳細ページでのみ使用できます。このオプションは、ユーザーによって変更されたレコードを更新します。
- b. アクションを除外するには、関連するチェック・ボックスをクリアします。すべてのアクションを除外するには、「なし」をクリックします。

常に使用可能なアクションの場合は、そのチェック・ボックスをクリアすることはできません。

- c. アクションの Web ページ・ラベルを設定するには、アクション名をクリックした後、「プロパティ」ペインの「値」フィールドをダブルクリックし、値を入力します。

関連する概念

247 ページの『SQL サポート』

199 ページの『「EGL データ・パーツおよびページ」ウィザード』

関連するタスク

201 ページの『単一テーブルの EGL Web アプリケーションの作成』

274 ページの『リレーショナル・データベース・テーブルからの EGL データ・パーツの作成』

276 ページの『EGL ウィザード用のデータベース接続の作成、編集、または削除』

119 ページの『EGL 設定の変更』

370 ページの『ローカル・マシンでの Web アプリケーションの開始』

EGL ページ・ハンドラーパーツの作成

ページ・ハンドラーパーツは、Java Server Faces JSP にデータとサービスを提供することにより、ユーザーと Web ページとのランタイム相互作用を制御します。JSP を作成すると、ページ・ハンドラーパーツが *EGLSource* フォルダー内のページ・ハンドラー と呼ばれるパッケージの中に自動的に作成されます。ページ・ハンドラーパーツ名は、対応する JSP と同じですが、.egl というファイル拡張子が付きます。

オプションとして、ページ・ハンドラーパーツを作成し、システムに JSP をプロジェクトに自動的に追加させることができます。ただし、EGL Web プロジェクト内に同じ名前の JSP ファイルがまだ存在しない場合に限りです。EGL ページ・ハンドラーパーツを作成するには、次のようにします。

1. EGL Web プロジェクトに ページ・ハンドラー という名前のパッケージが含まれていない場合は、それを作成する必要がある。Page Designer は、すべてのページ・ハンドラーパーツがページ・ハンドラー という名前のパッケージ内に存在することを必要とします。パッケージの作成方法の詳細については、『EGL パッケージの作成』を参照してください。

2. ページ・ハンドラーパーツを入れるページ・ハンドラー パッケージ内の EGL ファイルを識別する。そのファイルを EGL エディターで開いてください。まだ EGL ファイルがない場合は、EGL ファイルを作成する必要があります。
3. EGL 構文に従って、ページ・ハンドラーパーツの特性を入力する（詳細については、『EGL ソース形式のページ・ハンドラー・パーツ』を参照してください）。コンテンツ・アシストを使用して、ページ・ハンドラーパーツの構文のアウトラインをファイルに入れることができます。
4. EGL ファイルを保管する。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

207 ページの『ページ・ハンドラー』

関連するタスク

134 ページの『EGL パッケージの作成』

135 ページの『EGL ソース・ファイルの作成』

135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

524 ページの『EGL でのコンテンツ・アシスト』

725 ページの『命名規則』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL の Page Designer サポート

EGL Web プロジェクトで JSP ファイルを作成する場合、EGL が自動的にページ・ハンドラーを作成し、そのページ・ハンドラーにはユーザーがカスタマイズできる EGL 骨組みコードが含まれます。その後、Page Designer で、次のようにします。

1. コンポーネントをパレットから JSP ヘドドラッグします。
2. 「Attributes (属性)」ビューを使用して色などのコンポーネント固有の特性を設定し、バインディング (コンポーネントとデータまたはロジックとの間の関係) をセットアップします。

次のような EGL 固有の作業を行うことができます。

- EGL 変数を作成し、既存のページ・ハンドラー内に配置する。
- ページ・ハンドラー項目を JSP ユーザー・インターフェース・コンポーネントにバインドする。
- ページ・ハンドラー関数をボタンとハイパーリンク・コントロールにバインドする。これらの関数は、イベント・ハンドラーとして機能します。

Page Designer でソース・タブを使用した場合は、JSP ファイル (特に、JavaServer Faces ファイル) 内のコンポーネントをページ・ハンドラー内のデータ域および関数に手動でバインドできます。EGL に大/小文字の区別はありませんが、JSP ファイル内で参照される EGL 名は EGL 変数または関数の宣言と大/小文字が同じである必要があります、完全に一致しなかった場合は JavaServer Faces エラーが発生します。EGL 変数または関数を JSP フィールドにバインドした後は、EGL 変数または関数の大/小文字を変更しないようにすることをお勧めします。

命名問題の詳細については、『JSP ファイルおよび生成された Java Bean 内の EGL ID の変更』を参照してください。

ページ・ハンドラーのデータ域へのコンポーネントのバインド

JSP 上のコンポーネントは、ほとんどの場合、データと 1 対 1 で対応しています。例えば、テキスト・ボックスには、テキスト・ボックスのバインド先の EGL 項目の内容が表示されています。入力テキスト・ボックスでは、ユーザーがデータを変更すると、EGL 項目も更新されます。

チェック・ボックス・グループ、リスト・ボックス、ラジオ・ボタン・グループ、またはコンボ・ボックスを指定する場合、状態は複雑になります。この場合、以下の 2 種類のバインディングが必要になります。

- ユーザーに対して表示するテキストにコンポーネントをバインドするバインディング。例えば、リスト・ボックス内の項目のテキストです。
- ユーザーの選択を表す値を受け取るページ・ハンドラー・データ域にコンポーネントをバインドするバインディング。例えば、ユーザー選択リスト・ボックス項目の数値索引を受け取るデータ項目を作成できます。

プロパティ・ビューでは、次のいずれかの手順を実行して、ユーザーに対して表示するテキストにコンポーネントをバインドできます。

- 「**選択項目の追加**」を使用して、コンポーネントが単一文字ストリングに関連付けられていることを示す。これは、明示的に指定したり、またはページ・ハンドラー項目を識別することによって指定できます。
- 「**選択項目のセットを追加**」を使用して、コンポーネントが文字ストリングのリストに関連付けられていることを示す。これは、明示的に指定することも、ページ・ハンドラー域 (データ・テーブルまたは文字項目の配列など) を識別することによって指定することもできます。

または、単一選択コンポーネント (コンボ・ボックス、単一選択リスト・ボックス、またはラジオ・ボタン・グループ) を文字項目の配列にバインドできます。これを行うには、ページ・データ・ビューから配列をコンポーネントにドラッグします。

ユーザーの選択を表す値を受け取るデータ域にコンポーネントをバインドするには、ページ・データ・ビューまたはプロパティ・ビューで作業します。どのコンポーネントをバインドする場合でも (単純なテキスト・ボックスの場合でも) 手順は同じです。

値が 2 つの選択肢の 1 つにしかない場合は、項目プロパティ **boolean** が **yes** になっている EGL 項目にコンポーネントをバインドできます。コンポーネントは、次の 2 つの値のいずれかで項目を取り込みます。

- 文字項目の場合、値は **Y** (はい) または **N** (いいえ)
- 数値項目の場合、値は **1** (はい) または **0** (いいえ)

チェック・ボックスが表示される場合は、状況 (チェックマークが付いているかどうか) は、バインドされた項目の値によって異なります。

ページ・ハンドラー内のデータ項目に適用できるプロパティの詳細については、『**ページ項目のプロパティ**』を参照してください。

関数へのコンポーネントのバインド

コマンド・ボタンまたはコマンド・ハイパーリンクをページ面にドラッグした後は、次のようにして、既存の EGL 関数または Page Designer が作成するイベント・ハンドラーにコンポーネントをバインドできます。

- 次のいずれかの方法で、既存のイベント・ハンドラーにコンポーネントをバインドする
 - ページ・データ・ビュー内のアクション・ノードからイベント・ハンドラーをコンポーネントにドラッグする
 - クイック編集ビューでコンポーネントをオープンする
 - コンポーネントを右マウス・ボタンでクリックして、「**Edit Faces Command Event (Faces コマンド・イベントの編集)**」を選択する
- クイック編集ビューでコンポーネントをオープンしたとき、またはコンポーネントを右マウス・ボタンでクリックして「**Edit Faces Command Event (Faces コマンド・イベントの編集)**」を選択したときに、新規イベント・ハンドラーが Page Designer によって作成されるようにする

Page Designer がページ・ハンドラー内にイベント・ハンドラーを作成し、そのページ・ハンドラー関数にアクセスできるようになる場合は、関数の名前はツールに割り当てられたボタン ID にストリング「Action」を付け加えたものとなります。名前がページ・ハンドラー固有ではない場合は、Page Designer が関数名に番号を付加します。

関連する概念

『ページ・ハンドラー』

関連するタスク

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

214 ページの『EGL レコードと Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

738 ページの『ページ・ハンドラー・フィールドのプロパティ』

ページ・ハンドラー

EGL ページ・ハンドラー は、ページ・コード の 1 つの例です。これはユーザーの Web ページとのランタイム相互作用を制御するもので、以下のいずれのタスクを実行することもできます。

- JSP ファイルへ実行依頼するために、データ値を割り当てる。これらの値は、最終的に Web ページに表示されます。
- ユーザーまたは呼び出し先プログラムから戻されたデータを変更する。
- 制御を別の JSP ファイルへ転送する。

最も簡単な方法としては、Page Designer で JSP ファイルをカスタマイズしてページ・ハンドラーを作成する方法です。詳しくは、『EGL の Page Designer サポート』を参照してください。

ページ・ハンドラーには、変数と以下のようなロジックが組み込まれます。

- JSP が Web ページを初めてレンダリングしたときに呼び出される `OnPageLoad` 関数
- イベント・ハンドラーのセット (それぞれ、特にユーザーによるボタンやハイパーテキスト・リンクのクリックなどのユーザー・アクションに応答して呼び出される)
- オプションとして、Web ページ・入力フィールドの検証に使用される検証機能
- 他のページ・ハンドラー関数によってのみ呼び出すことのできる専用の関数

ページ・ハンドラーの変数は、2 とおりの方法でアクセスされます。

- ランタイム環境がデータに自動的にアクセスします。JSP のフィールドが、ページ・ハンドラーの項目にバインド されている場合、結果は次のようになります。
 - `OnPageLoad` 関数が実行された後で、Web ページが表示される前に、データがバインドされる JSP フィールドに各ページ・ハンドラー項目の値が書き込まれます。
 - バインドされた JSP フィールドのある書式をユーザーが処理依頼すると、処理依頼された書式の各フィールドの値が、関連するページ・ハンドラー項目にコピーされます。この場合のみ、コントロールがイベント・ハンドラーに渡されます。(ただし、この説明には検証ステップは含まれません。これについては、このトピックで後述します。)
- イベント・ハンドラーおよび `OnPageLoad` 関数は、データ・ストア (SQL データベースなど) や呼び出し先プログラムとの対話の他、データとの対話も行えます。

ページ・ハンドラーパーツはシンプルにする必要があります。パーツには、範囲検査のような単純なデータ妥当性検査が組み込まれる場合がありますが、他のプログラムを呼び出して、複合ビジネス・ロジックを実行することをお勧めします。例えば、データベース・アクセスは、呼び出し先プログラムに対して予約される必要があります。

ページ・ハンドラーに関連した出力

ページ・ハンドラーを保管すると、EGL は JSP ファイルをプロジェクト・フォルダー `WebContent\WEB-INF` に配置します。ただし、これは次の場合に限られます。

- JSP ファイル名を指定するページ・ハンドラーの **view** プロパティに値を割り当ててある
- フォルダー `WebContent\WEB-INF` に、指定した名前の JSP ファイルが入っていない

EGL は JSP ファイルを上書きしません。

ワークベンチ設定で、保管時の自動ビルドを設定している場合、ページ・ハンドラーを保管するたびにページ・ハンドラーの生成が行われます。いずれの場合も、ページ・ハンドラーを生成すると、以下のオブジェクトによって出力が構成されます。

- ページ *Bean*。これは、データを含み、Web ページの初期化、データ妥当性検査、およびイベント処理サービスを行う Java クラスです。

- <managed-bean> エlement。これは、プロジェクト内の JSF 構成ファイルに配置され、実行時にページ Bean を識別します。
- <navigation-rule> エlement。これは、JSF 出力 (ページ・ハンドラーの名前) を呼び出し先の JSP ファイルに関連付けるために、JSF アプリケーション構成ファイル内に作成されます。
- ページ・ハンドラーを保管した時点と同じ状態での JSP ファイル。

パーツ・ハンドラーによって使用されるすべてのデータ・テーブルおよびレコードも生成されます。

検証

JSP ベースの JSF タグがデータ変換、検証、またはイベント処理を行う場合、JSF ランタイムは、ユーザーが Web ページを処理依頼するとすぐに必要な処理を行います。エラーが検出された場合、JSF ランタイムは、ページ・ハンドラーに制御を渡さずに、ページを再表示することができます。ただし、ページ・ハンドラーが制御を受け取ると、ページ・ハンドラーは一連の EGL ベースの検証を行うことがあります。

EGL ベースの検証は、ページ・ハンドラーを宣言するときに以下の詳細を指定した場合に行われます。

- 個々の入力フィールドのエlementの編集 (最小入力長さなど)。
- 個々のフィールドの入力ベースの編集 (文字、数値)。
- 『*DataTable* パーツ』で説明しているような、個々の入力フィールドの *DataTable* の編集 (範囲、一致の有効、および一致の無効)。
- 個々の入力フィールドの編集機能。
- ページ・ハンドラー全体の編集機能。

ページ・ハンドラーは、ユーザーが値を変更した項目についてのみ、以下の順序で編集を監視します。

1. すべての基本的な入力ベースの編集 (一部が失敗した場合でも)
2. 前の編集が成功した場合、すべてのテーブルの編集 (一部が失敗した場合でも)
3. 前の編集が成功した場合、すべてのフィールド編集機能 (一部が失敗した場合でも)
4. 前の編集がすべて成功した場合、ページ・ハンドラー編集機能

ページ項目プロパティ **validationOrder** は、個々の入力編集され、フィールド・バリデーター関数が呼び出される順序を定義します。

validationOrder プロパティが指定されない場合、デフォルトは、ページ・ハンドラーに定義された項目の順序 (上から下) になります。ページ・ハンドラーの項目の一部で (すべてではなく) **validationOrder** が指定されている場合、**validationOrder** プロパティによる全項目の検証が指定順で最初に行われます。続いて、**validationOrder** プロパティを使用しない項目の検証が、ページ・ハンドラーの項目順 (上から下) で行われます。

ランタイム・シナリオ

このセクションでは、ユーザーと Web アプリケーション・サーバーとの実行時の対話についての技術的概要を示します。

ユーザーがページ・ハンドラーによってサポートされる JSP を呼び出すと、以下のステップが実行されます。

1. Web アプリケーション・サーバーは、以下のように環境を初期化します。
 - a. セッション・オブジェクトを構成して、ユーザーがアクセスするアプリケーションで複数の対話にわたってデータを保存します。
 - b. 要求オブジェクトを構成して、ユーザーの現在の対話のデータを保存します。
 - c. JSP を呼び出します。
2. JSP は、以下のように一連の JSF タグを処理して、Web ページを構成します。
 - a. ページ・ハンドラーのインスタンスの作成、ユーザー指定引数による `onPageLoad` 関数 (存在する場合) の呼び出し、および要求オブジェクトへのページ・ハンドラーの配置を行います。
 - b. Web ページに組み込むために、要求オブジェクトおよびセッション・オブジェクトに保管されているデータにアクセスします。

注: ページ・ハンドラーパーツには、`onPageLoadFunction` と呼ばれるプロパティがあります。これは、JSP の開始時に呼び出されるページ・ハンドラー関数を識別するものです。この関数は、渡されたユーザー提供の任意の引数を自動的に検索します。また、他のコードを呼び出すことや、追加のデータを Web アプリケーション・サーバーの要求またはセッション・オブジェクトに配置することができます。ただし、この関数は、他のページに制御を転送したり、ページをユーザーに最初に表示するときにエラー・メッセージを表示することはできません。

3. JSP はユーザーに Web ページを処理依頼し、Web アプリケーション・サーバーは、セッション・オブジェクトおよび JSP はそのまま、応答オブジェクトを破棄します。

スクリーン内のフィールドにあるユーザー提供のデータが HTML `<FORM>` タグと関連付けられており、書式を処理依頼する場合は、以下のステップが実行されます。

1. Web アプリケーション・サーバーは、以下のように環境を再初期化します。
 - a. 要求オブジェクトを構成します。
 - b. 処理依頼した書式の受信データを検証するため、ページ Bean に配置します。
 - c. JSP を再度呼び出します。
2. JSP は、以下のように一連の JSF タグを処理して、受信データをページ Bean に格納します。
3. ランタイム・ページ・ハンドラーは、以下のようにデータを検証します。
 - a. ページ・ハンドラーデータ宣言の指定に従って、比較的初歩的な編集 (最小入力長さなど) を行います。

- b. ページ・ハンドラーデータ宣言の指定に従って、項目に固有の任意の検証機能呼び出します。
- c. 別のフィールドの内容を基にして、単一のフィールドを少なくとも部分的に検証する場合、必要に応じてページ・ハンドラーバリデーター機能呼び出します。

(検証の詳細については、前のセクションを参照してください。)

- 4. エラーが発生した場合、EGL ランタイムがエラーを JSF キューに入れ、JSP は Web ページを組み込みメッセージとともに再表示します。エラーが発生しなかった場合の結果は、以下のようになります。
 - a. ページ Bean に保管されたデータは、レコード Bean に書き込まれる
 - b. 以降の処理は、ユーザーがクリックしたボタンまたはハイパーリンクに関連付けられている JSF タグで識別されるイベント・ハンドラーによって決定される

イベント・ハンドラーは、ランタイムの JSF ベースの構成ファイル内のマッピングを識別する JSF ラベルに処理を送ることができます。次に、呼び出すオブジェクトが JSP (通常は、EGL ページ・ハンドラーに関連付けられている JSP)、またはサーブレットのいずれであるのかを、このマッピングが識別します。

関連する概念

24 ページの『パーツの参照』
199 ページの『Web サポート』

関連する参照項目

205 ページの『EGL の Page Designer サポート』
732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
738 ページの『ページ・ハンドラー・フィールドのプロパティ』

JavaServer Faces のコントロールと EGL

JavaServer Faces (JSF) は、サーバー・サイド・ユーザー・インターフェース・コンポーネント・フレームワークです。簡単に言えば、JSF は Web ページ用のインターフェースを作成できる一連のツールとコンポーネントです。JSF コンポーネントは、Web ページにデータを表示し、ユーザーからの入力を受け入れることができます。

このトピックでは、JSF コンポーネントと EGL の関係について説明します。JSF の詳細については、『Faces アプリケーションの作成 - 概要 (Creating Faces applications - overview)』を参照してください。関連のチュートリアルを表示するには、「ヘルプ」>「チュートリアル・ギャラリー (Tutorials Gallery)」をクリックします。「実習 (Do and Learn)」を展開し、「JavaServer Faces で Web ページに動的情報を表示する (Display dynamic information on Web pages with JavaServer Faces)」を選択してください。

JSF コントロールを使用して Web ページに EGL データを表示するには、2 つの方法があります。

- ページ・データ・ビューにあるデータ項目から、または Page Designer ビューで作成したデータ項目から、自動的に JSF コントロールを作成できます。この方法

を使用するには、『*EGL* レコードと *Faces JSP* との関連付け (*Associating an EGL record with a Faces JSP*)』または『*EGL* データ項目の作成と *Faces JSP* への関連付け (*Creating an EGL data item and associating it with a Faces JSP*)』の説明に従い、「**EGL エlementを Web ページに表示するためのコントロールを追加する (Add controls to display the EGL element on the Web page)**」という名前のチェック・ボックスを選択します。

- JSF コントロールを手動で追加し、ページ・データ・ビュー内のデータへバインドすることができます。この方法を使用すると、デフォルトのレイアウトを使用せずに、ページ上の JSF コントロールのレイアウトをカスタマイズできます。この方法を使用するには、以下のいずれかのトピックを参照してください。
 - *EGL* ページ・ハンドラーへの *JavaServer Faces* 入出力コンポーネントのバインディング
 - *EGL* ページ・ハンドラーへの *JavaServer Faces* チェック・ボックス・コンポーネントのバインディング
 - *EGL* ページ・ハンドラーへの *JavaServer Faces* 単一選択コンポーネントのバインディング
 - *EGL* ページ・ハンドラーへの *JavaServer Faces* 複数選択コンポーネントのバインディング

ページ・ハンドラー内の *EGL* 関数を JSF コントロールにバインドすることもできます。『*EGL* ページ・ハンドラーへの *JavaServer Faces* 単一選択コンポーネントのバインディング』を参照してください。

関連するタスク

- 『*EGL* フィールドの作成と *Faces JSP* との関連付け』
- 214 ページの『*EGL* レコードと *Faces JSP* との関連付け』
- 215 ページの『*EGL* ページ・ハンドラーへの *JavaServer Faces* コマンド・コンポーネントのバインディング』
- 217 ページの『*EGL* ページ・ハンドラーへの *JavaServer Faces* 入出力コンポーネントのバインディング』
- 218 ページの『*EGL* ページ・ハンドラーへの *JavaServer Faces* チェック・ボックス・コンポーネントのバインディング』
- 219 ページの『*EGL* ページ・ハンドラーへの *JavaServer Faces* 単一選択コンポーネントのバインディング』
- 221 ページの『*EGL* ページ・ハンドラーへの *JavaServer Faces* 複数選択コンポーネントのバインディング』

関連する参照項目

- 205 ページの『*EGL* の Page Designer サポート』

EGL フィールドの作成と *Faces JSP* との関連付け

EGL プリミティブ・フィールドを作成し、それを *Faces JSP* に関連付けるには、次のようにします。

1. Page Designer で *Faces JSP* ファイルを開く。JSP ファイルを開くには、プロジェクト・エクスプローラーでその JSP ファイルをダブルクリックしてください。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。

注: 「設計」ビュー (または「ソース」ビュー) を右クリックして「ページ・コードの編集」をクリックすることによって、関連するページ・ハンドラーにアクセスすることができます。

2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. 「パレット」ビューで「EGL」ドロワーをクリックし、EGL データ・オブジェクト・タイプを表示する。
4. 「新規フィールド」をパレットから JSP にドラッグする。「EGL データ・フィールドの新規作成」ダイアログが表示されます。
5. フィールド名を「名前」フィールドに入力する。
6. 「型」ドロップダウン・リストからフィールドの型を選択し、フィールドのプリミティブ特性 (長さ、および場合によっては小数部) を指定する必要がある場合は、「大きさ (Dimensions)」テキスト・ボックスに情報を入力する。以下の型の項目を宣言する場合は、デフォルト・マスクが使用されます。
 - 日付 (マスク `yyyymmdd`)
 - 時刻 (マスク `hhmmss`)
 - タイム・スタンプ (マスク `yyyymmddhhmmss`)

型として `DataItem` パーツを指定したい場合は、リストの最後の値である **DataItem** を選択してください。この場合は、「DataItem パーツの選択」ダイアログが表示されるので、`DataItem` パーツをリストから選択するか、または名前を入力してから「OK」をクリックします。

7. データ項目の配列を作成する場合は、「配列」チェック・ボックスを選択し、「サイズ」テキスト・ボックスに整数を入力する。
8. ページ上にフィールドを組み込みたくない場合は、「Web ページに EGL エlementを表示するためのコントロールを追加する (Add controls to display the EGL element on the Web page)」チェック・ボックスをクリアし、「OK」をクリックする。このフィールドが、「ページ・データ」ビューで選択可能になります。後で、「ページ・データ」ビューから JSP にドラッグすることにより、JSP ファイルにそれを追加できます。
9. JSP ファイルにフィールドを組み込みたい場合は、以下の追加手順を行う。
10. 「Web ページに EGL エlementを表示するためのコントロールを追加する (Add controls to display the EGL element on the Web page)」というチェック・ボックスを選択し、「OK」をクリックする。「挿入のコントロール」ウィンドウが開きます。
11. 「挿入のコントロール」ウィンドウで、意図したフィールドの用途を示すラジオ・ボタンを選択する。
 - 出力用 (「既存レコードの表示」)
 - 入出力用 (「既存レコードの更新」)
 - 入力用 (「新規レコードの作成」)

この選択によって、使用可能なコントロールのタイプが影響を受けます。

12. フィールド・ラベルを変更するには、フィールド名の隣に表示されるラベルを選択し、新しい内容を入力する。

13. 示されたものと異なるコントロール・タイプを選択するには、「コントロール・タイプ」リストからタイプを選択する。
14. 「オプション」をクリックすると、「オプション」ダイアログが表示される。使用可能な固有のオプションは、フィールドを入力、出力、入出力のどれに使用するかによって決まります。いずれの場合でも、1 つのオプションは、フィールド・ラベルを囲む JSF タグ `<h:outputLabel>` を含めるか含めないかです。

「オプション」ダイアログでの作業が済んだならば、「OK」をクリックします。
15. 「完了」をクリックする。

関連する参照項目

205 ページの『EGL の Page Designer サポート』

37 ページの『プリミティブ型』

EGL レコードと Faces JSP との関連付け

EGL レコードを Faces JSP に関連付ける手順は、次のとおりです。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP ファイルが開かれます。「設計」タブをクリックして「設計」ビューにアクセスします。

注: 「設計」ビュー (または「ソース」ビュー) を右クリックして「ページ・コードの編集」をクリックすることによって、関連するページ・ハンドラーにアクセスすることができます。

2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. 「パレット」ビューで「EGL」ドロワーをクリックし、EGL データ・オブジェクト型を表示する。
4. 「レコード」をパレットから JSP ページにドラッグする。「レコード・パーツの選択」ダイアログが表示されます。
5. リストからレコードを選択する。
6. フィールド名を指定するか、デフォルト値を受け入れる。これはレコード・パーツ名です。
7. レコードの配列を宣言する場合は、「配列」チェック・ボックスを選択し、「サイズ」テキスト・ボックスに整数を入力する。
8. ページ上にレコードを組み込みたくない場合は、「Web ページに EGL エlementを表示するためのコントロールを追加する (Add controls to display the EGL element on the Web page)」チェック・ボックスをクリアし、「OK」をクリックする。これで、このレコードが、「ページ・データ」ビューで選択可能になります。後で、「ページ・データ」ビューから JSP にドラッグすることにより、JSP ファイルにそれを追加できます。
9. JSP ファイルにフィールドを組み込みたい場合は、以下の追加手順を行う。

10. 「Web ページに EGL エlementを表示するためのコントロールを追加する (Add controls to display the EGL element on the Web page)」チェック・ボックスを選択し、「OK」をクリックする。「挿入のコントロール」ウィンドウが開きます。
11. 「挿入のコントロール」ウィンドウで、意図したフィールドの用途を示すラジオ・ボタンを選択する。
 - 出力用 (「既存レコードの表示」)
 - 入出力用 (「既存レコードの更新」)
 - 入力用 (「新規レコードの作成」)

この選択によって、使用可能なコントロールのタイプが影響を受けます。
12. フィールドの順序を変更するには、上下の矢印を使用する。
13. リストされたフィールドのサブセットだけを選択したい場合は、「なし (None)」をクリックし、必要なフィールドを選択する。すべてのフィールドを選択するには、「すべて」をクリックします。
14. 各フィールドについて、以下のようにします。
 - a. フィールドを除外するには、関連するチェック・ボックスをクリアする。フィールドを組み込むには、チェック・ボックスが選択されていることを確認します。
 - b. フィールド・ラベルを変更するには、フィールド名の隣に表示されるラベルを選択し、新しい内容を入力する。
 - c. 示されたものと異なるコントロール・タイプを選択する (可能な場合) には、タイプのリストから選択を行う。
15. 「オプション」をクリックすると、「オプション」ダイアログが表示される。使用可能な特定のオプションは、フィールドを入力、出力、入出力のどれに使用するかによって決まります。いずれの場合でも、1 つのオプションは、フィールド・ラベルを囲む JSF タグ `<h:outputLabel>` を含めるか含めないかです。

「オプション」ダイアログでの作業が済んだならば、「OK」をクリックします。

16. 「完了」をクリックする。

関連する概念

141 ページの『レコード・パーツ』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

EGL ページ・ハンドラーへの JavaServer Faces コマンド・コンポーネントのバインディング

JavaServer Faces コマンド・コンポーネント (ボタンまたはハイパーテキスト・リンク) を EGL ページ・ハンドラー関数にバインドするには、次のようにします。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。

2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. パレット・ビューで、「**Faces コンポーネント**」ドロワーをクリックして、Faces コンポーネントのオブジェクト型を表示する。
4. コマンド・コンポーネントをパレットから JSP にドラッグする。コマンド・コンポーネントは、ラベルの中に **Command** というワードを含んでいます。コンポーネント・オブジェクトが JSP 上に配置されます。
5. 次の方法のいずれかを使用して、イベント・ハンドラーをコマンド・コンポーネントにバインドする。
 - コンポーネントを既存のイベント・ハンドラーへバインドするには、そのイベント・ハンドラーを「ページ・データ」ビュー内の「アクション」ノードから JSP 上のコンポーネント・オブジェクトへドラッグする。
 - コンポーネントにバインドされた新しいイベント・ハンドラーを作成するには、次のいずれかを行う。
 - a. コンポーネントを右クリックし、ポップアップ・メニューから「**イベントの編集**」をクリックする。
 - b. 「クイック編集」ビューを使用して、イベント・ハンドラーの EGL コードを入力する。「クイック編集」ビューの使用の詳細については、『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』を参照してください。

イベント・ハンドラーが Page Data ビューの中で可視となり、対応する関数とそのページ・ハンドラーへ追加されます。詳細については、『EGL ソース形式のページ・ハンドラー・パーツ』を参照してください。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

ページ・ハンドラー・コード用の「クイック編集」ビューの使用

「クイック編集」ビューを使用すると、ページ・ハンドラー・ファイルを開かずに JSP サーバー・イベントの EGL ページ・ハンドラー・コードを保守できます。

「クイック編集」ビューを使用するには、次のようにします。

1. Page Designer で JSP ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。
2. Page Designer で右クリックし、「**イベントの編集**」を選択する。「クイック編集」ビューが開きます。

3. 以下の手順を実行してコマンド・コンポーネント用のページ・ハンドラー関数を保守する。
 - a. JSP でコマンド・コンポーネントを選択する。
 - b. コマンド・コンポーネントに関連付けられているページ・ハンドラー関数がすでにコマンド・コンポーネントに存在する場合は、「クイック編集」ビューのスクリプト・エディター (右のペイン) 内にこの関数が表示される。コードに対して行った変更は、ページ・ハンドラーに反映されます。
 - c. 選択したコマンド・コンポーネントに関するページ・ハンドラー関数を作成するには、「クイック編集」ビューのイベント・ペイン (左のペイン) 内の「**コマンド**」をクリックし、「クイック編集」ビューのスクリプト・エディター (右のペイン) でクリックする。関数が表示されます。関数のページ・ハンドラー・コードを入力します。
4. 以下の手順を実行して、onPageLoad 関数を保守する。
 - a. JSP 内をクリックする。
 - b. 「クイック編集」ビューのイベント・ペイン (左のペイン) で「**onPageLoad**」をクリックする。
 - c. 「クイック編集」ビューのスクリプト・エディター (右のペイン) に onPageLoad 関数が表示される。コードに対して行った変更は、ページ・ハンドラーに反映されます。

関連する概念

207 ページの『ページ・ハンドラー』

関連する参照項目

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL ページ・ハンドラーへの JavaServer Faces 入出力コンポーネントのバインディング

JavaServer Faces 入出力コンポーネントを既存のページ・ハンドラー・データ域にバインドするには、次のようにします。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。
2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. パレット・ビューで、「**Faces コンポーネント**」ドロワーをクリックして、Faces コンポーネントのオブジェクト型を表示する。
4. 入出力コンポーネントをパレットから JSP にドラッグする。入出力コンポーネントには、ラベルに **Input** および **Output** のワードが入っています。コンポーネント・オブジェクトが JSP 上に配置されます。
5. コンポーネントを既存のページ・ハンドラー・データ域にバインドするために、次のいずれかを行う。
 - データ域を「ページ・データ」ビューから JSP 上のコンポーネント・オブジェクトへドラッグする。

- JSP 上でコンポーネント・オブジェクトを選択してから、「ページ・データ」ビュー内のデータ域を右クリックし、「バインド先コンポーネント名」を選択する。
- JSP 上でコンポーネント・オブジェクトを選択する。「プロパティ」ビューの「値」フィールドの隣にあるボタンをクリックし、「ページ・データ・オブジェクトの選択」リストからデータ域を選択して「OK」をクリックします。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL ページ・ハンドラーへの JavaServer Faces チェック・ボックス・コンポーネントのバインディング

JavaServer Faces チェック・ボックス・コンポーネントの独特な点は、バインド先データ域の項目プロパティ **isBoolean** (旧称 **boolean**) が **yes** に設定されている必要があることです。ブール・データ域宣言の例を以下に示します。

```
DataItem CharacterBooleanItem char(1)
{
    value = "N",
    isBoolean = yes
}
end

DataItem NumericBooleanItem smallInt
{
    value = "0",
    isBoolean = yes
}
end
```

JavaServer Faces チェック・ボックス・コンポーネントを既存の EGL ページ・ハンドラー・データ域にバインドするには、次のようにします。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。
2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. パレット・ビューで、「**Faces** コンポーネント」ドロワーをクリックして、Faces コンポーネントのオブジェクト型を表示する。
4. チェック・ボックス・コンポーネントをパレットから JSP にドラッグする。コンポーネント・オブジェクトが JSP 上に配置されます。
5. コンポーネントを既存のページ・ハンドラー・データ域にバインドするために、次のいずれかを行う。

- データ域を「ページ・データ」ビューから JSP 上のコンポーネント・オブジェクトへドラッグする。
- JSP 上でコンポーネント・オブジェクトを選択してから、「ページ・データ」ビュー内のデータ域を右クリックし、「バインド先コンポーネント名」を選択する。
- JSP 上でコンポーネント・オブジェクトを選択する。「プロパティ」ビューの「値」フィールドの隣にあるボタンをクリックし、「ページ・データ・オブジェクトの選択」リストからデータ域を選択して「OK」をクリックします。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL ページ・ハンドラーへの JavaServer Faces 単一選択コンポーネントのバインディング

単一選択コンポーネントを使用すると、値のリストから 1 つの選択を行うことができます。ユーザーの選択は、ページ・ハンドラー・データ域に保管されます。ラジオ・ボタン、単一選択リスト・ボックス、およびコンボ・ボックスは、JavaServer Faces 単一選択コンポーネントです。

バインディングとは、コンポーネントとデータ域の関係のことです。コンポーネントをデータ域にバインドするには、事前にそのデータ域をページ・ハンドラーで宣言しておく必要があります。単一選択コンポーネントには、次の 2 種類のバインディングが必要です。

- ユーザーが選択できる値が入った 1 つ以上のデータ域へのバインディング
- ユーザーの選択を受け取るデータ域へのバインディング

JavaServer Faces 単一選択コンポーネントを既存の EGL ページ・ハンドラー・データ域にバインドするには、次のようにします。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。
2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. パレット・ビューで、「Faces コンポーネント」ドロワーをクリックして、Faces コンポーネントのオブジェクト型を表示する。
4. 単一選択コンポーネントをパレットから JSP にドラッグする。コンポーネント・オブジェクトが JSP 上に配置されます。

5. ユーザーに対して表示したい値が入っている 1 つ以上のページ・ハンドラー・データ域にコンポーネントをバインドするために、次のいずれかの手順を実行する。
 - リスト項目が 1 つずつ入った個々のページ・ハンドラー・データ域に、コンポーネントをバインドできます。それぞれのデータ域について、次の手順を実行します。
 - a. JSP 上でオブジェクト・コンポーネントを選択する。
 - b. 「プロパティ」ビューで、「**選択項目の追加**」をクリックする。「名前」および「値」フィールドにデフォルト値が取り込まれます。
 - c. 「**名前**」フィールドをクリックし、ユーザーに対して表示したいテキストを入力する。
 - d. 「**値**」フィールドをクリックし、「**値**」フィールドの隣にあるボタンをクリックする。「ページ・データ・オブジェクトの選択」リストから個々のデータ域を選択して「**OK**」をクリックします。このデータ域は、後で受け取り側データ域へ移動される値を保持します。
 - ユーザーに対して表示したい複数の値が入っている 1 つのページ・ハンドラー配列データ域にコンポーネントをバインドできます。コンポーネントを配列データ域へバインドするには、次の手順を実行します。
 - a. JSP 上でオブジェクト・コンポーネントを選択する。
 - b. 「プロパティ」ビューで、「**選択項目のセットを追加**」をクリックする。「名前」および「値」フィールドにデフォルト値が取り込まれます。
 - c. 「**値**」フィールドをクリックし、「**値**」フィールドの隣にあるボタンをクリックする。「ページ・データ・オブジェクト (Page Data Object)」リストから配列データ域を選択して「**OK**」をクリックします。配列データ域内の値は、ユーザーに対して表示される値です。後で説明するプロパティによって、配列データ域内の値、またはそれと等価のインデックス値が、受け取り側データ域へ移動されるかどうかが決まります。
6. ユーザーに対して表示される値を提供するために配列データ域を使用する場合は、**selectFromListItem** と **selectType** の 2 つのプロパティを使用して受け取り側データ域を定義する必要があります。 **selectFromListItem** プロパティは、リスト項目を保持する配列を指します。 **selectType** プロパティは、受け取り側データ域にテキスト値またはインデックス値のどちらを取り込むかを示します。受け取り側データ域の例を以下に示します。

```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};

colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
 selectType = index};
```

7. コンポーネントを、ユーザーの選択を受け取るページ・ハンドラー・データ域にバインドするために、次のいずれかを行う。
 - データ域を「ページ・データ」ビューから JSP 上のコンポーネント・オブジェクトへドラッグする。
 - JSP 上でコンポーネント・オブジェクトを選択してから、「ページ・データ」ビュー内のデータ域を右クリックし、「**バインド先コンポーネント名**」を選択する。

- JSP 上でコンポーネント・オブジェクトを選択する。「プロパティ」ビューの「値」フィールドの隣にあるボタンをクリックし、「ページ・データ・オブジェクトの選択」リストからデータ域を選択して「OK」をクリックします。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL ページ・ハンドラーへの JavaServer Faces 複数選択コンポーネントのバインディング

複数選択コンポーネントを使用すると、値のリストから 1 つまたは複数の選択を行うことができます。ユーザーの選択は、ページ・ハンドラー配列データ域に保管されます。チェック・ボックス・グループと複数選択リスト・ボックスは、JavaServer Faces 複数選択コンポーネントです。

バインディングとは、コンポーネントとデータ域の関係のことです。コンポーネントをデータ域へバインドするには、事前にそのデータ域をページ・ハンドラーで宣言しておく必要があります。複数選択コンポーネントには、次の 2 種類のバインディングが必要です。

- ユーザーが選択できる値が入った 1 つ以上のデータ域へのバインディング
- ユーザーの選択を受け取る配列データ域へのバインディング

JavaServer Faces 複数選択コンポーネントを既存の EGL ページ・ハンドラー・データ域にバインドするには、次のようにします。

1. Page Designer で Faces JSP ファイルを開く。JSP ファイルを開いていない場合は、プロジェクト・エクスプローラーで JSP ファイルをダブルクリックします。Page Designer で JSP が開きます。「設計」タブをクリックして「設計」ビューにアクセスします。
2. 「ウィンドウ」メニューで、「ビューの表示」 > 「その他」 > 「基本」 > 「パレット」を選択する。
3. パレット・ビューで、「Faces コンポーネント」ドロワーをクリックして、Faces コンポーネントのオブジェクト型を表示する。
4. 複数選択コンポーネントをパレットから JSP にドラッグする。コンポーネント・オブジェクトが JSP 上に配置されます。
5. ユーザーに対して表示したい値が入っている 1 つ以上のページ・ハンドラー・データ域にコンポーネントをバインドするために、次のいずれかの手順を実行する。
 - 1 つずつリスト項目が入った個々のページ・ハンドラー・データ域へコンポーネントをバインドできます。それぞれのデータ域について、次の手順を実行します。

- a. JSP 上でオブジェクト・コンポーネントを選択する。
 - b. 「プロパティ」ビューで、「**選択項目の追加**」をクリックする。「名前」および「値」フィールドにデフォルト値が取り込まれます。
 - c. 「名前」フィールドをクリックし、ユーザーに対して表示したいテキストを入力する。
 - d. 「値」フィールドをクリックし、「値」フィールドの隣にあるボタンをクリックする。「ページ・データ・オブジェクトの選択」リストから個々のデータ域を選択して「**OK**」をクリックします。このデータ域は、後で受け取り側データ域へ移動される値を保持します。
- ユーザーに対して表示したい複数の値が入っている 1 つのページ・ハンドラー配列データ域へコンポーネントをバインドできます。コンポーネントを配列データ域へバインドするには、次の手順を実行します。
 - a. JSP 上でオブジェクト・コンポーネントを選択する。
 - b. 「プロパティ」ビューで、「**選択項目のセットを追加**」をクリックする。「名前」および「値」フィールドにデフォルト値が取り込まれます。
 - c. 「値」フィールドをクリックし、「値」フィールドの隣にあるボタンをクリックする。「ページ・データ・オブジェクトの選択」リストから配列データ域を選択して「**OK**」をクリックします。配列データ域内の値は、ユーザーに対して表示される値です。後で説明するプロパティによって、配列データ域内の値、またはそれと等価のインデックス値が、受け取り側データ域へ移動されるかどうかが決まります。
6. ユーザーに対して表示される値を提供するために配列データ域を使用する場合は、**selectFromListItem** と **selectType** の 2 つのプロパティを使用して受け取り側データ域を定義する必要があります。 **selectFromListItem** プロパティは、リスト項目を保持する配列を指します。 **selectType** プロパティは、受け取り側データ域にテキスト値またはインデックス値のどちらを取り込むかを示します。受け取り側データ域の例を以下に示します。


```
colorSelected char(10)
{selectFromListItem = "colorsArray",
 selectType = value};

colorSelectIdx smallInt
{selectFromListItem = "colorsArray",
 selectType = index};
```
 7. ユーザーの選択を受け取るページ・ハンドラー配列データ域にコンポーネントをバインドするために、次のようにする。
 - a. JSP 上でコンポーネント・オブジェクトを選択する。
 - b. 「プロパティ」ビューの「値」フィールドの隣にあるボタンをクリックする。
 - c. 「ページ・データ・オブジェクトの選択」リストからデータ域を選択する。
 - d. 「**OK**」をクリックする。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

関連する参照項目

205 ページの『EGL の Page Designer サポート』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

EGL レポートの作成

EGL レポートの概要

EGL は、オープン・ソースで Java ベースのレポート・ライブラリーである JasperReports の機能に基づいたレポートを作成できます。このライブラリーについて詳しくは、次の Web サイトを参照してください。

<http://jasperreports.sourceforge.net>

EGL は、報告書レイアウトに対するメカニズムは提供していません。次の操作を実行する必要があります。

- JasperReports 出力ファイル (ファイル拡張子は *jasper*) をインポートするか、または
- テキスト・エディターまたは特殊ツールを使用して、Workbench の「プロジェクト > すべてをビルド」をクリックしたときに JasperReports 出力ファイルに変換される設計ファイルを作成するかします。

以下は、設計ファイルの作成に使用する 2 つの特殊ツールです。

- この Web サイトで説明している JasperAssistant:

<http://www.jasperassistant.com>

- この Web サイトで説明している iReport:

<http://ireport.sourceforge.net>

ドライブ・レポート・プロダクションに書き込む EGL ファイルでは、データを JasperReports 出力ファイルに送り (またはそのファイルで指定されたデータ・ソースを受け入れ)、次に HTML または Adobe Acrobat PDF などの異なるフォーマットにすることの可能な、1 つまたは複数の出力ファイルにレポートをエクスポートします。

また、タイプ JasperReport の EGL ハンドラーをコード化する場合、レポートにデータを入力する段階で発生するユーザー・イベントに応答できます。たとえば、レポートの作成がほぼ完了したときに、そのレポートに実行固有の詳細を追加できます。ただし、イベント処理が確実に実行されるようにするためには、レポート・ハンドラーから生成された出力を JasperReport 出力ファイルが参照していることを確認する必要があります。

「EGL レポート・ハンドラー・ウィザード」を使用すると、簡単に EGL レポート・ハンドラーを作成できます。

レポートと対話する EGL コードを書き込む場合、システム・ライブラリー **ReportLib** にある機能を使用します。また、レポートを作成するコードでは、タイプ Report および ReportData の変数を作成します。

ここで取り上げた EGL パーツ (ハンドラー、レポート、レポート・データ) は、すべて定義済みです。

関連する概念

EGL レポート作成プロセスの概要

関連する参照項目

EGL レポート・ライブラリー

データ・ソース (Data sources)

EGL レポート・ハンドラー

EGL レポート作成プロセスの概要

このトピックでは、EGL プロジェクト用にレポートを作成および生成するための一般的なプロセスについて、その概要を示します。これらのプロセスに関する追加の詳細は、EGL レポート・タスクのヘルプ・トピックに記載されています。

レポートを作成するには、以下に述べる 3 つのプロセスを実行します。このうちの 2 つのプロセスは、XML 設計の作成とレポートを駆動するコードの作成で、これらは必須のプロセスです。第 3 のプロセスであるレポート・ハンドラーの作成は、オプションです。これらのプロセスは、説明されている順序で実行しなくてもかまいません。例えば、レポート・ハンドラーが必要な場合は、それを作成してから XML 設計文書を作成するか、設計文書とレポート・ハンドラーの作成を同時に行うことができます。ただし、その例外が、以下に示すステップ 2 の『レポート・ハンドラーと XML 設計文書のコードの相互関係』で説明されています。

XML 設計文書と、レポートを駆動するためのコードがないと、レポートを生成できません。

レポートを作成するために実行する 3 つのプロセスは、以下のとおりです。

1. XML 設計文書を作成して、レポートのレイアウト情報を指定します。この文書は、次のどちらの方法でも作成できます。
 - サード・パーティーの JasperReport 設計ツール (Jasper Assistant や iReports など) を使用する。
 - テキスト・エディターを使用して、新しいテキスト・ファイルに Jasper XML 設計情報を書き込む。

XML 設計文書は、`.jrxml` という拡張子を持っている必要があります。作成したファイルにこの拡張子が付いていない場合は、そのファイルを `.jrxml` ファイルとして名前変更してください。また、XML 設計文書は、必ず EGL レポート・ハンドラーやレポート起動コード・ファイルと一緒に、同じ EGL パッケージに保管してください。

作成した `.jrxml` ファイルは、`.jasper` ファイルへコンパイルされます。新しい `.jrxml` ファイルを作成しない場合は、前にコンパイルされた `.jasper` ファイルをインポートする必要があります。

2. レポート作成時のイベント処理ロジックを提供するレポート・ハンドラーを使用したい場合は、次のいずれかの方法でレポート・ハンドラーを作成できます。

- EGL レポート・ハンドラー・ウィザードを使用して、レポート・ハンドラー用の情報を指定する。
- 新規 EGL ソース・ファイルを作成し、レポート・ハンドラー・テンプレートを使用してハンドラーを挿入するか、手動でハンドラー・コードを入力する。

レポート・ハンドラーと XML 設計文書のコードの相互関係 .jrxml ファイルの中で、EGL レポート・ハンドラーから生成されたレポート・ハンドラー・ファイルを参照する `scriptletClass` を指定できます。以下の点に留意してください。

- .jrxml ファイルが、レポート・ハンドラーによって生成された Java コードを使用する場合は、.jrxml ファイルを作成する前に、レポート・ハンドラーを生成する必要があります。
 - レポート・ハンドラーを変更した場合は、.jrxml ファイルを再コンパイルする必要があります。
 - .jrxml ファイル内のコンパイル・エラーを解決する必要がある場合、または、レポート・ハンドラーに変更を加えた後に .jasper ファイルを再コンパイルしたい場合は、.jrxml ファイルを変更して保管する必要があります。
3. EGL ReportLib 関数を使用して、EGL プロジェクト内にレポート起動コードを作成します。レポート起動コードを作成するときは、EGL プログラム・パーツ・ウィザードを使用できます。

重要: レポート・ハンドラーおよびレポート起動コード・ファイルには、XML 設計文書と異なる名前を付ける必要があります。そのようにしなかった場合は、設計ファイルのコンパイルによって、Java コードが上書きされます。問題を回避するために、レポート・ハンドラーには `reportName_handler.egl` という名前を付け、XML 設計文書には `reportName_XML.jrxml` という名前を付けてください。例えば、レポート・ハンドラーに `abc_handler.egl`、設計文書に `abc_XML.jrxml` という名前を付けることができます。また、XML 設計ファイルには必ず固有の名前を付け、EGL プログラム・ファイルと競合しないようにしてください。

XML 設計文書、レポート・ハンドラー (使用したい場合)、およびレポート起動コードを作成した後、レポートのビルドと生成を行うには、以下のプロセスを実行します。

1. 「プロジェクト」>「すべてをビルド」を選択して EGL プロジェクトをビルドします。

EGL は、自動的に EGL レポート・ハンドラーから Java コードを生成し、XML 設計文書 (.jrxml ファイル) を .jasper ファイルへコンパイルします。

2. レポート呼び出しコードを持つ EGL プログラムを実行する。

EGL プログラムが実行された後、EGL によって使用された JasperReports プログラムは、生成されたレポートを、レポート起動コード内の `reportDestinationFileName` によって指定されたロケーションに自動的に保管します。

レポートを生成する JasperReports プログラムは、中間ファイル・フォーマットである .jprint ファイルも生成して保管します。このファイルがエクスポートされて、最終的なレポート・フォーマット (.pdf、.html、.xml、.txt、または .csv) になります。

プログラムは、1 つの .jprint を何度もエクスポートに再利用できます。

レポート起動コード内の `exportReport()` 関数により、EGL はレポートを指定されたフォーマットでエクスポートします。例えば、次のコードを指定すると、EGL はレポートを `.pdf` フォーマットでエクスポートします。

```
reportLib.exportReport(myReport, ExportFormat.pdf);
```

EGL は、エクスポートされたレポートを自動ではリフレッシュしません。レポートの設計を変更した場合、またはデータが変更された場合は、あらためてレポートへの埋め込みを行い、レポートを再エクスポートする必要があります。

関連する概念

EGL レポートの概要

関連するタスク

パッケージへの設計文書の追加 (Adding a design document to a package)

レポート・テンプレートの使用

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーの手操作による作成

レポートをドライブするコードの作成

レポートの実行

レポートのエクスポート

EGL でのコンテンツ・アシストの使用

関連する参照項目

EGL レポート・ライブラリー

データ・ソース (Data sources)

EGL レポート・ハンドラー

データ・ソース

EGL Report ライブラリーには、データを含む基本データ・ソースの参照、またはデータの取得方法についての情報の参照が含まれています。

次の文を使用して、データ・ソース情報を指定できます。

- *DataSource.databaseConnection*
- *DataSource.sqlStatement*
- *DataSource.reportData*

たとえば、*ReportLib.fillReport* 関数を使用する際に *fillReport (eglReport, DataSource.databaseConnection)* を指定する場合、EGL はデータベース接続を検索し、それを JasperReports エンジンに渡します。

データ・ソースとしてデータベース接続、レポート・データ、および SQL ステートメントを使用してレポートを生成する方法の例については、『EGL レポート・ドライバ関数のサンプル・コード』を参照してください。

関連する概念

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

922 ページの『EGL ライブラリー ReportLib』

925 ページの『fillReport()』

234 ページの『EGL レポート・ドライバ関数用のサンプル・コード』

ライブラリー内のデータ・レコード

EGL ライブラリーには、*ReportData* レコードと *Report* レコードが含まれています。

ReportData レコードには、レポートで使用する特定セットのデータの情報が入っています。このレコードには、次のフィールドが含まれています

フィールド	説明	データ・タイプ
<i>connectionName</i>	EGL プログラムで確立される接続の名前。	String
<i>sqlStatement</i>	EGL が実行する必要がある SQL ステートメント。レポート・データは、ステートメントの実行の結果から得られます。	String
<i>Data</i>	レコードの動的配列。	Any (これは EGL Any 型です。)

Report レコードには、特定のレポートの情報が入っています。このレコードには、次のフィールドが含まれています

フィールド	説明	データ・タイプ
<i>reportDesignFile</i>	.jrxml 拡張子を持つ XML ファイルであるレポート設計ファイルの名前。	String
<i>reportDestinationFile</i>	.jrprint ファイルのロケーション。	String
<i>reportExportFile</i>	保管されている最終の .xml、.pdf、.html、.txt、または.csv ファイルのロケーション。	String
<i>reportData</i>	レポートの基本データ・ソースとして使用されるレポート・データ。	

関連する概念

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

922 ページの『EGL ライブラリー ReportLib』

EGL レポート・ハンドラー

EGL レポート・ハンドラーは、レポートにデータを埋め込むときに発生するイベント処理のための追加機能を提供します。「新規 EGL レポート・ハンドラー (New EGL Report Handler)」ウィザードを使用して、レポート・ハンドラーの情報を指定できます。あるいは、手操作でレポート・ハンドラーを作成することもできます。

レポート・ハンドラー・ファイルが生成されると、EGL は次のファイルを作成します。

- *handlerName.java*
- *handlerName_lib.java* ファイル

handlerName

EGL レポート・ハンドラーの別名。

EGL が .java ファイルを生成する場合、クラス名は小文字です。XML 設計文書に入力されたすべてのクラス名が小文字であることを確認してください。

レポート・ハンドラー・コードのサンプル構文および例については、『EGL レポート・ハンドラーの手操作による作成』を参照してください。

技術的詳細情報: EGL レポート・ハンドラーは、JasperReport 型の EGL ハンドラー・パーツです。このレポート・ハンドラーは、JasperReports スクリプトレット・クラスにマップされます。レポート・ハンドラー Java 生成は、JRDefaultScriptlet クラスを拡張し、スクリプトレット関数を表す生成済み Java 関数を含む Java クラスを定義します。XML 設計文書の定義セクションには、スクリプトレット・クラスの名前が入っています。JasperReports エンジンでは、スクリプトレット・クラスをロードし、レポート定義で定義されたさまざまなメソッドを呼び出します。

(JasperReports スクリプトレットおよびスクリプトレット・クラスの詳細については、JasperReports 資料を参照してください。)

レポート・ハンドラーは、要求に応じて戻される *ReportData* レコードの内部リストを保持します。

関連する概念

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連するタスク

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

238 ページの『EGL レポート・ハンドラーの手操作による作成』

242 ページの『レポートをドライブするコードの作成』

関連する参照項目

232 ページの『追加の EGL レポート・ハンドラー関数』

922 ページの『EGL ライブラリー ReportLib』

231 ページの『事前定義されたレポート・ハンドラー関数』

事前定義されたレポート・ハンドラー関数

レポート・ハンドラーは、関数テンプレートとして使用できる次の事前定義関数を提供します。

関数	関数が作動する場合
<code>beforeReportInit();</code>	レポートの初期化前
<code>afterReportInit();</code>	レポートの初期化後
<code>beforePageInit();</code>	ページに入るとき
<code>afterPageInit();</code>	ページから出るとき
<code>beforeColumnInit();</code>	列の初期化前
<code>afterColumnInit();</code>	列の初期化後
<code>beforeGroupInit (groupName String);</code>	グループの初期化前。 <i>groupName</i> は、レポート内のグループの名前です。
<code>afterGroupInit(groupName String);</code>	グループの初期化後
<code>beforeDetailEval();</code>	すべてのフィールドの前。この関数が設定されると、各行は、印刷前にこの関数を呼び出します。
<code>afterDetailEval();</code>	すべてのフィールドの後。この関数が設定されると、各行は、印刷前にこの関数を呼び出します。

上記の関数のいずれかで、それ以外の関数を呼び出すことができます。たとえば、次のように `setReportVariable()` を呼び出すことができます。

```
function afterGroupInit(groupName String)
  if (groupName == "cat")
    setReportVariableValue ("NewGroupName", "dog");
  else
    setReportVariableValue ("NewGroupName", groupName);
  end
end
```

また、独自の関数を作成することもできます。カスタム関数の作成に関する情報については、JasperReports 資料を参照してください。

事前定義されたレポート・ハンドラー関数の使用例の詳細については、『EGL レポート・ハンドラーの手操作による作成』を参照してください。

関連する概念

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連するタスク

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

238 ページの『EGL レポート・ハンドラーの手操作による作成』

関連する参照項目

232 ページの『追加の EGL レポート・ハンドラー関数』

229 ページの『ライブラリー内のデータ・レコード』

追加の EGL レポート・ハンドラー関数

事前定義されたレポート・ハンドラー関数内から次の ReportLib 関数のいずれかを呼び出すことができます。

レポート・パラメーターの取得用の関数

関数	用途
getReportParameter (<i>parameter</i> String <u>in</u>)	埋め込み中のレポートから、指定のパラメーターの値を返します。

レポート変数の設定と取得用の関数

これらの変数をさまざまな理由で使用できます。たとえば、頻繁に使用される式を保管するため、または処理中の行で定義されている式で複雑な計算を実行するためなどです。

関数	用途
getReportVariableValue (<i>variable</i> String <u>in</u>)	埋め込み中のレポートから、指定の変数の値を返します。戻り値は ANY 型です。
setReportVariableValue (<i>variable</i> String <u>in</u> , <i>value</i> Any <u>in</u>);	指定の変数の値を、提供された値に設定します。

フィールド値の取得用の関数

関数	用途
getFieldValue (<i>fieldName</i> String <u>in</u>)	現在処理中の行について、指定のフィールド値を返します。戻り値は ANY 型です。

サブレポート用のデータの追加または取得用の関数

サブレポートは、別のレポート内から呼び出されるレポートです。場合によっては、メイン・レポートとサブレポートとの間でデータが交換されます。また、サブレポートは、別のサブレポートのメイン・レポートになることもできます。

関数	用途
addReportData (<i>rd</i> ReportData <u>in</u> , <i>dataSetName</i> String <u>in</u>);	指定された名前を持つレポート・データ・オブジェクトを、現在のレポート・ハンドラーに追加します。
getReportData (<i>dataSetName</i> String <u>in</u>)	指定された名前を持つレポート・データ・レコードを検索します。戻り値は ReportData 型です。

ここで説明した関数の使用例については、『EGL レポート・ハンドラーの手操作による作成』を参照してください。

関連する概念

226 ページの『EGL レポート作成プロセスの概要』

225 ページの『EGL レポートの概要』

関連するタスク

107 ページの『EGL 6.0 iFix への EGL コードのマイグレーション』

238 ページの『EGL レポート・ハンドラーの手操作による作成』

関連する参照項目

229 ページの『ライブラリー内のデータ・レコード』

230 ページの『EGL レポート・ハンドラー』

922 ページの『EGL ライブラリー ReportLib』

231 ページの『事前定義されたレポート・ハンドラー関数』

XML 設計文書内のデータ・タイプ

XML レポート設計文書では、データ・タイプは Java データ・タイプとして記述されます。設計文書で使用する EGL スクリプトレット・コードを作成する場合は、該当する EGL プリミティブ型に対応する Java データ・タイプを使用してください。EGL スクリプトレット・コードの呼び出しの結果戻されるデータは、Java データ・タイプを使用して宣言する必要があります。

次の表は、EGL プリミティブ型と Java データ・タイプとのマッピングのしかたを示しています。JavaReports 資料には、使用可能な Java データ・タイプについての情報が記載されています。

EGL プリミティブ型	Java データ・タイプ
bigint	java.lang.Long
bin	java.math.BigDecimal
blob	
char	java.lang.String
clob	
date	java.util.Date
dbchar	java.lang.String
decimal	java.math.BigDecimal
decimalfloat	java.lang.Double
float	java.lang.Float
hex	java.lang.Byte
int	java.lang.Integer
interval	java.lang.String
mbchar	java.lang.String
money	java.math.BigDecimal
numc	java.math.BigDecimal
pacf	java.math.BigDecimal
smallfloat	java.lang.Float
smallint	java.lang.Short

EGL プリミティブ型	Java データ・タイプ
string	java.lang.String
time	java.sql.Time
timestamp	java.sql.Timestamp
unicode	java.lang.String

関連する概念

- 225 ページの『EGL レポートの概要』
- 226 ページの『EGL レポート作成プロセスの概要』

関連するタスク

- 236 ページの『パッケージへの設計文書の追加』

関連する参照項目

- 229 ページの『ライブラリー内のデータ・レコード』
- 922 ページの『EGL ライブラリー ReportLib』
- 230 ページの『EGL レポート・ハンドラー』

EGL レポート・ドライバー関数用のサンプル・コード

このトピックには、次の 3 種類のデータ・ソースを使用したレポートの生成方法を示す、コードの断片が記載されています。

- データベース接続
- データ・レコード
- SQL ステートメント

次のコードの断片は、データ・ソースとしてデータベース接続を使用したレポートの生成を示しています。

```
//Variable declaration
myReport      Report;
myReportData  ReportData;

//Function containing report invocation code
function makeReport()
    //Initialize Report file locations
    myReport.reportDesignFile = "reportDesignFileName.jasper";
    myReport.reportDestinationFile =
        "reportDestinationFileName.jrprint";

    //Set the report data via a connection using the SQL statement
    //embedded in the report design
    sysLib.defineDatabaseAlias("alias", "databaseName");
    sysLib.connect("alias", "userid", "password");
    myReportData.connectionName="connectionName";
    myReport.reportData = myReportData;

    //Fill the report with data
    reportLib.fillReport(myReport, DataSource.databaseConnection);
    //Export the report in PDF format
    myReport.reportExportFile = "reportDesignFileName.pdf";
    reportLib.exportReport(myReport, ExportFormat.pdf);
end
```


次のコードの断片は、データ・ソースとしてレポート・データ・フレキシブル・レコードを使用したレポートの生成を示しています。

```
//Variable declaration
myReport      Report;
myReportData  ReportData;

//Function containing the report driving code
function makeReport()
    //Initialize myReport file locations
    myReport.reportDesignFile = "reportDesignFileName.jasper";
    myReport.reportDestinationFile =
        "reportDestinationFileName.jrprint";

    //Set the report data
    populateReportData();
    myReport.reportData = myReportData;

    //Fill the report with data
    reportLib.fillReport(myReport, DataSource.reportData);

    //Export the report in HTML format
    myReport.reportExportFile = "reportDesignFileName.html";
    reportLib.exportReport(myReport, ExportFormat.html);
end

function populateReportData()
    //Insert EGL code here which populates myReportData
    ...
end
```

次のコードの断片は、データ・ソースとして SQL ステートメントを使用したレポートの生成を示しています。

```
//Variable declaration
myReport      Report;
myReportData  ReportData;

//Function containing report driving code
function makeReport()
    //Initialize Report file locations
    myReport.reportDesignFile = "reportDesignFileName.jasper";
    myReport.reportDestinationFile = "reportDestinationFileName.jrprint";

    //Set the report data via a SQL statement
    myReportData.sqlStatement = "SELECT * FROM dataBaseTable";
    myReport.reportData = myReportData;

    //Fill the report with data
    reportLib.fillReport(myReport, DataSource.sqlStatement);

    //Export the report in text format
    myReport.reportExportFile = "reportOutputFileName.txt";
    reportLib.exportReport(myReport, ExportFormat.text);
end
```

関連する概念

226 ページの『EGL レポート作成プロセスの概要』

225 ページの『EGL レポートの概要』

関連するタスク

242 ページの『レポートをドライブするコードの作成』

関連する参照項目

229 ページの『ライブラリー内のデータ・レコード』
228 ページの『データ・ソース』
230 ページの『EGL レポート・ハンドラー』
922 ページの『EGL ライブラリー ReportLib』

パッケージへの設計文書の追加

レポートのレイアウトおよびその他の設計情報を指定する、XML 設計文書 (拡張子 .jrxml 付き) が必要です。

パッケージに設計文書を追加する手順は、次のとおりです。

1. 次の方法のどちらかで設計文書を作成する。
 - サード・パーティー製の JasperReports 設計ツール (JasperAssistant または iReports など) を使用する。作成するファイルに拡張子 .jrxml がない場合、拡張子 .jrxml を持つようにファイルの名前を変更してください。
 - テキスト・エディターを使用して、Jasper XML 設計情報を新しいテキスト・ファイルに書き込み、そのファイルを .jrxml ファイルとして保管する。
2. EGL レポート・ハンドラーおよびレポート・ドライバー・ファイルが入るのと同じ EGL パッケージに、XML 設計文書を置く。

新しい XML 設計文書を作成しない場合は、以前にコンパイルされた .jasper ファイルをインポートする必要があります。

「プロジェクト」>「すべてをビルド」を選択して、すべての EGL プロジェクト・コンポーネントをビルドすると、自動的に .jrxml ファイルが .jasper ファイルにコンパイルされます。

注: XML 設計文書とレポート・ハンドラーを同時に作成しようとする場合に従うべきガイドラインについては、『EGL レポート作成プロセスの概要 (EGL report creation process overview)』を参照してください。XML 設計文書がレポート・ハンドラーからレポート・データ・レコードを取得する方法を示す例については、『EGL レポート・ハンドラーの手操作による作成』を参照してください。

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーの手操作による作成

レポートをドライブするコードの作成

関連する参照項目

EGL レポート・ライブラリー

XML 設計文書内のデータ・タイプ

レポート・テンプレートの使用

次の EGL レポート・テンプレートのいずれかを選択し、変更することができます。

- データベース接続テンプレート
- レポート・データ・テンプレート
- SQL ステートメント・テンプレート
- レポート・ハンドラー・テンプレート

レポート・テンプレートを使用する手順は、次のとおりです。

1. 「ウィンドウ」>「設定」を選択する。
2. 設定のリストが表示されたならば、「EGL」を展開する。
3. 「エディター」を展開し、「テンプレート」を選択する。
4. テンプレートのリストをスクロールし、テンプレートを選択する。たとえば、レポート・ハンドラー・テンプレートを表示するには、**handler** を選択します。
5. 「編集」をクリックする。
6. ニーズに合わせてテンプレートを変更する。

レポート・ハンドラー・テンプレートを編集するには、**handler** と入力した後、**Ctrl+space** を押します。コード例を含めて、詳細については、『EGL レポート・ハンドラーの手操作による作成』を参照してください。

データ・ソース・テンプレートを編集するには、**jas** と入力した後、**Ctrl+space** を押します。

7. 「適用」をクリックした後、「OK」をクリックして、変更を保管する。

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL レポート・ハンドラーの手操作による作成

レポートをドライブするコードの作成

EGL でのコンテンツ・アシストの使用

テンプレートの設定の変更

関連する参照項目

EGL レポート・ハンドラー

EGL レポート・ライブラリー

EGL レポート・ドライバー関数のサンプル・コード

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーは、レポートへの埋め込み時に発生するイベントを処理するためのロジックを提供します。EGL レポート・ハンドラーを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。
2. ワークベンチで「ファイル」>「新規」>「その他」をクリックする。
3. 「新規」ウィンドウで、**EGL** を展開する。
4. 「レポート・ハンドラー (Report Handler)」をクリックする。
5. 「次へ」をクリックする。
6. EGL ファイルを入れるプロジェクトまたはフォルダーを選択し、パッケージを選択する。
7. レポート・ハンドラー名はファイル名と同じになるため、EGL パーツ名規則に準拠したファイル名を選択する。「EGL ソース・ファイル名」フィールドに、EGL ファイルの名前を入力する (例: myReportHandler)。
8. 「完了」をクリックする。

関連する概念

9 ページの『開発過程』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

572 ページの『生成される出力』

19 ページの『パーツ』

10 ページの『ランタイム構成』

関連するタスク

132 ページの『EGL Web プロジェクトの作成』

関連する参照項目

523 ページの『EGL エディター』

532 ページの『EGL ソース形式』

EGL レポート・ハンドラーの手操作による作成

「新規 EGL レポート・ハンドラー (New EGL Report Handler)」ウィザードを使用してレポート・ハンドラーを作成したくない場合は、手操作でレポート・ハンドラーを作成できます。

レポート・ハンドラーを手操作で作成する手順は、次のとおりです。

1. 新規 EGL ソース・ファイルを作成する。
2. 次のどちらかを実行する。
 - ・ 手操作でハンドラー・コードを入力する。
 - ・ 次のように、レポート・ハンドラー・テンプレートを使用してハンドラーを挿入する。
 - a. レポート・ハンドラー・テンプレートにナビゲートし、使用したいテンプレートを選択する。
 - b. 「編集」をクリックする。
 - c. **handler** を入力した後、**Ctrl+space** を押す。
 - d. 必要に応じてテンプレート・コードを変更する。

このトピックのこれ以降の部分では、次のものを示すコード例を記載しています。

- 手操作でレポート・ハンドラーを作成するための構文
- レポート・ハンドラーでレポート・パラメーターを取得する方法
- レポート変数を設定し、取得する方法
- フィールド値を取得する方法
- フレキシブル・レコードを追加する方法
- XML 設計文書がレポート・ハンドラーからレポート・データ・レコードを取得する方法

このコードをコピーし、ご使用のアプリケーションに合わせて変更することができます。

手操作でレポート・ハンドラーを作成するための構文を示すサンプル・コード

次のコードは、手操作で EGL レポート・ハンドラーを作成するための一般的な構文を示しています。

```
handler handlerName type jasperReport

// Use Declarations (optional)
use usePartReference;

// Constant Declarations (optional)
const constantName constantType = literal;

// Data Declarations (optional)
identifierName declarationType;

// Pre-defined Jasper callback functions (optional)
function beforeReportInit()
...
end

function afterReportInit()
...
end

function beforePageInit()
...
end

function afterPageInit()
...
end

function beforeColumnInit()
...
end

function afterColumnInit()
...
end

function beforeGroupInit(stringVariable string)
...
end

function afterGroupInit(stringVariable string)
...
end
```

```

function beforeDetailEval()
...
end

function afterDetailEval()
...
end

// User-defined functions (optional)
function myFirstFunction()
...
end

function mySecondFunction()
...
end
end

```

レポート・パラメーターの取得方法を示す例

次のコードの断片は、レポート・ハンドラーでレポート・パラメーターを取得する方法を示しています。

```

handler myReportHandler type jasperReport

// Data Declarations
report_title String;

// Jasper callback function
function beforeReportInit()
...

    report_title = getReportTitle();

...
end

...

// User-defined function
function getReportTitle() Returns (String)
    return (getReportParameter("ReportTitle"));
end

end

```

レポート変数を設定し、取得する方法を示す例

下に示すコードの断片は、レポート・ハンドラーでレポート変数を設定し、取得する方法を示しています。

```

handler myReportHandler type jasperReport

// Data Declarations
employee_serial_number int;

// Jasper callback function
function afterPageInit()
...
    employee_serial_number = getSerialNumberVar();
...
end

...

// User-defined function

```

```

function getSerialNumberVar() Returns (int)
    employeeName String;
    employeeName = "Ficus, Joe";
    setReportVariableValue("employeeNameVar", employeeName);
    return (getReportVariableValue("employeeSerialNumVar"));
end
end

```

レポート・ハンドラー内のレポート・フィールド値の取得方法を示す例

下に示すコードの断片は、レポート・ハンドラーでレポート・フィールド値を取得する方法を示しています。

```

handler myReportHandler type jasperReport

    // Data Declarations
    employee_first_name String;

    // Jasper callback function
    function beforeColumnInit()
        ...
        employee_first_name = getFirstNameField();
        ...
    end

    ...

    // User-defined function
    function getFirstNameField() Returns (String)
        fldName String;
        fldName = "fname";
        return (getFieldValue(fldName));
    end

end

```

レポート・ハンドラーでレポート・データ・フレキシブル・レコードを追加する方法を示す例

下に示すコード例は、レポート・ハンドラーでレポート・データ・フレキシブル・レコードを追加する方法を示しています。

```

handler myReportHandler type jasperReport

    // Data Declarations
    customer_array customerRecordType[];
    c customerRecordType;

    // Jasper callback function
    function beforeReportInit()
        customer ReportData;
        datasetName String;

        //create the ReportData object for the Customer subreport
        c.customer_num = getFieldValue("c_customer_num");
        c.fname = getFieldValue("c_fname");
        c.lname = getFieldValue("c_lname");
        c.company = getFieldValue("c_company");
        c.address1 = getFieldValue("c_address1");
        c.address2 = getFieldValue("c_address2");
        c.city = getFieldValue("c_city");
        c.state = getFieldValue("c_state");
        c.zipcode = getFieldValue("c_zipcode");
        c.phone = getFieldValue("c_phone");
        customer_array.appendElement(c);
    end
end

```



```

        customer.data = customer_array;
        datasetName = "customer";
        addReportData(customer, datasetName);
    end
end

```

XML 設計文書がレポート・ハンドラーからレポート・データ・レコードを取得する方法を示す例

下に示すコードの断片は、XML 設計文書がレポート・ハンドラーからレポート・データ・フレキシブル・レコードを取得する方法を示しています。

```

<jasperReport name="MasterReport"
  scriptletClass="subreports.SubReportHandler">
...

<subreport>
  <dataSourceExpression>
    <![CDATA[(JRDataSource)((subreports.SubReportHandler)
      ${REPORT_SCRIPTLET}).getReportData(
        new String("customer")))]>
  </dataSourceExpression>
  <subreportExpression class="java.lang.String">
    <![CDATA["C:/RAD/workspaces/Customer.jasper"]]>
  </subreportExpression>
</subreport>

...
</jasperReport>

```

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL ソース・ファイルの作成

EGL レポート・ハンドラーの作成

レポート・テンプレートの使用

関連する参照項目

EGL レポート・ライブラリー

EGL レポート・ハンドラー

事前定義されたレポート・ハンドラー関数

追加の EGL レポート・ハンドラー関数

レポートをドライブするコードの作成

「新規 EGL プログラム・パーツ」ウィザードを使用して、レポート・ライブラリーを使用してレポートを実行する新規 EGL 基本プログラムを作成します。

このレポート・ドライバーを作成する手順は、次のとおりです。

1. 「ファイル」>「新規」>「プログラム」を選択してから、EGL ファイルを入れるフォルダーを選択する。
2. パッケージを選択する。

3. ソース・ファイルのファイル名を指定し、*BasicProgram* タイプを選択し、「完了」をクリックする。
4. プログラム行を検出する。
5. `main()` 関数で、プログラム行の直後に、**jas** と入力し、続けて **Ctrl+space** を押して、レポート・ドライバーのコードを挿入する。
6. データ・ソース接続タイプとコードを含むウィンドウで、データ・ソース接続タイプの 1 つを選択する。
7. 既存のコードを変更するか、独自のコードを追加することができます。コードを変更する場合は、レポート・ドライバーが使用する変数に固有の値を挿入してください。これらの変数には、*reportDesignFileName*、*reportDestinationFileName*、*exportReportFile*、*alias*、*databaseName*、*userid*、*password*、および *connectionName* があります。

レポート呼び出し情報を示すコード

次のコードは、レポート呼び出し情報を示しています。

```
myReport      Report;
myReportData  ReportData;

myReport.reportDesignFile = "myReport_XML.jasper";
myReport.reportDestinationFile = "myReport.jrprint";
myReport.reportExportFile = "myReport.pdf";

myReportData.sqlStatement = "Select * From myTable";

myReport.reportData = myReportData;

ReportLib.fillReport(myReport, DataSource.sqlStatement);

ReportLib.exportReport(myReport, ExportFormat.pdf);
```

コード	説明
<code>myReport Report;</code>	これは、レポート・ライブラリー・レコード宣言です。
<code>myReportData ReportData;</code>	これは、レポート・ライブラリー・データ・レコード宣言です。
<code>myReport.reportDesignFile = "myReport_XML.jasper";</code>	この文は、レポートの作成に使用するレポート設計を定義します。
<code>myReport.reportDestinationFile = "myReport.jrprint";</code>	この文は、生成されるレポート出力のファイル名を指定します。
<code>myReport.reportExportFile = "myReport.pdf"</code>	この文は、エクスポートされる出力のファイル名を指定します。
<code>myReport.sqlStatement = "Select * From myTable";</code>	これは、レポートで使用される SQL SELECT ステートメントについての情報を提供します。
<code>myReport.reportData = myReportData;</code>	これは、レポート・データについての情報を提供します。
<code>ReportLib.fillReport(myReport, DataSource.sqlStatement);</code>	この文は、レポートのソース情報を指定します。
<code>ReportLib.exportReport(myReport, ExportFormat.pdf);</code>	この文は、レポート出力フォーマットを指定します。

コード・フラグメント例:

```
//location where the .jprint file is stored.  
abcReport.reportDestinationFile="C:¥¥temp¥¥MasterReport.jrprint";  
  
//location for the exported report.  
abcReport.reportExportFile="C:¥¥temp¥¥MasterReport.pdf";  
  
//perform the export.  
ReportLib.exportReport(abcReport, ExportFormat.pdf);
```

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーの手操作による作成

レポート・テンプレートの使用

関連する参照項目

EGL レポート・ハンドラー

EGL レポート・ライブラリー

EGL レポート・ドライバー関数のサンプル・コード

レポート用のファイルの生成およびレポートの実行

レポートのレイアウトおよびその他の設計情報を指定する、XML 設計文書 (拡張子 .jrxml 付き) が必要です。

EGL プロジェクト用のレポートをビルドし、生成する手順は、次のとおりです。

1. 「プロジェクト」>「すべてをビルド」を選択して EGL プロジェクトをビルドします。

EGL は、自動的に EGL レポート・ハンドラーから Java コードを生成し、XML 設計文書 (.jrxml ファイル) を .jasper ファイルにコンパイルします。

2. レポート呼び出しコードを持つ EGL プログラムを実行する。パッケージ・エクスプローラーでこれを行う一つの方法は、コードを含む .egl ファイルまでナビゲートし、右クリックすることです。その後、ポップアップ・メニューから「生成」を選択します。

レポートの生成に加えて、EGL が使用する JasperReports プログラムは、レポート呼び出しコード内の *reportDestinationFileName* によって指定されるロケーションに、生成されたレポートを自動的に保管します。

レポートを生成する JasperReports プログラムは、.jprint ファイルも生成し、保管します。このファイルは、最終的なレポート形式 (.pdf、.html、.xml、.txt、または .csv) にエクスポートされる中間ファイル形式です。プログラムは、複数のエクスポートに 1 つの .jprint ファイルを再使用することができます。

レポート呼び出しコード内の *exportReport()* 関数により、EGL は指定のフォーマットでレポートをエクスポートします。

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーの手操作による作成

レポートをドライブするコードの作成

レポートのエクスポート

関連する参照項目

EGL レポート・ライブラリー

EGL レポート・ハンドラー

レポートのエクスポート

記入されたレポートを PDF、HTML、XML、CSV (comma-separated values)、およびプレーン・テキスト出力としてエクスポートすることができます。EGL レポート・ドライバー・コード内の *exportReport()* 関数により、EGL は指定の形式でレポートをエクスポートします。

レポート・ドライバー・コード内の *exportReportFile* 値は、エクスポートされたファイルのロケーションを指定します。

エクスポートされたレポートの形式を指定するには、*exportReport()* 関数への呼び出しで次のパラメーターの 1 つを使用してください。

- *ExportFormat.html*
- *ExportFormat.pdf*
- *ExportFormat.text*
- *ExportFormat.xml*
- *ExportFormat.csv*

たとえば、次のコードでは、EGL はレポートを *.pdf* 形式でエクスポートします。

```
reportLib.exportReport(myReport, ExportFormat.pdf);
```

重要: EGL は、エクスポートされたレポートを自動的に更新することはありません。レポートの設計を変更する場合、またはデータが変更された場合は、レポートを再記入し、再度エクスポートする必要があります。

関連する概念

EGL レポートの概要

EGL レポート作成プロセスの概要

関連するタスク

EGL レポート・ハンドラーの作成

EGL レポート・ハンドラーの手操作による作成

レポートをドライブするコードの作成
レポートの実行

関連する参照項目

EGL レポート・ライブラリー

EGL レポート・ハンドラー

ファイルとデータベースによる作業

SQL サポート

次の表のように、EGL 生成コードは、いずれかのターゲット・システム上にあるリレーショナル・データベースにアクセスできます。

ターゲット・システム	リレーショナル・データベースへのアクセスのサポート
AIX、iSeries、Linux、Windows 2000/NT/XP、UNIX システム・サービス	JDBC を使用して DB2 UDB、Oracle、または Informix にアクセスできる

プログラムの作成時には、他のほとんどの言語でプログラムをコーディングしている場合と同じように、SQL ステートメントをコーディングすることができます。SQL ステートメントを簡単に作成できるよう、EGL は埋めるだけでよい SQL ステートメントのテンプレートがあります。

これ以外にも、EGL ステートメントをコーディングする場合には、SQL レコードを入出力オブジェクトとして使用することができます。このようにレコードを使用すると、提供されている SQL ステートメントをカスタマイズしたり、SQL をコーディングしなくても済むデフォルトをそのまま使用して、データベースにアクセスできます。

いずれの場合も、EGL の SQL サポートについて以下の点に注意してください。

- 特定のテーブル列に NULL があるかどうかをテストする場合は、列値を SQL レコードに入れて、つまり NULL 可能と宣言されているレコード項目に列値を入れて受け取る必要があります。詳細については、後述する『NULL の有無のテストおよびその設定』を参照してください。
- 以下の概説のセクションでは (SQL 用語に従って)、SQL ステートメントで参照されている各項目をホスト変数 と呼びます。ホスト という語は、SQL ステートメントが組み込まれている言語 (この場合は EGL プロシーチャー型言語) を指します。SQL ステートメント内のホスト変数は、次の例のように、コロンの後に記述されます。

```
select empnum, empname
from employee
where empnum >= :myRecord.empnum
for update of empname
```

EGL ステートメントと SQL

リレーショナル・データベースへのアクセスに利用できる EGL キーワードは、次の表のとおりです。この表には、各キーワードに対応する SQL ステートメントの概要も含まれています。例えば、EGL の **add** ステートメントをコーディングする場合は、SQL INSERT ステートメントを生成します。

多くのビジネス・アプリケーションでは、EGL の **open** ステートメントと各種の **get by position** ステートメントを使用します。このコードは、カーソル (これは、以下の振る舞いをする実行ランタイム・エンティティです) を宣言し、オープンし、処理するのに役立ちます。

- 検索基準を満たす行のリストである結果セット を戻す。
- 結果セット内の特定の行を指す。

EGL の **open** ステートメントを使用して、ストアード・プロシージャを呼び出すことができます。このプロシージャは、EGL の外部で作成されたロジックで構成され、データベース管理システムに保管され、結果セットも戻します。(また、EGL の **execute** ステートメントを使用してもストアード・プロシージャを戻すことができます。)

結果セットの処理の詳細については、後のセクションで説明します。

SQL ステートメントを明示的にコーディングする場合は、EGL の **execute** ステートメントと、 場合によっては EGL の **prepare** ステートメントを使用します。

キーワードおよび目的	SQL ステートメントの概要	SQL を変更できるかどうか
add データベース内に 1 行挿入する。 または、(SQL レコードの動的配列を使用している場合は) 配列の連続するエレメントの内容に基づき、行のセットを挿入する。	行の INSERT (挿入) (動的配列を指定している場合は、繰り返し発生します)。	可能
close 未処理の行を解放する。	カーソルの CLOSE (クローズ)	不可
delete 行をデータベースから削除する。	行の DELETE (削除)。行は、次の 2 つの方法で選択されました。 <ul style="list-style-type: none"> • forUpdate オプションを指定して get ステートメントを呼び出す場合 (同一のキー値を持つ複数の行の最初の行を選択する場合に該当) • forUpdate オプションを指定して open ステートメントを呼び出し、続いて get next ステートメントを呼び出す場合 (行のセットを選択し、取り出したデータをループ内で処理する場合に該当) 	不可

キーワードおよび目的	SQL ステートメントの概要	SQL を変更できるかどうか
forEach ループで実行される一連の文の始まりを示します。最初の反復が発生するのは、指定された結果セットが使用可能であり、(多くの場合) その結果セット内の最後の行が処理されるまでの間、継続している場合だけです。	EGL は forEach ステートメントを、ループ内で実行される SQL FETCH ステートメントに変換します。	不可
freeSQL 動的に準備された SQL ステートメントに関連したリソースを解放し、その SQL ステートメントに関連したすべてのオープン・カーソルをクローズする。		不可
get (キー値による取得とも呼ばれる) データベースから単一の行を読み取る。または、(SQL レコードの動的配列を使用する場合) 連続する行を配列内の連続するエレメントに読み込む。	行の SELECT (選択)。ただし、singleRow オプションを設定した場合のみ。それ以外の場合は、以下の規則が適用されます。 <ul style="list-style-type: none"> EGL では、get ステートメントが以下のものに変換されます。 <ul style="list-style-type: none"> SELECT または (forUpdate オプションを設定している場合は) SELECT FOR UPDATE でカーソルを DECLARE (宣言) カーソルの OPEN (オープン) 行の FETCH (取り出し) forUpdate オプションを指定しなかった場合は、EGL はカーソルもクローズします。 singleRow および forUpdate オプションは、動的配列ではサポートされていません。その場合、EGL ランタイムはカーソルを宣言してオープンし、連続した行を取り出した後、カーソルをクローズします。 	可能
get absolute open ステートメントによって選択された結果セット内の、番号で指定された行を読み取る。	EGL では、 get absolute ステートメントが SQL FETCH ステートメントに変換されます。	不可

キーワードおよび目的	SQL ステートメントの概要	SQL を変更できるかどうか
<p>get current</p> <p>open ステートメントによって選択された結果セット内の、現在すでにカーソルが位置付けられている行を読み取る。</p>	EGL では、 get current ステートメントが SQL FETCH ステートメントに変換されます。	不可
<p>get first</p> <p>open ステートメントによって選択された結果セット内の最初の行を読み取る。</p>	EGL では、 get first ステートメントが SQL FETCH ステートメントに変換されます。	不可
<p>get last</p> <p>open ステートメントによって選択された結果セット内の最後の行を読み取る。</p>	EGL では、 get last ステートメントが SQL FETCH ステートメントに変換されます。	不可
<p>get next</p> <p>open ステートメントによって選択された結果セット内の次の行を読み取る。</p>	EGL では、 get next ステートメントが SQL FETCH ステートメントに変換されます。	不可
<p>get previous</p> <p>open ステートメントによって選択された結果セット内の前の行を読み取る。</p>	EGL では、 get previous ステートメントが SQL FETCH ステートメントに変換されます。	不可
<p>get relative</p> <p>open ステートメントによって選択された結果セット内の、番号で指定された行を読み取る。この行は、結果セット内のカーソル位置との相対関係で識別されます。</p>	EGL では、 get relative ステートメントが SQL FETCH ステートメントに変換されます。	不可

キーワードおよび目的	SQL ステートメントの概要	SQL を変更できるかどうか
<p><code>execute</code></p> <p>(例えば、CREATE TABLE 型の) SQL データ定義ステートメントを実行する。または、(例えば、INSERT または UPDATE の) データ操作ステートメントを実行する。または、SELECT 文節から始まらない準備済み SQL ステートメントを実行する。</p>	<p>作成した SQL ステートメントは、データベース管理システムで使用できます。</p> <p>execute の主な用途は、次の例に示しているように、生成時に完全にフォーマット設定された単一 SQL ステートメントのコーディングです。</p> <pre>try execute #sql{ // no space after "#sql" delete from EMPLOYEE where department = :myRecord.department }; onException myErrorHandler(10); end</pre> <p>完全にフォーマット設定された SQL ステートメントの WHERE 文節には、ホスト変数を組み込むことができます。</p>	可能
<p><code>open</code></p> <p>後で get next ステートメントを使用して行のセットを取り出すために、リレーショナル・データベースからその行セットを選択する。</p>	<p>EGL は、open ステートメントを <code>call</code> ステートメントに変換する (ストアド・プロシージャにアクセスするため)、または以下のステートメントに変換する。</p> <ul style="list-style-type: none"> • SELECT または SELECT FOR UPDATE を使用した、カーソルの DECLARE (宣言) • カーソルの OPEN (オープン) 	可能

キーワードおよび目的	SQL ステートメントの概要	SQL を変更できるかどうか
<p>prepare</p> <p>SQL PREPARE ステートメントを指定する。このステートメントは、実行時にしかわからない詳細がオプションに含まれています。EGL の execute ステートメントを実行するか、(SQL ステートメントが SELECT で始まっている場合は) EGL の open または get ステートメントを実行して、準備済み SQL ステートメントを実行します。</p>	<p>EGL は prepare ステートメントを SQL の PREPARE ステートメントに変換します。このステートメントは、常に行実行時に構成されます。以下に示した EGL の prepare ステートメントの例では、各パラメーター・マーカー (?) は、その後の execute ステートメントの USING 文節によって解決されます。</p> <pre>myString = "insert into myTable " + "(empnum, empname) " + "value ?, ?"; try prepare myStatement from myString; onException // exit the program myErrorHandler(12); end try execute myStatement using :myRecord.empnum, :myRecord.empname; onException myErrorHandler(15); end</pre>	可能
<p>replace</p> <p>変更された行をデータベースに戻す。</p>	<p>行の UPDATE (更新)。行は、次の 2 つの方法で選択されました。</p> <ul style="list-style-type: none"> • forUpdate オプションを指定して get ステートメントを呼び出す場合 (同一のキー値を持つ複数の行の最初の行を選択する場合に該当)。または、 • forUpdate オプションを指定して open ステートメントを呼び出し、続いて get next ステートメントを呼び出す場合 (行のセットを選択し、取り出したデータをループ内で処理する場合に該当)。 	可能

注: いかなる場合でも、単一の EGL ステートメントをコーディングして、複数のデータベース表を更新することはできません。

結果セットの処理

一連の行を更新する一般的な方法は、以下のとおりです。

1. **forUpdate** オプションを指定して EGL の **open** ステートメントを実行し、カーソルを宣言してからオープンする。このオプションを指定することにより、選択された行が、それ以降の更新または削除に対してロックされます。
2. EGL **get next** ステートメントを実行して、1 つの行を取り出す。
3. ループの中で以下の操作を行う。
 - a. 取り出したデータの格納先のホスト変数内のデータを変更する。
 - b. EGL **replace** ステートメントを実行して、行を更新する。

- c. EGL **get next** ステートメントを実行して、別の行を取り出す。
4. EGL 関数 **commit** を実行して、変更をコミットする。

カーソルをオープンし、そのカーソルの行に対して実行される文は、結果セット ID によって互いに関連付けられています。この結果セット ID は、すべての結果セット ID、プログラム変数、およびプログラム内のプログラム・パラメーターに渡って一意である必要があります。この ID は、カーソルをオープンする **open** ステートメントで指定し、個々の行に影響を与える **get next**、**delete**、**replace** ステートメント、およびカーソルをクローズする **close** ステートメントで参照します。追加情報については、『*resultSetID*』を参照してください。

SQL をコーディングしている場合に、連続した行を更新する方法を次のコードに示します。

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate with
  #sql{
    select empname
    from EMPLOYEE
    where empnum >= :myRecord.empnum
    for update of empname
  };

onException
  myErrorHandler(8); // exits program
end

try
  get next from selectEmp into :myRecord.empname;
onException
  if (sysVar.sqlcode != 100)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode != 100)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    execute
    #sql{
      update EMPLOYEE
      set empname = :empname
      where current of selectEmp
    };
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next from selectEmp into :myRecord.empname;
  onException
    if (sysVar.sqlcode != 100)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;
```

上記の例で、あまり複雑なことをしたくない場合には、SQL レコードを考慮してみてください。SQL レコードを使用すると、コードを簡素化することができ、データベース管理システム全体で変わることのない入出力エラー値を使用することができます。次の例は上記の例と等価ですが、emp という SQL レコードを使用しています。

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(8);    // exits program
end

try
  get next emp;
onException
  if (sysVar.sqlcode not noRecordFound)
    myErrorHandler(8); // exit the program
  end
end

while (sysVar.sqlcode not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10); // exits program
  end

  try
    get next emp;
  on exception
    if (sysVar.sqlcode not noRecordFound)
      myErrorHandler(8); // exits program
    end
  end
end // end while; cursor is closed automatically
// when the last row in the result set is read

sysLib.commit;
```

SQL レコードについては、後述のセクションで説明します。

SQL レコードおよびその値

SQL レコードとは、SQL レコード・パーツを元にした変数です。このタイプのレコードを使用すると、あたかもファイルにアクセスしているかのように、リレーショナル・データベースと対話することができます。例えば、変数 EMP がデータベース表 EMPLOYEE を参照している SQL レコード・パーツに基づいている場合、EGL の **add** ステートメントで EMP を次のように使用することができます。

```
add EMP;
```

この場合、EGL は EMP のデータを EMPLOYEE に挿入します。SQL レコードにも状態情報が含まれるので、EGL ステートメントの実行後に その SQL レコードをテストして、データベース・アクセスが原因で発生した入出力エラー値に従って、タスクを条件付きで実行することができます。

```

VGVar.handleHardIOErrors = 1;

try
  add EMP;
onException
  if (EMP is unique)      // if a table row
                        // had the same key
    myErrorHandler(8);
  end
end

```

EMP のような SQL レコードを使用すると、以下のようにリレーショナル・データベースと対話することができます。

- SQL レコード・パーツおよび関連する SQL レコードを宣言する
- それぞれの EGL ステートメントが、SQL レコードを入出力オブジェクトとして使用する EGL ステートメントセットを定義する
- EGL ステートメントのデフォルトの振る舞いを受け入れるか、またはビジネス・ロジックに適切な SQL 変更を行う。

SQL レコード・パーツおよび関連するレコードの宣言

SQL レコード・パーツを宣言し、各レコード項目をリレーショナル・テーブルまたはビューの 1 つの列に関連付けます。EGL エディターの取り出し機能を使用すると、この関連づけを自動的に EGL に実行させることができます。これについては、『宣言時のデータベース・アクセス』で説明します。

SQL レコード・パーツが固定レコード・パーツでない場合は、プリミティブ・フィールドをその他の変数と同様に組み込むことができます。特に、次の種類の変数を組み込むことがよくあります。

- 他の SQL レコード。それぞれの存在が、親テーブルと子テーブルの間にある 1 つの 1 対 1 の関係を表しています。
- SQL レコードの配列。それぞれの存在が、親テーブルと子テーブルの間にある 1 つの 1 対多の関係を表しています。

プリミティブ型のフィールドのみが、データベースの列を表すことができます。

フィールドの前にレベル番号がある場合、SQL レコード・パーツは固定レコード・パーツです。以下の規則が適用されます。

- 各 SQL レコード・パーツ内の構造体は、フラット である (階層構造でない) ことが必要です。
- すべてのフィールドがプリミティブ・フィールドであることが必要ですが、BLOB、CLOB、または STRING 型であってはなりません。
- レコード・フィールドを構造体フィールド配列にすることはできません。

SQL レコード・パーツの宣言後に、そのパーツに基づく SQL レコードを宣言します。

SQL 関連の EGL ステートメントの定義

EGL ステートメントのセットを定義して、そのセットのそれぞれの EGL ステートメントが SQL レコードを、そのステートメントの入出力オブジェクトとして使用することができます。EGL は、ステートメントごとに、暗黙の SQL ステートメン

トを提供します。このステートメントはソース内にはありませんが、SQL レコードと EGL ステートメントの組み合わせによって、暗黙で存在しています。例えば、EGL の **add** ステートメントの場合、暗黙の SQL INSERT ステートメントによって、ある与えられたレコード項目の値が、そのレコード項目に関連付けられているテーブル列に挿入されます。テーブル列が割り当てられていないレコード項目が SQL レコードに含まれている場合、EGL は、レコード項目の名前は列の名前と同一であると想定して、暗黙の SQL ステートメントを生成します。

暗黙の SELECT ステートメントの使用: SQL レコードを使用し、かつ SQL SELECT ステートメントまたはカーソル宣言を生成する EGL ステートメントを定義すると、EGL は 暗黙の SQL SELECT ステートメントを生成します。(この文は、カーソル宣言があればそれに組み込まれます。)例えば、EMP という名前で、以下のレコード・パーツに基づいている変数を宣言したとします。

```
Record Employee type sqlRecord
{ tableNames = [{"EMPLOYEE"}],
  keyItems = ["empnum"] }
empnum decimal(6,0);
empname char(40);
end
```

さらに、次の **get** ステートメントをコーディングしたとします。

```
get EMP;
```

暗黙の SQL SELECT ステートメントは、次のようになります。

```
SELECT empnum, empname
FROM   EMPLOYEE
WHERE  empnum = :empnum
```

EGL はまた、スタンドアロンの SELECT ステートメントか (カーソル宣言が含まれていない場合)、カーソルに関連付けられている FETCH ステートメントに、INTO 文節を挿入します。INTO 文節は、SELECT ステートメントの最初の文節にリストされている列から値を受け取るホスト変数をリストします。

```
INTO :empnum, :empname
```

暗黙の SELECT ステートメントは、各列値を対応するホスト変数に読み込み、SQL レコードで指定されているテーブルを参照します。また、2 つの因子の組み合わせに依存する検索基準 (WHERE 文節) を持っています。

- レコード・プロパティ **defaultSelectCondition** に対して指定した値
- 2 つの値セットの間の関係 (等価関係など)
 - テーブル・キーを構成する列の名前
 - レコード・キーを構成するホスト変数の値

データを SQL レコードの動的配列に読み込むと、**get** ステートメントの場合にそうなることがあるように、特別な状態になります。

- カーソルがオープンしており、データベースの連続する行が、配列の連続するエレメントに読み込まれ、結果セットが解放され、カーソルがクローズされます。
- SQL ステートメントを指定しない場合は、検索基準はレコード・プロパティ **defaultSelectCondition** によって異なり、以下の値セットの間の関係 (特に、以上、という関係) にも依存します。
 - EGL ステートメントで項目を指定するときに間接的に指定された列の名前

– それら項目の値

defaultSelectCondition プロパティで指定されたホスト変数はすべて、動的配列の基礎となる SQL レコードの外側にある必要があります。

暗黙の SELECT ステートメントについての詳細 (キーワードごとに変わります) は、*get* および *open* ステートメントを参照してください。

カーソルを使用した SQL レコードの使用: SQL レコードを使用している場合は、結果セット ID を使用する場合と同様に、いくつかの EGL ステートメントで同一の SQL レコードを使用して、カーソル処理ステートメント間を関連付けることができます。ただし、結果セット ID によって示されるステートメント相互関係が、SQL レコードで示される関係よりも優先し、場合によっては、*resultSetID* を指定する必要があります。

また、特定の SQL レコードに対してオープンにしておくことのできるカーソルは 1 つだけです。別のカーソルが 同じ SQL レコードをオープンしているときに、EGL ステートメントによってあるカーソルがオープンしてしまうと、生成されたコードは自動的に最初のカーソルをクローズしてしまいます。

SQL ステートメントのカスタマイズ

SQL レコードを入出力オブジェクトとして使用する EGL ステートメントが指定された場合、次のいずれかの方法を使用することができます。

- 暗黙の SQL ステートメントを受け入れることができます。この場合は、SQL レコード・パーツに加えられた変更によって、実行時に使用される SQL ステートメントが影響を受けます。例えば、SQL レコードのキーとして別のレコード項目を使用するということを後で示すと、EGL は、その SQL レコード・パーツに基づいているカーソル宣言で使用される暗黙の SELECT ステートメントを変更します。
- その代わりに、SQL ステートメントを明示的にすることができます。この場合は、SQL ステートメントの詳細は、SQL レコード・パーツから切り離され、後からその SQL レコード・パーツに加えられた変更は、実行時に使用される SQL ステートメントには影響を与えません。

明示的な SQL ステートメントをソースから除去すると、暗黙の SQL ステートメントがある場合は、生成時にそれがもう一度使用可能になります。

レコード内でのレコードの使用例

プログラムで、ある部門の一連の従業員のデータを取り出せるようにするには、次のように 2 つのレコード・パーツと 1 つの関数を作成します。

```
DataItem DeptNo { column = deptNo } end
```

```
Record Dept type SQLRecord  
  deptNo DeptNo;  
  managerID CHAR(6);  
  employees Employee[];  
end
```

```
Record Employee type SQLRecord  
  employeeID CHAR(6);  
  empDeptNo DeptNo;  
end
```

```
Function getDeptEmployees(dept Dept)
  get dept.employees usingKeys dept.deptNo;
end
```

NULL の有無のテストおよびその設定

EGL は、コード内の変数のサブセット用に、内部に NULL 標識を保守する場合があります。デフォルトの振る舞いを受け入れた場合、EGL は以下の特性を持つ変数用にのみ、内部で 1 つの変数につき 1 つの NULL 標識を保守します。

- SQL レコード内に存在している
- プロパティ **isNullable** を *yes* に設定して宣言されている

SQL ステートメントでは、NULL 標識にホスト変数をコーディングしないでください (いくつかの言語ではコーディングすることがあります)。NULL 可能ホスト変数に NULL が含まれているかどうかをテストするには、EGL の **if** ステートメントを使用します。切り捨てられた値を検索できるかどうかをテストすることもできますが、これは NULL 標識が使用可能になっている場合だけです。

次の 2 つの方法のどちらかで、SQL テーブル列を NULL にすることができます。

- EGL の **set** ステートメントを使用して、NULL 可能ホスト変数を NULL にし、関連する SQL レコードをデータベースに書き込む。または、
- SQL ステートメントを最初から作成するか、EGL の **add** ステートメントまたは **replace** ステートメントに関連付けられている SQL ステートメントをカスタマイズして、適切な SQL 構文を使用する。

NULL の処理の詳細については、『*itemsNullable*』および『SQL 項目のプロパティ』を参照してください。

宣言時のデータベース・アクセス

コードが実行時にアクセスするデータベースと同じ特性を持つデータベースに (宣言時に) アクセスすることに関しては、以下のような利点があります。

- SQL レコードに関連付けられているテーブルまたはビューと等価なデータベース表またはビューにアクセスする場合に、EGL パーツ・エディターの取り出し機能を利用して、レコード項目を作成または上書きすることができます。取り出し機能は、データベース管理システムに保管されている情報にアクセスして、作成される項目の数およびデータ特性が表の列の数やデータ特性を反映することができますようにします。Retrieve 機能を起動した後、レコード項目の名前変更、レコード項目の削除、およびその他の変更を SQL レコードに対して行うことができます。
- 適切に構造化されたデータベースに宣言時にアクセスすることは、等価のデータベースに実行時にアクセスする SQL ステートメントが有効になることを保証する場合に有効です。

検索機能は、関連するテーブル列と各項目の名前が同じ (またはほとんど同じ) レコード項目を作成します。

DB2 の条件 **WITH CHECK OPTIONS** を使って定義されているビューは、取り出すことができない。

取り出し機能についての詳細は、『SQL テーブル・データの取り出し』を参照してください。命名の詳細については、『SQL 検索設定の変更』を参照してください。

宣言時にデータベースにアクセスするには、『SQL データベース接続設定の変更』を参照して、設定ページで接続情報を指定してください。

関連する概念

『動的 SQL』

334 ページの『作業論理単位』

799 ページの『resultSetID』

関連するタスク

272 ページの『SQL テーブル・データの検索』

126 ページの『SQL データベース接続設定の変更』

128 ページの『SQL 検索設定の変更』

関連する参照項目

605 ページの『add』

613 ページの『close』

506 ページの『データベースの権限とテーブル名』

270 ページの『デフォルト・データベース』

617 ページの『delete』

619 ページの『execute』

631 ページの『get』

644 ページの『get next』

271 ページの『Informix および EGL』

430 ページの『itemsNullable』

664 ページの『open』

679 ページの『prepare』

681 ページの『replace』

801 ページの『SQL データ・コードおよび EGL ホスト変数』

260 ページの『SQL 例』

71 ページの『SQL 項目のプロパティ』

803 ページの『SQL レコードの内部レイアウト』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

258 ページの『NULL の有無のテストおよびその設定』

動的 SQL

EGL ステートメントに関連する SQL ステートメントは、生成時に静的に指定して、すべての詳細を設定することができます。しかし動的 SQL を使用する場合、SQL ステートメントは、実行時に EGL ステートメントが呼び出されるたびに構築されます。

動的 SQL を使用するとランタイム処理の速度が遅くなりますが、以下のように、ランタイム値に応じてデータベース操作を変化させることができます。

- データベース照会では、選択基準、データの集約方法、または行が戻される順序を変更できます。これらの詳細は、WHERE、HAVING、GROUP BY、および ORDER BY 文節によって制御されます。この場合、prepare ステートメントを使用できます。

- 多くの種類の操作で、ランタイム値を使用して、アクセスするテーブルを判別できます。次の 2 つの方法のいずれかで、テーブルの動的仕様が完了します。
 - prepare ステートメントを使用
 - 『EGL ソース形式における SQL レコード・パーツ』で説明しているように、SQL レコードを使用して **tableNameVariables** プロパティの値を指定

関連する概念

247 ページの『SQL サポート』

関連する参照項目

506 ページの『データベースの権限とテーブル名』

679 ページの『prepare』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

SQL 例

次のいずれかの方法で SQL データベースにアクセスできます。

- 生成時に既知の形式を持つ SQL ステートメントを手動でコーディングする。
- SQL レコードを EGL ステートメントの入出力オブジェクトとして使用する (SQL ステートメントの形式が生成時に既知である場合)。
 - EGL ソース内に明示的な SQL ステートメントを配置すると、その SQL ステートメントが実行時に使用される。
 - それ以外の場合、暗黙的な SQL ステートメントが実行時に使用される。
- EGL **prepare** ステートメントをコーディングする。これにより、SQL PREPARE ステートメントが生成され、そのステートメントによって実行時に SQL ステートメントが作成される。

どの事例でも、メモリー領域として SQL レコードを使用して、操作が正常に終了したかどうかを簡単にテストすることができます。この節の例では、レコード・パーツが EGL ファイルに宣言されていること、および、そのパーツに基づいたレコードがそのファイルでプログラムに宣言されていることを前提としています。

- SQL レコード・パーツを次に示します。

```
Record Employee type sqlRecord
{
    tableName = ["employee"],
    keyItems = ["empnum"],
    defaultSelectCondition =
        #sqlCondition{ // #sql と中括弧の間にスペースなし
            aTableColumn = 4 -- 各 SQL コメントは
                                -- 二重ハイフンで開始する
        }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end
```

- SQL レコードを次に示します。

```
emp Employee;
```

SQL レコードおよび暗黙的なステートメントの詳細については、『SQL サポート』を参照してください。

SQL ステートメントのコーディング

SQL ステートメントをコーディングする準備として、変数を宣言します。

```
empnum decimal(6,0);
empname char(40);
```

SQL テーブルへの行の追加: 行を追加する準備として、変数に値を割り当てます。

```
empnum = 1;
empname = "John";
```

行を追加するには、次のように、EGL **execute** ステートメントを SQL INSERT ステートメントに関連付けます。

```
try
execute
#sql{
insert into employee (empnum, empname)
values (:empnum, :empname)
};
onException
myErrorHandler(8);
end
```

SQL テーブルからの行セットの読み取り: SQL テーブルから行セットを読み取る準備として、レコード・キーを識別します。

```
empnum = 1;
```

データを取得するには、一連の EGL ステートメントをコーディングします。

- 結果セットを選択するには、EGL **open** ステートメントを実行します。

```
open selectEmp
with #sql{
select empnum, empname
from employee
where empnum >= :empnum
for update of empname
}
into empnum, empname;
```

- 結果セットの次の行にアクセスするには、EGL の **get next** ステートメントを実行します。

```
get next from selectEmp;
```

open ステートメントに **into** 文節を指定しなかった場合は、**get next** ステートメントに **into** 文節を指定する必要があります。両方の文に **into** 文節を指定した場合は、**get next** ステートメントの **into** 文節が優先されます。

```
get next from selectEmp
into empnum, empname;
```

結果セットから最後のレコードが読み取られると、カーソルが自動的にクローズされます。

もっと複雑な例は、次のコードのとおりです。このコードは行セットを更新します。

```
VGVar.handleHardIOErrors = 1;

try
```

```

open selectEmp
  with #sql{
    select empnum, empname
    from employee
    where empnum >= :empnum
    for update of empname
  }
  into empnum, empname;
onException
  myErrorHandler(6); // プログラムを終了する
end

try
  get next from selectEmp;
onException
  if (sqlcode != 100)
    myErrorHandler(8); // プログラムを終了する
  end
end

while (sqlcode != 100)
  empname = empname + " " + "III";

  try
    execute
      #sql{
        update employee
        set empname = :empname
        where current of selectEmp
      };
  onException
    myErrorHandler(10); // プログラムを終了する
  end

  try
    get next from selectEmp;
  onException
    if (sqlcode != 100)
      myErrorHandler(8); // プログラムを終了する
    end
  end
end // end while; 結果セットの最後の行が読み取られると
// カーソルは自動的にクローズされる

sysLib.commit();

```

get next および while ステートメントをコーディングする代わりに、次のように結果セット内のそれぞれの行について文ブロックを実行する foreach ステートメントを使用できます。

```

VGVar.handleHardIOErrors = 1;

try
  open selectEmp
    with #sql{
      select empnum, empname
      from employee
      where empnum >= :empnum
      for update of empname
    }
    into empnum, empname;
  onException
    myErrorHandler(6); // プログラムを終了する
  end

  try

```



```

forEach (from selectEmp)
  empname = empname + " " + "III";

  try
    execute
      #sql{
        update employee
        set empname = :empname
        where current of selectEmp
      };
    onException
      myErrorHandler(10); // プログラムを終了する
    end
  end // end forEach 結果セットの最後の行が読み取られると
    // カーソルは自動的にクローズされる

onException
  // 条件が "sqlcode = 100" の場合、forEach に関連した例外ブロックは
  // 実行されない
  // "if (sqlcode != 100)" のテストは避ける。
  myErrorHandler(8); // プログラムを終了する
end

sysLib.commit();

```

暗黙的な SQL ステートメントでの SQL レコードの使用

EGL SQL レコードの使用を開始するには、SQL レコード・パーツを宣言します。

```

Record Employee type sqlRecord
{
  tableNames = [["employee"]],
  keyItems = ["empnum"],
  defaultSelectCondition =
    #sqlCondition{
      aTableColumn = 4 -- 各 SQL コメントは
                      -- 二重ハイフンで開始する
    }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end

```

レコード・パーツに基づいたレコードを宣言します。

```
emp Employee;
```

SQL テーブルへの行の追加: SQL テーブルに行を追加する準備として、EGL レコードの値を格納します。

```
emp.empnum = 1;
emp.empname = "John";
```

次の EGL add ステートメントを指定してテーブルに従業員を追加します。

```

try
  add emp;
onException
  myErrorHandler(8);
end

```

SQL テーブルからの行の読み取り: SQL テーブルから行を読み取る準備として、レコード・キーを識別します。

```
emp.empnum = 1;
```

以下のいずれかの方法で単一行を取得します。

- 一連の文を生成するように EGL get ステートメントを指定する (DECLARE cursor、OPEN cursor、FETCH row、および forUpdate がない場合は CLOSE cursor)。

```
try
  get emp;
onException
  myErrorHandler(8);
end
```

- 単一の SELECT ステートメントを生成するように EGL get ステートメントを指定する。

```
try
  get emp singleRow;
onException
  myErrorHandler(8);
end
```

以下のいずれかの方法で複数の行を処理します。

- EGL の open、get next、および while ステートメントを使用する。

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
onException
  myErrorHandler(6); // プログラムを終了する
end

try
  get next emp;
onException
  if (emp not noRecordFound)
    myErrorHandler(8); // プログラムを終了する
  end
end

while (emp not noRecordFound)
  myRecord.empname = myRecord.empname + " " + "III";

  try
    replace emp;
  onException
    myErrorHandler(10); // プログラムを終了する
  end

  try
    get next emp;
  onException
    if (emp not noRecordFound)
      myErrorHandler(8); // プログラムを終了する
    end
  end
end // end while; 結果セットの最後の行が読み取られると
// カーソルは自動的にクローズされる

sysLib.commit();
```

- EGL の open および forEach ステートメントを使用する。

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp forUpdate for emp;
```

```

onException
  myErrorHandler(6);    // プログラムを終了する
end

try
  foreach (from selectEmp)
    myRecord.empname = myRecord.empname + " " + "III";

    try
      replace emp;
    onException
      myErrorHandler(10); // プログラムを終了する
    end
  end // end foreach 結果セットの最後の行が読み取られると
    // カーソルは自動的にクローズされる

onException

  // 条件が noRecordFound の場合、foreach に関連した例外ブロックは
  // 実行されないのので
  // "if (not noRecordFound)" のテストは避ける
  myErrorHandler(8); // プログラムを終了する
end

sysLib.commit();

```

明示的な SQL ステートメントでの SQL レコードの使用

明示的な SQL ステートメントで SQL レコードを使用するには、SQL レコード・パーツを宣言します。このパーツと前のパーツは、SQL 項目プロパティの構文および計算された値の使用方法が異なります。

```

Record Employee type sqlRecord
{
  tableNameVariables = [{"empTable"}],
    // テーブル名変数を使用すると
    // テーブルが実行時に指定されることを
    // 意味する
  keyItems = ["empnum"]
}
empnum decimal(6,0) { isReadOnly = yes };
empname char(40);

// 計算された列のプロパティを指定する
aValue decimal(6,0)
{ isReadOnly = yes,
  column = "(empnum + 1) as NEWNUM" };
end

```

変数を宣言します。

```

emp Employee;
empTable char(40);

```

SQL テーブルへの行の追加: SQL テーブルに行を追加する準備として、EGL レコードとテーブル名変数に値を格納します。

```

emp.empnum = 1;
emp.empname = "John";
empTable = "Employee";

```

EGL add ステートメントを指定し、SQL ステートメントを変更してテーブルに従業員を追加します。

```
// テーブル名変数の前にコロンは付かない
try
  add emp
  with #sql{
    insert into empTable (empnum, empname)
    values (:empnum, :empname || ' ' || 'Smith')
  }

onException
  myErrorHandler(8);
end
```

SQL テーブルからの行の読み取り: SQL テーブルから行を読み取る準備として、レコード・キーを識別します。

```
emp.empnum = 1;
```

以下のいずれかの方法で単一行を取得します。

- 一連の文を生成するように EGL get ステートメントを指定する (DECLARE cursor、OPEN cursor、FETCH row、CLOSE cursor)。

```
try
  get emp into empname // into 文節はオプション。(SELECT
                        // ステートメントには指定できない。)

  with #sql{
    select empname
    from empTable
    where empnum = :empnum + 1
  }
onException
  myErrorHandler(8);
end
```

- 単一の SELECT ステートメントを生成するように EGL get ステートメントを指定する。

```
try
  get emp singleRow // into 文節は
                   // SQL レコードから派生し、
                   // select 文節内の列に基づく

  with #sql{
    select empname
    from empTable
    where empnum = :empnum + 1
  }
onException
  myErrorHandler(8);
end
```

以下のいずれかの方法で複数の行を処理します。

- EGL の open、get next、および while ステートメントを使用する。

```
try

// into 文節は
// SQL レコードから派生し、
// select 文節内の列に基づく
open selectEmp forUpdate
  with #sql{
    select empnum, empname
    from empTable
    where empnum >= :empnum
    order by NEWNUM -- uses the calculated value
    for update of empname
  } for emp;
```

```

onException
  myErrorHandler(8);    // プログラムを終了する
end

try
  get next emp;
onException
  myErrorHandler(9);    // プログラムを終了する
end

while (emp not noRecordFound)
  try
    replace emp
    with #sql{
      update :empTable
      set empname = :empname || ' ' || 'III'
    } from selectEmp;

    onException
      myErrorHandler(10); // プログラムを終了する
    end

    try
      get next emp;
    onException
      myErrorHandler(9); // プログラムを終了する
    end
  end // end while

  // "close emp" は不要。emp は
  // 結果セットから最後のレコードが読み取られたとき、
  // または (例外の場合は) プログラム終了時に
  // 自動的にクローズされるため。

  sysLib.commit();

```

- EGL の open および forEach ステートメントを使用する。

```

try
  // into 文節は
  // SQL レコードから派生し、
  // select 文節内の列に基づく
  open selectEmp forUpdate
  with #sql{
    select empnum, empname
    from empTable
    where empnum >= :empnum
    order by NEWNUM      -- uses the calculated value
    for update of empname
  } from emp;

  onException
    myErrorHandler(8);    // プログラムを終了する
  end

  try
    forEach (from selectEmp)

      try
        replace emp
        with #sql{
          update :empTable
          set empname = :empname || ' ' || 'III'
        } from selectEmp;

        onException
          myErrorHandler(9); // プログラムを終了する
        end
      end
    end
  end

```

```

end

end // foreach ステートメントを終了する。
// "close emp" は不要。emp は
// 結果セットから最後のレコードが読み取られたとき、
// または（例外の場合は）プログラム終了時に
// 自動的にクローズされるため。

onException
// 条件が noRecordFound の場合、foreach に関連した例外ブロックは
// 実行されない
// "if (not noRecordFound)" のテストは避ける
myErrorHandler(9); // プログラムを終了する
end

sysLib.commit();

```

EGL prepare ステートメントの使用

EGL prepare ステートメントをコーディングする場合、オプションで SQL レコード・パーツを使用できます。以下のパーツを宣言します。

```

Record Employee type sqlRecord
{
  tableNames = ["employee"],
  keyItems = ["empnum"],
  defaultSelectCondition =
    #sqlCondition{
      aTableColumn = 4 -- 各 SQL コメントは
                      -- 二重ハイフンで開始する
    }
}

empnum decimal(6,0) {isReadOnly=yes};
empname char(40);
end

```

変数を宣言します。

```

emp Employee;
empnum02 decimal(6,0);
empname02 char(40);
myString char(120);

```

SQL テーブルへの行の追加: 行を追加する前に、変数に値を割り当てます。

```

emp.empnum = 1; emp.empname = "John";
empnum02 = 2;
empname02 = "Jane";

```

SQL ステートメントを作成します。

- EGL prepare ステートメントをコーディングし、SQL レコードを参照します。これにより、カスタマイズできる SQL ステートメントが提供されます。

```

prepare myPrep
  from "insert into employee (empnum, empname) " +
  "values (?, ?)" for emp;

// SQL レコードを使用して
// 操作の結果をテストできる
if (emp is error)
  myErrorHandler(8);
end

```

- あるいは、SQL レコードへの参照を使用せずに EGL prepare ステートメントをコーディングします。

```

myString = "insert into employee (empnum, empname) " +
"values (?, ?)";

try
  prepare addEmployee from myString;
onException
  myErrorHandler(8);
end

```

前の個々の事例では、EGL prepare ステートメントに、EGL execute ステートメントによって提供されるデータのプレースホルダーが含まれています。次に execute ステートメントの例を 2 つ示します。

- レコード (SQL またはその他) から値を供給する場合:

```
execute addEmployee using emp.empnum, emp.empname;
```

- 個々の項目から値を供給する場合:

```
execute addEmployee using empnum02, empname02;
```

SQL テーブルからの行の読み取り: SQL テーブルから行を読み取る準備として、レコード・キーを識別します。

```
empnum02 = 2;
```

以下のいずれかの方法で複数の行を置換できます。

- EGL の open、while、および get next ステートメントを使用する。

```

myString = "select empnum, empname from employee " +
"where empnum >= ? for update of empname";

try
  prepare selectEmployee from myString for emp;
onException
  myErrorHandler(8);    // プログラムを終了する
end

try
  open selectEmp with selectEmployee
    using empnum02
    into emp.empnum, emp.empname;
onException
  myErrorHandler(9);    // プログラムを終了する
end

try
  get next from selectEmp;
onException
  myErrorHandler(10);   // プログラムを終了する
end

while (emp not noRecordFound)

  emp.empname = emp.empname + " " + "III";

  try
    replace emp
      with #sql{
        update employee
        set empname = :empname
      }
    from selectEmp;
onException
  myErrorHandler(11); // プログラムを終了する
end

```



```

    try
        get next from selectEmp;
    onException
        myErrorHandler(12); // プログラムを終了
    end
end // end while; 最後の行が読み取られると close が自動的に実行される

sysLib.commit();

```

- EGL の open および forEach ステートメントを使用する。

```

myString = "select empnum, empname from employee " +
           "where empnum >= ? for update of empname";

try
    prepare selectEmployee from myString for emp;
onException
    myErrorHandler(8); // プログラムを終了する
end

try
    open selectEmp with selectEmployee
        using empnum02
        into emp.empnum, emp.empname;
onException
    myErrorHandler(9); // プログラムを終了する
end

try
    forEach (from selectEmp)
        emp.empname = emp.empname + " " + "III";

        try
            replace emp
            with #sql{
                update employee
                set empname = :empname
            }
            from selectEmp;
        onException
            myErrorHandler(11); // プログラムを終了する
        end
    end // forEach を終了する。最後の行が読み取られると close が
        自動的に実行される
onException

    // 条件が noRecordFound の場合、forEach に関連した例外ブロックは
    実行されないので
    // "if (not noRecordFound)" のテストは避ける
    myErrorHandler(12); // プログラムを終了
end

sysLib.commit();

```

デフォルト・データベース

デフォルト・データベースは、リレーショナル・データベースです。このリレーショナル・データベースは、SQL に関連した I/O ステートメントが EGL 生成コード内で実行される場合、およびその他のデータベース接続が最新でない場合にアクセスされます。デフォルト・データベースは、実行単位の先頭から使用できます。ただし、同じ実行単位内の後続のアクセスでは、別のデータベースに動的に接続できます。詳細については、『*VGLib.connectionService*』を参照してください。

デフォルト・データベースが指定されています。生成時にオプション **genProperties** に対して GLOBAL または PROGRAM を設定すると、このプロパティは、ビルド記述子オプション **sqlDB** から、生成された値を受け取ります。

関連する概念

377 ページの『Java ランタイム・プロパティ』

798 ページの『実行単位』

247 ページの『SQL サポート』

関連するタスク

394 ページの『J2EE JDBC 接続のセットアップ』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

435 ページの『sqlDB』

980 ページの『connectionService()』

Informix および EGL

Informix データベースおよび EGL に固有の規則を次に示します。

- EGL または EGL 生成プログラムによってアクセスされる Informix データベースは、トランザクションに対応している必要があります。
- SQL ステートメントをコーディングする際に、Informix 表の識別にコロンの (:) を使用する場合は、以下の例のように、Informix 識別子をステートメントのその他の部分と区別するために引用符を使用します。

```
INSERT INTO "myDB:myTable"  
(myColumn) values (:myField)
```

```
INSERT INTO "myDB@myServer:myTable"  
(myColumn) values (:myField)
```

- EGL の SQL 検索機能を使用して非 ANSI Informix データベースのデータにアクセスする場合は、DECIMAL 型のデータベース列にスケール値が含まれている必要があります。例えば、DECIMAL (4) として列を定義するのではなく、DECIMAL (4,0) として列を定義します。
- Informix システム・スキーマに含まれるテーブルのデータを SQL 検索機能を使用して検索する場合は、特別な設定に変更する必要があります。詳細については、『SQL 検索における設定の変更』を参照してください。

関連する概念

247 ページの『SQL サポート』

関連するタスク

272 ページの『SQL テーブル・データの検索』

128 ページの『SQL 検索設定の変更』

SQL 特定のタスク

SQL テーブル・データの検索

EGL では、SQL テーブルの定義から SQL レコード項目を作成し、この項目を表示または結合する方法が提供されています。概要については、『SQL サポート』を参照してください。

以下のようにします。

1. SQL 設定が適切に設定されていることを確認する。詳細については、『SQL 検索設定の変更』を参照してください。
2. タスクの実行場所を決定する。
 - EGL ソース・ファイル内 (各 SQL レコードを開発した場合と同様)
 - 「アウトライン」ビュー内 (すでに SQL レコードが存在する場合)
3. EGL ソース・ファイルで作業を行っている場合は、以下の手順を実行する。
 - a. SQL レコードが存在しない場合は、このレコードを作成する。
 - 1) 「R」を入力し、Ctrl-Space を押して、「コンテンツ・アシスト」リスト内で SQL テーブル・エンタリ (通常は「表名付き SQL レコード」) を選択する。
 - 2) SQL レコードの名前を入力し、Tab を押してテーブル名を入力するか、テーブルのコンマで区切られたリストを入力するか、またはビューの別名を入力する。

最小限のコンテンツを入力して SQL レコードを作成することもできます。この方法は、レコード名がテーブル名と同じ場合に使用します。たとえば、以下のように入力します。

```
Record myTable type sqlRecord  
end
```

- b. レコードで任意の個所を右クリックします。
 - c. コンテキスト・メニューで、「SQL レコード」>「SQL を検索」をクリックする。
4. 「アウトライン」ビューで作業を行っている場合は、SQL レコードのエンタリを右クリックし、コンテキスト・メニューで「SQL を検索」をクリックする。

注: DB2 の条件 WITH CHECK OPTIONS を使用して定義されている SQL ビューは検索できません。

レコード項目の作成後に、同等のデータ項目パーツを作成することによって、生産性の利点を得ることができます。『SQL レコード・パーツからのデータ項目パーツの生成 (概要)』を参照してください。

関連する概念

273 ページの『SQL レコード・パーツからのデータ項目パーツの作成 (概説)』

247 ページの『SQL サポート』

関連するタスク

『SQL レコード・パーツからのデータ項目パーツの作成』

126 ページの『SQL データベース接続設定の変更』

128 ページの『SQL 検索設定の変更』

関連する参照項目

71 ページの『SQL 項目のプロパティ』

SQL レコード・パーツからのデータ項目パーツの作成 (概説)

SQL レコード・パーツで構造体項目を宣言した後、EGL エディターの特別なメカニズムを使用して、構造体項目と等価のデータ項目パーツを作成できます。利点として、実行時に関連する SQL レコード間でデータを転送するための 非 SQL レコード (通常は基本レコード) を容易に作成できます。

以下の構造体項目について考えます。

```
10 myHostVar01 CHAR(3);  
10 myHostVar02 BIN(9,2);
```

次のようにして、データ項目パーツを作成するよう要求することができます。

```
DataItem myHostVar01 CHAR(3) end  
  
DataItem myHostVar02 BIN(9,2) end
```

さらに、構造体項目宣言が上書きされる、という影響も受けます。

```
10 myHostVar01 myHostVar01;  
10 myHostVar02 myHostVar02;
```

この例に示すように、各データ項目パーツには関連する構造体項目と同じ名前が割り当てられ、構造体項目の `typedef` として機能します。また、各データ項目パーツは他の構造体項目の `typedef` としても使用できます。

データ項目パーツの基礎として構造体項目を使用するには、その構造体項目に名前が必要であり、有効な基本特性を持っている必要がありますが、`typedef` を指してはいけません。

関連する概念

247 ページの『SQL サポート』

関連するタスク

『SQL レコード・パーツからのデータ項目パーツの作成』

関連する参照項目

512 ページの『EGL ソース形式の DataItem パーツ』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

SQL レコード・パーツからのデータ項目パーツの作成

SQL レコード・パーツ内で構造体項目を宣言した後、EGL エディターの特殊なメカニズムを使用して、それらの構造体項目と等価のデータ項目パーツを作成できます。一般情報については、『SQL レコード・パーツからデータ項目パーツを作成する方法の概要 (Overview on creating dataitem parts from an SQL record part)』を参照してください。

「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「**ビューの表示**」>「**アウトライン**」を選択して、そのビューを開く。

「アウトライン」ビューで以下を実行します。

1. 特定の SQL レコード・パーツの場合は、**Ctrl** を押したまま、対象とする各構造体項目をクリックする。特定のレコードですべての構造体項目を選択するには、最上位の構造体項目をクリックし、**Shift** を押したまま、最下部の構造体項目をクリックします。
2. 選択した構造体項目を右クリックする。
3. コンテキスト・メニューで、「**データ項目パーツを作成**」をクリックする。

データ項目パーツは EGL ソース・ファイルの下部に書き込まれます。各構造体項目は、同等のパーツを参照するよう変更されます。

関連する概念

273 ページの『SQL レコード・パーツからのデータ項目パーツの作成 (概説)』

247 ページの『SQL サポート』

関連するタスク

272 ページの『SQL テーブル・データの検索』

関連する参照項目

804 ページの『EGL ソース形式の SQL レコード・パーツ』

リレーショナル・データベース・テーブルからの EGL データ・パーツの作成

EGL データ・パーツ・ウィザード

EGL データ・パーツ・ウィザードを使用すると、SQL レコード・パーツのほか、関連するデータ項目パーツ、およびライブラリー・ベースの関数パーツを、1 つ以上のリレーショナル・データベース表または事前に存在するビューから作成できます。

データベースに接続した後、次のことができます。

- 所定のデータベース表またはビューから行の作成、読み取り、更新、削除を行うために使用する SQL レコードのキー・フィールドを指定する。
- 行の作成、読み取り、または更新を行うための明示的な SQL ステートメントをカスタマイズする。(行を削除する SQL ステートメントをカスタマイズすることはできません。)
- 所定のデータベース表またはビューから一連の行を選択するために使用する SQL レコードのキー・フィールドを指定する。
- 一連の行を選択する明示的な SQL ステートメントをカスタマイズする。
- 各 SQL ステートメントを検証し、実行する。

出力には、以下のファイルが含まれます。

- 各レコード・パーツを定義している EGL ソース・ファイル
- 各レコード・パーツ用の EGL ライブラリー

- SQL レコード・パーツ内の構造体項目によって参照されるすべてのデータ項目が入っている EGL ソース・ファイル

「同じファイル内のレコードとライブラリー (Record and library in the same file)」チェック・ボックスを選択した場合は、ファイルの数を減らすことができます。

関連する概念

247 ページの『SQL サポート』

関連するタスク

276 ページの『EGL ウィザード用のデータベース接続の作成、編集、または削除』

『リレーショナル・データベース・テーブルからの EGL データ・パーツの作成』

277 ページの『EGL ウィザードでの SQL ステートメントのカスタマイズ』

リレーショナル・データベース・テーブルからの EGL データ・パーツの作成

別個の Web アプリケーションを作成せずに、リレーショナル・データベース・テーブルから EGL データ・パーツを作成するには、次のようにします。

1. 「ファイル」>「新規」>「その他...」を選択する。ウィザードを選択するためのダイアログが表示されます。
2. 「EGL」を展開し、「EGL データ・パーツ」をダブルクリックする。「EGL データ・パーツ」ダイアログが表示されます。
3. EGL プロジェクト名または EGL Web プロジェクト名を入力するか、ドロップダウン・リストから既存プロジェクトを選択する。パーツがそのプロジェクト内に生成されます。
4. 既存のデータベース接続をドロップダウン・リストから選択するか、次のようにして新規データベース接続を確立する。
 - 新規データベース接続を確立するには、「追加」をクリックし、ヘルプ・トピック『新規データベース接続ウィザード』と対話する。特定フィールドで必要な入力の種類に関する詳細については、そのフィールドを右クリックし、F1 を押してください。
 - データベース接続の編集または削除の詳細については、『EGL ウィザード用のデータベース接続の作成、編集、または削除』を参照してください。

データベースへの接続が作成されると、データベース・テーブルのリストが表示されます。

5. デフォルトのデータ項目の EGL ファイル名を受け入れたくない場合は、新しいファイル名を入力する。
6. 「データの選択」フィールドで、データ・パーツを宣言するのに役立つ列を含んでいるテーブルの名前をクリックする。複数のテーブルを選択するには、**Ctrl** キーを押したまま、さまざまなテーブル名をクリックします。強調表示された名前（単数または複数）を選択されたテーブルのリストへ転送するには、右矢印をクリックします。

7. 選択された (右側の) テーブルごとに、作成する EGL レコードの名前を指定するか、デフォルト名を受け入れる。 リストから 1 つまたは複数のテーブルを除去するには、求めるエントリーを強調表示にして、左矢印をクリックします。
8. ライブラリー・パーツと SQL レコード・パーツを同じファイルに組み込みたい場合は、チェック・ボックスを選択する。
9. 「次へ」をクリックする。
10. それぞれのテーブルごとに 1 つのタブを使用できる。それぞれのタブで、個々の行の読み取り中、更新中、および削除中に使用するキー・フィールドを選択し、右矢印をクリックします。複数のキー・フィールドを選択するには、**Ctrl** キーを押したまま、さまざまなフィールド名をクリックします。右側のリストからキー・フィールドを除去するには、フィールド名を強調表示にして、左矢印をクリックします。
11. 行セットを選択するときに使用する選択条件フィールドを選択し、右矢印をクリックする。複数のフィールドを選択するには、**Ctrl** キーを押したまま、さまざまなフィールド名をクリックします。右側のリストからフィールドを除去するには、フィールド名を強調表示にして、左矢印をクリックします。
12. 暗黙の SQL ステートメントをカスタマイズするには、『EGL ウィザードでの SQL ステートメントのカスタマイズ』を参照する。このオプションは、EGL の delete ステートメントには使用できません。
13. 「次へ」をクリックする。
14. 「EGL データ・パーツの生成」画面が表示される。これには、生成されるファイルのリスト (画面下部) も含まれます。
 - a. EGL パーツを受け取る EGL プロジェクトの名前を変更するには、「宛先プロジェクト」フィールドにプロジェクト名を入力するか、関連するドロップダウン・リストからプロジェクトの選択を行います。
 - b. 特定のパーツ型 (データまたはライブラリー) 用の EGL パッケージを指定するには、関連するフィールドにパッケージ名を入力するか、関連するドロップダウン・リストから名前を選択します。
15. 「完了」をクリックする。

関連する概念

274 ページの『EGL データ・パーツ・ウィザード』

199 ページの『「EGL データ・パーツおよびページ」ウィザード』

247 ページの『SQL サポート』

関連するタスク

201 ページの『単一テーブルの EGL Web アプリケーションの作成』

『EGL ウィザード用のデータベース接続の作成、編集、または削除』

277 ページの『EGL ウィザードでの SQL ステートメントのカスタマイズ』

EGL ウィザード用のデータベース接続の作成、編集、または削除: リレーショナル・データベース・テーブルからデータ・パーツを作成するか、リレーショナル・データベース・テーブルから Web アプリケーションを作成するための EGL ウィザードの最初の画面では、次に示す 2 つの方法のいずれかでデータベース接続を指定します。

- ドロップダウン・リストから既存の接続を選択する。または、

- 「新規データベース接続ウィザード」と対話する

そのウィザードを使用して接続を作成するには、「追加」をクリックし、必要に応じて情報を追加してください。特定フィールドで必要な入力の種類に関する詳細については、そのフィールドを右クリックし、F1 を押してください。

既存のデータベース接続を編集するには、次のようにします。

1. 「ウィンドウ」>「パースペクティブを開く」>「その他 (Other)」を選択する。
「パースペクティブの選択」ダイアログで、「すべて表示」チェック・ボックスを選択し、「データ」をダブルクリックする。
2. データベース・エクスプローラー・ビューで、データベース接続を右クリックしてから、「接続の編集」を選択する。データベース接続ウィザードの各ページを表示し、必要に応じて情報を変更します。ヘルプが必要な場合は、F1 を押します。
3. 編集を完了するには、「完了」をクリックする。

既存のデータベース接続を削除するには、次のようにします。

1. 「ウィンドウ」>「パースペクティブを開く」>「その他 (Other)」を選択する。
「パースペクティブの選択」ダイアログで、「すべて表示」チェック・ボックスを選択し、「データ」をダブルクリックする。
2. データベース・エクスプローラー・ビューで、データベース接続を右クリックしてから、「削除」を選択する。

関連する概念

274 ページの『EGL データ・パーツ・ウィザード』

199 ページの『「EGL データ・パーツおよびページ」ウィザード』

247 ページの『SQL サポート』

関連するタスク

201 ページの『単一テーブルの EGL Web アプリケーションの作成』

274 ページの『リレーショナル・データベース・テーブルからの EGL データ・パーツの作成』

119 ページの『EGL 設定の変更』

EGL ウィザードでの SQL ステートメントのカスタマイズ: リレーショナル・テーブルからデータ・パーツを作成するか、リレーショナル・テーブルから Web アプリケーションを作成するために EGL ウィザードを使用する場合は、次のように、読み取りまたは更新のようなアクションに関連した SQL ステートメントを変更できます。

1. 「編集アクション」リストからアクションを選択し、「SQL の編集」をクリックする。
2. SQL ステートメントの編集 (削除以外のすべてのアクションで可能) を行い、「検証」をクリックする。検証によって文の構文が正しいかどうかを確認でき、ホスト変数名の規則を準拠していることを確認することができる。文にエラーが含まれている場合は、メッセージが表示されます。エラーを訂正し、再度検証を行ってください。

「最後に戻す」は、文を最後に有効だった変更済みバージョンに変更します。以前の各バージョンは、ダイアログを閉じた後、使用不可になります。

3. 「実行」をクリックし、再度「実行」をクリックする。
4. SQL ステートメントにホスト変数の値が必要な場合は、「変数値の指定」ダイアログが表示される。「値」フィールドをダブルクリックしてホスト変数の値を入力し、**Enter** キーを押します。すべてのホスト変数に値を入力し終わったならば、「完了」をクリックします。

注: 文字 型として定義されたホスト変数の場合は、値を単一引用符で囲む必要があります。

5. SQL ステートメントの実行が終了したならば、「閉じる」をクリックする。
6. SQL ステートメントの編集が終了したならば、「OK」をクリックする。

関連する概念

274 ページの『EGL データ・パーツ・ウィザード』

199 ページの『「EGL データ・パーツおよびページ」ウィザード』

247 ページの『SQL サポート』

関連するタスク

201 ページの『単一テーブルの EGL Web アプリケーションの作成』

274 ページの『リレーショナル・データベース・テーブルからの EGL データ・パーツの作成』

276 ページの『EGL ウィザード用のデータベース接続の作成、編集、または削除』

119 ページの『EGL 設定の変更』

SQL レコードの SQL SELECT ステートメントの表示

EGL では、特定の SQL レコード・パーツの暗黙的な SQL SELECT ステートメントが提供されています。暗黙的な SQL SELECT ステートメントを表示するには、次のようにします。

1. SQL レコード・パーツを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「アプリケーションから開く」>「EGL エディター」を選択する。
2. SQL レコード・パーツ内をクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「SQL Record (SQL レコード)」>「デフォルト選択を表示」を選択する。
4. SQL SELECT ステートメントをデータベースに対して検証するには、**検証**を選択する。

注: 検証関数を使用する前に、DB2 UDB ユーザーは DEFERREDPREPARE オプションを設定する必要があります。このオプションは、**db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** コマンド を使用して、CLP (DB2 コマンド行プロセッサ) 内で対話式に設定できます。このコマンドにより、COMMON セクションにキーワードが配置されます。コマンド **db2 get cli cfg for section common** を実行し、キーワードがピックアップされていることを検証します。

関連する概念

247 ページの『SQL サポート』

関連するタスク

『SQL レコードの SQL SELECT ステートメントの検証』

関連する参照項目

804 ページの『EGL ソース形式の SQL レコード・パーツ』

SQL レコードの SQL SELECT ステートメントの検証

EGL では、特定の SQL レコード・パーツの暗黙的な SQL SELECT ステートメントが提供されています。暗黙的な SQL SELECT ステートメントをデータベースに対して検証するには、次のようにします。

1. SQL レコード・パーツを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「**アプリケーションから開く**」>「**EGL エディター**」を選択する。
2. SQL レコード・パーツ内をクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「**SQL Record (SQL レコード)**」>「**デフォルト選択を検証**」を選択する。

注: 検証関数を使用する前に、DB2 UDB ユーザーは DEFERREDPREPARE オプションを設定する必要があります。このオプションは、**db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** コマンドを使用して、CLP (DB2 コマンド行プロセッサ) 内で対話式に設定できます。このコマンドにより、COMMON セクションにキーワードが配置されます。コマンド **db2 get cli cfg for section common** を実行し、キーワードがピックアップされていることを検証します。

関連する概念

247 ページの『SQL サポート』

関連するタスク

278 ページの『SQL レコードの SQL SELECT ステートメントの表示』

関連する参照項目

804 ページの『EGL ソース形式の SQL レコード・パーツ』

EGL prepare ステートメントの構成

関数内で、SQL レコード・パーツに基づいて、次の種類の EGL ステートメントを構成できます。

- EGL **prepare** ステートメント、および
- 関連する EGL **execute**、**open**、または **get** ステートメント

以下のようにします。

1. EGL エディターで EGL ファイルを開く。このファイルには、関数とコード化された SQL ステートメントが含まれている必要があります。まだファイルを開いていない場合は、ワークベンチのプロジェクト・エクスプローラーで EGL ファイルを右クリックし、「**アプリケーションから開く**」>「**EGL エディター**」を選択します。
2. 関数内で、EGL **prepare** ステートメントを配置するロケーションをクリックしてから、右クリックする。コンテキスト・メニューが表示されます。

3. 「SQL Prepare ステートメントを追加」を選択する。
4. EGL **prepare** ステートメントを識別する名前を入力する。命名の規則については、『命名規則』を参照してください。
5. SQL レコード変数を定義した場合は、ドロップダウン・リストからこの変数を選択する。対応する SQL レコード・パーツ名が表示されます。SQL レコード変数を定義していない場合は、「SQL レコード変数名」フィールドに名前を入力し、「参照」ボタンを使用して SQL レコード・パーツ名を選択します。EGL ソース・コード内で、その名前を使用して SQL レコード変数を定義する必要があります。
6. ドロップダウン・リストから実行ステートメント型を選択する。
7. 実行ステートメントが open 型である場合は、結果セット ID を入力する。
8. 「OK」をクリックする。EGL ステートメントが関数内部に構成されます。

関連する概念

247 ページの『SQL サポート』

関連するタスク

279 ページの『SQL レコードの SQL SELECT ステートメントの検証』

278 ページの『SQL レコードの SQL SELECT ステートメントの表示』

関連する参照項目

725 ページの『命名規則』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

暗黙的なステートメントからの明示的な SQL ステートメントの構成

EGL では、SQL 関連 EGL input/output (I/O) ステートメントごとに 暗黙的な SQL ステートメントが提供されています。暗黙的な SQL ステートメントから明示的な SQL ステートメントを構成するには、次のようにします。

1. EGL I/O ステートメントを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「アプリケーションから開く」>「EGL エディター」を選択する。
2. EGL I/O ステートメントをクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. INTO 文節を付けずに明示的な SQL ステートメントを構成するには、「SQL Statement (SQL ステートメント)」>「追加」を選択する。INTO 文節を付けて明示的な SQL ステートメントを構成するには、「SQL Statement (SQL ステートメント)」>「into で追加」を選択します。暗黙的な SQL ステートメントは EGL I/O ステートメントに追加され、明示的な SQL ステートメントになります。

注: INTO 文節は、open、get、および get next ステートメントとともに使用する場合にのみ有効です。

関連する概念

247 ページの『SQL サポート』

関連するタスク

280 ページの『SQL 関連EGL ステートメントからのSQL ステートメントの除去』

282 ページの『明示的な SQL ステートメントのリセット』

『暗黙的または明示的な SQL ステートメントの検証』

『SQL 関連 EGL ステートメントの暗黙的な SQL の表示』

SQL 関連 EGL ステートメントの暗黙的な SQL の表示

EGL では、SQL 関連 EGL input/output (I/O) ステートメントごとに 暗黙的な SQL ステートメントが提供されています。 EGL I/O ステートメントの暗黙的な SQL を表示するには、次のようにします。

1. EGL I/O ステートメントを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「**アプリケーションから開く**」>「**EGL エディター**」を選択する。
2. EGL I/O ステートメントをクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「**SQL Statement (SQL ステートメント)**」>「**表示**」を選択する。

関連する概念

247 ページの『SQL サポート』

関連するタスク

280 ページの『暗黙的なステートメントからの明示的なSQL ステートメントの構成』

280 ページの『SQL 関連EGL ステートメントからのSQL ステートメントの除去』

282 ページの『明示的な SQL ステートメントのリセット』

『暗黙的または明示的な SQL ステートメントの検証』

暗黙的または明示的な SQL ステートメントの検証

暗黙的または明示的な SQL ステートメントをデータベースに対して検証するには、次のようにします。

1. SQL 関連 EGL ステートメントまたは明示的な SQL ステートメントを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「**アプリケーションから開く**」>「**EGL エディター**」を選択する。
2. EGL ステートメントまたは SQL ステートメントをクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「**SQL Statement (SQL ステートメント)**」>「**検証**」を選択する。

注: 検証関数を使用する前に、DB2 UDB ユーザーは DEFERREDPREPARE オプションを設定する必要があります。このオプションは、**db2 update cli cfg for section COMMON using DEFERREDPREPARE 0** コマンド を使用して、CLP (DB2 コマンド行プロセッサ) 内で対話式に設定できます。このコマンドにより、COMMON セクションにキーワードが配置されます。 コマンド **db2 get cli cfg for section common** を実行し、キーワードがピックアップされていることを検証します。

関連する概念

247 ページの『SQL サポート』

関連するタスク

280 ページの『暗黙的なステートメントからの明示的なSQL ステートメントの構成』

『SQL 関連 EGL ステートメントからの SQL ステートメントの除去』

『明示的な SQL ステートメントのリセット』

281 ページの『SQL 関連 EGL ステートメントの暗黙的な SQL の表示』

明示的な SQL ステートメントのリセット

EGL では、SQL 関連 EGL input/output (I/O) ステートメントごとに 暗黙的な SQL ステートメントが提供されています。暗黙的な SQL ステートメントは EGL I/O ステートメントに追加され、明示的な SQL ステートメントになります。明示的な SQL ステートメントを変更する場合は、以下の手順を実行して、暗黙的な SQL に基づいた SQL ステートメントに戻します。

1. 明示的な SQL ステートメントを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「アプリケーションから開く」>「EGL エディター」を選択する。
2. 明示的な SQL ステートメントをクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「SQL Statement (SQL ステートメント)」>「リセット」を選択する。

関連する概念

247 ページの『SQL サポート』

関連するタスク

280 ページの『暗黙的なステートメントからの明示的なSQL ステートメントの構成』

『SQL 関連 EGL ステートメントからの SQL ステートメントの除去』

281 ページの『暗黙的または明示的な SQL ステートメントの検証』

281 ページの『SQL 関連 EGL ステートメントの暗黙的な SQL の表示』

SQL 関連 EGL ステートメントからの SQL ステートメントの除去

EGL では、SQL 関連 EGL input/output (I/O) ステートメントごとに 暗黙的な SQL ステートメントが提供されています。暗黙的な SQL ステートメントは、EGL I/O ステートメントに追加され、明示的な SQL ステートメント になります (『暗黙的なステートメントからの明示的な SQL ステートメントの構成』を参照)。追加した SQL ステートメントを除去するには、次のようにします。

1. 明示的な SQL ステートメントを含む EGL ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで EGL ファイルを右クリックして、「アプリケーションから開く」>「EGL エディター」を選択する。
2. 明示的な SQL ステートメントをクリックし、右クリックする。コンテキスト・メニューが表示されます。
3. 「SQL Statement (SQL ステートメント)」>「除去」を選択する。EGL I/O ステートメントは残ります。

関連する概念

247 ページの『SQL サポート』

関連するタスク

280 ページの『暗黙的なステートメントからの明示的なSQL ステートメントの構成』

282 ページの『明示的な SQL ステートメントのリセット』

281 ページの『暗黙的または明示的な SQL ステートメントの検証』

281 ページの『SQL 関連 EGL ステートメントの暗黙的な SQL の表示』

暗黙的 SQL ステートメントを表示するための参照の解決

以下の EGL ステートメントを指定した場合を考えてください。

```
open myRecord;
```

EGL エディターがデフォルト SQL ステートメントの作成を試行する場合、エディターは `myRecord` という名前の変数を検索し、この変数の 基になる SQL レコード・パーツを識別しようとします。開発時で変数を使用できない場合や、変数が宣言されていない場合、エディターは、`myRecord` という名前の SQL レコード・パーツを、デフォルト SQL ステートメントの基本として使用しようとします。エディターは、SQL レコード・パーツ名を名前に持つ変数を作成するものと想定します。

SQL 関連の関数、変数 `myRecord` を含まないファイルに格納する場合は、以下のようになります。

1. プログラム・パーツでグローバル変数を宣言します。
2. プログラム・パーツで、ネスト関数として関数を作成します。
3. デフォルトの SQL ステートメントを作成し、必要に応じて変更してから、ファイルを保管します。
4. 関数を別のファイルに移動します。

関数をプログラム・パーツから移動した後は、開発時にレコード名を解決できません。また、エディターは、そのレコードを基にしたどのデフォルト SQL ステートメントも表示できません。

関連する概念

247 ページの『SQL サポート』

標準 JDBC 接続の作成方法について

生成された Java プログラムをデバッグしている場合、およびプログラム・プロパティ・ファイルに必要な値が含まれている場合は、実行時に標準 JDBC 接続が作成されます。値の派生方法も含め、プログラム・プロパティの意味の詳細については、『*Java ランタイム・プロパティ (参照)*』を参照してください。

JDBC 接続は以下の種類の情報を基にしています。

接続 URL

システム関数 `sysLib.connect` または `VGLib.connectionService` を呼び出す前にコードがデータベースへのアクセスを試みた場合、接続 URL はプロパティ `vgj.jdbc.default.database` の値になります。

システム関数 `sysLib.connect` または `VGLib.connectionService` の呼び出しに応じてコードがデータベースへのアクセスを試みた場合、接続 URL は プロパティ `vgj.jdbc.databaseSN` の値です。

接続 URL の形式の詳細については、『`sqlValidationConnectionURL`』を参照してください。

ユーザー ID

システム関数 `sysLib.connect` または `VGLib.connectionService` を呼び出す前にコードがデータベースへのアクセスを試みた場合、ユーザー ID はプロパティ `vgj.jdbc.default.userid` の値です。

これらのシステム関数のうちいずれか 1 つの呼び出しに応じてコードがデータベースへのアクセスを試みた場合、ユーザー ID は呼び出しで指定された値です。

パスワード

システム関数 `sysLib.connect` または `VGLib.connectionService` を呼び出す前にコードがデータベースへのアクセスを試みた場合、パスワードはプロパティ `vgj.jdbc.default.password` の値です。

これらのシステム関数のうちいずれか 1 つの呼び出しに応じてコードがデータベースへのアクセスを試みた場合、パスワードは呼び出しで指定された値です。システム機能を使用して、プログラム・プロパティ・ファイルでのパスワードの公開を回避することができます。

JDBC ドライバー・クラス

JDBC ドライバー・クラスは、プロパティ `vgj.jdbc.drivers` の値です。

関連する概念

380 ページの『プログラム・プロパティ・ファイル』

関連するタスク

394 ページの『J2EE JDBC 接続のセットアップ』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

957 ページの『`connect()`』

980 ページの『`connectionService()`』

428 ページの『`genProperties`』

584 ページの『Java ランタイム・プロパティ (詳細)』

604 ページの『EGL での JDBC ドライバーの要件』

435 ページの『`sqlIDB`』

436 ページの『`sqlID`』

438 ページの『`sqlPassword`』

438 ページの『`sqlValidationConnectionURL`』

436 ページの『`sqlJDBCDriverClass`』

VSAM サポート

EGL 生成 Java コードの VSAM サポートは、次のとおりです。

- AIX ベースのコードは、ローカルの VSAM ファイルにアクセスできる
- 以下のコードは、z/OS 上のリモートの VSAM ファイルにアクセスできる
 - Windows 2000/NT/XP で実行される EGL 生成 Java コード
 - Windows 2000/NT/XP で実行される EGL デバッガー

アクセスの前提条件

アクセスするには、まず最初に、VSAM ファイルを常駐させるシステム上で VSAM ファイルを定義する必要があります。Windows 2000/NT/XP (EGL デバッガーのためにあるいは実行時に) からのリモート・アクセスについては、以下のように、ワークステーションに Distributed File Manager (DFM) をインストールする必要もあります。

1. 次のファイルを EGL インストール・ディレクトリーに配置します。

`workbench\bin\VSAMWIN.zip`

2. 新規ディレクトリーにファイルを unzip し、INSTALL.README ファイルの指示に従います。

システム名

ローカルの VSAM ファイルにアクセスするには、リソース関連パーツでシステム名を指定し、オペレーティング・システムに応じた命名規則を使用します。EGL デバッガーまたは EGL 生成 Java コードからリモートの VSAM ファイルにアクセスするには、以下のように、システム名を定義します。

`%%machineName%qualifier.fileName`

machineName

SNA 構成で指定された、SNA LU 別名

qualifier.fileName

VSAM データ・セット名 (修飾子を含む)

命名規則は、汎用命名規則 (UNC) 形式と同様です。UNC 形式の詳細については、「*Distributed FileManager User's Guide*」を参照してください。この資料は EGL インストール・ディレクトリー内の以下のディレクトリーにあります。

`workbench\bin\VSAMWIN.zip`

MQSeries のサポート

EGL では任意のターゲット・プラットフォーム上で MQSeries メッセージ・キューへのアクセスがサポートされます。以下のいずれかの方法でアクセスを提供できます。

- MQSeries 関連の EGL キーワード (**add** や **get next** など) を MQ レコードで使用する。この場合、EGL によって MQSeries の詳細が隠蔽されるので、コードが取り扱うビジネスの問題に集中できます。

- MQSeries コマンドを直接呼び出す EGL 関数を呼び出す。この方法を使用した場合、EGL キーワードによってサポートされていない一部のコマンドが使用可能になります。

1 つのプログラムで上記の 2 つの方法を組み合わせで使用することもできます。しかし、通常はいずれか一方のみを使用します。

どの方法を使用するかにかかわらず、オプション・レコード をカスタマイズすることによってさまざまなランタイム条件を制御できます。このレコードは、EGL ランタイム・サービスが MQSeries への呼び出しで渡すグローバル基本レコードです。オプション・レコードをプログラム変数として宣言すると、EGL のインストール済みオプション・レコード・パーツを `typedef` として使用できます。つまり、インストール済みパーツを自身の EGL ファイルにコピーしてパーツをカスタマイズし、カスタマイズしたパーツを `typedef` として使用することができます。

どの方法を使用するかによって、EGL ランタイム・サービスで MQSeries に対してオプション・レコードを有効にする方法が異なります。

- EGL の **add** ステートメントや **get next** ステートメントを使用して作業する場合は、MQ レコードのプロパティを指定するときにオプション・レコードを指示します。特定のオプション・レコードを指示しない場合は、EGL によってデフォルトが使用されます。
- MQSeries を直接呼び出す EGL 関数を呼び出す場合は、関数の呼び出し時に引数としてオプション・レコードを使用します。この場合デフォルトは利用不能です。

オプション・レコードおよびデフォルトで MQSeries に渡される値の詳細については、『MQ レコード用のオプション・レコード』を参照してください。MQSeries 自体の詳細については、以下の資料を参照してください。

- 「*An Introduction to Messaging and Queueing*」 (GC33-0805?01)
- 「*MQSeries MQI Technical Reference*」 (SC33-0850)
- 「*MQSeries アプリケーション・プログラミング・ガイド*」 (SC88-7253-09)
- 「*MQSeries アプリケーション・プログラミング解説書*」 (SC88-7354-05)

接続

以下を実行して最初に文を呼び出したときに、キュー・マネージャー (接続キュー・マネージャー) に接続されます。

- メッセージ・キューにアクセスする EGL の **add** または **get next** ステートメント
- EGL 関数 **MQCONN** または **MQCONNEX** の呼び出し

1 度に接続できる接続キュー・マネージャーは 1 つのみです。ただし、接続キュー・マネージャーの管理下にあるキューであれば 1 度に複数のキューに接続できます。現行の接続キュー・マネージャー以外のキュー・マネージャーに直接接続するには、まず **MQDISC** を呼び出して最初のキュー・マネージャーを切断してから、**add**、**get next**、**MQCONN**、**MQCONNEX** のいずれかを呼び出して 2 番目のキュー・マネージャーに接続する必要があります。

また、リモート・キュー・マネージャー の管理下にあるキューにアクセスできます。リモート・キュー・マネージャーとは、接続キュー・マネージャーとの相互作用が可能なキュー・マネージャーです。2 つのキュー・マネージャー間のアクセスは、MQSeries 自体がこのようなアクセスを許可するよう構成されている場合にのみ可能です。

接続キュー・マネージャーへのアクセスが終了するのは、MQDISC を呼び出すか、コードが終了した場合です。

トランザクションにメッセージを組み込む

作業単位 (UOW) に `queue-access` ステートメントを組み込むことができます。これにより、キューに対する変更すべてを単一の処理ポイントでコミットまたはロールバックできます。文が作業単位内にある場合、以下のことが当てはまります。

- EGL の **get next** ステートメント (または EGL の MQGET 呼び出し) によってメッセージが除去されるのは、コミットが発生した場合のみである
- EGL の **add** ステートメント (または EGL の MQPUT 呼び出し) によってキューに入れられたメッセージは、コミットが発生した場合にのみ作業単位の外部から見える

`queue-access` ステートメントが作業単位内でない場合は、メッセージ・キューに対する変更はそれぞれ即時にコミットされます。

MQSeries 関連の EGL の **add** ステートメントまたは **get next** ステートメントは、プロパティ **includeMsgInTransaction** が MQ レコードに対して有効である場合に作業単位に組み込まれます。生成されたコードには以下のオプションが含まれています。

- MQGET の場合は MQGMO_SYNCPOINT
- MQPUT の場合は MQPMO_SYNCPOINT

プロパティ **includeMsgInTransaction** を MQ レコードに対して指定しないと、`queue-access` ステートメントが作業単位の外部で実行されます。生成されたコードには以下のオプションが含まれています。

- MQGET の場合は MQGMO_NO_SYNCPOINT
- MQPUT の場合は MQPMO_NO_SYNCPOINT

コードによって作業単位が終了すると、EGL はプログラムによってアクセスされているすべての回復可能リソースをコミットまたはロールバックします。これにはデータベース、メッセージ・キュー、および回復可能ファイルが含まれます。このような結果は、システム機能 (**sysLib.commit**、**sysLib.rollback**) を使用する場合、または EGL が MQSeries (MQCMIT, MQBACK) を呼び出す場合にも発生します。いずれの場合も適切な EGL システム機能が呼び出されます。

ロールバックが発生するのは、EGL ランタイム・サービスにより検出されたエラーが原因で EGL プログラムが早期に終了した場合です。

カスタマイズ

add ステートメントと **get next** ステートメントのデフォルト処理に依存せずに MQSeries との相互作用をカスタマイズする場合は、このセクションの情報を検討してください。

EGL dataTable パーツ

EGL dataTable パーツ一式は、MQSeries と対話する場合に有効です。実行時に、各パーツにより、EGL 提供の関数がメモリー・ベースのリストから値を検索できるようにします。次のセクションでは、データ・テーブルの配置方法の詳細について説明します。

カスタマイズを可能にする方法

カスタマイズを可能にするには、各種のインストール済み EGL ファイル を一切変更せずに プロジェクトに導入する必要があります。ファイルには以下のものがあります。

records.egl

プログラムで使用されるオプション・レコード用の **typedefs** として使用可能な、基本レコード・パーツが含まれています。また、それらのレコードによって使用される構造体パーツも含まれています。これによって柔軟性が提供されるため、独自のレコード・パーツを開発することができます。

functions.egl

以下の 2 組の関数が含まれています。

- MQSeries コマンド機能。MQSeries に直接アクセスします。
- 初期化関数。プログラムで使用されるオプション・レコードに初期値を設定できます。

mqrcode.egl、mqrc.egl、mqvalue.egl

コマンドおよび初期化関数によって使用される一連の EGL dataTable パーツが含まれています。

作業内容は次のとおりです。

1. ワークベンチにファイルをインポートするためのプロセスを使用し、それらのファイルを EGL プロジェクトに導入します。ファイルは以下のディレクトリーに存在します。

```
installationDir%egl%eclipse%plugins%  
com.ibm.etools.egl.generators_version%MqReusableParts
```

installationDir

製品のインストール・ディレクトリー。例えば、C:\Program Files\IBM\RSPPD\6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストール時に使用されていたディレクトリーを指定することが必要になる場合があります。

version

最新バージョンのプラグイン (例: 6.0.0)

2. これらのパーツをプログラムで使用可能にするには、1 つまたは複数の EGL import ステートメントをプログラム・ファイルに記述します。インポートするフ

ファイルが、コードを開発中のプロジェクト以外のプロジェクトにある場合は、作業中のプロジェクトが他のプロジェクトを参照するようにしてください。

詳細については、『インポート』を参照してください。

3. プログラムで、グローバル変数を宣言します。

- MQRC、MQRCODE、および MQVALUE を宣言します。それぞれが、変数と同じ名前を持つ dataTable パーツを typedef として使用する必要があります。
- MQSeries に渡すオプション・レコードごとに、オプション・レコード・パーツを typedef として使用する基本レコードを宣言します。各パーツの詳細については、『MQ レコード用のオプション・レコード』を参照してください。

4. 関数で、MQSeries に渡すオプション・レコードを初期化します。これを簡単に実行するには、インポートされた EGL 初期化関数を所定のオプション・レコードに対して呼び出します。それぞれの関数の名前は、レコードに対する typedef として使用されるパーツの名前の後ろに _INIT を付けた名前になります。例えば、MQGMO_INIT となります。

5. オプション・レコードに値を設定します。多くの場合、値を設定するには定数を表す EGL シンボルを割り当てます。これらの各 EGL シンボルは、MQSeries の資料に記載されたシンボルに基づきます。以下の例に示すように、個々のシンボルを連結して複数の EGL シンボルを指定することができます。

```
MQGMO.GETOPTIONS = MQGMO_LOCK  
                  + MQGMO_ACCEPT_TRUNCATED_MSG  
                  + MQGMO_BROWSE_FIRST
```

6. MQSeries サポートのカスタマイズ・フィーチャーを使用する特定プログラムを最初に生成するときに、そのプログラムによって使用されるデータ・テーブルも生成します。プログラムで使用されるすべてのデータ・テーブルを生成するには、ビルド記述子オプション **genTables** をデフォルトの YES にします。追加情報については、『DataTable パーツ』を参照してください。

MQSeries 関連の EGL キーワード

MQSeries 関連の EGL キーワード (例えば *add* や *scan*) を使用するときは、アクセス対象のそれぞれのメッセージ・キューごとに MQ レコードを定義します。レコード・レイアウトは、メッセージの形式です。

キーワードは次の表のとおりです。

キーワード	用途
add	<p>指定され たキューの末尾に MQ レコードの内容を追加する</p> <p>EGL add ステートメントでは、以下の 3 つの MQSeries コマンドが呼び出されます。</p> <ul style="list-style-type: none"> • MQCONN は、生成されたコードをキュー・マネージャーに接続する。これは、アクティブな接続が存在しない場合に呼び出されます。 • MQOPEN は、キューとの接続を確立する。これは、接続がアクティブであるがキューがまだ開いていない場合に呼び出されます。 • MQPUT は、レコードをキューに入れる。これは、それ以前の MQSeries 呼び出しでエラーが発生していない限り、常に呼び出されます。 <p>MQ レコードを追加した後は、メッセージ・キューから MQ レコードを読み取る前に、そのキューを閉じる必要があります。</p>
close	<p>MQ レコードに関連付けられたメッセージ・キューへのアクセスを解放する</p> <p>EGL close ステートメントでは、MQSeries MQCLOSE コマンドが呼び出されます。このコマンドは、プログラム終了時にも自動的に呼び出されます。</p> <p>別のプログラムがメッセージ・キューにアクセスする必要がある場合には、add または scan を実行した後でそのキューを閉じなければなりません。特に、プログラムが長い時間にわたって実行され、アクセスを必要としなくなった場合には、close を使用すると良いでしょう。</p>
scan	<p>キュー内の最初のメッセージをメッセージ・キュー・レコードの中に読み込んで、(デフォルトでは) そのメッセージをキューから除去する</p> <p>EGL scan ステートメントでは、以下の 3 つの MQSeries コマンドが呼び出されます。</p> <ul style="list-style-type: none"> • MQCONN は、生成されたコードをキュー・マネージャーに接続する。これは、アクティブな接続が存在しない場合に呼び出されます。 • MQOPEN は、キューとの接続を確立する。これは、接続がアクティブであるがキューがまだ開いていない場合に呼び出されます。 • MQGET は、レコードをキューから除去する。これは、それ以前の MQSeries 呼び出しでエラーが発生していない限り、常に呼び出されます。 <p>MQ レコードを読み取った後は、メッセージ・キューに MQ レコードを追加する前に、そのキューを閉じる必要があります。</p>

マネージャーとキューの指定

MQSeries 関連の EGL キーワードを使用するときは、以下のような状況においてキューを指定します。

- 宣言時に論理キュー名を指定します。これを行うには、MQ レコード・パーツの **queueName** プロパティを設定します。その論理キュー名は、実行時にアクセスされるキュー名のデフォルトとして機能します。ただし、多くの場合、キュー名は MQ レコードを物理キューと関連付ける手段としてのみ意味を持ちます。論理キュー名は 8 文字以内で指定できます。

- 生成時には、`buildDescriptor` パーツを使用して生成プロセスを制御します。
`buildDescriptor` パーツはリソース関連パーツを参照することができます。リソース関連パーツは、キュー名を物理キューの名前に関連付けます。
- 実行時に、レコード固有の変数 **`record.resourceAssociation`** を変更することによって、（宣言時または生成時に指定済みの）任意のキュー名をオーバーライドできます。

物理キューの名前は以下の形式になります。

queueManagerName:physicalQueueName

queueManagerName

キュー・マネージャーの名前。この名前を省略すると、コロンも省略されます。

physicalQueueName

物理キューの名前。指定したキュー・マネージャーによって認識される名前です。

あるメッセージ・キュー・レコードに対して初めて `add` または `scan` ステートメントを実行するときは、接続を行うキュー・マネージャー（デフォルトまたはそれ以外）を指定する必要があります。最も単純なケースでは、接続を行うキュー・マネージャーをまったく指定せずに、MQSeries 構成のデフォルト値を使用できます。

レコード固有の変数 **`record.resourceAssociation`** には、特定の MQ レコードについて少なくともメッセージ・キュー名が常に含まれます。

リモート・メッセージ・キュー

リモート・キュー・マネージャーによって制御されるキューにアクセスするには、以下のようにする必要があります。

- EGL `close` ステートメントを発行して、現在使用中のキューへのアクセスを解放する
- レコード固有の変数 **`record.resourceAssociation`** を設定して、リモート・キューに後でアクセスできるようにする

`record.resourceAssociation` を設定するには、キューとマネージャーの関係が MQSeries で設定されている仕方に応じて、以下の 2 つの方法のいずれかを使用します。

- 接続するキュー・マネージャーでリモート・キューがローカル定義されている場合は、**`record.resourceAssociation`** を以下のように設定します。
 - 接続するキュー・マネージャーと同じ値を受け入れる（接続するキュー・マネージャーの名前を指定するか、名前を指定しない。後者の場合はコロンを省略する。）
 - リモート・キューのローカル定義の名前を指定する。

この次に `add` または `scan` ステートメントを使用するとき、MQOPEN が発行されて、リモート・キューへのアクセスが確立されます。

- または、リモート・キュー・マネージャーの名前、およびリモート・キューの名前を使用して **`record.resourceAssociation`** を設定する。この場合、接続するキュー・マネージャーは変わりません。この次に `add` または `scan` ステートメントを使用するとき、MQOPEN が発行されて、既存の接続が使われます。

関連する概念

『直接 MQSeries 呼び出し』

285 ページの『MQSeries のサポート』

関連する参照項目

717 ページの『MQ レコードのプロパティ』

718 ページの『MQ レコード用のオプション・レコード』

直接 MQSeries 呼び出し

コードと MQSeries の間に介在する一連のインストール済み EGL 関数を使用できます。これについては、『MQSeries のサポート』に記載されています。

次の表には使用可能な関数と必要な引数が示されています。例えば、MQBACK (MQSTATE) は、MQBACK を呼び出すときに MQSTATE レコード・パーツに基づく引数を渡すことを示しています。レコード・パーツについては後述します。

MQSeries 関連の EGL 関数の呼び出し	結果
MQBACK (MQSTATE)	システム関数 sysLib.rollback を呼び出して、作業論理単位をロールバックする。このロールバックは、プログラムでアクセスされているすべての リカバリー可能リソース (データベース、メッセージ・キュー、リカバリー可能ファイルなど) に影響を与えます。
MQBEGIN (MQSTATE, MQBO)	作業論理単位を開始する。
MQCHECK_COMPLETION (MQSTATE)	MQSTATE に基づくレコードの mqdescription フィールドを設定する。この設定は最後に戻された理由コードに基づきます。関数 MQCHECK_COMPLETION は、EGL 関数 MQBEGIN、MQCLOSE、MQCONN、MQCONNEX、MQDISC、MQGET、MQINQ、MQOPEN、MQPUT、MQPUT1、および MQSET から自動的に呼び出されます。
MQCLOSE (MQSTATE)	MQSTATE.hobj が参照するメッセージ・キューを閉じる。
MQCMIT (MQSTATE)	システム関数 sysLib.commit を呼び出して、作業論理単位をコミットする。このコミットは、プログラムでアクセスされているすべての リカバリー可能リソース (データベース、メッセージ・キュー、リカバリー可能ファイルなど) に影響を与えます。
MQCONN (MQSTATE, qManagerName)	キュー・マネージャーに接続する。キュー・マネージャーは qManagerName (最大 48 文字のストリング) によって示されます。MQSeries は、以降の呼び出しで使用するための接続ハンドル (MQSTATE.hconn) を設定します。 注: コードで一度に接続できるキュー・マネージャーの数は 1 つです。

MQSeries 関連の EGL 関数の呼び出し	結果
MQCONNX(MQSTATE, qManagerName, MQCNO)	呼び出し方法を制御するオプションを使用して、キュー・マネージャーに接続する。キュー・マネージャーは qManagerName (最大 48 文字のストリング) によって示されます。MQSeries は、以降の呼び出しで使用するための接続ハンドル (MQSTATE.hconn) を設定します。
MQDISC (MQSTATE)	キュー・マネージャーから切断する。
MQGET(MQSTATE, MQMD, MQGMO, BUFFER)	メッセージをキューから読み取って除去する。バッファは 32767 バイトを超えてはなりませんが、EGL の get next ステートメントを使用している場合にはこの制限は適用されません。
MQINQ(MQSTATE, MQATTRIBUTES)	キューの属性を要求する。
MQNOOP()	EGL によってのみ使用される。
MQOPEN(MQSTATE, MQOD)	メッセージ・キューを開く。MQSeries は、以降の呼び出しで使用するためのキュー・ハンドル (MQSTATE.hobj) を設定します。
MQPUT(MQSTATE, MQMD, MQPMO, BUFFER)	メッセージをキューに追加する。バッファは 32767 バイトを超えてはなりませんが、EGL の add ステートメントを使用している場合にはこの制限は適用されません。
MQPUT1(MQSTATE, MQOD, MQMD, MQPMO, BUFFER)	キューを開き、1 つのメッセージを書き込んで、キューを閉じる。
MQSET(MQSTATE, MQATTRIBUTES)	キューの属性を設定する。

次の表には、MQSeries 関連の EGL 関数を呼び出したときに引数として使用されるオプション・レコードがリストされています。また、所定の引数に対して呼び出す必要のある初期化関数も併せてリストされています。

最初のステップは、MQSTATE レコード・パーツに基づく引数を初期化することです。以下の例では (その下の表に示すように)、引数の名前はレコード・パーツ名と同じであると想定されています。

```
MQSTATE_INIT(MQSTATE);
```

引数 (レコード・パーツ名)	初期化関数	説明
MQATTRIBUTES	なし	属性の配列と属性のセクター、およびコマンド MQINQ または MQSET で使用されるその他の情報
MQBO	MQBO_INIT (MQBO)	開始オプション
MQCNO	MQCNO_INIT (MQCNO)	接続オプション
MQGMO	MQGMO_INIT (MQGMO)	メッセージ取得オプション

引数 (レコード・パーツ名)	初期化関数	説明
MQIIH	MQIIH_INIT (MQIIH)	IMS™ 情報ヘッダー。IMS に送信される MQSeries メッセージの先頭に必要な情報を記述します (MQSeries の資料には、このヘッダーの使用は Windows 2000/NT/XP ではサポートされないと記述されています)。
MQINTATTRS	なし	整数属性の配列 (MQINQ または MQSET コマンド用)
MQMD	MQMD_INIT (MQMD, MQSTATE)	メッセージ記述子 (MQSeries バージョン 2)
MQMDE	MQMDE_INIT (MQMDE, MQSTATE)	メッセージ記述子拡張 (MQSeries バージョン 2 のフィールドのみを使用)
MQOD	MQOD_INIT (MQOD)	オブジェクト記述子
MQOO	MQOO_INIT (MQOO)	Open オプション
MQPMO	MQPMO_INIT (MQPMO)	メッセージ書き込みオプション
MQSELECTORS	なし	属性セクターの配列。EGL 関数を使用せずに MQSeries にアクセスする場合のみ使用します。
MQSTATE	MQSTATE_INIT (MQSTATE)	MQSeries への 1 つ以上の呼び出しでそれぞれ使用される引数の集合。例えば、EGL 関数 MQCONN または MQCONNx と接続すると、MQSeries は以降の呼び出しで使用するための接続ハンドル (MQSTATE.hconn) を設定します。
MQXQH	MQXQH_INIT (MQXQH, MQSTATE)	伝送キュー・ヘッダー

注: レコード・パーツはそれぞれが 1 つの構造体項目のみを含んでおり、構造体項目は構造体パーツを `typeDef` として使用します。このセットアップによって最大限の柔軟性が得られます。それぞれが一連の構造体パーツから成る独自のレコード・パーツを作成できます。

各構造体パーツ名はレコード・パーツ名の後ろに `_S` を付けた名前になります。例えば、レコード・パーツ `MQGMO` の場合は、`MQGMO_S` という名前の構造体パーツを使用します。

関連する概念

289 ページの『MQSeries 関連の EGL キーワード』

285 ページの『MQSeries のサポート』

141 ページの『レコード・パーツ』

30 ページの『Typedef』

関連する参照項目

644 ページの『get next』

956 ページの『commit()』

969 ページの『rollback()』

EGL コードの保守

EGL ソース・コード行のコメント化

1 行のコードをコメント化するには、次のようにします。

1. その行をクリックしてから、右クリックする。コンテキスト・メニューが表示されます。
2. 「コメント」を選択する。コメント標識 (*//*) が行の先頭に置かれます。

連続した複数行のコードをコメント化するには、次のようにします。

1. 先頭行をクリックする。左マウス・ボタンを押したまま、カーソルを終了行までドラッグします。マウス・ボタンを放すと、行の範囲が強調表示されます。
2. 右クリックし、コンテキスト・メニューから「コメント」を選択する。コメント標識 (*//*) が、選択した範囲の各行の先頭に置かれます。

行のコメントを解除するには、上記と同じ手順を使用します。ただし、コンテキスト・メニューから「コメント解除」を選択してください。

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

299 ページの『.egl ファイル内のパーツのオープン』

関連する参照項目

523 ページの『EGL エディター』

パーツの検索

EGL エディターでファイルを開いてある場合、検索基準を設定した後でパーツを検索できます。

1. EGL ファイルを開く。EGL エディターがアクティブでなければ、検索機能を使用できません。ただし、検索先はエディターで開かれているファイルだけに限られるわけではありません。
2. ワークベンチ・メニューで「検索」>「EGL」をクリックする。「検索」ダイアログが表示されます。
3. 「EGL 検索」タブがまだ表示されていない場合は、「EGL 検索」をクリックする。「検索」タブ全体で指定した条件が、結果に影響を及ぼす場合があることに注意してください。
4. 見つけたいパーツ名を入力する。あるいは、特定の文字パターンと一致する名前を含むパーツのリストを表示するには、名前に次のようなワイルドカード・シンボルを埋め込みます。
 - 疑問符 (?) は、任意の 1 文字を表します。
 - アスタリスク (*) は任意の一連の文字を表します。

たとえば、*myForm?Group* を入力すると、*myForm1Group* と *myForm2Group* という名前のパーツが見つかりますが、*myForm10Group* は見つかりません。

`myForm*Group` を入力すると、`myForm1Group`、`myForm2Group`、および `myForm10Group` という名前のパーツが見つかります。

5. 検索で大/小文字を区別する (`myFormGroup` と `MYFORMGROUP` を区別する) には、チェック・ボックスをクリックする。
6. 「検索 (Search For)」ボックスで、パーツの型を選択するか、「**任意のエレメント**」を選択して検索をすべてのパーツ型まで拡張する。
7. 「制限」ボックスで、検索をパーツ宣言かパーツ参照、またはその両方に制限するオプションを選択する。
8. 「有効範囲」ボックスで、ワークスペース内を検索するには「**ワークスペース (Workspace)**」、プロジェクト・エクスプローラーで現在、強調表示されているプロジェクト内を検索するには「**エンクロージング・プロジェクト**」、プロジェクトの定義済みプロジェクト・セット内を検索するには「**ワーキング・セット**」を選択する。「ワーキング・セット」の有効範囲を選択した場合は、「**選択**」ボタンをクリックして、既存のワーキング・セットを選択するか、新規ワーキング・セットを定義します。
9. 「**検索**」ボタンをクリックする。検索結果が「検索」ビューに表示される。
10. 「検索」ビューでファイルをダブルクリックすると、そのファイルが EGL エディターで開かれ、一致するパーツが強調表示される。ファイル内に複数の一致がある場合は、最初の一致が強調表示されます。

エディターの左マージンにある矢印は、一致するそれぞれのパーツのロケーションを示しています。

関連する概念

19 ページの『パーツ』

関連するタスク

299 ページの『.egl ファイル内のパーツのオープン』

関連する参照項目

523 ページの『EGL エディター』

パーツ参照の表示

プログラム、ライブラリー、ページ・ハンドラー、またはレポート・ハンドラー・パーツの中で参照されている EGL パーツの階層図を表示したり、それらのパーツにアクセスしたりすることができます。

1. 次に示す 2 つの方法のどちらかによって、パーツ参照ビューを開く。
 - プロジェクト・エクスプローラーで、プログラム、ライブラリー、ページ・ハンドラー、またはレポート・ハンドラー・パーツが入っている EGL ファイルを右クリックする。「**パーツ参照で開く**」を選択します。
 - 別の方法として、EGL エディターで次のようにして EGL ファイルを開く。
 - a. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「**ビューの表示**」>「**アウトライン**」を選択して、そのビューを開く。
 - b. 「アウトライン」ビューで、ファイルを右クリックしてから、「**パーツ参照で開く**」をクリックする。

2. プログラム、ライブラリー、ページ・ハンドラー、またはレポート・ハンドラー・パーツは、階層のトップレベルにある。参照されている各パーツは、その階層内のサブ項目です。ビューには、それぞれのパーツごとに、パラメーター、データ宣言、使用宣言、および関数が必要に応じて表示されます。
3. パーツをダブルクリックする。関連するソース・ファイルが EGL エディターで開かれ、パーツ名が強調表示されます。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

19 ページの『パーツ』

関連するタスク

300 ページの『プロジェクト・エクスプローラーでのEGL ソース・ファイルの位置決め』

『.egl ファイル内のパーツのオープン』

関連する参照項目

523 ページの『EGL エディター』

.egl ファイル内のパーツのオープン

わずかなキー・ストロークによって、ワークスペース内の任意の場所にある、ビルド・パーツ以外の EGL パーツにアクセスできます。

1. ワークベンチで、「ナビゲート (Navigate)」 > 「パーツを開く」をクリックするか、ツールバーの「パーツを開く」ボタンをクリックする。「パーツを開く」ダイアログが表示されます。
2. 見つけたいパーツ名を入力する。あるいは、特定の文字パターンと一致する名前を含むパーツのリストを表示するには、名前に次のようなワイルドカード・シンボルを埋め込みます。
 - 疑問符 (?) は、任意の 1 文字を表します。
 - アスタリスク (*) は任意の一連の文字を表します。

たとえば、*myForm?Group* を入力すると、*myForm1Group* と *myForm2Group* という名前のパーツが見つかりますが、*myForm10Group* は見つかりません。

*myForm*Group* を入力すると、*myForm1Group*、*myForm2Group*、および *myForm10Group* という名前のパーツが見つかります。

名前を入力すると、「パーツを開く」ダイアログの「パーツのマッチング」セクションに、修飾するパーツが表示されます。

3. パーツのリストから、オープンしたいパーツを選択する。ダイアログの「修飾子」セクションに、選択したパーツを保持するフォルダー、プロジェクト、パッケージ、およびソース・ファイルが入っているパスが表示されます。複数のパーツに同じ名前が付いている場合は、オープンしたいファイルのパスをクリックすることにより、特定のパーツを選択します。
4. 「OK」をクリックする。選択したパーツが含まれているソース・ファイルが EGL エディターで開かれ、そのパーツ名が強調表示されます。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

19 ページの『パーツ』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

『プロジェクト・エクスプローラーでの EGL ソース・ファイルの 位置決め』

関連する参照項目

523 ページの『EGL エディター』

プロジェクト・エクスプローラーでの EGL ソース・ファイルの 位置決め

EGL ソース・ファイルを編集する場合は、「プロジェクト・エクスプローラー」ビューでファイルをすばやく見つけることができます。「プロジェクト・エクスプローラーで表示」コンテキスト・メニュー・オプションは、次のことを行います。

- 「プロジェクト・エクスプローラー」ビューを開く (まだ開いていない場合)。
- ソース・ファイルを見つめる場合に必要となるプロジェクト・エクスプローラーのツリー・ノードを展開する。
- ソース・ファイルを強調表示する。

プロジェクト・エクスプローラーで EGL ソース・ファイルを見つめるには、次のようにします。

1. 開いている EGL ソース・ファイルのエディター領域内を右クリックする。コンテキスト・メニューが表示されます。
2. コンテキスト・メニューから「プロジェクト・エクスプローラーで表示」を選択する。

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

299 ページの『.egl ファイル内のパーツのオープン』

関連する参照項目

523 ページの『EGL エディター』

プロジェクト・エクスプローラーでの EGL ファイルの削除

プロジェクト・エクスプローラーで EGL ファイルを削除するには、次のことを行います。

1. EGL ファイルをクリックし、Delete (削除) キーを押す。または、EGL ファイルを右クリックし、コンテキスト・メニューが表示されたならば、「削除」を選択します。
2. ファイルを削除したいことの確認が求められます。ファイルを削除するには「はい」をクリックし、削除をキャンセルするには「いいえ」をクリックしてください。

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

300 ページの『プロジェクト・エクスプローラーでのEGL ソース・ファイルの位置決め』

EGL コードのデバッグ

EGL デバッガー

ワークベンチで作業している場合、EGL デバッガーによって、最初に出力を生成する必要なく、EGL コードをデバッグすることができます。これらのカテゴリが有効になります。

- ページ・ハンドラーを、J2EE コンテキストで使用されたプログラムとともにデバッグするには、ローカルな WebSphere Application Server テスト環境をデバッグ・モードで使用することができます。
 - Web アプリケーション内の J2EE の下で実行されているすべてのコードにその環境を使用する必要があります。
 - J2EE の下のバッチ・アプリケーションで実行されるプログラムにその環境を使用できます。
- その他のコード (J2EE の下で実行されないバッチ・アプリケーション、またはテキスト・アプリケーション) をデバッグするには、WebSphere テスト環境外部の起動構成を使用します。この場合、キー・ストロークを少し用いることでデバッグ・セッションを開始できます。

J2EE コンテキストでデプロイする対象のバッチ・プログラム上で作業している場合、起動構成を使用して、J2EE 以外のコンテキストのプログラムをデバッグすることができます。セットアップは簡単になりますが、以下のいくつかの値を調整する必要があります。

- 起動構成を使用する場合は、ビルド記述子オプション J2EE の値を NO に設定する必要があります。
- また、Java プロパティ値を、リレーショナル・データベースにアクセスする際の差に順応するように調整する必要があります。
 - J2EE の場合は、JNDI レジストリー内でデータ・ソースが結合されている先の名前である、*jdbc/MyDB* のようなストリングを指定します。このストリングは、以下のいずれかの方法で指定します。
 - ビルド記述子オプション *sqlJNDIName* を設定する
 - 「EGL SQL Database Connections preference (EGL SQL データベース接続設定)」ページの「接続 JNDI 名」フィールドに、値を設定する。詳しくは、『SQL データベース接続の設定』を参照してください。
 - J2EE 以外の場合は、*jdbc:db2:MyDB* のような接続 URL を指定します。このストリングは、以下のいずれかの方法で指定します。
 - ビルド記述子オプション *sqlIDB* を設定する
 - 「EGL SQL Database Connections preference (EGL SQL データベース接続設定)」ページの「Connection URL (接続 URL)」フィールドに、値を設定する。詳しくは、『SQL データベース接続の設定』を参照してください。

これ以降のセクションでは、ビルド記述子および EGL 設定の相互作用について説明します。

デバッガー・コマンド

EGL デバッガーと相互作用するには、以下のコマンドを使用します。

ブレークポイントを追加

処理が休止する行を識別します。コード実行が休止する場合は、可変値とともにファイルおよび画面の状況を調べることができます。

ブレークポイントは、除去されない限り、デバッグ・セッションから次のデバッグ・セッションに渡って記憶されます。

ブレークポイントを、ブランク行またはコメント行に設定することはできません。

ブレークポイントを使用不可にする

ブレークポイントを、非活動化しますが除去はしません。

ブレークポイントを使用可能にする

以前に使用不可にされたブレークポイントを活動化します。

ブレークポイントを除去

処理が行で自動的に休止しないように、ブレークポイントをクリアします。

Remove all breakpoints (すべてのブレークポイントを除去)

すべてのブレークポイントをクリアします。

実行

コードの実行を、次のブレークポイントまで、または実行単位が終了するまで行います。(いずれにしても、デバッガーは `main` 関数の最初の文で停止します。)

指定行まで実行

すべての文を、指定した行の文まで (その文は含まない) 実行します。

ステップイントゥ

次の EGL ステートメントを実行して、休止します。

以下に、特定の文タイプに **step into** コマンドを実行した場合の動作をリストします。

call

呼び出し先プログラムが EGL デバッガー内部で実行中の場合は、呼び出し先プログラムの最初の文で停止します。呼び出し先プログラムが EGL デバッガー外部で実行中の場合は、現在のプログラムの次の文で停止します。

EGL デバッガーはワークベンチ内のすべてのプロジェクト内で受信プログラムを検索します。

converse

ユーザー入力を待ちます。この入力によって、バリデーター関数に含まれる可能性のある次の実行中文で処理が停止します。

forward

コードがページ・ハンドラーに転送されると、デバッガーはユーザー入力を待ち、バリデーター関数に含まれる可能性のある次の実行文で停止します。

コードがプログラムに転送されると、デバッガーはそのプログラム内の最初の文で停止します。

function invocation

関数の最初の文で停止します。

JavaLib.invoke およびそれに関連した関数

次の Java ステートメントで停止するため、Java アクセス機能によって使用可能にされた Java コードをデバッグできます。

show、transfer

制御を受け取るプログラムの最初の文で停止します。ターゲット・プログラムは、EGL デバッガーで稼働する EGL ソースであり、EGL 生成コードではありません。

transfer to a transaction 書式の **show** ステートメントまたは **transfer** ステートメントのいずれかの後、EGL デバッガーは新規プログラム用のビルド記述子に切り替えるか、または (ビルド記述子を使用されていない場合は) 新規ビルド記述子の作成プロンプトを出します。新規プログラムは、以前に実行されたプログラムとは異なるプロパティ・セットを持つことができます。

EGL デバッガーはワークベンチ内のすべてのプロジェクト内で受信プログラムを検索します。

ステップオーバー

次の EGL ステートメントを実行して休止しますが、現行の関数から呼び出された関数の内部では停止しません。

以下に、特定の文タイプに **step over** コマンドを実行した場合の動作をリストします。

converse

ユーザー入力を待ってから、検証機能をスキップします (ブレークポイントが有効でない場合)。**converse** ステートメントの後の文で停止します。

forward

コードがページ・ハンドラーに転送されると、デバッガーはユーザー入力を待ち、次の実行中文で停止しますが、これはバリデータ関数には含まれません (ブレークポイントが有効でない場合)。

コードがプログラムに転送されると、デバッガーはそのプログラム内の最初の文で停止します。

show、transfer

制御を受け取るプログラムの最初の文で停止します。ターゲット・プログラムは、EGL デバッガーで稼働する EGL ソースであり、EGL 生成コードではありません。

transfer to a transaction 書式の **show** ステートメントまたは **transfer** ステートメントのいずれかの後、EGL デバッガーは新規プログラム用のビルド記述子に切り替えるか、または (ビルド記述子を使用されていない場合は) 新規ビルド記述子の作成プロンプトを出します。新規プログラムは、以前に実行されたプログラムとは異なるプロパティ・セットを持つことができます。

EGL デバッガーはワークベンチ内のすべてのプロジェクト内で受信プログラムを検索します。

ステップ・リターン

呼び出し側プログラムまたは関数に戻る必要のある文を実行します。それから、そのプログラムまたは関数内の制御を受け取る文で休止します。

バリデーター関数で **step return** コマンドを実行する場合、例外が有効になります。その場合、振る舞いは **step into** コマンドの振る舞いと同一になります。つまり、主に EGL デバッガーが次の文を実行して休止することを意味します。

EGL デバッガーは、以下の EGL ステートメントを NULL 演算子であるかのように扱います。

- **sysLib.audit**
- **sysLib.purge**
- **sysLib.startTransaction**

例えば、これらの文にブレークポイントを追加できますが、**step into** コマンドは後続の文に続けて実行されるだけで、他の文は影響を受けません。

最後に、関数で実行されている最後の文に **step into** または **step over** のコマンドを実行する（および、その文が **return**、**exit program**、または **exit stack** でない）と、関数にローカルな変数を検討できるように、処理は関数内で自ら休止します。この場合、デバッグ・セッションを継続するには、別のコマンドを実行します。

ビルド記述子の使用

ビルド記述子を使用すると、デバッグ環境のアスペクトを判別できます。EGL デバッガーは、以下の規則に従ったビルド記述子を選択します。

- デバッグ・ビルド記述子をプログラムまたはページ・ハンドラーに指定した場合、EGL デバッガーはそのビルド記述子を使用します。デバッグ・ビルド記述子の設定方法についての詳細は、『デフォルトのビルド記述子の設定』を参照してください。
- デバッグ・ビルド記述子を指定しなかった場合、EGL デバッガーは、ビルド記述子のリストから選択するか、または値 **None** を受け入れることを促すプロンプトを出します。値 **None** を受け入れると、EGL デバッガーはデバッグ・セッションで使用するためのビルド記述子を構成します。VisualAge Generator との互換性が有効であるかどうかは、設定によって決まります。
- **None** またはいくつかのデータベース接続情報が不足しているビルド記述子のいずれかを指定した場合、EGL デバッガーは設定を検討することによって接続情報を入手します。これらの設定方法についての詳細は、『SQL データベース接続設定の変更』を参照してください。

Java 環境において、テキスト・アプリケーションまたはバッチ・アプリケーションでの使用が意図されたプログラムをデバッグする場合、およびそのプログラムが Java 環境における異なる実行単位で使用が意図されたプログラムに制御を切り替える **transfer** ステートメントを実行する場合、EGL デバッガーは、受信プログラムに割り当てられたビルド記述子を使用します。ビルド記述子の選択項目は、以前に説明した規則に基づきます。

別のプログラムによって呼び出されたプログラムをデバッグする場合、EGL デバッガーは呼び出し先プログラムに割り当てられたビルド記述子を使用します。ビルド記述子の選択項目は、上記に説明した規則に基づきます。ただし、ビルド記述子を指定していない場合、呼び出し先プログラムが呼び出されるときに、デバッガーはビルド記述子のプロンプトを出しません。その代わり、呼び出し中プログラム用のビルド記述子が使用中のまま残されます。

注: これらのプログラムのいずれか (両方ではない) が VisualAge Generator との互換性を利用している場合は、呼び出し側および呼び出し先のプログラム用に異なるビルド記述子を使用する必要があります。VisualAge 互換性の生成時間状況は、ビルド記述子オプション **VAGCompatibility** の値によって判別されます。

コードのデバッグに使用するビルド記述子またはリソース関連パーツは、コードの生成に使用するものとは異なる可能性があります。

SQL データベース・アクセス

SQL データベースへのアクセスに使用するユーザー ID およびパスワードを判別するために、EGL デバッガーは、情報が検出されるか、またはすべてのソースが考慮されるまで、以下のソースを順番に考慮します。

1. デバッグ時に使用するビルド記述子。特に、ビルド記述子オプション **sqlID** および **sqlPassword**。
2. 『SQL データベース接続設定の変更』に説明のある SQL 設定ページ。このページでは、他の接続情報も指定します。
3. 接続時間に表示される対話式ダイアログ。このダイアログは、「**Prompt for SQL user ID and password when needed (SQL ユーザー ID およびパスワードが必要な場合はプロンプト表示する)**」チェック・ボックスを選択した場合にのみ表示されます。

call ステートメント

以前にも記述したように、EGL デバッガーは、EGL ソース・コードを解釈することによって、**transfer** または **show** ステートメントに応答します。ただし、EGL デバッガーは、**call** ステートメントに対しては、ビルド記述子に指定したリンケージ・オプション・パーツ (ある場合) を検討することによって応答します。参照したリンケージ・オプション・パーツに、呼び出し用の **callLink** エレメントが含まれる場合、結果は次のようになります。

- **callLink** プロパティ **remoteComType** を **DEBUG** に設定する場合、EGL デバッガーは EGL ソース・コードを解釈します。デバッガーは、**callLink** プロパティ **package** および **location** を参照することによって、ソースを見つけます。
- **callLink** プロパティ **remoteComType** を **DEBUG** に設定しない場合、デバッガーは、EGL 生成 Java プログラムを実行しているかのように EGL 生成コードを呼び出して、リンケージ・オプション・パーツ内の情報を使用します。

リンケージ情報がない場合、EGL デバッガーは EGL ソース・コードを解釈することによって **call** ステートメントに応答します。リンケージ情報は、以下の場合には使用できません。

- ビルド記述子を使用されていない。
- ビルド記述子を使用されているが、そのビルド記述子にリンケージ・オプション・パーツが指定されていない。
- ビルド記述子にリンケージ・オプション・パーツが指定されているが、参照されたパーツに、呼び出し先プログラムを参照する **callLink** エレメントがない。

デバッガーが EGL ソース・コードを実行する場合、呼び出し側から **step into** コマンドを実行することによって、そのプログラム内で文を実行できます。ただし、

デバッガーが生成済みコードを呼び出す場合、デバッガーはプログラム全体を実行して、**step into** コマンドは **step over** コマンドのように機能します。

デバッグ時に使用されるシステム・タイプ

システム・タイプの値は、`sysVar.systemType` で使用できます。また、2 番目の値は、VisualAge Generator との開発時間互換性を要求した場合に、`VGLib.getVAGSysType` で使用できます。

`sysLib.systemType` の値は、ビルド記述子オプション **system** の値と同じです。ただし、値が以下の 2 つのいずれかの場合で **DEBUG** に設定されている場合は異なります。

- ・『EGL デバッガーの設定の変更』の説明にあるように「**Set systemType to DEBUG (systemType を DEBUG に設定)**」設定を選択する場合。
- ・その設定の値に関わらず、デバッグ・セッション中に使用するためのビルド記述子として **NONE** を指定した場合。

システム関数 `VGLib.getVAGSysType` は、VisualAge Generator に相当する `sysLib.systemType` の値を返します。詳細については、『`VGLib.getVAGSysType`』のテーブルを参照してください。

EGL デバッガー・ポート

EGL デバッガーは、Eclipse ワークベンチとの通信を確立するためにポートを使用します。デフォルト・ポート番号は 8345 です。別のアプリケーションがこのポートを使用している場合やファイアウォールによってポートがブロックされている場合は、『EGL デバッガーの設定の変更』の説明に従い別の値を設定してください。

EGL デバッガー・ポートとして 8345 以外の値を指定した場合、J2EE サーバー上で EGL プログラムをデバッグするときは、以下のようにサーバー構成を編集する必要があります。

1. 「Environment (環境)」タブの「System Properties (システム・プロパティ)」セクションに進みます。
2. 「追加」をクリックします。
3. 「名前」に `com.ibm.debug.egl.port` と入力します。
4. 「値」に新規のポート番号を入力します。

推奨

EGL デバッガーを処理する準備ができれば、以下の推奨条件を考慮してください (ほとんどの条件では、コードのデバッグ時に `sysVar.systemType` が **DEBUG** に設定されていることを前提としています)。

- ・データベースから日付を検索していて、ランタイム・コードがその日付を ISO 形式以外の形式で受け取るようにする場合、日付を変換するための関数を書き込みます。ただし、この関数は、システム・タイプが **DEBUG** である場合にのみ呼び出すようにしてください。ISO 形式は `yyyy-mm-dd` で、デバッガーが使用できる唯一の形式です。
- ・デバッガーの実行時に使用するための外部 Java クラスを指定するには、『EGL デバッガーの設定の変更』の説明を参照して、クラスパスを変更します。例え

ば、MQSeries、JDBC ドライバー、または Java アクセス関数をサポートするには、追加のクラスが必要となる場合があります。

- (別の EGL 関数でなく) JSF によって呼び出されたページ・ハンドラー関数をデバッグする場合に、関数から抜けるには、「ステップオーバー」、「ステップイントゥ」、または「ステップ・リターン」でなく、「実行」を使用します。3 つのステップ・コマンドのいずれを使用しても、ページ・ハンドラーの生成済み Java コードが表示されます。この機能は EGL をデバッグする場合には不便です。ステップ・コマンドのいずれかを使用した場合に、生成 Java コードを離れて、ブラウザーに Web ページを表示するには、「実行」を使用します。
- SQL オプション WITH HOLD (または EGL 相当) を使用している場合、オプション WITH HOLD は、EGL 生成 Java 用または EGL デバッガー内で使用できないことを認識しておく必要があります。ランタイムにのみ呼び出される条件文内に commit ステートメントを配置することによって、部分的にこの制限を処理することができる場合があります。以下に例を示します。

```
if (systemType not debug)
    sysLib.commit();
end
```

EGL プログラムを J2EE サーバーでデバッグする場合、または EGL リスナーを使用してデバッグする場合は、EGL デバッガー・ポートの個数を指定するようにサーバーまたは EGL リスナーを構成する必要があります。

- J2EE サーバーを構成するには、サーバー構成を編集します。
 1. 「Environment (環境)」タブの「System Properties (システム・プロパティ)」セクションに進みます。
 2. 「追加」をクリックします。
 3. 「名前」に *com.ibm.debug.egl.port* と入力します。
 4. 「値」に新規ポート番号を入力します。
- EGL リスナーを構成するには、EGL リスナー起動構成を編集します。
 1. 「Arguments (引数)」タブに進みます。
 2. 「VM Arguments (VM 引数)」フィールドに、次のように入力します。

```
-Dcom.ibm.debug.egl.port=portNumber
```

portNumber

新規ポート番号

関連する概念

477 ページの『VisualAge Generator との互換性』

121 ページの『EGL デバッガーの文字エンコード・オプション』

285 ページの『VSAM サポート』

関連するタスク

126 ページの『SQL データベース接続設定の変更』

120 ページの『EGL デバッガーの設定の変更』

122 ページの『デフォルトのビルド記述子の設定』

関連する参照項目

456 ページの『callLink エLEMENTの remoteComType』

435 ページの『sqlDB』

436 ページの『sqlID』
437 ページの『sqlJNDIName』
438 ページの『sqlPassword』
983 ページの『getVAGSysType()』
1002 ページの『systemType』

J2EE 以外のアプリケーションのデバッグ

EGL デバッガーでの 非 J2EE アプリケーションの開始

EGL デバッグ・セッションでの EGL テキスト・プログラムまたは非 J2EE 基本プログラムのデバッグを開始するには、起動構成が必要です。起動構成は、プログラムのファイルのロケーションを定義し、プログラムの起動方法を指定します。EGL アプリケーションで起動構成を作成する (暗黙的作成) か、またはユーザーが作成 (『EGL デバッガーでの起動構成の作成』を参照) します。

暗黙的に作成された起動構成を使用してプログラムを起動するには、次のようにします。

1. 「プロジェクト・エクスプローラー」ビューで、起動したい EGL ソース・ファイルを右クリックする。また、EGL ソース・ファイルが EGL エディターで開いている場合は、「アウトライン」ビューでプログラムを右クリックします。
2. コンテキスト・メニューが表示されます。
3. 「EGL プログラムのデバッグ」をクリックする。起動構成が作成され、EGL デバッガーでプログラムが起動します。

暗黙的に作成された起動構成を表示するには、次のようにします。

1. ツールバーの「デバッグ」ボタンの横にある矢印をクリックする。コンテキスト・メニューが表示されます。
2. 「デバッグ」をクリックする。「デバッグ」ダイアログが表示されます。
「Name (名前)」フィールドに起動構成の名前が表示されます。暗黙的に作成された起動構成は、プロジェクトおよびソース・ファイル名に従って名前が付けられます。

注: 「デバッグ」ダイアログは、「Run (実行)」メニューの「デバッグ」をクリックしても表示されます。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

『EGL デバッガーでの起動構成の作成』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

EGL デバッガーでの起動構成の作成

EGL デバッグ・セッションでの EGL テキスト・プログラムまたは非 J2EE 基本プログラムのデバッグを開始するには、起動構成が必要です。起動構成は、プログラ

ムの起動方法を指定します。起動構成は、ユーザーが作成する (明示的作成) か、または EGL アプリケーションで作成 (『EGL デバッガーでの 非 J2EE プログラムの開始』を参照) します。

明示的に作成された起動構成を使用してプログラムを開始するには、次のようにします。

1. ツールバーの「デバッグ」ボタンの横にある矢印をクリックした後、「デバッグ」をクリックするか、または「Run (実行)」メニューで「デバッグ」を選択する。
2. 「デバッグ (Debug)」ダイアログが表示される。
3. 構成リストの「EGL プログラム」をクリックして、「新規」をクリックする。
4. 「プロジェクト・エクスプローラー」ビューで EGL ソース・ファイルを強調表示しなかった場合、起動構成の名前は *New_configuration* になる。「プロジェクト・エクスプローラー」ビューで EGL ソース・ファイルを強調表示した場合、起動構成の名前は、EGL ソース・ファイルと同じになります。起動構成の名前を変更したい場合は、「Name (名前)」フィールドに新しい名前を入力します。
5. 「Load (ロード)」タブの「Project (プロジェクト)」フィールド内の名前が正しくない場合は、「参照」をクリックする。プロジェクトのリストが表示されます。プロジェクトをクリックして、「OK」をクリックします。
6. 「EGL program source file (EGL プログラムのソース・ファイル)」フィールド内の名前が正しくないか、またはこのフィールドが空の場合は、「Search (検索)」をクリックする。EGL ソース・ファイルのリストが表示されます。ソース・ファイルをクリックして、「OK」をクリックします。
7. 「デバッグ」ダイアログのフィールドを変更した場合は、「Apply (適用)」をクリックして、起動構成設定を保管する。
8. 「デバッグ」をクリックして、EGL デバッガー内でプログラムを起動する。

注: 「Apply (適用)」を使用して起動構成設定を保管していない場合は、「Revert (戻り)」をクリックすると、すべての変更が除去されます。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

310 ページの『EGL デバッガーでの 非 J2EE アプリケーションの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

EGL リスナー起動構成の作成

EGL 生成 Java アプリケーションまたはラッパーから呼び出される非 J2EE EGL アプリケーションをデバッグするには、EGL リスナー起動構成が必要です。EGL リスナー起動構成を作成するには、次のようにします。

1. ツールバーの「デバッグ」ボタンの横にある矢印をクリックした後、「デバッグ」をクリックするか、または「Run (実行)」メニューで「デバッグ」を選択する。
2. 「デバッグ (Debug)」ダイアログが表示される。

3. 構成リスト内の「EGL リスナー」をクリックして、「新規」をクリックする。
4. リスナー起動構成に、*New_configuration* という名前が付く。起動構成の名前を変更したい場合は、「Name (名前)」フィールドに新しい名前を入力します。
5. ポート番号を入力しなければ、ポートはデフォルトの 8346 になる。それ以外の場合は、ポート番号を入力してください。それぞれの EGL リスナーごとに、独自のポートが必要です。
6. 「適用 (Apply)」をクリックして、リスナー起動構成を保管する。
7. 「デバッグ (Debug)」をクリックして、EGL リスナーを起動する。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

310 ページの『EGL デバッガーでの起動構成の作成』

310 ページの『EGL デバッガーでの 非 J2EE アプリケーションの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

J2EE アプリケーションのデバッグ

EGL Web デバッグのためのサーバーの準備

WebSphere Application Server で実行される EGL Web プログラムのデバッグを行う場合は、デバッグを行うためにサーバーを準備する必要があります。準備ステップは、1 サーバーに 1 回だけ行う必要があります。ワークベンチがシャットダウンされた場合でも、このステップを再度行う必要はありません。

デバッグを行うためにサーバーを準備するには、次のようにします。

1. WebSphere v5.1 テスト環境で作業をしている場合は、サーバーが停止していることを確認する。WebSphere Application Server v6.0 で作業をしている場合は、サーバーが稼働していることを確認します。この違いは、v6.0 コードが機能するサーバーである点にあります。
2. 「サーバー (Server)」ビューで、サーバーを右クリックする。コンテキスト・メニューが表示されます。
3. 「EGL デバッグを使用可能/使用不可にする」を選択する。メッセージにより、EGL デバッグが使用可能にされたことが示されます。
4. EGL でなく、生成された Java をデバッグしたい場合は、サーバーを再度右クリックし、「EGL デバッグを使用可能/使用不可にする」を選択する。メッセージにより、EGL デバッグが使用不可にされたことが示されます。

関連する概念

303 ページの『EGL デバッガー』

371 ページの『WebSphere Application Server と EGL』

199 ページの『Web サポート』

関連するタスク

313 ページの『EGL Web デバッグ・セッションの開始』

『EGL Web デバッグのためのサーバーの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

EGL Web デバッグのためのサーバーの開始

JNDI データ・ソースにアクセスする EGL ベースの Web アプリケーションで作業する場合は、現行のトピックの説明に従うことはできません。ただし、以前に Web アプリケーション・サーバーを構成してある場合は除きます。WebSphere 固有の背景情報については、『*WebSphere Application Server* と *EGL*』を参照してください。

また、EGL Web プログラムをデバッグしたい場合は、『*EGL Web デバッグのためのサーバーの準備*』で説明されているように、サーバーをその目的で準備する必要があります。

サーバーをデバッグ用に始動するには、次のようにします。

1. 「サーバー (Server)」ビューで、サーバーを右クリックする。
2. 「デバッグ (Debug)」>「サーバーでデバッグ」を選択する。

関連する概念

303 ページの『EGL デバッガー』

371 ページの『WebSphere Application Server と EGL』

199 ページの『Web サポート』

関連するタスク

312 ページの『EGL Web デバッグのためのサーバーの準備』

『EGL Web デバッグ・セッションの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

EGL Web デバッグ・セッションの開始

JNDI データ・ソースにアクセスする EGL ベースの Web アプリケーションで作業する場合は、現行のトピックの説明に従うことはできません。ただし、以前に Web アプリケーション・サーバーを構成してある場合は除きます。WebSphere 固有の背景情報については、『*WebSphere Application Server* と *EGL*』を参照してください。

また、EGL Web プログラムをデバッグしたい場合は、『*EGL Web デバッグのためのサーバーの準備*』で説明されているように、サーバーをその目的で準備する必要があります。すでに『*EGL Web デバッグのためのサーバーの開始*』の説明に従ってサーバーをデバッグ用に始動してある場合は、この手順の時間を節約できます。

EGL Web デバッグ・セッションを開始するには、次のようにします。

1. プロジェクト・エクスプローラーで、**WebContent** および **WEB-INF** フォルダを展開します。実行したい JSP ファイルを右クリックした後、「**デバッグ (Debug)**」>「**サーバーでデバッグ**」を選択します。「サーバーの選択」ダイアログが表示されます。

2. すでにこの Web プロジェクト用にサーバーを構成してある場合は、「**既存のサーバーを選択**」を選択し、リストからサーバーを選択します。「完了」をクリックして、サーバーを開始し (必要な場合)、サーバーへアプリケーションをデプロイし、アプリケーションを開始します。
3. この Web プロジェクト用にサーバーを構成していない場合は、以下のように作業を進めることができます。ただし、アプリケーションが JNDI データ・ソースにアクセスしない場合に限りです。
 - a. 「**手操作でサーバーを定義**」を選択します。
 - b. ホスト名を指定します。これは、(ローカル・マシンの場合) **localhost** です。
 - c. 実行時にアプリケーションをデプロイする予定の Web アプリケーション・サーバーに類似のサーバーについて、サーバー型の選択を行います。選択項目には、「**WebSphere v5.1 テスト環境**」と「**WebSphere v6.0 サーバー (WebSphere v6.0 Server)**」があります。
 - d. 現行プロジェクトの作業時に選択項目を変更する予定がない場合は、「**プロジェクトのデフォルトとしてサーバーを設定**」のチェック・ボックスを選択します。
 - e. ほとんどの場合、このステップは回避できますが、デフォルトと異なる設定を指定したい場合は、「**次へ**」をクリックして選択を行ってください。
 - f. 「完了」をクリックして、サーバーを開始し、サーバーへアプリケーションをデプロイし、アプリケーションを開始します。

関連する概念

303 ページの『EGL デバッガー』

371 ページの『WebSphere Application Server と EGL』

199 ページの『Web サポート』

関連するタスク

312 ページの『EGL Web デバッグのためのサーバーの準備』

313 ページの『EGL Web デバッグのためのサーバーの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

『EGL デバッガーでのブレークポイントの使用』

316 ページの『EGL デバッガーでの変数の表示』

EGL デバッガーでのブレークポイントの使用

ブレークポイントは、プログラムの実行を一時停止する場合に使用します。ブレークポイントは、EGL デバッグ・セッションの内側または外側で管理できます。ブレークポイントで作業する場合は、以下の点に留意する必要があります。

- ・「ソース」ビューの左マージンの青マーカーは、ブレークポイントが設定されていて使用可能であることを示す。
- ・「ソース」ビューの左マージンの白マーカーは、ブレークポイントが設定されているが使用不可であることを示す。
- ・左マージンにマーカーがない場合は、ブレークポイントが設定されていないことを示す。

Add or remove a breakpoint (ブレークポイントの追加または除去)

EGL ソース・ファイル内の単一のブレークポイントを追加または除去するには、以下の手順のうちいずれか 1 つを実行します。

- 「ソース」ビューの左マージン内のブレークポイント行にカーソルを配置し、ダブルクリックする。
- 「ソース」ビューの左マージン内のブレークポイント行にカーソルを配置し、右クリックする。コンテキスト・メニューが表示されます。適切なメニュー項目をクリックする。

Disable or enable a breakpoint (ブレークポイントを使用不可または使用可能にする)

EGL ソース・ファイル内の単一のブレークポイントを使用不可または使用可能にするには、次のようにします。

1. 「ブレークポイント」ビューで、ブレークポイントを右クリックする。コンテキスト・メニューが表示されます。
2. 適切なメニュー項目をクリックする。

Remove all breakpoints (すべてのブレークポイントを除去)

EGL ソース・ファイルからすべてのブレークポイントを除去するには、次のようにします。

1. 「ブレークポイント」ビューに表示されているブレークポイントを右クリックする。コンテキスト・メニューが表示されます。
2. 「すべてを除去」をクリックする。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

310 ページの『EGL デバッガーでの起動構成の作成』

310 ページの『EGL デバッガーでの 非 J2EE アプリケーションの開始』

『EGL デバッガーでのアプリケーションのステップスルー』

316 ページの『EGL デバッガーでの変数の表示』

EGL デバッガーでのアプリケーションのステップスルー

『EGL デバッガー』で説明されているように、EGL デバッガーでは、デバッグ・セッション時にプログラムの実行を制御するための以下のコマンドが提供されています。

再開 次のブレークポイントまたはプログラムが終了するまでコードを実行する。

指定行まで実行

「ソース」ビューで実行可能な行を選択し、この行に対してコードを実行する。

ステップイントゥ

次の EGL ステートメントを実行して、休止します。プログラムは、呼び出し先関数の最初の文で停止します。

ステップオーバー

次の EGL ステートメントを実行して休止するが、現在の関数から呼び出された関数内では停止しない。

ステップ・リターン

呼び出し側プログラムまたは関数に戻る。

「指定行まで実行」を除いて、各コマンドは以下のような方法でアクセスすることができます。

- 「デバッグ」ビューのツールバー上の適切なボタンをクリックする。
- 「Run (実行)」メニュー上の適切なメニュー項目をクリックする。
- 「デバッグ」ビューで強調表示されているスレッドを右クリックし、適切なメニュー項目をクリックする。

「指定行まで実行」を使用するには、プログラムが一時停止しているときに次のようにします。

1. 実行可能な行の「ソース」ビューの左マージンにカーソルを配置し、右クリックする。コンテキスト・メニューが表示されます。
2. 「指定行まで実行」をクリックする。

「指定行まで実行」を使用する場合は、以下の点に留意する必要があります。

- 「デバッグ」ビューまたは「Run (実行)」メニューでは、この操作は使用できない。
- 「指定行まで実行」は、使用可能なブレークポイントで停止する。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

310 ページの『EGL デバッガーでの起動構成の作成』

310 ページの『EGL デバッガーでの 非 J2EE アプリケーションの開始』

314 ページの『EGL デバッガーでのブレークポイントの使用』

『EGL デバッガーでの変数の表示』

EGL デバッガーでの変数の表示

プログラムが一時停止している場合は、プログラムの変数の現行値を表示できます。

プログラムの変数を表示するには、次のようにします。

1. 「変数」ビューで、ナビゲーター内のパーツを展開して変数を表示する。
2. 変数の型を表示するには、ツールバーの「型名の表示」ボタンをクリックする。
3. 変数の詳細を別のペインに表示するには、変数をクリックし、ツールバーの「Show Detail (詳細を表示)」ボタンをクリックする。

関連する概念

303 ページの『EGL デバッガー』

関連するタスク

310 ページの『EGL デバッガーでの起動構成の作成』

310 ページの『EGL デバッガーでの 非 J2EE アプリケーションの開始』

315 ページの『EGL デバッガーでのアプリケーションのステップスルー』

314 ページの『EGL デバッガーでのブレークポイントの使用』

EGL ビルド・パーツによる作業

ビルド・ファイルの作成

ビルド・ファイルを作成するには、次のようにします。

1. ファイルを入れるプロジェクトまたはフォルダーを識別する。プロジェクトまたはフォルダーがない場合は、プロジェクトまたはフォルダーを作成する必要があります。プロジェクトは EGL または EGL Web プロジェクトである必要があります。
2. ワークベンチで「ファイル」 > 「新規」 > 「EGL ビルド・ファイル」をクリックする。
3. EGL ビルド・ファイルを入れるプロジェクトまたはフォルダーを選択する。「ファイル名」フィールドに、EGL ビルド・ファイルの名前を入力する (例: MyEGLbuildParts)。ファイル名には .eglbld 拡張子が必要です。拡張子が欠落していたり無効な拡張子が指定されたりすると、自動的にファイル名の末尾に拡張子が付加されます。
4. 「完了」をクリックし、EGL ビルド・パーツ宣言が含まれていないビルド・ファイルを作成する。ビルド・ファイルが「プロジェクト・エクスプローラー」ビューに表示され、EGL ビルド・パーツ・エディターで自動的に開かれます。
5. ビルド・ファイルを作成する前に EGL ビルド・パーツを追加するには、「次へ」をクリックする。追加するビルド・パーツのタイプを選択し、「次へ」をクリックします。ビルド・パーツ名と説明を入力し、「完了」をクリックします。ビルド・ファイルが「プロジェクト・エクスプローラー」ビューに表示され、EGL ビルド・パーツ・エディターで自動的に開かれます。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

1 ページの『EGL の紹介』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

347 ページの『EGL ビルド・ファイルへの import ステートメントの追加』

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

132 ページの『EGL Web プロジェクトの作成』

汎用ビルド・オプションの設定

ビルド記述子パーツ

ビルド記述子パーツは生成プロセスを制御します。パーツには以下のような種類の情報があります。

- **ビルド記述子オプション** は、EGL 出力の生成および準備の方法を指定し、ビルド記述子オプションのサブセットによって他のビルド・パーツを生成プロセスに

組み込むことができます。特定のオプションに関する詳細については、『ビルド記述子オプション』を参照してください。

- **Java ランタイム・プロパティ** は以下のプロパティに値を割り当てます。
 - `vgj.datemask.gregorian.long.locale`。これらの値には、2 つのケースのいずれかで使用される日付マスクが含まれています。
 - システム変数 `VGVar.currentFormattedGregorianCalendar` 用に生成された Java コードが呼び出されます。
 - 項目プロパティ **dateFormat** が `systemGregorianCalendarFormat` に設定されている場合、10 文字以上の長さのページ項目またはテキスト書式フィールドが EGL によって検証されます。

locale の意味については、このセクションの最後で説明します。

- `vgj.datemask.gregorian.short.locale` には、項目プロパティ **dateFormat** が `systemGregorianCalendarFormat` に設定されている場合に、10 文字未満の長さのページ項目またはテキスト書式フィールドが EGL によって検証されるときに使用される、日付マスクが含まれます。

locale の意味については、このセクションの最後で説明します。

- `vgj.datemask.julian.long.locale` には、次の 2 つのケースのうちいずれかで使用する日付マスクがあります。
 - システム変数 `VGVar.currentFormattedJulianCalendar` 用に生成された Java コードが呼び出されます。
 - 項目プロパティ **dateFormat** が `systemJulianCalendarFormat` に設定されている場合、8 文字以上の長さのページ項目またはテキスト書式フィールドが EGL によって検証されます。

locale の意味については、このセクションの最後で説明します。

- `vgj.datemask.julian.short.locale` には、項目プロパティ **dateFormat** が `systemJulianCalendarFormat` に設定されている場合に、10 文字未満の長さのページ項目またはテキスト書式フィールドが EGL によって検証されるときに使用される、日付マスクが含まれます。

locale の意味については、このセクションの最後で説明します。

- `vgj.jdbc.database.SN` では、ご使用の Java コードで使用可能なデータベースを識別します。

デプロイメント時に *SN* の置換値を指定する場合は、プロパティ自体の名前をカスタマイズする必要があります。代わりにになる置換値は、`VGLib.connectionService` 内に含まれているサーバー名、または `sysLib.connect` の起動時に組み込まれるデータベース名のいずれかと一致する必要があります。

また、日付マスクのプロパティ名をカスタマイズする必要もあります。

- 実行単位内で、最初に有効になる各プロパティは、最後の修飾子 (*locale*) がプログラム・プロパティ **vgj.nls.code** の値と一致するような名前を持ちます。

- Web アプリケーションでは、プロパティのさまざまな集合は、プログラムがシステム変数 `sysLib.setLocale` に設定される場合に有効となります。

マスター・ビルド記述子: マスター・ビルド記述子を使用して、オーバーライドが不可能で、EGL のインストール先システムで発生するすべての生成について有効な情報を指定するよう、システム管理者から依頼される場合があります。『マスター・ビルド記述子』に記述されているメカニズムを使用して、システム管理者はパーツ、およびパーツが含まれている EGL ビルド・ファイルを名前によって識別できます。

マスター・ビルド記述子内の情報が特定の生成プロセスにとって不十分である場合、またはマスター・ビルド記述子が識別されない場合は、生成時にビルド記述子、および生成固有のパーツを含む EGL ビルド・ファイルを指定できます。マスター・ビルド記述子と同様に、生成固有のビルド記述子は EGL ビルド・ファイルのトップレベルになければなりません。

生成固有のビルド記述子からビルド記述子のチェーンを作成して、チェーン内の最初のビルド記述子が処理されてから 2 番目のビルド記述子が処理され、2 番目のビルド記述子が処理されてから 3 番目のビルド記述子が処理されるようにすることができます。所定のビルド記述子を定義するときは、ビルド記述子オプション **nextBuildDescriptor** を割り当ててチェーンを開始 (または継続) します。システム管理者は同じ技法を使用してマスター・ビルド記述子からチェーンを作成できます。チェーンング情報の意味については後述します。

ビルド記述子によって参照されるビルド・パーツは、参照ビルド記述子に可視である必要があります。『パーツの参照』で説明されている規則に従う必要があります。ビルド・パーツには、リンケージ・オプション・パーツ、リソース関連パーツ、次のビルド記述子などがあります。

オプションの優先順位: 与えられたビルド記述子オプション (または Java ランタイム・プロパティ) の場合、生成時間で最初に処理される値は有効であり、全体から見た優先順位は、次のようになります。

1. マスター・ビルド記述子
2. 生成固有のビルド記述子。このビルド記述子から拡張されたチェーンが後に続きます。
3. マスター・ビルド記述子から拡張されたチェーン

この方式は次の点で有用です。

- システム管理者は、マスター・ビルド記述子を設定することによって不変の値を指定できます。
- 生成固有のビルド記述子を使用して、生成に固有の値を割り当てることができます。
- プロジェクト・マネージャーは、1 つ以上のビルド記述子をカスタマイズすることによってデフォルトのセットを指定できます。たいていの場合、このような状況では生成固有のビルド記述子はプロジェクト・マネージャーによって開発されたチェーン内の最初のビルド記述子を指します。

デフォルト・オプションは、生成方法や準備方法が同様の一連のプログラムを開発するときに便利です。

- ・ システム管理者は、マスター・ビルド記述子から拡張されたチェーンを設定することにより、一般的なデフォルトのセットを作成できます。ただし、通常はこのフィーチャーは使用しません。

所定のビルド記述子が複数回使用される場合、そのビルド記述子の最初のアクセスのみが有効になります。また、特定のオプションについては最初の指定のみが有効になります。

例: 例えば、マスター・ビルド記述子に以下の（非現実的な）オプションと値のペアが含まれていると仮定します。

```
OptionX          02
OptionY          05
```

この例で、生成固有のビルド記述子 (myGen) には、以下のようなオプションと値のペアが含まれています。

```
OptionA          20
OptionB          30
OptionC          40
OptionX          50
```

myGen で示されているように次のビルド記述子は myNext01 であり、以下を含んでいます。

```
OptionA          120
OptionD          150
```

myNext01 で示されているように次のビルド記述子は myNext02 であり、以下を含んでいます。

```
OptionB          220
OptionD          260
OptionE          270
```

マスター・ビルド記述子で示されているように次のビルド記述子は myNext99 であり、以下を含んでいます。

```
OptionZ          99
```

EGL は以下の順序でオプション値を受け入れます。

1. マスター・ビルド記述子のオプションの値は以下のようになります。

```
OptionX          02
OptionY          05
```

これらのオプションは、他のオプションをすべてオーバーライドします。

2. 生成固有のビルド記述子 myGen の値は次のようになります。

```
OptionA          20
OptionB          30
OptionC          40
```

myGen の optionX の値は無視されました。

3. myNext01 および myNext02 のその他のオプションの値は次のようになります。

```
OptionD          150
OptionE          270
```

myNext01 の optionA の値は無視されました。myNext02 の optionD の値も同様に無視されました。

4. myNext99 のその他のオプションの値は次のようになります。

OptionZ 99

関連する概念

354 ページの『ビルド』
377 ページの『Java ランタイム・プロパティー』
24 ページの『パーツの参照』
『マスター・ビルド記述子』
19 ページの『パーツ』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』
336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』
325 ページの『ビルド記述子の汎用オプションの編集』
328 ページの『ビルド記述子での Java ランタイム・プロパティーの編集』
336 ページの『EGL ビルド・ファイル内のリソース関連パーツの編集』

関連する参照項目

415 ページの『ビルド記述子オプション』
584 ページの『Java ランタイム・プロパティー (詳細)』

957 ページの『connect()』
980 ページの『connectionService()』
971 ページの『setLocale()』
1007 ページの『currentFormattedGregorianCalendar()』
1008 ページの『currentFormattedJulianDate()』

マスター・ビルド記述子

インストールは、ビルド・オプション用のデフォルト値の独自のセットを提供して、それらのデフォルト値をオーバーライドできるかどうかを制御することができます。

マスター・ビルド記述子をセットアップするには、同じビルド・ファイル内に 2 つのビルド記述子パーツを作成し、ビルド記述子オプション **nextBuildDescriptor** を使用して、最初のパーツが 2 番目のパーツを参照するようにします。最初のパーツのオプションでは、オーバーライドできないオプション用のデフォルト値を指定します。2 番目のパーツのオプションでは、オーバーライドできるオプション用のデフォルト値を指定します。

ワークベンチにマスター・ビルド記述子をインストールするには、以下のようなプラグイン XML ファイルをワークベンチ・プラグイン・ディレクトリーに追加します。

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="egl.master.build.descriptor.plugin"
  name="EGL Master Build Descriptor Plug-in"
  version="5.0"
  vendor-name="IBM">
<requires />
```

```

<runtime />
<extension point =
"com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
<masterBuildDescriptor
file = "filePath.buildFileName"
name = "masterBuildPartName" />
</extension>
</plugin>

```

ファイル・パス (*filePath*) は、ワークスペース・ディレクトリーと関係しています。

EGL SDK を使用している場合は、マスター・ビルド記述子の名前とファイル・パス名を `eglmaster.properties` という名前のファイルに宣言します。このファイルは CLASSPATH 環境変数にリストされたディレクトリー内にあります。プロパティ・ファイルの形式は以下のとおりです。

```

masterBuildDescriptorName=masterBuildPartName
masterBuildDescriptorFile=fullyQualifiedPathforEGLBuildFile

```

関連する概念

- 354 ページの『ビルド』
- 319 ページの『ビルド記述子パーツ』
- 356 ページの『ビルド計画』
- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連するタスク

- 『EGL ビルド・ファイルへのビルド記述子パーツの追加』

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 531 ページの『`eglmaster.properties` ファイルの形式』
- 548 ページの『マスター・ビルド記述子 `plugin.xml` ファイルの形式』

EGL ビルド・ファイルへのビルド記述子パーツの追加

ビルド記述子パーツは生成プロセスを制御します。ビルド記述子にはオプション名とその関連値が含まれ、これらのオプションと値のペアによって EGL 出力の生成および作成方法を指定します。一部のオプションは、生成プロセスにあるリソース関連パーツなどの他の制御パーツを指定します。ビルド記述子パーツは EGL ビルド・ファイルに追加できます。詳細については、『ビルド記述子パーツ』を参照してください。ビルド記述子パーツを追加するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでビルド・ファイルを右クリックし、「パーツを追加」をクリックする。
4. 「ビルド記述子」ラジオ・ボタンをクリックして「次へ」をクリックする。

5. EGL パーツ名規則に準拠しているビルド記述子の名前を選択する。「名前」フィールドにビルド記述子の名前を入力する。
6. 「説明」フィールドにビルド・パーツの説明を入力する。
7. 「完了」をクリックする。ビルド記述子が EGL ビルド・ファイルで宣言され、EGL ビルド・パーツ・エディターでビルド記述子汎用オプションが表示されます。
8. オプションで、ビルド記述子のチェーンを作成して、チェーン内の最初のビルド記述子が処理されてから 2 番目のビルド記述子が処理され、2 番目のビルド記述子が処理されてから 3 番目のビルド記述子が処理されるようにすることもできます。ビルド記述子のチェーンを開始または継続したい場合は、「Options (オプション)」リストの「**nextBuildDescriptor**」オプション・フィールドで次のビルド記述子を指定します。「**nextBuildDescriptor**」オプション・フィールドにデータを設定するには、次のようにします。
 - a. 「Options (オプション)」リストのスクロール・バーを使用して、「**nextBuildDescriptor**」オプションが表示されるまでスクロールダウンする。
 - b. 「nextBuildDescriptor」行が強調表示されていない場合は、1 度クリックして行を選択する。
 - c. 「値」フィールドを 1 度クリックし、フィールドを編集モードにする。
 - d. 「値」フィールドに次のビルド記述子の名前を入力するか、あるいはドロップダウン・リストから既存のビルド記述子を選択できます。

関連する概念

319 ページの『ビルド記述子パーツ』

関連するタスク

『ビルド記述子の汎用オプションの編集』

328 ページの『ビルド記述子での Java ランタイム・プロパティの編集』

331 ページの『EGL ビルド・ファイルからのビルド記述子パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

725 ページの『命名規則』

ビルド記述子の汎用オプションの編集

ビルド記述子パーツは生成プロセスを制御します。汎用のビルド記述子オプションおよびシンボリック・パラメーターを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでビルド記述子を右クリックし、「**Open (開く)**」を選択する。「エディター」ビューの右上にはボタンが 2 つあります。「Show General Build Descriptor Options (汎用のビルド記述子オプションを表示)」 ボタ

ン (2 つのボタンのうちの最初のボタン) を押すようにします。 EGL ビルド・パーツ・エディターは、現行パーツ定義の汎用のビルド記述子オプションを表示します。

4. オプションで、ビルド記述子のチェーンを作成して、チェーン内の最初のビルド記述子が処理されてから 2 番目のビルド記述子が処理され、2 番目のビルド記述子が処理されてから 3 番目のビルド記述子が処理されるようにすることもできます。ビルド記述子のチェーンを開始または継続したい場合は、「**nextBuildDescriptor**」フィールドで次のビルド記述子を指定する。
「nextBuildDescriptor」行が強調表示されていない場合は、1 度クリックして行を選択し、「値」フィールドを 1 度クリックしてフィールドを編集モードにします。「値」フィールドに次のビルド記述子の名前を入力するか、あるいはドロップダウン・リストから既存のビルド記述子を選択できます。
5. EGL 出力の生成および準備について指定するには、「**ビルド・オプション・フィルター**」ドロップダウン・リストからオプションと値のペアのグループ化を選択する。定義したいオプションが強調表示されていない場合は、1 度クリックして行を選択し、「値」フィールドを 1 度クリックしてフィールドを編集モードにします。オプション値を入力するか、またはドロップダウン・リストが使用可能な場合は、既存の値を選択します。オプションと値のペアの表示を、定義した内容に制限したい場合は、「Show only options specified (指定したオプションのみを表示)」チェック・ボックスをクリックする。

関連する概念

319 ページの『ビルド記述子パーツ』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』

328 ページの『ビルド記述子での Java ランタイム・プロパティの編集』

331 ページの『EGL ビルド・ファイルからのビルド記述子パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

Java 生成のためのオプションの選択

ビルド記述子オプションは、ビルド記述子パーツ内に設定されます。のためのビルド記述子オプションを選択するには、EGL エディターを開始してビルド記述子パーツを編集します。

GUI からビルド記述子パーツの編集を開始する場合は、EGL エディターにすべての EGL ビルド記述子オプションがリストされたペインが含まれています。で生成されたプログラムに適用できるオプションのみを表示するには、「ビルド・オプション・フィルター」ドロップダウン・メニューからカテゴリーを選択します。

必要なオプションをそれぞれ選択して、値を設定します。値としては、リテラル、シンボリック、またはリテラルとシンボリックの組み合わせが可能です。シンボリック・パラメーターは、EGL パーツ・エディターで定義できます (詳細については、『汎用のビルド記述子オプションの編集』を参照してください)。

genDirectory および **destDirectory** という 2 つのビルド記述子オプションでは、値としてシンボリック・パラメーターか、または値の一部を使用できます。たとえ

ば、**genDirectory** の値として `C:¥genout¥%EZEENV%` を指定できます。また、Windows 環境用に生成する場合は、実際の生成ディレクトリーは `C:¥genout¥WIN` です。

リストされているすべてのオプションを指定する必要はありません。ビルド記述子オプションの値を指定しない場合、そのオプションが生成コンテキスト内で適用できるときには、オプションのデフォルトが使用されます。

マスターのビルド記述子をすでに指定している場合は、そのビルド記述子のオプションの値が、他のすべてのビルド記述子の値をオーバーライドします。『ビルド記述子パーツ』で説明しているとおり、マスターおよび生成ビルド記述子は、生成時に他のビルド記述子とチェーニングさせることができます。

関連する概念

319 ページの『ビルド記述子パーツ』

関連するタスク

325 ページの『ビルド記述子の汎用オプションの編集』
『Java ラッパーの生成』

関連する参照項目

415 ページの『ビルド記述子オプション』

Java ラッパーの生成

関連プログラムを生成すると、Java ラッパー・クラスを生成できます。ビルド記述子をセットアップする方法の詳細については、『Java ラッパー』を参照してください。

関連する概念

351 ページの『生成』
351 ページの『プロジェクトへの Java コードの生成』
『Java ラッパー』

関連するタスク

355 ページの『EGL 出力のビルド』
364 ページの『ディレクトリーへ生成される Java コードの処理』

関連する参照項目

415 ページの『ビルド記述子オプション』
595 ページの『Java ラッパー・クラス』
728 ページの『Java ラッパー生成の出力』

Java ラッパー: Java ラッパーは、以下の実行可能プログラムの間のインターフェースとして機能する一連のクラスです。

- 一方はサーブレットまたは手書きの Java プログラム
- 他方は生成されたプログラムまたは EJB セッション Bean

以下の特性を持つビルド記述子を使用すると、Java ラッパー・クラスを生成できます。

- ビルド記述子オプション **enableJavaWrapperGen** を **yes** または **only** に設定する。
- ビルド記述子オプション **linkage** は、ラッパーからプログラムへの呼び出しをガイドする **callLink** エLEMENTが含まれているリンケージ・オプション・パーツを参照する。
- 以下の 2 つのうち 1 つに該当する。
 - ラッパーからプログラムへの呼び出しは EJB セッション Bean 経由 (**callLink** エLEMENT、**linkType** プロパティーが **ejbCall** に設定されている場合)。
 - ラッパーからプログラムへの呼び出しは、リモート (**callLink** エLEMENT、**type** プロパティーが **remoteCall** に設定されている場合)、かつ、**callLink** エLEMENT、**javaWrapper** プロパティーは **yes** に設定されている。

Java ラッパー・クラスと EGL 生成プログラム間を EJB セッション Bean が仲介する場合、以下の特性を持つビルド記述子を使用すると、EJB セッションが生成されます。

- ビルド記述子オプション **enableJavaWrapperGen** を **yes** または **only** に設定する。
- ビルド記述子オプション **linkage** は、ラッパーから EJB セッション Bean への呼び出し (**callLink** エLEMENTの **type** プロパティーが **ejbCall** に設定されている場合) をガイドする **callLink** エLEMENTが含まれているリンケージ・オプション・パーツを参照する。

クラスの使用の詳細については、『Java ラッパー・クラス』を参照してください。
 クラス名の詳細については、『生成される出力 (参照)』を参照してください。

関連する概念

572 ページの『生成される出力』
 357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』
 10 ページの『ランタイム構成』

関連するタスク

327 ページの『Java ラッパーの生成』

関連する参照項目

573 ページの『生成される出力 (参照)』
 595 ページの『Java ラッパー・クラス』
 728 ページの『Java ラッパー生成の出力』

ビルド記述子での Java ランタイム・プロパティーの編集

ビルド記述子パーツを編集する場合は、以下の Java ランタイム・プロパティーに値を割り当てることができます。これらのランタイム・プロパティーの詳細については、『Java ランタイム・プロパティー (詳細)』を参照してください。

- `vgj.jdbc.database.SN`
- `vgj.datemask.gregorian.long.locale`
- `vgj.datemask.gregorian.short.locale`
- `vgj.datemask.julian.long.locale`

- `vgj.datemask.julian.short.locale`

割り当ては、Java コードを生成する場合にのみ使用されます。

プロパティを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでビルド記述子を右クリックし、「Open (開く)」を選択する。EGL パーツ・エディターに、現行パーツ定義の汎用のビルド記述子オプションが表示されます。
4. エディター・ツールバーの「Java ランタイム・プロパティを表示」ボタンをクリックする。
5. Java ランタイム・プロパティ `vgj.jdbc.database.SN` を追加するために、以下の手順を実行する。
 - a. 「接続のデータベース・マッピング」のタイトル付きの画面区域で、「追加」ボタンをクリックする。
 - b. システム・ワード `VGLib.connectionService` をコーディングする場合に使用する「サーバー名」を入力します。この値は、生成されるプロパティの名前の `SN` の代わりに使用されます。
 - c. 接続リストのデータベース・マッピング内の行が強調表示されていない場合は、1 度クリックして行を選択し、「JNDI 名または URL」フィールドを 1 度クリックしてこのフィールドを編集モードにする。J2EE 接続の場合と非 J2EE 接続の場合で意味が異なる値を入力します。
 - J2EE 接続 (実稼働環境で必要になるもの) に関連して、値は、JNDI レジストリーでデータ・ソースをバインドする名前です (例: `jdbc/MyDB`)。
 - 標準の JDBC 接続 (デバッグに使用できるもの) に関連して、値は、接続 URL です (例: `jdbc:db2:MyDB`)。
6. `VGVar.currentFormattedGregorianCalendar` (グレゴリオ日付の場合) か `VGVar.currentFormattedJulianDate` (ユリウス日付の場合) をコード化するか、または長さが 10 以上で `systemGregorianCalendarFormat` または `systemJulianDateFormat` の「dateFormat」プロパティを持つページ項目やテキスト書式フィールドを EGL で検証する場合に使用する日付マスクを割り当てるには、次のようにします。
 - a. 「日付マスク」のタイトル付きの画面区域で、「追加」ボタンをクリックする。
 - b. 「ロケール」列のリスト・ボックスでいずれかのコードを選択する。選択した値は、前記の日付マスク・プロパティのロケールの代わりに使用されます。実行時に使用されるエントリは、ロケールの値が Java ランタイム・プロパティ `vgj.nls.code` の値に一致するエントリのみです。

- c. 「日付マスク」リストの行が強調表示されていない場合は、1 度クリックして行を選択し、「長いグレゴリオ日付マスク」フィールドを 1 度クリックしてこのフィールドを編集モードにする。リスト・ボックスからマスクを選択するかマスクを入力します。D、Y、数字以外の文字をセパレーターとして使用できます。デフォルト値はロケール固有です。
 - d. 「日付マスク」リストの行が強調表示されていない場合は、1 度クリックして行を選択し、「長いユリウス日付マスク」フィールドを 1 度クリックしてこのフィールドを編集モードにする。リスト・ボックスからマスクを選択するかマスクを入力します。D、Y、数字以外の文字をセパレーターとして使用できます。デフォルト値はロケール固有です。
7. 長さが 10 より短くて `systemGregorianCalendar` または `systemJulianCalendar` の「**dateFormat**」プロパティを持つページ項目やテキスト書式フィールドを EGL で検証する場合に使用する日付マスクを割り当てるには、次のようにします。
- a. 「日付マスク」のタイトル付きの画面区域で、「**追加**」ボタンをクリックする。
 - b. 「ロケール」列のリスト・ボックスでいずれかのコードを選択する。選択した値は、前記の日付マスク・プロパティの**ロケール** の代わりに使用されます。実行時に使用されるエントリは、**ロケール** の値が Java ランタイム・プロパティ `vgj.nls.code` の値に一致するエントリのみです。
 - c. 「日付マスク」リストの行が強調表示されていない場合は、1 度クリックして行を選択し、「短いグレゴリオ日付マスク」フィールドを 1 度クリックしてこのフィールドを編集モードにする。リスト・ボックスからマスクを選択するかマスクを入力します。D、Y、数字以外の文字をセパレーターとして使用できます。デフォルト値はロケール固有です。
 - d. 「日付マスク」リストの行が強調表示されていない場合は、1 度クリックして行を選択し、「短いユリウス日付マスク」フィールドを 1 度クリックしてこのフィールドを編集モードにする。リスト・ボックスからマスクを選択するかマスクを入力します。D、Y、数字以外の文字をセパレーターとして使用できます。デフォルト値はロケール固有です。
8. 割り当てを除去するには、その割り当てをクリックし、「**除去**」ボタンをクリックする。

関連する概念

319 ページの『ビルド記述子パーツ』

377 ページの『Java ランタイム・プロパティ』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』

325 ページの『ビルド記述子の汎用オプションの編集』

331 ページの『EGL ビルド・ファイルからのビルド記述子パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

584 ページの『Java ランタイム・プロパティ (詳細)』

980 ページの『`connectionService()`』

1007 ページの『currentFormattedGregorianDate』

1008 ページの『currentFormattedJulianDate』

EGL ビルド・ファイルからのビルド記述子パーツの除去

EGL ビルド・ファイルからビルド記述子パーツを除去するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューで、ビルド記述子パーツを右クリックしてから、「除去」をクリックする。

関連する概念

319 ページの『ビルド記述子パーツ』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』

325 ページの『ビルド記述子の汎用オプションの編集』

328 ページの『ビルド記述子での Java ランタイム・プロパティの編集』

外部ファイル、プリンター、およびキューの関連付けの設定

リソース関連とファイル・タイプ

外部ファイル、プリンター、またはキューにアクセスする EGL 固定レコードは、論理ファイル名またはキュー名を持っています。（プリンターの場合、ほとんどのランタイム・システムの論理ファイル名は *printer* です。）名前は 8 文字以下で指定でき、そのレコードをシステム名に関連させる手段としてのみ意味を持ちます。この名前は、ターゲット・システムが物理ファイル、プリンター、またはキューにアクセスするために使用します。

ファイルまたはキューに関するファイル名またはキュー名は、システム名のデフォルトです。プリンターに関しては、デフォルトは存在しません。

デフォルトを受け入れずに、以下のいずれかまたは両方のアクションを実行できます。

- 生成時に、ビルド記述子を使用して生成プロセスを制御する。ビルド記述子は特定のリソース関連パーツを参照します。リソース関連パーツは、生成済みコードのデプロイ先ターゲット・プラットフォームで、システム名にファイル名を関連付けます。
- 実行時に（ほとんどの場合）、レコード固有の変数 `resourceAssociation`（ファイルまたはキューの場合）の値、またはシステム変数 `ConverseVar.printerAssociation`（印刷出力の場合）の値を変更できます。目的は、デフォルトまたはリソース関連パーツによって指定したシステム名をオーバーライドすることです。

リソース関連パーツは、以下のレコード・タイプには適用されません。

- `basicRecord` (基本レコードがデータ・ストアと対話しないため)
- `SQLRecord` (SQL レコードが ¥ と対話するため)

リソース関連パーツ: リソース関連パーツは一連の関連エレメントからなり、それぞれのエレメントには以下のような特性があります。

- 特定の論理ファイル名またはキュー名に固有である
- 特定のターゲット・システムに固有な一連のエントリーがある。各エントリーは、ターゲット・プラットフォーム上のファイル・タイプとシステム名 (さらに、場合によっては追加情報) を識別します。

関連エレメントは、以下の例のように、階層関係にあるプロパティと値のセットだと考えることができます。

```
// 関連エレメント
property: fileName
value:    myFile01

// 複数のプロパティを持つエントリー
property: system
value:    aix
property: fileType
value:    spool
property: systemName
value:    employee

// 2 番目のエントリー
property: system
value:    win
property: fileType
value:    seqws
property: systemName
value:    c:¥myProduct¥myFile.txt
```

この例では、ファイル名 `myFile01` は以下のファイルに関連しています。

- AIX の `employee`
- Windows 2000/NT/XP の `myFile.txt`

ファイル名は、有効な名前、アスタリスク、または有効な名前の後にアスタリスクという形式でなければなりません。アスタリスクは、1 つ以上の文字に相当するワイルドカードであり、一連の名前を指定できます。例えば、関連エレメントのファイル名に以下の値が含まれる場合、`myFile` という文字ストリングで始まるすべてのファイルという意味です。

```
myFile*
```

プログラムで使われる 1 つのファイル名に複数のエレメントが該当する場合、EGL では最初のエレメントが使用されます。例えば、ファイル名に以下のような値を順番に指定している一連の関連エレメントを考えます。

```
myFile
myFile*
*
```

最後の値に関連付けられたエレメントについて考えてください。この場合、`myFile` の値は 1 個のアスタリスクのみです。このようなエレメントはすべてのファイルに該当することが考えられます。しかし、特定のファイルについて見ると、最後のエ

レメントは、それ以前のエレメントがすべて該当しなかった場合にのみ該当します。例えばプログラムが myFile01 を参照する場合、参照の処理方法を定義する上で、2 番目のエレメントの指定するリンケージが 3 番目のエレメントよりも優先されます。

生成時に EGL は特定の 1 つの関連エレメントを選ぶとともに、最初の適切なエントリーを選択します。エントリーは、以下のいずれかの場合に適切と見なされます。

- 生成のターゲット・システムと **system** プロパティーとが一致する。
- **system** プロパティーの値が以下のとおりである
any

例えば生成のターゲットが AIX であれば、EGL では **aix** または **any** を参照している最初のエントリーが使用されます。

ファイル・タイプ: ファイル・タイプは、関連エレメントの特定のエントリーに関して必要なプロパティーを判別します。EGL ファイル・タイプは以下の表のとおりです。

ファイル・タイプ	説明
ibmcobol	EGL 生成 Java プログラムによってリモートでアクセスされる VSAM ファイルこの場合のシステム名の指定について詳しくは、『VSAM サポート』を参照してください。
mq	MQSeries メッセージ・キュー。このタイプのキューを扱う詳細な方法は、『MQSeries のサポート』を参照してください。
seqws	EGL 生成の Java プログラムがアクセスするシリアル・ファイル。
spool	AIX または Linux のスプール・ファイル。

レコード・タイプと VSAM: 3 つの固定レコード・タイプのそれぞれは、VSAM データ・セットにアクセスできます。ただし、アクセスできるのは、以下のように、そのレコードに関連するエレメント内のファイル・タイプが ibmcobol、vsam、または vsamrs の場合のみです。

- 固定レコードが indexedRecord タイプの場合、VSAM データ・セットは 1 次索引または代替索引付きのキー順データ・セットです。
- 固定レコードが relativeRecord タイプの場合、VSAM データ・セットは相対レコード・データ・セットです。
- 固定レコードが serialRecord タイプの場合、VSAM データ・セットは入力順データ・セットです。

詳細について: リソース関連について詳しくは、以下のトピックを参照してください。

- レコードとファイル・タイプの相互参照
- 関連エレメント

関連する概念

142 ページの『固定レコード・パーツ』
285 ページの『MQSeries のサポート』
19 ページの『パーツ』

143 ページの『レコード・タイプとプロパティ』

141 ページの『レコード・パーツ』

285 ページの『VSAM サポート』

関連タスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

336 ページの『EGL ビルド・ファイル内のリソース関連パーツの編集』

338 ページの『EGL ビルド・ファイルからのリソース関連パーツの除去』

関連する参照項目

407 ページの『関連エレメント』

793 ページの『レコードとファイル・タイプの相互参照』

920 ページの『recordName.resourceAssociation』

432 ページの『resourceAssociations』

439 ページの『system』

988 ページの『printerAssociation』

作業論理単位

リカバリー不能と分類されているリソース (Windows 2000 のシリアル・ファイルなど) を変更すると、その変更結果は比較的永続的になります。コードでも EGL ランタイム・サービスでも、この変更結果を単純に無効にすることはできません。リカバリー可能と分類されているリソース (リレーショナル・データベースなど) を変更すると、コードまたは EGL ランタイム・サービスでは、この変更をコミットして変更内容を永続的にしたり、最後に変更がコミットされた時点の状態に戻すために変更をロールバックしたりできます。

リカバリー可能リソースには以下のものがあります。

- リレーショナル・データベース
- リカバリー可能として構成されている CICS キューまたはファイル。
- MQSeries メッセージ・キュー。ただし、『MQSeries のサポート』で説明するように MQSeries レコードでリカバリー不可と指定されていない場合。

作業論理単位 により、グループとしてコミットまたはロールバックされる入力操作が識別されます。作業単位は、リカバリー可能リソースがコードにより変更された時点で開始され、次のいずれかが初めて発生した時点で終了します。

- 変更内容をコミットまたはロールバックするために、システム関数 **sysLib.commit** または **sysLib.rollback** がコードから呼び出された。
- コードでは処理されないハード・エラーに応答して EGL ランタイム・サービスがロールバックを実行した。この場合、実行単位のすべてのプログラムはメモリーから除去される。
- 以下の場合のように、暗黙的コミットが発生した。
 - プログラムが **show** ステートメントを発行した場合。
 - 実行単位内のトップレベルのプログラムが、『実行単位』で説明されているように、正常に終了した場合。
 - ページ・ハンドラーが **forward** ステートメントを発行したときのように、Web ページが表示された場合。

- プログラムが **converse** ステートメントを発行し、かつ、以下のいずれかに該当する場合。
 - VisualAge Generator との互換性モードで実行しておらず、プログラムがセグメント化されたプログラムである
 - **ConverseVar.commitOnConverse** が 1 に設定されている。
 - VisualAge Generator との互換性モードで実行中であり、**ConverseVar.segmentedMode** が 1 に設定されている。

Java の作業単位: Java 実行単位では、次のように処理が行われます。

- ハード・エラーが原因で Java プログラムが終了すると、ロールバックが実行され、カーソルが閉じ、ロックが解放された場合と同様の結果となる。
- 実行単位が完了すると、EGL がコミットを実行し、カーソルを閉じ、ロックを解放する。
- 複数のデータベースから読み込むために複数の接続を使用している場合には、作業単位では 1 つのデータベースのみを更新する必要があります。これは、シングル・フェーズ・コミットのみが実行可能であるためです。関連情報については、『*VGLib.connectionService*』を参照してください。
- EGL 生成の EJB セッション Bean を使用して EGL 生成のプログラムへアクセスすると、EJB セッション Bean のデプロイメント記述子であるトランザクション属性 (コンテナ・トランザクション・タイプとも呼ばれる) が、トランザクション制御に影響することがある。このトランザクション属性がトランザクション制御に影響を及ぼすのは、呼び出しのリンケージ・オプション・パーツである **callLink** エlementの **remoteComType** プロパティが **direct** である場合に限る。詳細については、『*callLink* Elementの *remoteComType*』を参照。

EJB セッション Bean はトランザクション属性 **REQUIRED** が設定された状態で生成されますが、この値はデプロイメント時に変更できます。このトランザクション属性については、Java の資料を参照してください。

関連する概念

285 ページの『MQSeries のサポート』

798 ページの『実行単位』

247 ページの『SQL サポート』

関連するタスク

394 ページの『J2EE JDBC 接続のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

270 ページの『デフォルト・データベース』

956 ページの『*commit()*』

980 ページの『*connectionService()*』

969 ページの『*rollback()*』

595 ページの『Java ラッパー・クラス』

450 ページの『*callLink* Elementの *luwControl*』

456 ページの『*callLink* Elementの *remoteComType*』

435 ページの『*sqlDB*』

EGL ビルド・ファイルへのリソース関連パーツの追加

リソース関連パーツは、ファイル名を、生成済みコードのデプロイ先にする予定のターゲット・プラットフォーム上のシステム・リソース名に関連付けます。リソース関連パーツは EGL ビルド・ファイルに追加できます。詳細については、『リソース関連とファイル・タイプ』を参照してください。リソース関連パーツを追加するには、以下のことを行います。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでビルド・ファイルを右クリックし、「パーツを追加」をクリックする。
4. 「リソース関連」をクリックして「次へ」をクリックする。
5. EGL パーツ名規則に準拠するリソース関連パーツ名を選択する。「名前」フィールドにリソース関連パーツ名を入力する。
6. 「説明」フィールドにパーツの説明を入力する。
7. 「完了」をクリックする。リソース関連パーツが EGL ビルド・ファイルに追加され、EGL ビルド・パーツ・エディターでリソース関連パーツ・ページが開く。

関連する概念

319 ページの『ビルド記述子パーツ』

331 ページの『リソース関連とファイル・タイプ』

関連するタスク

『EGL ビルド・ファイル内のリソース関連パーツの編集』

338 ページの『EGL ビルド・ファイルからのリソース関連パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

725 ページの『命名規則』

EGL ビルド・ファイル内のリソース関連パーツの編集

リソース関連パーツは、ファイル名を、生成済みコードのデプロイ先にする予定のターゲット・プラットフォーム上のシステム・リソース名に関連付けます。

リソース関連パーツを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。

2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「**ビューの表示**」>「**アウトライン**」を選択して、そのビューを開く。
3. 「アウトライン」ビューでリソース関連パーツを右クリックして「**オープン**」をクリックする。EGL パーツ・エディターは、現行のパーツ定義を表示します。
4. 新規の関連要素をパーツに追加するには、「**関連の追加**」をクリックするか、または Insert (挿入) キーを押して論理ファイル名を入力するか、または論理ファイル名を選択する。
5. 論理ファイル名と関連付けられているデフォルトのシステム名を変更するには、以下のいずれかを行うことができる。
 - 「関連要素」リスト内で対応する行を選択し、名前を 1 度クリックしてそのフィールドを編集モードにする。System ドロップダウン・リストから新規システム名を選択する。
 - 「選択したシステム・エントリーのプロパティ」リストでシステムのプロパティを 1 度クリックして、そのプロパティと関連づけられている値フィールドを編集モードにする。Value ドロップダウン・リストから新規システム名を選択する。
6. 論理ファイル名と関連付けられているデフォルトのファイル・タイプを変更するには、以下のいずれかを行う。
 - 「関連要素」リスト内で論理ファイル名に対応する行を選択し、名前を 1 度クリックしてそのフィールドを編集モードにする。「ファイル・タイプ」ドロップダウン・リストから新規ファイル・タイプを選択します。
 - 「関連要素」リスト内で論理ファイル名に対応する行を選択する。「選択したシステム・エントリーのプロパティ」リストで fileType のプロパティを 1 度クリックして、そのプロパティと関連づけられている値フィールドを編集モードにします。Value ドロップダウン・リストからファイル・タイプを選択します。
7. 必要に応じてリソース関連を変更する。
 - 複数のシステムおよび関連するプロパティのセットを論理ファイル名に関連付けるには、「関連要素」リスト内で論理ファイル名に対応する行を選択する。「関連要素」リストの下部で「**システムを追加**」をクリックする。追加された行が選択され、編集に使用できます。
 - システムおよび関連するプロパティを、関連付けられた論理ファイル名から除去するには、「関連要素」リスト内で論理ファイル名に対応する行を選択する。「関連要素」リストの下部で「**除去**」をクリックするか、あるいは Delete (削除) キーを押します。
 - 論理ファイル名と任意の関連システムを除去するには、「関連要素」リスト内で論理ファイル名に対応する行を選択する。「関連要素」リストの下部で「**除去**」をクリックするか、あるいは Delete (削除) キーを押します。

関連する概念

319 ページの『ビルド記述子パーツ』

331 ページの『リソース関連とファイル・タイプ』

関連するタスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

338 ページの『EGL ビルド・ファイルからのリソース関連パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

EGL ビルド・ファイルからのリソース関連パーツの除去

EGL ビルド・ファイルからリソース関連パーツを除去するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューで、リソース関連パーツを右クリックしてから、「除去」をクリックする。

関連する概念

331 ページの『リソース関連とファイル・タイプ』

関連するタスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

336 ページの『EGL ビルド・ファイル内のリソース関連パーツの編集』

呼び出しおよび転送オプションの設定

リンケージ・オプション・パーツ

リンケージ・オプション・パーツは、以下の点に関する詳細を指定します。

- 生成されたプログラムまたはラッパーが、他の生成済みコードをどのように呼び出すか
- 生成されたプログラムを別の生成プログラムへ非同期でどのように転送するか

最終となるリンケージ・オプションの指定方法:

- 生成時に指定されたリンケージ・オプションを実行時に有効にする。または
- デプロイメント時にリンケージ・プロパティ・ファイル内に指定されたリンケージ・オプションを、実行時に有効にする。このファイルは自作することもできますが、この状況では EGL で自動的に生成されます。
 - リンケージ・オプション・プロパティ **remoteBind** を **RUNTIME** に設定した上で、
 - ビルド記述子オプション **genProperties** を **GLOBAL** または **PROGRAM** に設定して、Java プログラムまたはラッパーを生成します。

ファイルの使用の詳細については、『リンケージ・プロパティ・ファイルのデプロイ』を参照してください。ファイルのカスタマイズの詳細については、『リンケージ・プロパティ・ファイル (参照)』を参照してください。

リンケージ・オプション・パーツのエレメント: リンケージ・オプション・パーツは一連のエレメントから構成されており、それぞれのエレメントには複数のプロパティーと値のセットが含まれます。エレメントには、以下のタイプがあります。

- **callLink** エレメントは、特定の呼び出しで EGL が使用するリンケージ規約を指定します。

callLink エレメントが常に呼び出し先プログラムに適用されます。以下のような関係が有効になります。

- callLink エレメントが、生成しているプログラムを参照している場合、そのエレメントは、ネイティブの Java コードから該当するプログラムへのアクセスを許可する Java ラッパーを生成するかどうかの判断の助けとなります。概要については、『Java ラッパー』を参照してください。Java ラッパーが EJB セッション Bean を経由してプログラムにアクセスするように指示すると、callLink エレメントも EJB セッション Bean を生成します。
 - Java プログラムを生成する場合、そのプログラムによって呼び出されるプログラムを callLink エレメントが参照していれば、その callLink エレメントによって、呼び出しが実装される方法が指定されます (例えば、呼び出しがローカルリモートか)。Java プログラムの発信により EJB Session Bean 経由の呼び出しを指示した場合、callLink エレメントは、EJB セッション Bean を生成します。
- **asynchLink** エレメントは、転送プログラムがシステム関数 `sysLib.startTransaction` を呼び出す場合にプログラムを別のプログラムへ非同期でどのように転送するかを指定します。
 - **transferToProgram** エレメントは、生成された COBOL プログラムの制御をどのように別のプログラムに転送し、処理を終了するかを指定します。このエレメントは、Java 出力には使用されません。*transfer to program* タイプの **transfer** ステートメントを発行するメインプログラムで使用される場合にのみ意味があります。
 - **transferToTransaction** エレメントは、生成されたプログラムの制御をどのようにトランザクションに転送し、処理を終了するかを指定します。このエレメントは、*transfer to transaction* タイプの **transfer** ステートメントを発行するメインプログラムで使用する場合にのみ意味があります。ターゲット・プログラムが VisualAge Generator または (別名のない場合) EGL で生成される場合には、このエレメントは不要です。

プログラムまたはレコードがどのエレメントを参照しているかを識別する: 各エレメント内で、プロパティー (例、**pgmName**) がプログラムまたはレコードがどのエレメントを参照しているかを識別します。特に指定のない場合は、プロパティーの値は、有効な名前、アスタリスク、または有効な名前の先頭部分の後にアスタリスクが付いたものになることがあります。アスタリスクは、1 つ以上の文字に相当するワイルドカードであり、一連の名前を指定できます。

pgmName プロパティーに対して、次の値を含む callLink エレメントを考慮してください。

`myProg*`

このエレメントは *myProg* の文字で始まるすべての EGL プログラム・パーツに関係します。

複数のエレメントが該当する場合、EGL では最初のエレメントが使用されます。例えば、以下のような複数の **pgmName** 値を順番に指定している一連の **callLink** エレメントを考えます。

```
YourProgram  
YourProg*  
*
```

最後の値に関連付けられたエレメントについて考えてください。この場合、**pgmName** の値は 1 個のアスタリスクのみです。このようなエレメントはすべてのプログラムに該当することが考えられます。しかし、個々のプログラムについて見ると、最後のエレメントは、それ以前のエレメントがすべて該当しなかった場合にのみ該当します。例えばプログラムが **YourProgram01** を呼び出す場合、呼び出しの処理方法を定義するうえで、2 番目のエレメント (**YourProg***) の指定するリンケージが 3 番目のエレメント (*) よりも優先されます。

ほとんどの場合、より特定名を指定したエレメントを先に、より一般的な名前を指定したエレメントを後にすべきです。前述の例では、アスタリスク付きのエレメントが適切にも最後に置かれて、デフォルトのリンケージ指定を提供するようになっています。

関連する概念

327 ページの『Java ラッパー』

19 ページの『パーツ』

関連するタスク

『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

395 ページの『リンケージ・プロパティ・ファイルのデプロイ』

339 ページの『リンケージ・オプション・パーツの **asynchLink** エレメントの編集』

341 ページの『リンケージ・オプション・パーツの **callLink** エレメントの編集』

344 ページの『リンケージ・オプション・パーツの **transfer** 関連エレメントの編集』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

410 ページの『**asynchLink** エレメント』

608 ページの『**call**』

442 ページの『**callLink** エレメント』

431 ページの『**linkage**』

708 ページの『リンケージ・プロパティ・ファイル (詳細)』

975 ページの『**startTransaction()**』

697 ページの『**transfer**』

1019 ページの『**transferToTransaction** エレメント』

EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加

リンケージ・オプション・パーツは、生成済み EGL プログラムが呼び出しを実装して転送する方法、およびこのプログラムがファイルにアクセスする方法を説明します。このタイプのパーツを追加するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。

- a. EGL ファイルを右クリックする。
- b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでビルド・ファイルを右クリックし、「パーツを追加」をクリックする。
4. 「リンケージ・オプション」をクリックし、「次へ」をクリックする。
5. EGL パーツ名規則に準拠するリンケージ・オプション・パーツ名を選択する。
「名前」フィールドにリンケージ・オプション・パーツ名を入力する。
6. 「説明」フィールドにパーツの説明を入力する。
7. 「完了」をクリックする。リンケージ・オプション・パーツが EGL ファイルに追加され、EGL ビルド・パーツ・エディターでリンケージ・オプション・パーツ・ページが開く。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

339 ページの『リンケージ・オプション・パーツのasynchLink エLEMENTの編集』

『リンケージ・オプション・パーツの callLink エLEMENTの編集』

344 ページの『リンケージ・オプション・パーツのtransfer 関連ELEMENTの編集』

346 ページの『EGL ビルド・ファイルからのリンケージ・オプション・パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

725 ページの『命名規則』

リンケージ・オプション・パーツの callLink エLEMENTの編集

リンケージ・オプション・パーツは、生成済み EGL プログラムが呼び出しを実装して転送する方法、およびこのプログラムがファイルにアクセスする方法を説明します。パーツの callLink エLEMENTを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでリンケージ・オプション・パーツを右クリックして、「オープン」をクリックする。EGL パーツ・エディターに現在のパーツ宣言が表示されます。

4. エディター・ツールバーの「CallLink エlementを表示」ボタンをクリックする。
5. 新規の callLink Elementを追加するには、「追加」をクリックするか、または Insert (挿入) キーを押して、プログラム名 (pgmName) を入力するか、または「プログラム名」ドロップダウン・リストからプログラム名を選択する。
6. プログラム名と関連付けられているデフォルトの呼び出しタイプを変更するには、以下のいずれかを行える。
 - 「callLink Element」リスト内で対応する行を選択し、「型」フィールド (localCall、remoteCall、ejbCall) を 1 度クリックしてそのフィールドを編集モードにする。Type ドロップダウン・リストから新規呼び出しタイプを選択します。
 - 「選択した callLink Elementのプロパティ」リストで type プロパティを 1 度クリックして、そのプロパティと関連づけられている値フィールドを編集モードにする。Value ドロップダウン・リストから新規呼び出しタイプを選択します。
7. プログラム名と関連付けられている他のプロパティが、呼び出しタイプに基づき「選択した callLink Elementのプロパティ」リストにリストされます。プロパティのいずれか 1 つの値を変更するには、プログラム名を選択します。「選択した callLink Elementのプロパティ」リストで、定義したいプロパティを 1 度クリックして、そのプロパティと関連付けられている値フィールドを編集モードにします。新規の値を定義するには、Value ドロップダウン・リストでオプションを選択するか、あるいは値フィールドに新規の値を入力します。一部のプロパティは、ドロップダウン・リストでのオプションの選択のみが可能です。それ以外のプロパティの場合は、値フィールド内での値の入力のみが可能です。
8. 必要に応じて callLink Element・リストを変更する。
 - callLink Elementを位置変更するには、Elementを選択して「上に移動」または「下に移動」のいずれかをクリックする。
 - callLink Elementを除去するには、Elementを選択して「除去」をクリックするか、あるいは Delete (削除) キーを押す。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

339 ページの『リンケージ・オプション・パーツのasynchLink Elementの編集』

344 ページの『リンケージ・オプション・パーツのtransfer 関連Elementの編集』

346 ページの『EGL ビルド・ファイルからのリンケージ・オプション・パーツの除去』

関連する参照項目

410 ページの『asynchLink Element』

442 ページの『callLink Element』

414 ページの『EGL ビルド・ファイル形式』

708 ページの『リンケージ・プロパティ・ファイル (詳細)』

Enterprise JavaBean (EJB) セッション Bean: EJB セッション Bean には以下のコンポーネントが含まれています。

- ホーム・インターフェース。実行時に EJB セッション Bean へのクライアント・アクセスを提供します。
- リモート Bean インターフェース。クライアントが直接使用可能なメソッドをリストします。
- Bean 実装。クライアントが間接的に使用可能なロジックが含まれています。

EJB セッション Bean は、あるプログラムと別のプログラム間、または EGL Java ラッパーとプログラム間の仲介をします。EJB セッション Bean は、生成時に使用されるリンケージ・オプション・パーツの設定に大きく依存しています。詳細については、『リンケージ・オプション・パーツ』、特に **callLink** エLEMENTの概説を参照してください。

出力ファイル名の詳細については、『生成される出力 (参照)』を参照してください。

関連する概念

572 ページの『生成される出力』

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

573 ページの『生成される出力 (参照)』

リンケージ・オプション・パーツの **asynchLink** ELEMENTの 編集

リンケージ・オプション・パーツは、生成済み EGL プログラムが呼び出しを実装して転送する方法、およびこのプログラムがファイルにアクセスする方法を説明します。パーツの **asynchLink** ELEMENTを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでリンケージ・オプション・パーツを右クリックして、「オープン」をクリックする。EGL ビルド・パーツ・エディターに現在のパーツ宣言が表示されます。
4. エディター・ツールバーの「AsynchLink ELEMENTを表示」ボタンをクリックする。
5. 新規の **asynchLink** ELEMENTを追加するには、「追加」をクリックするか、または Insert (挿入) キーを押して、レコード名 (recordName) を入力するか、または「レコード名」ドロップダウン・リストからレコード名を選択する。

6. レコード名と関連付けられているデフォルトのリンケージ・タイプを変更するには、以下のいずれかを実行できます。
 - 「asynchLink エlement」リスト内で対応する行を選択し、「型」フィールド (localAsynch、remoteAsynch) を 1 度クリックしてそのフィールドを編集モードにする。Type ドロップダウン・リストから新規リンケージ・タイプを選択します。
 - 「選択した asynchLink Elementのプロパティ」リストで type プロパティを 1 度クリックして、そのプロパティと関連づけられている値フィールドを編集モードにする。Value ドロップダウン・リストから新規リンケージ・タイプを選択します。
7. レコード名と関連付けられている他のプロパティは、「選択した asynchLink Elementのプロパティ」リストにリンケージ・タイプを基にしてリストされます。プロパティのいずれか 1 つの値を変更するには、レコード名を選択します。「選択した asynchLink Elementのプロパティ」リストで、定義したいプロパティを 1 度クリックして、そのプロパティと関連付けられている値フィールドを編集モードにします。新規の値を定義するには、Value ドロップダウン・リストでオプションを選択するか、あるいは値フィールドに新規の値を入力します。一部のプロパティは、ドロップダウン・リストでのオプションの選択のみが可能です。それ以外のプロパティの場合は、値フィールド内での値の入力のみが可能です。
8. 必要に応じて asynchLink Element・リストを変更する。
 - asynchLink Elementを位置変更するには、Elementを選択して「上に移動」または「下に移動」のいずれかをクリックする。
 - asynchLink Elementを除去するには、Elementを選択して「除去」をクリックするか、あるいは Delete (削除) キーを押す。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

341 ページの『リンケージ・オプション・パーツの callLink Elementの編集』
『リンケージ・オプション・パーツの transfer 関連Elementの 編集』

346 ページの『EGL ビルド・ファイルからのリンケージ・オプション・パーツの除去』

関連する参照項目

410 ページの『asynchLink Element』

414 ページの『EGL ビルド・ファイル形式』

708 ページの『リンケージ・プロパティ・ファイル (詳細)』

975 ページの『startTransaction()』

リンケージ・オプション・パーツの transfer 関連Elementの 編集

リンケージ・オプション・パーツは、生成済み EGL プログラムが呼び出しを実装して転送する方法、およびこのプログラムがファイルにアクセスする方法を説明します。パーツの transfer 関連要素を編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューでリンケージ・オプション・パーツを右クリックして、「オープン」をクリックする。EGL ビルド・パーツ・エディターに現在のパーツ宣言が表示されます。
4. エディター・ツールバーの「TransferLink 要素を表示」ボタンをクリックする。「プログラムへ転送」リストと「トランザクションへ転送」リストが表示されます。
5. 「プログラムへ転送」リストを編集するには、次のようにします。
 - a. 「プログラムへ転送」リストの下部にある「追加」をクリックするか、または Insert (挿入) キーを押して、リンク元プログラム (fromPgm) の名前を入力するか、または「From Program name (リンク元プログラム名)」ドロップダウン・リストからプログラム名を選択する。
 - b. リンク先プログラム (toPgm) の名前を編集するには、「プログラムへ転送」リスト内で対応する行を選択し、「リンク先プログラム」フィールドをクリックしてこのフィールドを編集モードにする。プログラム名を入力するか、または「リンク先プログラム」ドロップダウン・リストからプログラム名を選択します。
 - c. 別名が必要な場合は、「プログラムへ転送」リスト内で対応する行を選択し、「別名」フィールドを 1 度クリックしてこのフィールドを編集モードにする。別名を入力します。
 - d. プログラム名に関連付けられているデフォルトのリンケージ・タイプを変更するには、「プログラムへ転送」リスト内で対応する行を選択し、「リンク・タイプ (linkType)」フィールドを 1 度クリックしてこのフィールドを編集モードにする。「リンク・タイプ」ドロップダウン・リストから新規リンケージ・タイプを選択します。
6. トランザクションへ転送リストを編集するには、次のようにします。
 - a. 「トランザクションへ転送」リストの下部にある「追加」をクリックするか、または Insert (挿入) キーを押して、リンク先プログラム (toPgm) の名前を入力するか、または「To Program name (リンク先プログラム名)」ドロップダウン・リストからプログラム名を選択する。
 - b. 別名が必要な場合は、「トランザクションへ転送」リスト内で対応する行を選択し、「別名」フィールドを 1 度クリックしてこのフィールドを編集モードにする。別名を入力します。
 - c. プログラム名に関連付けられている「外部定義」プロパティを編集するには、「トランザクションへ転送」リスト内で対応する行を選択し、「外部定義」フィールドを 1 度クリックしてこのフィールドを編集モードにする。「外部定義」ドロップダウン・リストから外部定義されたプロパティを選択します。
 - d. 必要に応じてトランザクションへ転送リストを変更する。

- transferToTransaction エlementを位置変更するには、Elementを選択して「上に移動」または「下に移動」のいずれかをクリックする。
- transferToTransaction Elementを除去するには、Elementを選択して「除去」をクリックするか、あるいは Delete (削除) キーを押す。

注: 「プログラムへ転送」が関連するのは、以下の製品だけです。

- Rational Application Developer for iSeries
- Rational Application Developer for z/OS

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

339 ページの『リンケージ・オプション・パーツのasynchLink Elementの編集』

341 ページの『リンケージ・オプション・パーツの callLink Elementの編集』
『EGL ビルド・ファイルからのリンケージ・オプション・パーツの除去』

関連する参照項目

414 ページの『EGL ビルド・ファイル形式』

1019 ページの『transferToTransaction Element』

EGL ビルド・ファイルからのリンケージ・オプション・パーツの除去

EGL ビルド・ファイルからリンケージ・オプション・パーツを除去するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開くために、プロジェクト・エクスプローラーで次のようにする。
 - a. EGL ビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. 「アウトライン」ビューが表示されていない場合は、「ウィンドウ」メニューから「ビューの表示」>「アウトライン」を選択して、そのビューを開く。
3. 「アウトライン」ビューで、リンケージ・オプション・パーツを右クリックしてから、「除去」をクリックする。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

339 ページの『リンケージ・オプション・パーツのasynchLink Elementの編集』

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』
344 ページの『リンケージ・オプション・パーツのtransfer 関連ELEMENTの編集』

他の EGL ビルド・ファイルの参照の設定

EGL ビルド・ファイルへの import ステートメントの追加

import ステートメントを使用すると、EGL ビルド・ファイルで他のビルド・ファイル内のパーツを参照できます。インポート・フィーチャーの詳細については、『インポート』を参照してください。

import ステートメントを EGL ビルド・ファイルに追加するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで以下のことを行います。
 - a. プロジェクト・エクスプローラーでビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. ビルド・パーツ・エディターの「インポート」タブをクリックする。
3. 「追加」ボタンをクリックする。
4. インポートするファイルまたはフォルダーの名前を入力または選択し、「OK」をクリックする。

関連する概念

36 ページの『インポート』

関連するタスク

『EGL ビルド・ファイル内の import ステートメントの編集』

348 ページの『EGL ビルド・ファイルからの import ステートメントの除去』

EGL ビルド・ファイル内の import ステートメントの編集

EGL ビルド・ファイル内の import ステートメントを編集するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで以下のことを行います。
 - a. プロジェクト・エクスプローラーでビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. ビルド・パーツ・エディターの「インポート」タブをクリックする。import ステートメントが表示されます。
3. 変更したい import ステートメントを選択し、「編集」ボタンを押す。
4. インポートするファイルまたはフォルダーの名前を入力または選択し、「OK」をクリックする。

関連する概念

36 ページの『インポート』

関連するタスク

347 ページの『EGL ビルド・ファイルへの import ステートメントの追加』

『EGL ビルド・ファイルからの import ステートメントの除去』

EGL ビルド・ファイルからの import ステートメントの除去

EGL ビルド・ファイル内の import ステートメントを除去するには、次のようにします。

1. EGL ビルド・パーツ・エディターで EGL ビルド・ファイルを開く。ファイルを開いていない場合は、プロジェクト・エクスプローラーで以下のことを行います。
 - a. プロジェクト・エクスプローラーでビルド・ファイルを右クリックする。
 - b. 「アプリケーションから開く」>「EGL ビルド・パーツ・エディター」を選択する。
2. ビルド・パーツ・エディターの「インポート」タブをクリックする。import ステートメントが表示されます。
3. 除去したい import ステートメントを選択し、「除去」ボタンを押す。

関連する概念

36 ページの『インポート』

関連するタスク

347 ページの『EGL ビルド・ファイルへの import ステートメントの追加』

347 ページの『EGL ビルド・ファイル内の import ステートメントの編集』

EGL ビルド・パスの編集

概説については、以下のトピックを参照してください。

- パーツへの参照
- EGL ビルド・パスおよび *eglp*ath

プロジェクトを EGL プロジェクト・パスに組み込むには、次のようにします。

1. プロジェクト・エクスプローラーで、他のプロジェクトへリンクさせたいプロジェクトを右クリックし、「プロパティー (Properties)」をクリックする。
2. 「EGL ビルド・パス」プロパティー・ページを選択する。
3. ワークスペース内の他のすべてのプロジェクトのリストが「Projects (プロジェクト)」タブに表示される。参照するプロジェクトの横にあるチェック・ボックスをそれぞれクリックする。
4. プロジェクトを異なる順序で配置したりエクスポートするには、「順序およびエクスポート」タブをクリックし、以下を実行する。
 - ビルド・パスの順序でプロジェクトの位置を変更するには、プロジェクトを選択し、「上へ」と「下へ」ボタンを押す。

- プロジェクトをエクスポートするには、関連するチェック・ボックスを選択する。すべてのプロジェクトを一度に処理するには、「**全選択**」または「**全選択解除**」ボタンをクリックします。

5. 「**OK**」をクリックする。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 24 ページの『パーツの参照』
- 36 ページの『インポート』
- 19 ページの『パーツ』

関連する参照項目

- 517 ページの『EGL ビルド・パスおよび eglpath』
- 24 ページの『パーツの参照』
- 36 ページの『インポート』
- 19 ページの『パーツ』

EGL 出力の生成、準備、および実行

生成

生成とは、EGL パーツからの出力の作成です。

ワークベンチで出力を生成するには、ワークベンチ・バッチ・インターフェースまたは EGL ソフトウェア開発キット (EGL SDK) を使用します。EGL SDK は、ワークベンチから独立したファイル・ベースの生成のためのバッチ・インターフェースを提供します。

生成では EGL ファイルの保管版を使用します。

関連する概念

9 ページの『開発過程』

572 ページの『生成される出力』

関連するタスク

327 ページの『Java ラッパーの生成』

関連する参照項目

415 ページの『ビルド記述子オプション』

573 ページの『生成される出力 (参照)』

プロジェクトへの Java コードの生成

Java プログラムまたはラッパーを生成する場合は、ビルド記述子オプション **genProject** を設定し、プロジェクトに生成することをお勧めします (場合によっては必須です)。

プロジェクトへ生成すると、EGL からさまざまなサービスが提供されます。サービスはプロジェクト・タイプによって異なり、次に実行するタスクもそれによって異なります。

アプリケーション・クライアント・プロジェクト

アプリケーション・クライアント・プロジェクトに生成すると、EGL は以下のことを実行します。

- 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイル (fda6.jar および fdaj6.jar) への準備時アクセスを可能にする。

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar  
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

各エントリーの先頭の変数については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

- 以下のように、EGL JAR ファイルへのランタイム・アクセスを可能にする。

- アプリケーション・クライアント・プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトに JAR ファイルをインポートする
- アプリケーション・クライアント・プロジェクトのマニフェストを更新し、エンタープライズ・アプリケーション・プロジェクトの JAR ファイルを使用可能にする
- ランタイム値をデプロイメント記述子に書き込み、生成される J2EE 環境ファイルから記入項目を切り取りおよび貼り付けしなくてよいようにする。これについては、『デプロイメント記述子値の設定』を参照してください。

次に実行するタスクは以下のとおりです。

1. 生成されたプログラムを TCP/IP 経由で呼び出す場合は、『TCP/IP リスナーの設定』で説明されているように、リスナーへのランタイム・アクセスを可能にする。
2. 非 EGL JAR ファイルへのアクセスを提供する。
3. プロジェクトに出力ファイルを置いたので、次に J2EE ランタイム環境の設定を実行する。

EJB プロジェクト

EJB プロジェクトに生成すると、EGL は以下のことを実行します。

- 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイル (fda6.jar および fdaj6.jar) への準備時アクセスを可能にする。

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

各エントリーの先頭の環境変数については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

- 以下のように、EGL JAR ファイルへのランタイム・アクセスを可能にする。
 - EJB プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトに fda6.jar および fdaj6.jar ファイルをインポートする
 - EJB プロジェクトのマニフェストを更新し、エンタープライズ・アプリケーション・プロジェクトの fda6.jar および fdaj6.jar ファイルを実行時に使用可能にする
- JNDI 名を自動的に割り当て、EGL ランタイム・コードが EJB コードにアクセスできるようにする。ただし、このステップが実行されるのは、EJB セッション Bean を生成する場合のみです。
- ほとんどの場合、ランタイム値をデプロイメント記述子に書き込み、生成される J2EE 環境ファイルから記入項目を切り取りおよび貼り付けしなくてよいようにする。これについては、『デプロイメント記述子値の設定』を参照してください。

EGL が必要なセッションエレメントをデプロイメント記述子から検出できない場合、EGL はデプロイメント記述子にランタイム値を書き込みません。この状況は、例えば、Java プログラムがラッパーより先に生成された場合や、ビルド記述子オプション **sessionBeanID** がデプロイメント記述子内にない値に設定されている場合に生じます。セッションエレメントの詳細については、『*sessionBeanID*』を参照してください。

次に実行するタスクは以下のとおりです。

1. 非 EGL JAR ファイルへのアクセスを提供する。
2. デプロイメント・コードを生成する。
3. プロジェクトに出力ファイルを置いたので、次に J2EE ランタイム環境の設定を実行する。

J2EE Web プロジェクト

EGL は以下のことを実行します。

- プロジェクトの WebContent/WEB-INF/lib フォルダーに fda6.jar および fdaj6.jar をインポートし、EGL JAR ファイルへのアクセスを可能にする
- ランタイム値をデプロイメント記述子に書き込み、生成される J2EE 環境ファイルから記入項目を切り取りおよび貼り付けしなくてよいようにする。これについては、『デプロイメント記述子値の設定』を参照してください。

次に実行するタスクは以下のとおりです。

1. 非 EGL JAR ファイルへのアクセスの提供
2. プロジェクトに出力ファイルを置いたので、「EGL 生成コード用の J2EE ランタイム環境の設定」の説明に従って作業を進める。

Java プロジェクト

デバッグまたは実動用に非 J2EE Java プロジェクト内に生成している場合、EGL は以下のように動作します。

- 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイル (fda6.jar および fdaj6.jar) へのアクセスを可能にする。

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar  
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

各エントリーの先頭の変数については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

- プロパティー・ファイルを生成する。ただし、ビルド記述子に以下のオプション値が含まれている場合のみです。
 - **genProperties** が GLOBAL または PROGRAM に設定されている。
 - **J2EE** が NO に設定されている。

グローバル・プロパティー・ファイル (**rununit.properties**) を要求した場合、EGL はそのファイルを Java ソース・フォルダーに入れます。このフォルダーは、Java パッケージが入っているフォルダーです。(Java ソース・フォルダーは、プロジェクト内のフォルダーであるか、プロジェクト自体であることも考えられます。) 代わりにプログラム・プロパティー・ファイルを要求した場合、EGL はそのファイルをプログラムと一緒に配置します。

実行時に、プログラム・プロパティー・ファイルを使用して標準 JDBC 接続が設定されます。詳細については、『標準 JDBC 接続の作成方法について』を参照してください。

プロジェクトに出力ファイルを置いたので、以下のことを実行します。

- プログラムがリレーショナル・データベースにアクセスする場合は、ドライバがインストールされているディレクトリーが Java のビルド・パスに含まれていることを確認する。例えば、DB2 の場合は、db2java.zip を含むディレクトリーを指定します。
- コードが MQSeries にアクセスしている場合は、非 EGL JAR ファイルへのアクセスを提供する。
- リンケージ・プロパティー・ファイルをモジュールに置く。

存在しないプロジェクトに生成した場合の結果については、『*genProject*』を参照してください。

関連するタスク

- 368 ページの『EJB プロジェクトのデプロイメント・コードの生成』
- 395 ページの『リンケージ・プロパティー・ファイルのデプロイ』
- 386 ページの『デプロイメント記述子値の設定』
- 397 ページの『非 EGL JAR ファイルへのアクセスの提供』
- 368 ページの『変数 EGL_GENERATORS_PLUGINDIR の設定』
- 384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』
- 283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

- 427 ページの『*genProject*』
- 433 ページの『*sessionBeanID*』

ビルド

EGL または EGL Web プロジェクトで作業している場合、ビルド という用語は (通常は) コード生成を意味しません。

以下のメニュー・オプションは、独特な意味を持っています。

Build project (プロジェクトのビルド)

以下のようにして、プロジェクトのサブセットをビルドします。

1. 最後のビルド後にプロジェクト内で変更したすべての EGL ファイルを検証します
2. 以前のページ・ハンドラーの生成後に変更されたページ・ハンドラーを生成します
3. 最終コンパイル後に変更されたすべての Java ソースをコンパイルします。

メニュー・オプション「プロジェクトのビルド」は、ワークベンチ設定の「リソース変更時にビルドを自動的に実行する」を設定していない場合のみ使用可能です。この設定を指定した 場合は、EGL ファイルを保管するたびに前述のアクションが発生します。

すべてをビルド

「プロジェクトのビルド」と同じアクションを実行しますが、ワークスペース内のすべてのオープン・プロジェクトに対してアクションが実行される点が異なります。

Rebuild project (プロジェクトの再ビルド)

以下のように機能します。

1. プロジェクト内のすべての EGL ファイルを検証します
2. プロジェクト内のすべてのページ・ハンドラーを生成します
3. 最終コンパイル後に変更されたすべての Java ソースをコンパイルします。

Rebuild all (すべて再ビルド)

「プロジェクトの再ビルド」と同じアクションを実行しますが、ワークスペース内のすべてのオープン・プロジェクトに対してアクションが実行される点が異なります。

コードをプロジェクトに生成する場合、Java のコンパイルは、以下の状況ではローカルで発生します。

- プロジェクトをビルドまたは再ビルドするとき
- ソース・ファイルを生成するとき (ただし、ワークベンチ設定「リソース変更時にビルドを自動的に実行」をチェックした場合のみ)

コードをディレクトリーへ生成する場合、EGL は、以下の詳細を含む XML ファイル、ビルド計画 をオプションで作成します。

- 別のマシンへ転送されるファイルのロケーション
- TCP/IP 経由での転送に必要なその他の情報
- Java コンパイル文

リモート・プラットフォームで生成済み出力を準備するには、ビルド・サーバーがそのプラットフォーム上で稼働していることが必要です。

ビルド計画を作成し、時間が経過してからその計画を呼び出すこともできます。詳細については、『生成後のビルド計画の呼び出し』を参照してください。

関連する概念

319 ページの『ビルド記述子パーツ』

356 ページの『ビルド計画』

373 ページの『ビルド・サーバー』

9 ページの『開発過程』

関連するタスク

319 ページの『ビルド・ファイルの作成』

363 ページの『生成後のビルド計画の呼び出し』

関連する参照項目

415 ページの『ビルド記述子オプション』

EGL 出力のビルド

Java プログラムまたは Java ラッパーの EGL 出力をビルドするには、以下の手順を完了します。

1. プロジェクトまたはディレクトリーの中に Java ソース・コードを生成する。

- プロジェクトの中に生成 (推奨) し、Eclipse 設定がリソース変更時にビルドを自動的に実行するように設定されている場合は、ワークベンチが出力を準備します。
 - ディレクトリーの中に生成する場合は、生成プログラムの分散ビルド機能が出力を準備します。
2. 生成出力を準備する。このステップは、ビルド記述子オプション **buildPlan** または **prep** を no に設定していない場合に自動的に行われます。

関連する概念

354 ページの『ビルド』

9 ページの『開発過程』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

351 ページの『プロジェクトへの Java コードの生成』

572 ページの『生成される出力』

351 ページの『生成』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

364 ページの『ディレクトリーへ生成される Java コードの処理』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

573 ページの『生成される出力 (参照)』

ビルド計画

ビルド計画は、準備時に以下の詳細を使用できるようにするための XML ファイルです。

- ビルド・マシンで処理する必要のあるファイル
- これらを処理するのに必要なビルド・スクリプト
- 出力の配置場所

ビルド計画は、開発プラットフォーム上に置かれ、ビルド・クライアントに対してすべてのビルド・ステップを指示します。ステップごとに、ビルド・サーバーの要求が作成されます。

EGL では、Java プログラムまたはラッパーを生成すると、ビルド記述子オプション **buildPlan** を NO に設定した場合を除き、必ずビルド計画が生成されます。

ビルド計画の名前の詳細については、『生成される出力 (参照)』を参照してください。

関連する概念

372 ページの『ビルド・スクリプト』

572 ページの『生成される出力』

関連する参照項目

419 ページの『buildPlan』

573 ページの『生成される出力 (参照)』

Java プログラム、ページ・ハンドラー、およびライブラリー

プログラム・パーツを Java プログラムとして生成する場合、あるいはページ・ハンドラーまたは Java 関連ライブラリー・パーツを生成する場合、EGL では以下のそれぞれについてクラスとファイルが作成されます。

- プログラム、ページ・ハンドラー、またはライブラリー・パーツ
- そのパーツ自体で宣言されるか、そのパーツにより直接または間接的に呼び出される関数のいずれかで宣言される各レコード
- 使用されるデータ・テーブル、書式グループ、および書式ごと

クラス名の詳細については、『Java プログラム生成の出力』を参照してください。

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

573 ページの『生成される出力 (参照)』

727 ページの『Java プログラム生成の出力』

結果ファイル

結果ファイルには、ターゲット環境で実行されたコード準備ステップに関する状況情報が含まれています。このファイルは EGL によって生成される出力の準備が行われた場合にのみ生成されます。

以下のビルド記述子オプションを使用してディレクトリーへの生成を実行すると、自動的に準備が行われます。

- **prep** を YES に設定する
- **buildPlan** を YES に設定する

結果ファイルの名前の詳細、、『生成される出力 (参照)』を参照してください。

関連する概念

319 ページの『ビルド記述子パーツ』

572 ページの『生成される出力』

関連するタスク

364 ページの『ディレクトリーへ生成される Java コードの処理』

関連する参照項目

573 ページの『生成される出力 (参照)』

419 ページの『buildPlan』

432 ページの『prep』

ワークベンチでの生成方法

ワークベンチで生成するには、生成ウィザードまたは生成メニュー項目を使用します。生成メニュー項目を選択すると、EGL は、デフォルトのビルド記述子を使用します。デフォルトのビルド記述子を選択していない場合は、生成ウィザードを使用してください。デフォルトのビルド記述子の選択に関する詳細については、『デフォルトのビルド記述子の設定』を参照してください。

生成ウィザードを起動してワークベンチで生成するには、次のようにします。

1. プロジェクト・エクスプローラーでリソース名 (プロジェクト、フォルダー、またはファイル) を右クリックする。
2. 「ウィザードを使用して生成...」オプションを選択する。

生成ウィザードは、以下の 4 つのウィザード・ページから構成されます。

1. 先頭ページは、生成プロセスを開始するために選択した内容に基づいて生成されるパーツのリストを表示します。次ページへ進んで続行するためには、その前にこのリストで少なくとも 1 つのパーツを選択する必要があります。インターフェースは、リスト内のすべてのパーツを選択または選択解除できるようにするボタンを提供しています。
2. 2 番目のページは、先頭ページで選択したパーツを生成するために使用するビルド記述子を 1 つまたは複数選択できるようにしています。以下の 2 つのオプションが用意されています。
 - ワークスペース内にあるすべてのビルド記述子のドロップダウン・リストから 1 つのビルド記述子を選択し、すべてのパーツを生成するためにそのビルド記述子を使用する。
 - 先頭ページで選択したパーツごとにビルド記述子を選択する。パーツごとにビルド記述子を選択するには、テーブルを使用します。テーブルの最初の列にパーツ名が表示され、2 列目にはパーツごとのビルド記述子のドロップダウン・リストが表示されます。
3. 3 番目のページは、生成プロセスでユーザー ID およびパスワードが必要な場合に、宛先マシンと SQL データベースの両方で使用するユーザー ID およびパスワードを設定できるようにする。ここでユーザー ID およびパスワードを設定すると、生成されるパーツごとに指定されたビルド記述子内にリストされるユーザー ID およびパスワードがオーバーライドされます。永続的なストレージに機密情報を保持することを回避するために、ユーザー ID およびパスワードをビルド記述子ではなくこのページに設定することをお勧めします。
4. 4 番目のページは、ワークベンチ外の EGL プログラムを生成するために使用できるコマンド・ファイルを作成できるようにする。コマンド・ファイルは、(コマンド EGLCMD を使用して) ワークベンチ・バッチ・インターフェースまたは (コマンド EGLSDK を使用して) EGL SDK で参照できます。

コマンド・ファイルを作成するには、次のようにします。

- a. 「コマンド・ファイルを作成」チェック・ボックスを選択する。
- b. 完全修飾パスを入力するか、「参照」をクリックして標準の Windows の手順を使用することによって出力ファイルの名前を指定する。

- c. 「EGL パス (eglp_{path}) を自動的に挿入」チェック・ボックスを選択またはクリアし、eglp_{path} の初期値としてコマンド・ファイルに EGL プロジェクト・パスを含めるかどうかを指定する。詳細については『EGL コマンド・ファイル』を参照してください。
 - d. コマンド・ファイル作成時に出力の生成を回避するかどうかを示すラジオ・ボタンを選択する。
5. 「完了」をクリックする。

生成メニュー項目を使用してワークベンチで生成するには、以下の手順のうち 1 つを行います。

1. プロジェクト・エクスプローラーで 1 つまたは複数のリソース名 (プロジェクト、フォルダー、またはファイル) を選択する。複数のリソース名を選択するには、**Ctrl** キーを押した状態でクリックします。
2. 右クリックしてから「生成」メニュー・オプションを選択する。

または

1. プロジェクト・エクスプローラーでリソース名 (プロジェクト、フォルダー、またはファイル) をダブルクリックする。EGL エディターでファイルが開きます。
2. エディター・ペイン内を右クリックし、「生成」を選択する。

関連する概念

『ワークベンチでの生成』

関連するタスク

122 ページの『デフォルトのビルド記述子の設定』

関連する参照項目

518 ページの『EGLCMD』

529 ページの『EGLSDK』

573 ページの『生成される出力 (参照)』

ワークベンチでの生成

ワークベンチで出力を生成するには、以下のタスクを実行します。

- 生成中に参照された任意のパーツとともに、生成するパーツをロードします。
- 生成するパーツを選択します。ファイル、フォルダー、パッケージ、またはプロジェクトの生成プロセスを呼び出すと、EGL は、ユーザーが選択したコンテナ内にあるすべての基本パーツ (すべてのプログラム、ページ・ハンドラー、書式グループ、データ・テーブル、またはライブラリー) の出力を作成します。
- 生成を開始します。
- 進行状況をモニターします。

生成を簡単にするには、まず最初に、以下のタイプからデフォルトのビルド記述子を選択することをお勧めします。

- デバッグ・ビルド記述子 (EGL デバッガーの使用中に適切)
- ターゲット・システムのビルド記述子 (ランタイム環境でのパーツの生成およびデプロイに使用)

デフォルトのビルド記述子の選択方法についての詳細は、『デフォルトのビルド記述子の設定』を参照してください。

それぞれの 2 つのビルド記述子 (デバッグおよびターゲット・システム) は、以下のようにして識別できます。

- ファイル、フォルダー、パッケージ、またはプロジェクトのレベルにあるプロパティー
- ワークベンチ設定

特定の種類の低レベルのビルド記述子が、同じ種類の高位のビルド記述子よりも優先されます。例えば、現行パッケージに割り当てられたターゲット・システム・ビルド記述子は、プロジェクトに割り当てられたターゲット・システムのビルド記述子よりも優先されます。ただし、マスター・ビルド記述子は他のすべての記述子よりも優先されます (『ビルド記述子パーツ』を参照)。

以下の優先順位規則が基本パーツを含むすべてのファイルに適用されます。

- ファイルに固有のプロパティーは、他のすべてのプロパティーよりも優先される
- 関連したフォルダー・プロパティーは、パッケージ、プロジェクト、またはワークベンチのプロパティーよりも優先される
- 関連したパッケージ・プロパティーは、プロジェクト、またはワークベンチのプロパティーよりも優先される
- 関連したプロジェクト・プロパティーは、ワークベンチ・プロパティーよりも優先される
- 何のプロパティーも指定されていない場合は、ワークベンチ・プロパティーが使用される

生成の開始方法については、『ワークベンチでの生成』を参照してください。

関連する概念

319 ページの『ビルド記述子パーツ』

9 ページの『開発過程』

572 ページの『生成される出力』

関連するタスク

358 ページの『ワークベンチでの生成方法』

122 ページの『デフォルトのビルド記述子の設定』

関連する参照項目

573 ページの『生成される出力 (参照)』

ワークベンチのバッチ・インターフェースからの生成

Workbench バッチ・インターフェースからの生成を行うには、次のようにします。

1. Java クラスパスが以下の JAR ファイルへのアクセスを提供していることを確認する。

- startup.jar。これは、次のディレクトリーにあります。

`installationDir\eclipse`

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\RSPD\6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

- `eglutil.jar`。これは、次のディレクトリーにあります。

```
installationDir\egl\ eclipse\plugins\
com.ibm.etools.egl.utilities_version\runtime
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\RSPD\6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

2. 生成に必要なプロジェクトおよび EGL パーツがワークスペースに入っていることを確認する。
3. EGL コマンド・ファイルを開発する。
4. 通常、コードの生成、実行、およびテストを行う大規模なバッチ・ジョブに存在するコマンド `EGLCMD` を起動する。 `EGLCMD` を起動する場合は、必要なワークスペースを指定します。

関連する概念

『ワークベンチ・バッチ・インターフェースからの生成』

関連する参照項目

518 ページの『`EGLCMD`』

521 ページの『EGL コマンド・ファイル』

ワークベンチ・バッチ・インターフェースからの生成

ワークベンチ・バッチ・インターフェースは、ワークベンチにアクセスできるバッチ環境から EGL 出力を生成できる機能です。ワークベンチを実行しておく必要はありません。EGL コードの生成は、ワークスペースに以前にロードされたプロジェクトおよび EGL パーツのみに アクセスできます。

インターフェースを起動するには、バッチ・コマンド `EGLCMD` を使用します。このバッチ・コマンドは、ワークスペースと EGL コマンド・ファイルの両方を参照します。

関連する概念

9 ページの『開発過程』

572 ページの『生成される出力』

関連するタスク

360 ページの『ワークベンチのバッチ・インターフェースからの生成』

EGL ソフトウェア開発キット (SDK) からの生成

EGL SDK から生成するには、次のようにします。

1. コードを生成するマシン上に Java 1.3.1 (またはこれ以降のレベル) があることを確認する。適切なレベルの Java コードは、EGL をインストールするマシンに自動的にインストールされます。生成マシンとターゲット・マシンの Java レベルが互換性がある必要があります。
2. eglbatchgen.jar が Java クラスパスに入っていることを確認する。JAR ファイルは次のディレクトリに存在します。

`installationDir¥bin`

`installationDir`

製品のインストール・ディレクトリ。たとえば、`C:¥Program Files¥IBM¥RSPD¥6.0` など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリを指定することが必要になる場合があります。

3. EGL SDK が生成に必要な EGL ファイルにアクセスできることを確認する。
4. オプションで、EGL コマンド・ファイルを開発する。
5. 通常、コードの生成、実行、およびテストを行う大規模なバッチ・ジョブに存在するコマンド EGLSDK を起動する。

関連する概念

『EGL ソフトウェア開発キット (SDK) からの生成』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連する参照項目

517 ページの『EGL ビルド・パスおよび eglpath』

518 ページの『EGLCMD』

521 ページの『EGL コマンド・ファイル』

529 ページの『EGLSDK』

EGL ソフトウェア開発キット (SDK) からの生成

EGL ソフトウェア開発キット (SDK) は、Rational Developer 製品の以下のものにアクセスできない場合でも、バッチ環境で出力を生成できるフィチャーです。

- グラフィカル・ユーザー・インターフェース
- プロジェクト編成方法の詳細

EGL SDK を使用すると、Rational ClearCase® などのソフトウェア構成管理 (SCM) ツールから、通常の作業時間の後に実行するバッチ・ジョブの一部などとして生成を起動できます。

EGL SDK を起動するには、バッチ・ファイルまたはコマンド・プロンプトでコマンド EGLSDK を使用します。コマンド呼び出し自体は以下の 2 つの書式のいずれでも可能です。

- EGL ファイルおよびビルド記述子を指定できます。この場合、複数の生成を行うには複数のコマンドを記述します。
- 呼び出しで、1 回以上の生成を行うために必要な情報を含む EGL コマンド・ファイルを参照することもできます。

作業の方法にかかわらず、*eglp* に値を指定することにより、EGL SDK が import ステートメントを使用してパーツの参照を解決する場合に検索するディレクトリーのリストを指定できます。また、**genProject** の代わりにビルド記述子オプション **genDirectory** を指定する必要があります。

EGLSDK を使用するための前提条件およびプロセスについては、『EGL SDK からの生成方法』を参照してください。コマンド呼び出しに関する詳細については、『EGLSDK』を参照してください。

関連する概念

9 ページの『開発過程』

572 ページの『生成される出力』

関連するタスク

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

関連する参照項目

425 ページの『genDirectory』

518 ページの『EGLCMD』

517 ページの『EGL ビルド・パスおよび *eglp*』

529 ページの『EGLSDK』

生成後のビルド計画の呼び出し

ビルド計画を作成し、時間が経過してからその計画を呼び出したい場合があります。たとえば、生成時のネットワークの失敗によってリモートのマシンでコードの準備ができない場合があります。

このような場合にビルド計画を呼び出すには、以下の手順を完了します。

1. EGL のインストール先マシンで自動的に必要になるため、*eglb* が Java クラスパスにあることを確認します。JAR ファイルは次のディレクトリーに存在します。

```
installationDir%egl%eclipse%plugins%
com.ibm.etools.egl.batchgeneration_version
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\SPD6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

2. 同様に、PATH 変数にそのディレクトリーが含まれていることを確認します。

3. コマンド行から次のコマンドを入力します。

```
java com.ibm.etools.egl.distributedbuild.BuildPlanLauncher bp
```

bp ビルド計画ファイルの完全修飾パスです。生成されたファイルの名前の詳細については、『生成される出力 (参照)』を参照してください。

関連する概念

356 ページの『ビルド計画』

351 ページの『生成』

関連するタスク

355 ページの『EGL 出力のビルド』

関連する参照項目

415 ページの『ビルド記述子オプション』

573 ページの『生成される出力 (参照)』

Java の生成、その他トピック

ディレクトリーへ生成される Java コードの処理

このページでは、ディレクトリーへ生成されるコードの処理方法について説明します。ただし、コードをディレクトリーの中へ生成することは回避するようお勧めします。詳細については、『プロジェクトへの Java コードの生成』を参照してください。

コードをディレクトリーへ生成するには、ビルド記述子オプション **genDirectory** を指定し、ビルド記述子オプション **genProject** の指定は回避するようにします。

次に実行するタスクは、プロジェクト・タイプによって異なります。

アプリケーション・クライアント・プロジェクト

アプリケーション・クライアント・プロジェクトに対しては、次のようにします。

1. 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイルへの準備時アクセスを提供する。

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar  
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

各エントリーの先頭の変数の詳細については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

2. 以下のように、fda6.jar、fdaj6.jar、および (TCP/IP 経由で生成したプログラムを呼び出す場合は) EGLTcpipListener.jar へのランタイム・アクセスを提供する。
 - 以下のディレクトリーから JAR ファイルにアクセスする。

```
installationDir%egl%eclipse%plugins%  
com.ibm.etools.egl.generators_version%runtime
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\RSPD\6.0 など。これから使用しようとしている製品をイン

ストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

アプリケーション・クライアント・プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトにこれらのファイルをコピーします。

- アプリケーション・クライアント・プロジェクトのマニフェストを更新し、JAR ファイル (エンタープライズ・アプリケーション・プロジェクトに保管したもの) を使用可能にする。
3. 非 EGL JAR ファイルへのアクセスを提供する (オプション・タスク)。
 4. 以下の規則に従って、生成される出力をプロジェクトへインポートする。
 - フォルダー *appClientModule* には、生成される出力が入っているパッケージのトップレベルのフォルダーを組み込む必要がある。
 - *appClientModule* の下のフォルダー名の階層が、Java パッケージの名前と一致している

たとえば、生成される出力をパッケージ *my.trial.package* からインポートする場合は、以下のロケーションに常駐するフォルダーにその出力をインポートする必要があります。

appClientModule/my/trial/package

5. J2EE 環境ファイルを生成した場合は、そのファイルを更新する。
6. デプロイメント記述子を更新する。
7. プロジェクトに出力ファイルを置いたので、次に J2EE ランタイム環境のセットアップを続行する。

EJB プロジェクト

EJB プロジェクトに対しては、次のようにします。

1. 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイル (*fda6.jar* および *fdaj6.jar*) への準備時アクセスを提供する。

EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar

各エントリーの先頭の変数の詳細については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

2. 以下のように、EGL JAR ファイルへのランタイム・アクセスを提供にする。
 - 以下のディレクトリーから *fda6.jar* および *fdaj6.jar* にアクセスする。

installationDir¥egl¥eclipse¥plugins¥
com.ibm.etools.egl.generators_version¥runtime

installationDir

製品のインストール・ディレクトリー。たとえば、*C:¥Program Files¥IBM¥RSPD¥6.0* など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持し

ていた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

EJB プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトにこれらのファイルをコピーします。

- EJB プロジェクトのマニフェストを更新し、`fda6.jar` および `fdaj6.jar` ファイル (エンタープライズ・アプリケーション・プロジェクトに保管したもの) を使用可能にする。
3. 非 EGL JAR ファイルへのアクセスを提供する (オプション・タスク)。
 4. 以下の規則に従って、生成される出力をプロジェクトへインポートする。
 - フォルダ `ejbModule` には、生成される出力が入っているパッケージのトップレベルのフォルダを組み込む必要がある。
 - `ejbModule` の下のフォルダ名の階層が、Java パッケージの名前と一致している必要がある。

たとえば、生成される出力をパッケージ `my.trial.package` からインポートする場合は、以下のロケーションに常駐するフォルダにその出力をインポートする必要があります。

`ejbModule/my/trial/package`

5. J2EE 環境ファイルを生成した場合は、そのファイルを更新する。
6. デプロイメント記述子を更新する。
7. JNDI 名を設定する。
8. デプロイメント・コードを生成する。
9. プロジェクトに出力ファイルを置いたので、次に J2EE ランタイム環境のセットアップを続行する。

J2EE Web プロジェクト

Web プロジェクトに対しては、次のようにします。

1. Web プロジェクト・フォルダに `fda6.jar` および `fdaj6.jar` をコピーし、EGL JAR ファイルへのアクセスを提供する。これを行うには、以下のディレクトリーにある外部 jar をインポートします。

```
installationDir%egl%eclipse%plugins%  
com.ibm.etools.egl.generators_version%runtime
```

installationDir

製品のインストール・ディレクトリー。たとえば、`C:\Program Files\IBM\RSPPD\6.0` など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

ファイルの宛先は、以下のプロジェクト・フォルダです。

`WebContent/WEB-INF/lib`

2. 非 EGL JAR ファイルへのアクセスを提供する (オプション)。
3. 以下の規則に従って、生成される出力をプロジェクトへインポートする。
 - フォルダー *WebContent* には、生成される出力が入っているパッケージのトップレベルのフォルダーを組み込む必要がある。
 - *WebContent* の下のフォルダー名の階層が、Java パッケージの名前と一致している必要がある。

たとえば、生成される出力をパッケージ *my.trial.package* からインポートする場合は、以下のロケーションに常駐するフォルダーにその出力をインポートする必要があります。

```
WebContent/my/trial/package
```

4. デプロイメント記述子を更新する。
5. プロジェクトに出力ファイルを置いたので、次に J2EE ランタイム環境のセットアップを続行する。

Java プロジェクト

非 J2EE 環境で使用するコードの生成中に、次の組み合わせのビルド記述子オプションを使用するのであれば、プロパティー・ファイルを生成します。

- **genProperties** が GLOBAL または PROGRAM に設定されている。
- **J2EE** が NO に設定されている。

グローバル・プロパティー・ファイル (**rununit.properties**) を要求した場合、EGL はそのファイルをトップレベル・ディレクトリーに入れます。代わりにプログラム・プロパティー・ファイルを要求した場合、EGL はそのファイルをプログラムと一緒に、パッケージ名の最後の修飾子に対応するフォルダーに入れるか、トップレベル・ディレクトリーに入れます。(トップレベル・ディレクトリーが使用されるのは、EGL ソース・ファイルの中でパッケージ名が指定されなかった場合です。)

実行時に、プログラム・プロパティー・ファイル内の値を使用して、標準 JDBC 接続がセットアップされます。詳細については、『標準 JDBC 接続の作成方法について』を参照してください。

Java プロジェクトの場合、作業は次のようになります。

1. 以下のエントリーをプロジェクトの Java のビルド・パスに追加することによって、EGL JAR ファイルへのアクセスを提供する。

```
EGL_GENERATORS_PLUGINDIR/runtime/fda6.jar  
EGL_GENERATORS_PLUGINDIR/runtime/fdaj6.jar
```

各エントリーの先頭の変数の詳細については、『変数 *EGL_GENERATORS_PLUGINDIR* の設定』を参照してください。

2. プログラムがリレーショナル・データベースにアクセスする場合は、ドライバーがインストールされているディレクトリーが Java のビルド・パスに含まれていることを確認する。たとえば、DB2 の場合は、db2java.zip が入っているディレクトリーを指定します。
3. 生成されるコードが MQSeries にアクセスする場合は、非 EGL JAR ファイルへのアクセスを提供する。
4. プログラム・プロパティー・ファイル (存在する場合) がトップレベル・プロジェクト・フォルダーに入っており、グローバル・プロパティー・ファイ

ル (存在する場合は `rununit.properties`) が、パッケージ名の最後の修飾子に対応するフォルダーかトップレベル・プロジェクト・フォルダーに入っていることを確認する。(トップレベル・フォルダーが使用されるのは、EGL ソース・ファイルの中でパッケージ名が指定されなかった場合です。)

5. リンケージ・プロパティ・ファイルをプロジェクトに置く (オプション・タスク)。

関連する概念

351 ページの『プロジェクトへの Java コードの生成』

関連するタスク

『EJB プロジェクトのデプロイメント・コードの生成』

395 ページの『リンケージ・プロパティ・ファイルのデプロイ』

386 ページの『デプロイメント記述子値の設定』

397 ページの『非 EGL JAR ファイルへのアクセスの提供』

389 ページの『EJB プロジェクトの JNDI 名の設定』

『変数 `EGL_GENERATORS_PLUGINDIR` の設定』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

388 ページの『デプロイメント記述子の手操作による更新』

387 ページの『J2EE 環境ファイルの更新』

関連する参照項目

425 ページの『`genDirectory`』

427 ページの『`genProject`』

EJB プロジェクトのデプロイメント・コードの生成

EJB プロジェクトへの生成を行い、デプロイメント記述子のプロパティを指定すれば、EJB のリモート・アクセスを可能にするスタブおよびスケルトンを生成できます。

1. プロジェクト・エクスプローラーで、プロジェクト名を右クリックし、次に「**デプロイ**」をクリックする。
2. 『ワークベンチからの EJB デプロイメント・コードの生成』のヘルプ・ページに指定されている指示に従う。

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

変数 `EGL_GENERATORS_PLUGINDIR` の設定

ワークベンチ・クラスパス変数 `EGL_GENERATORS_PLUGINDIR` には、ワークベンチ内の EGL プラグインへの完全修飾パスが含まれています。この変数は、EGL プログラムを Java タイプのプロジェクト、アプリケーション・クライアント、または EJB へ生成する場合に、Java のビルド・パス内で使用します。

`EGL_GENERATORS_PLUGINDIR` を参照するクラスパス・エラーが発生すると、変数が設定されない場合があります。たとえば、EGL パーツで作業を行う前に、

Concurrent Versions System (CVS) のようなソフトウェア構成管理システムから EGL 関連のプロジェクトをチェックアウトすると、問題が発生します。

この変数は、EGL パーツの作成、EGL コードの生成、または以下の手順の実行によって設定することができます。

1. 「ウィンドウ」、「設定」の順に選択します。
2. 「設定」ページで、「Java」、「Classpath Variables (クラスパス変数)」の順に選択します。
3. 「新規...」を選択します。
4. 「New Variable Entry (新規変数エントリ)」ページで、**EGL_GENERATORS_PLUGINDIR** と入力して、次のディレクトリーを指定します。

```
installationDir%egl%eclipse%plugins%  
com.ibm.etools.egl.generators_version
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\RSPP\6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

変数を設定した後で、プロジェクトを再ビルドします。

関連する概念

351 ページの『プロジェクトへの Java コードの生成』

関連する参照項目

427 ページの『genProject』

ローカル・マシンでの EGL 生成 Java コードの実行

ローカル・マシンでの基本またはテキスト・ユーザー・インターフェース Java アプリケーションの開始

EGL で生成された基本 (バッチ) またはテキスト・ユーザー・インターフェース (TUI) Java アプリケーションをローカル・マシン上で開始するには、次のようにします。

1. EGL ソース・コードから Java ソース・コードを生成します。詳細については、『ワークベンチでの生成』を参照してください。
2. プロジェクト・エクスプローラーで、「JavaSource」フォルダーを展開し、実行したいアプリケーションの Java ソース・ファイルを選択します。
3. 「ワークベンチ (Workbench)」メニューで、「実行」>「次を実行」>「Java アプリケーション」を選択します。または、ワークベンチ・ツールバーで「実行」ボタンの隣にある下矢印をクリックしてから、「次を実行」>「Java アプリケーション」を選択します。

関連する概念

359 ページの『ワークベンチでの生成』

関連するタスク

358 ページの『ワークベンチでの生成方法』

ローカル・マシンでの Web アプリケーションの開始

JNDI データ・ソースにアクセスする EGL ベースの Web アプリケーションで作業する場合は、現行のトピックの説明に従うことはできません。ただし、以前に Web アプリケーション・サーバーを構成してある場合は除きます。WebSphere 固有の背景情報については、『*WebSphere Application Server と EGL*』を参照してください。

Web アプリケーションを開始するには、以下の手順を実行します。

1. EGL ソース・コードから Java ソース・コードを生成します。詳細については、『ワークベンチでの生成』を参照してください。
2. プロジェクト・エクスプローラーで、**WebContent** および **WEB-INF** フォルダを展開する。実行したい JSP を右クリックした後、「実行」>「サーバーで実行」を選択します。「サーバーの選択」ダイアログが表示されます。
3. すでにこの Web プロジェクト用にサーバーを構成してある場合は、「既存のサーバーを選択」を選択し、リストからサーバーを選択します。「完了」をクリックして、サーバーを開始し、サーバーへアプリケーションをデプロイし、アプリケーションを開始します。
4. この Web プロジェクト用にサーバーを構成していない場合は、以下のように作業を進めることができます。ただし、アプリケーションが JNDI データ・ソースにアクセスしない場合に限りです。
 - a. 「手操作でサーバーを定義」を選択します。
 - b. ホスト名を指定します。これは、(ローカル・マシンの場合) **localhost** です。
 - c. 実行時にアプリケーションをデプロイする予定の Web アプリケーション・サーバーに類似のサーバーについて、サーバー型の選択を行います。選択項目には、「**WebSphere v5.1 テスト環境**」と「**WebSphere v6.0 サーバー (WebSphere v6.0 Server)**」があります。
 - d. 現行プロジェクトの作業時に選択項目を変更する予定がない場合は、「**プロジェクトのデフォルトとしてサーバーを設定**」のチェック・ボックスを選択します。
 - e. ほとんどの場合、このステップは回避できますが、デフォルトと異なる設定を指定したい場合は、「次へ」をクリックして選択を行ってください。
 - f. 「完了」をクリックして、サーバーを開始し、サーバーへアプリケーションをデプロイし、アプリケーションを開始します。

関連する概念

359 ページの『ワークベンチでの生成』

371 ページの『*WebSphere Application Server と EGL*』

199 ページの『Web サポート』

関連するタスク

358 ページの『ワークベンチでの生成方法』

WebSphere Application Server と EGL

EGL で書かれた、ワークベンチ内の J2EE アプリケーションを実行するかデバッグするときは、多くの場合、次のいずれかの IBM ランタイム環境を使用します。

- WebSphere v5.1 テスト環境。これは、Java サブレット・バージョン 2.3 (および、それ以降) と EJB バージョン 2.0 (および、それ以降) をサポートします。
- WebSphere Application Server v6.0。これは、Java サブレット・バージョン 2.4 (および、それ以降) と EJB バージョン 2.1 (および、それ以降) をサポートします。

J2EE データ・ソースを使用しないコードをデバッグするか実行する場合、これら 2 つの環境でコードを実行するプロセスは、ほとんど同じで、2、3 回のマウス・クリックだけで済みます。

しかし、J2EE データ・ソースにアクセスする必要がある場合は、状態は次のとおりです。

- WebSphere v5.1 テスト環境で作業をしている場合は、次の 2 つのステップを任意の順序で実行します。
 1. サーバー構成を定義するときに、データ・ソースを識別します。
 2. アプリケーションが、そのデータ・ソースのサーバー構成エントリを確実に参照するようにします。

この 2 番目のステップには、プロジェクトに固有のデプロイメント記述子内で JNDI 名を指定することも含まれます。JNDI 名は、次のいずれかの時点で指定します。

- プロジェクトを作成するとき、または
- デプロイメント記述子を更新するとき

サーバー構成の詳細については、『*WebSphere Application Server v5.x の構成 (Configuring WebSphere Application Server v5.x)*』を参照してください。

- WebSphere Application Server v6.0 で作業をしている場合は、次の 2 つのステップを任意の順序で実行します。
 1. 次のいずれか可能な方法で、サーバーに対してデータ・ソースを識別します。
 - 推奨されるように、アプリケーション・デプロイメント記述子 (application.xml) を更新するとき、または
 - 管理コンソールでサーバーを構成するとき

アプリケーション・デプロイメント記述子の更新方法の詳細については、『*WebSphere Application Server v6.0 用のデータ・ソースをテストするためのサーバーのセットアップ (Setting up a server to test data sources for WebSphere Application Server v6.0)*』を参照してください。管理コンソールの使用方法の詳細については、『*WebSphere Application Server v6.x の構成 (Configuring WebSphere Application Server v6.x)*』を参照してください。

2. アプリケーションが、そのデータ・ソースのサーバー構成エントリを確実に参照するようにします。

この 2 番目のステップには、プロジェクトに固有のデプロイメント記述子内で JNDI 名を指定することも含まれます。JNDI 名は、次のいずれかの時点で指定します。

- プロジェクトを作成するとき、または
- デプロイメント記述子を更新するとき

管理コンソールで作業する代わりに、アプリケーション・デプロイメント記述子を更新することの利点は、次のとおりです。

- データ・ソースを識別するために必要な追加のサーバー構成を行わなくても、J2EE バージョン 1.4 をサポートする任意の Web アプリケーション・サーバーにエンタープライズ・アプリケーションをデプロイすることができます。
- サーバーが稼働しているかどうかに関係なく、アプリケーション・デプロイメント記述子を更新できます。
- アクションが、WebSphere Application Server コンポーネント内でなく、Rational Developer 製品の開発コンポーネント内にあるので、利便性があります。

データ・ソース情報をどのように更新しても、サーバーは変更をほとんど即時に使用できます。

関連する概念

199 ページの『Web サポート』

ビルド・スクリプト

ビルド・スクリプトはビルド計画によって呼び出されて、生成されたファイルから出力するための準備を行うファイルです。以下に例を示します。

- Java コンパイラー、またはその他の .exe (バイナリー) ファイルや .bat (テキスト) ファイルは、開発システム上のビルド・サーバーが使用できるか、リモートの Windows 2000/NT/XP 上のビルド・サーバーに送信されます。
- スクリプト (.scr file) またはいくつかのバイナリー・コードは、USS ビルド・サーバーに送信されます。

ビルド記述子オプション **destHost** を設定して、ビルド・マシンのアドレスを指定します。

Java ビルド・スクリプト

Java コードを実行に向けて準備するため、EGL では javac (Java コンパイラー) コマンドおよびそのパラメーターをビルド計画に含めてビルド・マシンに送信します。ビルド・マシンでは、javac コマンドとコマンドに必要な入力を使用されます。

関連する概念

354 ページの『ビルド』

356 ページの『ビルド計画』

373 ページの『ビルド・サーバー』

関連する参照項目

415 ページの『ビルド記述子オプション』

421 ページの『destDirectory』
422 ページの『destHost』
423 ページの『destPassword』
423 ページの『destUserID』

727 ページの『Java プログラム生成の出力』
728 ページの『Java ラッパー生成の出力』

ビルド・サーバー

ビルド・サーバーではクライアント・システムからの要求を受け、そのクライアントから送信されたソース・コードを元にして実行可能ファイルが作成されます。ビルド・サーバーはビルド・クライアントから要求が送信される前に開始されている必要があります。通常、ビルド・サーバーは複数のクライアントからの要求を処理します。同時ビルド要求が受信されると、複数のスレッドが開始される場合があります。

生成プログラム環境では、オペレーティング・システムが、例えば、Windows 2000 などのターゲット生成システムであるマシン上でビルド・サーバーを始動します。生成プログラムによって、Java ソース・コードが生成されます。指定したビルド・サーバーに Java コードが送信され、そのサーバーで Java コンパイラーが起動されます。

Windows 用の Java コードを生成する場合は、生成が実行されたマシンと同じマシンで Java 出力をビルドできます。これはローカル・ビルドと呼ばれます。この場合、ビルド・サーバーを始動する必要はありません。ローカル・ビルドを実行する場合は、ビルド記述子から **destHost** オプションを省略してください。

関連する概念

354 ページの『ビルド』
372 ページの『ビルド・スクリプト』

関連するタスク

『AIX、Linux、または Windows 2000/NT/XP 上でのビルド・サーバーの開始』

関連する参照項目

415 ページの『ビルド記述子オプション』

AIX、Linux、または Windows 2000/NT/XP 上でのビルド・サーバーの開始

AIX、Linux、または Windows 2000/NT/XP 上でリモート・ビルド・サーバーを開始するには、「Command Prompt (コマンド・プロンプト)」ウィンドウに ccublds コマンドを入力します。入力する構文は以下のとおりです。

```
►► ccublds -p portno -v -a 0 2 ◀◀
```


ここで、

- p サーバーがクライアントと連絡するために listen するポート番号 (*portno*) を指定します。
- V サーバーの冗長レベルを指定します。このパラメーターは、3 回まで (最大冗長レベル) 指定できます。
- a 以下のように認証モードを指定します。
 - 0 サーバーは、どのクライアントが要求するビルドでも実行します。このモードは、セキュリティが重要ではない環境の場合にのみお勧めします。
 - 2 サーバーは、クライアントのビルドを受け入れる前に、有効なユーザー ID およびパスワードを提供することをクライアントに要求します。この場合のユーザー ID およびパスワードは、ビルド・サーバーが稼働しているホスト・マシンの所有者によって最初に構成されたものです。この構成は、以下で説明するセキュリティ・マネージャーを使用して行います。

ビルド・サーバーから戻されるメッセージの言語の設定

Windows 上のビルド・サーバーは、次の表に示すいずれかの言語でメッセージを戻します。デフォルトは英語です。

言語	コード
ブラジル・ポルトガル語	ptb
中国語 (簡体字)	chs
中国語 (繁体字)	cht
英語 (米国)	enu
フランス語	fra
ドイツ語	deu
イタリア語	ita
日本語	jpn
韓国語	kor
スペイン語	esp

英語以外の言語を指定する場合は、ビルド・サーバーを開始する前に、環境変数 **CCU_CATALOG** を英語以外のメッセージ・カタログに設定する必要があります。必要な値は以下のような形式です (1 行)。

```
installationDir¥egl¥eclipse¥plugins  
¥com.ibm.etools.egl.distributedbuild¥executables  
¥ccu.cat.xxx
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:¥Program Files¥IBM¥RSPD¥6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

xxx

ビルド・サーバーでサポートされている言語コード (上記の表にリストされているコードのいずれか)

セキュリティ・マネージャー

セキュリティ・マネージャーは、ビルド・サーバーが、ビルド要求を出すクライアントを認証するために使用するサーバー・プログラムです。

セキュリティ・マネージャー用の環境の設定: セキュリティ・マネージャーは、以下の Windows 環境変数を使用します。

CCUSEC_PORT

セキュリティ・マネージャーが listen するポートの番号を設定します。デフォルト値は 22825 です。

CCUSEC_CONFIG

構成データが保管されているファイルのパス名を設定します。デフォルトのパス名は C:\temp\ccuconfig.bin です。このファイルが見つからない場合は、セキュリティ・マネージャーがそのファイルを作成します。

CCU_TRACE

この変数を * に設定すると、セキュリティ・マネージャーの診断目的のトレースが開始されます。

セキュリティ・マネージャーの開始: セキュリティ・マネージャーを開始するには、以下のコマンドを出します。

```
java com.ibm.ertools.egl.distributedbuild.security.CcuSecManager
```

セキュリティ・マネージャーの構成: セキュリティ・マネージャーを構成するには、グラフィカル・インターフェースを持つ構成ツールを使用します。以下のコマンドを出せば、この構成ツールを実行することができます。

```
java com.ibm.ertools.egl.distributedbuild.security.CCUconfig
```

構成ツールが実行されたならば、「サーバー項目」タブを選択してください。ビルド・サーバーがサポートするユーザーを追加するには、「追加...」ボタンをクリックする。ここで、ユーザー ID に対するパスワードを定義しなければなりません。ユーザーに対する以下の制限事項および特権を定義することができます。

- このユーザーが指定できるロケーション。すなわち、ccubldc コマンドに対する -la パラメーターの値。複数のロケーションを定義するには、セミコロンで区切ります。
- このユーザーが指定できるビルド・スクリプトの名前。(EGL のビルド計画は、javac コマンドをビルド・スクリプトとしてのみ使用します。)
- このユーザーが、クライアントからビルド・スクリプトを送信できるかどうか。ccubldc コマンドの -ft パラメーターを使用します。(EGL 生成プログラムでは、-ft パラメーターは使用しません。Java 生成出力の作成以外の目的でビルドを使用する場合に、このパラメーターを指定します。)

これらの定義は、CCUSEC_CONFIG で指定されているファイル内の永続的なストレージに保持され、セッション全体にわたって記憶されます。

関連する概念

372 ページの『ビルド・スクリプト』

373 ページの『ビルド・サーバー』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

EGL 生成 Java 出力のデプロイ

Java ランタイム・プロパティ

EGL 生成の Java プログラムは、プログラムが使用するデータベースやファイルにアクセスする方法などの情報を提供する一連のランタイム・プロパティを使用します。

J2EE 環境では

J2EE 環境で実行される生成された Java プログラムに関して、次のような状態が可能となります。

- EGL は、J2EE デプロイメント記述子内に直接ランタイム・プロパティを生成できます。この場合、EGL はすでに存在するプロパティを上書きし、存在しないプロパティを追加します。プログラムは実行時に J2EE デプロイメント記述子にアクセスします。
- また、EGL は J2EE 環境ファイル内にランタイム・プロパティを生成することもできます。このファイル内のプロパティをカスタマイズした後、J2EE デプロイメント記述子にコピーできます。
- ランタイム・プロパティがまったく生成されないようにすることもできます。その場合は、必要なプロパティを自身で作成する必要があります。

J2EE モジュールでは、すべてのプログラムが同じランタイム・プロパティを持ちます。これは、モジュール内のすべてのコードが同じデプロイメント記述子を共用するためです。

WebSphere Application Server では、次のように、プロパティは `env-entry` タグとして `web.xml` ファイル内で指定され、そのファイルが Web プロジェクトへ関連付けられます。

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>

<env-entry>
  <env-entry-name>vgj.nls.number.decimal</env-entry-name>
  <env-entry-value>.</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

非 J2EE Java 環境では

J2EE 環境以外で実行される生成済みの Java プログラムに関連して、ランタイム・プロパティをプログラム・プロパティ・ファイル内に生成するか、そのファイルを手作業でコーディングできます。(プログラム・プロパティ・ファイルには、デプロイメント記述子に使用可能なさまざまな情報が書き込まれていますが、プロパティの形式は異なります。)

非 J2EE Java 環境では、プロパティをいくつかのプロパティ・ファイル内に指定でき、それらのファイルは次の順序で検索されます。

- **user.properties**

- 次のような名前のファイル。

programName.properties

programName

実行単位内にある最初のプログラム

- **rununit.properties**

あるユーザーに固有のプロパティを指定するときは、**user.properties** を使用するのが適切です。EGL は、このファイルの内容を生成しません。

実行単位の最初のプログラムがファイルまたはデータベースにはアクセスせず、以下のことを行うプログラムを呼び出す場合は、**rununit.properties** を使用することが適しています。

- 発呼者を生成する場合、プログラム用に名付けられたプロパティ・ファイルを生成することができます。その内容には、データベース関連、またはファイル関連のプロパティは含まれない場合があります。
- 呼び出し先プログラムを生成する場合、**rununit.properties** を生成することができます。その内容は、両プログラムで使用可能です。

これらのファイルは、いずれも必須のものではなく、単純なプログラムでは、どのファイルも必要ではありません。

デプロイメント時には、以下の規則が適用されます。

- ユーザー・プロパティ・ファイル (存在する場合は **user.properties**) は、Java システム・プロパティ *user.home* で判別されるユーザー・ホーム・ディレクトリーの中にあります。
- プログラム・プロパティ・ファイル (存在する場合) のロケーションは、プログラムがパッケージに入っているかどうかによって異なります。分かりやすいように、規則の例を次に示します。
 - プログラム P がパッケージ x.y.z に入っており、MyProject/JavaSource へデプロイされる場合、プログラム・プロパティ・ファイルは MyProject/JavaSource/x/y/z に入っている必要があります。
 - プログラム P がパッケージに入っておらず、myProject/JavaSource へデプロイされる場合、プログラム・プロパティ・ファイルは (グローバル・プロパティ・ファイルのように) MyProject/JavaSource に入っている必要があります。

いずれの場合でも、MyProject/JavaSource はクラスパスに入っている必要があります。

- グローバル・プロパティ・ファイル (存在する場合は **rununit.properties**) は、プログラムと一緒に、クラスパスで指定されたディレクトリーに入っている必要があります。

Java プロジェクトへの出力を生成した場合、EGL は (**user.properties** 以外の) プロパティ・ファイルを適切なフォルダーに配置します。

古いバージョンの EGL または VisualAge Generator で生成された Java コードと同じ実行単位内で使用する Java コードを生成する場合、プロパティ・ファイルをデプロイする規則は、実行単位内の最初のプログラムが EGL 6.0 以降で生成されたか (その場合は、ここで述べた規則が適用されます)、最初のプログラムが古いバージョンの EGL または VisualAge Generator でされたか (その場合は、プロパティ・ファイルはクラスパス内の任意のディレクトリーに入れることができ、グローバル・ファイルの名前は **vgj.properties** です) によって異なります。

最後に、最初のプログラムが古いソフトウェアで生成されている場合は、非グローバル・プログラム・プロパティ・ファイルの代わりに実行単位全体で使用される代替のプロパティ・ファイルを指定できます。詳細については、『*Java ランタイム・プロパティ (詳細)*』でプロパティ **vgj.properties.file** の説明を参照してください。

ビルド記述子およびプログラム・プロパティ

選択内容は、ビルド記述子オプション値として EGL へ実行依頼されます。

- J2EE デプロイメント記述子にプロパティを生成するには、**J2EE** を **YES** に設定し、**genProperties** を **PROGRAM** または **GLOBAL** に設定します。その後、J2EE プロジェクト内に生成します。
- J2EE 環境ファイルにプロパティを生成するには、**J2EE** を **YES** に設定し、**genProperties** を **PROGRAM** または **GLOBAL** に設定します。その後、次のいずれかを実行します。
 - ディレクトリーの中に生成する (ビルド記述子オプション **genProject** ではなく **genDirectory** を使用する場合)。または、
 - 非 J2EE プロジェクトの中に生成する。
- 生成されたプログラムと同じ名前を持つプログラム・プロパティ・ファイルを生成するには、**J2EE** を **NO** に設定し、**genProperties** を **PROGRAM** に設定します。その後、J2EE プロジェクト以外のプロジェクト内に生成します。
- プログラム・プロパティ・ファイル **rununit.properties** を生成するには、**J2EE** を **NO** に設定し、**genProperties** を **GLOBAL** に設定します。その後、J2EE 以外のプロジェクト内に生成します。
- プロパティをまったく生成しない場合は、**genProperties** を **NO** に設定します。

追加情報について

プロパティをデプロイメント記述子または J2EE 環境ファイルに生成する方法の詳細については、『*デプロイメント記述子値の設定*』を参照してください。

ランタイム・プロパティの意味の詳細については、『*Java ランタイム・プロパティ (詳細)*』を参照してください。

EGL コード内でランタイム・プロパティにアクセスする方法の詳細については、『*sysLib.getProperty*』を参照してください。

関連する概念

303 ページの『EGL デバッガー』

351 ページの『プロジェクトへの Java コードの生成』

388 ページの『J2EE 環境ファイル』
『プログラム・プロパティー・ファイル』
798 ページの『実行単位』

関連するタスク

364 ページの『ディレクトリーへ生成される Java コードの処理』
384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』
386 ページの『デプロイメント記述子値の設定』
388 ページの『デプロイメント記述子の手操作による更新』
387 ページの『J2EE 環境ファイルの更新』

関連する参照項目

428 ページの『genProperties』
431 ページの『J2EE』
584 ページの『Java ランタイム・プロパティー (詳細)』
967 ページの『getProperty()』

EGL 生成コード用の非 J2EE ランタイム環境の設定

プログラム・プロパティー・ファイル

プログラム・プロパティー・ファイル には、J2EE 以外の環境で実行される Java プログラムのみにアクセス可能な形式の Java ランタイム・プロパティーが含まれています。概説については、『Java ランタイム・プロパティー』を参照してください。

プログラム・プロパティー・ファイルはテキスト・ファイルです。コメントを除く各エントリーは次の形式をとります。

propertyName = *propertyValue*

propertyName

『Java ランタイム・プロパティー (詳細)』で説明されているいずれかのプロパティー

propertyValue

実行時にプログラムで使用可能なプロパティー値

コメントは、最初の非テキスト文字がポンド記号 (#) である任意の行です。

サンプル・ファイルの一部を以下に示します。

```
# This file contains properties for generated
# Java programs that are being debugged in a
# non-J2EE Java project
vgj.nls.code = ENU
vgj.datemask.gregorian.long.ENU = MM/dd/yyyy
```

生成されたファイルに付けられる名前の詳細については、『生成される出力 (参照)』を参照してください。

関連する概念

303 ページの『EGL デバッガー』
377 ページの『Java ランタイム・プロパティー』

関連するタスク

573 ページの『生成される出力 (参照)』

関連する参照項目

428 ページの『genProperties』

431 ページの『J2EE』

584 ページの『Java ランタイム・プロパティ (詳細)』

J2EE の外側での Java アプリケーションのデプロイ

J2EE の外側で Java アプリケーションをデプロイするには、次のようにします。

1. 『Java 用 EGL ランタイム・コードのインストール』にある手順に従う。
2. EGL 生成のコードを jar ファイルにエクスポートする。このとき、jasper、プロパティ、および tab ファイルなどの java 以外の拡張子を持つ、生成された出力ファイルを必ずインクルードするようにしてください。
3. 手動で作成したすべての Java コードを JAR ファイルにエクスポートする。
4. ターゲット・マシンの classpath にエクスポート・ファイルを組み込む。

関連するタスク

『Java 用 EGL ランタイム・コードのインストール』

Java 用 EGL ランタイム・コードのインストール

生成された Java アプリケーション用の EGL ランタイム・コードは、次の Web サイトから Zip ファイルで入手することができます。

<http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist>

サポートされる分散プラットフォームは、AIX、HP-UX、Linux (Intel™)、iSeries、Solaris、および Windows 2000/NT/XP です。(サポートされるバージョンの製品の前提条件を参照してください。)EGL は、AIX、HP-UX、および Solaris のために、32-ビットおよび 64-ビットのサポートを提供します。

上記の Web サイトからダウンロードした Zip ファイルの内容は次のとおりです。

- サポートされているすべての分散プラットフォームで共通する Java コードが含まれた Jar ファイル
- プラットフォーム特化コード

以下のようにします。

1. デプロイされた EGL アプリケーションを J2EE アプリケーション・サーバーの外側で実行するための各マシンの EGLRuntimes ディレクトリーでファイルを解凍する。(これらのファイルは、J2EE アプリケーションのデプロイに使用されるいずれかのエンタープライズ・アーカイブ (EAR) ファイルにすでに含まれています。)
2. デプロイメント・マシンの classpath に jar ファイルを組み込む。
3. プラットフォーム特化コードを各デプロイメント・マシンのディレクトリーにコピーし、それらのマシンに適した環境変数を次のように設定する。

AIX (32- または 64-ビット・サポート) の場合

関連ファイルは Aix ディレクトリー (64-ビット・サポート用) または Aix64

ディレクトリーにあります。 Web サイトからコピーしたプラットフォーム特化コードを含むディレクトリーを参照するように、環境変数 `PATH` および `LIBPATH` を変更します。

HP-UX (32- または 64-ビット・サポート) の場合

関連ファイルは **hpux** ディレクトリー (64-ビット・サポート用) または **hpux64** ディレクトリーにあります。 Web サイトからコピーしたプラットフォーム特化コードを含むディレクトリーを参照するように、環境変数 `PATH` および `LIBPATH` を変更します。

iSeries の場合

関連ファイルは **Iseries** ディレクトリーにあります。 `qshell` で、ファイルのアップロード先のディレクトリーに変更し、「install」オプションを使用して `setup.sh` スクリプトを実行します。

> `setup.sh install`

また、ほかのいくつかの環境変数を設定する必要があります。これらの環境変数の設定方法に関する情報については、「envinfo」オプションを使用してスクリプトを実行します。

> `setup.sh envinfo`

インストール中に作成された `symlink` をなんらかの理由で削除した場合、「link」オプションを使用するとこの `symlink` を再作成できます。

> `setup.sh link`

Linux の場合

関連ファイルは **Linux** ディレクトリーにあります。 Web サイトからコピーしたプラットフォーム特化コードを含むディレクトリーを参照するように、環境変数 `PATH` および `LIBPATH` を変更します。

Solaris (32- または 64-ビット・サポート用) の場合

関連ファイルは **Solaris** ディレクトリー (64-ビット・サポート用) または **Solaris64** ディレクトリーにあります。 Web サイトからコピーしたプラットフォーム特化コードを含むディレクトリーを参照するように、環境変数 `PATH` および `LIBPATH` を変更します。

Windows 2000/NT/XP の場合

関連ファイルは **Win32** ディレクトリーにあります。 Web サイトからコピーしたプラットフォーム特化コードを含むディレクトリーを参照するように、環境変数 `PATH` を変更します。

関連するタスク

381 ページの『J2EE の外側での Java アプリケーションのデプロイ』

ターゲット・マシンの CLASSPATH への JAR ファイルの組み込み

EGL 生成コードまたは手操作で作成した Java コードが入っている JAR ファイルを、ターゲット・マシンの `CLASSPATH` に組み込む必要があります。このプロセスのステップは、システムによって異なります。詳細については、使用しているオペレーティング・システムの資料を参照してください。

EGL ランタイムの UNIX curses ライブラリーの設定

AIX または Linux に EGL テキスト・プログラムをデプロイすると、EGL ランタイムは、UNIX curses ライブラリーを使用しようとしています。UNIX curses ライブラリーの環境がセットアップされていないか、またはそのライブラリーがサポートされていない場合、EGL ランタイムは、Java Swing テクノロジーを使用しようとし、そのテクノロジーも使用不可の場合、プログラムは失敗します。

UNIX curses ライブラリーは、ユーザーが端末エミュレーター・ウィンドウまたは文字端末から EGL プログラムを実行する場合に必要となります。

AIX または Linux 上の UNIX curses 端末ライブラリーにアクセスするために、EGL ランタイムを使用可能に設定するには、UNIX シェル環境でいくつかのステップを実行する必要があります。最初の 2 つの各ステップにおいて、*installDir* はランタイム・インストール・ライブラリーを指します。

1. 共有オブジェクト *libCursesCanvas6.so* を組み込むために、*LD_LIBRARY_PATH* 環境変数を変更する。このオブジェクトは、ランタイム・インストール・ライブラリーで提供されます。

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH: /installDir/bin
```

2. *fda6.jar* および *fdaj6.jar* を追加するために、*CLASSPATH* 環境変数を変更する。

```
export CLASSPATH=$CLASSPATH:  
/installDir/lib/fda6.jar: /installDir/lib/fdaj6.jar
```

前述の情報は、単一行に入力する必要があります。

3. 次の例のように、*TERM* 環境変数を適切な端末設定に設定する。

```
export TERM=vt100
```

端末に例外が発生した場合、*xterm*、*dtterm*、*vt220* など、さまざまな端末設定を試してみます。

4. 次の例のように、UNIX シェルから EGL Java プログラムを実行する。

```
java myProgram
```

プログラムが常駐するディレクトリーが *CLASSPATH* 環境変数によって識別されることを確認してください。

UNIX で Curses ライブラリーを使用する方法の追加情報については、UNIX の *man* ページを参照してください。

呼び出し先となる非 J2EE アプリケーションの TCP/IP リスナーのセットアップ

呼び出し元で TCP/IP を使用して、非 J2EE Java の呼び出し先プログラムとデータを交換したい場合は、呼び出し先プログラム用に TCP/IP リスナーをセットアップする必要があります。

TCP/IP を使用して、非 J2EE Java の呼び出し先プログラムと通信している場合は、このプログラム用に *CSOTcpipListener* と呼ばれるスタンドアロン Java プログラムを構成する必要があります。詳しくは、次のように行う必要があります。

- CSOTcpipListener の実行時に使用するクラスパスに、fda6.jar、fdaj6.jar、および呼び出し先プログラムが含まれているディレクトリまたはアーカイブが含まれていることを確認する。
- Java ランタイム・プロパティ **tcpiplistener.port** を、CSOTcpipListener がデータを受け取るポートの番号に設定する。

次のいずれかの方法で、スタンドアロン TCP/IP リスナーを開始することができます。

- ワークベンチからリスナーを開始するには、Java アプリケーションの起動構成を使用する。この場合、起動構成のプログラム引数でプロパティ・ファイルの名前を指定できます。あるいは、ファイル tcpiplistener.properties をデフォルトとして使用している場合は、そのファイルはフォルダー内に置かず、起動構成の作成時にプロジェクトの下に直接置く必要があります。
- コマンド行からリスナーを開始するには、以下のようにプログラムを実行する。

```
java CSOTcpipListener propertiesFile
```

```
propertiesFile
```

TCP/IP リスナーによって使用されるプロパティ・ファイルへの完全修飾パス。プロパティ・ファイルを指定しないと、リスナーは現行ディレクトリで次のファイルを開こうとします。

```
tcpiplistener.properties
```

関連するタスク

397 ページの『非 EGL JAR ファイルへのアクセスの提供』

EGL 生成コード用の J2EE ランタイム環境のセットアップ

EGL 生成 Java プログラムおよびラッパーは、ランタイム構成 にリストされているプラットフォームにおいて、WebSphere Application Server v6.0 などの J2EE 1.4 サーバー上で実行されます。

生成された Java クラスを J2EE モジュールに組み込む際の基本タスクは以下のとおりです。

1. 次の 2 つのうちいずれかの方法で出力ファイルをプロジェクトに置く。
 - プロジェクトへ生成する。これは推奨する手法です。
 - ディレクトリへ生成してから、ファイルをプロジェクトへインポートする。
2. モジュールにリンケージ・プロパティ・ファイルを置く (『リンケージ・プロパティ・ファイルのデプロイ』を参照)。
3. 重複する JAR ファイルを除去する。
4. エンタープライズ・アーカイブ (.ear) ファイルをエクスポートする。これには、Web アプリケーション・アーカイブ (.war) ファイルやその他の .ear ファイルが含まれている場合があります。手順の詳細については、エクスポートに関するヘルプ・ページを参照してください。
5. .ear ファイルを、アプリケーションのホストになる J2EE サーバーにインポートする。手順の詳細については、J2EE サーバーの 資料を参照してください。

場合によっては、次のタスクも完了する必要があります。

- 394 ページの『J2EE JDBC 接続のセットアップ』
- 389 ページの『CICSJ2C 呼び出し用 J2EE サーバーのセットアップ』
- 390 ページの『J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用の TCP/IP リスナーのセットアップ』
- 383 ページの『呼び出し先となる非 J2EE アプリケーションの TCP/IP リスナーのセットアップ』

関連する概念

- 9 ページの『開発過程』
- 351 ページの『プロジェクトへの Java コードの生成』
- 357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』
- 338 ページの『リンケージ・オプション・パーツ』
- 396 ページの『リンケージ・プロパティー・ファイル』
- 10 ページの『ランタイム構成』

関連するタスク

- 381 ページの『J2EE の外側での Java アプリケーションのデプロイ』
- 395 ページの『リンケージ・プロパティー・ファイルのデプロイ』
- 『重複する JAR ファイルの除去』
- 368 ページの『EJB プロジェクトのデプロイメント・コードの生成』
- 364 ページの『ディレクトリーへ生成される Java コードの処理』
- 397 ページの『非 EGL JAR ファイルへのアクセスの提供』
- 386 ページの『デプロイメント記述子値の設定』
- 389 ページの『EJB プロジェクトの JNDI 名の設定』
- 394 ページの『J2EE JDBC 接続のセットアップ』
- 389 ページの『CICSJ2C 呼び出し用 J2EE サーバーのセットアップ』
- 390 ページの『J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用の TCP/IP リスナーのセットアップ』
- 283 ページの『標準 JDBC 接続の作成方法について』
- 388 ページの『デプロイメント記述子の手操作による更新』
- 387 ページの『J2EE 環境ファイルの更新』

関連する参照項目

- 584 ページの『Java ランタイム・プロパティー (詳細)』
- 708 ページの『リンケージ・プロパティー・ファイル (詳細)』

重複する JAR ファイルの除去

複数の J2EE モジュールを単一の ear ファイルに置く場合は、以下のように、重複する JAR ファイルを除去します。

1. 重複する各 JAR ファイルのコピーを ear のトップレベルに移動する。
2. 重複する JAR ファイルを J2EE モジュールから削除する。
3. 影響を受ける各 J2EE モジュールのビルド・パスが ear 内の JAR ファイルを指していることを確認する。詳しくは、各 J2EE モジュールに対して次のようにします。
 - a. プロジェクト・エクスプローラーまたは J2EE のビューの中で、モジュールを右クリックする。
 - b. 「Edit Module Dependencies (モジュール依存関係の編集)」を選択する

- c. 「モジュール依存関係」ダイアログが表示されたならば、ear のトップレベルからアクセスして JAR ファイルを選択し、「完了」をクリックする

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

デプロイメント記述子値の設定

重要なタスクは、ランタイム値（環境変数値に似ている）を J2EE モジュールのデプロイメント記述子に入れることです。たとえば、次の表にリストされているワークベンチ・エディターと対話することができます。また、いずれにしても、値を再割り当てしたい場合はエディターを使用できます。

プロジェクト・タイプ	デプロイメント記述子の名前	値の割り当て方法
アプリケーション・クライアント	application-client.xml	XML エディター、「設計」タブを使用
EJB	ejb-jar.xml	EJB エディター、「Beans」タブを使用
J2EE Web	web.xml	web.xml エディター、「Environment (環境)」タブを使用

デプロイメント記述子を更新する方法として、内容を自動的に追加することをお勧めします。これは、たとえば以下の条件がすべて満たされた場合に行われます。

- Java プログラムまたはラッパーを生成する。
- ビルド記述子オプション **genProperties** が GLOBAL または PROGRAM に設定されている。
- **J2EE** を YES に設定して、J2EE ランタイムのために生成している。
- **genProject** を有効な J2EE プロジェクトに設定する。

EGL は既存のデプロイメント記述子からプロパティを削除することはありませんが、以下のことを行います。

- すでに存在するプロパティを上書きする。
- 存在しないプロパティを付加する。

デプロイメント記述子を更新するもう 1 つの方法は、J2EE 環境ファイルから値を貼り付けることです。J2EE 環境ファイルは、以下の条件がすべて満たされている場合に、生成の出力になります。

- Java プログラムを生成する。
- ビルド記述子オプション **genProperties** が GLOBAL または PROGRAM に設定されている。
- **J2EE** を YES に設定して、J2EE ランタイムのために生成している。
- ディレクトリに生成する場合のように、**genProject** を有効な J2EE プロジェクトに設定しない。

J2EE 環境ファイルからアプリケーション・クライアントまたは EJB プロジェクトのデプロイメント記述子にエントリを貼り付ける前に、『J2EE 環境ファイルの更

新』に記述されているとおりにファイル内のエントリーの順序を変更する必要があります。J2EE Web プロジェクトで作業している場合は、エントリーの順序を変更する必要はありません。

デプロイメント記述子プロパティの詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

関連する概念

- 388 ページの『J2EE 環境ファイル』
- 351 ページの『プロジェクトへの Java コードの生成』
- 380 ページの『プログラム・プロパティ・ファイル』

関連するタスク

- 364 ページの『ディレクトリーへ生成される Java コードの処理』
- 384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』
- 『J2EE 環境ファイルの更新』
- 388 ページの『デプロイメント記述子の手操作による更新』

関連する参照項目

- 425 ページの『genDirectory』
- 428 ページの『genProperties』
- 431 ページの『J2EE』
- 584 ページの『Java ランタイム・プロパティ (詳細)』

J2EE 環境ファイルの更新

J2EE 環境ファイルには、次の例のような一連のエントリーが含まれます。

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-value>ENU</env-entry-value>
  <env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

サブエレメントの順序は、名前、値、型です。J2EE Web プロジェクトの場合はこれが正しい順序ですが、アプリケーション・クライアントおよび EJB プロジェクトの場合は、名前、型、値に順序を変更する必要があります。上記の例では、サブエレメントの順序を次のように変更します。

```
<env-entry>
  <env-entry-name>vgj.nls.code</env-entry-name>
  <env-entry-type>java.lang.String</env-entry-type>
  <env-entry-value>ENU</env-entry-value>
</env-entry>
```

ディレクトリーではなくプロジェクトへ直接に生成する場合は、このステップを回避できます。プロジェクトへ生成すると、EGL は、使用しているプロジェクトの型を判別することができ、環境エントリーを適切な順序で生成します。

関連するタスク

- 384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』
- 386 ページの『デプロイメント記述子値の設定』

関連する参照項目

- 584 ページの『Java ランタイム・プロパティ (詳細)』

J2EE 環境ファイル

J2EE 環境ファイル プロパティーと値のペアを含むテキスト・ファイルで、Java プログラムの生成時に指定した情報から派生します。情報のソースはビルド記述子、リソース関連パーツ、およびリンケージ・オプション・パーツです。

Java プログラムの環境を構成する際に、ランタイム・デプロイメント記述子に含める情報の基礎として J2EE 環境ファイルを使用できます。

J2EE 環境ファイルの名前の詳細については、『生成される出力 (参照)』を参照してください。

デプロイメント記述子の値を設定するさまざまな方法の詳細については、『デプロイメント記述子値の設定』を参照してください。

関連する概念

10 ページの『ランタイム構成』

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

386 ページの『デプロイメント記述子値の設定』

関連する参照項目

573 ページの『生成される出力 (参照)』

428 ページの『genProperties』

435 ページの『sqlDB』

デプロイメント記述子の手操作による更新

生成された J2EE 環境ファイルからデプロイメント記述子を更新する場合は、次のようにします。

1. 『デプロイメント記述子値の設定』で概要の情報を読む。
2. アプリケーション・クライアントまたは EJB プロジェクトで作業した場合は、『J2EE 環境ファイルの更新』に説明されているとおりに、生成された環境エントリーのサブエレメントの順序が正しいことを確認する必要がある。
3. 以下のように、環境エントリーをプロジェクトのデプロイメント記述子にコピーする。
 - a. デプロイメント記述子のバックアップ・コピーを作成する。
 - b. J2EE 環境ファイルを開く。これは、*programName-env.txt* ファイルと呼ばれます。環境エントリーをクリップボードへコピーする。
 - c. デプロイメント記述子をダブルクリックする。
 - d. 「ソース (Source)」タブをクリックする。
 - e. エントリーを適切なロケーションに貼り付ける。

デプロイメント記述子の詳細については、『Java ランタイム・プロパティー (詳細)』を参照してください。

関連するタスク

386 ページの『デプロイメント記述子値の設定』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

387 ページの『J2EE 環境ファイルの更新』

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

EJB プロジェクトの JNDI 名の設定

EJB プロジェクトの JNDI 名を設定するには、以下のようにします。

1. `ejb-jar.xml` (デプロイメント記述子) を右クリックして、コンテキスト・メニューを開く。
2. EJB エディターを使用して、プロジェクト内の次のファイルを開く。
`¥ejbModule¥META-INF¥ejb-jar.xml`
3. 「Beans (Bean)」タブをクリックする。
4. リストで、生成したばかりの EJB の名前をクリックする。
5. 「WebSphere Bindings (WebSphere バインディング)」で、JNDI 名を入力する。
JNDI 名は、EGL ランタイム・コードで使用するために以下のとおりにする必要があります。
 - プログラム名の先頭文字 (大文字)
 - プログラム名の後続の文字 (小文字)
 - 文字 EJB (大文字)

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

CICSJ2C 呼び出し用 J2EE サーバーのセットアップ

プロトコル CICSJ2C を介してアクセスされる各 CICS トランザクションに対して、J2EE サーバーで `ConnectionFactory` をセットアップする必要があります。

生成される Java ラッパーが CICSJ2C 呼び出しを行っている場合は、以下のいずれかの方法でセキュリティを処理することができます (その場合、ラッパー指定の値が J2EE サーバーの値をオーバーライドします)。

- ラッパーの `CSOCallOptions` オブジェクトでユーザー ID とパスワードを設定する。
- J2EE サーバーの `ConnectionFactory` 構成でユーザー ID とパスワードを設定する。
- CICS 領域をセットアップして、ユーザー認証が必要でなくなるようにする。

WebSphere 390 からプログラムを呼び出す場合、以下の制限事項が適用されます。

- `callLink` エレメントのプロパティ `luwControl` が `CLIENT` に設定されている場合、呼び出しは失敗する。WebSphere 390 接続実装は拡張作業単位をサポートしていません。
- デプロイメント記述子プロパティ `csocicsj2c.timeout` の設定は無効である。

デフォルトでは、タイムアウトは発生しません。ただし、マクロ `DFHXCOPT` によって生成される EXCI オプション・テーブルでパラメーター `TIMEOUT` を設

定して、EXCI が DPL コマンド (ECI 要求) の完了を待つ時間を指定することはできません。0 の設定は、無期限に待つことを意味します。

詳細については、『*Java Connectors for CICS: Featuring the J2EE Connector Architecture*』(SG24-6401-00) を参照してください。これは、Web サイト <http://www.redbooks.ibm.com> から入手できます。

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用の TCP/IP リスナーのセットアップ

呼び出し元で TCP/IP を使用して、J2EE アプリケーション・クライアント・モジュール内の呼び出し先プログラムとデータを交換したい場合は、呼び出し先プログラムの TCP/IP リスナーをセットアップする必要があります。

以下の状況が有効であることを確認する必要があります。

- モジュールのマニフェスト (.MF) ファイルに指定されているように、EGL 固有の TCP/IP リスナーがモジュールのメイン・クラスである
- モジュールのデプロイメント記述子 (application-client.xml) に指定されているように、ポートがリスナーに割り当てられている

J2EE 1.2 のレベルでプロジェクトで作業している場合は、リスナーを使用して初期化されているアプリケーション・クライアント・プロジェクトをセットアップした後、プロジェクト内に EGL コードを生成することをお勧めします。この順序 (リスナーの次に EGL) に従うのに失敗したか、または J2EE 1.3 のレベルでプロジェクトの作業をしている場合は、『*既存のアプリケーション・クライアント・プロジェクトからリスナーへのアクセスの提供*』で説明されている手順に従う必要があります。

リスナーを使用して初期化されているアプリケーション・クライアント・プロジェクトのセットアップ

リスナーを使用して初期化されているアプリケーション・クライアント・プロジェクトをセットアップするには、次のようにします。

1. 「ファイル」>「インポート」をクリックする。
2. 「選択」ページで「アプリケーション・クライアント JAR ファイル」をダブルクリックする。
3. 「アプリケーション・クライアントのインポート」ページで詳細をいくつか指定する。
 - a. 「アプリケーション・クライアント・ファイル」フィールドで、TCP/IP リスナーへのアクセスをセットアップ (ただし、含めない) する JAR ファイルを指定する。

```
installationDir¥egl¥eclipse¥plugins¥  
com.ibm.etools.egl.generators_¥version¥runtime¥EGLTcpListener.jar
```

installationDir

製品のインストール・ディレクトリー。たとえば、C:\Program Files\IBM\RSPPD\6.0 など。これから使用しようとしている製品をインス

ツールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

最新バージョンのプラグイン (例: 6.0.0)

TCP/IP リスナー自体は、プロジェクトに最初に EGL コードを生成するときにアプリケーション・クライアント・プロジェクトに配置される fdaj6.jar にあります。

- b. ラベル「**Application Client project (アプリケーション・クライアント・プロジェクト)**」の後にある「**新規**」ラジオ・ボタンをクリックする。
- c. 「**New Project Name (新規プロジェクト名)**」フィールドにアプリケーション・クライアント・プロジェクトの名前を入力する。次に、「**Use default (デフォルトを使用)**」チェック・ボックスをオンまたはオフにします。チェック・ボックスをオンにすると、プロジェクトはプロジェクト名のワークスペース・ディレクトリーに保管されます。チェック・ボックスをオフにした場合は、「**New project location (新規プロジェクト・ロケーション)**」フィールドにプロジェクト名を指定します。
- d. アプリケーション・クライアント・プロジェクトを含むエンタープライズ・アプリケーション・プロジェクトの名前を指定する。
 - 既存の J2EE 1.2 エンタープライズ・アプリケーション・プロジェクトを使用している場合は、ラベル「**Enterprise application project (エンタープライズ・アプリケーション・プロジェクト)**」の後にある「**Existing (既存)**」ラジオ・ボタンをクリックする。この場合、「**Existing project name (既存のプロジェクト名)**」フィールドにプロジェクト名を指定します。
 - 新規エンタープライズ・アプリケーション・プロジェクトを作成する場合は、次のようにする。
 - 1) ラベル「**Enterprise application project (エンタープライズ・アプリケーション・プロジェクト)**」の後にある「**新規**」ラジオ・ボタンをクリックする。
 - 2) 「**New Project Name (新規プロジェクト名)**」フィールドにエンタープライズ・アプリケーション・プロジェクトの名前を入力する。
 - 3) 「**Use default (デフォルトを使用)**」チェック・ボックスをオンまたはオフにする。
 - 4) チェック・ボックスをオンにすると、プロジェクトはプロジェクト名のワークスペース・ディレクトリーに保管されます。チェック・ボックスをオフにした場合は、「**New project location (新規プロジェクト・ロケーション)**」フィールドにプロジェクト名を指定します。
4. 「完了」をクリックする。
5. プロジェクトに EGL 出力を生成すると自動的に追加される JAR ファイル (fda6.jar, fdaj6.jar) を指す 2 つの警告メッセージを無視する。

アプリケーション・クライアント・プロジェクトでは、デプロイメント記述子プロパティー **tcpiplistener.port** がリスナーがデータを受け取るポートの番号に設定されています。デフォルトでは、このポート番号は 9876 です。ポート番号を変更するには、以下のようにします。

1. 「プロジェクト・エクスプローラー」ビューでアプリケーション・クライアント・プロジェクト、appClientModule、および META-INF を順に展開する。
2. 「application-client.xml」>「アプリケーションから開く」>「Deployment Descriptor editor (デプロイメント記述子エディター)」をクリックする。
3. デプロイメント記述子エディターには、「source (ソース)」タブがある。このタブをクリックし、値 9876 を変更します。この値は、以下のようなグループ内の最後のタグのコンテンツです。

```
<env-entry-name>tcpiplistener.port</env-entry-name>
<env-entry-type>java.lang.Integer</env-entry-name>
<env-entry-value>9876</env-entry-value>
```
4. デプロイメント記述子を保管するには、**Ctrl-S** を押す。

既存のアプリケーション・クライアント・プロジェクトからリスナーへのアクセスの提供

リスナーを使用して初期化されていないアプリケーション・クライアント・プロジェクトに EGL コードを生成する場合は、デプロイメント記述子 (application-client.xml) およびマニフェスト・ファイル (MANIFEST.MF) を更新する必要があります。

1. 「プロジェクト・エクスプローラー」ビューでアプリケーション・クライアント・プロジェクト、appClientModule、および META-INF を順に展開する。
2. 「application-client.xml」>「アプリケーションから開く」>「Deployment Descriptor Editor (デプロイメント記述子エディター)」をクリックする。
3. デプロイメント記述子エディターには、「Source (ソース)」タブがある。このタブをクリックします。テキストで、タグ <display-name> を保持する行のすぐ下に以下のエントリーを追加します (ただし、ポート 9876 がすでにマシンで使用されている場合は、9876 を他の番号に置換します)。

```
<env-entry>
  <env-entry-name>tcpiplistener.port</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-name>
  <env-entry-value>9876</env-entry-value>
</env-entry>
```
4. デプロイメント記述子を保管するには、**Ctrl-S** を押す。
5. 「プロジェクト・エクスプローラー」ビューで「MANIFEST.MF」>「アプリケーションから開く」>「JAR 依存関係エディター」をクリックする。
6. JAR 依存関係エディターには、「Dependencies (依存性)」タブがある。このタブをクリックします。
7. 「Dependencies (依存性)」セクションで fda6.jar および fdaj6.jar が選択されていることを検討する。
8. 「Main Class (メイン・クラス)」で、「Main-Class (メイン・クラス)」フィールドに次の値を入力するか、または参照メカニズムを使用して次の値を指定する。

```
CS0TcpiListenerJ2EE
```
9. マニフェスト・ファイルを保管するには、**Ctrl-S** を押す。

アプリケーション・クライアント・プロジェクトのデプロイ

TCP/IP リスナーを開始するには、以下の 2 つの手順のうちのいずれかに従います。

- WebSphere アプリケーション・クライアント用の起動構成を使用して、ワークベンチからリスナーを開始する。
 1. J2EE パースペクティブに切り替える。
 2. 「Run (実行)」 > 「Run (実行)」をクリックする。
 3. 「Launch Configurations (起動構成)」ページで「**WebSphere v5 Application Client (WebSphere v5 アプリケーション・クライアント)**」(J2EE 1.3 のレベルでプロジェクトを処理している場合に必要) または「**WebSphere v4 Application Client (WebSphere v4 アプリケーション・クライアント)**」をクリックする。
 4. 既存構成を選択する。または「新規」をクリックして、構成をセットアップします。
 - a. 「Application (アプリケーション)」タブでエンタープライズ・アプリケーション・プロジェクトを選択する。
 - b. 「Arguments (引数)」タブで引数を追加する。


```
-CCjar=myJar.jar
```

myJar.jar
アプリケーション・クライアント JAR ファイルの名前。この引数は、ear ファイル内に複数のクライアント JAR ファイルがある場合にのみ必要です。通常、値は、アプリケーション・クライアント・プロジェクト名の後に拡張子 .jar を付けたものです。

プロジェクト名と JAR ファイル名の関連を確認する場合は、次のようにします。

 - 1) 「プロジェクト・エクスプローラー」ビューでエンタープライズ・アプリケーション・プロジェクトおよび META-INF を順に展開する。
 - 2) 「application.xml」 > 「アプリケーションから開く」 > 「Deployment Descriptor Editor (デプロイメント記述子エディター)」をクリックする。
 - 3) デプロイメント記述子エディターには、「Module (モジュール)」タブがある。このタブをクリックします。
 - 4) ページの左端で JAR ファイルをクリックし、(ページの右端で) この JAR ファイルに関連付けられているプロジェクト名を確認する。
- WebSphere Application Server (WAS) がインストールされている場合は、WAS インストール・ディレクトリーのサブディレクトリー bin にある launchClient.bat を使用できる。

コマンド・プロンプトから以下のように launchClient を呼び出すことができます。

```
launchClient myCode.ear -CCjar=myJar.jar
```

myCode.ear

エンタープライズ・アーカイブの名前。

myJar.jar

ワークベンチ・プロシージャーに関連して説明されているアプリケーション・クライアント JAR ファイルの名前。

launchClient.bat の詳細については、WebSphere Application Server の資料を参照してください。

関連するタスク

397 ページの『非 EGL JAR ファイルへのアクセスの提供』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

J2EE JDBC 接続のセットアップ

実行時にリレーショナル・データベースに接続する場合は、プログラムで使用するデータ・ソースを定義する必要があります。それを行うための指示は、WebSphere サーバー管理コンソールのヘルプ・システムに記載されています。

データ・ソースを定義する際は、以下のプロパティに値を割り当てます。

JNDI name

JNDI レジストリー内でデータベースがバインドされる名前と一致する値を指定します。

- J2EE モジュールがデフォルトで使うデータベースに接続するデータ・ソースを定義している場合は、データ・ソース定義で指定されている JNDI 名が、実行時に使用される J2EE デプロイメント記述子内の **vgj.jdbc.default.database** プロパティの値と一致することを確認してください。
- システム関数 **VGLib.connectionService** の実行時にアクセスされるデータ・ソースを定義している場合は、データ・ソース定義で指定されている JNDI 名が、実行時に使用される J2EE デプロイメント記述子内の適切な **vgj.jdbc.database.SN** プロパティの値と一致することを確認してください。

データベース名

データベース管理システムに認識されているデータベースの名前を指定します。

ユーザー ID

データベースに接続するユーザー名を指定します。

データ・ソース定義がデフォルト・データベースを参照する場合、「ユーザー ID」フィールドで指定する値は、実行時に使用される J2EE デプロイメント記述子の **vgj.jdbc.default.userid** プロパティで設定されている値によってオーバーライドされます。ただし、そうなるのは、**vgj.jdbc.default.userid** と **vgj.jdbc.default.password** の両方の値を指定している場合だけです。同様に、データ・ソース定義がシステム関数 **sysLib.connect** または **VGLib.connectionService** を介してアクセスされるデータベースを参照する場合、「ユーザー ID」フィールドで指定する値は、そのシステム関数への呼び出しで指定するユーザー ID によってオーバーライドされます。ただし、そうなるのは、呼び出しがユーザー ID とパスワードの両方を渡す場合だけです。

この名前は、認証別名をセットアップするときに指定します。その別名を定義できる表示に到達するには、管理コンソールで次の順に選択します。「セキュリティ

「セキュリティー (Security)」 > 「GlobalSecurity」 > 「認証 (Authentication)」 > 「JAAS 構成 (JAAS Configuration)」 > 「J2C 認証データ (J2C Authentication Data)」。

パスワード

データベースに接続するためのパスワードを指定します。データ・ソース定義がデフォルト・データベースを参照する場合、「パスワード」フィールドで指定する値は、実行時に使用される J2EE デプロイメント記述子の

vgj.jdbc.default.password プロパティで設定されている値によってオーバーライドされます。ただし、そうなるのは、**vgj.jdbc.default.userid** と

vgj.jdbc.default.password の両方の値を指定している場合だけです。同様に、データ・ソース定義が システム関数 **VGLib.connectionService** を介してアクセスされるデータベースを参照する場合、「パスワード」フィールドで指定する値は、そのシステム関数への呼び出しで指定するパスワードによってオーバーライドされます。ただし、そうなるのは、呼び出しがユーザー ID とパスワードの両方を渡す場合だけです。

このパスワードは、認証別名をセットアップするときに指定します。その別名を定義できる表示に到達するには、管理コンソールで次の順に選択します。「セキュリティー (Security)」 > 「GlobalSecurity」 > 「認証 (Authentication)」 > 「JAAS 構成 (JAAS Configuration)」 > 「J2C 認証データ (J2C Authentication Data)」。

複数のデータ・ソースを定義することもできます。その場合は、システム関数 **VGLib.connectionService** を使用してデータ・ソースを切り替えることができます。

生成される値の派生方法も含め、デプロイメント記述子プロパティの意味の詳細については、『Java ランタイム・プロパティ (参照)』を参照してください。

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

604 ページの『EGL での JDBC ドライバーの要件』

980 ページの『**connectionService()**』

リンケージ・プロパティ・ファイルのデプロイ

リンケージ・プロパティ・ファイルは、そのファイルを使用する Java プログラムと同じ J2EE アプリケーションに入っている必要があります。このファイルがアプリケーションのトップレベル・ディレクトリーに入っている場合は、パス情報を指定せずに、Java ランタイム・プロパティ **cso.linkageOptions.LO** をこのファイル名に設定します。このファイルがアプリケーションのトップレベル・ディレクトリーの下にある場合は、トップレベル・ディレクトリーから始まるパスを使用し、それぞれのレベルごとにスラッシュ (/) を組み込みます。これは、アプリケーションが Windows プラットフォーム上で稼働する場合でも同様です。

J2EE プロジェクトを開発している場合、トップレベル・ディレクトリーは、モジュールが常駐するプロジェクトの **appClientModule**、**ejbModule**、または **Web コンテ**

ンツ・ディレクトリーに対応します。Java プロジェクトを開発している場合、トップレベル・ディレクトリーは、プロジェクト・ディレクトリーです。

リンケージ・プロパティー・ファイルをフォーマット設定および識別する方法の詳細については、『リンケージ・プロパティー・ファイル (参照)』を参照してください。

関連する概念

377 ページの『Java ランタイム・プロパティー』
338 ページの『リンケージ・オプション・パーツ』
『リンケージ・プロパティー・ファイル』

関連するタスク

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』
386 ページの『デプロイメント記述子値の設定』

関連する参照項目

442 ページの『callLink エlement』
100 ページの『例外処理』
708 ページの『リンケージ・プロパティー・ファイル (詳細)』

リンケージ・プロパティー・ファイル

リンケージ・プロパティー・ファイル は、Java 実行時に使用されるテキスト・ファイルで、生成された Java プログラムまたはラッパーが、生成された Java プログラムを異なるプロセスで呼び出す方法についての詳細を提供します。

このファイルは、Java プログラムまたはラッパーのリンケージ・オプションを生成時ではなく実行時に設定した場合のみ適用できます。ファイルは生成することも、最初から作成することもできます。

ファイルがいつ生成されるか、およびファイル形式の詳細については、『リンケージ・プロパティー・ファイル (詳細)』を参照してください。生成されたファイルの名前の詳細については、『生成される出力 (参照)』を参照してください。デプロイメントの詳細については、『リンケージ・プロパティー・ファイルのデプロイ』を参照してください。

関連する概念

572 ページの『生成される出力』

関連するタスク

395 ページの『リンケージ・プロパティー・ファイルのデプロイ』
384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

573 ページの『生成される出力 (参照)』
428 ページの『genProperties』
708 ページの『リンケージ・プロパティー・ファイル (詳細)』

非 EGL JAR ファイルへのアクセスの提供

EGL 生成 Java コードをデバッグおよび実行するために、非 EGL jar ファイルへのアクセスを提供しなければならない場合があります。それらのファイルへのアクセスを提供するプロセスは、プロジェクト・タイプによって異なります。

アプリケーション・クライアント・プロジェクト

インタープリター型デバッガーを使用する前に、『EGL デバッガーの設定の変更』で説明されているように、CLASSPATH 変数の非 EGL JAR ファイルを参照します。

コードを実行する (EGL Java デバッガーを使用するかどうかに関係なく) 前に、次のようにします。

1. アプリケーション・クライアント・プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトに、ファイル・システムのディレクトリーから必要な JAR ファイルをインポートする。
 - a. 「プロジェクト・エクスプローラー」ビューでエンタープライズ・アプリケーション・プロジェクトを右クリックし、「インポート」をクリックする。
 - b. 「選択」ページで「ファイル・システム」をクリックする。
 - c. 「ファイル・システム」ページで、JAR ファイルが常駐するディレクトリーを指定する。
 - d. ページの右側で、必要な JAR ファイルを選択する。
 - e. 「完了」をクリックする。
2. 実行時にアプリケーション・クライアント・プロジェクトのマニフェストを更新し、エンタープライズ・アプリケーション・プロジェクトの JAR ファイルを使用可能にする。
 - a. 「プロジェクト・エクスプローラー」ビューでアプリケーション・クライアント・プロジェクトを右クリックし、「プロパティー」をクリックする。
 - b. 「プロパティー」ページの左側で「Java JAR 依存関係」をクリックする。
 - c. ページの右側に「Java JAR 依存関係」と呼ばれるセクションが表示されたならば、必要な jar ファイルに対応する各チェック・ボックスを設定する。
 - d. 「OK」をクリックする。

EJB プロジェクト

インタープリター型デバッガーを使用する前に、『EGL デバッガーの設定の変更』で説明されているように、CLASSPATH 変数の非 EGL JAR ファイルを参照します。

コードを実行する (EGL Java デバッガーを使用するかどうかに関係なく) 前に、次のようにします。

1. EJB プロジェクトを参照する各エンタープライズ・アプリケーション・プロジェクトに、ファイル・システムのディレクトリーから必要な JAR ファイルをインポートする。

- a. 「プロジェクト・エクスプローラー」ビューでエンタープライズ・アプリケーション・プロジェクトを右クリックし、「インポート」をクリックする。
 - b. 「選択」ページで「ファイル・システム」をクリックする。
 - c. 「ファイル・システム」ページで、JAR ファイルが常駐するディレクトリーを指定する。
 - d. ページの右側で、必要な JAR ファイルを選択する。
 - e. 「完了」をクリックする。
2. 実行時に EJB プロジェクトのマニフェストを更新し、エンタープライズ・アプリケーション・プロジェクトの JAR ファイルを使用可能にする。
 - a. 「プロジェクト・エクスプローラー」ビューで EJB プロジェクトを右クリックし、「プロパティー」をクリックする。
 - b. 「プロパティー」ページの左側で「Java JAR 依存関係」をクリックする。
 - c. ページの右側に「Java JAR 依存関係」と呼ばれるセクションが表示されたならば、必要な jar ファイルに対応する各チェック・ボックスを設定する。
 - d. 「OK」をクリックする。

Java プロジェクト

インタープリター型デバッガーでコードを実行する前に、『EGL デバッガーの設定の変更』で説明されているように、CLASSPATH 変数の非 EGL JAR ファイルを参照します。

EGL Java デバッガーでコードを実行する前に、プロジェクトの Java のビルド・パスにエントリーを追加します。

1. 「プロジェクト・エクスプローラー」ビューで Java プロジェクトを右クリックし、「プロパティー」をクリックする。
2. 「プロパティー」ページの左側で「Java のビルド・パス」をクリックする。
3. ページの右側に「Java のビルド・パス」セクションが表示されたならば、「ライブラリー」タブをクリックする。
4. 追加される各 JAR ファイルに対して、「外部 JAR の追加」をクリックし、参照メカニズムを使用してファイルを選択する。
5. 「プロパティー」ページを閉じるには、「OK」をクリックする。

J2EE Web プロジェクト

インタープリター型デバッガーを使用する前に、『EGL デバッガーの設定の変更』で説明されているように、CLASSPATH 変数の非 EGL JAR ファイルを参照します。

コードを実行する (EGL Java デバッガーを使用するかどうかに関係なく) 前に、ファイル・システムから以下の Web プロジェクト・フォルダーに jar ファイルをインポートします。

Web Content/WEB-INF/lib

ディレクトリー内の一組の JAR ファイルのインポート・プロセスは、以下のとおりです。

1. 「プロジェクト・エクスプローラー」ビューで Web プロジェクトを展開し、「Web コンテンツ」を展開し、「WEB-INF」を展開して、「lib」を右クリックし、「インポート」をクリックする。
2. 「選択」ページで「ファイル・システム」をクリックする。
3. 「ファイル・システム」ページで、JAR ファイルが常駐するディレクトリーを指定する。
4. ページの右側で、必要な JAR ファイルを選択する。
5. 「完了」をクリックする。

以下の JAR ファイル要件が有効になります。

- MQSeries にアクセスする生成済み Java プログラムでは、MQSeries Classes for Java が必要となります。特に、Java プログラムは、以下の jar ファイルを必要とします (ただし、準備時には必要ありません)。
 - com.ibm.mq.jar
 - com.ibm.mqbind.jar

WebSphere MQ V5.2 がある場合、ソフトウェアは、IBM WebSphere MQ SupportPac™ MA88 にあります。IBM Web サイト (www.ibm.com) で MA88 を検索してください。ソフトウェアをダウンロードして、インストールします。これで、ソフトウェアをインストールしたディレクトリーの Java¥lib サブディレクトリーから JAR ファイルにアクセスすることができます。

WebSphere MQ V5.3 がある場合、同等のソフトウェアを入手するには、カスタム・インストールを行い、Java Messaging を選択します。これで、MQSeries インストール・ディレクトリーの Java¥lib サブディレクトリーから jar ファイルにアクセスすることができます。

- プロトコル CICSJ2C を使用して CICS for z/OS にアクセスする 生成済み Java プログラムまたはラッパーは、実行時にのみ connector.jar および cicsj2ee.jar にアクセスする必要があります。これらのファイルは、CICS Transaction Gateway をインストールすると、使用可能になります。

注: J2EE で EGL Java デバッガーが実行されている場合は、CICS のアクセスが可能となります。ただし、デバッガーが J2EE 以外で実行されている場合、または常に J2EE の外側で実行される EGL インタープリター型デバッガーを実行している場合には、CICS に対する呼び出しの試行は、失敗します。

- SQL テーブルにアクセスする生成済み Java プログラムには、データベース管理システムとともにインストールされるファイルが必要です。
 - DB2 UDB の場合、ファイルは以下のいずれかです。

```
sqllib¥java¥db2java.zip
sqllib¥java¥db2jcc.jar
```

2 番目のファイルは、DB2 UDB の 資料で説明されているように DB2 UDB Version 8 以降で使用可能です。

- Informix の場合、ファイルは以下のとおりです。

```
ifxjdbc.jar
ifxjdbcx.jar
```

- Oracle の場合は、Oracle の資料を参照してください。

データベース・ファイルは実行時に必要であり、準備時には SQL ステートメントを検証するために使用できます。

関連するタスク

120 ページの『EGL デバッガーの設定の変更』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

EGL リファレンス

EGL での代入の互換性

以下の場合には、代入の互換性規則（後で説明します）が適用されます。

- コードがある非参照変数を別の非参照変数に代入する場合、または
- 引数と関数呼び出し内の関連するパラメーターの間でデータを転送する場合に、受信関数のパラメーターに修飾子 IN（この場合は引数がソース）、または OUT（この場合はパラメーターがソース）が付いている場合のみ。ただし、パラメーターがページ・ハンドラーの `onPageLoad` 関数に含まれる場合は、代入の互換性規則は適用されません。この場合の詳細については、『EGL での参照の互換性』を参照してください。

代入の互換性は、以下の型分類に基づいています。

- テキスト型は CHAR、MBCHAR、STRING、UNICODE。
- 数値型は
BIN、INT、BIGINT、SMALLINT、DECIMAL、NUM、NUMBER、FLOAT、SMALLFLOAT、MONEY。
- 日時型は DATE、INTERVAL、TIME、TIMESTAMP。
- HEX は独自のカテゴリーに分類されます。
- VisualAge Generator レガシー型は DBCHAR、NUMC、PACF。各型は VisualAge Generator 規則に準拠します。

代入の互換性規則は以下のとおりです。

- 任意のテキスト型のフィールドは任意のテキスト型のフィールドに代入できる。
- 任意の数値型のフィールドは任意の数値型のフィールドに代入できる。
- 任意の日時型のフィールドは任意のテキスト型または数値型のフィールドに代入できる。
- STRING 型または CHAR 型のフィールドは、HEX 型のフィールドと相互に代入できる。
- CHAR 型のフィールドは NUM 型のフィールドと相互に代入できる。
- 数値型のフィールドをテキスト型のフィールドに代入するには、システム関数 **StrLib.formatNumber** を使用する。
- DATE、TIME、または TIMESTAMP 型のフィールドをテキスト型のフィールドに代入するには、以下の該当するシステム関数を使用する。
 - **StrLib.formatDate** (DATE の場合)
 - **StrLib.formatTime** (TIME の場合)
 - **StrLib.formatTimestamp** (TIMESTAMP の場合)
- テキスト型のフィールドを DATE、TIME、または TIMESTAMP 型のフィールドに代入するには、以下の該当するシステム関数を使用します。
 - **ConverseLib.dateValue** (DATE の場合)

- `ConverseLib.timeValue` (TIME の場合)
- `ConverseLib.timestampValue` (TIMESTAMP の場合)

各種の数値型への代入

任意の数値型 (NUMC および PACF を含む) の値を、任意の数値型とサイズのフィールドに代入できます。EGL は、値をターゲットの形式で保持するために必要な変換を行います。

必要に応じて、無効のゼロが追加または切り捨てられます。(値の整数部分の先頭からのゼロは無効です。同様に、値の小数部の後続桁も無効です。)

いずれの数値型の場合でも、システム変数 `sysVar.overflowIndicator` を使用すると、代入または算術計算によって算術オーバーフローが発生したかどうかをテストできます。また、システム変数 `VGVar.handleOverflow` を設定すると、そうしたオーバーフローの結果を指定できます。

算術オーバーフローが発生した場合、ターゲット・フィールドの値は変更されません。算術オーバーフローが発生しない場合、ターゲット・フィールドに代入される値は、ターゲット・フィールドの宣言に従って位置合わせされます。

NUM 型のフィールドを別のフィールドにコピーするときに、ソース・フィールドの実行時の値が 108.314 であると想定しましょう。

- ターゲット・フィールドで 7 桁 (小数点以下の桁数は 1) を収容できる場合、ターゲット・フィールドは値 000108.3 を受け取り、数値オーバーフローは検出されません。(小数値の精度の情報が失われても、オーバーフローとは見なされません。)
- ターゲット・フィールドで 4 桁 (小数点以下の桁数は 2) を収容できる場合は、数値オーバーフローが検出され、ターゲット・フィールドの値は変更されません。

固定小数点型のフィールドに浮動小数点値 (FLOAT 型または SMALLFLOAT 型) を代入すると、ターゲットの値は必要に応じて切り捨てられます。例えば、ソース値が 108.357 で、固定小数点のターゲットの小数点以下の桁が 1 つの場合、ターゲットは 108.3 を受け取ります。

その他の異なる型への代入

その他の異なる型への代入について、以下に詳しく説明します。

- NUM 型の値を CHAR 型のターゲットに代入する場合は、ソースの宣言に小数部がない場合にのみ、代入が有効となります。この操作は、CHAR から CHAR への代入と同等です。

例えば、ソースの長さが 4 で値が 21 の場合、データの内容は "0021" となり、長さの不一致によってエラー条件は発生しません。

- ターゲットの長さが 5 の場合、値は "0021 " として格納されます (右端に単一バイトのスペースが付加されます)。
- ターゲットの長さが 3 の場合、値は "002" として格納されます (右端の 1 桁が切り捨てられます)。

NUM 型の負の値が CHAR 型の値に代入される場合、フィールドにコピーされる最後のバイトは印刷不能文字となります。

- CHAR 型の値を NUM 型のターゲットに代入する場合は、以下の場合のみ代入が有効となります。
 - ソース (フィールドまたはテキスト式) が数字のみで、他の文字を含まない。
 - ターゲットの宣言に小数部がない。

この操作は、NUM から NUM への代入と同等です。

例えば、ソースの長さが 4 で値が "0021" の場合、データの内容は数値の 21 と等しくなります。長さの不一致があると、次のような結果が生じます。

- ターゲットの長さが 5 の場合、値は 00021 として格納されます (数値のゼロが左端に埋め込まれます)。
 - ターゲットの長さが 3 の場合、値は 021 として格納されます (無効な数字が切り捨てられます)。
 - ターゲットの長さが 1 の場合、値は 1 として格納されます。
- NUMC 型の値を CHAR 型のターゲットに代入する場合は、2 段階の操作で代入が可能になります。値が正の場合は、符号が除去されます。
 1. NUMC の値を NUM 型のターゲットに代入する。
 2. NUM の値を CHAR 型のターゲットに代入する。

NUMC 型のターゲットの値が負の場合、CHAR 型のターゲットにコピーされる最後のバイトは、印刷不能文字となります。

- CHAR 型の値を HEX 型のターゲットに代入する場合は、ソース内の文字が 16 進数文字 (0 から 9、A から F、a から f) の範囲内にある場合にのみ、代入が有効となります。
- HEX 型の値を CHAR 型のターゲットに代入する場合は、数字と英大文字 (A から F) がターゲットに格納されます。
- MONEY 型の値を CHAR 型のターゲットに代入することは無効です。MONEY から CHAR に変換する最良の方法は、システム関数 **strLib.formatNumber** を使用することです。
- NUM 型または CHAR 型の値を DATE 型のターゲットに代入することは、ソース値がマスク `yyyyMMdd` に従った有効な日付である場合にのみ有効です。詳細については、『*DATE*』のトピックを参照してください。
- NUM 型または CHAR 型の値を TIME 型のターゲットに代入することは、ソース値がマスク `hhmmss` に従った有効な時刻である場合にのみ有効です。詳細については、『*TIME*』のトピックを参照してください。
- CHAR 型の値を TIMESTAMP 型のターゲットに代入することは、ソース値が TIMESTAMP フィールドのマスクに従った有効なタイム・スタンプである場合にのみ有効です。以下に例を示します。

```
// 2 月 30 日は有効な日付でないので無効
myTS timestamp("yyyyMMdd");
myTS = "20050230";
```

フル・マスクの先頭の文字が欠落している場合 (例えば、マスクが「dd」の場合)、EGL は、上位レベルの文字 (この例では「yyyyMM」) が、マシン・クロックに従って現在の瞬間を表すものと見なします。次の文を使用すると、2 月にランタイム・エラーが発生します。

```
// 2 月に実行された場合は無効
myTS timestamp("dd");
myTS = "30";
```

- TIME 型または DATE 型の値を NUM 型のターゲットに代入することは、NUM から NUM への代入と等価です。
- TIME、DATE、または TIMESTAMP 型の値を CHAR 型のターゲットに代入することは、CHAR から CHAR への代入と等価です。

文字型に関する埋め込みと切り捨て

ターゲットが非 STRING 文字型 (DBCHAR および HEX を含む) であり、ソース値を格納するのに必要とされる以上のスペースがある場合は、EGL は右側にデータを埋め込みます。

- CHAR 型または MBCHAR 型のターゲットの埋め込みには、単一バイトの空白が使用されます。
- DBCHAR 型のターゲットの埋め込みには、2 バイトの空白が使用されます。
- UNICODE 型のターゲットの埋め込みには、ユニコードの 2 バイトの空白が使用されます。
- HEX 型のターゲットの埋め込みには、2 進ゼロが使用されます。例えば、ソース値が "0A" であるなら、2 バイトのターゲットに "000A" ではなく "0A00" として値が格納されます。

ソース値を格納するためのスペースが文字型のターゲットで不足している場合は、EGL によって右側の値が切り捨てられます。通知されるエラーはありません。以下の場合には、特殊な問題が発生することがあります。

- ランタイム・プラットフォームで EBCDIC 文字セットがサポートされている
- assignment ステートメントにより、MBCHAR 型のリテラルまたは MBCHAR 型の項目が、より短い MBCHAR 型の項目にコピーされる
- 1 バイトずつ切り捨てを行うと、最後のシフトイン文字が除去されたり、DBCHAR 文字が分割されたりする

このような場合、EGL は、MBCHAR 型の有効なストリングがターゲット項目に格納されるように文字を切り捨て、必要があれば、終端に単一バイトの空白を付加します。

タイム・スタンプ間の代入

TIMESTAMP 型の項目を別の TIMESTAMP 型のフィールドに代入する場合は、以下の規則が適用されます。

- ソース・フィールドのマスクに、ターゲット・フィールドが必要とする相対的に上位レベルのエントリーが欠落している場合、それらのターゲット・エントリーには、代入時点でのマシンのクロックに従って代入が行われます。次に例を示します。

```

- sourceTimeStamp timestamp ("MMdd");
  targetTimeStamp timestamp ("yyyyMMdd");

  sourceTimeStamp = "1201";

  // このコードが 2004 年に実行された場合、次の文が
  // targetTimeStamp に 20041201 を代入する
  targetTimeStamp = sourceTimeStamp;
- sourceTimeStamp02 timestamp ("ssff");
  targetTimeStamp02 timestamp ("mmssff");

  sourceTimeStamp02 = "3201";

  // 次の代入は、分を含んでいる
  // それは、assignment ステートメントが実行されたときの現行の分である
  targetTimeStamp02 = sourceTimeStamp02;
- ソース項目のマスクに、ターゲット・フィールドが必要とする相対的に下位レ
  ベルのエントリーが欠落している場合、それらのターゲット・エントリーに
  は、有効な最低の値が代入されます。次に例を示します。
- sourceTimeStamp timestamp ("yyyyMM");
  targetTimeStamp timestamp ("yyyyMMdd");

  sourceTimeStamp = "200412";

  // 日に関係なく、次の文は
  // targetTimeStamp に 20041201 を代入する
  targetTimeStamp = sourceTimeStamp;
- sourceTimeStamp02 timestamp ("hh");
  targetTimeStamp02 timestamp ("hhmm");

  sourceTimeStamp02 = "11";

  // 分に関係なく、次の文は
  // targetTimeStamp02 に 1100 を代入する
  targetTimeStamp02 = sourceTimeStamp02;

```

固定構造内の副構造化フィールドへの代入、または副構造化フィールドからの代入

副構造化フィールドを副構造化されていないフィールドに代入したり、その逆を行ったりすることができます。また、2 つの副構造化フィールドの間で値を代入できます。例えば、*myNum* および *myRecord* という名前の変数が、次のパーツに基づいているものとします。

```

DataItem myNumPart
  NUM(12)
end

Record myRecordPart type basicRecord
  10 topMost CHAR(4);
  20 next01 HEX(4);
  20 next02 HEX(4);
end

```

数学的システム変数の範囲外では、HEX 型の値を NUM 型の項目に代入することは無効です。ただし、**myNum = topMost** という書式の代入は、**topMost** が CHAR 型であるため、有効です。一般に、代入は、assignment 文内のフィールドのプリミティブ型に基づいて実行され、従属項目のプリミティブ型は考慮されません。

副構造のある項目のプリミティブ型は、デフォルトで `CHAR` に設定されています。副構造化フィールドにデータを代入したり、副構造化フィールドからデータを代入したりする場合、宣言時に別のプリミティブ型を指定しないと、`CHAR` 型のフィールドに関する前述の規則が代入時に適用されます。

固定レコードの代入

固定レコードから別の固定レコードへの代入は、`CHAR` 型の副構造化項目から別の `CHAR` 型の副構造化項目への代入と同等です。長さの不一致があると、受け取った値の右端に単一バイトのブランクが付加されたり、受け取った値の右端から 1 バイトの文字が除去されたりします。代入では、従属構造フィールドのプリミティブ型は考慮されません。

以下の例外が適用されます。

- レコードの内容を代入できるのは、レコードまたは、`CHAR` 型、`HEX` 型、または `MBCHAR` 型のフィールドです。その他の型のフィールドには代入できません。
- レコードが受け取ることができるのは、レコードからのデータ、文字列リテラルからのデータ、または `CHAR` 型、`HEX` 型、または `MBCHAR` 型のフィールドからのデータです。数値リテラルからのデータ、または `CHAR` 型、`HEX` 型、`MBCHAR` 型以外のフィールドからのデータを受け取ることはできません。

最後に、`SQL` レコードと、異なる型のレコードとの間で代入を行う場合は、各構造化フィールドに先行する 4 バイトの領域のためのスペースを非 `SQL` レコードに確保する必要があります。

関連する概念

207 ページの『ページ・ハンドラー』

関連する参照項目

407 ページの『代入』

564 ページの『関数パラメーター』

570 ページの『EGL ソース形式の関数パーツ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

781 ページの『プログラム・パラメーター』

783 ページの『EGL ソース形式のプログラム・パーツ』

関連する参照項目

44 ページの『`DATE`』

93 ページの『EGL ステートメント』

940 ページの『`formatNumber()`』

1012 ページの『`handleOverflow`』

657 ページの『`move`』

997 ページの『`overflowIndicator`』

37 ページの『プリミティブ型』

810 ページの『サブストリング』

47 ページの『`TIME`』

代入

EGL 割り当ては、あるメモリー領域から別のメモリー領域にデータをコピーします。また、数式やテキスト式の結果をソース・フィールドにコピーできます。

▶▶ `target = source ;` ▶▶

target フィールド、レコード、固定レコード、またはシステム変数

ターゲット・フィールドが CHAR 型、DBCHAR 型、または UNICODE 型の場合に、assignment ステートメントの左側にサブストリングを指定できます。サブストリング領域には値が入力されますが (必要に応じてブランクが埋め込まれます)、割り当てられたテキストはサブストリング領域外に拡張されず、必要に応じて切り捨てられます。構文の詳細については、『サブストリング』を参照してください。

source レコード、固定レコード、または数値あるいは文字式

代入の例を以下に示します。

```
z = a + b + c;  
myDate = VGVar.currentShortGregorianDate;  
myUser = sysVar.userID;  
myRecord01 = myRecord02;  
myRecord02 = "USER";
```

EGL の assignment ステートメントの振る舞いは、*move* で説明している **move** ステートメントと異なります。

割り当て規則は『EGL の割り当ての互換性』で説明しています。

関連する概念

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

401 ページの『EGL での代入の互換性』

657 ページの『move』

810 ページの『サブストリング』

関連エレメント

『リソース関連パーツ』で説明されているとおり、リソース関連パーツは関連要素で構成されています。各要素はファイル名 (プロパティー 408 ページの『fileName』) に固有であり、一連のエントリーを含んでいます。各エントリーは、以下のプロパティーを持ちます。

- 409 ページの『system』
- 408 ページの『fileType』

system および **fileType** プロパティーの値により、以下のリストから追加のプロパティーが選択可能です。

- 408 ページの『commit』
- 408 ページの『conversionTable』

- 『formFeedOnClose』
- 409 ページの 『replace』
- 409 ページの 『systemName』
- 410 ページの 『text』

commit

コミットメント制御を使用可能にするかどうかを示します (iSeries 上の EGL で生成された Java プログラム用)。

次の値から 1 つを選択します。

NO (デフォルト)

sysLib.commit または sysLib.rollback の使用は影響を与えません。

YES

sysLib.commit および sysLib.rollback を使用して作業論理単位の終了を定義することができます。

conversionTable

MQSeries メッセージ・キューのアクセス時に、生成された Java プログラムで使用する変換テーブルの名前を指定します。

追加情報については、『データ変換』を参照してください。

fileType

ターゲット・システムでファイル編成を指定します。*seqws* などの明示的タイプを選択することができます。または、値 *default* を選択することができます。この値はプロパティ **fileType** のデフォルト値です。デフォルトを使用すると、ファイル・タイプが以下のように自動的に選択されます。

- ターゲット・システムと EGL レコード・タイプの特定の組み合わせの場合
- ファイル名が *printer* である印刷出力の場合

『レコードとファイル・タイプの相互参照』では、*default* を選択した場合に使用される値だけでなく、明示的 **fileType** 値を示します。

fileName

1 つ以上のレコードで指定される論理ファイル名を指します。この名前を 1 つ以上のターゲット・システムの物理リソースに関連付ける関連要素を作成します。(印刷出力の場合は、値 *printer* を指定します。)

論理ファイル名にはグローバル置換文字としてアスタリスク (*) を使用できます。ただし、この文字は最後の文字としてのみ有効です。詳細については、『リソース関連とファイル・タイプ』を参照してください。

formFeedOnClose

印刷書式の出力が終了したときに用紙送りが発行されたかどうかを示します。(コードが **print** ステートメントを発行すると、印刷書式が生成されます。)

このプロパティは、以下のいずれかの場合で **fileName** 値が *printer* である場合にのみ使用可能です。

- **system** 値が *aix*、*iSeriesj*、または *linux* であり、**fileType** 値が *seqws* または *spool* である。
- **system** 値が *win* で、**fileType** 値が *seqws* である。

次の値から 1 つを選択します。

YES

用紙送りが行われる (デフォルト)

NO

用紙送りが行われない

replace

ファイルにレコードを追加する場合、ファイル全体が置き換えられるのか、レコードがファイルの終わりに付加されるのかを指定します。このエントリは、以下の場合にのみ使用されます。

- Java コードを生成している。
- レコードがファイル・タイプ **seqws** である。

次の値から 1 つを選択します。

NO

ファイルへの付加 (デフォルト)

YES

ファイルの置換

system

ターゲット・プラットフォームを指定します。次の値から 1 つを選択します。

aix

AIX

iseriesj

iSeries

linux

Linux

win

Windows 2000/NT/XP

any

ターゲットのプラットフォームを問わない。詳細については、『リソース関連とファイル・タイプ』を参照してください。

systemName

ファイル名に関連付けられたファイルまたはデータ・セットのシステム・リソース名を指定します。値の中にスペースまたは以下の文字のいずれかが含まれる場合には、値を単一引用符または二重引用符で囲みます。

% = , () /

text

生成された Java プログラムがシリアル・レコードを介してファイルにアクセスするときに、次の処理を実行させるかどうかを指定します。

- **add** 操作中に、行の終わりを示す文字を付加する。UNIX 以外のプラットフォームでは、付加される文字は復帰文字と改行文字です。UNIX プラットフォームでは、改行文字が唯一の付加対象文字です。
- **get** または **get next** の操作中に、行の終わりを示す文字を除去する。

次の値から 1 つを選択します。

NO

デフォルトでは、行の終わりを示す文字の付加も除去も行いません。

YES

生成されたプログラムが、レコードが行の終わりを示す文字で終わることを前提としているプロダクトとのデータ交換を行う場合に便利であるように、デフォルトを変更します。

関連する概念

331 ページの『リソース関連とファイル・タイプ』

関連タスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

336 ページの『EGL ビルド・ファイル内のリソース関連パーツの編集』

338 ページの『EGL ビルド・ファイルからのリソース関連パーツの除去』

関連する参照項目

507 ページの『データ変換』

579 ページの『入出力エラー値』

793 ページの『レコードとファイル・タイプの相互参照』

asynchLink エlement

リンケージ・オプション・パーツの *asynchLink* Element は、生成済みプログラムが、発信元プログラムがシステム関数 `sysLib.startTransaction` を呼び出すときに、別のプログラムを非同期に呼び出す方法を指定します。

デフォルトの振る舞いを受け入れる場合は、*asynchLink* Element の指定を省略できます。この場合は、作成されたトランザクションは、同一の Java パッケージから開始されることが前提です。

それぞれのElementには、プロパティ `recordName` が含まれています。このプロパティは、アクションが変更される特定の `sysLib.startTransaction` 関数でも参照されるレコードを参照します。

他のプロパティは、**package** です。このプロパティは、呼び出されるプログラムのソースが、起動側のパッケージとは異なるパッケージに入っている場合のみ必要です。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

412 ページの『asynchLink エlementの package』

412 ページの『asynchLink エlementの recordName』

リモート呼び出し用 csouidpwd.properties ファイル

後述される状態では、ユーザーは **csouidpwd.properties** ファイルを作成し、そのファイルにアクセスできるようにする必要があります。このファイルには、Java プログラムまたはラッパーからのリモート呼び出しのために必要な認証の詳細が含まれます。

状態は以下のとおりです。

- リンケージ・オプション・パーツである **callLink** Elementのプロパティ **remoteComType** は、JAVA400、CICSJ2C または、CICSECI に設定されていて、
- ユーザー ID およびパスワードが必須であり、
- 次のいずれかのケースが当てはまる。
 - Java プログラムから呼び出しが行われるが、コードは、ブランク以外の値のシステム関数 **SysLib.setRemoteUser** を最初に呼び出さない。または、
 - Java ラッパーから呼び出しが行われるが、そのラッパーを含む Java コードが、ブランク以外の値の CSOCallOptions メソッド **setUserId** および **setPassword** を呼び出さない。

SysLib.setRemoteUser の呼び出し (または、適切な CSOCallOptions メソッドの呼び出し) では、ブランクのユーザー ID およびパスワードが提供され、同等のプロパティの値が **csouidpwd.properties** から 検索されます。

ユーザーのタスクは以下のとおりです。

1. 以下のとおりに (各行に分かれて)、フォーマットされるプロパティ設定を含むことが出来る **csouidpwd.properties** ファイルを作成します。

CSOUID=userid

userid は、リモート呼び出し用のユーザー ID です

CSOPWD=password

password は、リモート呼び出し用のパスワードです

2. ファイルがクラスパスによって参照されるディレクトリーであることを確認します。適切なディレクトリーは、ユーザーのプロジェクトの **JavaSource** フォルダです。

関連する概念

327 ページの『Java ラッパー』

関連する参照項目

595 ページの『Java ラッパー・クラス』
456 ページの『callLink エlementの remoteComType』
972 ページの『setRemoteUser()』

asynchLink エlementの package

リンケージ・オプション・パーツの asynchLink エlementの **package** プロパティは、呼び出されるプログラムを含んでいるパッケージの名前を指定します。デフォルトは、呼び出し側プログラムのパッケージです。

生成される Java プログラムで使用されるパッケージ名は EGL プログラムのパッケージ名ですが、小文字で指定します。EGL が asynchLink エlementから出力を生成する場合は、**package** の値が (必要な場合) 小文字に変更されます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

410 ページの『asynchLink エlement』

『asynchLink エlementの recordName』

asynchLink エlementの recordName

リンケージ・オプション・パーツの asynchLink エlementの **recordName** プロパティは、システム関数 sysLib.startTransaction で使用されるレコードの名前を指定します。この場合には、レコード名は、asynchLink エlementにどのプログラムまたはトランザクションが関連付けられているかを識別するために使用します。

レコード名にはグローバル置換文字としてアスタリスク (*) を使用できます。ただし、この文字は最後の文字としてのみ有効です。詳細については、『リンケージ・オプション・パーツ』を参照してください。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

410 ページの『asynchLink エlement』

『asynchLink エlementの package』

975 ページの『startTransaction()』

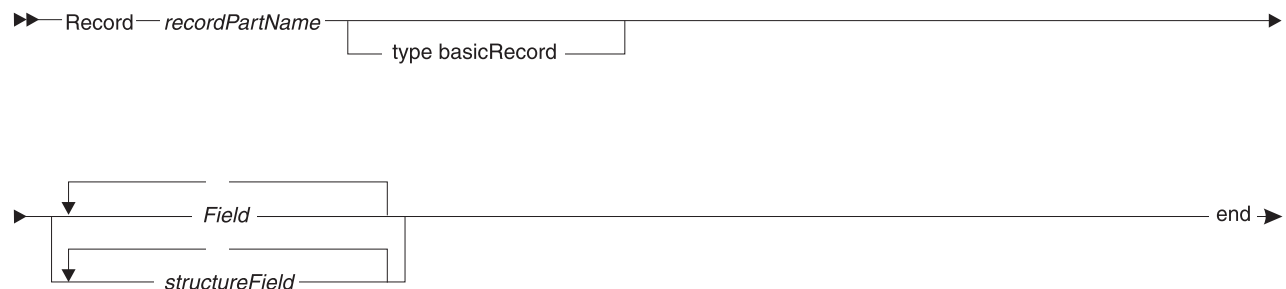
EGL ソース形式の基本レコード・パーツ

EGL ファイルでタイプ `basicRecord` のレコード・パーツを宣言します。これについては、『EGL ソース形式』で説明しています。

基本レコード・パーツの例を次に示します。

```
Record myBasicRecordPart type basicRecord
  10 myField01 CHAR(2);
  10 myField02 CHAR(78);
end
```

基本レコードの構文図は、以下のとおりです。



Record *recordPartName* **basicRecord**

パーツをタイプ `basicRecord` として識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

field

レコード内で該当する変数。詳細については、『レコード・パーツ』を参照してください。各変数宣言はセミコロンで終了します。

structureField

固定構造体フィールド。詳細については、『EGL ソース形式の構造体フィールド』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 142 ページの『固定レコード・パーツ』
- 24 ページの『パーツの参照』
- 19 ページの『パーツ』
- 141 ページの『レコード・パーツ』
- 62 ページの『EGL での変数の参照』
- 30 ページの『Typedef』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 512 ページの『EGL ソース形式の `DataItem` パーツ』
- 532 ページの『EGL ソース形式』
- 570 ページの『EGL ソース形式の関数パーツ』
- 578 ページの『EGL ソース形式の索引付きレコード・パーツ』

714 ページの『EGL ソース形式の MQ レコード・パーツ』
725 ページの『命名規則』
37 ページの『プリミティブ型』
783 ページの『EGL ソース形式のプログラム・パーツ』
793 ページの『可変長レコードをサポートするプロパティ』
796 ページの『EGL ソース形式の相対レコード・パーツ』
799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』
808 ページの『EGL ソース形式の構造体フィールド』

ビルド・パーツ

EGL ビルド・ファイル形式

.eglbld ファイルの構造を次に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGL PUBLIC "-//IBM//DTD EGL 5.1//EN" "">
<EGL>
  <!-- import ステートメントはここに配置します -->
  <!-- パーツはここに配置します -->
</EGL>
```

ここでの作業内容は、<EGL> 要素の内側に import ステートメントとパーツを配置することです。

<import> 要素を指定して、チェーン内の次のビルド記述子が含まれているファイルを参照するか、ビルド記述子によって参照されるビルド・パーツを参照します。

import ステートメントの例を次に示します。

```
<import file="myBldFile.eglbld"/>
```

次のリストからパーツを宣言します。

- <BuildDescriptor>
- <LinkageOptions>
- <ResourceAssociations>

以下に簡単な例を示します。

```
<EGL>
<import file="myBldFile.eglbld"/>
<BuildDescriptor name="myBuildDescriptor"
  genProject="myNextProject"
  system="WIN"
  J2EE="NO"
  genProperties="GLOBAL"
  genDataTables="YES"
  dbms="DB2"
  sqlValidationConnectionURL="jdbc:db2:SAMPLE"
  sqlJDBCDriverClass="COM.ibm.db2.jdbc.app.DB2Driver"
  sqlDB="jdbc:db2:SAMPLE"
</BuildDescriptor>
</EGL>
```

ビルド・ファイル DTD は、確認することができます。ビルド・ファイル DTD は、次のサブディレクトリーに入っています。

```
installationDir¥egl¥eclipse¥plugins¥  
com.ibm.etools.egl_version¥dtd
```

installationDir

製品のインストール・ディレクトリー。例えば、C:¥Program Files¥IBM¥RSPD¥6.0 など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

ファイル名 (egl_wssd_6_0.dtd など) は、文字 *egl* およびアンダースコアで開始します。*wssd* の文字は Rational Web デベロッパーおよび Rational Application Developer を指します。*wsed* の文字は、Rational Application Developer for z/OS を指し、*wdsc* の文字は Rational Application Developer for iSeries を指します。

関連する概念

36 ページの『インポート』

19 ページの『パーツ』

関連するタスク

135 ページの『EGL ソース・ファイルの作成』

関連する参照項目

523 ページの『EGL エディター』

ビルド記述子オプション

次の表は、すべてのビルド記述子オプションをまとめたものです。

ビルド記述子オプション	ビルド・オプション・ フィルター	説明
buildPlan	• Java ターゲット	ビルド計画を作成するかどうかを指定します。
cicsj2cTimeout	• デバッグ • Java ターゲット • Java iSeries	Java ランタイム・プロパティー cso.cicsj2c. timeout に値を割り当てます。このプロパティーは、プロトコル CICSJ2C を使用する呼び出しにおいて、タイムアウトが発生するまでの時間をミリ秒単位で指定します。
commentLevel	• Java ターゲット • Java iSeries	出力ソース・コードに、EGL システム・コメントをどの程度含めるかを指定します。
currencySymbol	• デバッグ • Java ターゲット	1 から 3 文字で構成される通貨記号を指定します。

ビルド記述子オプション	ビルド・オプション・ フィルター	説明
dbms	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	生成されたプログラムがアクセスするデータベースのタイプを指定します。
decimalSymbol	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	Java ランタイム・プロパティ vgj.nls.number.decimal に文字を割り当てます。このプロパティは、10 進記号として使用する文字を指定します。
destDirectory	<ul style="list-style-type: none"> • Java ターゲット 	準備の出力を保管するディレクトリーの名前を指定します (Java を生成する場合のみ)。
destHost	<ul style="list-style-type: none"> • Java ターゲット 	ビルド・サーバーが存在するターゲット・マシンの名前または TCP/IP アドレス (数値) を指定します。
destPassword	<ul style="list-style-type: none"> • Java ターゲット 	準備を実行するマシンにログオンするために EGL で使用するパスワードを指定します。
destPort	<ul style="list-style-type: none"> • Java ターゲット 	リモート・ビルド・サーバーがビルド要求を listen するポートを指定します。
destUserID	<ul style="list-style-type: none"> • Java ターゲット 	準備を実行するマシンにログオンする場合は、EGL で使用するユーザー ID を指定します。
eliminateSystemDependentCode	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	検証時に、ターゲット・システムで実行されないコードを EGL が無視するかどうかを示します。
enableJavaWrapperGen	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	Java ラッパー・クラスの生成を許可するかどうかを指定します。
genDataTables	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	データ・テーブルを生成するかどうかを示します。
genDirectory	<ul style="list-style-type: none"> • Java ターゲット 	生成済み出力ファイルと準備状況ファイルが格納されるディレクトリーの完全修飾パスを指定します。
genFormGroup	<ul style="list-style-type: none"> • Java ターゲット 	生成しているプログラムの使用宣言内で参照される書式グループを生成するかどうかを示します。

ビルド記述子オプション	ビルド・オプション・ フィルター	説明
genHelpFormGroup	<ul style="list-style-type: none"> • Java ターゲット 	生成しているプログラムの使用宣言内で参照されるヘルプ書式グループを生成するかどうかを示します。
genProject	<ul style="list-style-type: none"> • Java ターゲット 	Java 生成の出力をワークベンチ・プロジェクトに格納し、Java ランタイムのセットアップに必要なタスクを自動化します。
genProperties	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	生成される Java ランタイム・プロパティがある場合には、その種類を指定します。また、リンケージ・プロパティ・ファイルを生成するかどうかを指定することもあります。
itemsNullable	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	コードがプリミティブ・フィールドを NULL に設定できる環境を指定します。
J2EE	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	J2EE 環境で動作する Java プログラムを生成するかどうかを指定します。
linkage	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	生成の各側面を処理するリンケージ・オプション・パーツ名を指定します。
nextBuildDescriptor (『ビルド記述子パーツ』を参照)	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	チェーン内の次のビルド記述子を指定します。
prep	<ul style="list-style-type: none"> • Java ターゲット 	戻りコード <= 4 で生成が完了したときに EGL で準備を開始するかどうかを指定します。
resourceAssociations	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	ターゲット・プラットフォーム上のファイル/キューとレコード・パーツとを関連付けるリソース関連パーツ名を指定します。
sessionBeanID	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	J2EE デプロイメント記述子内の既存のセッション要素の名前を指定します。

ビルド記述子オプション	ビルド・オプション・ フィルター	説明
sqlCommitControl	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	デフォルトのデータベースに変更が加えられる度にコミットするかどうかを指定する Java ランタイム・プロパティの生成が可能になります。
sqlDB	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	生成したプログラムで使用するデフォルト・データベースを指定します。
sqlID	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	ユーザー ID を指定します。 このユーザー ID は、生成時に SQL ステートメントを検証する際、または実行時に、データベースと接続するために使用されます。
sqlJDBCDriverClass	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	ドライバー・クラスを指定します。このドライバー・クラスは、生成時に SQL ステートメントを検証したり、非 Java デバッグ・セッション中に、データベースと接続するときに使用されます。
sqlJNDIName	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	J2EE で実行される、生成された Java プログラムで使用するデフォルト・データベースを指定します。
sqlPassword	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	パスワードを指定します。このパスワードは、生成時に SQL ステートメントを検証する際、または実行時にデータベースと接続するときに使用されます。
sqlValidationConnectionURL	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	URL を指定します。この URL は、生成時に SQL ステートメントを検証する際に、データベースと接続するために使用されます。
system	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	生成される出力のカテゴリーを指定します
targetNLS	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	ランタイムの出力で使用するターゲット各国語コードを指定します。

ビルド記述子オプション	ビルド・オプション・ フィルター	説明
VAGCompatibility	<ul style="list-style-type: none"> • デバッグ • Java ターゲット • Java iSeries 	生成プロセスにおいて特殊なプログラム構文の使用を許可するかどうかを示します。
validateSQLStatements	<ul style="list-style-type: none"> • Java ターゲット • Java iSeries 	SQL ステートメントをデータベースと突き合わせて検証するかどうかを指定します。

関連する概念

319 ページの『ビルド記述子パーツ』

377 ページの『Java ランタイム・プロパティー』

関連するタスク

324 ページの『EGL ビルド・ファイルへのビルド記述子パーツの追加』

325 ページの『ビルド記述子の汎用オプションの編集』

関連する参照項目

584 ページの『Java ランタイム・プロパティー (詳細)』

buildPlan

ビルド記述子オプション **buildPlan** は、ビルド計画を作成するかどうかを指定します。有効な値は YES と NO です。デフォルトは、YES です。

ビルド計画は、ビルド記述子オプション **genDirectory** で指定したディレクトリーに格納されます。

Java コードをプロジェクト内に生成するときは、特別な事例が有効です。その場合は、**buildPlan** の設定にかかわらず、ビルド計画は作成されません。ただし、次のいずれかの状況で、準備は実行されます。

- プロジェクトを再ビルドするとき。
- ソース・ファイルを生成するとき。(ただし、ワークベンチ設定「リソース変更時にビルドを自動的に実行」を確認した場合のみ。)

ビルド計画を作成し、時間が経過してからその計画を呼び出すこともできます。詳細については、『生成後のビルド計画の呼び出し』を参照してください。

関連する概念

356 ページの『ビルド計画』

関連するタスク

363 ページの『生成後のビルド計画の呼び出し』

関連する参照項目

415 ページの『ビルド記述子オプション』

cicsj2cTimeout

Java コードを生成する際、ビルド記述子オプション **cicsj2cTimeout** は、Java ランタイム・プロパティ **cso.cicsj2c.timeout** に値を割り当てます。このプロパティにより、CICSJ2C プロトコルを使用した呼び出し時に、タイムアウトとするまでの時間数をミリ秒で指定します。

このランタイム・プロパティのデフォルト値は 30000 (30 秒) です。値を 0 に設定すると、タイムアウトは発生しません。値には、0 以上を設定する必要があります。

WebSphere 390 で呼び出し先プログラムを実行する場合、プロパティ **cso.cicsj2c.timeout** は、呼び出しに影響を与えません。詳細については、『*CICSJ2C 呼び出しにおける J2EE サーバーの設定*』を参照してください。

関連する概念

377 ページの『Java ランタイム・プロパティ』

関連するタスク

389 ページの『CICSJ2C 呼び出し用 J2EE サーバーのセットアップ』

関連する参照項目

415 ページの『ビルド記述子オプション』

584 ページの『Java ランタイム・プロパティ (詳細)』

commentLevel

ビルド記述子オプション **commentLevel** は、出力ソース・コードに EGL システム・コメントをどの程度含めるのかを指定します。

有効な値は以下のとおりです。

- 0 最小限のコメントを出力に含めます。(EGL が生成する名前の別名に関するコメントも含まれます)
- 1 レベル 0 で含まれるコメントの他に、スクリプト文を実装するために生成されるコードの直前に、スクリプト文が記述されます。

デフォルトは 1 です。

コメント・レベルを上げても、準備済みコードのサイズやパフォーマンスに影響はありません。ただし、出力のサイズは大きくなり、出力の生成、転送、および準備に要する時間は長くなります。

関連する参照項目

415 ページの『ビルド記述子オプション』

currencySymbol

ビルド記述子オプション **currencySymbol** は、Java 出力の場合のみ使用可能で、1 文字から 3 文字で構成される通貨記号を指定します。このオプションを指定しない場合、デフォルト値は、出力を生成するシステムのロケールから派生します。

キーボード上にない文字を指定するには、「Alt (前面)」キーを押して、数字キーボードを使用して、文字の 10 進数コードを入力します。例えば Euro の 10 進コードは、Windows 2000/NT/XP では 0128 です。

関連する概念

319 ページの『ビルド記述子パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

dbms

ビルド記述子オプション **dbms** は、生成されたプログラムがアクセスするデータベースのタイプを指定します。次の値から 1 つを選択します。

- DB2 (デフォルト値)
- INFORMIX
- ORACLE

関連する参照項目

415 ページの『ビルド記述子オプション』

271 ページの『Informix および EGL』

decimalSymbol

Java コードを生成する場合、ビルド記述子オプション **decimalSymbol** は、Java ランタイム・プロパティー **vgj.nls.number.decimal** に文字を割り当てます。このプロパティーは、10 進記号として使用する文字を指定します。ビルド記述子オプション **decimalSymbol** を指定しない場合、10 進記号として使用する文字は、Java ランタイム・プロパティー **vgj.nls.code** に関連付けられているロケールによって決まります。

この値は、1 文字で指定する必要があります。

関連する概念

377 ページの『Java ランタイム・プロパティー』

関連する参照項目

415 ページの『ビルド記述子オプション』

584 ページの『Java ランタイム・プロパティー (詳細)』

destDirectory

ビルド記述子オプション **destDirectory** は、準備の出力を保管するディレクトリーを指定します。このオプションは、プロジェクト内ではなく、ディレクトリー内に生成する場合にのみ有効です。

完全修飾ファイル・パスを指定するときは、最後のディレクトリー以外はすべて存在していなければなりません。例えば、Windows 2000 上で c:\buildout を指定した場合、buildout ディレクトリーが存在しないときは、EGL によって buildout ディレクトリーが作成されます。しかし、c:\interim\buildout を指定したときに interim ディレクトリーが存在しない場合は、準備が失敗します。

相対ディレクトリー (USS 上の myid/mysource など) を指定した場合、出力は、最下位のディレクトリーに格納されます。これは、次に説明するデフォルト・ディレクトリーを基準とした相対ディレクトリーとなります。

destDirectory のデフォルト値は、ビルド記述子オプション **destHost** の状況の影響を受けます。

- **destHost** を指定した場合、**destDirectory** のデフォルト値は、ビルド・サーバーが開始されたディレクトリーとなります。
- **destHost** を指定しない場合は、生成を実行するマシン上で準備が実行されます。**destDirectory** のデフォルト値は、ビルド記述子オプション **genDirectory** によって提供されます。

ビルド記述子オプション **destUserID** で指定されたユーザーは、準備の出力を受け取るディレクトリーに対して、書き込み権限を持つ必要があります。

UNIX 変数 (\$HOME など) を使用して、USS 上のディレクトリー構造のパーツを識別することはできません。

関連する参照項目

415 ページの『ビルド記述子オプション』

『destHost』

427 ページの『genProject』

destHost

ビルド記述子オプション **destHost** は、ビルド・サーバーが存在するターゲット・マシンの名前または TCP/IP アドレス (数値) を指定します。デフォルト値はありません。

Java 出力を準備する場合は、次のことが該当します。

- **destHost** はオプション。
- **destHost** は、プロジェクト内ではなく、ディレクトリー内に生成する場合にのみ有効。
- **destDirectory** を指定せずに **destHost** を指定した場合は、ビルド・サーバーが開始されたディレクトリーが、ソースおよび準備出力を受け取るディレクトリーとなる。
- **destHost** を指定しない場合は、生成が実行されるマシン上で準備が実行される。**destDirectory** を指定しない場合は、ビルド記述子オプション **genDirectory** で指定されたディレクトリーが、ソースおよび準備出力を受け取るディレクトリーとなる。
- UNIX 環境には、大文字小文字の区別がある。

名前または TCP/IP アドレスには、最大 64 文字まで入力できます。Windows NT® 上で開発を行う場合は、TCP/IP アドレスではなく、名前を指定する必要があります。

destHost の値の例を次に 2 つ示します。

abc.def.ghi.com

9.99.999.99

関連する参照項目

415 ページの『ビルド記述子オプション』

421 ページの『destDirectory』

『destPassword』

『destPort』

destPassword

ビルド記述子オプション **destPassword** は、準備を実行するマシンにログオンするために EGL で使用するパスワードを指定します。

このオプションおよび本ページでの説明は、プロジェクト内ではなくディレクトリー内に生成する場合、およびビルド記述子オプション **destHost** の値を指定する場合にのみ有効です。

パスワードは、ビルド記述子オプション **destUserID** で指定されたユーザー ID のアクセスを提供します。すべてのターゲット・システムで、パスワードの値に大文字小文字の区別があります。

デフォルト値はありません。

destPassword を使用すると、EGL ビルド・ファイルにパスワードが保管されます。セキュリティー・リスクを避けるには、ビルド記述子オプションを設定しないようにします。生成を開始すると、対話式生成ダイアログまたはコマンド行でパスワードを設定できます。

関連する参照項目

415 ページの『ビルド記述子オプション』

422 ページの『destHost』

『destUserID』

destPort

ビルド記述子オプション **destPort** は、リモート・ビルド・サーバーがビルド要求を listen するポートを指定します。

このオプションは、プロジェクト内ではなくディレクトリー内に生成する場合、およびビルド記述子オプション **destHost** の値を指定する場合にのみ有効です。

デフォルト値はありません。

関連する参照項目

415 ページの『ビルド記述子オプション』

422 ページの『destHost』

destUserID

ビルド記述子オプション **destUserID** は、準備を実行するマシンにログオンするために EGL で使用するユーザー ID を指定します。

このオプションおよび本ページでの説明は、プロジェクト内ではなくディレクトリー内に生成する場合、およびビルド記述子オプション **destHost** の値を指定する場合にのみ有効です。

destUserID で指定されたユーザーは、ディレクトリーに対して、書き込み権限を持つ必要があります。すべてのターゲット・システムで、オプションの値に大文字小文字の区別があります。

デフォルト値はありません。

関連する参照項目

415 ページの『ビルド記述子オプション』

422 ページの『destHost』

423 ページの『destPassword』

eliminateSystemDependentCode

ビルド記述子オプション **eliminateSystemDependentCode** は、EGL が、検証時にターゲット・システムで実行されないプログラムを無視するかどうかを指定します。有効な値は *yes* (デフォルト) と *no* です。現在の生成における出力が、複数のシステムで実行される場合にのみ、*no* を指定します。

オプション **eliminateSystemDependentCode** は、システム関数 **sysVar.systemType** との関連でのみ有効です。この関数は、生成時にどのコードが検証されるかには影響を与えません。例えば、次の **add** ステートメントは、Windows 用に生成している場合でも、検証は行われます。

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

ターゲット・システムで実行されないコードを検証しないようにするには、以下のいずれかの処置をとります。

- ビルド記述子オプション **eliminateSystemDependentCode** を *yes* に設定します。現行の例では、**add** ステートメントは、ビルド記述子オプションが、*yes* に設定されている場合には検証は行われません。ただし、生成プログラムがシステム依存コードを除去できるのは、論理式 (この場合は **sysVar.systemType IS AIX**) が単純で生成時に評価できる場合だけであることに注意してください。
- または、検証しない文を 2 番目のプログラムに移動し、元のプログラムが新規プログラムを以下のように条件付きで呼び出すようにします。

```
if (sysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

関連する概念

319 ページの『ビルド記述子パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

enableJavaWrapperGen

プログラムを生成するコマンドの発行時に、ビルド記述子オプション **enableJavaWrapperGen** によって、以下の 3 つの選択肢から選択できます。

YES (デフォルト)

プログラムは生成され、関連する Java ラッパー・クラス、および関連する EJB セッション Bean (該当する場合) の生成が可能になります。

ONLY

プログラムは生成されません。ただし、関連する Java ラッパー・クラス、および関連する EJB セッション Bean (該当する場合) の生成が可能になります。

NO

プログラムが生成されます。ただし Java ラッパー・クラスも、関連する EJB セッション Bean (該当する場合) も生成されません。

Java ラッパー・クラスおよび EJB セッション Bean の実際の生成では、生成時に使用されるリンケージ・オプション・パーツに、適切な設定が必要です。概要については『Java ラッパー』を参照してください。

関連する概念

327 ページの『Java ラッパー』

関連する参照項目

595 ページの『Java ラッパー・クラス』

genDataTables

ビルド記述子オプション **genDataTables** は、生成しているプログラム内で参照されるデータ・テーブルを生成するかどうかを示します。参照は、プログラムの使用宣言およびプログラム・プロパティ **msgTablePrefix** 内で行われます。

有効な値は *yes* (デフォルト) と *no* です。

以下の場合、値を *no* に設定します。

- プログラムで参照されるデータ・テーブルが以前に生成されている。また、
- これらのテーブルが生成後に変更されていない。

詳細については、『DataTable パーツ』を参照してください。

関連する概念

319 ページの『ビルド記述子パーツ』

155 ページの『DataTable』

関連する参照項目

415 ページの『ビルド記述子オプション』

783 ページの『EGL ソース形式のプログラム・パーツ』

1020 ページの『使用宣言』

genDirectory

ビルド記述子オプション **genDirectory** は、生成済み出力ファイルと準備状況ファイルが格納されるディレクトリーの完全修飾パスを指定します。

ワークベンチ内、またはワークベンチ・バッチ・インターフェースから生成を行う場合は、次の規則が適用されます。

Java 生成の場合

genProject または **genDirectory** を指定する必要がありますが、両方を指定するとエラーが発生します。また、iSeries 用のコードを生成する場合は、**genProject** を指定する必要があります。

EGL SDK から生成を行う場合は、次の規則が適用されます。

- **genDirectory** を指定する必要があります。
- **genProject** を指定すると、エラーが発生します。
- iSeries 用の Java コードは生成できません。

Java コードのデプロイの詳細については、『ディレクトリーへ生成される Java コードの処理』を参照してください。

関連する概念

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

361 ページの『ワークベンチ・バッチ・インターフェースからの生成』

359 ページの『ワークベンチでの生成』

関連するタスク

364 ページの『ディレクトリーへ生成される Java コードの処理』

関連する参照項目

415 ページの『ビルド記述子オプション』

425 ページの『genDirectory』

427 ページの『genProject』

genFormGroup

ビルド記述子オプション **genFormGroup** は、生成中のプログラムの使用宣言で参照される書式グループを生成するかどうかを示します。有効な値は *yes* (デフォルト) と *no* です。

ヘルプ書式グループ (存在する場合) は、このオプションの影響を受けません。ただし、ビルド記述子オプション **genHelpFormGroup** の影響を受けます。

関連する概念

319 ページの『ビルド記述子パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

『genHelpFormGroup』

1020 ページの『使用宣言』

genHelpFormGroup

ビルド記述子オプション **genHelpFormGroup** は、生成中のプログラムの使用宣言で参照されるヘルプ書式グループを生成するかどうかを示します。有効な値は *yes* (デフォルト) と *no* です。

メイン書式グループは、このオプションの影響を受けません。ただし、ビルド記述子オプション **genFormGroup** の影響を受けます。

関連する概念

319 ページの『ビルド記述子パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

426 ページの『genFormGroup』

1020 ページの『使用宣言』

genProject

ビルド記述子オプション **genProject** は、Java 生成出力をワークベンチ・プロジェクトに配置し、Java ランタイム設定で必要なタスクを自動化します。その設定、および **genProject** を使用する利点については、『プロジェクトへの Java コードの生成』を参照してください。

genProject を使用するには、プロジェクト名を指定します。これにより、ビルド記述子オプション **buildPlan**、**genDirectory**、および **prep** は無視され、次のいずれかの場合に、準備が実行されます。

- プロジェクトを再ビルドするとき。
- ソース・ファイルを生成するとき。(ただし、ワークベンチ設定「リソース変更時にビルドを自動的に実行」を確認した場合のみ。)

ワークベンチに存在しないプロジェクトの名前をオプション **genProject** に設定すると、EGL は、その名前を使用して Java プロジェクトを作成します。ただし、以下の例外があります。

- ページ・ハンドラーを生成しているときに、関連 JSP を含むプロジェクト以外のプロジェクトを指定した場合、その別プロジェクトが存在しないと、EGL は EGL Web プロジェクトを作成します。(ただし、関連 JSP を含むプロジェクト内にページ・ハンドラーを生成することをお勧めします。)
- 2 つ目の例外は EJB 処理に関連するものです。リンケージ・オプション・パーツの **callLink** エlement **type** プロパティが **ejbCall** であるときに Java ラッパーを生成すると、例外が発生します (ラッパーから EGL 生成のプログラムへの呼び出しの場合)。その場合、EGL は、**genProject** の値を使用して EJB プロジェクトを作成します。また、必要に応じて、EJB プロジェクト名に **EAR** を付けた名前の新規エンタープライズ・アプリケーション・プロジェクトを作成します。

プロジェクトの生成以外に、EGL は以下のことを実行します。

- EGL は、プロジェクト内にフォルダーを作成する。パッケージ構造は、トップレベル・フォルダー **JavaSource** の下から始まります。フォルダー名を右マウス・ボタンでクリックし、「**Refactor** (リファクタリング)」を選択すると、**JavaSource** の名前を変更することができます。
- Java の設定ページ (インストールされた JRE) に JRE 定義を指定した場合、EGL は、クラスパス変数 **JRE_LIB** を追加します。この変数には、現在使用している JRE の実行時 JAR ファイルへのパスが含まれます。

ワークベンチ内、またはワークベンチ・パッチ・インターフェースから生成を行う場合は、次の規則が適用されます。

Java 生成の場合

genProject または **genDirectory** のいずれかを指定する必要はありません。いずれも指定しない場合、Java 出力は、生成される EGL ソース・ファイルを含むプロジェクト内に生成されます。

ページ・ハンドラーを生成しているときに、指定したプロジェクトが存在する場合、そのプロジェクトは EGL Web プロジェクトでなければなりません。セッション EJB を生成しており、指定したプロジェクトが存在する場合、プロジェクトは EJB プロジェクトとなります。

COBOL 生成の場合

genDirectory を指定する必要があります。EGL では **genProject** の設定はすべて無視されます。

EGL SDK から生成を行う場合は、次の規則が適用されます。

- **genDirectory** を指定する必要があります。
- **genProject** を指定すると、エラーが発生します。
- iSeries 用の Java コードは生成できません。

関連する概念

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

361 ページの『ワークベンチ・バッチ・インターフェースからの生成』

359 ページの『ワークベンチでの生成』

351 ページの『プロジェクトへの Java コードの生成』

関連するタスク

関連する参照項目

415 ページの『ビルド記述子オプション』

419 ページの『buildPlan』

425 ページの『genDirectory』

432 ページの『prep』

460 ページの『callLink エLEMENT の type』

genProperties

ビルド記述子オプション **genProperties** は、生成される Java ランタイム・プロパティがある場合にはその種類を指定します。また、リンケージ・プロパティ・ファイルを生成するかどうかを指定する場合があります。このビルド記述子オプションが有効であるのは、どの種類の出力も使用可能な Java プログラムを生成する場合、またはリンケージ・プロパティ・ファイルのみ使用可能なラッパーを生成する場合に限ります。

有効な値は以下のとおりです。

NO (デフォルト)

ランタイム・プロパティまたはリンケージ・プロパティは EGL では生成されません。

PROGRAM

効果には以下のものがあります。

- J2EE の外部で実行するプログラムを生成する場合、生成されるプログラムに固有のプロパティー・ファイルが EGL により生成されます。ファイルの名前は以下のとおりです。

`pgmAlias.properties`

`pgmAlias`

実行時のプログラム名。

- **PROGRAM** または **GLOBAL** を指定することにより、その他の効果があります。
 - J2EE 内で実行されるプログラムを生成する場合、EGL により J2EE 環境ファイルまたはデプロイメント記述子が生成されます。詳細については、『デプロイメント記述子値の設定の代替について』を参照してください。
 - Java ラッパーまたは呼び出し側プログラムを作成する場合は、EGL によりリンケージプロパティー・ファイルが生成される場合があります。ファイルが生成される状態の詳細については、『リンケージ・プロパティー・ファイル (参照)』を参照してください。

GLOBAL

効果には以下のものがあります。

- J2EE の外部で実行するプログラムを生成する場合、実行単位を通して使用されるが実行単位の初期プログラムには指定されていないプロパティー・ファイルが EGL により生成されます。プロパティー・ファイルの名前は **rununit.properties** です。

このオプションは、実行単位の最初のプログラムがファイルまたはデータベースにアクセスせず、それを実行するプログラムを呼び出す場合に特に役立ちます。

呼び出し側プログラムを生成する場合、そのプログラム用に指定されるプロパティー・ファイルを作成したり、内容にデータベース関連のプロパティーを含まないようにすることができます。呼び出し先プログラムを生成する場合、**rununit.properties** を生成することができます。その内容は、両方のプログラムで使用可能です。

- **GLOBAL** または **PROGRAM** を指定することにより、その他の効果があります。
 - J2EE 内で実行されるプログラムを生成する場合、EGL により J2EE 環境ファイルまたはデプロイメント記述子が生成されます。詳細については、『デプロイメント記述子値の設定の代替について』を参照してください。
 - Java ラッパーまたは呼び出し側プログラムを作成する場合は、EGL によりリンケージプロパティー・ファイルが生成される場合があります。ファイルが生成される状態の詳細については、『リンケージ・プロパティー・ファイル (参照)』を参照してください。

詳細については、『Java ランタイム・プロパティー』、および『リンケージ・プロパティー・ファイル』を参照してください。

関連する概念

388 ページの『J2EE 環境ファイル』

377 ページの『Java ランタイム・プロパティー』

338 ページの『リンケージ・オプション・パーツ』
396 ページの『リンケージ・プロパティ・ファイル』

関連するタスク

415 ページの『ビルド記述子オプション』
386 ページの『デプロイメント記述子値の設定』

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

itemsNullable

ビルド記述子オプション **itemsNullable** は、コードがプリミティブ・フィールドを NULL に設定できる環境を指定します。

有効な値は以下のとおりです。

NO

以下の場合を除いて、プリミティブ・フィールドを NULL に設定できません。

- フィールドが SQL レコード内にあり、かつ
- SQL 項目プロパティ **isNullable** が yes に設定されている。

itemsNullable のこの設定がデフォルトであり、振る舞いについては、以前のバージョンの EGL と一貫性があります。

YES

固定レコード以外の任意のレコード内にある任意のプリミティブ・フィールドを NULL に設定します。この振る舞いは、Informix 製品 I4GL と一貫性があります。

次の表は、このオプションの決定による効果を示しています。

表 9. **itemsNullable** の効果

操作	ItemsNullable = NO	ItemsNullable = YES
NULL フィールドを別のフィールドに割り当てる	ソースの値は 0 またはブランクであり、この割り当てにより、値と (ターゲットが NULL 可能な場合) NULL 状態の両方がコピーされます。	ターゲットが NULL 可能な場合、このターゲットは NULL に設定されます。それ以外の場合、ターゲットは 0 またはブランクに設定されます。
数式で NULL フィールドを使用する	このフィールドは、0 が含まれているかのように扱われます。	この式は NULL に評価されます。
テキスト式で NULL フィールドを使用する	このフィールドは、スペースが含まれているかのように扱われます。	このフィールドは、空ストリングであるかのように扱われます。
論理式で NULL フィールドを使用する	この式は、フィールドの値が 0 またはブランクであるかのように扱われ、次の例は TRUE に評価されます。 0 == null	この式は、NULL が NULL と比較される場合のみ、TRUE に評価されます。次の例は、これと異なり、FALSE に評価されます。 0 == null

表 9. itemsNullable の効果 (続き)

操作	ItemsNullable = NO	ItemsNullable = YES
フィールドを空に設定する	NULL 状態は設定されません。	NULL 状態が設定されます。
レコードを空に設定する	NULL 状態は設定されません。	NULL 状態が設定されます。

関連する参照項目

415 ページの『ビルド記述子オプション』

J2EE

ビルド記述子オプション **J2EE** は、J2EE 環境で動作する Java プログラムを生成するかどうかを指定します。有効な値は以下のとおりです。

NO (デフォルト)

J2EE 環境では動作しないプログラムを生成します。プログラムは、データベースに直接接続します。また、その環境は、プロパティ・ファイルで定義されます。

YES

J2EE 環境で実行するプログラムを生成します。プログラムは、データ・ソースを使用してデータベースに接続します。また、その環境は、デプロイメント記述子で定義されます。

ページ・ハンドラーを生成すると、このオプションで指定する内容とは関係なく、J2EE は常に YES に設定されます。

関連する概念

303 ページの『EGL デバッガー』

関連する参照項目

415 ページの『ビルド記述子オプション』

linkage

ビルド記述子オプション **linkage** は、生成の各側面を処理するリンケージ・オプション・パーツ名を指定します。このオプションは、生成の際に必要なく、デフォルト値也没有ありません。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

442 ページの『callLink エlement』

nextBuildDescriptor

ビルド記述子オプション **nextBuildDescriptor** は、チェーン内に存在する次のビルド記述子を指定します。詳細については、『ビルド記述子パーツ』を参照してください。

関連する概念

319 ページの『ビルド記述子パーツ』

関連する参照項目

415 ページの『ビルド記述子オプション』

prep

ビルド記述子オプション **prep** は、戻りコード ≤ 4 で生成が完了したときに EGL で準備を開始するかどうかを指定します。有効な値は YES と NO です。デフォルトは、YES です。

prep を NO に設定しても、後でコードを準備できます。詳細については、『生成後のビルド計画の呼び出し』を参照してください。

これらケースを考慮してください。

- をディレクトリー内に生成する場合、EGL は、ビルド記述子オプション **genDirectory** に指定されたディレクトリーへの準備メッセージを結果ファイルに書き込みます。
- をプロジェクト内に生成する場合 (オプション **genProject**)、オプション **prep** は有効ではなく、準備は、次のいずれかの場合に実行されます。
 - プロジェクトを再ビルドするとき。
 - ソース・ファイルを生成するとき。(ただし、ワークベンチ設定「リソース変更時にビルドを自動的に実行」を確認した場合のみ。)

生成したビルド計画をカスタマイズしたい場合は、次のステップを実行します。

- オプション **prep** に NO を設定します。
- オプション **buildPlan** に YES (デフォルト) を設定します。
- 出力を生成します。
- ビルド計画をカスタマイズします。
- *buildPlan* の説明に従って、ビルド計画を呼び出します。

関連する概念

357 ページの『結果ファイル』

関連するタスク

363 ページの『生成後のビルド計画の呼び出し』

関連する参照項目

415 ページの『ビルド記述子オプション』

419 ページの『buildPlan』

573 ページの『生成される出力 (参照)』

425 ページの『genDirectory』

427 ページの『genProject』

resourceAssociations

ビルド記述子オプション **resourceAssociations** は、ターゲット・プラットフォーム上のファイル/キューとレコード・パーツとを関連付けるリソース関連パーツ名を指定します。このオプションは、生成の際に必要なく、デフォルト値也没有ありません。

関連する概念

331 ページの『リソース関連とファイル・タイプ』

関連するタスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

関連する参照項目

415 ページの『ビルド記述子オプション』

407 ページの『関連エレメント』

793 ページの『レコードとファイル・タイプの相互参照』

sessionBeanID

ビルド記述子オプション **sessionBeanID** は、J2EE デプロイメント記述子内の既存のセッション要素の名前を指定します。次のことを行くと、環境エントリーがセッション要素に格納されます。

- Java プラットフォーム用のプログラムを生成する (**system** に「AIX (AIX)」、「WIN (WIN)」または「USS (USS)」を設定して)
- EJB プロジェクト内に生成する (**genProject** に「EJB プロジェクト (EJB Project)」を設定して)
- 環境プロパティの生成を要求する (**genProperties** に GLOBAL または PROGRAM を設定して)

オプション **sessionBeanID** は、次の場合に役に立ちます。

1. EJB セッション Bean とともに Java ラッパーを生成します。EJB プロジェクト・デプロイメント記述子 (ファイル ejb-jar.xml) 内で、EGL は、環境エントリーを使用せずにセッション要素を作成します。

EJB セッション Bean とセッション要素は、どちらも次のように命名されます。

ProgramnameEJBBean

Programname は、EJB セッション Bean 経由でデータを受け取るランタイム・プログラムの名前です。名前の先頭の文字は大文字で表し、残りの文字は小文字で表します。

この例では、プログラムの名前は ProgramA、セッション要素と EJB セッション Bean の名前は ProgramaEJBBean です。

2. EJB セッション Bean を生成した後は、Java プログラム自体を生成します。ビルド記述子オプション **genProperties** が YES に設定されているため、EGL は、J2EE 環境エントリーをデプロイメント記述子内、つまりステップ 1 で設定されたセッション要素内に生成します。
3. ProgramB を生成します。ProgramB は、ProgramA の helper クラスとして使用する Java プログラムです。**system** と **genProject** の値は、ステップ 2 で使用する値と同じです。また、環境エントリーを生成し、**sessionBeanID** にセッション要素の名前を設定します。

sessionBeanID を使用すると、EGL により、ステップ 2 で作成されたセッション要素 (ProgramaEJBBean) に 2 番目のプログラムの環境エントリーが格納されます。

ステップ 2 の ProgramA の生成時、EGL は、後に続くデプロイメント記述子の位置に環境エンタリー **vgj.nls.code** および **vgj.nls.number.decimal** を作成しています。ただし、エンタリー **vgj.jdbc.default.database** は、ProgramB でのみ使用され、ステップ 3 で作成されています。

```
<ejb-jar id="ejb-jar_ID">
  <display-name>EJBTest</display-name>
  <enterprise-beans>
    <session id="ProgramaEJBBean">
      <ejb-name>ProgramaEJBBean</ejb-name>
      <home>test.ProgramaEJBHome</home>
      <remote>test.ProgramaEJB</remote>
      <ejb-class>test.ProgramaEJBBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>vgj.nls.code</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>ENU</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.nls.number.decimal</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>.</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>vgj.jdbc.default.database</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>jdbc/Sample</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
</ejb-jar>
```

環境エンタリーを追加するには、セッション要素をデプロイメント記述子に格納しておく必要があります。セッション要素は Java ラッパーの生成時に作成されます。そのため、Java ラッパーは、関連するプログラムを生成する前に作成することをお勧めします。

次の場合は、プログラムを EJB プロジェクト内に生成します。ただし、環境エンタリーは、デプロイメント記述子ではなく、J2EE 環境ファイルに格納されます。

- **sessionBeanID** が設定されているが、**sessionBeanID** の値と同じセッション要素がデプロイメント記述子内に存在しない、または
- **sessionBeanID** が設定されておらず、プログラムに指定されたセッション要素がデプロイメント記述子内に存在しない。これは、プログラムがラッパーより先に生成された場合に起こります。

EJB プロジェクトの場合、環境エンタリー名 (**vgj.nls.code** など) は、各セッション要素につき 1 つだけしか指定できません。環境エンタリーがすでに存在する場合、EGL は、新しいエンタリーを作成するのではなく、エンタリーの型と値を更新します。

EGL では、デプロイメント記述子から環境エンタリーは削除されません。

sessionBeanID にデフォルト値はありません。

関連する参照項目

415 ページの『ビルド記述子オプション』

sqlCommitControl

ビルド記述子オプション **sqlCommitControl** を使用することにより、デフォルトのデータベースに変更が加えられるたびにコミットするかどうかを指定する Java ランタイム・プロパティの生成が可能になります。

このプロパティ (`vgj.jdbc.default.database.autoCommit`) が生成されるのは、ビルド記述子オプション **genProperties** も PROGRAM または GLOBAL に設定される場合のみです。Java ランタイム・プロパティは、生成時のユーザーの決定にかかわらず、デプロイメント時に設定できます。

sqlCommitControl の有効な値は、以下のとおりです。

NOAUTOCOMMIT

コミットは自動的に行われません。振る舞いは以前のバージョンの EGL と一貫性があります。Java ランタイム・プロパティは、デフォルトで偽に設定されます。

この場合のコミットおよびロールバックの規則については、『作業論理単位』を参照してください。

AUTOCOMMIT

コミットは自動的に行われます。振る舞いは以前のバージョンの Informix 製品 I4GL と一貫性があります。Java ランタイム・プロパティは、真に設定されます。

関連する概念

415 ページの『ビルド記述子オプション』

377 ページの『Java ランタイム・プロパティ』

関連する参照項目

415 ページの『ビルド記述子オプション』

270 ページの『デフォルト・データベース』

428 ページの『genProperties』

sqlDB

ビルド記述子オプション **sqlDB** は、J2EE の外部で実行される生成された Java プログラムで使用するデフォルト・データベースを指定します。値は接続 URL です (例: `jdbc:db2:MyDB`)。

オプション **sqlDB** には、大/小文字の区別があり、デフォルト値はありません。このオプションは、J2EE 以外の Java プログラムを生成する場合にのみ使用します。このオプションは、Java ランタイム・プロパティ **vgj.jdbc.default.database** に値を割り当てます。ただし、オプション **genProperties** が GLOBAL または PROGRAM に設定されている場合に限りです。

検証のために使用するデータベースを指定するには、**sqlValidationConnectionURL** を設定します。

関連する概念

377 ページの『Java ランタイム・プロパティ』

247 ページの『SQL サポート』

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 428 ページの『genProperties』
- 584 ページの『Java ランタイム・プロパティ (詳細)』
- 438 ページの『sqlPassword』
- 438 ページの『sqlValidationConnectionURL』
- 『sqlJDBCClass』
- 441 ページの『validateSQLStatements』

sqlID

ビルド記述子オプション **sqlID** は、ユーザー ID を指定します。このユーザー ID は、生成時に SQL ステートメントを検証する際に、データベースと接続するために使用されます。データベースを指定するには、**sqlValidationConnectionURL** を設定します。

Java プログラムを生成する場合、EGL は、**sqlID** の値も Java ランタイム・プロパティ **vgj.jdbc.default.userid** に割り当てます。このプロパティは、実行時にデフォルト・データベースに接続するためのユーザー ID を識別します。デフォルト・データベースは、**sqlDB** で指定します。

オプション **sqlID** には、大文字小文字の区別があり、デフォルト値はありません。

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 584 ページの『Java ランタイム・プロパティ (詳細)』
- 435 ページの『sqlDB』
- 438 ページの『sqlPassword』
- 438 ページの『sqlValidationConnectionURL』
- 『sqlJDBCClass』
- 441 ページの『validateSQLStatements』

sqlJDBCClass

ビルド記述子オプション **sqlJDBCClass** は、ドライバー・クラスを指定します。このドライバー・クラスは、生成時に SQL ステートメントを検証する際に、EGL が使用するデータベースと接続するために使用されます。データベースを指定するには、**sqlValidationConnectionURL** を設定します。データベースには、JDBC を介してアクセスします。

以下の場合には、EGL は、プロパティ・ファイル内で **sqlJDBCClass** の値も Java ランタイム・プロパティ **vgj.jdbc.drivers** に割り当てます。

- **genProperties** が GLOBAL または PROGRAM に設定されている。
- **J2EE** が NO に設定されている。

このドライバー・クラスにはデフォルトがなく、ドライバーごとに形式が異なります。

- Windows 用の IBM DB2 APP DRIVER の場合、ドライバー・クラスは次のようになります。

`COM.ibm.db2.jdbc.app.DB2Driver`

- Windows 用の IBM DB2 NET DRIVER の場合、ドライバー・クラスは次のようになります。

```
COM.ibm.db2.jdbc.net.DB2Driver
```

- IBM DB2 UNIVERAL DRIVER for Windows の場合には、ドライバー・クラスは以下のとおりです (小文字で com を付けます)。

```
com.ibm.db2.jcc.DB2Driver
```

- Oracle JDBC シン・クライアント・サイド・ドライバーの場合、ドライバー・クラスは次のようになります。

```
oracle.jdbc.driver.OracleDriver
```

- IBM Informix JDBC ドライバーの場合、ドライバー・クラスは次のようになります。

```
com.informix.jdbc.IfxDriver
```

その他のドライバー・クラスについては、各ドライバーの資料を参照してください。

ドライバー・クラスを複数指定する場合は、クラス名とクラス名の間をコロン (:) で区切ります。これを行うのは、ある Java プログラムが別の Java プログラムへのローカル呼び出しを行うにもかかわらず、異なるデータベース管理システムにアクセスする場合です。

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 271 ページの『Informix および EGL』
- 435 ページの『sqlDB』
- 436 ページの『sqlID』
- 438 ページの『sqlPassword』
- 438 ページの『sqlValidationConnectionURL』
- 441 ページの『validateSQLStatements』

sqlJNDIName

ビルド記述子オプション **sqlJNDIName** は、J2EE で稼働する生成された Java プログラムで使用するデフォルト・データベースを指定します。値は、デフォルトのデータ・ソースが JNDI レジストリーで結合される名前です。例えば、jdbc/MyDB です。

オプション **sqlJNDIName** には、大/小文字の区別があり、デフォルト値はありません。このオプションは、J2EE 用の Java プログラムを生成する場合にのみ使用します。このオプションは、Java ランタイム・プロパティー **vgj.jdbc.default.database** に値を割り当てます。ただし、オプション **genProperties** が GLOBAL または PROGRAM に設定されている場合に限りです。

検証のために使用するデータベースを指定するには、**sqlValidationConnectionURL** を設定します。

関連する概念

- 377 ページの『Java ランタイム・プロパティー』
- 247 ページの『SQL サポート』

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 428 ページの『genProperties』
- 584 ページの『Java ランタイム・プロパティ (詳細)』
 - 『sqlPassword』
 - 『sqlValidationConnectionURL』
- 436 ページの『sqlJDBCClass』
- 441 ページの『validateSQLStatements』

sqlPassword

ビルド記述子オプション **sqlPassword** は、パスワードを指定します。このパスワードは、生成時に SQL ステートメントを検証する際に、データベースと接続するために使用されます。データベースを指定するには、**sqlValidationConnectionURL** を設定します。

Java プログラムを生成する場合、EGL は、**sqlPassword** の値も Java ランタイム・プロパティ **vgj.jdbc.default.password** に割り当てます。このプロパティは、実行時にデフォルト・データベースに接続するためのパスワードを識別します。デフォルト・データベースは、**sqlDB** で指定します。

オプション **sqlPassword** には、大文字小文字の区別があり、デフォルト値はありません。

関連する概念

- 377 ページの『Java ランタイム・プロパティ』

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 584 ページの『Java ランタイム・プロパティ (詳細)』
- 435 ページの『sqlDB』
- 436 ページの『sqlID』
 - 『sqlValidationConnectionURL』
- 436 ページの『sqlJDBCClass』
- 441 ページの『validateSQLStatements』

sqlValidationConnectionURL

ビルド記述子オプション **sqlValidationConnectionURL** は、URL を指定します。この URL は、生成時に SQL ステートメントを検証する際に、EGL が使用するデータベースと接続するために使用されます。データベースには、JDBC を介してアクセスします。

この URL にはデフォルトがなく、ドライバーごとに形式が異なります。

- Windows 用の IBM DB2 APP DRIVER の場合、URL は次のようになります。

```
jdbc:db2:dbName
```

dbName

データベース名

- Oracle JDBC シン・クライアント・サイド・ドライバーの場合、URL は、データベース・ロケーションごとに異なります。データベースがローカル・マシンにある場合、URL は次のようになります。

`jdbc:oracle:thin:dbName`

データベースがリモート・サーバーにある場合、URL は次のようになります。

`jdbc:oracle:thin:@host:port:dbName`

host

データベース・サーバーのホスト名

port

ポート番号

dbName

データベース名

- IBM Informix JDBC ドライバーの場合、URL は、次のようになります。(実際の URL は 1 行で指定します)

`jdbc:informix-sqli://host:port
/dbName:informixserver=servername;
user=userName;password=password`

host

データベース・サーバーが配置されているマシンの名前。

port

ポート番号

dbName

データベース名

serverName

データベース・サーバーの名前

userName

Informix ユーザー ID

password

ユーザー ID に関連したパスワード

- その他のドライバーについては、各ドライバーの資料を参照してください。

関連する参照項目

415 ページの『ビルド記述子オプション』

271 ページの『Informix および EGL』

435 ページの『sqlDB』

436 ページの『sqlID』

438 ページの『sqlPassword』

436 ページの『sqlJDBCdriverClass』

441 ページの『validateSQLStatements』

system

ビルド記述子オプション **system** は、生成のターゲット・プラットフォームを指定します。このオプションは、必ず指定する必要があります。デフォルト値はありません。有効な値は以下のとおりです。

AIX

生成で、AIX 上で動作可能な Java プログラムを作成することを指定します。

ISERIESJ

生成で、iSeries 上で動作可能な a Java プログラムを作成することを指定します。

LINUX

生成で、Linux (Intel プロセッサ) 上で実行可能な Java プログラムを作成することを指定します。

USS

生成で、z/OS UNIX システム・サービス上で動作可能な Java プログラムを作成することを指定します。

WIN

生成で、Windows 2000/NT/XP 上で動作可能な Java プログラムを作成することを指定します。

関連する概念

572 ページの『生成される出力』

338 ページの『リンケージ・オプション・パーツ』

10 ページの『ランタイム構成』

関連する参照項目

415 ページの『ビルド記述子オプション』

442 ページの『callLink エlement』

573 ページの『生成される出力 (参照)』

271 ページの『Informix および EGL』

targetNLS

ビルド記述子オプション **targetNLS** は、ランタイムのメッセージを識別するために使用する各国語コードを指定します。

次の表は、サポートされる言語をまとめたものです。指定した言語のコード・ページは、ターゲット・プラットフォーム上にロードする必要があります。

コード	言語
CHS	中国語 (簡体字)
CHT	中国語 (繁体字)
DES	スイス・ドイツ語
DEU	ドイツ語
ENP	大文字の英語 (Windows 2000、Windows NT、および z/OS UNIX システム・サービスではサポートされません)
ENU	米国英語
ESP	スペイン語
FRA	フランス語
ITA	イタリア語
JPN	日本語
KOR	韓国語
PTB	ブラジル・ポルトガル語

EGL により、開発マシンの Java ロケールが、サポート言語のいずれかに関連しているかどうかが判別されます。関連していると判断された場合は、**targetNLS** のデフォルト値がサポート言語になります。それ以外の場合は、**targetNLS** にはデフォルト値はありません。

関連する参照項目

415 ページの『ビルド記述子オプション』

VAGCompatibility

ビルド記述子オプション **VAGCompatibility** は、生成プロセスによって、特別なプログラム構文が許可されているかどうかを示します。『*VisualAge Generator との互換性*』を参照してください。有効な値は、*no* と *yes* です。

EGL の **VAGCompatibility** 設定により、ビルド記述子のデフォルト値が決定されます。EGL SDK で生成しており設定が使用可能ではない場合は、**VAGCompatibility** のデフォルト値は *no* です。

ユーザー・プログラムまたはページ・ハンドラーが特別な構文を使用している場合にのみ、*yes* を指定します。

関連する概念

319 ページの『ビルド記述子パーツ』

477 ページの『*VisualAge Generator との互換性*』

関連する参照項目

415 ページの『ビルド記述子オプション』

validateSQLStatements

ビルド記述子オプション **validateSQLStatements** は、SQL ステートメントをデータベースと突き合わせて検証するかどうかを指定します。**validateSQLStatements** を正しく使用するには、オプション **sqlValidationConnectionURL** を指定し、通常は、**sql** で始まるその他のオプション (後述) も指定する必要があります。

有効な値は YES と NO です。デフォルトは NO です。SQL ステートメントを検証すると、コードの生成に要する時間が増えます。

SQL の検証を要求すると、生成プラットフォームからアクセスされるデータベース・マネージャーは、SQL ステートメントを動的に準備します。

SQL ステートメントの検証には、次の制限事項があります。

- 動的 SQL を使用し、SQL レコードに基づく SQL ステートメントについては、検証を実行できません。
- 検証プロセスでは、生成環境内のデータベース・マネージャーによっては検出されるが、ターゲット・プラットフォーム上のデータベース・マネージャーによっては検出されないエラーが示されることがあります。
- 検証が実行されるのは、JDBC ドライバーで SQL prepare ステートメントの検証がサポートされている場合、および (場合によっては)、そのような検証を実行するようにドライバーが構成されている場合のみです。詳細については、JDBC ドライバーの資料を参照してください。

関連する参照項目

- 415 ページの『ビルド記述子オプション』
- 436 ページの『sqlIID』
- 438 ページの『sqlPassword』
- 438 ページの『sqlValidationConnectionURL』
- 436 ページの『sqlJDBCdriverClass』

ビルド・スクリプト

EGL ビルド・スクリプトに必要なオプション

EGL ビルド・スクリプトでは、DB2 UDBを使用している場合に、特定の準備オプションが必要になります。

DB2 プリコンパイラーに必要なオプション

以下のオプションは、DB2 を使用するために必要です。これらは、fdaptcl ビルド・スクリプトに組み込まれています。

- HOST(COB2)
- APOSTSQL
- QUOTE

callLink エlement

リンケージ・オプション・パーツの callLink Elementでは、呼び出しに使用されるリンケージの型を指定します。各Elementには以下のようなプロパティがあります。

- pgmName
- type

使用可能な追加のプロパティは、**type** プロパティの値によって決まります。これについては、以下のセクションで説明します。

- 『callLink type が localCall の場合 (デフォルト)』
- 443 ページの『callLink type が remoteCall の場合』
- 443 ページの『callLink type が ejbCall の場合』

callLink type が localCall の場合 (デフォルト)

同じスレッド内に存在する生成済みの Java プログラムを呼び出す Java プログラムを生成している場合は、プロパティ **type** を localCall に設定します。この場合、EGL ミドルウェアは使用されておらず、呼び出し先プログラムが **pgmName** で識別される callLink Elementに対して、以下のプロパティが有効です。

- 444 ページの『callLink Elementの alias』
- 451 ページの『callLink Elementの package』
- 453 ページの『callLink Elementの pgmName』
- 460 ページの『callLink Elementの type』

呼び出し先プログラムが呼び出し側と同じパッケージにあり、以下のいずれかの条件が有効の場合は、呼び出しで **callLink** エlementを指定する必要はありません。

- 呼び出し先プログラムの外部名を指定しない。
- 呼び出し先プログラムの外部名が、そのプログラムのパーツ名と同じである。

Java ラッパーを生成するときは、**type** の値に **localCall** を指定できません。

callLink type が remoteCall の場合

Java プログラムまたはラッパーを生成しており、別のスレッドで実行されるプログラムを Java コードで呼び出す場合は、プロパティ **type** を **remoteCall** に設定してください。呼び出しは、生成済みの EJB セッション Bean 経由ではありません。この場合、EGL ミドルウェアは使用されており、呼び出し先プログラムが **pgmName** で識別される **callLink** Elementに対して、以下のプロパティが有効です。

- 444 ページの『callLink Elementの alias』
- 445 ページの『callLink Elementの conversionTable』
- 449 ページの『callLink Elementの location』
- 451 ページの『callLink Elementの package』 (別のパッケージに格納されている Java プログラムを生成済みコードが呼び出す場合にのみ使用される)
- 453 ページの『callLink Elementの pgmName』
- 455 ページの『callLink Elementの remoteBind』
- 456 ページの『callLink Elementの remoteComType』
- 458 ページの『callLink Elementの remotePgmType』
- 459 ページの『callLink Elementの serverID』
- 460 ページの『callLink Elementの type』

callLink type が ejbCall の場合

以下の状況に対処するために **callLink** Elementが必要な場合は、プロパティ **type** に **ejbCall** を設定します。

- Java ラッパーを作成し、関連する生成済みプログラムを生成済み EJB セッション Bean 経由で呼び出そうとしている場合
- Java プログラムを作成し、別の生成済みプログラムを生成済み EJB セッション Bean 経由で呼び出そうとしている場合

この場合、EGL ミドルウェアは使用されており、呼び出し先プログラムが **pgmName** で識別される **callLink** Elementに対して、以下のプロパティが有効です。

- 444 ページの『callLink Elementの alias』
- 445 ページの『callLink Elementの conversionTable』
- 449 ページの『callLink Elementの location』
- 451 ページの『callLink Elementの package』 (EJB セッション Bean が存在しないパッケージに格納されている Java プログラムを生成済み Java コードが呼び出す場合にのみ使用される)

- 452 ページの『callLink エLEMENTの parmForm』 (CICS 上で実行されるプログラムを生成済み Java コードが呼び出す場合にのみ使用される)
- 453 ページの『callLink エLEMENTの pgmName』
- 453 ページの『callLink エLEMENTの providerURL』
- 455 ページの『callLink エLEMENTの remoteBind』
- 456 ページの『callLink エLEMENTの remoteComType』
- 458 ページの『callLink エLEMENTの remotePgmType』
- 459 ページの『callLink エLEMENTの serverID』
- 460 ページの『callLink エLEMENTの type』

関連する概念

338 ページの『リンケージ・オプション・パーツ』
10 ページの『ランタイム構成』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

『callLink エLEMENTの alias』
445 ページの『callLink エLEMENTの conversionTable』

448 ページの『callLink エLEMENTの linkType』
449 ページの『callLink エLEMENTの location』

451 ページの『callLink エLEMENTの package』

453 ページの『callLink エLEMENTの pgmName』
453 ページの『callLink エLEMENTの providerURL』
455 ページの『callLink エLEMENTの remoteBind』
456 ページの『callLink エLEMENTの remoteComType』
458 ページの『callLink エLEMENTの remotePgmType』
459 ページの『callLink エLEMENTの serverID』
460 ページの『callLink エLEMENTの type』

callLink エLEMENTの alias

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **alias** は、プロパティ **pgmName** で識別されるプログラムの実行時の名前を指定します。このプロパティは、生成するプログラムによって呼び出されるプログラムが **pgmName** で参照される場合にのみ有効です。

このプロパティの値は、プログラムの宣言時に指定した別名があれば、その別名と同じにしなければなりません。プログラム宣言時に別名を指定していない場合は、callLink エLEMENTのプロパティ **alias** にプログラム・パーツ名を設定するか、または何も設定しないようにします。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』

453 ページの『callLink エLEMENTの pgmName』

callLink エLEMENTの conversionTable

リンケージ・オプション・パーツの callLink エLEMENTのプロパティー

conversionTable は、呼び出し時のデータ変換で使用される変換テーブルの名前を指定します。このプロパティーは、生成されたプログラム/ラッパーによって呼び出されるプログラムが **pgmName** で識別されるときのみ有効です。

次の制限が適用されます。

- 非 Java プログラムの呼び出しでは、呼び出し側プログラムで使用されている文字セット (ASCII または EBCDIC) に従って、デフォルト変換が実行されます。次の場合は、**conversionTable** に値を指定する必要があります。
 - 呼び出し側が Java コードであり、文字セット (EBCDIC または ASCII) をサポートするマシン上で稼働している。
 - 呼び出し先プログラムが非 Java であり、それ以外の文字セットをサポートするマシン上で稼働している。
- EGL 生成の Java コードで Java プログラムを呼び出す場合は、双方向テキストの場合を除き、変換テーブルの指定は無効となります。
- プロパティー **conversionTable** を使用できるのは、プロパティー **type** の値が **ejbCall** または **remoteCall** である場合のみです。

次の値から 1 つを選択します。

conversion table name

呼び出し側は指定された変換テーブルを使用します。テーブル・リストについては、『データ変換』を参照してください。

- * デフォルトの変換テーブルを使用します。選択されたテーブルはクライアント・マシンのロケールまたは (クライアントが Web アプリケーション・サーバー上で稼働している場合) そのサーバーのロケールに基づきます。認識できないロケールが見つかった場合は、英語を前提とします。

テーブル・リストについては、『データ変換』を参照してください。

programControlled

呼び出し側は、実行時にシステム項目 **sysVar.callConversionTable** で指定された変換テーブル名を使用します。 **sysVar.callConversionTable** がブランクの場合、型変換は行われません。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

- 510 ページの『双方向言語テキスト』
- 442 ページの『callLink エlement』
- 507 ページの『データ変換』
- 453 ページの『callLink エlementの pgmName』
- 960 ページの『convert()』
- 440 ページの『targetNLS』
- 460 ページの『callLink エlementの type』

callLink エlementの ctgKeyStore

リンケージ・オプション・パーツの callLink エlementの **ctgKeyStore** プロパティは、Java ツール keytool.exe、または CICS Transaction Gateway ツール IKEYMANを使用して生成された鍵ストアの名前です。このプロパティは、**remoteComType** プロパティの値が CICSSSL に設定されている場合は必要です

関連する概念

- 338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

- 442 ページの『callLink エlement』
- 『callLink エlementの ctgKeyStorePassword』
- 456 ページの『callLink エlementの remoteComType』

callLink エlementの ctgKeyStorePassword

リンケージ・オプション・パーツの callLink エlementの **ctgKeyStorePassword** プロパティは、鍵ストアの生成時に使用されるパスワードです。

関連する概念

- 338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

- 442 ページの『callLink エlement』
- 『callLink エlementの ctgKeyStore』
- 456 ページの『callLink エlementの remoteComType』

callLink エlementの ctgLocation

リンケージ・オプション・パーツの callLink エlementの **ctgLocation** プロパティは、CICS Transaction Gateway (CTG) サーバーにアクセスするための URL です。**remoteComType** プロパティの値が CICSECI または CICSSSL の場合に使用します。**ctgPort** プロパティを設定することによって関連するポートを指定します。

関連する概念

- 338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

442 ページの『callLink エlement』

456 ページの『callLink エlementの remoteComType』

callLink エlementの ctgPort

リンケージ・オプション・パーツの callLink エlementの **ctgPort** プロパティは、CICS Transaction Gateway (CTG) サーバーにアクセスするためのポートです。**remoteComType** プロパティの値が CICSECI または CICSSSL の場合に使用されます。**ctgLocation** プロパティを設定することによって、関連する URL を指定します。

CICSSSL の場合には、**ctgPort** の値は CTG JSSE リスナーが要求を listen 中の TCP/IP ポートです。**ctgPort** が指定されていない場合には、CTG デフォルト・ポート 8050 が使用されます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

442 ページの『callLink エlement』

446 ページの『callLink エlementの ctgLocation』

456 ページの『callLink エlementの remoteComType』

callLink エlementの JavaWrapper

リンケージ・オプション・パーツの callLink エlementの **javaWrapper** プロパティは、生成中のプログラムを起動できる Java ラッパー・クラスを生成できるかどうかを指定します。

有効な値は以下のとおりです。

No (デフォルト)

Java ラッパー・クラスの生成を許可しません。

Yes

生成を許可します。ビルド記述子オプション **enableJavaWrapperGen** が **yes** または **only** に設定されている場合のみ生成されます。

javaWrapper プロパティのユーザー選択は、リモート呼び出しの設定を行う場合のみ有効です。これは、**callLink** プロパティの **type** の値が **remoteCall** の場合のみ発生するためです。一方、EJB によってプログラムの呼び出しを設定している場合には、**javaWrapper** の値は必ず **yes** です。ローカル呼び出しを設定している場合には、**javaWrapper** の値は必ず **no** です。

ワークベンチで生成している場合、またはワークベンチ・バッチ・インターフェースから生成している場合には、ビルド記述子オプション **genProject** は、そのクラスを受信するプロジェクトを識別します。**genProject** が指定されていない場合 (または、EGL SDK で生成している場合には)、ラッパー・クラスは、ビルド記述子オプション **genDirectory** によって指定されたディレクトリに配置されます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

442 ページの『callLink エlement』

425 ページの『genDirectory』

427 ページの『genProject』

callLink エlementの linkType

リンケージ・オプション・パーツの callLink エlementのプロパティ **linkType** は、プロパティ **type** の値が localCall であるときのリンケージのタイプを指定します。

Java プログラムを生成する場合、**linkType** が有効となるのは、生成するプログラムによって呼び出されるプログラムがプロパティ **pgmName** で参照されるときです。Java ラッパーを生成する場合、プロパティ **type** は、remoteCall か ejbCall にする必要があります。**linkType** は使用できません。

次のリストから値を選択してください。

DYNAMIC

呼び出しが同じスレッド内の Java プログラムに対するものであることを指定します。DYNAMIC がデフォルト値となります。

STATIC

STATIC は DYNAMIC と等価です。ただし、Rational Application Developer for iSeries または Rational Application Developer for z/OS を使用している場合は除きます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エlementの編集』

関連する参照項目

442 ページの『callLink エlement』

453 ページの『callLink エlementの pgmName』

460 ページの『callLink エlementの type』

callLink エlementの library

リンケージ・オプション・パーツの callLink エlementの **library** プロパティは、**type** プロパティの値が ejbCall または remoteCall の場合には、呼び出し先プログラムが含まれている DLL またはライブラリーを指定します。

- EGL 生成の Java プログラムが iSeries 上の非 EGL 生成のリモート・プログラム (C または C++ サービス・プログラムなど) を呼び出す場合、呼び出し先プログラムは iSeries ライブラリーに属しており、**library** プロパティは呼び出し先のエントリー・ポイントを含むプログラムの名前を参照します。その他の callLink プロパティを以下のように設定します。

- **pgmName** プロパティをエントリー・ポイントの名前に設定する。
- **remoteComType** プロパティを **direct** または **distinct** に設定する。
- **remotePgmType** プロパティを **externallyDefined** に設定する。
- **location** プロパティを **iSeries** ライブラリーの名前に設定する。
- この他の場合、呼び出し側プログラムが **iSeries** 上にない EGL 生成の Java プログラムであるときは、**library** プロパティはローカル側でネイティブ・プログラムとして呼び出されるエントリー・ポイントを含む DLL の名前を参照します。エントリー・ポイントは、**pgmName** プロパティで示されます。ただし、エントリー・ポイントの名前と DLL が異なる場合のみ **library** プロパティを指定する必要があります。

ネイティブ DLL を呼び出すには、以下のように他の **callLink** プロパティを設定します。

- **remoteComType** プロパティを **direct** に設定する。
- **remotePgmType** プロパティを **externallyDefined** に設定する。
- **type** プロパティを **remoteCall** に設定する。これは、Java プログラムが実行中のマシンで DLL が呼び出されているにも関わらず、EGL ミドルウェアが使用されているためです。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

442 ページの『**callLink** エlement』

callLink エlementの location

リンケージ・オプション・パーツの **callLink** Elementのプロパティ **location** は、呼び出し先プログラムのロケーションを実行時に決定する方法を指定します。プロパティ **location** は、以下が揃った状況で適用できます。

- プロパティ **type** の値が **ejbCall** または **remoteCall** である。
- プロパティ **remoteComType** の値が **JAVA400**、**CICSECI**、**CICSSSL**、**CICSJ2C**、または **TCPIP** の場合。および、
- 次のいずれかが該当する場合。
 - Java プログラムを生成する場合、生成するプログラムによって呼び出されるプログラムがプロパティ **pgmName** で参照される
 - Java ラッパーを生成する場合、その Java ラッパー経由で呼び出されるプログラムが **pgmName** で参照される

次のリストから値を選択してください。

programControlled

呼び出し先プログラムのロケーションは、呼び出しの実行時にシステム関数 **sysVar.remoteSystemID** から取得することを指定します。

system name

呼び出し先プログラムが存在するロケーションを指定します。

Java プログラムまたはラッパーを生成する場合、このプロパティの意味は、プロパティ **remoteComType** の指定によって異なります。

- **remoteComType** の値が JAVA400 の場合、**location** は iSeries システム ID を参照します。
- **remoteComType** の値が CICSECI または CICSSSL の場合、**location** は CICS システム ID を参照します。
- **remoteComType** の値が CICSJ2C の場合、**location** は、呼び出しによって起動される CICS トランザクション用に設定する ConnectionFactory オブジェクトの JNDI 名を示します。J2EE サーバーを設定するときは、ConnectionFactory オブジェクトを設定します。詳細については、『*CICSJ2C 呼び出し用 J2EE サーバーの設定*』を参照してください。規則により、ConnectionFactory オブジェクト名は、次の例のように eis/ で開始します。
eis/CICS1
- **remoteComType** の値が TCPIP の場合、**location** は TCP/IP ホスト名を示し、デフォルト値は存在しません。
- 次のすべての条件に該当する場合、**location** は呼び出し先プログラムのライブラリーを参照します。
 - 呼び出し先プログラムが EGL で生成された Java プログラムであり、iSeries 上でローカルに実行される
 - **remoteComType** の値が DIRECT または DISTINCT である
 - **remotePgmType** の値が EXTERNALLYDEFINED である

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

389 ページの『CICSJ2C 呼び出し用 J2EE サーバーのセットアップ』

関連する参照項目

442 ページの『callLink エLEMENT』

453 ページの『callLink エLEMENTの pgmName』

456 ページの『callLink エLEMENTの remoteComType』

460 ページの『callLink エLEMENTの type』

callLink エLEMENTの luwControl

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **luwControl** は、呼び出し側と呼び出し先プログラムのどちらで作業単位を制御するかを指定します。このプロパティは、次の状況でのみ適用されます。

- プロパティ **type** の値が remoteCall である。
- Java プログラムまたはラッパーを生成する。
 - Java プログラムを生成する場合、生成するプログラムによって呼び出される CICS ベースのプログラムがプロパティ **pgmName** で参照される。
 - Java ラッパーを生成する場合、その Java ラッパー経由で呼び出される CICS ベースのプログラムが **pgmName** で参照される。

次の値から 1 つを選択します。

CLIENT

作業単位が呼び出し側の制御下にあることを指定します。呼び出し先プログラムによる更新は、呼び出し側がコミットまたはロールバックを要求するまでコミットまたはロールバックされません。呼び出し先プログラムがコミットまたはロールバックを発行すると、ランタイム・エラーが発生します。

呼び出し側の制御下にある作業単位が呼び出し先プログラムのあるプラットフォームでサポートされない場合、CLIENT がデフォルト値となります。

呼び出し元が、IBM Toolbox for Java を介して iSeries ベースの COBOL プログラムと通信する Java ラッパーまたはプログラムの場合、CLIENT を使用できます。このケースでは、呼び出しの **remoteComType** の値は JAVA400 です。

SERVER

呼び出し先プログラムによって開始された作業単位は、呼び出し側プログラムによって制御されたすべての作業単位から独立していることを指定します。呼び出し先プログラムでは、次の規則が適用されます。

- 回復可能リソースに対する最初の変更によって作業単位が開始される。
- システム関数の **sysLib.commit** および **sysLib.rollback** の使用が有効である。

EGL で生成された Java コードから VisualAge Generator COBOL プログラムを呼び出す場合、呼び出し先プログラムが戻ると、コミット (異常終了の場合はロールバック) が自動的に発行されます。コマンドは、呼び出し先プログラムによって行われた変更のみに有効です。

プロパティ **type** が **ejbCall** の場合、実行時の振る舞いは、SERVER で説明したとおりです。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

334 ページの『作業論理単位』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』

956 ページの『commit()』

969 ページの『rollback()』

453 ページの『callLink エLEMENTの pgmName』

460 ページの『callLink エLEMENTの type』

callLink エLEMENTの package

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **package** は、呼び出し先の Java プログラムが存在する Java パッケージを識別します。このプロパティが有効であるのは、プロパティ **type** が **ejbcall**、**localCall**、または **remoteCall** の場合です。

Java プログラムを生成する場合、**package** が有効となるのは、生成するプログラムによって呼び出されるプログラムがプロパティ **pgmName** で参照されるときで

す。Java ラッパーを生成する場合、**package** が有効となるのは、その Java ラッパー経由で呼び出されるプログラムがプロパティ **pgmName** で参照されるときです。

package プロパティを指定しない場合、呼び出し先プログラムは呼び出し側と同じパッケージ内にあると見なされます。

生成 Java プログラムで使用されるパッケージ名は EGL プログラムのパッケージ名ですが、小文字で表記されます。EGL が **callLink** エLEMENTの出力を生成するとき、**package** の値は (必要に応じて) 小文字に変更されます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの **callLink** ELEMENTの編集』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

442 ページの『**callLink** ELEMENT』

453 ページの『**callLink** ELEMENTの **pgmName**』

460 ページの『**callLink** ELEMENTの **type**』

callLink ELEMENTの parmForm

リンケージ・オプション・パーツの **callLink** ELEMENTのプロパティ **parmForm** は、呼び出しパラメーターの形式を指定します。

Java プログラムを生成する場合、**parmForm** は、次の状況で適用されます。

- 生成するプログラムで呼び出される CICS ベースのプログラムがプロパティ **pgmName** で参照される。
- プロパティ **type** が **ejbCall** または **remoteCall**。いずれの場合も、有効な **parmForm** 値 (後述) は **COMMDATA** (デフォルト) および **COMMPTR**。

Java ラッパーを生成する場合、**parmForm** は、次の場合に適用されます。

- Java ラッパー経由で呼び出される生成済み COBOL プログラムがプロパティ **pgmName** で参照される。
- プロパティ **type** が **ejbCall** または **remoteCall**。いずれの場合も、有効な **parmForm** 値 (後述) は **COMMDATA** (デフォルト) または **COMMPTR**。

次のリストから値を選択してください。

COMMDATA

呼び出し側がビジネス・データを、(データへのポインターを渡すのではなく) **COMMAREA** 内に格納することを指定します。

それぞれの引数値は、位置合わせを考慮せずに、先行する値に続けてバッファーに移されます。

プロパティ **type** が **ejbCall** または **remoteCall** の場合は、**COMMDATA** がデフォルト値になります。

COMMPTR

呼び出し側が以下のように振る舞うことを指定します。

- 渡される引数ごとに 1 つのポインターが対応するように、一連の 4 バイトのポインターを COMMAREA 内に作成する。
- 最後のポインターの高位ビットを 1 に設定する。

COMMPTR は、プロパティ **type** が localCall の場合、デフォルト値となります。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』

448 ページの『callLink エLEMENTの linkType』

452 ページの『callLink エLEMENTの parmForm』

『callLink エLEMENTの pgmName』

460 ページの『callLink エLEMENTの type』

callLink エLEMENTの pgmName

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **pgmName** は、callLink エLEMENTが参照するプログラム・パーツ名を指定します。

プログラム名にグローバル置換文字としてアスタリスク (*) を使用できます。ただし、この文字は最後の文字としてのみ有効です。詳細については、『リンケージ・オプション・パーツ』を参照してください。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』

callLink エLEMENTの providerURL

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **providerURL** は、EGL 生成の Java プログラムまたはラッパーが使用するネーム・サーバーのホスト名とポート番号を指定します。このネーム・サーバーにより、EJB セッション Bean が特定され、EGL 生成の Java プログラムが呼び出されます。このプロパティは、次の形式をとらなければなりません。

```
iiop://hostName:portNumber
```

hostName

ネーム・サーバーが動作しているマシンの IP アドレスまたはホスト名

portNumber

ネーム・サーバーが `listen` しているポート番号

プロパティ `providerURL` は、以下が揃った状況でのみ適用できます。

- プロパティ `type` の値が `ejbCall` である。
- プロパティ `pgmName` が、生成された Java プログラムまたはラッパーから呼び出されるプログラムを参照する。

ポート番号の前にピリオドまたはコロン of のいずれかがある場合に問題が発生しないように、URL を二重引用符で囲みます。

`providerURL` の値を指定しない場合、デフォルトが使用されます。デフォルトでは、EJB クライアントは、ローカル・ホスト上に存在して、ポート 900 で `listen` するネーム・サーバーを探すよう設定されます。このデフォルトは、次の URL と同等です。

```
"iiop://"
```

次の `providerURL` 値では、EJB クライアントが `bankserver.mybank.com` という名前で、ポート 9019 で `listen` するリモート・ネーム・サーバーを探すよう指示しています。

```
"iiop://bankserver.mybank.com:9019"
```

次のプロパティ値では、EJB クライアントが `bankserver.mybank.com` という名前で、ポート 900 で `listen` するリモート・ネーム・サーバーを探すよう指示しています。

```
"iiop://bankserver.mybank.com"
```

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの `callLink` エLEMENTの編集』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

442 ページの『`callLink` ELEMENT』

453 ページの『`callLink` ELEMENTの `pgmName`』

460 ページの『`callLink` ELEMENTの `type`』

callLink ELEMENTの refreshScreen

リンケージ・オプション・パーツ、`callLink` ELEMENTの `refreshScreen` プロパティは、呼び出されたプログラムが制御を戻すときに、画面を自動で最新表示するかどうかを指定します。有効な値は `yes` (デフォルト) と `no` です。

テキスト書式を表示する実行単位に呼び出し側が存在する場合、`refreshScreen` を `no` に設定します。以下の状況で適用されます。

- 呼び出し先プログラムがテキスト書式を表示しない
- 呼び出し側が、呼び出し後に全画面テキスト書式に書き込む

refreshScreen プロパティは、以下の場合のみ適用します。

- **callLink type** プロパティが **localCall**
- **remoteComType** プロパティがダイレクトまたは明確になっていて、**callLink type** プロパティが **remoteCall** の場合。

プロパティは、**call** ステートメントに **noRefresh** インディケーターが組み込まれている場合には、無視されます。

関連する参照項目

608 ページの『**call**』

callLink エLEMENTの remoteBind

リンケージ・オプション・パーツの **callLink** エLEMENTのプロパティ **remoteBind** は、生成時または実行時のどちらでリンケージ・オプションを決定するかを指定します。このプロパティは、次の状況でのみ適用されます。

- プロパティ **type** の値が **ejbCall** または **remoteCall** である。
- Java プログラムまたはラッパーを生成する。生成されるプログラムによって呼び出されるプログラムが、プロパティ **pgmName** で参照され、エントリーがプログラムからプログラムへの呼び出しを参照する場合。または、生成されるプログラムが、プロパティで参照され、エントリーがラッパーからプログラムへの呼び出しを参照する場合。

次の値から 1 つを選択します。

GENERATION

生成時に指定されたリンケージ・オプションが、実行時に必ず使用されます。
GENERATION はデフォルト値です。

RUNTIME

生成時に指定されたリンケージ・オプションが、デプロイメント時に変更される可能性があります。この場合は、ランタイム環境にリンケージ・プロパティ・ファイルを組み込む必要があります。

EGL は、以下のすべてが揃った状況下でリンケージ・プロパティ・ファイルを生成します。

- Java プログラムまたはラッパーを生成する。
- プロパティ **remoteBind** を **RUNTIME** に設定している。
- ビルド記述子オプション **genProperties** を **GLOBAL** または **PROGRAM** に設定して生成を行っている。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

396 ページの『リンケージ・プロパティ・ファイル』

関連するタスク

395 ページの『リンケージ・プロパティ・ファイルのデプロイ』

341 ページの『リンケージ・オプション・パーツの **callLink** エLEMENTの編集』

関連する参照項目

442 ページの『**callLink** エLEMENT』

428 ページの『genProperties』

708 ページの『リンケージ・プロパティ・ファイル (詳細)』

453 ページの『callLink エLEMENTの pgmName』

460 ページの『callLink エLEMENTの type』

callLink エLEMENTの remoteComType

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **remoteComType** は、次の場合に使用する通信プロトコルを指定します。

- プロパティ **type** の値が **ejbCall** または **remoteCall** である。
- Java プログラムまたはラッパーを生成する。
 - Java プログラムを生成する場合、生成するプログラムによって呼び出されるプログラムがプロパティ **pgmName** で参照される
 - Java ラッパーを生成する場合、その Java ラッパー経由で呼び出されるプログラムが **pgmName** で参照される

次の値から 1 つを選択します。

DEBUG

呼び出し側プログラムが Java ランタイムまたは Java デバッグ環境で実行されている場合でも、呼び出し先プログラムが EGL デバッガーで実行されます。この設定は以下の場合に使用することができます。

- EGL Java ラッパーを使用して EGL で書かれたプログラムを呼び出す Java プログラムを実行している。
- EGL で書かれたプログラムを呼び出す EGL 生成の呼び出し側プログラムを実行している。

前の状況は WebSphere テスト環境外部で発生することがありますが、JSP が EGL で書かれたプログラムを呼び出すときのように、その環境内で発生する場合もあります。いずれの場合も、EGL 生成のプログラムではなく、EGL ソースが呼び出されます。

WebSphere テスト環境を使用している場合は、呼び出し側と呼び出し先プログラムの両方がその環境で実行されている必要があります。リモート・マシンから呼び出しを行うことはできません。

DEBUG を使用する場合、同じ **callLink** エLEMENTに以下のプロパティを設定します。

- **library**。呼び出し先プログラムを含むプロジェクトの名前を付けます。
- **package**。呼び出し先プログラムを含むパッケージを識別します。ただし、呼び出し側と呼び出し元プログラムが同じパッケージ内にはない場合はこのプロパティを設定する必要はありません。

呼び出し側が EGL デバッガーを実行しておらず、WebSphere テスト環境で実行されていない場合は、以下に示す callLink エLEMENTのプロパティを設定する必要があります。

- **serverid** (リスナーのポート番号が 8346 でない場合は、番号を指定します)
- **location** (Eclipse ワークベンチを実行しているマシンのホスト名を含んでいる必要があります)

DIRECT

ローカル呼び出し側プログラムまたはラッパーで直接ローカル呼び出しを使用する場合に指定します。これは、呼び出し側と呼び出し先のコードを同じスレッド内で実行するときに適しています。呼び出される TCP/IP リスナーはありません。また、プロパティ **location** の値は無視されます。デフォルトは DIRECT です。

呼び出し側の Java プログラムは EGL ミドルウェアを使用しませんが、呼び出し側のラッパーはそのミドルウェアを使用して、EGL と Java プリミティブ型の間のデータ変換を処理します。

EGL で生成された Java コードから、EGL 以外で生成されたダイナミック・リンク・ライブラリー (DLL) または C か C++ のプログラムを呼び出す場合は、**remoteComType** 値 DISTINCT を使用することをお勧めします。

DISTINCT

プログラムをローカルで呼び出すときに新規実行単位を開始することを指定します。EGL ミドルウェアが使用されているので、呼び出しはリモートであるとみなされます。

この値は、ダイナミック・リンク・ライブラリー (DLL) または C か C++ のプログラムを呼び出す EGL 生成 Java プログラムに使用できます。

CICSECI

CICS Transaction Gateway (CTG) ECI インターフェースを使用することを指定します。また、CICS にアクセスする非 J2EE コードをデバッグまたは実行する際にも必要です。

CTG Java クラスを使用してこのプロトコルを実装します。CTG サーバーの URL とポートを指定するには、**callLink** エレメント、プロパティ **ctgLocation** および **ctgPort** に値を割り当てます。呼び出し先プログラムが存在する CICS 領域を識別するには、**location** プロパティを指定します。

CICSJ2C

CICS Transaction Gateway 対応の J2C コネクタを使用することを指定します。

CICSSSL

CICS Transaction Gateway (CTG) の Secure Socket Layer (SSL) 機能を使用することを指定します。SSL の JSSE 実装がサポートされています。

CTG Java クラスを使用してこのプロトコルを実装します。CTG サーバーの追加情報を指定するには、以下の **callLink** エレメントのプロパティに値を割り当てます。

- **ctgKeyStore**
- **ctgKeyStorePassword**
- **ctgLocation**
- **ctgPort**。この場合、CTG JSSE リスナーが要求を **listen** している TCP/IP ポートである。**ctgPort** を指定しない場合、CTG デフォルト・ポート 8050 が使用される。

呼び出し先プログラムが存在する CICS 領域を識別するには、**location** プロパティを指定します。

JAVA400

Java ラッパーまたはプログラムと、iSeries 用に (EGL または VisualAge Generator によって) 生成された COBOL プログラムとの通信に、IBM Toolbox for Java を使用することを指定します。

TCPIP

EGL ミドルウェアで TCP/IP を使用することを指定します。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

446 ページの『callLink エLEMENTの ctgKeyStore』

446 ページの『callLink エLEMENTの ctgKeyStorePassword』

446 ページの『callLink エLEMENTの ctgLocation』

447 ページの『callLink エLEMENTの ctgPort』

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

389 ページの『CICSJ2C 呼び出し用 J2EE サーバーのセットアップ』

390 ページの『J2EE アプリケーション・クライアント・モジュールにおける
呼び出し先アプリケーション用のTCP/IP リスナーのセットアップ』

383 ページの『呼び出し先となる非J2EE アプリケーションのTCP/IP
リスナーのセットアップ』

callLink エLEMENTの remotePgmType

リンケージ・オプション・パーツの callLink エLEMENTのプロパティー **remotePgmType** は、呼び出すプログラムの種類を指定します。このプロパティーは、次の状況で適用されます。

- プロパティー **type** の値が **ejbCall** または **remoteCall** である。
- 次のいずれかが該当する場合。
 - (ラッパーではなく) プログラムを生成する場合、生成するプログラムが呼び出すプログラムを、プロパティー **pgmName** が参照する。

呼び出し先プログラムは、次のいずれかの種類となります。

- EGL 生成の Java プログラム
- 非 EGL 生成のダイナミック・リンク・ライブラリー (DLL) または C か C++ のプログラム
- CICS 上で動作し、CICS コマンドを含むプログラム
- Java ラッパーを生成する場合、その Java ラッパー経由で呼び出されるプログラムが **pgmName** で参照される。

EGL

呼び出し先プログラムは、EGL または VisualAge Generator で生成された COBOL または Java プログラムです。この場合、呼び出し側は、Java プログラムまたは Java ラッパーです。この値はデフォルトです。

関連する概念

338 ページの『リンケージ・オプション・パーツ』
10 ページの『ランタイム構成』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』
448 ページの『callLink エLEMENTの library』
453 ページの『callLink エLEMENTの pgmName』
460 ページの『callLink エLEMENTの type』

callLink エLEMENTの serverID

リンケージ・オプション・パーツの callLink エLEMENTの **serverID** プロパティは、以下の値のいずれかを指定します。

- 呼び出し先プログラムのリスナーの TCP/IP ポート番号。TCP/IP プロトコルが使用中の場合に限ります。このケースにはデフォルトは存在しません。
- CICS トランザクションを呼び出し中の ID。ただし、ECI インターフェースまたは CICS Transaction Gateway の Secure Socket Layer 機能を使用して CICS にアクセスする場合に限ります。このケースでは、デフォルトは CICS サーバー・システムのミラー・トランザクションです。

このプロパティは、以下がすべて揃った状況でのみ使用されます。

- プロパティ **type** の値が **ejbCall** または **remoteCall** である。
- **remoteComType** の値が **TCPIP**、**CICSECI**、または **CICSSSL** である。
- Java プログラムまたはラッパーを生成する。
 - Java プログラムを生成する場合、生成するプログラムによって呼び出されるプログラムがプロパティ **pgmName** で参照される
 - Java ラッパーを生成する場合、その Java ラッパー経由で呼び出されるプログラムが **pgmName** で参照される

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』
390 ページの『J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用のTCP/IP リスナーのセットアップ』
383 ページの『呼び出し先となる非J2EE アプリケーションのTCP/IP リスナーのセットアップ』

関連する参照項目

442 ページの『callLink エLEMENT』
453 ページの『callLink エLEMENTの pgmName』
456 ページの『callLink エLEMENTの remoteComType』
460 ページの『callLink エLEMENTの type』

callLink エLEMENTの type

リンケージ・オプション・パーツの callLink エLEMENTのプロパティ **type** は、呼び出しの種類を指定します。次の値から 1 つを選択します。

ejbCall

生成された Java プログラムまたはラッパーが、EJB セッション Bean を使用するプログラム呼び出しを実装すること、および EJB セッション Bean がプロパティ **pgmName** で示されたプログラムにアクセスすることを示しています。値 **ejbCall** は、以下の 2 つのいずれかの場合に適用できます。

- Java ラッパーを生成しており、ラッパーはそのプログラムを EJB セッション Bean を介して呼び出す。この場合、プロパティ **pgmName** はラッパーから呼び出されるプログラムを参照し、**ejbCall** の使用により EJB セッション Bean が生成されます。
- EJB セッション Bean を介して、生成されたプログラムを呼び出す Java プログラムを生成している。この場合、プロパティ **pgmName** は呼び出し先プログラムを参照し、EJB セッション Bean は生成されません。

いずれのケースでも、EJB セッション Bean を使用する場合、EJB セッション Bean を生成するためにも Java ラッパーを生成する必要があります。

生成されたセッション Bean はエンタープライズ Java サーバーにデプロイする必要があり、次のどちらかが該当しなければなりません。

- EJB セッション Bean を配置するために使用されるネーム・サーバーが、その Session Bean を呼び出すコードと同じマシン上に存在する。
- プロパティ **providerURL** が、ネーム・サーバーの存在する場所を示している。

EJB セッション Bean を使用する場合、リンケージ・オプション・パーツを指定して、呼び出し側プログラムまたはラッパーを生成する必要があります。この場合、呼び出し先プログラム用のプロパティ **type** の値は **ejbCall** です。セッション Bean の使用の決定をデプロイメント時に行うことはできません。ただし、プロパティ **remoteBind** を **RUNTIME** と設定した場合、EJB セッション Bean が生成されたプログラムにアクセスする方法についてはデプロイメント時に決定できます。しかし、この決定は生成時に行う方が、効率的です。

localCall

呼び出しで EGL ミドルウェアを使用しないことを指定します。この場合、呼び出し先プログラムは、呼び出し側と同一プロセス内になります。

localCall はデフォルト値であり、

remoteCall

呼び出しが EGL ミドルウェアを使用することを指定します。これにより、渡されるデータの終わりに 12 バイトが付加されます。これらのバイトによって、呼び出し側は、呼び出し先プログラムからの戻り値を受け取ることができます。

呼び出し側が Java コードの場合、通信は、プロパティ **remoteComType** で指定されたプロトコルによって処理されます。プロトコルの選択は、呼び出し先プログラムが同一スレッド内にあるか、異なるスレッド内にあるかを示しています。

呼び出しで可変長レコードが渡される場合は、次のことが適用されます。

- レコードに対して指定された最大長のスペースが予約されます。
- callLink プロパティ **type** の値が remoteCall または ejbCall の場合、可変長項目が存在する場合は、それがレコード内部にある必要があります。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

341 ページの『リンケージ・オプション・パーツの callLink エLEMENTの編集』

関連する参照項目

442 ページの『callLink エLEMENT』
 448 ページの『callLink エLEMENTの linkType』
 449 ページの『callLink エLEMENTの location』
 452 ページの『callLink エLEMENTの parmForm』
 453 ページの『callLink エLEMENTの pgmName』
 453 ページの『callLink エLEMENTの providerURL』
 456 ページの『callLink エLEMENTの remoteComType』

C 関数と EGL との併用

EGL プログラムは C 関数を呼び出すことができます。

C 関数を EGL から呼び出す手順は、次のとおりです。

ご使用の EGL プログラムで使用する C 関数を確認したら、以下を行う必要があります。

1. EGL スタック・ライブラリーおよびアプリケーション・オブジェクト・ファイルを、IBM Web サイトからコンピューターにダウンロードする。
2. すべての C コードを 1 つの共用ライブラリーにコンパイルし、適切なプラットフォーム固有のスタック・ライブラリーにリンクする。
3. 関数テーブルを作成する。
4. この関数テーブルと適切なプラットフォーム固有のアプリケーション・オブジェクト・ファイルを共用ライブラリーにコンパイルし、この共用ライブラリーを、ステップ 2 で作成した共用ライブラリーとスタック・ライブラリーにリンクする。

1. EGL スタック・ライブラリーとアプリケーション・オブジェクト・ファイルをダウンロードする

EGL スタック・ライブラリーとアプリケーション・オブジェクト・ファイルをダウンロードする手順は、次のとおりです。

1. EGL Support Web サイトにアクセスする。
 - Rational Application Developer の URL は、次のとおりです。
<http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rad/60/redist>
 - Rational Web Developer の URL は、次のとおりです。
<http://www3.software.ibm.com/ibmdl/pub/software/rationalsdp/rwd/60/redist>

2. **EGLRuntimesV60IFix001.zip** ファイルを適切なディレクトリーにダウンロードする。
3. **EGLRuntimesV60IFix001.zip** を unzip して、以下のファイルがあることを確認する。

プラットフォーム固有のスタック・ライブラリーの場合

- AIX : EGLRuntimes/Aix/bin/libstack.so
- Linux : EGLRuntimes/Linux/bin/libstack.so
- Win32 :
EGLRuntimes/Win32/bin/stack.dll
EGLRuntimes/Win32/bin/stack.lib

.

プラットフォーム固有のアプリケーション・オブジェクト・ファイルの場合

- AIX : EGLRuntimes/Aix/bin/application.o
- Linux : EGLRuntimes/Linux/bin/application.o
- Win32 : EGLRuntimes/Win32/bin/application.obj

.

2. すべての C コードを共用ライブラリーにコンパイルする

C コードは、ポップ外部関数を使用して EGL から値を受け取り、戻り外部関数を使用して EGL に値を戻します。ポップ外部関数については、『EGL からの値の受け取り』で、戻り外部関数については、『EGL への値の戻り』で説明します。

すべての C コードを共用ライブラリーにコンパイルする手順は、次のとおりです。

1. 標準メソッドを使用して、すべての C コードを 1 つの共用ライブラリーにコンパイルし、適切なプラットフォーム固有の EGL スタック・ライブラリーにリンクする。
2. 次のプラットフォーム固有の例では、**file1.c** と **file2.c** は、EGL から呼び出された関数を含む C ファイルです。

AIX では、以下ようになります (ld コマンドは単一行上に存在する必要があります)。

```
cc -c -Iincl_dir file1.c file2.c
ld -G -b32 -bexpall -bnoentry
    -brtl file1.o file2.o -Lstack_lib_dir
    -lstack -o lib1_name -lc
```

Linux では、以下ようになります (gcc コマンドは単一行上に存在する必要があります)。

```
cc -c -Iincl_dir file1.c file2.c
gcc -shared file1.o file2.o -Lstack_lib_dir
    -lstack -o lib1_name
```

Windows では、以下ようになります (link コマンドは単一行上に存在する必要があります)。

```

cl /c -lincl_dir file1.c file2.c
link /DLL file1.obj file2.obj
    /LIBPATH:stack_lib_dir
    /DEFAULTLIB:stack.lib /OUT:lib1_name

```

incl_dir

ヘッダー・ファイルのディレクトリー・ロケーション

stack_lib_dir

スタック・ライブラリーのディレクトリー・ロケーション

lib1_name

出力ライブラリーの名前

注: C コードが IBM Informix ESQL/C ライブラリー関数 (BIGINT、DECIMAL、DATE、INTERVAL、DATETIME) を使用している場合、ESQL/C ライブラリーにもリンクする必要があります。

3. 関数テーブルを作成する

関数テーブルとは、EGL プログラムから呼び出されるすべての C 関数の名前を含む C ソース・ファイルです。次の関数テーブル例では、**c_fun1** と **c_fun2** が C 関数の名前です。コード内で識別されるすべての関数は、上記のステップ 2 で作成した C 共用ライブラリーからエクスポート済みでなければなりません。

```

#include <stdio.h>
struct func_table {

    char *fun_name;
    int (*fptr)(int);
};

extern int c_fun1(int);
extern int c_fun2(int);
/* 他の関数に対しても同様のプロトタイプ */

struct func_table ftab[] =
{
    "c_fun1", c_fun1,
    "c_fun2", c_fun2,
    /* 他の関数に対しても同様 */
    "", NULL
};

```

上記の例を基にして関数テーブルを作成し、適切な C 関数を使用して関数テーブルにデータを取り込みます。 "", **NULL** で関数テーブルの終わりを示します。

4. 関数テーブルとプラットフォーム固有のアプリケーション・オブジェクトを共用ライブラリーにコンパイルする

アプリケーション・オブジェクト・ファイルは、EGL コードと C コードとの間のインターフェースです。

次の 2 つの成果物を 1 つの共用ライブラリーにコンパイルし、上記のステップ 2 で作成したスタック・ライブラリーおよびライブラリーにリンクします。

- 関数テーブル
- アプリケーション・オブジェクト・ファイル

次のサンプルを使用して新規の共用ライブラリーをコンパイルします。ここで **ftable.c** は、関数テーブルの名前、**mylib** は、ステップ 2 で作成した C 共用ライブラリーの名前、**lib_dir** は、**mylib** のディレクトリー・ロケーションです。*dllName* プロパティーまたは *vgj.defaultI4GLNativeLibrary* Java ランタイム・プロパティーを使用して **lib2_name** を指定します。

AIX では、以下ようになります (ld コマンドは単一行上に存在する必要があります)。

```
cc -c ftable.c
ld -G -b32 -bexpall -bnoentry
    -brtl ftable.o application.o
    -Lstack_lib_dir -lstack -Llib_dir
    -lmylib -o lib2_name -lc
```

Linux では、以下ようになります (gcc コマンドは単一行上に存在する必要があります)。

```
cc -c ftable.c
gcc -shared ftable.o application.o
    -Lstack_lib_dir -lstack -Llib_dir
    -lmylib -o lib2_name
```

Windows では、以下ようになります (link コマンドは単一行上に存在する必要があります)。

```
cl /c ftable.c
link /DLL ftable.obj application.obj
    /LIBPATH:stack_lib_dir
    /DEFAULTLIB:stack.lib
    /LIBPATH:lib_dir
    /DEFAULTLIB:mylib.lib /OUT:lib2_name
```

これら 3 つのライブラリーを一緒にリンクします。

C 共用ライブラリー、関数テーブル、およびスタック・ライブラリーがリンクされると、EGL コードから C 関数を呼び出すことができる状態になります。EGL 内で C 関数を呼び出す方法については、『EGL プログラムからの C 関数の呼び出し』を参照してください。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

465 ページの『C の BIGINT 関数』

466 ページの『C データ型と EGL プリミティブ型』

467 ページの『C の DATE 関数』

467 ページの『C の DATETIME および INTERVAL 関数』

468 ページの『C の DECIMAL 関数』

470 ページの『EGL プログラムからの C 関数の呼び出し』

474 ページの『C の Return 関数』

471 ページの『C の Stack 関数』

C の BIGINT 関数

注: 次の BIGINT 関数は、IBM Informix ESQL/C のユーザーのみが使用できます。
これらの関数を使用するには、ESQL/C ユーザーは、自身の C コードを手動で ESQL/C ライブラリーにリンクする必要があります。

BIGINT データ型は、 $-2^{63}-1$ から $2^{63}-1$ の範囲の数を表すための、マシンに依存しないメソッドです。ESQL/C は、BIGINT データ型を C 言語の他のデータ型に変換するのを容易にするルーチンを提供します。

BIGINT データ型は、内部で **ifx_int8_t** 構造体と一緒に表示されます。この構造体に関する情報は、ヘッダー・ファイル **int8.h** 内で検出されます。このヘッダー・ファイルは、ESQL/C 製品に含まれています。このファイルを、BIGINT 関数を使用するすべての C ソース・ファイルにインクルードしてください。

int8 型の数の演算はすべて、**int8** データ型用の次の ESQL/C ライブラリー関数を使用して実行する必要があります。その他の演算、変更、または分析を実行すると、予測不能の結果が生じる可能性があります。ESQL/C ライブラリーは、**int8** 数を操作し、**int8** 型の数をその他のデータ型に変換（あるいはその逆の変換）を行うことができる、次の関数を提供します。

関数名	説明
ifx_int8add()	2 つの BIGINT 型の値を加算する。
ifx_int8cmp()	2 つの BIGINT 型の数を比較する。
ifx_int8copy()	ifx_int8_t 構造体をコピーする。
ifx_int8cvasc()	C char 型の値を BIGINT 型の数値に変換する。
ifx_int8cvdbl()	C double 型の値を BIGINT 型の数値に変換する。
ifx_int8cvdec()	decimal 型の値を BIGINT 型の値に変換する。
ifx_int8cvflt()	float 型の値を BIGINT 型の値に変換する。
ifx_int8cvint()	C int 型の数値を BIGINT 型の数値に変換する。
ifx_int8cvlong()	C の long (64 ビット・マシン上では int) 型の値を BIGINT 型の値に変換する。
ifx_int8cvlong_long()	C の long long 型 (8 バイト値、32 ビットの long long 、および 64 ビットの long) の値を BIGINT 型の値に変換する。
ifx_int8div()	2 つの BIGINT 数を除算する。
ifx_int8mul()	2 つの BIGINT 数を乗算する。
ifx_int8sub()	2 つの BIGINT 数を減算する。
ifx_int8toasc()	BIGINT 型の値を C char 型の値に変換する。
ifx_int8todbl()	BIGINT 型の値を C の double 型の値に変換する。
ifx_int8todec()	BIGINT 型の数値を decimal 型の数値に変換する。
ifx_int8toflt()	BIGINT 型の数値を C の float 型の数値に変換する。
ifx_int8toint()	BIGINT 型の値を C の int 型の値に変換する。
ifx_int8tolong()	BIGINT 型の値を C の long (64 ビット・マシン上では int) 型の値に変換する。
ifx_int8tolong_long()	C の long long (64 ビット・マシン上では long) 型を BIGINT 型の値に変換する。

関連する参照項目

個々の関数の詳細については、次のものを参照してください。

「IBM Informix ESQL/C Programmer's Manual」

467 ページの『C の DATE 関数』

467 ページの『C の DATETIME および INTERVAL 関数』

468 ページの『C の DECIMAL 関数』

470 ページの『EGL プログラムからの C 関数の呼び出し』

C データ型と EGL プリミティブ型

次の表では、C データ型、I4GL データ型、および EGL プリミティブ型との間のマッピングを示します。

C データ型	対応する I4GL データ型	対応する EGL プリミティブ型
char	CHAR または CHARACTER	UNICODE(1)
char	NCHAR	UNICODE(size)
char	NVARCHAR	STRING
char	VARCHAR	STRING
int	INT または INTEGER	INT
short	SMALLINT	SMALLINT
ifx_int8_t	BIGINT	BIGINT
dec_t	DEC または DECIMAL(p,s) または NUMERIC(p)	DECIMAL(p)
dec_t	MONEY	MONEY
double	FLOAT	FLOAT
float	SMALLFLOAT	SMALLFLOAT
loc_t	TEXT	CLOB
loc_t	BYTE	BLOB
int	DATE	DATE
dtime_t	DATETIME	TIMESTAMP
intvl_t	INTERVAL	INTERVAL

関連する参照項目

53 ページの『BIN および整数型』

52 ページの『BLOB』

51 ページの『CLOB』

44 ページの『DATE』

54 ページの『DECIMAL』

54 ページの『FLOAT』

45 ページの『INTERVAL』

470 ページの『EGL プログラムからの C 関数の呼び出し』

43 ページの『MBCHAR』

55 ページの『MONEY』

55 ページの『NUM』
37 ページの『プリミティブ型』
56 ページの『SMALLFLOAT』
47 ページの『TIME』
47 ページの『TIMESTAMP』

C の DATE 関数

注: 次の DATE 関数は、IBM Informix ESQL/C のユーザーのみが使用できます。
これらの関数を使用するには、ESQL/C ユーザーは、自身の C コードを手動で ESQL/C ライブラリーにリンクする必要があります。

以下のデータ操作関数が ESQL/C ライブラリー内にあります。これらの関数は、ストリング形式と内部 DATE 形式との間で日付を変換します。

関数名	説明
rdatestr()	内部 DATE を文字ストリング形式に変換する。
rdayofweek()	日付の曜日を内部形式で返す。
rdefmtdate()	指定のストリング形式を内部 DATE に変換する。
rfmtdate()	内部 DATE を指定のストリング形式に変換する。
rjulmdy()	指定の DATE から月、日、および年を返す。
rleapyear()	指定した年がうるう年であるかどうかを判別する。
rmidyjul()	月、日、および年から内部 DATE を返す。
rstodate()	文字ストリング形式を内部 DATE に変換する。
rtoday()	システム日付を内部 DATE として返す。

関連する参照項目

個々の関数の詳細については、次のものを参照してください。
「IBM Informix ESQL/C Programmer's Manual」

465 ページの『C の BIGINT 関数』
『C の DATETIME および INTERVAL 関数』
468 ページの『C の DECIMAL 関数』
470 ページの『EGL プログラムからの C 関数の呼び出し』

C の DATETIME および INTERVAL 関数

注: 次の DATETIME 関数および INTERVAL 関数は、IBM Informix ESQL/C のユーザーのみが使用できます。これらの関数を使用するには、ESQL/C ユーザーは、自身の C コードを手動で ESQL/C ライブラリーにリンクする必要があります。

DATETIME データ型および INTERVAL データ型は、それぞれ **dttime_t** 構造体および **intrvl_t** 構造体と共に内部で表示されます。これらの構造体に関する情報は、ヘッダー・ファイル **datetime.h** 内で見つけることができます。このヘッダー・ファ

イルは、ESQL/C 製品に含まれています。このファイルを、DATETIME 関数および INTERVAL 関数を使用するすべての C ソース・ファイルにインクルードしてください。

datetime データ型および **interval** データ型に次の ESQL/C ライブラリー関数を使用して、これらの型の値ですべての演算を実行する必要があります。

関数名	説明
dtaddinv()	間隔値を日時値に加算する。
dtcurrent()	現在の日付および時刻を取得する。
dtcvasc()	ANSI 準拠の文字ストリングを日時値に変換する。
dtcvfmtasc()	指定の書式を持つ文字ストリングを日時値に変換する。
dtextend()	日時値の修飾子を変更する。
dtsub()	日時値を別の日時値から減算する。
dsubinv()	間隔値を日時から減算する。
dttoasc()	日時値を ANSI 準拠の文字ストリングに変換する。
dttofmtasc()	日時値を、指定の書式を持つ文字ストリングに変換する。
incvasc()	ANSI 準拠の文字ストリングを間隔値に変換する。
incvfmtasc()	指定の書式を持つ文字ストリングを間隔値に変換する。
intoasc()	間隔値を ANSI 準拠の文字ストリングに変換する。
intofmtasc()	間隔値を、指定の書式を持つ文字ストリングに変換する。
invdivdbl()	間隔値を数値で除算する。
invdivinv()	間隔値を別の間隔値で除算する。
invextend()	間隔値を別の間隔修飾子に拡張する。
invmuldbl()	間隔値を数値で乗算する。

関連する参照項目

個々の関数の詳細については、次のものを参照してください。

「IBM Informix ESQL/C Programmer's Manual」

465 ページの『C の BIGINT 関数』

467 ページの『C の DATE 関数』

『C の DECIMAL 関数』

470 ページの『EGL プログラムからの C 関数の呼び出し』

C の DECIMAL 関数

注: 次の DECIMAL 関数は、IBM Informix ESQL/C のユーザーのみが使用できます。これらの関数を使用するには、ESQL/C ユーザーは、自身の C コードを手動で ESQL/C ライブラリーにリンクする必要があります。

DECIMAL データ型は、小数点の有無にかかわらず、-128 から +126 の範囲の指数を持つ最高 32 桁までの有効数字を表す、マシンから独立した方法です。 ESQL/C

は、DECIMAL 型の数値から、C 言語で利用できる各データ型 (あるいはその逆) に変換するのを容易にするルーチンを提供します。DECIMAL 型の数値は、底を 100 とする指数と仮数 (または小数部分) で構成されます。正規形では、仮数の最初の桁は、ゼロより大きくなければなりません。

DECIMAL データ型は、内部で **dec_t** 構造体と一緒に表示されます。**decimal** 構造体および型定義 **dec_t** は、ヘッダー・ファイル **decimal.h** 内で検出されます。このヘッダー・ファイルは、ESQL/C 製品に含まれています。このファイルを、decimal 関数を使用するすべての C ソース・ファイルにインクルードしてください。

decimal 型の数の演算はすべて、**decimal** データ型用の次の ESQL/C ライブラリー関数を使用して実行する必要があります。その他の演算、変更、または分析を実行すると、予測不能の結果が生じる可能性があります。

関数名	説明
deccvasc()	C の int1 型を DECIMAL 型に変換する。
dectoasc()	DECIMAL 型を C の int1 型に変換する。
deccvint()	C の int 型を DECIMAL 型に変換する。
dectoint()	DECIMAL 型を C の int 型に変換する。
deccvlong()	C の int4 型を DECIMAL 型に変換する。
dectolong()	DECIMAL 型を C の int4 型に変換する。
deccvflt()	C の float 型を DECIMAL 型に変換する。
dectoflt()	DECIMAL 型を C の float 型に変換する。
deccvdbl()	C の double 型を DECIMAL 型に変換する。
dectodbl()	DECIMAL 型を C の double 型に変換する。
decadd()	2 つの DECIMAL 数値を加算する。
decsub()	2 つの DECIMAL 数値を減算する。
decmul()	2 つの DECIMAL 数値を乗算する。
decdiv()	2 つの DECIMAL 数値を除算する。
deccmp()	2 つの DECIMAL 数値を比較する。
deccopy()	DECIMAL 数値をコピーする。
decevt()	DECIMAL 値を ASCII ストリングに変換する。
decfcvt()	DECIMAL 値を ASCII ストリングに変換する。

関連する参照項目

個々の関数の詳細については、次のものを参照してください。

「IBM Informix ESQL/C Programmer's Manual」

465 ページの『C の BIGINT 関数』

467 ページの『C の DATE 関数』

467 ページの『C の DATETIME および INTERVAL 関数』

470 ページの『EGL プログラムからの C 関数の呼び出し』

EGL プログラムからの C 関数の呼び出し

EGL プログラムから C 関数を呼び出すことができます。下記の手順を実行する前に、『C 関数と EGL との併用』で説明されているように C コードをコンパイルし、リンクする必要があります。

C 関数を EGL プログラムから呼び出す手順は、次のとおりです。

1. 関数呼び出し ステートメントを使用して、以下を指定する。
 - C 関数の名前
 - C 関数に渡すすべての引数
 - EGL プログラムに戻すすべての変数
2. 関数定義を含む EGL ネイティブ・ライブラリー・パーツ を作成する。
3. USE ステートメントを使用して、呼び出し側モジュール内に EGL ネイティブ・ライブラリー・パーツを指定する。

例えば、次の関数呼び出し文は、C 関数 `sendmsg()` を呼び出します。

```
sendmsg(chartype, 4, msg_status, return_code);
```

この文は、2 つの引数 (`chartype` と `4` のそれぞれ) を関数に渡し、2 つの引数 (`msg_status` と `return_code` のそれぞれ) が戻されることを期待します。これを明確にするには、ネイティブ・ライブラリー内で関数を次のように定義します。

```
Library I4GLFunctions type nativeLibrary
{callingConvention = "I4GL", dllName = "mydll"}
Function sendmsg(chartype char(10) in, i int in, msg_status int out,
return_code int out)
end
end
```

渡される引数は、「in」パラメーターを使用して指定され、戻される引数は、「out」パラメーターを使用して指定されます。

callingConvention

引数スタック・メカニズムを使用して、引数が関数と呼び出し側コードとの間で渡されることを指定します。

dllName

この関数が存在する C 共用ライブラリーを指定します。

注: この C 共用ライブラリー名は、`vgj.defaultI4GLNativeLibrary` システム・プロパティを使用して指定することもできます。 `dllName` とシステム・プロパティの両方が指定されている場合は、`dllName` が使用されます。 EGL `nativeLibrary` について詳しくは、「*nativeLibrary* 型のライブラリー・パーツ」ヘルプ・トピックを参照してください。

C 関数は、引数スタック (この場合、2 つの引数) にプッシュされる値の数を指定する整数引数を受け取ります。これは、C 関数内のスタックからポップされる値の数です。また、この関数は、EGL プログラムに制御を戻す前に、`msg_status` 引数および `return_code` 引数の値を戻す必要があります。ポップ外部関数については、『EGL からの値の受け取り』で、戻り外部関数については、『EGL への値の戻り』で説明します。

C 関数は、正しい数のスタック値が渡されたことを前提としません。C 関数は、その関数用にスタックされた EGL 引数の数を確認するために、その整数引数をテストします。

次の例は、正確に 1 つの引数を必要とする C 関数を示しています。

```
int nxt_bus_day(int nargs);
{
    int theDate;
    if (nargs != 1)
    {
        fprintf(stderr,
            "nxt_bus_day: wrong number of parms (%d)\n",
            nargs );
        ibm_lib4gl_returnDate(0L);
        return(1);
    }
    ibm_lib4gl_popDate(&theDate);
    switch(rdayofweek(theDate))
    {
        case 5: /* 金曜日 -> 月曜日に変更 */
            ++theDate;
        case 6: /* 土曜日 -> 月曜日*/
            ++theDate;
        default: /* (sun..thur) 翌日にジャンプ */
            ++theDate;
    }
    ibm_lib4gl_returnDate(theDate); /* スタックの結果 */
    return(1) /* スタックされたカウントを戻す */
}
```

この関数は、所定の日付の翌営業日の日付を戻します。この関数は、正確に 1 つの引数を受け取る必要があるので、渡された引数の数をチェックします。この関数が異なる数の引数を受け取った場合、確認メッセージを出してプログラムを終了します。

関連する参照項目

- 465 ページの『C の BIGINT 関数』
- 466 ページの『C データ型と EGL プリミティブ型』
- 150 ページの『EGL ライブラリー・パーツの作成』
- 467 ページの『C の DATE 関数』
- 467 ページの『C の DATETIME および INTERVAL 関数』
- 468 ページの『C の DECIMAL 関数』
- 560 ページの『関数呼び出し』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 『C の Stack 関数』
- 474 ページの『C の Return 関数』
- 461 ページの『C 関数と EGL との併用』

C の Stack 関数

C 関数を呼び出すために、EGL は引数スタック を使用します。これは、関数と呼び出し側コード間で引数を渡すメカニズムです。EGL 呼び出し関数は、その引数をスタックにプッシュし、呼び出し先の C 関数は、引数をスタックからポップして値を使用します。呼び出し先の関数は、その戻り値をスタックにプッシュし、呼び出し側は、戻り値をスタックからポップして値を取得します。ポップ外部関数および戻り外部関数は、引数スタック・ライブラリーと一緒に提供されます。ポップ外

部関数は、引数スタックからそれぞれがポップする値のデータ型に応じて以下のよう
に記述されます。戻り外部関数は、『C の *Return* 関数』で説明します。

注: ポップ関数は、当初 IBM Informix 4GL (I4GL) と一緒に使用されていたので、
関数名に「4gl」が含まれています。

値を戻すためのライブラリー関数

引数スタックから数値をポップするには、次のライブラリー関数を C 関数から呼び
出すことができます。

- extern void ibm_lib4gl_popMInt(int *iv)
- extern void ibm_lib4gl_popInt2(short *siv)
- extern void ibm_lib4gl_popInt4(int *liv)
- extern void ibm_lib4gl_popFloat(float *fv)
- extern void ibm_lib4gl_popDouble(double *dfv)
- extern void ibm_lib4gl_popDecimal(dec_t *decv)
- extern void ibm_lib4gl_popInt8(ifx_int8_t *bi)

次の表、および下記の同様な表では、I4GL バージョン 7.31 より前のバージョン
と、バージョン 7.31 およびそれ以降のバージョンとの間で、戻り関数名をマップし
ています。

バージョン 7.31 より前の名前	バージョン 7.31 およびそれ以降の名前
popint	ibm_lib4gl_popMInt
popshort	ibm_lib4gl_popInt2
poplong	ibm_lib4gl_popInt4
popflo	ibm_lib4gl_popFloat
popdub	ibm_lib4gl_popDouble
popdec	ibm_lib4gl_popDecimal

上記の各関数は、値をポップするためのすべてのライブラリー関数と同様に、以下
のアクションを実行します。

1. 引数スタックから 1 つの値を除去する。
2. 必要な場合、そのデータ型を変換する。スタックの値が指定した型に変換できな
い場合、エラーが発生します。
3. 値を指定の変数にコピーする。

構造型 **dec_t** および **ifx_int8_t** は、C プログラムにおける DECIMAL および
BIGINT データを表すのに使用されます。 **dec_t** と **ifx_int8_t** 構造型、および
DECIMAL 変数と BIGINT 変数を操作および印刷するためのライブラリー関数につ
いての詳細は、「*IBM Informix ESQL/C Programmer's Manual*」を参照してくださ
い。

文字ストリングをポップするためのライブラリー関数

文字の値をポップするには、次のライブラリー関数を呼び出すことができます。

- extern void ibm_lib4gl_popQuotedStr(char *qv, int len)

- extern void ibm_lib4gl_popString(char *qv, int len)
- extern void ibm_lib4gl_popVarChar(char *qv, int len)

バージョン 7.31 より前の名前	バージョン 7.31 およびそれ以降の名前
popquote	ibm_lib4gl_popQuotedStr
popstring	ibm_lib4gl_popString
popvchar	ibm_lib4gl_popVarChar

ibm_lib4gl_popQuotedStr() および **ibm_lib4gl_popVarChar()** の両方は、**len** バイトをストリング・バッファー ***qv** に正確にコピーします。

ibm_lib4gl_popQuotedStr() は、必要に応じてスペースが埋め込まれますが、**ibm_lib4gl_popVarChar()** は、フルの長さまで埋め込まれません。バッファーにコピーされる最終バイトは、ストリングを終了する NULL バイトであるので、最大のストリング・データ長は **len-1** です。スタック引数が **len-1** を超える場合、その末尾バイトが失われます。

len 引数は、受信ストリング・バッファーの最大サイズを設定します。

ibm_lib4gl_popQuotedStr() を使用すると、スタック上の値が空ストリングであっても、正確に **len** バイト (末尾ブランク・スペースおよび NULL を含む) を受け取ります。 **ibm_lib4gl_popQuotedStr()** によって検索されたストリングの真のデータ長を求めるには、ポップされた値から末尾スペースを削除する必要があります。

注: 関数 **ibm_lib4gl_popString()** と **ibm_lib4gl_popQuotedStr()** は、**ibm_lib4gl_popString()** が自動的にすべての末尾ブランクを除去することを除いて、同一です。

時刻値をポップするためのライブラリー関数

DATE、INTERVAL、および DATETIME (TIMESTAMP) 値をポップするには、次のライブラリー関数を呼び出すことができます。

- extern void ibm_lib4gl_popDate(int *datv)
- extern void ibm_lib4gl_popInterval(intrvl_t *iv, int qual)

TIMESTAMP 値をポップするには、次のライブラリー関数を呼び出すことができます。

- extern void ibm_lib4gl_popDateTime(dtime_t *dtv, int qual)

バージョン 7.31 より前の名前	バージョン 7.31 およびそれ以降の名前
popdate	ibm_lib4gl_popDate
popdtime	ibm_lib4gl_popDateTime
popinv	ibm_lib4gl_popInterval

構造型 **dtime_t** および **intrvl_t** は、C プログラムにおける DATETIME および INTERVAL データを表すのに使用されます。 **qual** 引数は、DATETIME または INTERVAL 修飾子のバイナリー表記を受け取ります。 **dtime_t** と **intrvl_t** 構造型、

および DATE 変数、 DATETIME 変数、INTERVAL 変数を操作および印刷するためのライブラリー関数についての詳細は、「*IBM Informix ESQL/C Programmer's Manual*」を参照してください。

BYTE または TEXT 値をポップするためのライブラリー関数

BYTE または TEXT 引数をポップするには、次のライブラリー関数を呼び出すことができます。

```
• extern void ibm_lib4gl_popBlobLocator(loc_t **blob)
```

バージョン 7.31 より前の名前	バージョン 7.31 およびそれ以降の名前
poplocator	ibm_lib4gl_popBlobLocator

構造型 **loc_t** は、BYTE または TEXT 値を定義します。詳細は「*IBM Informix ESQL/C Programmer's Manual*」を参照してください。

すべての BYTE または TEXT 引数は、BYTE または TEXT としてポップする必要があります。これは、EGL が自動データ型変換を行わないからです。

関連する参照項目

- 465 ページの『C の BIGINT 関数』
- 466 ページの『C データ型と EGL プリミティブ型』
- 461 ページの『C 関数と EGL との併用』
- 467 ページの『C の DATE 関数』
- 467 ページの『C の DATETIME および INTERVAL 関数』
- 468 ページの『C の DECIMAL 関数』
- 470 ページの『EGL プログラムからの C 関数の呼び出し』
- IBM Informix ESQL/C Programmer's Manual
- 『C の Return 関数』

C の Return 関数

C 関数を呼び出すために、EGL は**引数スタック** を使用します。これは、関数と呼び出し側コード間で引数を渡すメカニズムです。 EGL 呼び出し関数は、その引数をスタックにプッシュし、呼び出し先の C 関数は、引数をスタックからポップして値を使用します。呼び出し先の関数は、その戻り値をスタックにプッシュし、呼び出し側は、戻り値をスタックからポップして値を取得します。ポップ外部関数および戻り外部関数は、引数スタック・ライブラリーと一緒に提供されます。戻り外部関数については、以下で説明し、使用されるポップ外部関数については『*C の Stack 関数*』で説明します。

外部戻り関数は、その引数を呼び出し側関数の外部に割り当てられたストレージにコピーします。このストレージは、戻り値がポップされると解放されます。この状態では、値を関数のローカル変数から戻すことが可能になります。

注: 戻り関数は、当初 IBM Informix 4GL (I4GL) と一緒に使用されていたので、関数名に「4gl」が含まれています。

値を戻すためのライブラリー関数

値を戻すために使用可能なライブラリー関数は、次のとおりです。

- extern void ibm_lib4gl_returnMInt(int iv)
- extern void ibm_lib4gl_returnInt2(short siv)
- extern void ibm_lib4gl_returnInt4(int lv)
- extern void ibm_lib4gl_returnFloat(float *fv)
- extern void ibm_lib4gl_returnDouble(double *dfv)
- extern void ibm_lib4gl_returnDecimal(dec_t *decv)
- extern void ibm_lib4gl_returnQuotedStr(char *str0)
- extern void ibm_lib4gl_returnString(char *str0)
- extern void ibm_lib4gl_returnVarChar(char *vc)
- extern void ibm_lib4gl_returnDate(int date)
- extern void ibm_lib4gl_returnDateTime(dtime_t *dtv)
- extern void ibm_lib4gl_returnInterval(intrvl_t *inv)
- extern void ibm_lib4gl_returnInt8(ifx_int8_t *bi)

次の表では、I4GL バージョン 7.31 より前のバージョンと、バージョン 7.31 およびそれ以降のバージョンとの間で、戻り関数名をマップしています。

バージョン 7.31 より前の名前	バージョン 7.31 およびそれ以降の名前
retint	ibm_lib4gl_returnMInt
retshort	ibm_lib4gl_returnInt2
retlong	ibm_lib4gl_returnInt4
retflo	ibm_lib4gl_returnFloat
retlub	ibm_lib4gl_returnDouble
retdec	ibm_lib4gl_returnDecimal
retquote	ibm_lib4gl_returnQuotedStr
retstring	ibm_lib4gl_returnString
retvchar	ibm_lib4gl_returnVarChar
retdate	ibm_lib4gl_returnDate
retmtime	ibm_lib4gl_returnDateTime
retinv	ibm_lib4gl_returnInterval

ibm_lib4gl_returnQuotedStr() の引数は、ヌル終了ストリングです。

ibm_lib4gl_returnString() 関数は、対称性のためだけにインクルードされ、内部で **ibm_lib4gl_returnQuotedStr()** を呼び出します。

C 関数は、どの書式でもデータを戻すことができます。変換が可能な場合、EGL は、値をポップするときに必要に応じてデータ型を変換します。データ型の変換が可能でない場合は、エラーが発生します。

EGL から呼び出された C 関数は、常に **return(n)** ステートメントで終了する必要があります。この場合、*n* は、スタック上にプッシュされた戻り値の数です。何も戻さない関数は、**return(0)** で終了する必要があります。

関連する参照項目

- 465 ページの『C の BIGINT 関数』
- 466 ページの『C データ型と EGL プリミティブ型』
- 470 ページの『EGL プログラムからの C 関数の呼び出し』
- 461 ページの『C 関数と EGL との併用』
- 467 ページの『C の DATE 関数』
- 467 ページの『C の DATETIME および INTERVAL 関数』
- 468 ページの『C の DECIMAL 関数』
- 471 ページの『C の Stack 関数』

コメント

EGL ファイルでコメント を作成するには、次の方法のいずれかを使用します。

- 二重スラッシュ (*//*) は、以降の文字が行末文字までコメントであることを示す
- 単一行コメントまたは複数行コメントは、先頭のスラッシュとアスタリスク (*/**) および末尾のアスタリスクとスラッシュ (**/*) によって区切られる。この書式のコメントは、空白文字が有効な場所であればどこでも有効です。

コメントは、以下の例に示すとおり、実行可能文の内側または外側に置くことができます。

```
/* the assignment e = f occurs if a == b or if c == d */
if (a == b           // one comparison
    || /* OR; another comparison */ c == d)
    e = f;
end
```

EGL では組み込みコメントがサポートされていないので、以下のようなエントリーはエラーの原因となります。

```
/* this line starts a comment /* and
   this line ends the comment, */
   but this line is not inside a comment at all */
```

最初の 2 つの行のコメントに、2 番目のコメント区切り (*/**) が含まれています。EGL が 3 番目の行をソース・コードとして解釈しようとした場合にのみエラーになります。

以下は有効です。

```
a = b;  /* this line starts a comment // and
         this line ends the comment */
```

最後の例の二重スラッシュ (*//*) は、それ自体が外側のコメントの一部になっています。

記号 `#sql{` と `}` の間では、前に説明した EGL コメントは無効です。以下の記述が適用されます。

- SQL コメントは、行の先頭から、または空白に続いて二重ハイフン (*--*) で始まり、行の最後まで続けられます。
- 文字列リテラルの内部でコメントは使用できません。文字列リテラル内の一連の文字は、以下のコンテキスト内にあってもテキストとして解釈されません。
 - 準備文

- SQLRecord 型のレコードの **defaultSelectCondition** プロパティ

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

関連する参照項目

532 ページの『EGL ソース形式』

93 ページの『EGL ステートメント』

VisualAge Generator との互換性

EGL は、VisualAge Generator 4.5 に代わるものであり、主に既存のプログラムを新規の開発環境にマイグレーションできるようにするための構文をいくつか組み込んでいます。この構文は、EGL の設定 **VAGCompatibility** が選択されている場合、または (生成時またはデバッグ時に) ビルド記述子オプション **VAGCompatibility** が *yes* に設定されている場合に、開発環境でサポートされます。この設定値は、ビルド記述子オプションのデフォルト値も設定します。

以下の記述は、VisualAge Generator との互換性が有効な場合に適用されます。

- 3 つの無効文字 (- @ #) は ID では有効ですが、ハイフン (-) およびポンド記号 (#) は、いかなる場合も最初の文字としては無効です。『命名規則』を参照してください。
- 指標を指定しないで静的な 1 次元の構造体項目の配列を参照すると、配列指標はデフォルトの 1 になります。詳しくは、『配列』を参照してください。
- 『プリミティブ型』で説明しているように、プリミティブ型の NUMC および PACF は使用可能です。
- プリミティブ型の DECIMAL の項目に偶数の長さを指定している場合は、その項目が SQL ホスト変数として使用される場合を除き、EGL はその長さを 1 だけ増やします。
- 『SQL 項目のプロパティ』で説明しているように、SQL 項目プロパティ **SQLDataCode** を使用できます。
- *call* ステートメントでは、呼び出しオプションのセットを使用できます。
- **externallyDefined** オプションは、*show* および *transfer* ステートメントにあります。
- 以下のシステム変数を使用できます。
 - VGVVar.handleSysLibraryErrors
 - ConverseVar.segmentedMode
- システム関数は以下のとおりです。
 - VGLib.getVAGSysType
 - VGLib.connectionService
- 以下の書式の文を発行できます。

display printForm

printForm

プログラムに対して可視になっている印刷書式の名前。

その場合、**display** は *print* と等価です。

- 以下のプログラム・プロパティは、すべてのケースに使用可能であり、特に VisualAge Generator で作成されたコードに役立ちます。

- **allowUnqualifiedItemReferences**
- **handleHardIOErrors** (*no* に設定された場合)
- **includeReferencedFunctions**
- **localSQLScope** (*yes* に設定された場合)
- **throwNrfEofExceptions** (*yes* に設定された場合)

詳細については、『EGL ソース形式のプログラム・パーツ』を参照してください。

- テキスト書式のプロパティ **value** を設定すると、ユーザーがその書式を戻した後でしか、そのプロパティの内容をプログラムで使用することはできません。この理由から、プログラムに設定する値は、そのプログラム内の項目に対して有効である必要はありません。

VisualAge Generator プログラムを EGL にマイグレーションする方法の詳細については、『EGL に関する追加情報のソース』を参照してください。

関連する概念

13 ページの『EGL に関する追加情報のソース』

関連する参照項目

78 ページの『配列』
 608 ページの『call』
 792 ページの『入力書式』
 792 ページの『入力レコード』
 725 ページの『命名規則』
 739 ページの『pfKeyEquate』
 37 ページの『プリミティブ型』
 681 ページの『print』
 783 ページの『EGL ソース形式のプログラム・パーツ』
 696 ページの『show』
 71 ページの『SQL 項目のプロパティ』
 980 ページの『connectionService()』
 983 ページの『getVAGSysType()』
 1013 ページの『handleSysLibraryErrors』
 990 ページの『segmentedMode』
 697 ページの『transfer』

ConsoleUI

ConsoleField プロパティとフィールド

タイプ ConsoleField の変数では、以下のプロパティが必須です。

- **fieldLen** (ただし、ConsoleField が定数フィールドの場合を除く)
- **position**

定数 `ConsoleField` にはありませんが、**name** フィールドも必須です。

`ConsoleField` のプロパティは次のとおりです。

fieldLen

インタレストの最大値を表示するために必要な桁数を指定します。定数 `consoleFields` では、このプロパティは設定しません。**fieldLen** は、**value** プロパティに含まれている、表示される値が占有する文字数です。

データ型: *INT*

例: *fieldLen = 20*

デフォルト: *none*

position

書式内でのコンソール・フィールドのロケーション。このプロパティには、行番号と列番号を表す 2 つの正整数の配列が含まれます。行番号は書式の先頭から計算されます。同様に、列番号は書式の左端から計算されます。

データ型: *INT[]*

例: *position = [2, 3]*

デフォルト: *[1,1]*

segments

各フィールド・セグメントの行、列、および長さを指定します。フィールド・セグメントは `consoleField` のサブセクションで、区切り文字が使用できます。複数行のテキスト・ボックスの外観を作成するには、同じ書式列の複数行にわたってフィールド・セグメントを 1 つずつスタックします。これにより、セグメントの集合が 1 つのフィールドとして動作するようになります。

データ型: *INT[3][]*

例: *segments = [[5,1,10],[6,1,10]]*

デフォルト: *none*

segments に値を指定した場合、**position** の値は無視されます。また、**fieldLen** には、すべてのセグメントを合わせた長さを設定する必要があります。

複数のセグメントを指定した場合、`ConsoleField` の動作も **lineWrap** フィールドの影響を受けます。

validValues

ユーザー入力に有効な値のリストを指定します。

データ型: 単一および 2 つの値の要素からなる配列リテラル

例: *validValues = [[1,3], 5, 12]*

デフォルト: *none*

詳細については、『*validValues*』を参照してください。

`consoleField` 配列のプロパティには、上記のプロパティ (**segments** を除く) の他、以下のプロパティも含まれます。

columns

`ConsoleField` タイプの配列で、エレメントを表示するために使用する列数を指定

します。たとえば、この配列にエレメントが 5 つあるときに、**columns** プロパティの値が 2 である場合、書式の 1 行目と 2 行目にはエレメントが 2 個ずつ、3 行目には 1 個表示されます。

データ型: *INT*

例: *columns = 3*

デフォルト: *1*

このプロパティは、**ConsoleField** タイプの配列に対してのみ意味を持ちます。画面上での配列エレメントの配置 (横方向か、縦方向か) は、プロパティ **orientIndexAcross** によって決まります。

linesBetweenRows

配列エレメントを含む行と行の間のブランク行の数を指定します。

データ型: *INT*

例: *linesBetweenRows = 3*

デフォルト: *0*

このプロパティは、**ConsoleField** タイプの配列に対してのみ意味を持ちます。

orientIndexAcross

後述の例にあるように、配列エレメントが画面の横方向に並んでいるかどうかを示します。

データ型: *Boolean*

例: *orientIndexAcross = yes*

デフォルト: *yes*

このプロパティは、**consoleField** タイプの配列に対してのみ意味を持ちます。

プロパティ **orientIndexAcross** を *yes* に設定した場合、連続する配列エレメントは左から右に表示されます。次の 2 列の例では、表示される連続エレメントの数値は、エレメントのインデックス番号と同じです。

```
1  2
3  4
5
```

プロパティ **orientIndexAcross** を *no* に設定した場合、連続する配列エレメントは上から下に表示されます。

```
1  4
2  5
3
```

spacesBetweenColumns

フィールドの列と列の間に入るスペースの数を指定します。

データ型: *INT*

例: *spacesBetweenColumns = 3*

デフォルト: *1*

このプロパティは、**consoleField** タイプの配列に対してのみ有効です。

ConsoleField のフィールドは次のとおりです。

align

align フィールドは、データの長さがフィールドの長さよりも短い場合の、変数フィールドにおけるデータの位置を指定します。

データ型: *AlignKind*

例: *align = left*

デフォルト: 文字データまたはタイム・スタンプ・データについては *left*、数値については *right*

実行時に更新されるか? される

このオプションの値は、次のとおりです。

left

フィールドの左側にデータを配置します。開始スペースは除去され、フィールドの末尾に置かれます。

none

データを位置調整しません。この設定は、文字データについてのみ有効です。

right

フィールドの右側にデータを配置します。末尾スペースは除去され、フィールドの先頭に置かれます。

autonext

ユーザーが現在の `ConsoleField` に値を入力した後で、カーソルを次のフィールドに移動するかどうかを表します。

データ型: *Boolean*

例: *autonext = yes*

デフォルト: *None*

実行時に更新されるか? される

『*ConsoleUI* パーツと関連する変数』で説明されているとおり、どの `ConsoleField` が次に来るかはタブ順序により決まります。

binding

デフォルトで、`ConsoleField` がバウンドされている変数名を指定します。

データ型: *String*

例: *binding = "myVar"*

デフォルト: *None*

実行時に更新されるか? されない

バインディングの概要については、『*ConsoleUI* パーツと関連する変数』を参照してください。

caseFormat

大/小文字の区別について、入出力をどのように扱うかを指定します。

データ型: *CaseFormatKind*

例: *caseFormat = lowerCase*

デフォルト: *defaultCase*

実行時に更新されるか? される

このオプションの値は、次のとおりです。

defaultCase (デフォルト)

大/小文字には影響を与えません。

lowerCase

可能であれば、文字を小文字に変換します。

upperCase

可能であれば、文字を大文字に変換します。

color

ConsoleField 内のテキストの色を指定します。

データ型: *ColorKind*

例: *color = red*

デフォルト: *white*

実行時に更新されるか? される。ただし、フィールドの更新後、*ConsoleField* が表示されている (またはフォーカスがある) 場合のみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

defaultColor または white (デフォルト)

白

black

黒

blue

青

cyan

シアン

green

緑

magenta

マゼンタ

red

赤

yellow

黄色

comment

コメント を指定します。これは、カーソルが ConsoleField 内にある場合に、ウィンドウ固有のコメント行 (もしあれば) に表示されるテキストです。

データ型: *String*

例: *"Employee name"*

デフォルト: *Empty string*

実行時に更新されるか? されない

commentKey

コメント を含むリソース・バンドルを検索するために使用されるキーを指定し

ます。コメントは、カーソルが `ConsoleField` 内にある場合に、ウィンドウ固有のコメント行 (もしあれば) に表示されるテキストです。**comment** と **commentKey** の両方を指定した場合は **comment** が使用されます。

データ型: *String*

例: `commentKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? されない

`messageResource` で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

dataType

データ型を示す文字列を指定します。この値は、ユーザー入力 (= 1.5 など) が特定の種類の SQL 列と互換性を持つかどうかを検証するために使用されます。このフィールドは、`ConsoleField` (または関連する `ConsoleForm`) の **openUI** ステートメントに、ステートメント・プロパティ **isConstruct** が含まれている場合のみ意味を持ちます。

データ型: *String*

例: `dataType = "NUMBER"`

デフォルト: *Empty string*

実行時に更新されるか? されない

数値入力に関しては、ユーザーによる浮動小数点値の入力を許可する場合 (この場合、> 1.5 は有効な値) は値 `"NUMBER"` を、それ以外の場合は、整数を表す文字列、たとえば `"INT"` を指定します。

dateFormat

出力の書式をどのように設定するかを指定します。ただし、**dateFormat** は、`ConsoleField` が日付を受け付ける場合のみを指定してください。

データ型: *String* または日付関連のシステム定数

例: `dateFormat = isoDateFormat`

デフォルト: *none*

実行時に更新されるか? されない

有効な値は以下のとおりです。

"pattern"

『日付、時刻、およびタイム・スタンプ・フォーマット指定子』で説明されているように、*pattern* の値は一連の文字から構成されます。

文字は、完全な日付仕様の先頭または末尾から除去することが出来ますが、途中の部分を除去することはできません。

defaultDateFormat

ランタイム Java ロケールで指定された日付形式。

isoDateFormat

「yyyy-MM-dd」パターン (International Standards Organization (ISO) により指定された日付形式)。

usaDateFormat

「MM/dd/yyyy」パターン (IBM USA 標準規格の日付形式)。

eurDateFormat

「dd.MM.yyyy」パターン (IBM 欧州標準規格の日付形式)。

jisDateFormat

「yyyy-MM-dd」パターン (日本工業規格 (JIS) 日付形式)。

systemGregorianCalendar

8 または 10 文字のパターン。このパターンでは、dd (その月の日)、MM (月)、および yy か yyyy (年) が d、M、y または数字以外の文字で区切られています。

この Java ランタイム・プロパティでの書式は次のとおりです。

```
vgj.datemask.gregorian.long.NLS
```

NLS

Java ランタイム・プロパティ **vgj.nls.code** に指定されている NLS (各国語サポート) コード。このコードは、『targetNLS』にリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

systemJulianCalendar

6 または 8 文字のパターン。このパターンでは、DDD (その年、1 月 1 日からの通算日数)、および yy か yyyy (年) が、D、y、または数字以外の文字で区切られています。

この Java ランタイム・プロパティでの書式は次のとおりです。

```
vgj.datemask.julian.long.NLS
```

NLS

Java ランタイム・プロパティ **vgj.nls.code** に指定されている NLS (各国語サポート) コード。このコードは、『targetNLS』にリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

editor

ユーザーとデータとの対話に使用されるプログラムを指定します。ただし、ConsoleField が LOB タイプの変数にバウンドされている場合にのみ、意味を持ちます。

データ型: *String*

例: *editor = "/bin/vi"*

デフォルト: *none*

実行時に更新されるか? される

PATH または LIBPATH にある実行可能ファイルの名前を指定できます。また、この実行可能ファイルの完全修飾パスを指定することもできます。

help

以下の状態に当てはまる場合に表示されるテキストを指定します。

- カーソルが `ConsoleField` にあり、さらに、
- ユーザーが `ConsoleLib.key_help` で指定されたキーを押した。

データ型: *String*

例: `help = "Update the value"`

デフォルト: *Empty string*

実行時に更新されるか? される

helpKey

以下の状態に当てはまる場合に表示されるテキストを含むリソース・バンドルを検索するためのアクセス・キーを指定します。

- カーソルが `ConsoleField` にあり、さらに、
- ユーザーが `ConsoleLib.key_help` で指定されたキーを押した。

help と **helpKey** の両方が指定されている場合、**help** が使用されます。

データ型: *String*

例: `helpKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? される

`messageResource` で説明されているように、リソース・バンドルは、システム変数 `ConsoleLib.messageResource` により識別されます。

highlight

`ConsoleField` を表示するときに使用される特殊効果 (もしあれば) を指定します。

データ型: *HighlightKind[]*

例: `highlight = [reverse, underline]`

デフォルト: *[noHighLight]*

実行時に更新されるか? される。ただし、**highlight** フィールドの更新後、`ConsoleField` が表示されている (またはフォーカスがある) 場合にのみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

noHighlight (デフォルト)

特殊効果を生じさせません。この値は他のどの値よりも優先的に使用されます。

blink

効果はありません。

reverse

テキストおよび背景色を反転します。たとえば、ディスプレイで黒い背景に白い文字が表示されている場合、背景が黒くなり、テキストが白になります。

アンダーライン

影響を受けるエリアの下部に下線を配置します。値 **reverse** が指定されている場合でも、下線の色はテキストの色になります。

initialValue

表示の初期値を指定します。

データ型: *String*

例: `initialValue = "200"`

デフォルト: *Empty string*

実行時に更新されるか? される

openUI ステートメントの **setInitial** プロパティが **true** に設定されている場合、**consoleField** にある **initialValue** プロパティの値が使用されます。ただし、**openUI** プロパティが **false** である場合、代わりにバウンド変数の現行値が表示され、**initialValue** プロパティの値は無視されます。

initialValueKey

表示に使用される初期値を含むリソース・バンドルを検索するためのアクセス・キーを指定します。**initialValue** と **initialValueKey** の両方を指定した場合、**initialValue** が使用されます。

データ型: *String*

例: `initialValueKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? される

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

inputRequired

値を入力しなければ、現在のフィールドから別の個所にユーザーがナビゲートできないようにするかどうかを表します。

initialValue と **initialValueKey** の両方を指定した場合、**initialValue** が使用されます。

データ型: *String*

例: `initialValueKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? されない

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

intensity

表示フォントを強調する度合いを指定します。

データ型: *IntensityKind[]*

例: `intensity = [bold]`

デフォルト: *[normalIntensity]*

実行時に更新されるか? される。ただし、*intensity* フィールドの更新後、*ConsoleField* が表示されている (またはフォーカスがある) 場合にのみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

normalIntensity (デフォルト)

特殊効果を生じさせません。この値は他のどの値よりも優先的に使用されます。

bold

テキストを太字フォントで表示します。

dim

現在のところ、効果はありません。将来、入力フィールドが無効化されている場合、または強調すべきではない場合に、適宜、テキストの輝度を低くして表示できるようになる可能性があります。

invisible

書式上にフィールドがあることを示さないようにします。

isBoolean

ConsoleField がブール値を表しているかどうかを示します。フィールド **isBoolean** は有効な *ConsoleField* 値を制限し、入力または出力に便利です。

数値フィールドの値は 0 (false) または 1 (true)。

文字フィールドの値は、各国語に依存した語または語のサブセットにより示されます。また、特定の値はロケールにより決定されます。例えば英語では、3 つ以上の文字のブール値フィールドは、値 *yes* (true) または *no* (false) をとり、1 文字のブール値フィールドは *y* または *n* のように値が切り捨てられます。

データ型: *Boolean*

例: *isBoolean* = *yes*

デフォルト: *no*

実行時に更新されるか? されない

lineWrap

テキストの切り捨てを回避するために、ラッピングが必要になるたびにテキストを新しい行に折り返す方法を表します。

データ型: *LineWrapType*

例: *value* = *compress*

デフォルト: *character*

実行時に更新されるか? される

このオプションの値は、次のとおりです。

character (デフォルト)

フィールド内のテキストはホワイト・スペースで分割するのではなく、フィールド・セグメントの境界線のある文字の位置で分割します。

compress

可能な場合、テキストはホワイト・スペースの位置で分割されます。ユーザーが (別の *consoleField* にナビゲートするか、または **Esc** キーを押すこと

により) `consoleField` を離れると、この値はバウンド変数に割り当てられ、テキストの折り返しに使用された余分なスペースはすべて除去されます。

word

可能な場合、フィールド内のテキストはホワイト・スペースで分割されます。値がバウンド変数に割り当てられると、ワード境界で折り返すために値がどのようにパッドされたのかを反映して、スペースが追加されます。

segments プロパティーで制御されているので、**lineWrap** フィールドは複数セグメントを持つ `ConsoleField` に対してのみ有効です。

masked

ユーザーがパスワードを入力するときに、`ConsoleField` 内の各文字がアスタリスク (*) として表示されるかどうかを表します。

データ型: *Boolean*

例: *masked = yes*

デフォルト: *no*

実行時に更新されるか? される

minimumInput

有効な入力の最小文字数を表します。

データ型: *INT*

例: *minimumInput = 4*

デフォルト: *no*

実行時に更新されるか? されない

name

実行時に名前が解決されるコンテキストのプログラミングで使用する `ConsoleField` 名。name フィールドの値は、変数名と同じにすることを特にお勧めします。

データ型: *String*

例: *name = "myField"*

デフォルト: *none*

実行時に更新されるか? されない

numericFormat

出力に対してどのように書式を設定するかを指定します。ただし、`ConsoleField` が数値を受け付ける場合のみ、**numericFormat** を指定します。

データ型: *String*

例: *numericFormat = "-###@"*

デフォルト: *none*

実行時に更新されるか? されない

有効な文字は以下のとおりです。

数字のプレースホルダー。

* 先行ゼロ用の充てん文字文字としてアスタリスク (*) を使用します。

& 先行ゼロ用の充てん文字文字としてゼロを使用します。

- # 先行ゼロ用の充てん文字としてスペースを使用します。
- < 数値を左そろえにします。
- , 位置に先行ゼロが含まれていない限り、ロケールに依存する数字分離記号を使用します。
- . ロケールに依存する小数点を使用します。
- 0 未満の値には負符号 (-) を使用します。0 以上の値にはスペースを使用します。
- + 0 未満の値には負符号 (-) を使用します。0 以上の値には正符号 (+) を使用します。
- (会計処理などで、負の値の前に左括弧を表示します。
-) 会計処理などで、負の値の後ろに右括弧を表示します。
- \$ 値の前に、ロケールに依存する通貨記号を付けます。
- @ 値の後ろに、ロケールに依存する通貨記号を付けます。

pattern

ConsoleField のコンテンツのタイプが文字である場合に、入出力フォーマットのパターンを指定します。

データ型: *String*

例: `pattern = "(###) ###-####"`

デフォルト: *none*

実行時に更新されるか? されない

以下の制御文字が使用可能です。

- A は文字のプレースホルダーです。どのようなサブセットが文字とみなされるかは、ロケールに依存します。
- # は数値のプレースホルダーです。
- X は必要とされる任意の種類の文字のプレースホルダーです。

上記 3 種類以外の文字は入力、または出力に含まれます。ただし、出力ではオーバーレイ文字はすべて失われます。

- 出力パターンが "(###) ###-####" である場合、値 "6219655561212" は次のように表示されます。

(219) 555-1212

ユーザーは、オリジナルの値に含まれる 6 を使用することはできません。また、データ・ストアが更新にされるとこれらの 6 は失われます。

- 入力の場合、カーソルはリテラル文字をスキップし、プレースホルダー文字があるところのみ入力を許可します。この例では、ユーザーが「2195551212」と入力すると、文字列 "(219) 555-1212" が ConsoleField の値となり、このバウンド変数に入れられます。

protect

ConsoleField がユーザーによる更新から保護されているかどうかを表します。

データ型: *Boolean*

例: `protect = yes`

デフォルト: `no`

実行時に更新されるか? されない

このオプションの値は、次のとおりです。

No (デフォルト)

ユーザーがフィールドの値を上書きできるよう設定します。

Yes

ユーザーが `consoleField` の値を上書きできないよう設定します。また、次のような場合には、ユーザーが `consoleField` にナビゲートしようとしても、カーソルはこのフィールドをスキップします。

- ユーザーが、タブ順序の直前の `consoleField` を操作しているときに、(a) **Tab** キーを押した、または、(b) **autonext** フィールドが `yes` に設定されているときに、直前の `consoleField` に値を入力しようとした。
- ユーザーがタブ順序で次に来る `consoleField` を操作しているときに **Shift + Tab** キーを押した。
- ユーザーが矢印キーを使って、直前、または次の `consoleField` に移動した。

保護されている、または保護されていない `consoleField` に変数をバインドすることができます。`openUI` プロパティ **setInitial** の設定により、バウンド変数の値が表示されるかどうかが決まります。

プログラムが、保護されている `consoleField` に移動しようとする、ランタイム・エラーが発生します。

SQLColumnName

`ConsoleField` に関連付けられているデータベース表の列の名前を指定します。この名前は、`ConsoleField` (または関連する `ConsoleForm`) の `openUI` ステートメントに、ステートメント・プロパティ **isConstruct** が含まれている場合のみ、検索条件の作成に使用されます。

データ型: `String`

例: `SQLColumnName = "ID"`

デフォルト: `none`

実行時に更新されるか? される

timeFormat

出力に対してどのように書式を設定するかを指定します。ただし、`ConsoleField` が時刻を受け付ける場合のみ、**timeFormat** を指定します。

データ型: `String` または時刻関連のシステム定数

例: `timeFormat = isoTimeFormat`

デフォルト: `none`

実行時に更新されるか? されない

有効な値は以下のとおりです。

"pattern"

『日付、時刻、およびタイム・スタンプ・フォーマット指定子』で説明されているとおり、*pattern* の値は一連の文字から構成されます。

文字は、完全な時刻仕様の先頭または末尾から除去することが出来ますが、途中の部分を除去することはできません。

defaultTimeFormat

ランタイム Java ロケールで指定された時刻形式。

isoTimeFormat

「HH:mm:ss」パターン (International Standards Organization (ISO)) により指定された時刻形式)

usaTimeFormat

「hh:mm AM」パターン (IBM USA 標準規格の時刻形式)。

eurTimeFormat

「HH:mm:ss」パターン (IBM 欧州標準規格の時刻形式)。

jisTimeFormat

「HH:mm:ss」パターン (日本工業規格の時刻形式)。

timestampFormat

出力の書式をどのように設定するのかを指定します。ただし、**timestampFormat** は、**ConsoleField** がタイム・スタンプを受け付ける場合にのみ指定します。

データ型: *String* またはタイム・スタンプ関連のシステム定数

例: *timestampFormat = jdbcTimestampFormat*

デフォルト: *none*

実行時に更新されるか? されない

有効な値は以下のとおりです。

"pattern"

『日付、時刻、およびタイム・スタンプ・フォーマット指定子』で説明されているとおり、*pattern* の値は一連の文字から構成されます。

文字は、完全なタイム・スタンプ仕様の先頭または末尾から除去することが出来ますが、途中の部分を除去することはできません。

defaultTimestampFormat

ランタイム Java ロケールで指定されたタイム・スタンプ形式。

db2TimeStampFormat

「yyyy-MM-dd-HH:mm:ss.ffffff」パターン (IBM DB2 のデフォルトのタイム・スタンプ・フォーマット)。

odbcTimeStampFormat

「yyyy-MM-dd HH:mm:ss.ffffff」パターン (ODBC のタイム・スタンプ・フォーマット)。

value

consoleField に表示される現行値。**ConsoleLib.displayForm** を呼び出して、**consoleField** に指定値を表示するように、コードでこの値を設定できます。

データ型: *String*

例: `value = "View"`

デフォルト: `none`

実行時に更新されるか? される

verify

ConsoleField から別の個所に移動しようとした後で、ユーザーに対して、同じ値の再入力を求めるプロンプトを表示するかどうかを指定します。

データ型: `String`

例: `value = "View"`

デフォルト: `none`

実行時に更新されるか? される

このオプションの値は、次のとおりです。

No (デフォルト)

EGL ランタイムは、特殊なプロンプトは発行しません。

Yes

ユーザーが ConsoleField を離れようとする、EGL ランタイムは次のように動作します。

- カーソルをそのままにしておいて、`consoleField` をクリアします。
- ユーザーに再入力を促すメッセージを表示します。
- ユーザーがもう一度、`consoleField` から移動しようとしたときに、2 つの入力値を比較します。

値が一致した場合は、この値がバウンド変数によって受信され、処理は通常通り継続されます。値が一致しなかった場合は、2 回のユーザー入力の 1 回目の前に入力されていた値に `consoleField` 定数が戻され、カーソルはこのフィールドに表示されたままになります。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

584 ページの『Java ランタイム・プロパティー (詳細)』

669 ページの『openUI』

776 ページの『validValues』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL consoleUI の ConsoleForm プロパティー

ConsoleForm タイプのレコード・パーツのプロパティーは次のとおりです。このうち、`formSize` のみが必須プロパティーです。

delimiters

入力フィールドの前後に表示される文字を指定します。これらの文字は、プロパティー `showBrackets` の値が `yes` である場合のみ表示されます。

データ型: *String literal*

例: `delimiters = "<>/"`

デフォルト: `"[]"`

可能であれば、1 番目の文字はそれぞれの非定数 `ConsoleField` の前に、2 番目の文字はその後に表示されます。ただし、3 番目の文字は、単一位置で区切られた 2 つの非定数 `ConsoleField` の間に表示されます。

指定した文字数が 3 文字未満の場合、指定されていない文字については、デフォルト文字が使用されます。4 つ以上の文字を指定した場合、4 番目以降の文字は無視されます。

formSize

書式のディメンション。このフィールドには、行数と列数を表す 2 つの正整数の配列が含まれます。

データ型: *INT[2]*

例: `size = [24, 80]`

デフォルト: *none*

ディメンションのどちらかが、書式が表示されるウィンドウのサイズを超える場合、ウィンドウのディメンションに収まるように書式サイズが縮小されます。ただし、`ConsoleField` がウィンドウに収まらない場合はプログラムは終了します。

name

実行時に名前が解決されるコンテキストのプログラミングで使用する `Form` 名。`name` フィールドがある場合には、このフィールドの値は、変数名と同じにすることをお勧めします。

データ型: *String*

例: `name = "myForm"`

デフォルト: *none*

`name` フィールドは、**`ConsoleLib.displayFormByName`** などのシステム関数で使用されます。

showBrackets

非定数 `ConsoleFields` が、大括弧のような 1 組の区切り文字で区切られるかどうかを表します。

データ型: *Boolean*

例: `showBrackets = no`

デフォルト: *yes*

その他の詳細については、プロパティ **`delimiters`** を参照してください。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『`ConsoleUI` パーツおよび関連変数』

669 ページの『`openUI`』

関連タスク

190 ページの『`consoleUI` を使用したインターフェースの作成』

EGL consoleUI の Menu フィールド

タイプ `Menu` の変数にあるフィールドの定義は以下のとおりです。フィールド `labelText` または `labelTextKey` は、指定する必要があります。

`labelText`

`menuItems` リストの左側に表示されるラベル。

データ型: *String literal*

例: `labelText = "Options: "`

デフォルト: *none*

実行時に更新されるか? されない

`labelKey`

メニュー・ラベルを含むリソース・バンドルを検索するためのキーを指定します。`labelText` と `labelKey` の両方を指定した場合、`labelText` が使用されます。

データ型: *String*

例: `labelKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? されない

`messageResource` で説明されているとおり、リソース・バンドルは、システム変数 `ConsoleLib.messageResource` により識別されます。

`menuItems`

メニュー項目の配列。各メニュー項目はプログラムで宣言されるか、またはキーワード `new` を使って動的に作成されます。2 番目のオプションの詳細については、『`consoleUI` での `new` の使用』を参照してください。

データ型: *MenuItem[]*

例: `menuItems = [myItem, new MenuItem {name = "Remove", labelText = "Delete all"}].`

デフォルト: *none*

実行時に更新されるか? されない

プログラムに `menuItem` を追加するときは、以下の構文を使用します。

```
myMenu.MenuItems.addElement(myMenuItem)
```

`myMenu`

`Menu` タイプの変数の名前。

`myMenuItem`

`MenuItem` タイプの変数の名前。

`menuItems` が存在しないメニューに対して `openUI` ステートメントを実行すると、プログラムは終了します。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

78 ページの『配列』

192 ページの『ConsoleUI パーツおよび関連変数』

669 ページの『openUI』

『EGL consoleUI の MenuItem フィールド』

195 ページの『ConsoleUI での new の使用』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL consoleUI の MenuItem フィールド

タイプ MenuItem の変数にある consoleFields フィールドの定義は以下のとおりです。必須の consoleFields はありません。ユーザーの選択は、3 つのフィールド **accelerators**、**labelText**、または **labelKey** のいずれの設定により決まります。

accelerators

ユーザーによる menuItem の選択と等しいキー・ストロークを表します。これらの各キー・ストロークによって、menuItem 選択に対応する **openUI** ステートメントの OnEvent 文節が実行されます。

データ型: *String[]*

例: *accelerators = ["F1", "ALT_F1"]*

デフォルト: *none*

実行時に更新されるか? されない

comment

コメント を指定します。これは、menuItem が選択されているときに、menuItem 専用のコメント行に表示されるテキストです。

データ型: *String*

例: *"Delete the record"*

デフォルト: *Empty string*

実行時に更新されるか? される

コメント行は、メニュー行の 1 行下に表示されます。

commentKey

コメント を含むリソース・バンドルを検索するために使用されるキーを指定します。コメントは、menuItem が選択されているときに、menuItem 専用コメント行 (もしあれば) に表示されるテキストです。**comment** と **commentKey** の両方を指定した場合は **comment** が使用されます。

データ型: *String*

例: *commentKey = "myKey"*

デフォルト: *Empty string*

実行時に更新されるか? される

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

help

以下の状態に当てはまる場合に表示されるテキストを指定します。

- menuItem が選択されていて、さらに、

- ユーザーが **ConsoleLib.key_help** で指定されたキーを押した。

データ型: *String*

例: `help = "Deletion is permanent"`

デフォルト: *Empty string*

実行時に更新されるか? される

helpKey

以下の状態に当てはまる場合に表示されるテキストを含むリソース・バンドルを検索するためのアクセス・キーを指定します。

- menuItem が選択されていて、さらに、
- ユーザーが **ConsoleLib.key_help** で指定されたキーを押した。

help と **helpKey** の両方が指定されている場合、**help** が使用されます。

データ型: *String*

例: `helpKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? される

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

labelText

menuItem を表すラベル。

データ型: *String literal*

例: `labelText = "Delete".`

デフォルト: *none*

実行時に更新されるか? されない

labelKey

menuItem ラベルを含むリソース・バンドルを検索するためのキーを指定します。**labelText** と **labelKey** の両方を指定した場合、**labelText** が使用されます。

データ型: *String*

例: `labelKey = "myKey"`

デフォルト: *Empty string*

実行時に更新されるか? されない

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

name

実行時に名前が解決されるコンテキストのプログラミングで使用される menuItem 名。この名前は、特に、menuItem の選択に応答する **openUI** ステートメントで使用されます。

name フィールドの値は、変数名と同じにすることをお勧めします。

データ型: *String*

例: `name = "myItem"`

デフォルト: *none*

実行時に更新されるか? されない

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

494 ページの『EGL consoleUI の Menu フィールド』

669 ページの『openUI』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL consoleUI の PresentationAttributes フィールド

以下のリストは、PresentationAttributes タイプのいかなるシステム変数でも設定、または取得できるフィールドを定義したものです。

color

色を指定します。

データ型: *ColorKind*

例: *color = red*

デフォルト: *white*

実行時に更新されるか? される。ただし、*color* フィールドの更新後に出力が表示されている場合のみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

defaultColor または **white** (デフォルト)

白

black

黒

blue

青

cyan

シアン

green

緑

magenta

マゼンタ

red

赤

yellow

黄色

highlight

出力を表示するときに使用される特殊効果 (もしあれば) を指定します。

データ型: *HighlightKind[]*

例: *highlight* = [*reverse*, *underline*]

デフォルト: [*noHighLight*]

実行時に更新されるか? される。ただし、*highlight* フィールドの更新後に出力が表示されている場合のみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

noHighlight (デフォルト)

特殊効果を生じさせません。この値は他のどの値よりも優先的に使用されます。

blink

現在のところ、効果はありません。

reverse

テキストおよび背景色を反転します。たとえば、ディスプレイで黒い背景に白い文字が表示されている場合、背景が黒くなり、テキストが白になります。

アンダーライン

影響を受けるエリアの下部に下線を配置します。値 **reverse** が指定されている場合でも、下線の色はテキストの色になります。

intensity

表示フォントを強調する度合いを指定します。

データ型: *IntensityKind*[]

例: *intensity* = [*bold*]

デフォルト: [*normalIntensity*]

実行時に更新されるか? される。ただし、*intensity* フィールドの更新後に出力が表示されている場合のみ、更新を目で見て確認することができます。

このオプションの値は、次のとおりです。

normalIntensity (デフォルト)

特殊効果を生じさせません。この値は他のどの値よりも優先的に使用されます。

bold

テキストを太字フォントで表示します。

dim

現在のところ、効果はありません。将来、すべての入力フィールドが無効化されている場合、適宜、テキストの輝度を低くして表示できるようになる可能性があります。

invisible

書式上にテキストが存在する兆候をすべて除去します。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

825 ページの『*currentDisplayAttrs*』

825 ページの『*currentRowAttrs*』

826 ページの『defaultDisplayAttributes』
826 ページの『defaultInputAttributes』
192 ページの『ConsoleUI パーツおよび関連変数』
669 ページの『openUI』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL consoleUI の Prompt フィールド

タイプ Prompt の変数にあるフィールドの定義は以下のとおりです。これらのフィールドのうち、必須のフィールドはありません。

isChar

プロンプトの表示後、ユーザーが最初に行ったキー・ストロークで操作が終了するかどうかを表します。

データ型: *Boolean*

例: *isChar = yes*

デフォルト: *no*

実行時に更新されるか? される

このオプションの値は、次のとおりです。

no (デフォルト)

操作は、ユーザーが **Enter** キーを押すか、プロンプトを表示する **openUI** ステートメントの **OnEvent** 文節に関連付けられているキーを押すかしたときに終了します。プロンプトがバウンドされている変数は、入力された文字を受け取ります。

yes

ユーザーの最初のキー・ストロークにより操作が終了します。この文字が印刷可能である場合、このプロンプトがバウンドされている変数は、この文字を受け取ります。

どちらの場合も、タイプ **ON_KEY** の **OnEvent** 文節を設定することにより、特定のキー・ストロークに応答することができます。

message

ユーザーに対して表示されるプロンプトのテキストを指定します。

データ型: *String*

例: *message = "Type here: "*

デフォルト: *Empty string*

実行時に更新されるか? コードが **openUI** ステートメントを発行する前に更新される

messageKey

プロンプト・テキストを含むリソース・バンドルを検索するために使用されるキーを指定します。**message** と **messageKey** の両方を指定した場合、**message** が使用されます。

データ型: *String*

例: *messageKey = "promptText"*

デフォルト: *Empty string*

実行時に更新されるか? される

messageResource で説明されているとおり、リソース・バンドルは、システム変数 **ConsoleLib.messageResource** により識別されます。

responseAttr

ユーザー入力の表示に使用されるプレゼンテーション属性を指定します。

データ型: *PresentationAttributes literal*

例: *responseAttr {color = green, highlight = [underline], intensity = [bold]}*

デフォルト: *no*

実行時に更新されるか? される

このフィールドは、**isChar** フィールドが *no* に設定されている場合にのみ、効果があります。

responseAttr 値の詳細については、『*EGL consoleUI の PresentationAttributes フィールド*』を参照してください。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

584 ページの『Java ランタイム・プロパティ (詳細)』

840 ページの『messageResource』

669 ページの『openUI』

497 ページの『EGL consoleUI の PresentationAttributes フィールド』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

EGL consoleUI の Window フィールド

以下のリストでは、タイプ Window の変数におけるフィールドが定義されています。どのフィールドも必須ではありませんが、実際には**サイズ**は必要です。

color

ウィンドウで次の種類の出力を表示する際に使用される色を指定します。

- **consoleForms** のラベル
- プロンプトの入力フィールド
- ウィンドウ・ボーダー
- **ConsoleLib.displayAtPosition** などのシステム関数の出力

データ型: *ColorKind*

例: *color = red*

デフォルト: *white*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にウィンドウを開いた場合にのみ、更新を目で見て確認できます。

このオプションの値は、次のとおりです。

defaultColor または **white** (デフォルト)

白

black

黒

blue

青

cyan

シアン

green

緑

magenta

マゼンタ

red

赤

yellow

黄

commentLine

Window フィールド **hasCommentLine** が *yes* に設定されている場合にコメント (存在する場合) が表示される行数を設定します。行数は、画面ウィンドウのコンテンツ領域の上部から (この場合、最初の行は 1)、または (値が負の場合) 同領域の下部から計算されます (この場合、最後の行が -1、最後から 2 番目の行が -2、以下同様)。

データ型: *INT*

例: *commentLine = 10*

デフォルト: ウィンドウの最後の行 (画面ウィンドウがオープン状態でも、コメントはウィンドウの最後から 2 番目の行にあります)

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新を目で見て確認できます。

値の妥当性は実行時にのみ決定されます。

formLine

フォームが表示される行数を設定します。行数は、画面ウィンドウのコンテンツ領域の上部から (この場合、最初の行は 1)、または (値が負の場合) 同領域の下部から計算されます (この場合、最後の行が -1、最後から 2 番目の行が -2、以下同様)。

データ型: *INT*

例: *formLine = 8*

デフォルト: 3

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にウィンドウが表示された場合にのみ猛進に視覚効果があります。

値の妥当性は実行時にのみ決定されます。

hasBorder

ウィンドウがボーダーによって囲まれているかどうかを示します。値が「yes」の場合、ボーダーの色は Window フィールド *color* で指定されます。

データ型: *Boolean*

例: *hasBorder = yes*

デフォルト: *no*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

hasCommentLine

カーソルが *consoleField* に置かれたときに表示されるテキスト・エントリーである *comments* のために、ウィンドウが行を予約するかどうかを示します。値が「yes」の場合、行数は Window フィールド *commentLine* で指定されます。

データ型: *Boolean*

例: *hasCommentLine = yes*

デフォルト: *no*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

highlight

ウィンドウで次の種類の出力を表示する際に使用される特殊効果（使用可能な場合）を指定します。

- *consoleForms* のラベル
- プロンプトの入力フィールド
- ウィンドウ・ボーダー
- **ConsoleLib.displayAtPosition** などのシステム関数の出力

データ型: *HighlightKind[]*

例: *highlight = [reverse, underline]*

デフォルト: *[noHighLight]*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にウィンドウが表示された場合にのみ猛進に視覚効果があります。

このオプションの値は、次のとおりです。

noHighlight (デフォルト)

特殊効果は実行されません。この値を使用すると他のすべての値がオーバーライドされます。

blink

現時点で効果はありません。

reverse

テキストおよび背景色を反転し、（例えば）黒色の背景色に白色文字を表示した場合に背景色を黒くしてテキストを白色にします。

アンダーライン

対象となる領域の下にアンダーラインを配置します。アンダーラインの色は、値 **Reverse** も指定されているため、テキストの色が逆の場合でも、テキストの色となります。

intensity

ウィンドウで次の種類の出力を表示する際に使用される表示フォントの強度を指定します。

- consoleForms のラベル
- プロンプトの入力フィールド
- ウィンドウ・ボーダー
- **ConsoleLib.displayAtPosition** などのシステム関数の出力

データ型: *IntensityKind[]*

例: *intensity = [bold]*

デフォルト: *[normalIntensity]*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

このオプションの値は、次のとおりです。

normalIntensity (デフォルト)

特殊効果は実行されません。この値を使用すると他のすべての値がオーバーライドされます。

bold

テキストを太字フォントで表示します。

dim

現時点で効果はありません。将来的に、入力フィールドが無効の場合に、必要に応じて、低輝度でテキストが表示されるようになる可能性があります。

invisible

書式上にフィールドがあることを示さないようにします。

menuLine

Window に使用可能なメニューが表示される際の行数を設定します。行数は、画面ウィンドウのコンテンツ領域の上部から (この場合、最初の行は 1)、または (値が負の場合) 同領域の下部から計算されます (この場合、最後の行が -1、最後から 2 番目の行が -2、以下同様)。

データ型: *INT*

例: *menuLine = 2*

デフォルト: *1*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

値の妥当性は実行時にのみ決定されます。

messageLine

Window に使用可能メッセージが表示される際の行数を設定します。行数は、画面ウィンドウのコンテンツ領域の上部から (この場合、最初の行は 1)、または (値が負の場合) 同領域の下部から計算されます (この場合、最後の行が -1、最後から 2 番目の行が -2、以下同様)。

データ型: *INT*

例: *messageLine = 3*

デフォルト: *2*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

値の妥当性は実行時にのみ決定されます。

name

名前が実行時に解決される際にプログラミング・コンテキストで使用されるウィンドウ名。できるだけ名前フィールドの値と変数の名前が同じになるようにすることをお勧めします。

データ型: *String*

例: *name = "myWindow"*

デフォルト: *none*

実行時に更新可能か? *No*

position

画面ウィンドウのコンテンツ領域内のウィンドウの左上部コーナーの位置。フィールドには 2 つの整数の配列が含まれます。行数は列番号の後に配置されます。行数は、画面ウィンドウのコンテンツ領域の上部から (この場合、最初の行は 1)、または (値が負の場合) 同領域の下部から計算されます (この場合、最後の行が -1、最後から 2 番目の行が -2、以下同様)。行数は、コンソール・ウィンドウのコンテンツ領域の左から計算され、最初の列が 1 となります。

データ型: *INT[2]*

例: *position = [2, 3]*

デフォルト: *[1,1]*

実行時に更新可能か? *No*

promptLine

Window に使用可能なプロンプトが表示される際の行数を設定します。行数は、コンソール・ウィンドウのコンテンツ領域の上部から、または (値が負の場合) 同領域の下部から計算されます。

データ型: *INT*

例: *promptLine = 4*

デフォルト: *1*

実行時に更新可能か? 「Yes」。ただし、フィールドが更新された後にフィールドを開いた場合にのみ更新に視覚効果があります。

値の妥当性は実行時にのみ決定されます。

size

ウィンドウ・ディメンションを表す 2 つの正整数の配列。行数は列数の後に配置されます。

データ型: *INT[2]*

例: *size = [24, 80]*

デフォルト: *none*

実行時に更新可能か? *No*

値は実用的な目的では必須です。**size** の値が欠落しているウィンドウを表示すると、実行時に、コンテンツを表示しきれない小さなウィンドウが現れてしまいます。

ディメンションが画面ウィンドウのコンテンツ領域で有効なサイズを越えている場合、実行時にエラーが発生します。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

669 ページの『openUI』

関連タスク

190 ページの『consoleUI を使用したインターフェースの作成』

containerContextDependent

関数パーツのプロパティ **containerContextDependent** を指定すると、このプロパティを含む関数パーツ内からの関数参照を解決するために使用される名前空間を拡張することができます。有効な値は *no* (デフォルト) と *yes* です。

新規コードを作成する場合は、この機能を使用されないことをお勧めします。このプロパティは主に、VisualAge Generator からのプログラムのマイグレーションに使用できます。ただし、このプロパティを *yes* に設定すると、以下のような影響が出ます。

- 名前検索の通常のステップで、編集時に参照が解決されなくても、EGL エディターは、解決されない参照をエラーとしてフラグを立てません。
- 名前検索の通常のステップで、生成時に参照が解決されない場合は、その関数パーツを含んでいるプログラム、ライブラリー、またはページ・ハンドラーの名前空間を調べることによって検索が続行されます。
- 関数をコンテナ (プログラム、ページ・ハンドラー、またはライブラリー) の物理的な内部でなく、EGL ソース・ファイルのトップレベルで宣言した場合、その関数がライブラリー関数を呼び出すことができるのは、以下の状態の場合だけです。
 - コンテナに、ライブラリーを参照する *use* ステートメントが含まれている
 - 呼び出し側関数内で、**containerContextDependent** プロパティが *yes* に設定されている

関連する概念

24 ページの『パーツの参照』

関連する参照項目

570 ページの『EGL ソース形式の関数パーツ』

] 1020 ページの『使用宣言』

データベースの権限とテーブル名

許可 ID は、データベース・マネージャーとプログラム間の接続が確立したときに、データベース・マネージャーに渡される文字ストリングです。許可 ID は、プログラムが別のプログラムを準備するかどうかや、エンド・ユーザーが SQL テーブルにアクセスできるかどうかに関係なく、データベース・マネージャーに渡されます。この文字ストリングは、準備するプログラムまたはエンド・ユーザーが保持しているデータベース・アクセス許可を検査する際に必要となるユーザー ID です。

許可 ID のソースは、データベース・アクセスが行われるシステムごとに異なります。

EGL 生成 Java プログラムの状況を次に示します。

- 許可 ID は、データベース・マネージャーとプログラム間の接続が確立したときに、データベース・マネージャーが受け取る ID です。
 - デフォルト・データベースに対する許可 ID は、Java ランタイム・プロパティ `vgj.jdbc.default.userid` に指定された値です。
 - システム関数 `sysLib.connect` または `VGLib.connectionService` を呼び出す場合、許可 ID は、`userID` パラメーターに指定された値です。

許可 ID は、テーブル名を指定すると使用できます。その場合、テーブル名修飾子は、次の構文に従って指定します。

`tableOwner.myTable`

tableOwner

データベース・マネージャーが認識する修飾子です。テーブルを識別する際に必要となります。テーブル作成時の修飾子は、テーブル作成者の許可 ID です。

myTable

テーブル名を指定します。

許可 ID の詳細については、データベース・マネージャーの資料を参照してください。

関連する概念

259 ページの『動的 SQL』

377 ページの『Java ランタイム・プロパティ』

247 ページの『SQL サポート』

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

957 ページの『`connect()`』

980 ページの『`connectionService()`』

データ変換

ランタイム環境が異なるとデータを解釈する方法にも差が生じるので、ある環境から別の環境に渡すデータは、プログラム上での変換が必要な場合があります。データ変換が実行されるタイミングは、Java の実行時です。

実行時の状態が次のような場合にも、コード上で変換テーブルが使用されます。

- EGL 生成の Java コードが、CICS for z/OS 上でプログラムを呼び出す場合。

この場合は、呼び出し先プログラムを参照する `callLink` エlementの変換テーブルを指定できます。または、(`callLink` Elementの中で) システム変数 `sysVar.callConversionTable` が実行時に変換テーブルを識別するよう指示することもできます。

- EGL 生成のプログラム (EBCDIC 文字セットをサポートするプラットフォーム上の) が、ASCII 文字セットをサポートするプラットフォーム上のプログラムに非同期に転送する場合。これは、転送側プログラムがシステム関数 `sysLib.startTransaction` を呼び出すときに発生する場合があります。

この場合は、制御の転送先のプログラムを参照する `asynchLink` Elementの変換テーブルを指定できます。または、(`asynchLink` Elementの中で) システム変数 `sysVar.callConversionTable` が実行時に変換テーブルを識別するよう指示することもできます。

- EGL 生成の Java プログラムが、一連のアラビア語またはヘブライ語の文字が含まれるテキストまたは印刷書式を表示するか、これらの文字のユーザーからの入力を受け入れるテキスト書式を表示する場合。

この場合は、システム変数 `sysVar.formConversionTable` で双方向変換テーブルを指定します。

例えば、コードが 2 つの再定義レコードのいずれかに値を割り当てるときに、それぞれが、別のプログラムに渡されるレコードとして、同じメモリー領域を参照する場合に、実行時変換を使用します。値を割り当てる再定義レコードに応じて、渡すデータの特性が異なると想定します。この場合、データ変換の要件は、生成時には不明です。

これ以降は、次に示す項目の詳細を説明します。

-
- 『起動側が Java コードの場合のデータ変換』
- 509 ページの『変換アルゴリズム』

起動側が Java コードの場合のデータ変換

以下の規則は Java コードに適用されます。

- 生成されたコードに適用されます。Java プログラムまたはラッパーが、生成された Java プログラムを呼び出すと、実行時に呼び出される 1 組の EGL クラスに従って、呼び出し側で変換が実行されます。起動側が使用しているのとは異なるコード・ページを使用するリモート・プラットフォームに呼び出し側がアクセスする場合でも、ほとんどの場合は、変換を要求する必要がありません。ただし、次のような場合は、変換テーブルを指定する必要があります。

- 呼び出し側が Java コードであり、1 つのコード・ページをサポートするマシン上で稼働している。
- 呼び出し先プログラムが非 Java であり、別のコード・ページをサポートするマシン上で稼働している。

この場合のテーブル名は、実行時に必要な種類の変換を表すシンボルになります。

- 生成された Java プログラムがリモートの MQSeries メッセージ・キューにアクセスすると、実行時に呼び出される一連の EGL クラスに従って、起動側で変換が実行されます。起動側が使用しているのとは異なるコード・ページを使用するリモート・プラットフォームに呼び出し側がアクセスする場合は、『関連要素』に示す、MQSeries メッセージ・キューを参照する変換テーブルを指定します。

次の表は、生成された Java コードが実行時にアクセスできる変換テーブルをまとめたものです。各変換テーブル名のフォーマットは、CSOJx で、c および x の意味は、次のとおりです。

- x 呼び出し先のプラットフォームでサポートされているコード・ページ番号を表します。各番号は、『*Character Data Representation Architecture Reference and Registry*, SC09-2190』に指定されています。各変換テーブルによってサポートされているコード化文字セットは、レジストリーによって識別できます。

言語	呼び出し先プログラムのプラットフォーム		
	UNIX	Windows 2000/NT/XP	z/OS UNIX システム・サービスまたは iSeries Java
アラビア語	CSOJ1046	CSOJ1256	CSOJ420
中国語 (簡体字)	CSOJ1381	CSOJ1386	CSOJ1388
中国語 (繁体字)	CSOJ950	CSOJ950	CSOJ1371
キリル文字言語	CSOJ866	CSOJ1251	CSOJ1025
デンマーク語	CSOJ850	CSOJ850	CSOJ277
東ヨーロッパ言語	CSOJ852	CSOJ1250	CSOJ870
英語 (英国)	CSOJ850	CSOJ1252	CSOJ285
英語 (米国)	CSOJ850	CSOJ1252	CSOJ037
フランス語	CSOJ850	CSOJ1252	CSOJ297
ドイツ語	CSOJ850	CSOJ1252	CSOJ273
ギリシャ語	CSOJ813	CSOJ1253	CSOJ875
ヘブライ語	CSOJ856	CSOJ1255	CSOJ424
日本語	CSOJ943	CSOJ943	CSOJ1390 (カタカナ SBCS)、CSOJ1399 (ラテン語 SBCS)
韓国語	CSOJ949	CSOJ949	CSOJ1364
ポルトガル語	CSOJ850	CSOJ1252	CSOJ037
スペイン語	CSOJ850	CSOJ1252	CSOJ284
スウェーデン語	CSOJ850	CSOJ1252	CSOJ278
スイス・ドイツ語	CSOJ850	CSOJ1252	CSOJ500

呼び出し先プログラムのプラットフォーム			
言語	UNIX	Windows 2000/NT/XP	z/OS UNIX システム・サービスまたは iSeries Java
トルコ語	CSOJ920	CSOJ1254	CSOJ1026

プログラムを Java から呼び出す場合に、リンケージ・オプション・パーツで変換テーブルの値を指定しない場合は、デフォルトの変換テーブルは英語 (米国) 用の変換テーブルになります。

変換アルゴリズム

レコードと構造体のデータ変換は、副構造のない構造体項目の宣言が基本になります。

型が CHAR、DBCHAR、または MBCHAR のデータは、(EGL 生成の起動側で変換される場合) Java 変換テーブルに従って変換されます。

充てん文字データ項目 (名前のないデータ項目) の場合や、型が DECIMAL、PACF、HEX、または UNICODE のデータ項目の場合は、変換が実行されません。

MBCHAR データの EBCDIC から ASCII への変換では、変換ルーチンによってシフトイン・シフトアウト (SO/SI) 文字が削除され、この文字数と同じ数のブランクがデータ項目の末尾に追加されます。ASCII から EBCDIC への変換では、変換ルーチンによって 2 バイトのストリングの前後に SO/SI 文字が挿入され、フィールドに収容できる有効な最後の文字で値が切り捨てられます。MBCHAR フィールドが可変長レコード内にあり、現行レコードの末尾が MBCHAR フィールド内にある場合、レコード長は SO/SI 文字の挿入または削除を反映するように調整されます。レコード長は、現行レコードの末尾がどこにあるかを示しています。

型が BIN のデータ項目では、呼び出し側または呼び出し先のプラットフォームで Intel の 2 進数形式が使用され、もう一方のプラットフォームではその形式が使用されていない場合、バイト数を基準にしたデータ項目の順序が変換ルーチンによって逆になります。

型が NUM または NUMC のデータ項目では、変換ルーチンが CHAR アルゴリズムを使用して、最後のバイトを除くすべてのバイトを変換します。符号ハーフバイト (フィールドの最後のバイトのうちの最初の半バイト) は、次の表に示す 16 進値に従って変換されます。

NUM 型の EBCDIC	NUMC 型の EBCDIC	ASCII
F (正符号)	C	3
D (負符号)	D	7

関連する参照項目

407 ページの『関連エレメント』

510 ページの『双方向言語テキスト』

双方向言語テキスト

アラビア語およびヘブライ語などの双方向 (bidi) 言語とは、テキストが右から左の順にユーザーに表示され、テキスト内の数値およびラテン語英字ストリングが左から右に表示される言語のことです。また、プログラム変数内に文字が現れる順序は異なる場合があります。テキストは通常は論理 順、つまり文字が入力フィールドに入力される順に保管されます。

これらの順序付けおよび他の関連する表示特性に相違があるため、プログラムは、双方向テキスト・ストリングをある形式から別の形式に変換できる必要があります。bidi 変換属性は、プログラムとは別に作成される bidi 変換テーブル (.bct) ファイルで指定します。プログラムは変換テーブルの名前を参照して、属性変換を実行する方法を指示します。

どのような場合でも、bidi 変換テーブル参照は、.bct 拡張子を除く 1 から 8 文字のファイル名として指定します。例えば、mybct.bct という名前の bidi 変換テーブルを作成してある場合は、プログラムの先頭に次の文を追加することによって、プログラムに formConversionTable という値を設定できます。

```
sysVar.formConversionTable = "mybct.bct" ;
```

作業内容は次のとおりです。

- 実行する変換を指定する bidi 変換テーブルを作成します。テキストまたは印刷書式で表示されるデータを変換するために必要なテーブルは異なることに注意してください。
- bidi 言語テキストを含むテキストまたは印刷書式を使用するプログラムを生成する場合は、書式を表示する前に、変換テーブル名をシステム関数 `sysVar.formConversionTable` に割り当てるプログラムに文を追加します。

bidi 変換テーブル・ウィザード・プラグイン (ファイル BidiConversionTable.zip に入っている) を使用して、次のように bidi 変換テーブルをビルドします。

1. 次の Web サイトからファイルをダウンロードします。
2. ワークベンチ・ディレクトリーにファイルを unzip します。
3. ウィザードの実行を開始するには、「ファイル」>「新規」>「その他」>「**BidiConversionTable**」をクリックします。

EGL プログラムで使用されるテーブルの名前は 8 文字以下で、.bct 拡張子が付いていなければなりません。

4. ウィザードの実行中に F1 を押すと、ヘルプが表示され、テーブルを作成するための適切なオプションを選択するのに役立ちます。

関連する参照項目

データの初期化

EGL で生成されたプログラムがレコードを自動的に初期化する場合 (後述のような場合)、最下位レベルの構造体の各項目は、プリミティブ型に対応する値に設定されます。書式初期化も同様ですが、デフォルトをオーバーライドする値を書式宣言で割り当てることができる点が異なります。

初期化は、以下の状態でも行われます。

- 変数 (特に項目、レコード、または静的配列) の **initialized** プロパティを **yes** に設定する。
- ロジックに **set** の特定のバリエーションを組み込む。

次の表に、初期化値の詳細を示します。

プリミティブ型	初期化値
ANY	変数の型は未定義です。
BIN 型および整数型 (BIGINT、INT、および SMALLINT)、HEX、FLOAT、SMALLFLOAT	2 進ゼロ
CHAR、MBCHAR	単一バイトのブランク
DATE、TIME、TIMESTAMP	マシン・クロックの (TIMESTAMP の場合は、マスクが必要とするバイト数の) 現行値
DBCHAR	2 バイトのブランク
DECIMAL、MONEY、NUM、NUMC、PACF	数値のゼロ
INTERVAL	前に正符号が付いた (マスクが必要とするバイト数の) 数値のゼロ
UNICODE	ユニコードのブランク (16 進の 0020)

構造体では、最下位レベルの構造体の項目のみが考慮されます。例えば、HEX 型の構造体の項目が CHAR 型の構造体の項目に従属している場合、メモリ領域は 2 進ゼロで初期化されます。

プログラムまたは関数パラメーターとして受け取られるレコードまたは項目は、自動的に初期化されません。

EGL で生成されたプログラムは、ローカルかグローバルかにかかわらず、レコードを初期化します。

関連する概念

- 149 ページの『関数パーツ』
- 139 ページの『DataItem パーツ』
- 147 ページの『プログラム・パーツ』
- 141 ページの『レコード・パーツ』
- 28 ページの『固定構造体』

関連する参照項目

93 ページの『EGL ステートメント』

685 ページの『set』

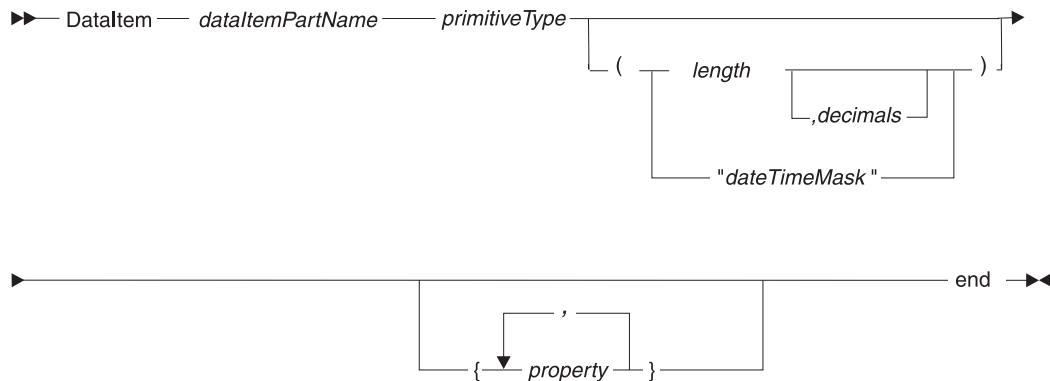
EGL ソース形式の DataItem パーツ

DataItem パーツを EGL ファイルに宣言します。詳細については、『EGL ソース形式』に説明します。

データ項目パーツの例を次に示します。

```
DataItem myDataItemPart
    BIN(9,2)
end
```

dataItem パーツの構文図は、以下のとおりです。



DataItem *dataItemPartName* ... **end**

パーツを **dataItem** パーツとして識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

primitiveType

dataItem パーツに割り当てられるプリミティブ型。

length

dataItem パーツの長さを反映する整数。パーツに基づく変数の値には、指定された数の文字または数字が含まれます。

decimals

MONEY 以外の固定の数値型 (特に、BIN、DECIMAL、NUM、NUMC、または PACF) には、*decimals* を指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

"dateTimeMask"

INTERVAL 型または TIMESTAMP 型の項目には、*"dateTimeMask"* を指定できます。これは、項目の値の特定の位置に意味 (「年の桁」など) を割り当てるものです。このマスクは、データと一緒に格納されません。

property

項目のプロパティ。詳細については、『*EGL プロパティとオーバーライドの概要*』に説明します。

関連する概念

- 139 ページの『*DataItem パーツ*』
- 15 ページの『*EGL プロジェクト、パッケージ、およびファイル*』
- 68 ページの『*EGL プロパティの概要*』
- 24 ページの『*パーツの参照*』
- 19 ページの『*パーツ*』

関連するタスク

- 812 ページの『*EGL ステートメントおよびコマンドの構文図*』

関連する参照項目

- 532 ページの『*EGL ソース形式*』
- 570 ページの『*EGL ソース形式の関数パーツ*』
- 578 ページの『*EGL ソース形式の索引付きレコード・パーツ*』
- 714 ページの『*EGL ソース形式の MQ レコード・パーツ*』
- 725 ページの『*命名規則*』
- 37 ページの『*プリミティブ型*』
- 783 ページの『*EGL ソース形式のプログラム・パーツ*』
- 796 ページの『*EGL ソース形式の相対レコード・パーツ*』
- 799 ページの『*EGL ソース形式のシリアル・レコード・パーツ*』
- 804 ページの『*EGL ソース形式の SQL レコード・パーツ*』

EGL ソース形式の DataTable パーツ

dataTable パーツは EGL ファイルで宣言します。これについては、『*EGL プロジェクト、パッケージ、およびファイル*』で説明しています。このパーツは生成可能なパーツです。つまり、このパーツは、ファイルの最上位にあり、ファイルと同じ名前を持つ必要があります。

1 つの dataTable は、プログラムの使用宣言、または (プログラムの唯一のメッセージ・テーブルの場合は) プログラムの **msgTablePrefix** プロパティによって、そのプログラムに関連付けられます。dataTable は、ページ・ハンドラーの使用宣言により、ページ・ハンドラーに関連しています。

dataTable パーツの一例を以下に示します。

```
DataTable myDataTablePart type basicTable
{
    { shared = yes }
    myColumn1 char(10);
    myColumn2 char(10);
    myColumn3 char(10);

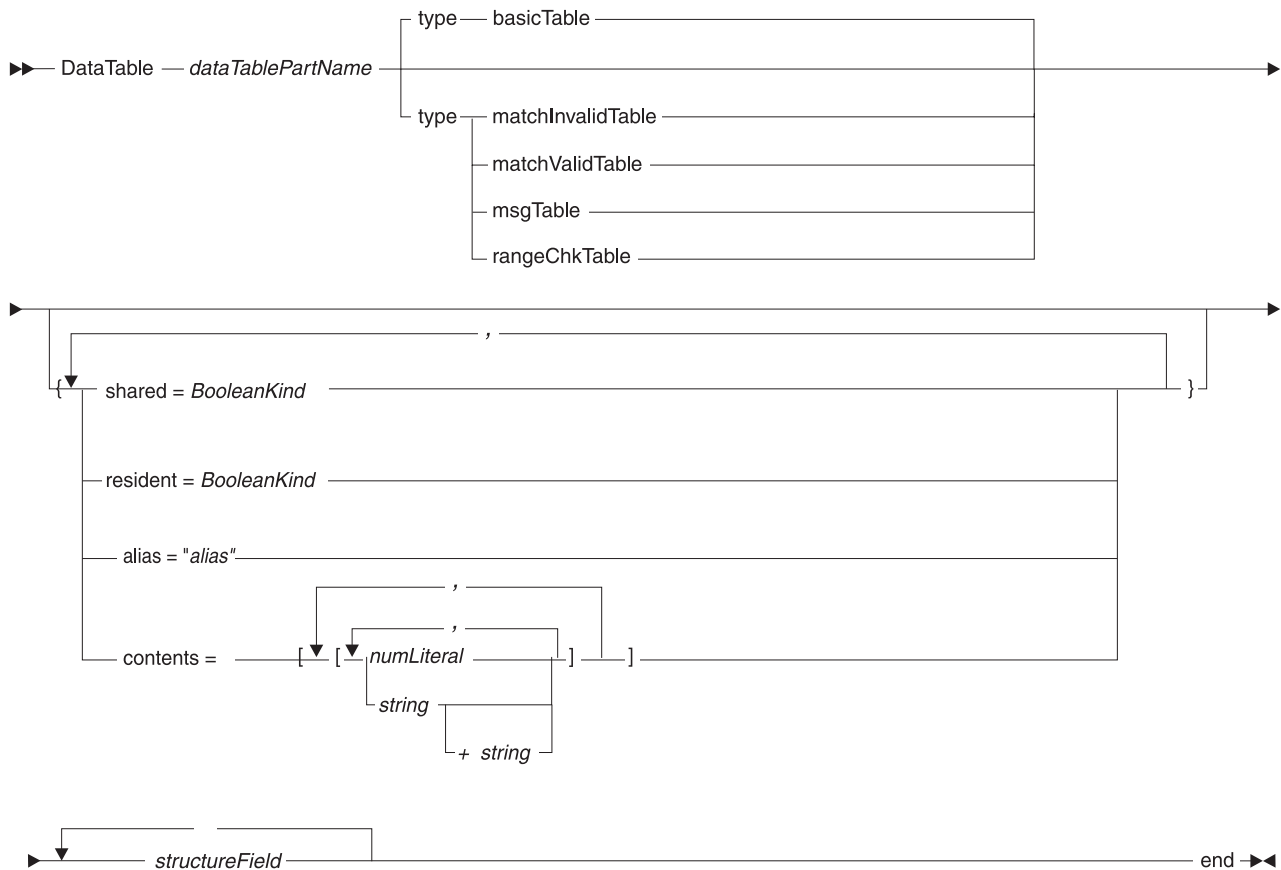
    { contents = [
        [ "row1 col1", "row1 col2", "row1 " + "col3" ] ,
        [ "row2 col1", "row2 col2", "row2 " + "col3" ] ,
    ] }
```

```

        [ "row3 col1", "row3 col2", "row3 col3"      ]
      ]
    }
  end

```

dataTable パーツの構文図は、以下のとおりです。



dataTable dataTablePartName ... end

パーツを **dataTable** として識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

basicTable (デフォルト)

プログラム・ロジックで使用される情報（例えば、国およびそれに関連付けられているコード）を含んでいます。

matchInvalidTable

テキスト・フィールドの **validatorDataTable** プロパティで指定され、ユーザーの入力は、**dataTable** の最初の列の値とは異なっている必要があることを示します。検証に失敗すると、EGL ランタイムは以下のように応答します。

- **validatorDataTable** プロパティで参照されるテーブルにアクセスする。
- テキスト・フィールド固有の **validatorDataTableMsgKey** プロパティで識別されるメッセージを取り出す。
- 書式固有の **msgField** プロパティで識別されるテキスト・フィールドにメッセージを表示する。

matchValidTable

テキスト・フィールドの **validatorDataTable** プロパティで指定され、ユーザーの入力が **dataTable** の最初の列の値に一致する必要があることを示します。検証に失敗すると、EGL ランタイムは以下のように応答します。

- **validatorDataTable** プロパティで参照されるテーブルにアクセスする。
- テキスト・フィールド固有の **validatorDataTableMsgKey** プロパティで識別されるメッセージを取り出す。
- 書式固有の **msgField** プロパティで識別されるフィールドにメッセージを表示する。

msgTable

ランタイム・メッセージが含まれています。メッセージは、以下の状況で表示されます。

- テーブルが、プログラムのメッセージ・テーブルである。テーブルとプログラムの関連付けは、プログラムのプロパティ **msgTablePrefix** が、**dataTable** の名前の先頭の 1 から 4 文字であるテーブル接頭部を参照している場合に行われます。この名前の残りの文字は、以下の表にリストしている各国語コードの 1 つです。

言語	各国語コード
ブラジル・ポルトガル語	PTB
中国語 (簡体字)	CHS
中国語 (繁体字)	CHT
英語、大文字	ENP
英語 (米国)	ENU
フランス語	FRA
ドイツ語	DEU
イタリア語	ITA
日本語、カタカナ (1 バイト文字セット)	JPN
韓国語	KOR
スペイン語	ESP
スイス・ドイツ語	DES

- プログラムは、『*ConverseLib.displayMsgNum*』および『*ConverseLib.validationFailed*』で説明されている 2 つのメカニズムの内の 1 つによってメッセージを取り出し、それを表示します。

rangeChkTable

テキスト・フィールドの **validatorDataTable** プロパティで指定され、ユーザーの入力が、少なくとも 1 つのデータ・テーブル行の第 1 列の値と第 2 列の値との間の値に一致する必要があることを示します。(範囲は包括的です。ユーザーの入力が任意の行の最初の列または 2 番目の列の値と一致している場合は、有効です。)検証に失敗すると、EGL ランタイムは以下のように応答します。

- **validatorDataTable** プロパティで参照されるテーブルにアクセスする。
- テキスト・フィールド固有の **validatorDataTableMsgKey** プロパティで識別されるメッセージを取り出す。

- 書式固有の **msgField** プロパティで識別されるフィールドにメッセージを表示する。

"alias"

生成された出力の名前に取り込まれるストリング。別名を指定しなかった場合は、dataTable 名が代わりに使用されます。

shared

dataTable の同じインスタンスが、同じ実行単位の複数のプログラムによって使用されるかどうかを示します。有効な値は、yes および no (デフォルト) です。**shared** の値が no の場合は、実行単位内の各プログラムは、dataTable の固有のコピーを持ちます。

このプロパティは、dataTable の同じインスタンスが、同じ実行単位の各プログラムによって使用されるかどうかを示します。**shared** の値が no の場合は、実行単位内の各プログラムは、dataTable の固有のコピーを持ちます。

実行時に行われた変更は、dataTable にアクセスできるすべてのプログラムに対して可視になっており、その dataTable がアンロードされるまで、その変更はそのまま維持されます。いつ dataTable がアンロードされるかは、ほとんどの場合、**resident** プロパティ (後述) の値によって決まります。詳しくは、このプロパティの説明を参照してください。

resident

dataTable にアクセスしたすべてのプログラムが終了した後も、その dataTable をメモリー内に保持しておくかどうかを示します。

有効値は yes および no です。デフォルトは no です。

resident プロパティを yes に設定すると、**shared** の値に関係なく、dataTable は共用されます。

dataTable を常駐させておくことの利点は、以下のとおりです。

- dataTable は、以前に実行されたプログラムによって書き込まれた値をすべて保持する。
- 追加のロード処理を行わなくても、テーブルに即時にアクセスできる。

常駐 dataTable は、実行単位が終了するまでロードされたままです。ただし、非常駐 dataTable は、それを使用しているプログラムが終了するとアンロードされます。

注: dataTable は、EGL ランタイムが使用宣言を処理するときではなく、プログラムが最初にアクセスしたときに、(必要であれば) メモリーにロードされず。

contents

dataTable セルの値。各セルの種類は、以下のいずれかです。

- 数値リテラル
- 文字列リテラル、または文字列リテラルの連結

ある特定の行の内容の種類には、最上位の構造体フィールド (各フィールドは、列定義を表します) との互換性が必要です。

structureField

構造体フィールド。詳細については、『EGL ソース形式の構造体フィールド』を参照してください。

関連する概念

155 ページの『DataTable』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

798 ページの『実行単位』

関連する参照項目

725 ページの『命名規則』

808 ページの『EGL ソース形式の構造体フィールド』

847 ページの『displayMsgNum()』

849 ページの『validationFailed()』

1020 ページの『使用宣言』

EGL ビルド・パスおよび egldpath

各 EGL プロジェクトおよび EGL Web プロジェクトは、他のプロジェクト内のパーツを参照することができるように、EGL ビルド・パスに関連付けられています。EGL ビルド・パスが使用される状況およびビルド・パスの順序が重要である理由について詳しくは、『パーツへの参照』を参照してください。

EGL ビルド・パスを指定する場合、ビルド・パスにリストされている 1 つ以上のプロジェクトを選択してエクスポート することができます。次に、宣言されているプロジェクトを参照するプロジェクトがある場合、以下の例に示すように、エクスポートされたプロジェクトそれぞれが、プロジェクト参照に使用できるようになります。

- プロジェクト A の EGL ビルド・パスは、以下のプロジェクトで構成されている (この順序で)。

A、B、C、D

プロジェクト B およびプロジェクト D がエクスポートされます。

- プロジェクト L の EGL ビルド・パスは、以下のプロジェクトで構成されている (この順序で)。

L、J、A、Z

- プロジェクト L の有効なビルド・パスには、プロジェクト A からエクスポートされたプロジェクトも含まれています。この場合、プロジェクト L の EGL ビルド・パスは、実際は以下ようになります。

L、J、A、B、D、Z

エクスポートされたプロジェクトは、エクスポート元のプロジェクトの後に配置されます。また、プロジェクトは、エクスポート元のプロジェクトのビルド・パスにリストされている順序で配置されます。

プロジェクトのビルド・パスには、通常、このプロジェクト自体も含まれ、ビルド・パスの順序の最初に配置されます (推奨どおり)。プロジェクト内に複数の EGL ソース・フォルダーがある場合は、すべての EGL ソース・フォルダーが、こ

のプロジェクトを参照するプロジェクトで使用されるフォルダーの順序で、このプロジェクトの EGL ビルド・パスにリストされている必要があります。

同じ名前のパッケージを異なるプロジェクトまたは同一プロジェクトの異なるフォルダーで使用しないようにしてください。

EGL SDK 内に生成する場合は、以下のようになります。

- プロジェクト情報が使用不可になる。
- コマンド行引数 *eglp*ath が、EGL ビルド・パスの機能を置き換える。*eglp*ath は、EGL SDK がパーツの参照を解決する場合に検索するオペレーティング・システム・ディレクトリーのリストです。
- *eglp*ath が使用される場合の規則は、EGL ビルド・パスが使用される場合の規則と同等である。ただし、ディレクトリーは、プロジェクトのようにエクスポートすることができません。

EGL SDK を使用する場合、同じ名前のパッケージを異なるディレクトリーで使用しないようにしてください。

関連する概念

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

24 ページの『パーツの参照』

関連するタスク

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

関連する参照項目

529 ページの『EGLSDK』

EGLCMD

コマンド EGLCMD を使用すると、ワークベンチ・バッチ・インターフェースにアクセスできます。詳細については、『ワークベンチ・バッチ・インターフェースからの生成』を参照してください。

構文

EGLCMD を呼び出す際の構文を次に示します。

パスは、二重引用符で囲みます。

-buildDescriptorFile *bdFile*

ビルド記述子を含むビルド・ファイルの絶対パスまたは相対パスを指定します。
相対パスは、コマンドを実行するディレクトリーを基準にした相対パスとなります。

パスは、二重引用符で囲みます。

-buildDescriptorName *bdName*

生成をガイドするビルド記述子パーツの名前を指定します。ビルド記述子は、
EGL ビルド (.eglbuild) ファイルのトップレベルになければなりません。

-sqlID *sqlID*

ビルド記述子オプション *sqlID* の値を設定します。

-sqlPassword *sqlPW*

ビルド記述子オプション *sqlPassword* の値を設定します。

-destUserid *destID*

ビルド記述子オプション *destUserID* の値を設定します。

-destPassword *destPW*

ビルド記述子オプション *destPassword* の値を設定します。

コマンド *EGLCMD* を呼び出すときに指定するビルド記述子オプションは、EGL コマンド・ファイルにリストされているビルド記述子のオプションよりも優先されます。

例

コマンドの例を次に示します。スペースの関係で、1 行のコマンドを複数行に分割しているものもあります。

```
java EGLCMD "commandfile.xml"

java EGLCMD "commandfile.xml" -data "c:\myWorkSpace"

java EGLCMD generate
-generateFile "c:\myProg.eglpgm"
-data "myWorkSpace"
-buildDescriptorFile "c:\myBuild.eglbuild"
-buildDescriptorName myBuildDescriptor

java EGLCMD "myCommand.xml"
-data "my WorkSpace"
-sqlID myID -sqlPassword myPW
-destUserID myUserID -destPassword myPass
```

関連する概念

361 ページの『ワークベンチ・バッチ・インターフェースからの生成』

関連するタスク

360 ページの『ワークベンチのバッチ・インターフェースからの生成』

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

423 ページの『destPassword』

423 ページの『destUserID』
436 ページの『sqlID』
438 ページの『sqlPassword』

EGL コマンド・ファイル

EGL コマンド・ファイルには、ワークベンチの外部で出力を生成する際に処理したい EGL ファイルを指定します。この EGL ファイルは、ワークベンチ・バッチ・インターフェース (コマンド EGLCMD) または EGL SDK (コマンド EGLSDK) のいずれを使用しているかにかかわらず指定できます。EGL コマンド・ファイルは、次の 2 つの方法で作成できます。

- 後述する規則に従って、手作業で作成する。
- 「EGL 生成」ウィザードで作成する。(詳細については、『ワークベンチでの生成』を参照してください)

コマンド・ファイルは XML ファイルです。ファイル名には、拡張子 `.xml` を付ける必要があります。ファイル名には、大文字と小文字を自由に使用できます。ファイル・コンテンツは、次の文書型定義 (DTD) に準拠している必要があります。

```
installationDir¥egl¥eclipse¥plugins¥  
com.ibm.etools.egl.utilities_version¥  
dtd¥eglcommands_5_1.dtd
```

installationDir

製品のインストール・ディレクトリー。例えば、`C:¥Program Files¥IBM¥RSPD¥6.0` など。これから使用しようとしている製品をインストールする前に Rational Developer 製品をインストールし、保持していた場合は、以前のインストールで使用されていたディレクトリーを指定することが必要になる場合があります。

version

インストール済みのプラグインのバージョン (例: 6.0.0)

次の表は、DTD でサポートされている要素と属性をまとめたものです。要素名および属性名には、大文字小文字の区別があります。

要素	属性	属性値
EGLCOMMANDS (必須)	eglp ath	<p>『<i>eglp</i>ath』で説明しているように、eglp_{ath} 属性は、EGL が import ステートメントでパーツ名を解決するときに検索するディレクトリーを指定します。この属性は、必要に応じて指定します。属性を指定すると、1 つ以上のディレクトリー名を含む引用符付きストリングが参照されます。(引用符付きストリング内にディレクトリー名が複数存在する場合は、ディレクトリー名とディレクトリー名の間がセミコロンで区切られています)</p> <p>この属性は、コマンド EGLSDK がコマンド・ファイルを参照している場合にのみ使用されます。コマンド EGLCMD が使用中の場合、eglp_{ath} の値は無視されます。その代わりに、import ステートメントは、EGL プロジェクト・パスに従って解決されます。詳細については、『インポート』を参照してください。</p>
buildDescriptor (省略可: 『ビルド記述子パーツ』で説明するマスター・ビルド記述子を使用している場合は、この値を指定する必要はありません)	name	<p>生成をガイドするビルド記述子パーツ名。ビルド記述子は、EGL ビルド (.eglbld) ファイルのトップレベルになければなりません。</p> <p>EGLCMD または EGLSDK を呼び出すときに指定するビルド記述子オプションは、EGL コマンド・ファイルにリストされているビルド記述子のオプションよりも優先されます。</p>
	file	<p>ビルド記述子を含む EGL ファイルの絶対パスまたは相対パス。EGLCMD に指定する相対パスは、Enterprise Developer ワークスペースのパス名を基準にした相対パスです。EGLSDK に指定する相対パスは、コマンドを実行するディレクトリーを基準にした相対パスです。</p> <p>パス名にスペースが含まれる場合は、パス名を二重引用符で囲む必要があります。</p>
generate (オプション)	file	<p>処理したいパーツを含む EGL ファイルの絶対パスまたは相対パス。EGLCMD に指定する相対パスは、Enterprise Developer ワークスペースのパス名を基準にした相対パスです。EGLSDK に指定する相対パスは、コマンドを実行するディレクトリーを基準にした相対パスです。</p> <p>パス名にスペースが含まれる場合は、パス名を二重引用符で囲む必要があります。</p> <p>ファイル属性を省略すると、生成は実行されません。</p>

コマンド・ファイルの例

このセクションでは、コマンド・ファイルを 2 つ説明します。EGL プログラム・ファイルが存在するディレクトリー内で EGLSDK コマンドを実行する場合は、

EGLCMD コマンドと EGLSDK コマンドのいずれを使用するかにかかわらず、2 つのファイルで生成される結果は同じになります。

次のコマンド・ファイルには、ビルド記述子 myBDescPart を使用してプログラム myProgram を生成する generate コマンドが含まれています。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:¥mydata¥entdev¥workspace¥projectinteract">
  <generate file="projectinteract¥myProgram.eglpgm">
    <buildDescriptor name="myBDescPart" file="projectinteract¥mybdesc.eglbld"/>
  </generate>
</EGLCOMMANDS>
```

次の例には、generate コマンドが 2 つ含まれています。どちらのコマンドも、マスター・ビルド記述子を暗黙的に使用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EGLCOMMANDS PUBLIC "-//IBM//DTD EGLCOMMANDS 5.1//EN" "">
<EGLCOMMANDS eglpath="C:¥mydata¥entdev¥workspace¥projecttrade">
  <generate file="projecttrade¥program2.eglpgm"/>
  <generate file="projecttrade¥program3.eglpgm"/>
</EGLCOMMANDS>
```

関連する概念

- 319 ページの『ビルド記述子パーツ』
- 362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』
- 361 ページの『ワークベンチ・バッチ・インターフェースからの生成』
- 36 ページの『インポート』

関連するタスク

- 362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』
- 360 ページの『ワークベンチのバッチ・インターフェースからの生成』
- 358 ページの『ワークベンチでの生成方法』

関連する参照項目

- 518 ページの『EGLCMD』
- 521 ページの『EGL コマンド・ファイル』
- 517 ページの『EGL ビルド・パスおよび eglpath』
- 529 ページの『EGLSDK』

EGL エディター

EGL ファイル (extension .egl) を変更するには、コンテンツ・アシストによるガイドがある EGL ソース・エディターを使用します。

EGL ビルド・ファイル (extension .eglbld) を変更するには、以下のいずれかの手順を実行します。

- ビルド・ファイルの作成
-
- ビルド記述子パーツの追加
- リンケージ・オプション・パーツの追加
-

- リソース関連パーツの追加

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

19 ページの『パーツ』

関連する参照項目

『EGL でのコンテンツ・アシスト』

EGL でのコンテンツ・アシスト

EGL エディターには、ソース・ファイルに追加できる情報を処理するコンテンツ・アシスト があります。1 から 2 回のキー・ストロークでパーツ、変数、関数の名前を完了するか、または *template* (パーツの概要) をソース・ファイル内に配置します。

コンテンツ・アシストを活動化するキー・ストロークは、**Ctrl+ Space** です。

関連するタスク

124 ページの『テンプレートの設定の変更』

135 ページの『コンテンツ・アシストを使用した EGL テンプレートの使用』

EGL での列挙型

EGL では、プロパティまたはフィールドの値が特定の列挙型 に制限されている場合があります。列挙型は、事前定義値の一種です。例えば、プロパティ **color** は、列挙型 **ColorKind** の値を受け入れます。この列挙型の有効な値には *white* や *red* があります。

列挙型の名前を使用して列挙型値を修飾できます。したがって、上に示した値は *ColorKind.white* および *ColorKind.red* と記述できます。ただし、列挙型値を修飾する必要があるのは、ご使用のコードが列挙型値と同じ名前を持つ変数または定数にアクセスできる場合のみです。例えば、*red* という名前の変数が有効範囲内にある場合、シンボル *red* は、列挙型値ではなく変数を参照します。

次の列挙型のリストに、列挙型値を示します。ただし、これらの値の説明については、列挙型が意味を持つプロパティまたはフィールドのコンテキストに関連する、別の場所に記載されています。

AlignKind

center

left

none

right

Boolean

yes

no

CallingConventionKind

I4GL

Library

CaseFormatKind

defaultCase

lower

upper

ColorKind

black (コンソール・フィールドに対してのみ有効)

blue

cyan

defaultColor

green

magenta

red

yellow

white

DataSource

databaseConnection

reportData

sqlStatement

DeviceTypeKind

doubleByte

singleByte

DisplayUseKind

button

hyperlink

input

output

secret

table

EventKind

AFTER_DELETE

AFTER_FIELD

AFTER_OPENUI

AFTER_INSERT

AFTER_ROW

BEFORE_DELETE

BEFORE_FIELD

BEFORE_OPENUI

BEFORE_INSERT

BEFORE_ROW
ON_KEY
MENU_ACTION

ExportFormat

html
pdf
text
xml

HighlightKind

blink
defaultHighlight
noHighlight
reverse
underline

IndexOrientationKind

across
down

IntensityKind

bold
defaultHighlight
dim
invisible
normalIntensity

LineWrapKind

character
compress (コンソール・フィールドに対してのみ有効)
word

OutlineKind

bottom
left
right
top

注: sysLib.box は、[left,right,top,bottom] と等価の定数です。 sysLib.noOutline は、アウトラインを付けないことを意味する定数です。

PfKeyKind

pf*n*、ただし、(1 <= *n* <=24)

ProtectKind

skip

no

yes

SelectTypeKind

index

value

SignKind

leading

none

parens

trailing

関連する概念

68 ページの『EGL プロパティの概要』

62 ページの『EGL での変数の参照』

EGL 予約語

EGL の予約語には、次の 2 つのカテゴリがあります。

- SQL ステートメント以外の予約語
- SQL ステートメントの予約語

SQL ステートメント以外の予約語

SQL ステートメント以外の予約語を次に示します。これらの予約語は、大文字と小文字を自由に組み合わせたものです。

- absolute、add、all、any、as
- bigInt、bin、bind、blob、boolean、by、byName、byPosition
- call、case、char、clob、close、const、continue、converse、current
- dataItem、dataTable、date、dbChar、decimal、decrement、delete、display、dliCall
- else、embed、end、escape、execute、exit、externallyDefined
- false、field、first、float、for、forEach、form、formGroup、forUpdate、forward、freeSql、from、function
- get、goto
- handler、hex、hold
- if、import、in、inOut、insert、int、interval、into、is、isa
- label、languageBundle、last、library、like
- matches、mbChar、money、move
- new、next、no、noRefresh、not、nullable、num、number、numc
- onEvent、onException、open、openUI、otherwise、out
- pacf、package、pageHandler、passing、prepare、previous、print、private、program、psb
- record、ref、relative、replace、return、returning、returns
- scroll、self、set、show、singleRow、smallFloat、smallInt、sql、sqlCondition、stack、string
- this、time、timeStamp、to、transaction、transfer、true、try、type
- unicode、update、url、use、using、usingKeys

- when、while、with、withinParent
- yes

SQL ステートメントの予約語

SQL ステートメントの予約語を次に示します。これらの予約語は、大文字と小文字を自由に組み合わせたものです。

- absolute、action、add、alias、all、allocate、alter、and、any、are、as、asc、assertion、at、authorization、avg
- begin、between、bigint、binaryLargeObject、bit、bit_length、blob、boolean、both、by
- call、cascade、cascaded、case、cast、catalog、char、char_length、character、character_length、characterLargeObject、characterVarying、charLargeObject、charVarying、check、clob、close、coalesce、collate、collation、column、comment、commit、connect、connection、constraint、constraints、continue、convert、copy、corresponding、count、create、cross、current、current_date、current_time、current_timestamp、current_user、cursor
- data、database、date、dateTime、day、deallocate、dec、decimal、declare、default、deferrable、deferred、delete、desc、describe、diagnostics、disconnect、distinct、domain、double、doublePrecision、drop
- else、end、endExec、escape、except、exception、exec、execute、exists、explain、external、extract
- false、fetch、first、float、for、foreign、found、from、full
- get、getCurrentConnection、global、go、goto、grant、group
- having、hour
- identity、image、immediate、in、index、indicator、initially、inner、input、insensitive、insert、int、integer、intersect、into、is、isolation
- join
- key
- language、last、leading、left、level、like、local、long、longint、lower、ltrim
- match、max、min、minute、module、month
- national、nationalCharacter、nationalCharacterLargeObject、nationalCharacterVarying、nationalCharLargeObject、nationalCharVarying、natural、nchar、ncharVarying、nclob、next、no、not、null、nullIf、number、numeric
- octet_length、of、on、only、open、option、or、order、outer、output、overlaps
- pad、partial、position、prepare、preserve、primary、prior、privileges、procedure、public
- raw、read、real、references、relative、restrict、revoke、right、rollback、rows、rtrim、runtimeStatistics
- schema、scroll、second、section、select、session、session_user、set、signal、size、smallint、some、space、sql、sqlcode、sqlerror、sqlstate、substr、substring、sum、system_user
- table、tablespace、temporary、terminate、then、time、timestamp、timezone_hour、timezone_minute、tinyint、to、trailing、transaction、translate、translation、trim、true
- uncatalog、union、unique、unknown、update、upper、usage、user、using
- values、varbinary、varchar、varchar2、varying、view
- when、whenever、where、with、work、write

- year
- zone

関連する参照項目

93 ページの『EGL ステートメント』

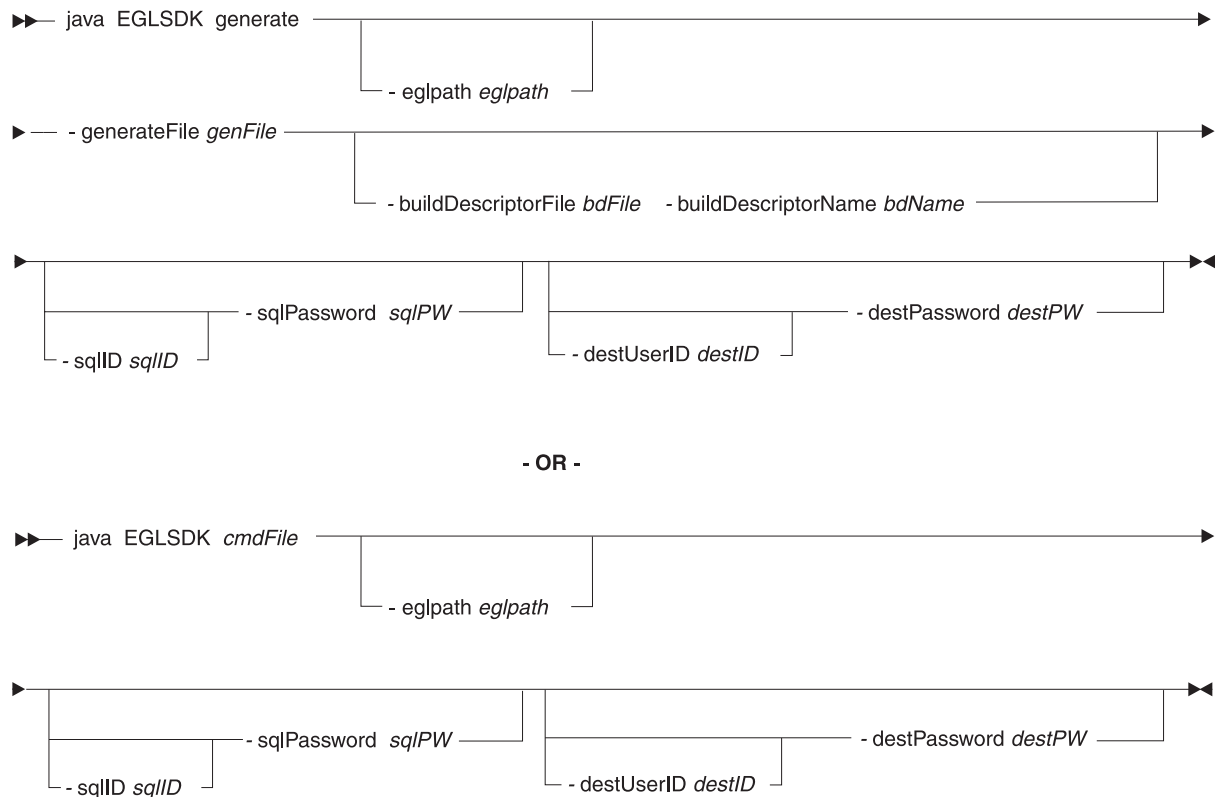
725 ページの『命名規則』

EGLSDK

コマンド EGLSDK を使用すると、EGL ソフトウェア開発キット (EGL SDK) を利用できます。詳細については、『EGL SDK からの生成』を参照してください。

構文

EGLSDK を呼び出す際の構文を次に示します。



generate

出力を生成するために使用する EGL ファイルおよびビルド記述子パーツをコマンド自体が参照することを示します。この場合、コマンド EGLSDK は、コマンド・ファイルを参照しません。

cmdFile

EGL コマンド・ファイル に記述されたファイルの絶対パスまたは相対パスを指定します。相対パスは、コマンドを実行するディレクトリーを基準にした相対パスとなります。

パスは、二重引用符で囲みます。

?eglpath *eglpath*

『*eglpath*』で説明しているように、*eglpath* オプションは、EGL が import ステートメントでパーツ名を解決するときに検索するディレクトリーを指定します。1 以上のディレクトリー名を含むストリングを引用符を付けて指定します。ディレクトリー名とディレクトリー名の間は、セミコロンで区切ります。

?generateFile *genFile*

処理したいパーツを含む EGL ファイルの絶対パスまたは相対パス。相対パスは、コマンドを実行するディレクトリーを基準にした相対パスとなります。パスは、二重引用符で囲みます。

?buildDescriptorFile *bdFile*

ビルド記述子を含むビルド・ファイルの絶対パスまたは相対パス。相対パスは、コマンドを実行するディレクトリーを基準にした相対パスとなります。パスは、二重引用符で囲みます。

?buildDescriptorName *bdName*

生成をガイドするビルド記述子パーツの名前。ビルド記述子は、EGL ビルド (*eglbld*) ファイルのトップレベルになければなりません。

?sqlID *sqlID*

ビルド記述子オプション *sqlID* の値を設定します。

?sqlPassword *sqlPW*

ビルド記述子オプション *sqlPassword* の値を設定します。

?destUserid *destID*

ビルド記述子オプション *destUserID* の値を設定します。

?destPassword *destPW*

ビルド記述子オプション *destPassword* の値を設定します。

コマンド EGLSDK を呼び出すときに指定する *eglpath* の値は、EGL コマンド・ファイル内のどの *eglpath* の値よりも優先されます。同様に、コマンドを呼び出すときに指定するビルド記述子オプションは、EGL コマンド・ファイルにリストされているどのビルド記述子のオプションよりも優先されます。

例

コマンドの例を次に示します。スペースの関係で、1 行のコマンドを複数行に分割しているものもあります。

```
java EGLSDK "commandfile.xml"

java EGLSDK "commandfile.xml"
    -eglpath "c:¥myGroup;h:¥myCorp"

java EGLSDK generate
    -eglpath "c:¥myGroup;h:¥myCorp"
    -generateFile "c:¥myProg.eglpgm"
    -buildDescriptorFile "c:¥myBuild.eglbld"
    -buildDescriptorName myBuildDescriptor

java EGLSDK "myCommand.xml"
    -sqlID myID -sqlPassword myPW
    -destUserID myUserID -destPassword myPass
```

関連する概念

319 ページの『ビルド記述子パーツ』

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

36 ページの『インポート』

323 ページの『マスター・ビルド記述子』

関連するタスク

362 ページの『EGL ソフトウェア開発キット (SDK) からの生成』

関連する参照項目

423 ページの『destPassword』

423 ページの『destUserID』

517 ページの『EGL ビルド・パスおよび eglpath』

436 ページの『sqlID』

438 ページの『sqlPassword』

812 ページの『EGL ステートメントおよびコマンドの構文図』

eglmaster.properties ファイルの形式

eglmaster.properties ファイルは、EGL SDK がマスター・ビルド記述子の名前とファイル・パス名を指定するために使用する Java プロパティー・ファイルです。このプロパティー・ファイルは、EGLSDK コマンドを呼び出す プロセスの CLASSPATH 変数で指定されるディレクトリーに組み込む必要があります。 eglmaster.properties ファイルの形式は以下のとおりです。

```
masterBuildDescriptorName=desc
masterBuildDescriptorFile=path
```

ここで、

desc

マスター・ビルド記述子の名前

path

EGL ファイルの完全修飾パス名。EGL ファイルでは、EGL SDK が使用するマスター・ビルド記述子が宣言されます。

このファイルの内容は、Java プロパティー・ファイルの規則に従う必要があります。スラッシュ (/) か、2 つのバックスラッシュまたは円記号 (¥¥) を使用して、パス名内のファイル名を区切ることもできます。

プロパティー・ファイルでは、**masterBuildDescriptorName** キーワード と **masterBuildDescriptorFile** キーワードの両方を指定する必要があります。そうしないと、eglmaster.properties ファイルは無視されます。

以下に、eglmaster.properties ファイルの内容の例を示します。

```
# Specify the name of the master build descriptor:
masterBuildDescriptorName=MYBUILDDESRIPTOR
# Specify the file that contains the master build descriptor:
masterBuildDescriptorFile=d:/egl/buildddescriptors/master.egl
```

関連する概念

323 ページの『マスター・ビルド記述子』

関連するタスク

326 ページの『Java 生成のためのオプションの選択』

関連する参照項目

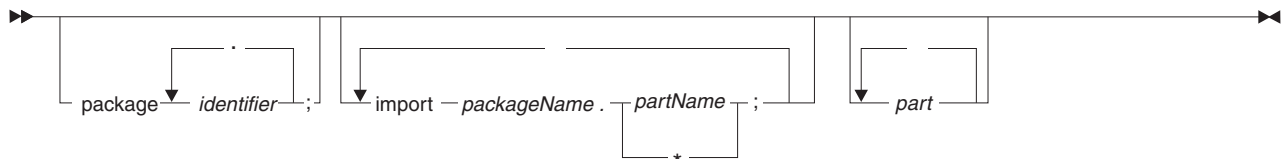
415 ページの『ビルド記述子オプション』

529 ページの『EGLSDK』

548 ページの『マスター・ビルド記述子 plugin.xml ファイルの形式』

EGL ソース形式

それぞれ拡張子 `.egl` を持ち、次のように構成される EGL ソース・ファイルで、ロジック、データ、およびユーザー・インターフェース・パーツを宣言します。



package *identifier*

ファイルがあるパッケージの名前を指定します。各 ID はピリオドで分離します。

概要については、『EGL プロジェクト、パッケージ、およびファイル』を参照してください。

import *packageName*

インポートするパッケージのフルネームを指定します。概要については、『インポート』を参照してください。

partName

インポートする 1 パーツを指定します。

* パッケージのすべてのパーツをインポートすることを示します。

part

EGL ロジック、データ、またはユーザー・インターフェース・パーツの 1 つ。

EGL ファイルでは、パーツの内側または外側に、コメントを含めることができます。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

36 ページの『インポート』

24 ページの『パーツの参照』

19 ページの『パーツ』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

413 ページの『EGL ソース形式の基本レコード・パーツ』

476 ページの『コメント』
512 ページの『EGL ソース形式の DataItem パーツ』
513 ページの『EGL ソース形式の DataTable パーツ』
550 ページの『EGL ソース形式の FormGroup パーツ』
553 ページの『EGL ソース形式の書式パーツ』
570 ページの『EGL ソース形式の関数パーツ』
578 ページの『EGL ソース形式の索引付きレコード・パーツ』
700 ページの『EGL ソース形式のライブラリー・パーツ』
714 ページの『EGL ソース形式の MQ レコード・パーツ』
732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
783 ページの『EGL ソース形式のプログラム・パーツ』
796 ページの『EGL ソース形式の相対レコード・パーツ』
799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』

EGL システム例外

EGL システム例外は、コード全体で使用可能ですが、ほとんどの場合、**onException** ブロックで使用されます。概要については『例外処理』を参照してください。

各 EGL システム例外には、少なくとも以下のフィールドがあります。

code

例外を識別するストリング。例えば、「com.ibm.egl.InvocationException」、またはこれと同等の定数である「SysLib.InvocationException」があります。

description

例外の意味を説明するストリング。

EGL システム例外には、次のものがあります。

SysLib.FileIOException

ファイル・アクセス時に発生したエラーを識別します。メッセージ・キューのリレーショナル・データベースのアクセス時に発生するエラーでは、この例外は生じません。例外特有のフィールドは以下のとおりです。

errorCode

SysVar.ErrorCode でも戻される 8 文字の状況コード。詳細については、『SysVar.ErrorCode』を参照してください。

fileName

アクセスされるファイルの論理名。詳細については、『リソース関連とファイル・タイプ』を参照してください。

SysLib.InvocationException

call ステートメントで発生したエラーを識別します。

例外特有のフィールドは以下のとおりです。

errorCode

SysVar.ErrorCode でも戻される 8 文字の状況コード。詳細については、『SysVar.ErrorCode』を参照してください。

name

呼び出されるプログラムの名前。

SysLib.LobProcessingException

LOB または CLOB 型のフィールドの処理中に発生したエラーを識別します。
例外特有のフィールドは以下のとおりです。

itemName

フィールドの名前。

operation

失敗した EGL システム関数の名前。

resource

フィールドに付加されたファイル (付加されている場合) の名前。

SysLib.MQIOException

MQSeries メッセージ・キューのアクセス時に発生したエラーを識別します。例外特有のフィールドは以下のとおりです。

errorCode

SysVar.ErrorCode でも戻される 8 文字の状況コード。詳細については、『SysVar.ErrorCode』を参照してください。

mqConditionCode

MQSeries API 呼び出しからの完了コード。詳細については、『VGVar.mqConditionCode』を参照してください。

name

アクセスされるキューの論理名。詳細については、『リソース関連とファイル・タイプ』を参照してください。

SysLib.SQLException

リレーショナル・データベースのアクセス時に発生したエラーを識別します。例外特有のフィールドは以下のとおりです。

sqlca

SQL 通信域。詳細については、『SysVar.sqlca』を参照してください。

sqlcode

SQL 戻りコード。詳細については、『SysVar.sqlcode』を参照してください。

sqlErrd

6 エレメントからなる配列で、各エレメントには、最後の SQL I/O オプションから戻された、対応する SQL 通信域 (SQLCA) 値を含みます。詳細については、『VGVar.sqlErrd』を参照してください。

sqlErrmc

JDBC 以外を介してデータベースにアクセスした場合の、sqlcode に関連したエラー・メッセージ。詳細については、『VGVar.sqlErrmc』を参照してください。

sqlState

最近完了した SQL 入出力操作の SQL 状態値。詳細については、『SysVar.sqlState』を参照してください。

sqlWarn

最後の SQL 入出力操作に関して SQL 通信域 (SQLCA) で戻された警告バイトが各エレメントに含まれ、インデックスが SQL SQLCA の説明内の警告番号よりも 1 大きい、11 個のエレメントからなる配列。詳細は、『VGVar.sqlState』を参照してください。

関連する概念

331 ページの『リソース関連とファイル・タイプ』

関連する参照項目

533 ページの『EGL システム例外』

996 ページの『errorCode』

1000 ページの『sqlca』

1001 ページの『sqlcode』

1002 ページの『sqlState』

1014 ページの『mqConditionCode』

1015 ページの『sqlerrd』

1016 ページの『sqlerrmc』

1017 ページの『sqlWarn』

EGL システム制限

EGL ファイル内のパーツの数および階層レベルの数に関しては、EGL では制限が定義されていません。ただし、以下の制限があります。

- 1 つのプログラムで使用できる変数とリテラルは、変数内のフィールドも含めて、32767 個までである。
- 1 つの call ステートメントで 30 個を超える引数を使用できない。これらの制限事項は、引数のサイズ全体に対して適用される。
 - 呼び出しにおいて、**type** プロパティの値が remoteCall または ejbCall の場合、32,567 を超えることはできない。

これらのプロパティは、両方ともリンケージ・オプション・パーツの callLink エレメント内にあります。

- 1 つのフィールドのバイト数は、32767 バイト以内でなければならない。
- 通常、1 つの数値リテラルまたはフィールドは、符号や小数点を除いて 32 桁以内です。ただし、**mathLib.round** 関数を呼び出して作成された結果を受け取るフィールドは、符号や小数点を除いて 31 文字以内でなければなりません。
- 1 つの静的配列の次元は 7 を超えてはならず、エレメントは合計 32,767 個を超えてはならない。
- 動的配列の場合は、次のようになる。
 - 動的配列の次元は 14 を超えてはならない。動的レコード配列の次元数は、レコード構造体の次元数に 1 (レコード配列宣言用) を加えた数です。
 - 動的配列の最大エレメント数は、2,147,483,647 個である。この値は最大サイズが指定されていない場合に有効ですが、割り当て可能なサイズは実行時に使用可能なメモリーによっても制限されます。
 - リモート呼び出し時に渡すことができるすべての引数の合計サイズは、プロトコルでサポートされている最大バッファ・サイズによって制限される。

関連する参照項目

442 ページの『callLink エlement』
914 ページの『round()』
725 ページの『命名規則』
452 ページの『callLink エlementの parmForm』
460 ページの『callLink エlementの type』

式

式とは、プログラムや関数のスクリプトを作成する際に指定する、一連のオペランドや演算子のことを言います。

それぞれの式は、実行時に特定の型の値に解決されます。数式 は数値に解決され、文字列式 は一連の文字に解決され、論理式 は真または偽に解決されます。また、日時式 は日付、時間間隔、時刻、またはタイム・スタンプに解決されます。

式の評価は、一連の優先順位規則に従って行われ、(ある 1 つの優先順位レベル内では) 左から右へ評価されますが、小括弧を使用することにより、強制的に別の順序にすることもできます。ネストされた括弧付きの副次式は、それを囲んでいる括弧付きの副次式の前に評価され、すべての括弧付きの式は、その式全体が評価される前に評価されます。

ある 1 つの評価レベルでは、最初のオペランドが式 (または副次式) の型を決定します。以下の例を考えてください。

```
"A value = " + 1 + 2
```

第 1 オペランドは文字型であり、この式は次の値を持つテキスト式です。

```
"A value = 12"
```

次のような別のテキスト式を考えてみます。

```
"A value = " + (1 + 2)
```

この場合の値は、次のようになります。

```
"A value = 3"
```

関連する参照項目

『日時式』
538 ページの『論理式』
546 ページの『数式』
547 ページの『テキスト式』

日時式

日時式 は、コンテキストに従って、DATE 型、INT 型、INTERVAL 型、TIME 型、または TIMESTAMP 型の値に解決されます。日時式には、以下のいずれかを含める必要があります。

- 以下のいずれかの値を含む変数。
- 日時値を戻す関数呼び出し。次のシステム関数は、文字列リテラルまたは定数から日時値を作成します。
 - **DateTimeLib.dateValue** は、日付を生成します。

- **DateTimeLib.intervalValue** は、間隔を生成します。
- **DateTimeLib.timeValue** は、時刻を作成します。
- **DateTimeLib.timeStampValue** は、タイム・スタンプを作成します。

また、システム関数 **DateTimeLib.extend** は、DATE 型、TIME 型、または TIMESTAMP 型の入力フィールドよりも長い、または短いタイム・スタンプ値を戻します。

次の表は、日時式で有効な算術演算の型を要約したものです。以下に示すように、日時式には、数値を戻す数式が含まれる場合がありますが、一部のケースのみです。

日時式での算術演算

オペランド 1 の型	演算子	オペランド 2 の型	結果の型	コメント
DATE	-	DATE	INT	
DATE	+/-	NUMBER	DATE	
NUMBER	+	DATE	DATE	
TIME STAMP	-	TIMESTAMP	INTERVAL	INTERVAL(dd, ss)。ただし、オペランド 1 とオペランド 2 の両方が以下のいずれかである場合は除きます。 <ul style="list-style-type: none"> • TIMESTAMP(yyyy) • TIMESTAMP(yyyyMM) • TIMESTAMP(MM) 上記の 3 つの場合、結果は INTERVAL(yyyyMM) です。
DATE	-	TIMESTAMP	INTERVAL	INTERVAL(ddssmmffffff)
TIME STAMP	-	DATE	INTERVAL	INTERVAL(ddHHmmssffffff)
TIME STAMP	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	TIMESTAMP	TIMESTAMP	
DATE	+/-	INTERVAL	TIMESTAMP	
INTERVAL	+	DATE	TIMESTAMP	
INTERVAL	+/-	INTERVAL	INTERVAL	オペランド 1 とオペランド 2 はどちらも、年と月、または日と時刻値が必要です。
INTERVAL	*/	NUMBER	INTERVAL	

関連する参照項目

- 407 ページの『代入』
- 853 ページの『dateValue()』
- 854 ページの『extend()』

855 ページの『intervalValue()』
859 ページの『timeValue()』
858 ページの『timeStampValue()』
536 ページの『式』
『論理式』
546 ページの『数式』
726 ページの『演算子と優先順位』
37 ページの『プリミティブ型』
547 ページの『テキスト式』

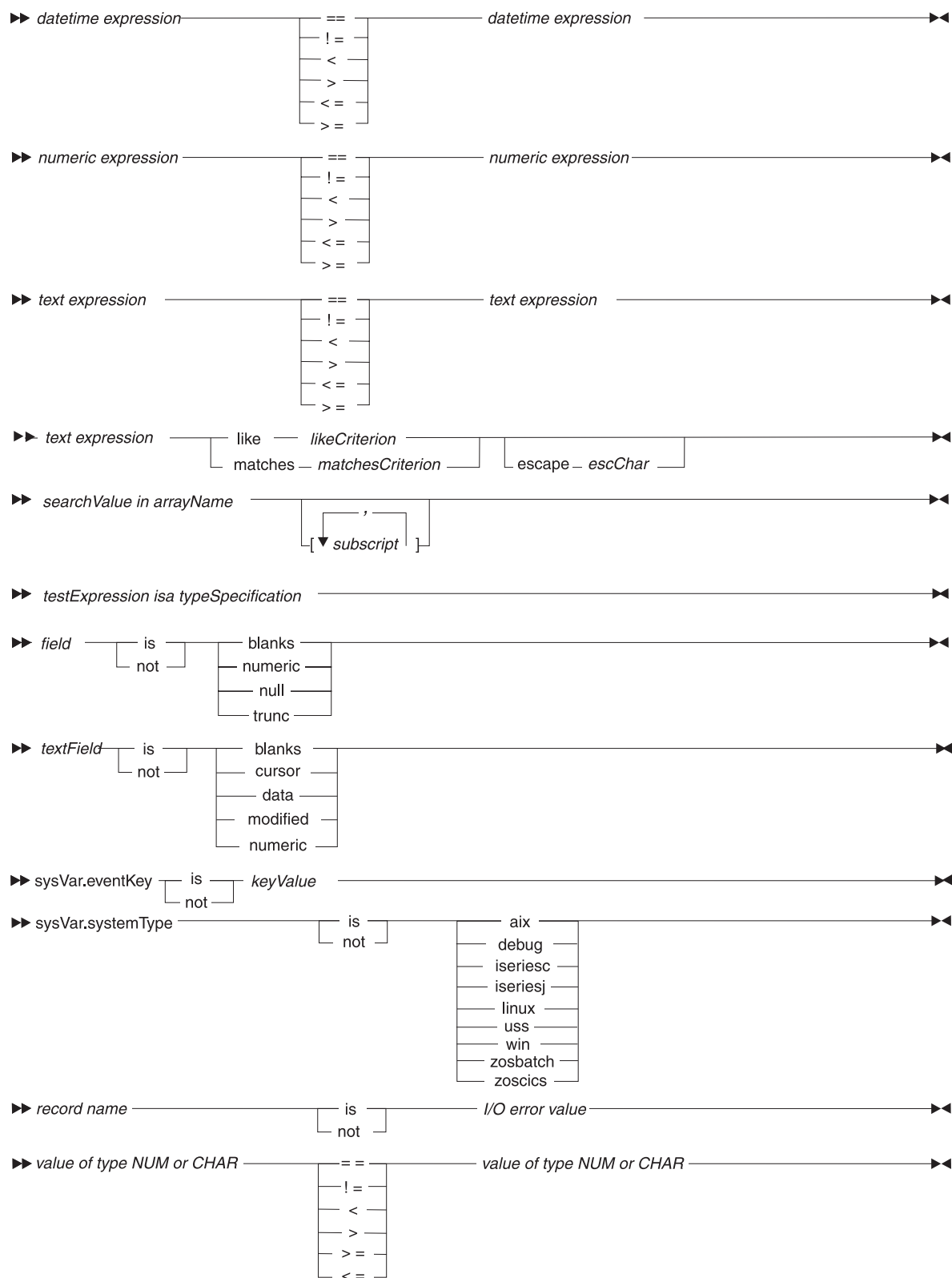
810 ページの『サブストリング』

論理式

論理式 は、真または偽に解決され、**if** ステートメントか **while** ステートメント、あるいは (状況によっては) **case** ステートメントの中で基準として使用されます。

基本論理式

基本論理式は、オペランド、比較演算子、および 2 番目の演算子から構成されます。詳細については、次の構文図と表を参照してください。



第 1 オペランド	比較演算子	第 2 オペランド
日時式	次のいずれか ==, !=, <, >, <=, >=	日時式 最初および 2 番目の式は、互換タイプのものでなければなりません。 日付/時刻を比較する場合、より大符号 (>) は遅い時刻を意味し、より小符号 (<) は早い時刻を意味します。
数式	次のいずれか ==, !=, <, >, <=, >=	数式
ストリング式	次のいずれか ==, !=, <, >, <=, >=	ストリング式
ストリング式	like	<i>likeCriterion</i> 。これは、ストリング式 を左から右へ文字位置ごとに比較する対象となる文字フィールドまたはリテラルです。この機能の用途は、SQL 照会におけるキーワード like の用途によく似ています。 <i>escChar</i> は 1 文字からなるフィールドまたはリテラルで、1 つのエスケープ文字へ解決されます。 詳細については、『 <i>like</i> 演算子
ストリング式	matches	<i>matchCriterion</i> 。これは、ストリング式 を左から右へ文字位置ごとに比較する対象となる文字フィールドまたはリテラルです。この機能の用途は、UNIX または Perl における正規表現 の用途によく似ています。 <i>escChar</i> は 1 文字からなるフィールドまたはリテラルで、1 つのエスケープ文字へ解決されます。 詳細については、『 <i>matches</i> 演算子
NUM 型または CHAR 型の値 (第 2 オペランドの説明を参照)	次のいずれか ==, !=, <, >, <=, >=	NUM 型または CHAR 型の値。次のものを指定できます。 <ul style="list-style-type: none"> NUM 型で小数部を持たないフィールド 整数リテラル CHAR 型のフィールドまたはリテラル
<i>searchValue</i>	in	<i>arrayName</i> 。詳細については、『 <i>in</i> 』を参照してください。

第 1 オペランド	比較演算子	第 2 オペランド
SQL レコードに入っていないフィールド	次のいずれか <ul style="list-style-type: none"> • is • not 	次のいずれか <ul style="list-style-type: none"> • ブランク (文字フィールドの値がブランクのみであるかどうかをテストする場合のみ) • 数値 (CHAR 型または MBCHAR 型のフィールドの値が数値であるかどうかをテストする場合)
SQL レコード内のフィールド	次のいずれか <ul style="list-style-type: none"> • is • not 	次のいずれか <ul style="list-style-type: none"> • ブランク (文字フィールドの値がブランクのみであるかどうかをテストする場合のみ) • null (set ステートメントまたはリレーショナル・データベースからの読み取りによって、フィールドが NULL に設定されたかどうかをテストする場合) • 数値 (CHAR 型または MBCHAR 型のフィールドの値が数値であるかどうかをテストする場合) • trunc (1 バイトまたは 2 バイト文字の値がリレーショナル・データベースからフィールドに最後に読み取られたとき、非ブランク文字の右側が削除されたかどうかをテストする場合) <p>trunc テストが真に解決されるのは、データベースの列がフィールドよりも長い場合に限られます。値がフィールドに移動した後、またはフィールドが NULL に設定された後、テストの値は偽になります。</p>

第 1 オペランド	比較演算子	第 2 オペランド
<i>textField</i> (テキスト書式内のフィールドの名前)	次のいずれか <ul style="list-style-type: none"> • <i>is</i> • <i>not</i> 	次のいずれか <ul style="list-style-type: none"> • ブランク (テキスト・フィールドの値がブランクまたは NULL に限定されているかどうかをテストする場合) <p>ブランクかどうかのテストは、ユーザーの書式への最後の入力に基づき、書式フィールドの現行内容に基づくわけではありません。 <i>is</i> を使用するテストでは、以下の場合には真になります。</p> <ul style="list-style-type: none"> – ユーザーの最後の入力がブランクまたは NULL。 – プログラムの開始以降、またはタイプ <i>set form initial</i> の <i>set</i> ステートメントが実行されて以降、ユーザーがデータを入力していない。 <ul style="list-style-type: none"> • カーソル (ユーザーが指定のテキスト・フィールドにカーソルを残しているかどうかをテストする場合) • データ (ブランクまたは NULL 以外のデータが指定のテキスト・フィールドにあるかどうかをテストする場合) • 変更 (フィールドの変更データ・タグが設定されているかどうかをテストする場合。『変更データ・タグとプロパティ』に説明があります。) • 数値 (CHAR 型または MBCHAR 型のフィールドの値が数値であるかどうかをテストする場合)
<i>ConverseVar.eventKey</i>	次のいずれか <ul style="list-style-type: none"> • <i>is</i> • <i>not</i> 	詳細については、『 <i>ConverseVar.eventKey</i> 』を参照してください。
<i>sysVar.systemType</i>	次のいずれか <ul style="list-style-type: none"> • <i>is</i> • <i>not</i> 	詳細については、『 <i>sysVar.systemType</i> 』を参照してください。 <i>is</i> または <i>not</i> を使用して、 <i>VGLib.getVAGSysType</i> により戻された値をテストすることはできません。
<i>record name</i>	次のいずれか <ul style="list-style-type: none"> • <i>is</i> • <i>not</i> 	レコード編成に適合した入出力エラー値。『入出力エラー値』を参照してください。

次の表は、比較演算子をまとめたものです。比較演算子は、それぞれ真または偽に解決される式の中で使用されます。

演算子	用途
==	= 演算子は、2 つのオペランドの値が同じであることを示します。
!=	!= 演算子は、2 つのオペランドの値が異なっているかどうかを示します。
<	<i>less than</i> 演算子は、1 番目のオペランドの値が 2 番目のオペランドの値より小さいかどうかを示します。
>	<i>greater than</i> 演算子は、1 番目のオペランドの値が 2 番目のオペランドの値より大きいかどうかを示します。
<=	<i>less than or equal to</i> 演算子は、1 番目のオペランドの値が 2 番目のオペランドの値以下であることを示します。
>=	<i>greater than or equal to</i> 演算子は、1 番目のオペランドの値が 2 番目のオペランドの値以上であることを示します。
in	<i>in</i> 演算子は、1 番目のオペランドが、配列を参照する 2 番目のオペランドの値であるかどうかを示します。詳細については、『 <i>in</i> 』を参照してください。
is	<i>is</i> 演算子は、1 番目のオペランドが 2 番目のオペランドのカテゴリに含まれるかどうかを示します。詳細については、前の表を参照してください。
like	<i>like</i> 演算子は、『 <i>like</i> 演算子』で説明されているように、2 つあるオペランドの最初のオペランドに含まれている文字が、第 2 オペランドと一致するかどうかを示します。
matches	<i>matches</i> 演算子は、『 <i>matches</i> 演算子』で説明されているように、2 つあるオペランドの最初のオペランドに含まれている文字が、第 2 オペランドと一致するかどうかを示します。
not	<i>not</i> 演算子は、1 番目のオペランドが 2 番目のオペランドのカテゴリに含まれないかどうかを示します。詳細については、前の表を参照してください。

次の表とそれに続く説明は、オペランドの型に関する互換性の規則を示しています。

第 1 オペランドのプリミティブ型	第 2 オペランドのプリミティブ型
BIN	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
CHAR	CHAR, DATE, HEX, MBCHAR, NUM, TIME, TIMESTAMP
DATE	CHAR, DATE, NUM, TIMESTAMP
DBCHAR	DBCHAR
DECIMAL	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT
HEX	CHAR, HEX
MBCHAR	CHAR, MBCHAR
MONEY	BIN, DECIMAL, FLOAT, MONEY, NUM, NUMC, PACF, SMALLFLOAT

第 1 オペランドのプリミティブ型	第 2 オペランドのプリミティブ型
NUM	BIN、CHAR、DATE、DECIMAL、FLOAT、MONEY、NUM、NUMC、PACF、SMALLFLOAT、TIME
NUMC	BIN、DECIMAL、FLOAT、MONEY、NUM、NUMC、PACF、SMALLFLOAT
PACF	BIN、DECIMAL、FLOAT、MONEY、NUM、NUMC、PACF、SMALLFLOAT
TIME	CHAR、NUM、TIME、TIMESTAMP
TIMESTAMP	CHAR、DATE、TIME、TIMESTAMP
UNICODE	UNICODE

表の説明を以下に詳しく示します。

- 数値型 (BIN、DECIMAL、FLOAT、MONEY、NUM、NUMC、PACF、SMALLFLOAT) の値は、任意の数値型およびサイズを持つ値と比較できます。EGL は、必要に応じて、一時的な変換を実行します。小数部が同等の場合の = 比較は、小数部が全く同じでなくとも、真に評価されます (例えば、1.4 と 1.40)。
- CHAR 型の値は、その各文字が 16 進数文字 (0 から 9、A から F、a から f) の範囲内にある場合にのみ、HEX 型の値と比較できます。EGL は、CHAR 型の値に含まれる小文字を、一時的に大文字に変換します。
- 比較において、文字型 (CHAR、DBCHAR、HEX、MBCHAR、UNICODE) の値が 2 つ含まれており、片方の値のバイト数が他方よりも少ない場合は、一時的な変換によって、短い方の値の右側に埋め込みが行われます。
 - MBCHAR 型の値と比較する場合は、CHAR 型の値の右側に、1 バイトのブランクが埋められます。
 - HEX 型の値と比較する場合は、CHAR 型の値の右側に、2 進ゼロが埋められます。
 - DBCHAR の値の右側には、2 バイトのブランクが埋められます。
 - UNICODE の値の右側には、ユニコードの 2 バイトのブランクが埋められます。
 - HEX 型の値の右側には、2 進ゼロが埋められます。例えば、値 "0A" を 2 バイトに拡張する必要がある場合、比較用の値は、"000A" ではなく、"0A00" となります。
- CHAR 型の値は、次の条件が該当する場合にのみ、NUM 型の値と比較できます。
 - CHAR 型の値に 1 バイトの数字のみが含まれており、その他の文字は含まれていない場合。
 - NUM 型の値の定義に小数点が含まれていない場合。

CHAR 型と NUM 型の比較は、次のように行われます。

- 一時的な変換により、NUM の値が CHAR 形式に変換されます。数字は左そろえられ、必要に応じて 1 バイトのブランクが付加されます。例えば、長さ

- 4 の NUM 型のフィールドが 7 の値を持つ場合、その値は、右側に空白が 3 つ付加された「7」として扱われます。
- フィールドの長さが同じでない場合は、一時的な変換によって、短い方の値の右側に空白が埋められます。
 - 比較では、バイト単位で値が検査されます。次の 2 つの例を考えてください。
 - 長さが 2 で値が「7」（空白を含む）の CHAR 型のフィールドは、長さが 1 で値が 7 の NUM 型のフィールドと同じです。なぜなら、NUM 型のフィールドに基づく一時的なフィールドには、最後の空白も含まれるからです。
 - 値が「8」の CHAR 型のフィールドは、値が 534 の NUM 型のフィールドより大きくなります。なぜなら、ASCII または EBCDIC の検索順序では、「8」が「5」の後にくるからです。

複合論理式

より複雑な式を作成するには、*and* (&&) 演算子または *or* 演算子 (||) のいずれかを使用して、比較的単純な一組の式を結合します。その他に、*not* 演算子 (!) を使用することもできますが、これについては後で説明します。

論理式が、基本論理式を *or* 演算子で結合したものからなる場合、EGL は、優先順位の規則に従って、式を評価します。ただし、基本論理式の 1 つが真に解決された場合は、評価が停止されます。次の例を考えてください。

```
field01 == field02 || 3 in array03 || x == y
```

field01 と field02 が等しくない場合は、評価が続行されます。しかし、値 3 が array03 内にある場合は、式全体が真であることが証明されるため、最後の基本論理式 (x == y) は評価されません。

同様に、基本論理式を *and* 演算子で結合した場合、その 1 つが偽に解決されると、EGL は評価を停止します。次の例では、field01 が field02 と同じでないことが判明した時点で、評価が停止されます。

```
field01 == field02 && 3 in array03 && x == y
```

論理式で括弧をペアで使用すると、次のことが可能になります。

- 評価の順序を変更する。
- 式の意味を明確にする。
- *not* 演算子 (!) を使用可能にする。*not* 演算子は、その直後の論理式の値とは反対のブール値 (真または偽) に解決されます。*not* 演算子の直後の式は、括弧で囲む必要があります。

例

次の例では、value1 には「1」、value2 には「2」のように指定されているとします。

```
/* == 真となる式 */
value5 < value2 + value4

/* == 偽となる式 */
!(value1 is numeric)
```

```

/* == 生成された出力が真となる
    Windows 2000、Windows NT、または
    z/OS UNIX システム・サービス */
sysVar.systemType is WIN || sysVar.systemType is USS

/* == 真となる式 */
(value6 < 5 || value2 + 3 >= value5) && value2 == 2

```

関連する概念

171 ページの『変更データ・タグおよびプロパティ』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

611 ページの『case』
 536 ページの『日時式』
 100 ページの『例外処理』
 536 ページの『式』
 579 ページの『入出力エラー値』
 656 ページの『if, else』
 575 ページの『in 演算子』
 707 ページの『like 演算子』
 707 ページの『like 演算子』
 『数式』
 726 ページの『演算子と優先順位』
 37 ページの『プリミティブ型』
 547 ページの『テキスト式』
 987 ページの『eventKey』
 983 ページの『getVAGSysType()』
 1002 ページの『systemType』
 699 ページの『while』

数式

数式 を解くと数値が生成されます。数式は、assignment ステートメントの右辺など、さまざまな状況で指定されます。数式は、以下のもので構成されます。

- 次のいずれかの数値オペランド
 - 数値が入っている変数。項目の前には、符号を付けることができます。
 - 数値リテラル。数値リテラルは符号で始めることができますが、必ず一連の数字を持つ必要があります。小数点を 1 つのみ含めることができます。
 - 数値を戻す関数呼び出し。

数値リテラルの型は、次のように、そのリテラルの値によって暗黙に指定されます。

- 4 桁以下の整数は SMALLINT 型
- 5 桁から 8 桁までの整数は INT 型
- 9 桁から 18 桁までの整数は BIGINT 型
- 小数点を含んでいる数値は NUM 型

- 2 つの数値オペランドとその間に挟まれる数値演算子。
- 比較的単純な一組の式を数値演算子で組み合わせて形成した、より複雑な式。

数式で括弧を使用すると、評価の順序を変更したり、数式の意味を明確にしたりできます。

以下の例では、`intValue1 = 1`、`intValue2 = 2 ...` であり、それぞれの値には小数部がないものとします。

```
/* == -8。小括弧を使用して * と + の
   通常の優先順位をオーバーライド */
intValue2 * (intValue1 - 5)

/* == -2。最後の演算子として、単項マイナスを使用 */
intValue2 + -4

/* == 1.4。少なくとも 1 つの小数部を持つ項目に
   式を割り当てた場合 */
intValue7 / intValue5

/* == 2。これは剰余であり
   整数値として表現される */
intValue7 % intValue5
```

小括弧が正符号 (+) の使用に及ぼす影響を示した例については、『式』を参照してください。

計算された中間の値で 128 を超えるビット数が必要となる場合は、数式で予期せぬ結果が生成されることがあります。

関連する参照項目

536 ページの『日時式』
 536 ページの『式』
 538 ページの『論理式』
 726 ページの『演算子と優先順位』
 37 ページの『プリミティブ型』
 『テキスト式』

テキスト式

テキスト式 を解くと一連の文字になり、このような式の指定はさまざまな状況で行われます。`assignment` ステートメントの右辺などがこの例です。テキスト式は、以下のいずれかにすることができます。

- 一連の文字を含む変数。
- 文字列リテラル。これは、二重引用符で区切られた一連の文字です。このリテラルは、**STRING** 型です。
- リテラルのサブストリング、または一連の文字が入ったリテラルまたは変数のサブストリング。詳細については、『サブストリング』を参照してください。
- スtringのフォーマットを設定して一連の文字を戻す任意のStringのシステム・ワードを呼び出すこと。詳細については、『Stringのフォーマット設定 (システム・ワード) (*String formatting (system words)*)』を参照してください。
- 前記のような種類の一連の値。それぞれの値同士は正符号 (+) の連結演算子によって区切られます。次の文は、*WebSphere* を `myString` に代入します。

```
myString = "Web" + "Sphere";
```

括弧をプラス (+) 記号で使用する場合は影響を示す例については、『式』を参照してください。

- 一連の文字を戻す、その他すべての関数呼び出し。

エスケープ文字 (\) が前に付いた任意の文字は、テキスト式に含まれます。特に、エスケープ文字を使用すると、以下の文字をリテラル、フィールド、または戻り値に含めることができます。

- 二重引用符 (")
- 円記号 (\\$)
- バックスペース (\b として示される)
- 用紙送り (\f として示される)
- 改行文字 (\n として示される)
- 復帰 (\r として示される)
- タブ (\t として示される)

以下に例を示します。

```
myString = "He said, \"Escape while you can!\"";  
myString2 = "Is a backslash (\\) needed?";
```

リテラルの終わりに引用符が付いていないと、エラーが発生します。

```
myString3 = "Escape is impossible\"";
```

テキスト式内のそれぞれの値は、その式が使用されるコンテキストに有効であることが必要です。例えば、CHAR 型の項目へ代入される式の中で UNICODE 型の項目を使用することはできません。詳細については、『代入』を参照してください。

関連する参照項目

- 407 ページの『代入』
- 536 ページの『日時式』
- 536 ページの『式』
- 538 ページの『論理式』
- 546 ページの『数式』
- 726 ページの『演算子と優先順位』
- 37 ページの『プリミティブ型』
- 810 ページの『サブストリング』

マスター・ビルド記述子 plugin.xml ファイルの形式

マスター・ビルド記述子 plugin.xml ファイルは、ワークベンチがマスター・ビルド記述子の名前とファイル・パス名を指定するために使用する XML ファイルです。これが必要になるのは、ワークベンチから生成している場合または EGLCMD コマンドを使用している場合に、特定のオプションを生成に使用させるためにマスター・ビルド記述子が必要なときだけです。この plugin.xml ファイルは、plugins ディレクトリー内のディレクトリーに置く必要があります。ファイルの形式は以下のとおりです。

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  id="id"
  name="plg"
  version="5.0"
  vendor-name="com">
  <requires />
  <runtime />
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor">
    <masterBuildDescriptor file = "bfil" name = "mas" />
  </extension>
</plugin>
```

ここで、

id プラグインの ID

plg

プラグインの名前

com

会社の名前

bfil

マスター・ビルド記述子を含むファイルのパス名。書式は *project/folder/file* で、Enterprise Developer のワークスペース・ディレクトリーに対して相対的になっています。ここで、

project

プロジェクト・ディレクトリーの名前

folder

プロジェクト・ディレクトリー内のディレクトリーの名前

file

マスター・ビルド記述子が格納されているファイルの名前

mas

マスター・ビルド記述子の名前

このファイルの内容は、XML ファイルの規則に従っている必要があります。パス名の中でファイル名を区切るには、スラッシュ (/) を使用する必要があります。

名前属性とファイル属性の両方を指定する必要があります。そうしないと、plugin.xml ファイルは無視されます。

以下に、plugin.xml ファイルの内容の例を示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Example master BuildDescriptor Plugin -->

<plugin
  id="example.master.BuildDescriptor.plugin"
  name="Example master BuildDescriptor plug-in"
  version="5.0"
  vendor-name="IBM">
  <requires />
  <runtime />
  <!-- ===== -->
  <!-- -->
  <!-- Register the master BuildDescriptor -->
  <!-- -->
  <!-- ===== -->
  <extension point =
    "com.ibm.etools.egl.generation.base.framework.masterBuildDescriptor" >
```



```

        <masterBuildDescriptor file
        = "myProject/myFolder/myFile.eglbuild" name = "masterBD" />
    </extension>
</plugin>

```

関連する概念

319 ページの『ビルド記述子パーツ』

323 ページの『マスター・ビルド記述子』

関連するタスク

360 ページの『ワークベンチのバッチ・インターフェースからの生成』

358 ページの『ワークベンチでの生成方法』

関連する参照項目

415 ページの『ビルド記述子オプション』

518 ページの『EGLCMD』

531 ページの『eglmaster.properties ファイルの形式』

EGL ソース形式の FormGroup パーツ

formGroup パーツは EGL ファイルで宣言します。これについては、『*EGL* ソース形式』で説明しています。このパーツはプライマリー・パーツです。つまり、プライマリー・パーツは、ファイルの最上位にあり、ファイルと同じ名前を持っている必要があります。

プログラムは、そのプログラムの使用宣言で参照されている書式グループに関連付けられている書式しか使用できません。

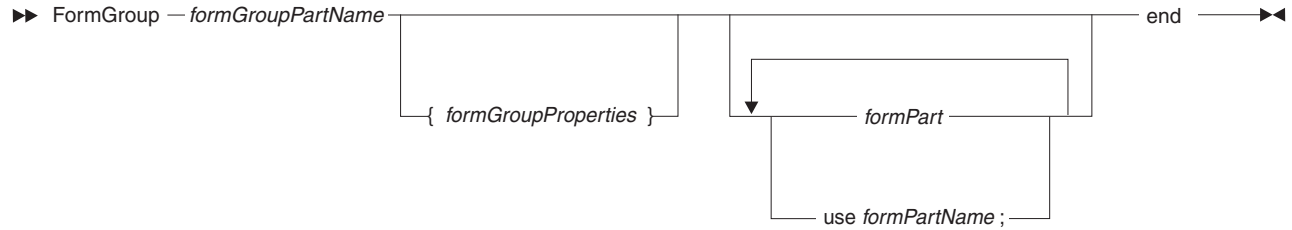
formGroup パーツの例は以下のとおりです。

```

FormGroup myFormGroup
{
    validationBypassKeys = [pf3],
    helpKey = "pf1",
    pfKeyEquate = yes,
    screenFloatingArea
    {
        screenSize = [24,80],
        topMargin = 0,
        bottomMargin = 0,
        leftMargin = 0,
        rightMargin = 0
    },
    printFloatingArea
    {
        pageSize = [60,80],
        topMargin = 3,
        bottomMargin = 3,
        leftMargin = 5,
        rightMargin = 5
    }
}
use myForm01;
use myForm02;
end

```

formGroup パーツのダイアグラムは、次のとおりです。



FormGroup *formGroupName* ... **end**

パーツを書式グループとして識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

formGroupProperties

それぞれがコンマで区切られた、一連のプロパティ。各プロパティについては、後述します。

formPart

テキスト書式または印刷書式。これについては、『EGL ソース形式の書式パーツ』で説明しています。

use *formPartName*

書式グループに組み込まれていない書式へのアクセスを提供する使用宣言。

書式グループ・プロパティを次に示します。

alias

生成された出力の名前に取り込まれるストリング。別名を指定しなかった場合は、formGroup パーツ名 が代わりに使用されます。

validationBypassKeys = [*bypassKeyValue*]

EGL ランタイムに入力フィールドの検証をスキップさせる 1 つ以上のユーザー・キー・ストロークを示します。このプロパティは、プログラムを即座に終了させるキー・ストロークを予約するときに便利です。各 *bypassKeyValue* オプションは以下のとおりです。

pf*n*

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

複数のキー値を指定する場合は、次の例のように、一連の値を大括弧で区切り、各値をコンマで区切ります。

```
validationBypassKeys = [pf3, pf4]
```

helpKey = "*helpKeyValue*"

EGL ランタイムにヘルプ書式をユーザーに表示させる、ユーザー・キー・ストロークを示します。*helpKeyValue* オプションは以下のとおりです。

pf*n*

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

pfKeyEquate = yes, pfKeyEquate = no

ユーザーが大きな番号のファンクション・キー (PF13 から PF24) を押したときに登録されるキー・ストロークが、ユーザーがそのキーよりも 12 だけ小さなファンクション・キーを押したときに登録されるキー・ストロークと同じであるかどうかを指定します。詳細については、『*pfKeyEquate*』を参照してください。

screenFloatingArea { properties }

画面への出力に使用される浮動域を定義します。浮動域の概要については、『書式パーツ』を参照してください。プロパティの詳細については、次のセクションを参照してください。

printFloatingArea { properties }

印刷可能出力に使用される浮動域を定義します。浮動域の概要については、『書式パーツ』を参照してください。プロパティの詳細については、『印刷浮動域のプロパティ』を参照してください。

画面の浮動域のプロパティ

screenFloatingArea の後のプロパティのセットは中括弧 ({ }) で区切られ、各プロパティはコンマで区切られます。プロパティには、次のものがあります。

screenSize = [rows, columns]

オンライン表示域の行および列の数。これには、余白として使用される行または列が含まれます。デフォルトは以下のとおりです。

screenSize=[24,80]

topMargin= rows

表示域の上部にブランクとして残されている行の数。デフォルトは 0 です。

bottomMargin= rows

表示域の下部にブランクとして残されている行の数。デフォルトは 0 です。

leftMargin= columns

表示域の左側にブランクとして残されている列数。デフォルトは 0 です。

rightMargin= columns

表示域の右側にブランクとして残されている列数。デフォルトは 0 です。

印刷浮動域のプロパティ

printFloatingArea の後のプロパティのセットは中括弧 ({ }) で区切られ、各プロパティはコンマで区切られます。プロパティには、次のものがあります。

pageSize = [rows, columns]

印刷可能表示域の行および列の数。これには、余白として使用される行または列が含まれます。印刷浮動域を指定する場合は、このプロパティは必須です。

deviceType = singleByte, deviceType = doubleByte

浮動域宣言が、1 バイト出力 (デフォルト) または 2 バイト出力のどちらをサポートしているプリンターに対するものなのかを指定します。いずれかの書式に DBCHAR 型または MBCHAR 型の項目が含まれている場合は、**doubleByte** を指定してください。

topMargin = *rows*

表示域の上部にブランクとして残されている行の数。デフォルトは 0 です。

bottomMargin = *rows*

表示域の下部にブランクとして残されている行の数。デフォルトは 0 です。

leftMargin = *columns*

表示域の左側にブランクとして残されている列数。デフォルトは 0 です。

rightMargin = *columns*

表示域の右側にブランクとして残されている列数。デフォルトは 0 です。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

163 ページの『FormGroup パーツ』

164 ページの『書式パーツ』

関連する参照項目

532 ページの『EGL ソース形式』

『EGL ソース形式の書式パーツ』

725 ページの『命名規則』

739 ページの『pfKeyEquate』

1020 ページの『使用宣言』

EGL ソース形式の書式パーツ

書式パーツは、*EGL* ソース形式 で記述された EGL ファイル内に宣言します。書式パーツが 1 つの書式グループだけからアクセスされる場合は、書式パーツを *formGroup* パーツに埋め込むことをお勧めします。書式パーツが複数の書式グループからアクセスされる場合は、EGL ファイルのトップレベルに書式パーツを指定する必要があります。

テキスト書式の例を次に示します。

```
Form myTextForm type textForm
{
    formsize= [24, 80],
    position= [1, 1],
    validationBypassKeys=[pf3, pf4],
    helpKey="pf1",
    helpForm="myHelpForm",
    msgField="myMsg",
    alias = "form1"
}

* { position=[1, 31], value="Sample Menu" } ;
* { position=[3, 18], value="Activity:" } ;
* { position=[3, 61], value="Command Code:" } ;

activity char(42)[5] { position=[4,18], protect=skip } ;

commandCode char(10)[5] { position=[4,61], protect=skip } ;

* { position=[10, 1], value="Response:" } ;
response char(228) { position=[10, 12], protect=skip } ;

* { position=[13, 1], value="Command:" } ;
myCommand char(70) { position=[13,10] } ;
```

```

* { position=[14, 1], value="Enter=Run F3=Exit" } ;

myMsg char(70) { position=[20,4] };

end

```

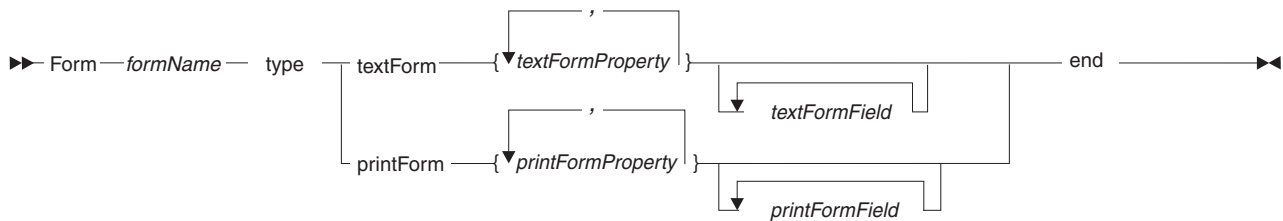
印刷書式の例を次に示します。

```

Form myPrintForm type printForm
{
  formSize= [48, 80],
  position= [1, 1],
  msgField="myMsg",
  alias = "form2"
}
* { position=[1, 10], value="Your ID: " } ;
ID char(70) { position=[1, 30] };
myMsg char(70) { position=[20, 4] };
end

```

書式パーツのダイアグラムは、次のとおりです。



Form formName ... end

パーツを書式として識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

textForm

書式がテキスト書式であることを示します。

textFormProperty

テキスト書式のプロパティ。詳細については、『テキスト書式』を参照してください。

textFormField

テキスト書式のフィールド。詳細については、『Form fields』を参照してください。

printForm

書式が印刷書式であることを示します。

printFormProperty

印刷書式のプロパティ。詳細については、『印刷書式』を参照してください。

printFormField

印刷書式のフィールド。詳細については、『Form fields』を参照してください。

テキスト書式のプロパティ

テキスト書式プロパティは以下のとおりです。

formSize = [rows, columns]

オンライン表示領域の行数および列数。このプロパティは必須です。

列値は、表示領域に水平に表示できる 1 バイト文字の数に相当します。

position = [row, column]

表示領域内の、書式が表示される行と列。このプロパティを省略すると、書式は浮動書式となり、浮動域に表示されます (書式全体を浮動域に表示できる次の空き行に表示されます)。

validationBypassKeys = [bypassKeyValue]

EGL ランタイムに入力フィールドの検証をスキップさせる 1 つ以上のユーザー・キー・ストロークを示します。このプロパティは、プログラムを即座に終了させるキー・ストロークを予約するときに便利です。 *bypassKeyValue* オプションは以下のとおりです。

pf*n*

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

複数のキー値を指定する場合は、次の例のように、値のセットを括弧で括り、各値をコンマで区切ります。

```
validationBypassKeys = [pf3, pf4]
```

helpKey = "*helpKeyValue*"

EGL ランタイムにヘルプ書式をユーザーに表示させる、ユーザー・キー・ストロークを示します。 *helpKeyValue* オプションは以下のとおりです。

pf*n*

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

helpForm = "*formName*"

テキスト書式に固有のヘルプ書式の名前。

msgField = "*fieldName*"

検証エラーの応答として、または `ConverseLib.displayMsgNum` が実行されたことに対する応答としてメッセージを表示するテキスト書式フィールドの名前。

alias = "*alias*"

EGL ランタイムが使用する 8 文字以下の別名。

印刷書式のプロパティ

印刷書式プロパティは以下のとおりです。

formsize = [rows, columns]

オンライン表示領域の行数および列数。このプロパティは必須です。

列値は、表示領域に水平に表示できる 1 バイト文字の数に相当します。

position = [row. column]

表示領域内の、書式が表示される行と列。このプロパティを省略すると、書式は浮動書式となり、浮動域に表示されます (書式全体を浮動域に表示できる次の空き行に表示されます)。

msgField = "fieldName"

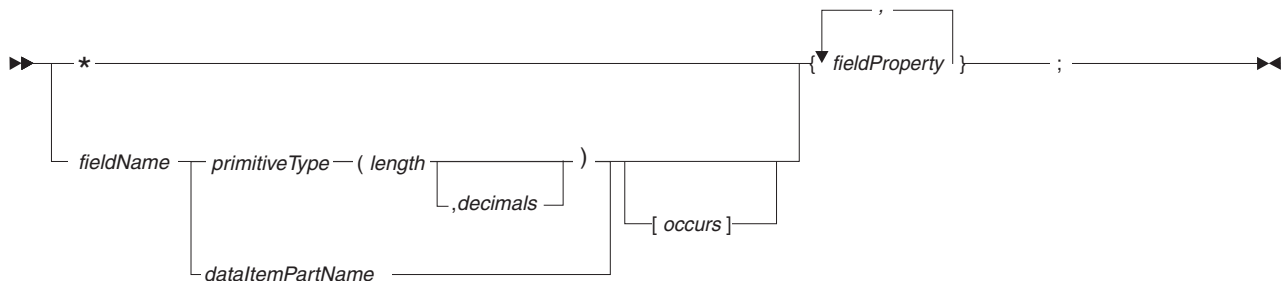
ConverseLib.displayMsgNum が実行されたことに対する応答としてメッセージを表示するテキスト書式フィールドの名前。

alias = "alias"

EGL ランタイムが使用する 8 文字以下の別名。

書式フィールド

書式フィールドのダイアグラムは、次のとおりです。



- * フィールドが定数フィールドであることを示します。定数フィールドは名前を持たず、フィールド固有の **value** プロパティで指定された定数値を持ちます。ユーザーのコード内の文は定数フィールドの値にアクセスできません。

fieldProperty

テキスト書式フィールドのプロパティ。詳細については、『テキスト書式フィールドのプロパティ』を参照してください。

fieldName

フィールド名を指定します。命名の規則については、『命名規則』を参照してください。

ユーザーのコードは名前付きフィールドの値にアクセスできます。名前付きフィールドは変数フィールドとも呼ばれます。

ある行で始まり、別の行で終了する変数フィールドがテキスト書式に含まれる場合、テキスト書式は画面幅が書式幅と等しい画面上にのみ表示できます。

occurs

フィールド配列内の要素数。1 次元配列のみがサポートされています。詳細については、『フィールド配列用』を参照してください。

primitiveType

フィールドに割り当てられるプリミティブ型。この指定は最大長に影響しますが、すべての数値フィールドは NUM 型として生成されます。

DBCHAR 型のフィールドを含む書式は、2 バイト文字セットをサポートするシステムおよびデバイスでのみ使用できます。同様に、MBCHAR 型のフィールドを含む書式は、マルチバイト文字セットをサポートするシステムおよびデバイスでのみ使用できます。

プリミティブ・タイプ FLOAT、SMALLFLOAT、および UNICODE は、テキスト書式または印刷書式ではサポートされていません。

length

フィールドの長さ。フィールドに格納できる文字数または桁数の最大値を表す整数です。

decimals

decimals は、数値タイプ (BIN、DECIMAL、NUM、NUMC、または PACF) に対して指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

dataItemPartName

フィールドの形式のモデルである *dataItem* パーツ名 (『*typeDef*』を参照)。
dataItem パーツは書式パーツから可視でなければなりません (『パーツの参照』を参照)。

テキスト書式フィールドのプロパティ

テキスト書式フィールドでのみ使用できるプロパティについては、後で説明します。より広範囲に使用される以下のプロパティも使用可能です。

- 744 ページの『align』
- 747 ページの『currency』
- 748 ページの『currencySymbol』
- 748 ページの『dateFormat』
- 753 ページの『fillCharacter』
- 755 ページの『isBoolean』
- 758 ページの『lineWrap』
- 759 ページの『lowerCase』
- 759 ページの『masked』
- 763 ページの『numericSeparator』
- 763 ページの『outline』
- 767 ページの『sign』
- 770 ページの『timeFormat』
- 771 ページの『timeStampFormat』
- 772 ページの『upperCase』
- 778 ページの『zeroFormat』

任意のフィールド用

以下のプロパティは、書式の任意のフィールドに対して有用です。

position = [row, column]

フィールドに先行する属性バイトの行と列。このプロパティは必須です。

value = "stringLiteral"

フィールドに表示される文字ストリング。引用符が必要です。

このプロパティは、dataItem パーツの宣言内など、任意の項目に対して指定できます。

注: VisualAge Generator との互換性モードが有効な場合、テキスト書式プロパティ **value** を設定すると、そのプロパティの内容は、ユーザーが書式を戻した後にのみプログラム内で使用できるようになります。この理由により、プログラムで設定した値は、プログラム内の項目に対して有効である必要はありません。

fieldLen = lengthInBytes

フィールドの長さ。フィールドに表示できる 1 バイト文字の数です。この値には先行する属性バイトは含まれません。

数値フィールドに対する **fieldLen** の値は、フィールドに格納可能な最大の数値を表示できる長さに、(数値に小数点以下の桁がある場合は) 小数点を足した大きさになければなりません。CHAR、DBCHAR、MBCHAR、または UNICODE 型のフィールドに対する **fieldLen** の値は、2 バイト文字と SI/SO 文字を考慮した大きさになければなりません。

デフォルトの **fieldLen** は、すべてのフォーマット設定文字を含めて、可能な最大のプリミティブ型の数字を表示するために必要なバイト数です。

変数テキスト・フィールド用

以下のプロパティは、変数テキスト・フィールドに対して有効です。

cursor = no, **cursor** = yes

書式が最初に表示されるときに、画面上のカーソルがフィールドの始めにあるかどうかを示します。書式内の 1 つのフィールドだけについて、カーソル・プロパティを **yes** に設定できます。デフォルトは **no** です。

modified = no, **modified** = yes

ユーザーによる値の変更とは関係なく、プログラムがフィールドを変更されるとみなすかどうかを指示します。詳しくは、『変更データ・タグおよびプロパティ』を参照してください。

デフォルトは **no** です。

protect = no, **protect** = skip, **protect** = yes

ユーザーがフィールドにアクセスできるようにするかを指定します。有効な値は以下のとおりです。

no (変数フィールドのデフォルト)

ユーザーがフィールドの値を上書きできるよう設定します。

skip (定数フィールドのデフォルト)

ユーザーがフィールドの値を上書きできないよう設定します。また、次のいずれかの場合には、カーソルがフィールドをスキップします。

- ユーザーがタブ順序の直前のフィールドを操作しているときに、**Tab** を押したか、そのフィールドに内容を入力した場合。または、

- ユーザーがタブ順序の次のフィールドを操作していて、**Shift + Tab** を押した場合。

yes

ユーザーがフィールドの値を上書きできないよう設定します。

validationOrder = *integer*

検証順序におけるフィールドの位置を示します。フィールドが検証されるデフォルト順序は、画面上に表示されたフィールドの順序 (左から右へ、上から下へ) です。

フィールド配列用

1 次元配列は、テキスト書式および印刷書式でサポートされています。次の例のように、配列宣言の **occurs** プロパティの値は 1 より大きい値です。

```
myArray char(1)[3];
```

配列エレメントは、配列の最初の要素に対して指定された配置を基準にして配置されます。デフォルトの振る舞いでは、要素を縦方向の連続行に配置します。

デフォルトの振る舞いを変更するには、以下のプロパティを使用します。

columns = *numberOfElements*

各行の配列の要素数。デフォルトは 1 です。

linesBetweenRows = *numberOfLines*

配列エレメントを含む各行の間にある行の数。デフォルトは 0 です。

spacesBetweenColumns = *numberOfSpaces*

各配列エレメントの間にあるスペースの数。デフォルトは 1 です。

indexOrientation = *down*, **indexOrientation** = *across*

プログラムが配列の要素を参照する方法を指定します。

- **indexOrientation** を *down* に設定すると、要素の番号は上から下、次に左から右の順になり、ある特定の列内の要素に連続する番号が付けられます。
indexOrientation の値は、デフォルトでは *down* です。
- **indexOrientation** を *across* に設定すると、要素の番号は左から右、次に上から下の順になり、ある特定の行内の要素に連続する番号が付けられます。

配列エレメントのプロパティをオーバーライドすることができます。例えば、次のフィールド宣言では、**cursor** プロパティは `myArray` の 2 番目の要素でオーバーライドされます。

```
myArray char(10)[5]
{position=[4,61], protect=skip, myArray[2] { cursor = yes} };
```

関連する概念

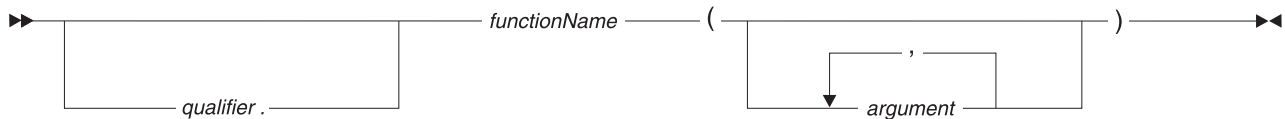
- 171 ページの『変更データ・タグおよびプロパティ』
- 68 ページの『EGL プロパティの概要』
- 166 ページの『印刷書式』
- 24 ページの『パーツの参照』
- 168 ページの『テキスト書式』
- 30 ページの『Typedef』

関連する参照項目

- 70 ページの『フィールド表示プロパティ』
- 71 ページの『フォーマット設定プロパティ』
- 725 ページの『命名規則』
- 55 ページの『NUM』
- 37 ページの『プリミティブ型』
- 847 ページの『displayMsgNum()』
- 72 ページの『検証プロパティ』

関数呼び出し

関数呼び出しは、EGL で生成された関数かシステム関数を実行します。呼び出された関数が終了すると、処理は、その呼び出しの直後にある文へ進むか、(複雑な事例では) 式の中または引数リストの中で必要とされる次の処理へ進みます。



qualifier

次のシンボルのいずれかです。

- 関数が属するライブラリーの名前。
- 関数が属するパッケージの名前。オプションで、ピリオド、および関数が属するライブラリーの名前が続きます。
- *this* (現行プログラム内の関数を識別する)

修飾子が不要な環境の詳細については、『パーツへの参照』を参照してください。

function name

呼び出し関数の名前。

argument

次の値のいずれかです。

- リテラル
- 定数
- 可変長
- より複雑な数式、テキスト式、または日時式 (関数呼び出しやサブストリングを含むことができますが、パラメーターのアクセス修飾子は **IN** でなければなりません)。

引数として EGL 生成関数へ渡される変数への影響は、対応するパラメーターが **IN**、**OUT**、**INOUT** のいずれで修飾されているかによって異なります。詳細については、『関数パラメーター』を参照してください。

呼び出された関数が値を戻す場合、その呼び出しを以下のように使用できます。

- 完全な EGL ステートメントとして使用する (この場合、関数は値を戻さず、直後にセミコロンを伴います)。
- **assignment** ステートメントの中のソース値として使用する。

- 式の中のオペランドとして使用する。
- 関数呼び出しの引数として使用する。

関数呼び出しの中で呼び出された関数は、関数の中、または関数呼び出し自体の中で同じ変数が使用されていると、変数の値が変更される副次作用を起こすことがあります。次の例を考えてみます。ここでは、関数 `Sum` は 3 つの引数の合計を返し、関数 `Increment` は渡された引数に 1 を加算することを想定しています。

```
b INT = 1;
x INT = Sum( Increment(b), b, Increment(b) );
```

`Increment` への引数が関連付けられているパラメーターが `INOUT` で修飾されている場合、上記の文の効果は次のようになります。

- `b = 1`
- 最初 (左端) の `Increment` の呼び出しは、`b` の値を変更し、`Increment` から戻った直後の `b` は 2 になります。
- `Sum` の呼び出しの中で 2 番目の引数は 2 です。
- 2 番目 (右端) の `Increment` の呼び出しは、`b` の値を変更し、`Increment` から戻った直後の `b` は 3 になります。
- `Sum` の実行後に `x` が受け取る値は 7 です。なぜなら、この関数のロジックは、値 2、2、および 3 を使用したからです。

`Sum` の呼び出しの中で、2 番目の引数が関連付けられているパラメーターが `INOUT` で修飾されている場合、この引数の評価は `Increment` の両方の呼び出しの後に行われます。上記のコードの効果は、次のようになります。

- `b = 1`
- 最初 (左端) の `Increment` の呼び出しは、`b` の値を変更し、`Increment` から戻った直後の `b` は 2 になります。
- 2 番目 (右端) の `Increment` の呼び出しは、`b` の値を変更し、`Increment` から戻った直後の `b` は 3 になります。
- `Sum` 内のロジックが実行を開始し、その後でのみ、2 番目の引数へ関連付けられたメモリーが参照されます。そのメモリー内の値は 3 です。
- `Sum` の実行後に `x` が受け取る値は 8 です。なぜなら、この関数のロジックは、値 2、3、および 3 を使用したからです。

一般的な規則として、副次作用は、式が評価されるとき通常の順序を参照することによって識別できます。それは、左から右ですが、小括弧によってオーバーライドできます。`INOUT` を使用すると、次に示すように、さらに複雑になります。

パラメーターのアクセス修飾子が `IN` または `OUT` の場合、『割り当ての互換性』で説明された互換性規則が適用されます。パラメーターのアクセス修飾子が `INOUT` の場合 (またはパラメーターがページ・ハンドラーの `onPageLoad` 関数に含まれる場合)、『参照の互換性』で説明された互換性規則が適用されます。

その他に以下の規則が適用されます。

リテラル

アクセス修飾子が `IN` または `INOUT` の場合、リテラルを引数としてコード化

できます。EGL で生成されたコードは、パラメーター型の一時変数を作成し、その変数を値で初期化して関数に渡します。

固定レコード

引数が固定レコードの場合、パラメーターは固定レコードでなければなりません。

basicRecord 型ではない固定レコードには、以下の規則が適用されます。

- 引数の型とパラメーターの型は同一でなければならない。
- アクセス修飾子は INOUT 型でなければならない。

basicRecord 型の固定レコードに関して、引数およびパラメーターの型は変更できる。

- アクセス修飾子が IN 型の場合、引数のサイズはパラメーターのサイズより大きいとか等しくなければならない。
- アクセス修飾子が OUT 型または INOUT 型の場合、引数のサイズはパラメーターのサイズより小さいとか等しくなければならない。

関連する概念

149 ページの『関数パーツ』

24 ページの『パーツの参照』

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連するタスク

407 ページの『代入』

関連する参照項目

401 ページの『EGL での代入の互換性』

93 ページの『EGL ステートメント』

564 ページの『関数パラメーター』

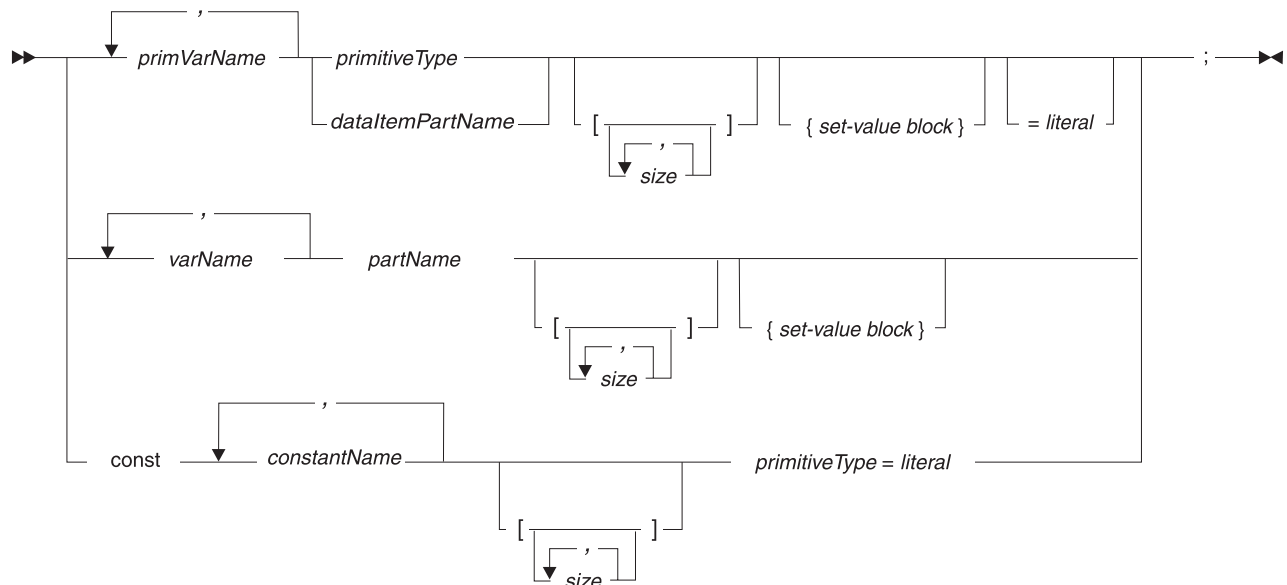
570 ページの『EGL ソース形式の関数パーツ』

37 ページの『プリミティブ型』

795 ページの『EGL での参照の互換性』

関数変数

関数内の各変数ごとの構文図を以下に示します。



primVarName

ローカル・プリミティブ変数の名前を指定します。関数内での詳しい使用法については、『変数および定数の参照』を参照してください。その他の規則については、『命名規則』を参照してください。

primitiveType

プリミティブ・フィールドの型。この型に応じて、以下の情報が必要になります。

- パラメーターの長さ。メモリー領域の文字数または桁数を表す整数です。パラメーターの長さ。メモリー領域の文字数または桁数を表す整数です。
- 一部の数値型には、小数点以下の桁数を表す整数を指定できます。小数点は、データとともに保管されません。
- INTERVAL 型または TIMESTAMP 型の項目には、日時マスクを指定できます。これは、項目の値の特定の位置に意味（「年の桁」など）を割り当てるものです。

dataItemPartName

プログラムに対して可視の *dataItem* パーツ名。可視性についての詳細は、『パーツの参照』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『*Typedef*』を参照してください。

size

配列内の要素数。要素の数を指定すると、その配列は指定した要素数で初期化されます。

set-value block

詳細については、『*EGL* プロパティと設定値のブロック』を参照してください。

= literal

プリミティブ変数の初期値を指定します。

varName

変数の名前。パートに基づく任意の型を使用できます。

partName

プログラムに対して可視になっている、または事前定義されたパーツ名。可視性についての詳細は、『[パーツの参照](#)』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『[Typedef](#)』を参照してください。

const *constantName* *primitiveType*=*literal*

定数の名前、型および値。引用符付きストリング (文字型の場合)、数値 (数値型の場合)、または適切な型付きの値の配列 (配列の場合) を指定します。以下に例を示します。

```
const myString String = "Great software!";
const myArray BIN[] = [36, 49, 64];
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

命名の規則については、『[命名規則](#)』を参照してください。

関連する概念

149 ページの『[関数パーツ](#)』

19 ページの『[パーツ](#)』

24 ページの『[パーツの参照](#)』

62 ページの『[EGL での変数の参照](#)』

812 ページの『[EGL ステートメントおよびコマンドの構文図](#)』

30 ページの『[Typedef](#)』

関連するタスク

570 ページの『[EGL ソース形式の関数パーツ](#)』

関連する参照項目

78 ページの『[配列](#)』

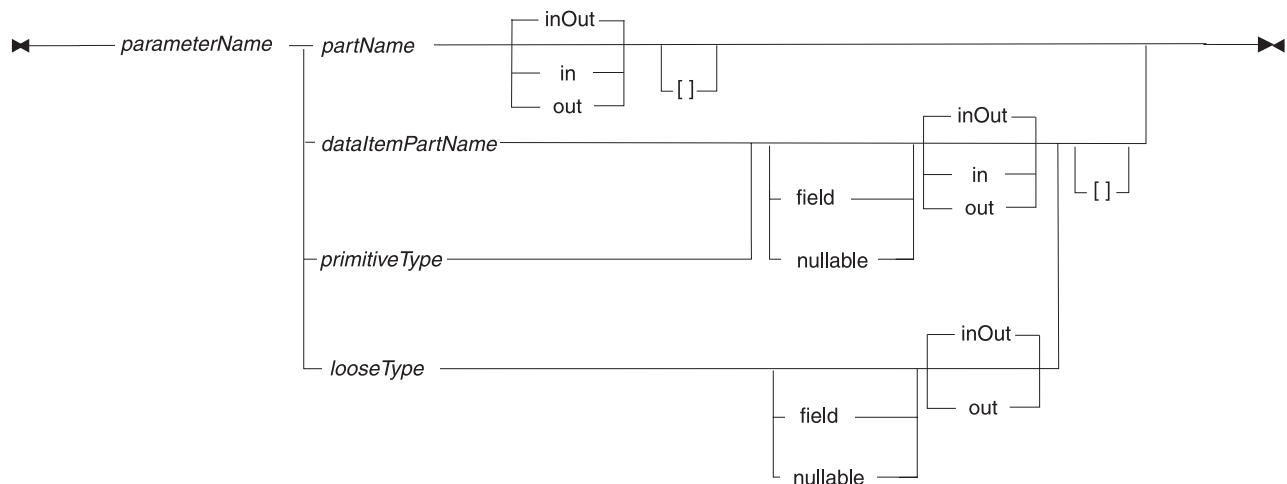
45 ページの『[INTERVAL](#)』

725 ページの『[命名規則](#)』

47 ページの『[TIMESTAMP](#)』

関数パラメーター

関数パラメーターの構文図は、以下のとおりです。



parameterName

パラメーターの名前を指定します。パラメーターの名前は、レコードまたはデータ項目、あるいはレコードまたはデータ項目の配列とすることができます。命名の規則については、『命名規則』を参照してください。

修飾子 **inOut** または **out** を指定すると、パラメーター値に加えた変更を呼び出し側関数で使用できます。これらの修飾子については、後で 568 ページの『inOut とそれに関連する修飾子の影響』のセクションで説明します。

あるパラメーターを含む関数によって呼び出された関数からは、そのパラメーターは可視ではありません。ただし、パラメーターを引数として他の関数に渡すことができます。

大括弧 ([]) で終了するパラメーターは動的配列であり、その他の仕様はその配列の各要素の性質を宣言します。

inOut

関数は引数値を入力として受け取り、呼び出し側は関数の終了時に、パラメーターに加えられた変更があれば、それを受け取ります。ただし、引数がリテラルまたは定数である場合、引数は修飾子 **in** が有効である場合とまったく同じように扱われます。

inOut 修飾子が必要になるのは、パラメーターが項目であり、修飾子 **field** を指定した場合です。後者の修飾子は、パラメーターが *blanks* や *numeric* など、テスト可能な書式フィールド属性を持つことを示します。

パラメーターがレコード (固定レコードではない) の場合は、**inOut** 修飾子のみが有効です。

パラメーターが固定レコードの場合は、以下の規則が適用されます。

- そのレコードを使用して現行の関数 (または現行の関数によって呼び出される関数) の中でファイルまたはデータベースにアクセスする意図がある場合は、**inOut** 修飾子を指定するか、デフォルトでその修飾子を受け入れる必要があります。
- レコード・タイプが引数とパラメーターについて同じである (例えば、両方ともシリアル・レコードである) 場合、ファイルの終わりの状況など、レコード固有の状態情報が呼び出し側に戻されます。ただし、**inOut** 修飾子が有効である場合に限られます。

inOut 修飾子が有効な場合、『EGL での参照の互換性』で説明しているように、関連する引数はパラメーターと参照の互換性がなければなりません。

in 関数は引数値を入力として受け取りますが、呼び出し側は、パラメーターに加えられた変更の影響を受けません。

field 修飾子を持つ項目に **in** 修飾子を使用することはできません。また、(固定レコード以外の) レコードや、現行の関数の中、または現行の関数によって呼び出される関数の中でファイルまたはデータベースにアクセスするために使用される固定レコードに、**in** 修飾子を指定することはできません。

out

関数は引数値を入力として受け取らず、入力値は、『データの初期化』で説明されている規則に従って初期化されます。パラメーター値は、関数が結果を戻す時に引数に割り当てられます。

引数がリテラルまたは定数である場合、引数は修飾子 **in** が有効である場合とまったく同じように扱われます。

field 修飾子が付いたパラメーターに **out** 修飾子を使用することはできません。また、レコードや、現行の関数の中または現行の関数によって呼び出される関数の中でファイルまたはデータベースにアクセスするために使用される固定レコードに、**out** 修飾子を指定することはできません。

partName

関数に可視であり、パラメーターの **typedef** (形式のモデル) として機能するレコード・パーツ。可視のパーツの詳細については、『パーツの参照』を参照してください。

固定レコードに対する入力または出力 (I/O) には、以下のことが当てはまります。

- 同一プログラムの別の関数から渡された固定レコードには、レコードがパラメーターと同じレコード・タイプである場合に限り、入出力エラー値 *endOfFile* などのレコードの状態が含まれます。同様に、レコードの状態の変更は呼び出し側に戻されるので、レコード・パラメーターに対して入出力を実行する場合、現在の関数、呼び出し側、または現在の関数によって呼び出される関数において、そのレコードのテストが行われることがあります。

ライブラリー関数はレコードの状態を受け取りません。

- 固定レコードに対して行われる入出力操作では、引数に指定されたレコード・プロパティーではなく、パラメーターに指定されたレコード・プロパティーが使用されます。
- **indexedRecord**、**mqRecord**、**relativeRecord**、または **serialRecord** タイプの固定レコードの場合、レコード宣言と関連付けられたファイルまたはメッセージ・キューは、プログラム・リソースではなく実行単位リソースとして扱われます。レコード・プロパティー **fileName** (または **queueName**) が同じ値を持つ場合、ローカル・レコード宣言は同じファイル (またはキュー) を共有します。実行単位のファイルまたはキューと関連付けるレコード数にかかわらず、物理ファイルを一度に 1 つずつファイルまたはキュー名に関連付けることができ、EGL は、ファイルをクローズし、再オープンすることによりこの規則を実行します。

dataItemPartName

関数に可視であり、パラメーターの `typedef` (形式のモデル) として機能する `dataItem` パーツ。

primitiveType

プリミティブ・フィールドの型。この型に応じて、以下の情報が必要になります。

- パラメーターの長さ。メモリー領域の文字数または桁数を表す整数です。
- 一部の数値型には、小数点以下の桁数を表す整数を指定できます。小数点は、データとともに保管されません。
- INTERVAL 型または TIMESTAMP 型の項目には、日時マスクを指定できます。これは、項目の値の特定の位置に意味 (「年の桁」など) を割り当てるものです。

looseType

`loose type` は関数パラメーターのためにのみ使用する特殊な種類の基本タイプです。パラメーターが、ある範囲の引数長さを受け入れる場合に、このタイプを使用します。利点は、関数を繰り返し呼び出し、毎回異なる長さの引数を渡すことができる点です。

有効な値は以下のとおりです。

- CHAR
- DBCHAR
- HEX
- MBCHAR
- NUMBER
- UNICODE

パラメーターがプリミティブ型と長さの数値を受け入れるようにする場合、`loose type` として `NUMBER` を使用します。この場合、パラメーターに渡される数値は小数点以下の桁を持たない必要があります。

パラメーターが特定のプリミティブ型であるが、任意の長さのストリングを受け入れるようにする場合は、`CHAR`、`DBCHAR`、`MBCHAR`、`HEX`、または `UNICODE` を `loose type` として指定し、引数が対応するプリミティブ型であることを確認します。

`loose type` のパラメーターに対する、関数内のステートメントの操作の結果は、引数の定義によって決定します。

`loose type` は、ライブラリーに宣言される関数には使用できません。

プリミティブ型の詳細については、『プリミティブ型』を参照してください。

field

パラメーターが `blanks` または `numeric` などの書式フィールド属性を持つことを示します。これらの属性は論理式でテストすることができます。

field 修飾子を使用できるのは、**inOut** 修飾子を指定した場合か、デフォルトで **inOut** 修飾子を受け入れた場合だけです。

field 修飾子は、`nativeLibrary` 型のライブラリー内の関数パラメーターには使用できません。

nullable

パラメーターの以下の特性を示します。

- パラメーターが `NULL` 設定可能である。
- 論理式で切り捨てや `NULL` があるかどうかをテストするために必要な状態情報にパラメーターでアクセスする。

パラメーターに渡される引数が `SQL` レコード内の構造体項目である場合にのみ、**nullable** 修飾子は意味があります。以下の規則が適用されます。

- 項目プロパティ **isNullable** が `yes` に設定されている場合にのみ、パラメーターは、`NULL` 設定したり、`NULL` があるかどうかをテストすることができます。
- **isNullable** の値にかかわらず、切り捨てをテストする機能が使用可能です。に設定されている場合にのみ、切り捨てをテストする機能が使用可能です。
- **nullable** は、修飾子 **inOut**、**in**、または **out** が有効であるかどうかに関係なく指定できます。

inOut とそれに関連する修飾子の影響

修飾子 **inOut**、**out**、および **in** をよく理解するために、次の例を検討してください。この例では、さまざまな実行ポイントで、さまざまな変数の値を（コメントの中で）示しています。

```
program inoutpgm
a int;
b int;
c int;

function main()
a = 1;
b = 1;
c = 1;

func1(a,b,c);

// a = 1
// b = 3
// c = 3
end

function func1(x int in, y int out, z int inout)
// a = 1          x = 1
// b = 1          y = 0
// c = 1          z = 1

x = 2;
y = 2;
z = 2;

// a = 1          x = 2
// b = 1          y = 2
// c = 2          z = 2

func2();
func3(x, y, z);
// a = 1          x = 2
// b = 1          y = 3
```

```

// c = 3          z = 3

end

function func2()
// a = 1
// b = 1
// c = 2

end

function func3(q int in, r int out, s int inout)
// a = 1          x = 未解決    q = 2
// b = 1          y = 未解決    r = 2
// c = 2          z = 未解決    s = 2

q = 3;
r = 3;
s = 3;

// a = 1          x = 未解決    q = 3
// b = 1          y = 未解決    r = 3
// c = 3          z = 未解決    s = 3

end

```

関連する概念

[149 ページの『関数パーツ』](#)
[151 ページの『basicLibrary タイプのライブラリー・パーツ』](#)
[151 ページの『basicLibrary タイプのライブラリー・パーツ』](#)
[19 ページの『パーツ』](#)
[24 ページの『パーツの参照』](#)
[62 ページの『EGL での変数の参照』](#)
[30 ページの『Typedef』](#)

関連する参照項目

[413 ページの『EGL ソース形式の基本レコード・パーツ』](#)
[511 ページの『データの初期化』](#)
[532 ページの『EGL ソース形式』](#)
[570 ページの『EGL ソース形式の関数パーツ』](#)
[578 ページの『EGL ソース形式の索引付きレコード・パーツ』](#)
[45 ページの『INTERVAL』](#)
[538 ページの『論理式』](#)
[714 ページの『EGL ソース形式の MQ レコード・パーツ』](#)
[725 ページの『命名規則』](#)
[37 ページの『プリミティブ型』](#)
[795 ページの『EGL での参照の互換性』](#)
[796 ページの『EGL ソース形式の相対レコード・パーツ』](#)
[799 ページの『EGL ソース形式のシリアル・レコード・パーツ』](#)
[804 ページの『EGL ソース形式の SQL レコード・パーツ』](#)
[47 ページの『TIMESTAMP』](#)

EGL ソース形式の関数パーツ

EGL ファイルに関数を宣言することができます。詳細については、『EGL ソース形式』に説明します。

以下の例では、2 つの組み込み関数を含むプログラム・パーツ、スタンドアロンの関数、およびスタンドアロンのレコード・パーツを示します。

```
Program myProgram(employeeNum INT)
{includeReferencedFunctions = yes}

// プログラム・グローバル変数
employees record_ws;
employeeName char(20);

// 必要組み込み関数
Function main()

    // 従業員名の初期化
    recd_init();

    // 渡された employeeNum に基づいて
    // 目的の従業員名を取得
    employeeName = getEmployeeName(employeeNum);
end

// 別の組み込み関数
Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
end

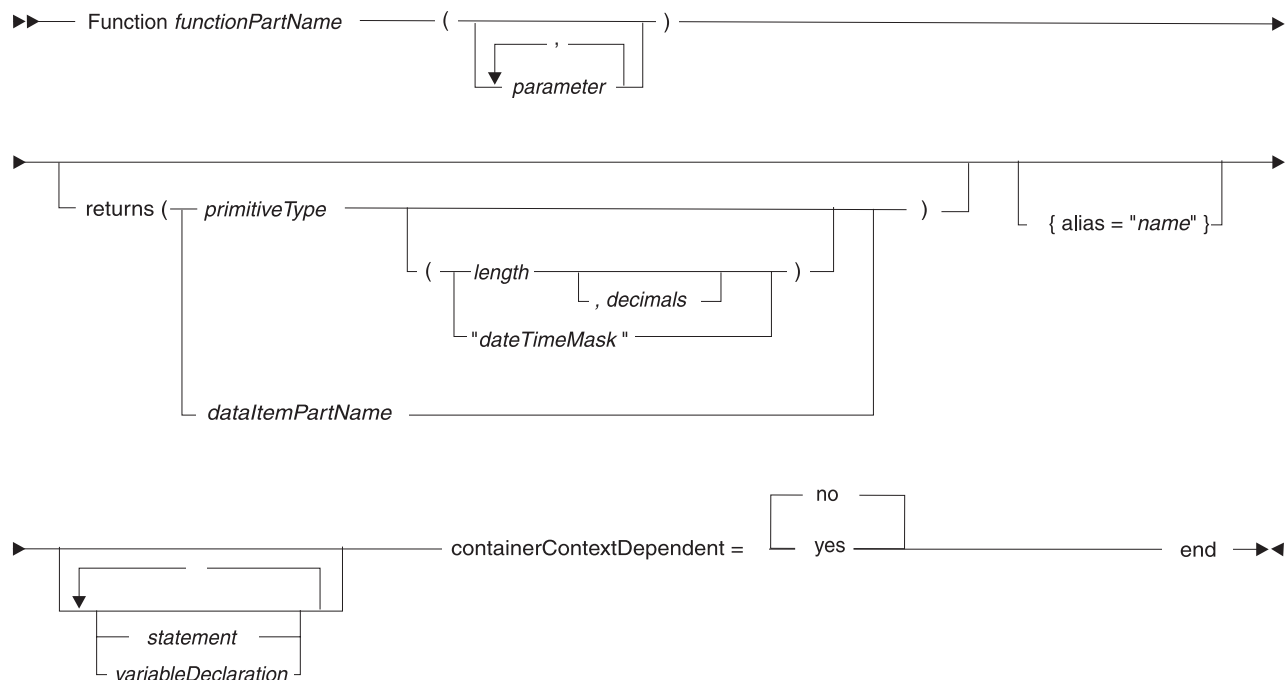
// スタンドアロンの関数  Function getEmployeeName(employeeNum INT) returns
(CHAR(20))

    // ローカル変数
    index BIN(4);
    index = syslib.size(employees.name);
    if (employeeNum > index)
        return("Error");
    else
        return(employees.name[employeeNum]);
    end

end

// 従業員の typedef として機能するレコード・パーツ
Record record_ws type basicRecord
    10 name CHAR(20)[2];
end
```

関数パーツの構文図は、以下のとおりです。



Function *functionPartName* ... end

パーツを関数として識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

parameter

関数全体で使用でき、呼び出し元の関数から値を受け取るメモリー領域となるパラメーター。パラメーターを宣言するために使用する構文については、『関数パラメーター』を参照してください。

returns (*returnType*)

関数が呼び出し側に戻すデータを記述します。戻りの型の特性は、呼び出し側関数内の値を受け取る変数と同じでなければなりません。

{alias = name}

関数が `nativeLibrary` 型のライブラリー内にある場合にのみ有効です。このコンテキストでは、*name* は DLL ベースの関数名で、EGL 関数名にデフォルト設定されます。DLL ベースの関数を使用して EGL 関数に名前を付ける際に検証エラーが発生した場合、**alias** プロパティを明示的に設定してください。

dataItemPartName

関数に可視であり、戻り値の `typedef` (形式のモデル) として機能する `dataItem` パーツ。

primitiveType

呼び出し側に戻されるデータのプリミティブ型。

length

呼び出し側に戻されるデータの長さ。*length* は、戻される値の文字数または桁数を表す整数です。

decimals

一部の数値型には、*decimals* を指定できます。これは、小数点以下の桁数を表

す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

"dateTimeMask"

TIMESTAMP 型および INTERVAL 型には、*"dateTimeMask"* を指定できます。これは、日時値の特定の位置に意味（「年の桁」など）を割り当てるものです。このマスクは、データと一緒に格納されません。

statement

EGL ステートメント。詳細については、『EGL ステートメント』に説明します。ほとんどはセミコロンで終了します。

variableDeclaration

『関数変数』で説明されている変数宣言。

containerContextDependent

宣言されている関数によって呼び出される関数を解決するために使用される名前空間を拡張するかどうかの指示。デフォルトは *no* です。

このインディケーターは、VisualAge Generator からマイグレーションされたコードの中で使用します。詳細については、『*containerContextDependent*』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 149 ページの『関数パーツ』
- 36 ページの『インポート』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 19 ページの『パーツ』
- 24 ページの『パーツの参照』
- 62 ページの『EGL での変数の参照』
- 812 ページの『EGL ステートメントおよびコマンドの構文図』
- 30 ページの『Typedef』

関連する参照項目

- 78 ページの『配列』
- 505 ページの『containerContextDependent』
- 93 ページの『EGL ステートメント』
- 560 ページの『関数呼び出し』
- 564 ページの『関数パラメーター』
- 562 ページの『関数変数』
- 45 ページの『INTERVAL』
- 579 ページの『入出力エラー値』
- 725 ページの『命名規則』
- 37 ページの『プリミティブ型』
- 47 ページの『TIMESTAMP』

生成される出力

生成される出力は以下の表のとおりです。各種の出力ファイルに付けられる名前の詳細については、『生成される出力 (参照)』を参照してください。

出力の型	用途	生成タイプ
ビルド計画	ターゲット・プラットフォームで実行されるコード準備ステップをリストする	Java または Java ラッパー
Enterprise JavaBean (EJB) セッション Bean	EJB コンテナで実行される	Java ラッパー
Java プログラムと関連クラス	J2EE の外側で、あるいは J2EE クライアント・アプリケーション、Web アプリケーション、または EJB コンテナのコンテキストで実行する	Java
Java ラッパー	EGL 以外で生成された Java コードから、EGL で生成されたプログラムを呼び出す	Java ラッパー
J2EE 環境ファイル	Java デプロイメント記述子に挿入するためのエントリーを提供する	Java
ライブラリー (生成された出力)	その他の生成された出力が使用する関数および値を提供する	Java
リンケージ・プロパティ・ファイル	EGL で生成された Java コードから呼び出しが実行される方法を示す (ただし、生成時でなくデプロイメント時に最終決定される場合のみ)	Java または Java ラッパー
ページ・ハンドラー・パーツ	Web ページとのユーザーの実行時対話を制御する出力を生成する	Java
プログラム・プロパティ・ファイル	J2EE 以外の Java プロジェクトで Java プログラムをデバッグしている場合にのみアクセス可能な形式の、Java ランタイム・プロパティが含まれている	Java
結果ファイル	ターゲット・プラットフォームで行われたコード準備ステップの状況情報を提供する	Java または Java ラッパー

関連する概念

1 ページの『EGL の紹介』

357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』

377 ページの『Java ランタイム・プロパティ』

10 ページの『ランタイム構成』

関連するタスク

355 ページの『EGL 出力のビルド』

関連する参照項目

『生成される出力 (参照)』

生成される出力 (参照)

EGL 生成の出力は、通常、Java、または Java ラッパーのいずれを生成しているかによって異なります。次の表に、特定の EGL パーツ以外から生成される出力のファイル名を示します。

出力の型	ファイル名
356 ページの『ビルド計画』	<i>aliasBuildPlan.xml</i>

343 ページの『Enterprise JavaBean (EJB) セッション Bean』	<i>aliasEJBHome.java</i> (ホーム・インターフェースの場合)、 <i>aliasEJB.java</i> (リモート bean インターフェースの場合)、および <i>aliasEJBBean.java</i> (bean 実装の場合)
388 ページの『J2EE 環境ファイル』	<i>alias-env.txt</i>
380 ページの『プログラム・プロパティー・ファイル』	<i>alias.properties</i>
357 ページの『結果ファイル』	<i>alias_Results_timeStamp.xml</i>

alias

プログラム・パーツで指定される別名。別名を指定しない場合は、プログラム・パーツ名が使用されます。ただし、プログラム・パーツ名は、ランタイム環境で許される最大文字数に (必要に応じて) 切り捨てられます。

alias のその他の特性は、出力の種類によって決まります。

- Java プログラムを生成する場合、*alias* の大文字小文字はソース・コードのものがそのまま使用されます。
- Java ラッパーを生成する場合、ラッパーおよび EJB セッション Bean の命名規則は以下のようになります。
 - *alias* の最初の文字は大文字です。
 - 後続の文字はすべて小文字です。ただし、アンダースコアやハイフンは削除され、その後続の文字は大文字になります。

timeStamp

ファイルが作成された日時。この形式には、開発オペレーティング・システムの設定値が反映されます。

ファイル名の詳細については、適切な参照トピックを参照してください。

-
- 727 ページの『Java プログラム生成の出力』
- 728 ページの『Java ラッパー生成の出力』

関連する概念

356 ページの『ビルド計画』

343 ページの『Enterprise JavaBean (EJB) セッション Bean』

572 ページの『生成される出力』

351 ページの『生成』

388 ページの『J2EE 環境ファイル』

380 ページの『プログラム・プロパティー・ファイル』

357 ページの『結果ファイル』

関連する参照項目

727 ページの『Java プログラム生成の出力』

728 ページの『Java ラッパー生成の出力』

「Generation Results (生成結果)」ビュー

「Generation Results (生成結果)」ビューには、ワークベンチで実行される生成結果としてコード準備メッセージが表示されます。これらのメッセージは、エラー、警告、または情報を示すメッセージです。このビューを使用できるのは、ワークベンチから生成している場合のみです。形式は以下のとおりです。

msgid message

msgid

メッセージ ID です。例えば、IWN.VAL.4610.e は Enterprise Developer 検証エラー番号 4610 のメッセージ ID です。

message

メッセージのテキストです。

生成結果は、プライマリー・パーツ (プログラム、ページ・ハンドラー、書式グループ、データ・テーブル、ライブラリー) ごとに別々のタブでビューに表示されます。結果は、検証結果と生成結果の組み合わせで示される場合もあります。

このビューはいつでも開くことができますが、データが表示されるのは出力の生成が済んでからです。

関連する概念

9 ページの『開発過程』

572 ページの『生成される出力』

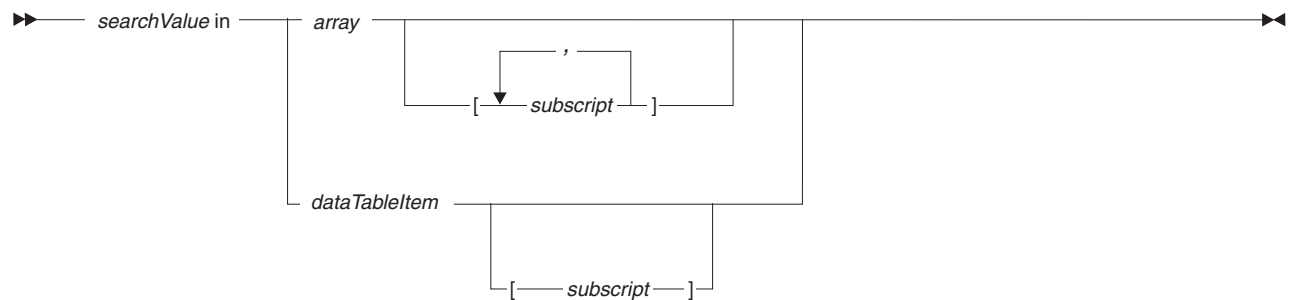
351 ページの『生成』

関連する参照項目

573 ページの『生成される出力 (参照)』

in 演算子

演算子 **in** は、以下の形式で基本的な論理式に使用される 2 項演算子です。



searchValue

リテラルまたは項目。ただし、システム変数ではありません。

array 1 次元または多次元配列。多次元配列が指定された場合でも、演算子 **in** の演算は、その中の 1 次元配列で実行されます。

subscript

整数、または整数に解決される項目 (またはシステム変数)。添え字の値は、配列内の特定のエレメントを参照する指標です。

配列の添え字として使用される項目そのものには、配列エレメントを使用できません。以下のそれぞれの例では、 `myItemB[1]` が添え字と配列エレメントの両方になっています。そのため、以下の構文は無効 です。

```
/* 次の構文は無効です */
myItemA[myItemB[1]]

// 次の構文は無効ですが
// myItemB が myItemB[1] であるため
// 1 次元配列の最初のエレメントは以下になります
myItemA[myItemB]
```

dataTableItem

`dataTable` 項目の名前。この項目は、データ・テーブルの中の列を表します。 **in** 演算子は、列が 1 次元の配列であるかのようにその列と対話します。

論理式は、生成されたプログラムが検索値を検出した場合に真に解決されます。検索は、最後の配列添え字によって識別されるエレメントから開始されます。配列 が 1 次元配列である場合、最後の添え字はオプションで、デフォルトでは 1 に設定されます。配列 が多次元配列である場合は、以下のことが当てはまります。

- 添え字は各次元ごとに存在する必要がある
- 生成されるプログラムでが、最後の添え字以外の一連の添え字によって識別される 1 次元配列が検索される
- 検索は、最後の配列添え字によって識別されるエレメントから開始される

1 次元配列と多次元配列のどちらの場合でも、検索は、調査されている 1 次元配列の最後のエレメントで終了します。

in を含む論理式は、以下の場合に偽に解決されます。

- 検索値が見付からない
- 最後の添え字値が、検索している 1 次元配列内のエントリーの数よりも大きい

基本的な論理式が真に解決されると、演算子 **in** は、システム変数 **sysVar.arrayIndex** を、検索値が含まれているエレメントの添え字値に設定します。式が偽に解決されると、演算子は **sysVar.arrayIndex** をゼロに設定します。

1 次元配列の場合の例

構造体項目 `myString` が、3 文字の配列という副構造を持つと想定します。

```
structureItem name="myString" length=3
structureItem name="myArray" occurs=3 length=1
```

次の表は、`myString` が "ABC" である場合の演算子 **in** の結果を示すものです。

論理式	式の値	sysVar. ArrayIndex の 値	コメント
"A" in myArray	真	1	1 次元配列の添え字は、デフォルトで 1 になる
"C" in myArray[2]	真	3	検索は 2 番目のエレメントから開始される

論理式	式の値	sysVar. ArrayIndex の 値	コメント
"A" in myArray[2]	偽	0	検索は最後のエレメントで終了する

多次元配列の場合の例

配列 myArray01D が、3 文字の配列という副構造を持つと想定します。

```
structureItem name="myArray01D" occurs=3 length=3
structureItem name="myArray02D" occurs=3 length=1
```

この例では、myArray01D は 1 次元配列で、各エレメントには、3 文字の配列という副構造を持つ文字列が含まれています。また myArray02D は 2 次元配列で、各エレメント (myArray02D[1,1] など) には 1 文字が含まれています。

myArray01D の内容が "ABC"、"DEF"、および "GHI" の場合、myArray02D の内容は以下ようになります。

```
"A"  "B"  "C"
"D"  "E"  "F"
"G"  "H"  "I"
```

次の表は、演算子 **in** の結果を示すものです。

論理式	式の値	sysVar. ArrayIndex の 値	コメント
"DEF" in myArray01D	真	2	1 次元配列への参照には添え字は必要ない。デフォルトで、検索は最初のエレメントから開始されます
"C" in myArray02D[1]	--	--	多次元配列への参照には、各次元の添え字が含まれていなければならないため、この式は無効
"I" in myArray02D[3,2]	真	3	検索は 3 行目の 2 番目のエレメントから開始される
"G" in myArray02D[3,2]	偽	0	検索は、調査されている行の最後のエレメントで終了する
"G" in myArray02D[2,4]	偽	0	2 番目の添え字が、検索を実行できる列数より大きく設定されている

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

78 ページの『配列』

538 ページの『論理式』
726 ページの『演算子と優先順位』
993 ページの『arrayIndex』

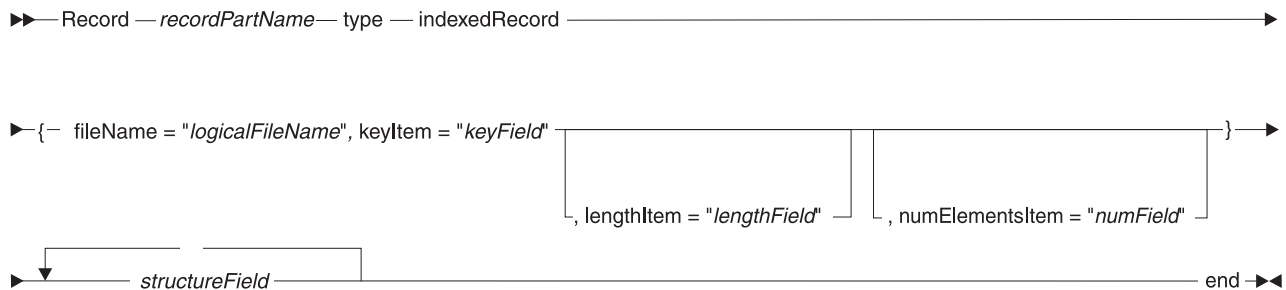
EGL ソース形式の索引付きレコード・パーツ

EGL ファイルでタイプ `indexedRecord` のレコード・パーツを宣言します。これについては、『EGL ソース形式』で説明しています。

索引付きレコード・パーツの例を次に示します。

```
Record myIndexedRecordPart type indexedRecord
{
    fileName = "myFile",
    keyItem  = "myKeyItem"
}
10 myKeyItem CHAR(2);
10 myContent CHAR(78);
end
```

索引付きレコード・パーツの構文図は、以下のとおりです。



Record recordPartName indexedRecord

パーツをタイプ `indexedRecord` として識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

fileName = "logicalFileName"

ファイル名を指定します。入力の意味については、『リソース関連 (概説)』を参照してください。命名の規則については、『命名規則』を参照してください。

keyItem = "keyItem"

キー項目。同一レコード内で固有の構造体項目のみを指定できます。`keyItem` には、非修飾参照を使用する必要があります。例えば、`myRecord.myItem` ではなく `myItem` を使用します。(関数では、他の構造体項目を参照するのと同様に、その構造体項目を参照できます。)

lengthItem = "lengthItem"

長さ項目。詳細については、『可変長レコードをサポートするプロパティ』を参照してください。

numElementsItem = "numElementsItem"

要素項目の数。詳細については、『可変長レコードをサポートするプロパティ』を参照してください。

structureItem

構造体項目。詳細については、『EGL ソース形式の構造体項目』を参照してください。

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』
24 ページの『パーツの参照』
19 ページの『パーツ』
141 ページの『レコード・パーツ』
62 ページの『EGL での変数の参照』
331 ページの『リソース関連とファイル・タイプ』
30 ページの『Typedef』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

78 ページの『配列』
512 ページの『EGL ソース形式の DataItem パーツ』
532 ページの『EGL ソース形式』
570 ページの『EGL ソース形式の関数パーツ』
714 ページの『EGL ソース形式の MQ レコード・パーツ』
725 ページの『命名規則』
37 ページの『プリミティブ型』
783 ページの『EGL ソース形式のプログラム・パーツ』
793 ページの『可変長レコードをサポートするプロパティ』
796 ページの『EGL ソース形式の相対レコード・パーツ』
799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』
808 ページの『EGL ソース形式の構造体フィールド』

入出力エラー値

次の表は、データベース、ファイル、および MQSeries メッセージ・キューに影響を及ぼす入出力 (I/O) 操作の EGL エラー値を説明しています。ハード・エラーに関連した値がコードで有効となるのは、『例外処理』で説明されているとおり、システム変数 `VGVar.handleHardIOErrors` が 1 に設定されている場合のみです。

エラー値	エラーのタイプ	レコードのタイプ	エラー値の意味
deadLock	ハード	SQL	2 つのプログラム・インスタンスがレコードを変更しようとしています、システム介入がない限りどちらも変更できません。
duplicate	ソフト	索引付きまたは相対	コードが、すでに存在するキーを持つレコードにアクセスしようとし、その試行は成功しました。詳細については、『duplicate』を参照してください。

エラー値	エラーのタイプ	レコードのタイプ	エラー値の意味
endOfFile	ソフト	索引付き、 相対、シリアル	詳細については、『 <i>endOfFile</i> 』を参照してください。
ioError	ハード または ソフト	任意	EGL で入出力操作から非ゼロの戻りコードが受け取られました。
format	ハード	任意	アクセスされたファイルがレコード定義と非互換です。詳細については、『 <i>format</i> 』を参照してください。
fileNotAvailable	ハード	任意	<i>fileNotAvailable</i> はどの入出力操作でも生じる可能性があります。例えば、別のプログラムがファイルを使用していること、またはファイルにアクセスするために必要なリソースが不足していることを表す場合があります。
fileNotFound	ハード	索引付き、 メッセージ・キュー、 相対、 シリアル	ファイルが見つかりませんでした。
full	ハード	索引付き、 相対、シリアル	<i>full</i> は、以下の場合に設定されます。 • 索引付きまたはシリアル・ファイルが満杯である
hardIOError	ハード	任意	ハード・エラーが発生しました。これは、 <i>endOfFile</i> 、 <i>noRecordFound</i> 、 <i>duplicate</i> 以外のエラーです。
noRecordFound	ソフト	任意	詳細については、『 <i>noRecordFound</i> 』を参照してください。
unique	ハード	索引付き、 相対、または SQL	UNQ は固有 (<i>unique</i>) を表します。コードが、すでに存在するキーを持つレコードを追加または置換しようとしたが、その試行は失敗しました。詳細については、『 <i>unique</i> 』を参照してください。

duplicate

索引付きまたは相対レコードの場合、**duplicate** が設定される条件は、以下のとおりです。

- **add** ステートメントが、キーまたはレコード ID がすでにファイルまたは代替索引に存在するレコードを挿入しようとし、挿入が成功した。
- **replace** ステートメントがレコードを正常に上書きし、置換値に別のレコードの代替索引キーと同じキーが含まれている。
- **get**、**get next**、または **get previous** ステートメントは、レコードを正常に読み取り (または、*set record position* という書式の **set** ステートメントが正常に実行され)、2 番目のレコードが同じキーを持つ。

duplicate 設定は、アクセス・メソッドが情報を戻す場合にのみ戻されます。これは、一部のオペレーティング・システムに当てはまりますが、他のオペレーティング・システムには当てはまりません。このオプションは、SQL データベース・アクセス中は使用不可です。

endOfFile

endOfFile が設定される条件は、以下のとおりです。

- 関連するファイル・ポインターがファイルの終わりにある場合に、コードがシリアル・レコードまたは相対レコードに対して **get next** ステートメントを発行する。ファイルの最終レコードが直前の **get** または **get next** ステートメントによってアクセスされたときに、ポインターが末尾にある。
- 関連するファイル・ポインターがファイルの終わりにある場合に、コードが索引付きレコードに対して **get next** ステートメントを発行する。これは、以下の状況で生じます。
 - ファイルの最終レコードが直前の **get** または **get next** ステートメントによってアクセスされた。または、
 - 以下のいずれかの条件が該当する場合に、ファイルの最終レコードが、直前の *set record position* 型の **set** ステートメントによってアクセスされた。
 - キー値がファイルの最終レコードのキーと一致した。
 - キー値のすべてのバイトが 16 進の FF に設定されている。(*set record position* 型の **set** ステートメントが、すべて 16 進の FF に設定されたキー値を指定して実行された場合、文は位置ポインターをファイルの終わりに設定します。)
- 関連するファイル・ポインターがファイル先頭にある場合に、コードが索引付きレコードに対して **get previous** ステートメントを発行する。これは、以下の状況で生じます。
 - ファイルの最初のレコードが直前の **get** または **get previous** ステートメントによってアクセスされた。
 - コードの直前のアクセスが、同じファイルに対するものではなかった。または
 - *set record position* 型の **set** ステートメントがキーを指定して実行されたが、ファイル内でそのキーの前にキーがなかった。
- **get next** ステートメントが、空のファイルまたは未初期化ファイルから索引付きレコードにデータを取り出そうとした。

(空のファイルは、すべてのレコードが削除されたファイルです。未初期化ファイルは、一度もレコードが追加されたことがないファイルです。)

- **get previous** ステートメントが、空のファイルから索引付きレコードにデータを取り出そうとした。

format

format はどの種類の入出力操作によっても生じる可能性があります、特に以下の理由で設定されます。

• レコード形式

ファイル形式 (固定長または可変長) が EGL レコード形式と異なる。

- レコード長

固定長レコードの場合、ファイル内のレコードの長さが EGL レコードの長さと異なる。可変長レコードの場合、ファイル内のレコードの長さが EGL レコードの長さより大きい。

- ファイル・タイプ

レコードに指定されているファイル・タイプが、実行時のファイル・タイプと一致しない。

- キー長

ファイル内のキー長が、EGL 索引付きレコード内のキー長と異なる。

- キー・オフセット

ファイル内のキー位置が、EGL 索引付きレコード内のキー位置と異なる。

noRecordFound

noRecordFound が設定される条件は、以下のとおりです。

- 索引付きレコードの場合、**get** ステートメントに指定されたキーと一致するレコードが検出されなかった。
- EGL で生成された Java の場合、VSAM ファイルが空または未初期化である場合に、コードが索引付きレコードに対して **get next** または **get previous** ステートメントを発行する。
- 相対レコードの場合、**get** ステートメントに指定されたレコード ID と一致するレコードが検出されなかった。あるいは、**get next** ステートメントがファイルの終わりを越えるレコードにアクセスしようとした。
- SQL レコードの場合、指定された **SELECT** ステートメントと一致する行が検出されなかった。あるいは、調査対象となる選択された行が残っていないときに **get next** ステートメントが実行された。

unique

索引付きまたは相対レコードの場合、**unique** が設定される条件は、以下のとおりです。

- **add** ステートメントが、キーまたはレコード ID がすでにファイルまたは代替索引に存在するレコードを挿入しようとしたが、重複のために挿入が失敗した。
- 置換値に別のレコードの代替索引キーと同じキーが含まれているので、**replace** ステートメントがレコードを上書きできなかった。

unique 設定は、アクセス・メソッドが情報を戻す場合にのみ戻されます。これは、一部のオペレーティング・システムに当てはまりますが、他のオペレーティング・システムには当てはまりません。

追加または置換されている SQL 行が、固有索引にすでに存在するキーを持っている場合は、SQL データベース・アクセス時に **unique** が設定されます。対応する sqlcode は -803 です。

関連する参照項目

605 ページの『add』
407 ページの『関連エレメント』
613 ページの『close』

617 ページの『delete』
100 ページの『例外処理』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
650 ページの『get previous』
538 ページの『論理式』
664 ページの『open』
679 ページの『prepare』
681 ページの『replace』

isa 演算子

演算子 **isa** は、与えられた式が特定のタイプであるかどうかをテストする 2 項演算子です。主な目的は、ANY 型のフィールドに保持されているデータのタイプをテストすることです。

この演算子は、次の形式の基本的な論理式の中で使用されます。

testExpression **isa** *typeSpecification*

testExpression

数値式、テキスト式、日時式のいずれか。これは、単一のフィールドからテラルで構成されていてもかまいません。

typeSpecification

タイプ指定。これは、次のいずれかにすることができます。

- パーツ名。
- **STRING** などのプリミティブ型の指定。ただし、そのプリミティブ型を長さへ関連付けることができる場合は、次の例のように、長さを指定する必要があります。
 - **BIN**(9)
 - **CHAR**(5)

日時マスクを含めないでください。

- タイプ指定 (前述のとおり) の直後に 1 対の大括弧を付けたもの。この場合、完全な指定は、特定のタイプ、長さ (該当する場合)、および次元数を備えた動的配列を示します。

この論理式は、*testExpression* が *typeSpecification* で識別されたタイプに一致する場合は真、それ以外の場合は偽へと解決されます。

関連する参照項目

78 ページの『配列』

538 ページの『論理式』

726 ページの『演算子と優先順位』

Java ランタイム・プロパティ (詳細)

次の表で、デプロイメント記述子またはプログラム・プロパティ・ファイルに含めることができるプロパティおよび J2EE 環境ファイル (存在する場合) に生成される値のソースを説明します。各プロパティの Java 型は、「説明」列に特記していない限り `java.lang.String` です。

ランタイム・プロパティ	説明	生成される値のソース
<code>cso.cicsj2c.timeout</code>	<p>プロトコル CICSJ2C を使用する呼び出しで、タイムアウトが発生するまでの時間をミリ秒単位で指定します。デフォルト値は 30000 であり、30 秒を表します。値を 0 に設定すると、タイムアウトは発生しません。値には、0 以上を設定する必要があります。</p> <p>この場合の Java の型は <code>Java.lang.Integer</code> です。</p> <p>コードが WebSphere 390 で実行されている場合は、このプロパティは呼び出しに影響を及ぼしません。詳細については『CICSJ2C 呼び出し用 J2EE サーバーの設定』を参照してください。</p>	ビルド記述子オプション <code>cicsj2cTimeout</code>
<code>cso.linkageOptions.LO</code>	<p>生成されたプログラムまたはラッパーが他のプログラムを呼び出す方法についてガイドするリンケージ・プロパティ・ファイルの名前を指定します。LO は、生成時に使用されたリンケージ・オプション・パーツ名です。詳細については、『リンケージ・プロパティ・ファイルのデプロイ』を参照してください。</p>	LO は、ビルド記述子オプション <code>linkage</code> から取られ、デフォルト値はリンケージ・オプション・パーツの名前の直後に拡張子 <code>.properties</code> を付けたものです。
<code>tcpiplistener.port</code>	<p>(クラス <code>CSOTcpipListener</code> または <code>CSOTcpipListenerJ2EE</code> の) EGL TCP/IP リスナーが <code>listen</code> するポートの番号を指定します。デフォルト値はありません。詳細については、TCP/IP リスナーのセットアップに関連するトピックを参照してください。</p> <p>この場合の Java の型は <code>Java.lang.Integer</code> です。</p>	生成されません

ランタイム・プロパティ	説明	生成される値のソース
tcpiplistener.trace.file	1 つ以上の EGL TCP/IP リスナー (それぞれ、クラス CSOTcpipListener または CSOTcpipListenerJ2EE をもつ) のアクティビティを記録するファイルの名前を指定します。デフォルトのファイルは tcpiplistener.out です。	生成されません。トレースは IBM で使用するためのものです。
tcpiplistener.trace.flag	<p>1 つ以上の EGL TCP/IP リスナー (それぞれ、クラス CSOTcpipListener または CSOTcpipListenerJ2EE をもつ) のアクティビティをトレースするかどうかを指定します。次のいずれかを選択します。</p> <ul style="list-style-type: none"> プロパティ tcpiplistener.trace.flag で識別されるファイルにアクティビティを記録する場合は 1 アクティビティを記録しない場合は 0 (デフォルト値) <p>この場合の Java の型は <code>Java.lang.Integer</code> です。詳細については、<i>TCP/IP</i> リスナーのセットアップに関連するトピックを参照してください。</p>	生成されません。トレースは IBM で使用するためのものです。
vgj.datemask. gregorian.long.locale	<p>以下の 2 つの場合のいずれかに使用される日付マスクが含まれています。</p> <ul style="list-style-type: none"> システム変数 <code>VGVar.currentFormattedGregorianCalendar</code> 用に生成された Java コードが呼び出される場合、または 項目プロパティ dateFormat が <code>systemGregorianCalendarFormat</code> に設定されている場合に、10 以上の長さのページ項目またはテキスト書式のフィールドが EGL により検証される場合 <p><code>locale</code> は、プロパティ vgj.nls.code に指定されているコードです。Web アプリケーションでは、<code>sysLib.setLocale</code> に別の値を割り当て、使用中の日付マスク・プロパティを変更できます。</p>	長いグレゴリオ日付マスクのビルド記述子値。デフォルト値はロケール固有です。

ランタイム・プロパティ	説明	生成される値のソース
vgj.datemask. gregorian.short.locale	<p>項目プロパティ dateFormat が <i>systemGregorianCalendar</i> に設定されている場合に、10 未満の長さのページ項目またはテキスト書式フィールドが EGL により検証されるときに使用される日付マスクが含まれています。</p> <p><i>locale</i> は、プロパティ vgj.nls.code に指定されているコードです。 Web アプリケーションでは、<i>sysLib.setLocale</i> に別の値を割り当て、使用中の日付マスク・プロパティを変更できます。</p>	短いグレゴリオ日付マスクのビルド記述子値。デフォルト値はロケール固有です。
vgj.datemask. julian.long.locale	<p>以下の 2 つの場合のいずれかに使用される日付マスクが含まれています。</p> <ul style="list-style-type: none"> システム変数 <i>VGVar.currentFormattedJulianDate</i> 用に生成される Java コードが呼び出される場合、または 項目プロパティ dateFormat が <i>systemJulianDateFormat</i> に設定されている場合に、10 以上の長さのページ項目またはテキスト書式のフィールドが EGL により検証される場合 <p><i>locale</i> は、プロパティ vgj.nls.code に指定されているコードです。 Web アプリケーションでは、<i>sysLib.setLocale</i> に別の値を割り当て、使用中の日付マスク・プロパティを変更できます。</p>	長いユリウス日付マスクのビルド記述子値。デフォルト値はロケール固有です。
vgj.datemask. julian.short.locale	<p>項目プロパティ dateFormat が <i>systemJulianDateFormat</i> に設定されている場合に、10 未満の長さのページ項目またはテキスト書式フィールドが EGL により検証されるときに使用される日付マスクが含まれています。</p> <p><i>locale</i> は、プロパティ vgj.nls.code に指定されているコードです。 Web アプリケーションでは、<i>sysLib.setLocale</i> に別の値を割り当て、使用中の日付マスク・プロパティを変更できます。</p>	短いユリウス日付マスクのビルド記述子値。デフォルト値はロケール固有です。

ランタイム・プロパティ	説明	生成される値のソース
vgj.default.databaseDelimiter	システム関数 SysLib.loadTable および SysLib.unLoadTable で値と値との区切り文字に使用されるシンボルを指定します。デフォルト値はパイプ記号 () です。	
vgj.default.dateFormat	システム変数 StrLib.defaultDateFormat の初期値を設定します。有効な値の詳細については、『日付、時刻、およびタイム・スタンプの各指定子』を参照してください。	
vgj.defaultI4GLNativeLibrary	nativeLibrary 型のライブラリーによってアクセスされる DLL 名を指定します。ライブラリー・プロパティ dllName を指定しなかった場合、このプロパティが必要です。	
vgj.default.moneyFormat	システム変数 StrLib.defaultMoneyFormat の初期値を設定します。有効な値の詳細については、『formatNumber()』を参照してください。	
vgj.default.numericFormat	システム変数 StrLib.defaultNumericFormat の初期値を設定します。有効な値の詳細については、『formatNumber()』を参照してください。	
vgj.default.timeFormat	システム変数 StrLib.defaultTimeFormat の初期値を設定します。有効な値の詳細については、『日付、時刻、およびタイム・スタンプの各指定子』を参照してください。	
vgj.default.timestampFormat	システム変数 StrLib.defaultTimestampFormat の初期値を設定します。有効な値の詳細については、『日付、時刻、およびタイム・スタンプの各指定子』を参照してください。	

ランタイム・プロパティ	説明	生成される値のソース
vgj.jdbc.database.SN	<p>システム関数 <code>sysLib.connect</code> または <code>VGLib.connectionService</code> によりデータベース接続を行う場合に使用する JDBC データベース名を指定します。</p> <p>標準 (非 J2EE) 接続と比較した場合、J2EE 接続については値の意味が異なります。</p> <ul style="list-style-type: none"> J2EE 接続 (実稼働環境で必要になるもの) に関連して、値は、JNDI レジストリーでデータ・ソースをバインドする名前です (例: <code>jdbc/MyDB</code>)。 標準の JDBC 接続 (デバッグに使用できるもの) に関連して、値は、接続 URL です (例: <code>jdbc:db2:MyDB</code>)。 <p>デプロイメント時に <code>SN</code> の置換値を指定する場合は、プロパティ自体の名前をカスタマイズする必要があります。代わりになる置換値は、<code>VGLib.connectionService</code> の呼び出しに含まれているサーバー名、または <code>sysLib.connect</code> の呼び出しに含まれているデータベース名のいずれかと一致する必要があります。</p>	指定された「サーバー名」に関連付けられたデータベース名のビルド記述子値
vgj.jdbc.default.database.autoCommit	<p>デフォルト・データベースへのすべての変更の後でコミットが生じるかどうかを指定します。</p> <p>『<code>sqlCommitControl</code>』で説明されているように、有効な値は <code>true</code> と <code>false</code> です。</p>	ビルド記述子オプション <code>sqlCommitControl</code>

ランタイム・プロパティ	説明	生成される値のソース
vgj.jdbc.default. database. <i>programName</i>	<p>先行するデータベース接続が存在しない場合に SQL 入出力操作に使用するデフォルト・データベース名を指定します。EGL には、<i>programName</i> の置換値としてプログラム名 (ある場合はプログラム別名) が含まれているため、各プログラムは独自のデフォルト・データベースを持ちます。プログラム名はオプションですが、このようなプログラム固有のプロパティで参照されていないプログラムのデフォルトとして、vgj.jdbc.default.database という名前のプロパティが使用されます。</p> <p>非 J2EE 接続と比較した場合、J2EE 接続についてはプロパティ自体の値の意味が異なります。</p> <ul style="list-style-type: none"> • J2EE 接続に関連して、値は、JNDI レジストリーでデータ・ソースをバインドする名前です (例: jdbc/MyDB)。 • 標準の JDBC 接続に関連して、値は、接続 URL です (例: jdbc:db2:MyDB)。 	<p>以下の接続タイプに依存します。</p> <ul style="list-style-type: none"> • J2EE 接続の場合、ビルド記述子オプション sqlJNDIName • 非 J2EE 接続の場合、ビルド記述子オプション sqlDB
vgj.jdbc.default.password	<p>vgj.jdbc.default.database で識別されるデータベース接続にアクセスするためのパスワードを指定します。</p> <p>J2EE 環境ファイルでのパスワードの公開を回避するには、次のタスクのいずれか 1 つを行います。</p> <ul style="list-style-type: none"> • システム関数 sysLib.connect または VGLib.connectionService を使用してプログラムおよび関数スクリプトでパスワードを指定する。または • Web アプリケーション・サーバーのデータ・ソース指定にユーザー ID およびパスワードを含める。詳細については『J2EE JDBC 接続のセットアップ』を参照してください。 	ビルド記述子オプション sqlPassword
vgj.jdbc.default.userid	vgj.jdbc.default.database で識別されるデータベース接続にアクセスするためのユーザー ID を指定します。	ビルド記述子オプション sqlID

ランタイム・プロパティ	説明	生成される値のソース
vgj.jdbc.drivers	vgj.jdbc.default.database で識別されるデータベース接続にアクセスするためのドライバ・クラスを指定します。このプロパティはデプロイメント記述子または J2EE 環境ファイルには存在せず、標準の (非 J2EE) JDBC 接続の場合にのみ使用します。	ビルド記述子オプション sqlJDBCClass
vgj.messages.file	<p>作成またはカスタマイズするメッセージが含まれているプロパティ・ファイルを指定します。このファイルが検索されるのは、次の場合です。</p> <ul style="list-style-type: none"> • EGL ランタイムが関数 SysLib.getMessage の呼び出しに応答する場合。作成したメッセージが戻されます。詳細については、SysLib.getMessage を参照してください。 • EGL ランタイムが consoleUI アプリケーションを処理していて、システム変数 ConsoleLib.messageResource で識別されたファイルから、ヘルプまたはコメント・テキストを表示使用としたときに、その変数に値が入っていない場合。 • 『EGL ランタイム・メッセージのメッセージのカスタマイズ』に説明されているように、EGL が Java ランタイム・メッセージを表示しようとする場合。 	
vgj.nls.code	<p>プログラムの 3 文字の NLS コードを指定します。有効な値のリストについては、『targetNLS』を参照してください。</p> <p>このプロパティを設定しない場合は、次の規則が適用されます。</p> <ul style="list-style-type: none"> • 値は、デフォルトでデフォルト Java ロケールに対応する NLS コードになります。 • デフォルトの Java ロケールが EGL によってサポートされているいずれの NLS コードにも対応しない場合は、値は ENU です。 	ビルド記述子オプション targetNLS

ランタイム・プロパティ	説明	生成される値のソース
<code>vgj.nls.currency</code>	通貨記号として使用する文字を指定します。デフォルトは、 vgj.nls.code に関連付けられたロケールによって決定されます。	ビルド記述子オプション currencySymbol
<code>vgj.nls.number.decimal</code>	10 進記号として使用する文字を指定します。デフォルトは、 vgj.nls.code に関連付けられたロケールによって決定されます。	ビルド記述子オプション decimalSymbol
<code>vgj.properties.file</code>	<p>非 J2EE 実行単位内の最初のプログラムが VisualAge Generator で生成されたか、6.0 より前のバージョンの EGL で生成された場合にのみ使用します。</p> <p>vgj.properties.file は、代替のプロパティ・ファイルを指定します。このファイルは、非 J2EE 実行単位全体を通じて、任意の非グローバル・プログラム・プロパティ・ファイルの代わりに使用されます。グローバル・ファイルの使用は影響を受けません。(最初のプログラムが古い EGL または VisualAge Generator で生成されている実行単位内では、グローバル・ファイルは vgj.properties と呼ばれます。)</p> <p>プロパティ vgj.properties.file によって参照されるファイルは、次の例のように、このプロパティをコマンド行ディレクティブの中に組み込んだ場合にのみ使用されます。</p> <pre>java -Dvgj.properties.file= c:¥new.properties</pre> <p>vgj.properties.file の値には、プロパティ・ファイルへの完全修飾パスが含まれます。</p> <p>プロパティ vgj.properties.file をプロパティ・ファイルの中で指定しても効果はありません。</p>	
<code>vgj.ra.QN.conversionTable</code>	<p>QN で識別される MQSeries メッセージ・キューのアクセス時に、生成された Java プログラムで使用される変換テーブルの名前を指定します。有効な値は <code>programControlled</code>、<code>NONE</code>、または変換テーブル名です。デフォルトは <code>NONE</code> です。</p>	リソース関連プロパティ conversionTable

ランタイム・プロパティ	説明	生成される値のソース
<code>vgj.ra.FN.fileType</code>	<p><i>FN</i> (レコード・パーツで識別されるファイルまたはキューの名前) に関連付けられたファイルのタイプを指定します。『レコードとファイル・タイプの相互参照』に説明されているように、このプロパティ値は <code>seqws</code> または <code>mq</code> です。</p> <p>このデプロイメント記述子プロパティは、プログラムが使用する論理ファイルごとに指定する必要があります。</p>	リソース関連プロパティ fileType
<code>vgj.ra.FN.replace</code>	<p><i>FN</i> (レコードで識別されるファイル名) に関連付けられたレコードに対する <code>add</code> ステートメントの効果を指定します。次の 2 つの値のいずれかを選択します。</p> <ul style="list-style-type: none"> 文がファイル・レコードを置換する場合は 1 文がファイルにレコードを追加する場合は 0 (デフォルト) <p>この場合の Java の型は <code>java.lang.Integer</code> です。</p>	リソース関連プロパティ replace
<code>vgj.ra.FN.systemName</code>	<p><i>FN</i> (レコード・パーツで識別されるファイルまたはキューの名前) に関連付けられた物理ファイルまたはメッセージ・キューの名前を指定します。</p> <p>このデプロイメント記述子プロパティは、プログラムが使用する論理ファイルごとに指定する必要があります。</p>	リソース関連プロパティ systemName

ランタイム・プロパティ	説明	生成される値のソース
vgj.ra.FN.text	<p>生成された Java プログラムがシリアル・レコードを介してファイルにアクセスするときに、このプログラムに次に示すことを実行させるかどうかを指定します。</p> <ul style="list-style-type: none"> • add 操作中に、行の終わりを示す文字を付加する。UNIX 以外のプラットフォームでは、付加される文字は復帰文字と改行文字です。UNIX プラットフォームでは、改行文字が唯一の付加対象文字です。 • get next 操作中に、行の終わりを示す文字を除去する。 <p><i>FN</i> はシリアル・レコードに関連付けられたファイル名です。</p> <p>次の値から 1 つを選択します。</p> <ul style="list-style-type: none"> • 変更する場合は 1 • 変更しない場合は 0 (デフォルト) <p>この場合の Java の型は <code>java.lang.Integer</code> です。</p>	リソース関連プロパティ text
vgj.trace.device.option	<p>トレース・データの宛先 (存在する場合)。次の値から 1 つを選択します。</p> <ul style="list-style-type: none"> • <code>System.out</code> に書き込む場合は 0 • <code>System.err</code> に書き込む場合は 1 • vgj.trace.device.spec で指定されたファイルに書き込む場合は 2 (デフォルト)。ただし、VSAM 入出力トレースの場合は、<code>vsam.out</code> へ書き込む。 <p>この場合の Java の型は <code>java.lang.Integer</code> です。</p>	生成される値 (存在する場合) は 2 です。
vgj.trace.device.spec	<p>vgj.trace.device.option を 2 に設定する場合の出力ファイルの名前を指定します。例外は、VSAM 入出力トレースは <code>vsam.out</code> に書き込まれるということです。</p>	生成される値 (存在する場合) は <code>vgjtrace.out</code> です。

ランタイム・プロパティ	説明	生成される値のソース
vgj.trace.type	<p>ランタイム・トレース設定を指定します。対象とするトレースについて次の値を合計します。</p> <ul style="list-style-type: none"> • -1: すべてトレースする • 0: トレースしない (デフォルト) • 1: 汎用トレース (関数呼び出しおよび call ステートメントを含む) • 2: 演算を処理するシステム関数 • 4: ストリングを処理するシステム関数 • 16: call ステートメントで渡されるデータ • 32: 呼び出し時に使用されるリンクージ・オプション • 128: JDBC 入出力 • 256: ファイル入出力 • 512: vgj.jdbc.default.password 以外のすべてのプロパティ <p>この場合の Java の型は java.lang.Integer です。</p>	生成される値 (存在する場合) は 0 です。

関連する概念

377 ページの『Java ランタイム・プロパティ』

151 ページの『basicLibrary タイプのライブラリー・パーツ』

396 ページの『リンクージ・プロパティ・ファイル』

関連するタスク

395 ページの『リンクージ・プロパティ・ファイルのデプロイ』

394 ページの『J2EE JDBC 接続のセットアップ』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

390 ページの『J2EE アプリケーション・クライアント・モジュールにおける呼び出し先アプリケーション用のTCP/IP リスナーのセットアップ』

383 ページの『呼び出し先となる非J2EE アプリケーションのTCP/IP リスナーのセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

442 ページの『callLink エlement』

420 ページの『cicsj2cTimeout』

957 ページの『connect()』

980 ページの『connectionService()』

1007 ページの『currentFormattedGregorianCalendar』

1008 ページの『currentFormattedJulianCalendar』

1011 ページの『currentShortGregorianCalendar』

1011 ページの『currentShortJulianDate』
 49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』
 421 ページの『decimalSymbol』
 936 ページの『defaultDateFormat』
 937 ページの『defaultMoneyFormat』
 937 ページの『defaultNumericFormat』
 937 ページの『defaultTimeFormat』
 938 ページの『defaultTimestampFormat』
 940 ページの『formatNumber()』
 966 ページの『getMessage()』
 431 ページの『linkage』
 708 ページの『リンケージ・プロパティ・ファイル (詳細)』
 967 ページの『loadTable()』
 713 ページの『EGL Java ランタイム用のメッセージのカスタマイズ』
 793 ページの『レコードとファイル・タイプの相互参照』
 971 ページの『setLocale()』
 435 ページの『sqlCommitControl』
 435 ページの『sqlDB』
 436 ページの『sqlID』
 436 ページの『sqlJDBCdriverClass』
 437 ページの『sqlJNDIName』
 438 ページの『sqlPassword』
 440 ページの『targetNLS』
 976 ページの『unloadTable()』

Java ラッパー・クラス

プログラム・パーツを Java ラッパーとして生成することを要求すると、EGL は、次のそれぞれについて、ラッパー・クラスを生成します。

- 生成済みのプログラム
- そのプログラム内のパラメーターとして宣言される各固定レコードまたは書式
- パラメーターとして宣言される各動的配列。配列が固定レコードの配列である場合は、動的配列のクラスは固定レコード自体のクラスに追加されます。
- 次の特性を持つ各構造体項目
 - ラッパー・クラスが生成される固定レコードまたは書式の 1 つの中にある構造体項目
 - 少なくとも 1 つの従属構造体項目を持つ構造体項目 (つまり、副構造がある構造体項目)
 - 配列である構造体項目 (この場合は、副構造配列)

副構造配列を持つ固定レコード・パーツの例を次に示します。

```

Record myPart type basicRecord
  10 MyTopStructure CHAR(20)[5];
    20 MyStructureItem01 CHAR(10);
    20 MyStructureItem02 CHAR(10);
end
  
```

プログラム・ラッパー・クラス、パラメーター・ラッパー・クラス、動的配列ラッパー・クラス、および副構造体項目付き配列ラッパー・クラスとしてのプログラムのラッパー・クラスについては、後に説明します。

EGL により、各パラメーター・ラッパー・クラス、動的配列ラッパー・クラス、または副構造体項目付き配列ラッパー・クラスごとに BeanInfo クラスが生成されます。BeanInfo クラスは、関連ラッパー・クラスを Java 準拠の Java Bean として使用できるようにします。通常、BeanInfo クラスと対話することはありません。

ラッパーを生成すると、呼び出し先プログラムのパラメーター・リストは、BLOB、CLOB、STRING、Dictionary、ArrayDictionary、または非固定レコード型のパラメーターをインクルードできません。

ラッパー・クラスの使用法の概要

ラッパー・クラスを使用して VisualAge Generator で生成されたプログラムと通信するには、ネイティブ Java プログラムで次のことを行います。

- ネイティブ Java コードと生成済みプログラム間でデータ変換を行うようなミドルウェア・サービスを提供するクラス (CSOPowerServer のサブクラス) をインスタンス化します。

```
import com.ibm.javart.v6.cso.*;

public class MyNativeClass
{
    /* ミドルウェアの変数を宣言します */
    CSOPowerServer powerServer = null;

    try
    {
        powerServer = new CSOLocalPowerServerProxy();
    }
    catch (CSOException exception)
    {
        System.out.println("Error initializing middleware"
            + exception.getMessage());
        System.exit(8);
    }
}
```

- 次のことを行うプログラム・ラッパー・クラスをインスタンス化します。
 - データ構造を割り振り、動的配列がある場合にはそれを組み込む
 - 生成されたプログラムにアクセスするメソッドへのアクセスを提供する

コンストラクターを呼び出すには、ミドルウェア・オブジェクトが必要です。

```
myProgram = new MyprogramWrapper(powerServer);
```

- パラメーター・ラッパー・クラスに基づく変数を宣言します。

```
Mypart myParm = myProgram.getMyParm();
Mypart2 myParm2 = myProgram.getMyParm2();
```

プログラムに動的配列のパラメーターがある場合は、次のように、動的配列ラッパー・クラスを基にして、それぞれ追加変数を宣言します。

```
myRecArrayVar myParm3 = myProgram.getMyParm3();
```

動的配列との対話の詳細については、601 ページの『動的配列ラッパー・クラス』を参照してください。

- 通常 (直前のステップで示したように)、プログラム・ラッパー・オブジェクトに割り振られたメモリーを参照および変更するには、パラメーター変数を使用します。
- ユーザー ID およびパスワードを設定しますが、次の場合に限ります。
 - Java ラッパーが、iSeries Toolbox for Java を介して iSeries ベース・プログラムにアクセスする場合
 - リモート・アクセスを認証する CICS for z/OS 領域上で生成プログラムが実行される場合

データベースへのアクセスでは、ユーザー ID とパスワードは使用されません。

ユーザー ID とパスワードを設定および確認するには、プログラム・オブジェクトの `callOptions` 変数を使用します。例を次に示します。

```
myProgram.callOptions.setUserID("myID");
myProgram.callOptions.setPassword("myWord");
myUserID = myProgram.callOptions.getUserID();
myPassword = myProgram.callOptions.getPassword();
```

- 生成されたプログラムにアクセスします。通常は、プログラム・ラッパー・オブジェクトの `execute` メソッドを呼び出します。


```
myProgram.execute();
```
- ミドルウェア・オブジェクトを使用して、データベース・トランザクションの制御を設定します。ただし、これは、次の場合に限ります。
 - プログラム・ラッパー・オブジェクトが CICS for z/OS 上の生成プログラムにアクセスしているか、IBM Toolbox for Java を介して iSeries ベース COBOL プログラムにアクセスしている場合後者のケースでは、呼び出しの **remoteComType** の値は **JAVA400** です。
 - ラッパー・クラスの生成に使用するリンケージ・オプション・パーツで、データベース作業単位をクライアント (この場合はラッパー) で制御するように指定した。詳細については、『*callLink* エレメント』の **luwControl** の説明を参照してください。

データベース作業単位をクライアントで制御する場合、処理では、ミドルウェア・オブジェクトの `commit` メソッドと `rollback` メソッドを使用する必要があります。

```
powerServer.commit();
powerServer.rollback();
```

- ミドルウェア・オブジェクトを閉じ、ガーベッジ・コレクションの準備をします。

```
if (powerServer != null)
{
    try
    {
        powerServer.close();
        powerServer = null;
    }

    catch(CSOException error)
    {
        System.out.println("Error closing middleware")
    }
}
```

```

        + error.getMessage());
    System.exit(8);
}
}

```

プログラム・ラッパー・クラス

プログラム・ラッパー・クラスには、生成済みプログラムの各パラメーター用のプライベート・インスタンス変数が含まれています。パラメーターがレコードまたは書式の場合、変数は、関連するパラメーター・ラッパー・クラスのインスタンスを参照します。パラメーターがデータ項目の場合、変数は、Java プリミティブ型です。

EGL と Java 型の間の変換については、このヘルプ・ページの最後に表でまとめています。

プログラム・ラッパー・オブジェクトには、次の **public** メソッドが含まれます。

- 各パラメーター用の **get** メソッドと **set** メソッド。名前の形式は次のとおりです。

purposeParmname()

purpose

ワード **get** または **set**

Parmname

データ項目、レコード、または書式の名前。名前の先頭は大文字で表し、残りの文字は、603 ページの『Java ラッパー・クラスの命名規則』で説明されている命名規則により決定されます。

- プログラム呼び出し用の **execute** メソッド。このメソッドが使用されるのは、呼び出し時に引数として渡されるデータが、プログラム・ラッパー・オブジェクトに割り振られたメモリー内に存在する場合です。

インスタンス変数に値を割り当てる代わりに、次のことが可能です。

- パラメーター・ラッパー・オブジェクト、動的配列ラッパー・オブジェクト、およびプリミティブ型用にメモリーを割り振る。
- 割り振ったメモリーに値を割り当てる。
- プログラム・ラッパー・オブジェクトの **call** メソッド (**execute** メソッドではなく) を呼び出し、プログラムにこれらの値を渡す。

プログラム・ラッパー・オブジェクトには、**callOptions** 変数も含まれます。

callOptions 変数は、次の目的を持ちます。

- 呼び出しのリンケージ・オプションが生成時に設定されるように Java ラッパーを生成した場合、**callOptions** 変数にはリンケージ情報が含まれます。リンケージ・オプションをいつ設定するかについては、『*callLink* エレメント』の **remoteBind** を参照してください。
- 呼び出しのリンケージ・オプションが実行時に設定されるように Java ラッパーを生成した場合、**callOptions** 変数にはリンケージ・プロパティ・ファイルの名前が含まれます。ファイルの名前は **LO.properties** です。**LO** は生成で使用するリンケージ・オプション・パーツ名です。

- いずれの場合も、callOptions 変数によって、ユーザー ID およびパスワードを設定または取得する以下のメソッドが提供されます。

```
setPassword(password)
setUserid(userid)
getPassword()
getUserid()
```

このユーザー ID およびパスワードは、以下のいずれかの値に callLink エレメントの **remoteComType** プロパティを設定するときに使用します。

- CICSECI
- CICSJ2C
- JAVA400

最後に、プリミティブ型のパラメーターが変更されたときに、ネイティブ Java コードが通知を必要とする状況を考えましょう。このような通知を可能にするには、プログラム・ラッパー・オブジェクトの **addPropertyChangeListener** メソッドを呼び出して、ネイティブ・コードを listener として登録する必要があります。そうした場合、次の状況のいずれかによって PropertyChange イベントが起動されると、実行時、ネイティブ・コードが通知を受け取ることができるようになります。

- ネイティブ・コードにより、プリミティブ型のパラメーター上で **set** メソッドを呼び出す。
- 生成されたプログラムにより、変更されたデータがプリミティブ型のパラメーターに戻される。

PropertyChange イベントについては、Sun Microsystems, Inc. の JavaBean 仕様に記述されています。

パラメーター・ラッパー・クラスのセット

パラメーター・ラッパー・クラスは、生成されるプログラム内のパラメーターとして宣言されるレコードごとに作成されます。通常、パラメーター・ラッパー・クラスは、パラメーターを参照する変数を宣言するためにのみ使用します。例を次に示します。

```
Mypart myRecWrapperObject = myProgram.getMyrecord();
```

この場合は、プログラム・ラッパー・オブジェクトによって割り振られたメモリーが使用されます。

パラメーター・ラッパー・クラスを使用すると、プログラム・オブジェクトの call メソッド (execute メソッドではなく) を呼び出す場合に必要なメモリーを宣言することもできます。

パラメーター・ラッパー・クラスには、次に示すように、一連のプライベート・インスタンス変数が含まれます。

- パラメーターの各低レベル構造体項目用の Java プリミティブ型の変数 (ただし、配列でもなく副構造付き配列内にも存在しない構造体項目の場合に限る)。
- 配列ではあるが副構造は持たない各 EGL 構造体項目用の Java プリミティブ型の配列。

- それ自体は副構造付き配列内に存在しない各副構造付き配列用の内部の配列クラスのオブジェクト。インナー・クラスは、ネストされたインナー・クラスを持つことで、従属副構造付き配列を表すことができます。

パラメーター・ラッパー・クラスには、`public` メソッドがいくつか含まれます。

- 一連の `get` メソッドと `set` メソッドを使用すると、各インスタンス変数を取得および設定できます。各メソッド名の形式を次に示します。

`purposesiName()`

`purpose`

ワード `get` または `set`

`siName`

構造体項目の名前。名前の先頭は大文字で表し、残りの文字は、603 ページの『Java ラッパー・クラスの命名規則』で説明されている命名規則により決定されます。

注: 充てん文字として宣言した構造体項目は、プログラム呼び出し内に含まれます。ただし、これらの構造体項目の `public get` メソッドまたは `set` メソッドは、配列ラッパー・クラスには含まれません。

- メソッド `equals` を使用すると、同一クラスの別のオブジェクトに格納されている変数が、パラメーター・ラッパー・オブジェクト内に格納されている値と同じであるかどうかを確認できます。このメソッドが `true` を戻すのは、クラスと値が同じ場合に限りです。
- メソッド `addChangeListener` が呼び出されるのは、Java プリミティブ型の変数の変更通知をプログラムが必要としている場合です。
- `get` メソッドと `set` メソッドの 2 番目のセットを使用すると、SQL レコード・パラメーター内の各構造体項目ごとに `NULL` 標識を取得および設定できます。これらのメソッド名の形式を次に示します。

`purposesiNameNullIndicator()`

`purpose`

ワード `get` または `set`

`siName`

構造体項目の名前。名前の先頭は大文字で表し、残りの文字は、603 ページの『Java ラッパー・クラスの命名規則』で説明されている命名規則により決定されます。

副構造体項目付き配列ラッパー・クラスのセット

副構造体項目付き配列ラッパー・クラスは、パラメーター・クラスのインナー・クラスであり、関連するパラメーター内の副構造付き配列を表します。副構造体項目付き配列ラッパー・クラスには、配列自体の構造体項目および配列以下の構造体項目を参照する一連のプライベート・インスタンス変数が含まれています。

- 配列の各低レベル構造体項目用の Java プリミティブ型の変数 (ただし、配列でもなく副構造付き配列内にも存在しない構造体項目の場合に限る)。
- 配列ではあるが副構造は持たない各 EGL 構造体項目用の Java プリミティブ型の配列。

- それ自体は副構造付き配列内に存在しない各副構造付き配列用の内部の副構造体項目付き配列ラッパー・クラスのオブジェクト。インナー・クラスは、ネストされたインナー・クラスを持つことで、従属副構造付き配列を表すことができます。

副構造体項目付き配列ラッパー・クラスには、次のメソッドが含まれます。

- 各インスタンス変数用の一連の **get** メソッドと **set** メソッド。

注: 名前のない充てん文字として宣言した構造体項目はプログラム呼び出し内で使用されますが、これらの構造体項目の **public get** または **set** メソッドは副構造体項目付き配列ラッパー・クラスに含まれません。

- メソッド **equals** を使用すると、同一クラスの別のオブジェクトに格納されている変数が、副構造体項目付き配列ラッパー・オブジェクト内に格納されている値と同じであるかどうかを確認できます。このメソッドが **true** を戻すのは、クラスと値が同じ場合に限りです。
- メソッド **addPropertyChangeListener**。Java プリミティブ型の変数の変更を通知する必要がある場合に使用します。

通常、パラメーター・ラッパー・クラス内にある最上位の副構造体項目付き配列ラッパー・クラスの名前は、次のような書式で表現されます。

ParameterClassname.ArrayClassname

例えば、次のレコードについて考えましょう。

```
Record CompanyPart type basicRecord
10 Departments CHAR(20)[5];
  20 CountryCode CHAR(10);
    20 FunctionCode CHAR(10)[3];
      30 FunctionCategory CHAR(4);
        30 FunctionDetail CHAR(6);
end
```

パラメーター **Company** が **CompanyPart** に基づいている場合は、インナー・クラスの名前として、ストリング **CompanyPart.Departments** を使用します。

インナー・クラスのインナー・クラスにより、ドット構文の使用が拡張されます。この例では、**Departments** のインナー・クラス名として、シンボル **CompanyPart.Departments.Functioncode** を使用します。

副構造体項目付き配列ラッパー・クラスの命名の詳細については、『*Java ラッパー生成の出力*』を参照してください。

動的配列ラッパー・クラス

動的配列ラッパー・クラスは、生成されるプログラム内のパラメーターとして宣言される動的配列ごとに作成されます。次の EGL プログラム・シグニチャーについて考えます。

```
Program myProgram(intParms int[], recParms MyRec[])
```

動的配列ラッパー・クラス名は **IntParmsArray** and **MyRecArray** です。

動的配列ラッパー・クラスを使用して、次の例のように、動的配列を参照する変数を宣言します。

```
IntParamsArray myIntArrayVar = myProgram.getIntParams();
MyRecArray      myRecArrayVar = myProgram.getRecParams();
```

動的配列ごとに変数を宣言した後、次のように、要素を追加する場合があります。

```
// Java プリミティブの配列への追加は
// 1 ステップ・プロセスです。
myIntArrayVar.add(new Integer(5));

// レコードの配列または書式への追加には、
// 複数のステップが必要です。この場合、
// 新規レコード・オブジェクトを割り振ることから始めます。
MyRec myLocalRec = (MyRec)myRecArrayVar.makeNewElement();

// 値を割り当てるステップは、この例では表示されていません。
// ただし、値を割り当てた後、
// レコードを配列に追加します。
myRecArrayVar.add(myLocalRec);

// 次に、プログラムを実行します。
myProgram.execute();

// プログラムが戻ったときに、
// 配列の要素数を決定できます。
int myIntArrayVarSize = myIntArrayVar.size();

// 整数配列の最初の要素を得て、
// Integer オブジェクトにキャストします。
Integer firstIntElement = (Integer)myIntArrayVar.get(0);

// レコード配列の 2 番目の要素を得て、
// MyRec オブジェクトにキャストします。
MyRec secondRecElement = (MyRec)myRecArrayVar.get(1);
```

例で示されるように、EGL では、宣言した変数进行操作する方法がいくつか提供されます。

動的配列クラスのメソッド	用途
<code>add(int, Object)</code>	<code>int</code> により指定される位置にオブジェクトを挿入し、現行および後続の要素を右方にシフトします。
<code>add(Object)</code>	オブジェクトを動的配列の最後に追加します。
<code>addAll(ArrayList)</code>	<code>ArrayList</code> を動的配列の最後に追加します。
<code>get()</code>	配列内のすべての要素を含む <code>ArrayList</code> オブジェクトを検索します。
<code>get(int)</code>	<code>int</code> により指定される位置にある要素を検索します。
<code>makeNewElement()</code>	配列に特有のタイプの新規要素を割り振り、その要素を検索しますが、動的配列には追加しません。
<code>maxSize()</code>	動的配列内の要素の最大数（実際の数ではない）を示す整数を検索します。
<code>remove(int)</code>	<code>int</code> により指定される位置にある要素を除去します。
<code>set(ArrayList)</code>	動的配列の代替として、指定の <code>ArrayList</code> を使用します。
<code>set(int, Object)</code>	<code>int</code> により指定される位置にある要素の代替として、指定のオブジェクトを使用します。
<code>size()</code>	動的配列にある要素の数を検索します。

主に、次のような場合に例外が発生します。

- **get** または **set** メソッドに、無効な指標を指定した場合
- 配列の各要素クラスに非互換のクラスの要素を追加 (または設定) した場合
- それ以上の増加をサポートできない最大サイズに動的配列が到達しているときに、要素を追加した場合。つまり、メソッド **addAll** がこの理由により失敗した場合は、メソッドにより要素は追加されません。

Java ラッパー・クラスの命名規則

EGL では、次の規則に従い名前が作成されます。

- 名前がすべて大文字の場合、すべて小文字にします。
- 名前がキーワードの場合、その前にアンダーラインを付けます。
- 名前にハイフンまたはアンダーラインが含まれる場合は除去し、その後ろの文字を大文字にします。
- ドル記号 (\$)、アットマーク (@)、またはポンド記号 (#) が名前に含まれる場合は **ダブル・アンダースコア** (__) に置き換え、名前の前にアンダースコア (__) をつけます。
- 名前がクラス名として使用されている場合、最初の文字を大文字にします。

動的配列ラッパー・クラスには、以下の規則が適用されます。

- 通常、クラス名は、配列の各要素の基本であるパーツ宣言名 (データ項目、書式、またはレコード) に基づきます。例えば、レコード・パーツが **MyRec** と呼ばれ、配列宣言が **recParms myRec[]** である場合、関連動的配列ラッパー・クラスは **MyRecArray** と呼ばれます。
- 配列が関連パーツ宣言を持たない項目宣言に基づいている場合、動的配列クラス名は配列名に基づきます。例えば、配列宣言が **intParms int[]** である場合、関連動的配列ラッパー・クラスは **IntParmsArray** と呼ばれます。

データ型の相互参照

次の表は、生成されたプログラム内の EGL プリミティブ型、および生成されたラッパー内の Java データ型の関係をまとめたものです。

EGL プリミティブ型	文字または桁の長さ	長さ (バイト)	小数部	Java データ・タイプ	Java の最大精度
CHAR	1 から 32767	2 から 32766	NA	String	NA
DBCHAR	1 から 16383	1 から 32767	NA	String	NA
MBCHAR	1 から 32767	1 から 32767	NA	String	NA
UNICODE	1 から 16383	2 から 32766	NA	String	NA
HEX	2 から 75534	1 から 32767	NA	byte[]	NA
BIN、SMALLINT	4	2	0	short	4
BIN、INT	9	4	0	int	9
BIN、BIGINT	18	8	0	long	18
BIN	4	2	>0	float	4
BIN	9	4	>0	double	15
BIN	18	8	>0	double	15
DECIMAL、PACF	1 から 3	1 から 2	0	short	4

EGL プリミティブ型	文字または桁の長さ	長さ (バイト)	小数部	Java データ・タイプ	Java の最大精度
DECIMAL、PACF	4 から 9	3 から 5	0	int	9
DECIMAL、PACF	10 から 18	6 から 10	0	long	18
DECIMAL、PACF	1 から 5	1 から 3	>0	float	6
DECIMAL、PACF	7 から 18	4 から 10	>0	double	15
NUM、NUMC	1 から 4	1 から 4	0	short	4
NUM、NUMC	5 から 9	5 から 9	0	int	9
NUM、NUMC	10 から 18	10 から 18	0	long	18
NUM、NUMC	1 から 6	1 から 6	>0	float	6
NUM、NUMC	7 から 18	7 から 18	>0	double	15

関連する概念

327 ページの『Java ラッパー』
10 ページの『ランタイム構成』

関連するタスク

327 ページの『Java ラッパーの生成』

関連する参照項目

442 ページの『callLink エlement』
722 ページの『Java ラッパーの別名の割り当て方法』
708 ページの『リンケージ・プロパティ・ファイル (詳細)』
728 ページの『Java ラッパー生成の出力』
455 ページの『callLink エlementの remoteBind』

EGL での JDBC ドライバーの要件

JDBC ドライバーの要件は、EGL のデバッグ時であろうと実行時であろうと、以下のように、データベース管理システムによって異なります。

DB2 UDB

DB2 Universal ドライバーは EGL と互換性がありますが、関連するアプリケーション・ドライバーには互換性がありません。特に、アプリケーション・ドライバーは、forUpdate オプションを含んでいる EGL の **open** または **get** ステートメントを処理できません。

IBM では、Net ドライバーをまったく使用しないことを推奨しています。

J2EE アプリケーションを WebSphere Application Server v6.x で実行している場合は、DB2 バージョン 8.1.6 以上が必要です。それらのアプリケーションを WebSphere v5.x テスト環境で実行している場合は、DB2 バージョン 8.1.3 以上が必要です。

Informix

受け入れ可能な Informix JDBC ドライバーは 2.21.JC6 以上です。このドライバー・レベルは、JDBC 3.0 に準拠していないため、EGL **open** ステートメント

では HOLD オプションをサポートしません。現在、Informix JDBC 3.0 準拠のドライバーが使用可能になっているので、HOLD オプションをサポートします。

Oracle

Oracle 10i に付属の JDBC ドライバーは、受け入れ可能です。

EGL と共に使用される任意の JDBC ドライバーには、以下の規則が適用されます。

- ドライバーは JDBC 2.0 以上をサポートする必要があります。
- 以下のコンテキストで、値 `java.sql.ResultSet.CONCUR_UPDATABLE` が許可される必要があります。
 - `java.sql.Connection.createStatement(int,int)` への第 2 引数としてのコンテキスト
 - `java.sql.Connection.prepareStatement(String,int,int)` および
`java.sql.Connection.prepareCall(String,int,int)` への第 3 引数としてのコンテキスト
- EGL の **open** ステートメントで `hold` オプションをサポートしたい場合、ドライバーは JDBC 3.0 をサポートしている必要があります、以下のコンテキストで値 `java.sql.ResultSet.HOLD_CURSORS_OVER_COMMIT` が許可される必要があります。
 - `java.sql.Connection.createStatement(int,int,int)` への第 3 引数としてのコンテキスト
 - `java.sql.Connection.prepareStatement(String,int,int,int)` および
`java.sql.Connection.prepareCall(String,int,int,int)` への第 4 引数としてのコンテキスト

どのデータベース管理システムの場合でも、サード・パーティーまたはフォース・パーティー・ベンダー製の JDBC ドライバーが受け入れ可能です。

関連するタスク

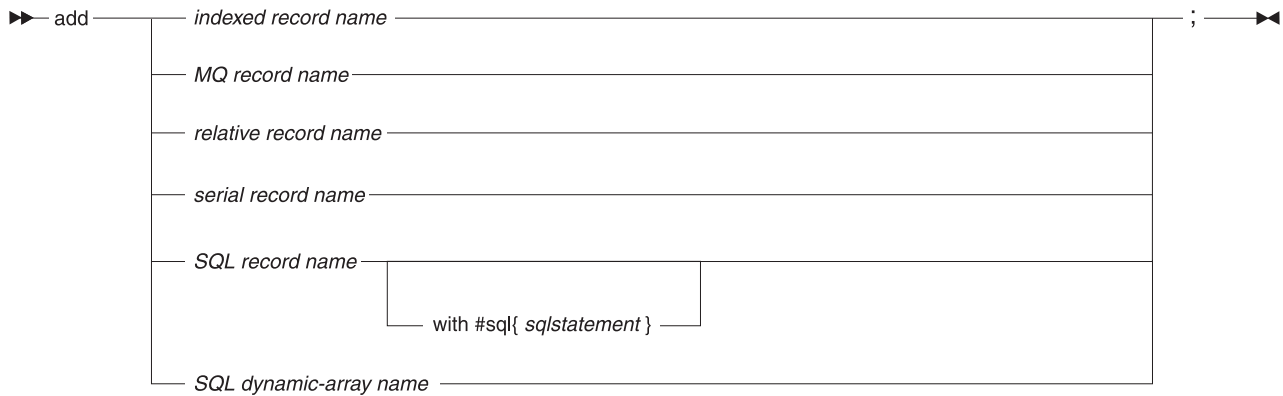
394 ページの『J2EE JDBC 接続のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

キーワード

add

EGL **add** ステートメントは、ファイル、メッセージ・キュー、またはデータベースの中にレコードを書き込みます。またはデータベースにレコード・セットを書き込みます。



record name

追加する入出力オブジェクトの名前。索引付きレコード、MQ レコード、相対レコード、シリアル・レコード、SQL レコードなどを指定します。

with #sql{ sqlStatement }

明示的な SQL INSERT ステートメント。#sql の後にスペースを入れないでください。

SQL dynamic-array name

SQL レコードの動的配列の名前。各エレメントは、エレメント固有のキー値によって指定されるデータベース内の位置に挿入されます。操作は、最初のエラー時に、またはすべてのエレメントが挿入されたときに停止します。

以下に例を示します。

```

if (userRequest == "A")
  try
    add record1;
  onException
    myErrorHandler(12);
end
end

```

add ステートメントの振る舞いは、レコード・タイプによって異なります。SQL 処理の詳細については、SQL レコードを参照してください。

索引付きレコード

索引付きレコードを追加するときは、レコード内のキーによって、ファイル内でのレコードの論理位置が判別されます。すでに使用されているファイル位置にレコードを追加すると、ハード入出力エラー UNIQUE、または (重複が許可されている場合は) ソフト入出力エラー DUPLICATE になります。

MQ レコード

MQ レコードを追加するとき、レコードはキューの最後に入れられます。この追加処理は、add が 以下に示す 1 つ以上の MQSeries 呼び出しを呼び出すことによって行われます。

- MQCONN は、生成されたコードをデフォルトのキュー・マネージャーに接続する。これは、アクティブな接続が存在しない場合に呼び出されます。
- MQOPEN は、キューとの接続を確立する。これは、接続がアクティブであるがキューがまだ開いていない場合に呼び出されます。

- MQPUT は、レコードをキューに入れる。これは、それ以前の MQSeries 呼び出しでエラーが発生していない限り、常に呼び出されます。

相対レコード

相対レコードを追加するときは、キー項目によって、ファイル内でのレコードの位置が指定されます。ただし、すでに使用されているファイル位置にレコードを追加すると、ハード入出力エラー UNIQUE になります。

レコード・キー項目は、そのレコードを使用するすべての関数で使用できなければなりません。以下のいずれかをレコード・キー項目とすることができます。

- 同じレコード内の項目
- プログラムに対してグローバルであるか、または **add** ステートメントを実行している関数に対してローカルであるレコード内の項目
- プログラムに対してグローバルであるか、または **add** ステートメントを実行している関数に対してローカルであるデータ項目

シリアル・レコード

シリアル・レコードを追加するとき、レコードはファイルの末尾に入れられます。

生成されたプログラムでシリアル・レコードが追加されてから、同じファイルに対して **get next** ステートメントが発行されると、**get next** ステートメントが実行される前に、ファイルが閉じられてから再び開かれます。したがって、**add** ステートメントの後に実行される **get next** ステートメントでは、ファイル内の最初のレコードが読み取られます。この振る舞いは、**get next** ステートメントと **add** ステートメントが別々のプログラムにあり、一方のプログラムで他方のプログラムが呼び出される場合にも起こります。

複数のプログラムで同時に同じファイルが開かないようにしておくことをお勧めします。

SQL レコード

エラー条件は、以下のとおりです。

- INSERT 以外の型の SQL ステートメントを指定する
- SQL INSERT ステートメントの全部ではなく一部の文節を指定する
- 以下のいずれかの特性を持つ SQL INSERT ステートメントを指定する (または暗黙的な SQL ステートメントを受け入れる)
 - 複数の SQL テーブルに関連付けられている
 - 読み取り専用として宣言したホスト変数のみを組み込む
 - 存在しないか、関連するホスト変数との互換性がない列に関連付けられている

明示的 SQL ステートメントを指定しないで SQL レコードを追加すると、結果は以下ようになります。

- 生成される SQL INSERT ステートメントの形式は以下のようになります。

```
INSERT INTO tableName
(column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

- テーブル内でのデータの論理位置は、レコード内のキー値によって判別されます。キーを持たないレコードは、SQL 表定義とデータベースの規則に従って処理されます。
- レコード・パーツにおけるレコード項目と SQL 表の列の関連付けの結果、生成されたコードにより、関連する SQL テーブル列に各レコード項目のデータが格納されます。
- レコード項目が読み取り専用として宣言されている場合、生成された SQL INSERT ステートメントにはそのレコード項目が含まれず、データベース管理システムでは、関連する SQL テーブル列の値がデフォルト値 (その列が定義されたときに指定された値) に設定されます。

SQL レコードの動的配列を使用する例は、以下のとおりです。

```
try
  add employees;
onException
  sysLib.rollback();
end
```

関連する概念

24 ページの『パーツの参照』

143 ページの『レコード・タイプとプロパティ』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

613 ページの『close』

617 ページの『delete』

631 ページの『get』

644 ページの『get next』

650 ページの『get previous』

100 ページの『例外処理』

619 ページの『execute』

579 ページの『入出力エラー値』

664 ページの『open』

679 ページの『prepare』

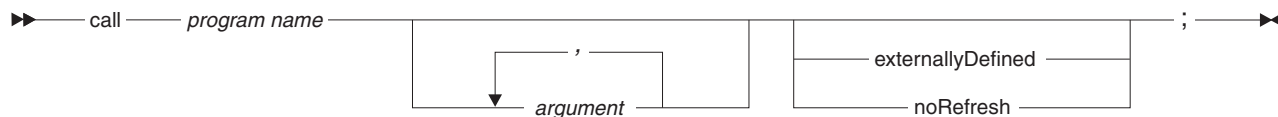
93 ページの『EGL ステートメント』

681 ページの『replace』

71 ページの『SQL 項目のプロパティ』

call

EGL call ステートメントは、制御を別のプログラムに移動し、オプションで一連の値を渡します。呼び出し先プログラムが終了すると、制御が呼び出し元に戻ります。また、変数によって渡されたデータが呼び出し先プログラムで変更されると、呼び出し元で使用可能なストレージ域も変更されます。



program name

呼び出し先プログラムの名前。プログラムは、EGL で生成されたプログラムであるか、外部定義されている と見なされます。

指定された名前を予約語にすることはできません。呼び出し側が EGL 予約語と同じ名前を持つ非 EGL プログラムを呼び出す必要がある場合は、`call` ステートメントで別のプログラム名を使用し、リンケージ・オプション・パーツ、**callLink** エレメントを使用して別名を指定します。この別名が、実行時に使用される名前です。

呼び出し先プログラムが Java プログラムの場合、その呼び出し先プログラムの名前は大文字小文字の区別があります。つまり、*calledProgram* と **CALLEDPROGRAM** は異なることになります。それ以外の場合、プログラム名に大文字小文字の区別があるかどうかは、呼び出し先プログラムがあるシステムで判断します。UNIX では大文字小文字の区別があり、その他のシステムでは大文字小文字の区別はありません。

EGL デバッガーでは、呼び出し先プログラムの名前には大文字小文字の区別がありません。

argument

それぞれがコンマで区切られた一連の値参照の 1 つ。`argument` には、プリミティブ変数、書式、レコード、固定レコード、非数値リテラル、非数値定数、または (生成時に EGL が呼び出し先プログラムにアクセスできる場合は) より複雑な日時式、数式、またはテキスト式を指定できます。ANY 型、ArrayDictionary 型、BLOB 型、CLOB 型、DataTable 型、または Dictionary 型のフィールドを渡すことはできません。また、これらの型の配列またはこれらの型を含むレコードを渡すことはできません。

externallyDefined

プログラムが外部定義されていることを示す標識。この標識を使用できるのは、VisualAge Generator との互換性があるプロジェクト・プロパティを設定する場合のみです。

EGL 以外で生成されたプログラムは、`call` ステートメントではなく、生成時に使用されるリンケージ・オプション・パーツを使用して、外部定義されていると識別することをお勧めします。(関連するプロパティは、リンケージ・オプション・パーツ、`callLink` エレメントにあり、**externallyDefined** と呼ばれます。)

noRefresh

呼び出し先プログラムが制御を戻すときに、画面を最新表示しないことを示す標識。

この標識は、プログラム・プロパティ **VAGCompatibility** が選択されている場合、または (生成時に) ビルド記述子オプション **VAGCompatibility** が `yes` に設定されている場合に、(開発時に) サポートされます。

この標識は、呼び出し側がテキスト書式を画面に表示する実行単位にあり、以下のいずれかの状態が有効である場合に適切です。

- 呼び出し先プログラムがテキスト書式を表示しない
- 呼び出し側が、呼び出し後に全画面テキスト書式に書き込む

画面を最新表示するための設定は、**call** ステートメントではなく、生成時に使用されるリンケージ・オプション・パーツを使用して示すことをお勧めします。(関連するプロパティは、リンケージ・オプション・パーツ、**callLink** エlementにあり、**refreshScreen** と呼ばれます。)

以下に例を示します。

```
if (userRequest == "C")
  try
    call programA;
  onException
    myErrorHandler(12);
  end
end
```

call ステートメントの引数の数、型、および順序は、呼び出し先プログラムによって予期される値の数、型、および順序と一致している必要があります。

各引数で渡されるバイト数を予期されるバイト数と同じにすることを特にお勧めします。長さのミスマッチがエラーの原因となるのは、そのミスマッチのランタイム修正によって型ミスマッチが生じる場合のみであり、以下のようになります。

- 呼び出し先の Java プログラムが受け取るバイト数が少ない場合は、渡されるデータの最後にブランクが埋め込まれます。
- 呼び出し先の Java プログラムが受け取るバイト数が多い場合は、渡されるデータの最後が切り捨てられます。

例えば、**NUM** 型のデータ項目にブランクが追加される場合はエラーが生じますが、**CHAR** 型のデータ項目にブランクが追加されてもエラーは生じません。

リテラルおよび定数には、以下の規則が適用されます。

- 渡されるリテラルまたは定数のサイズは、受取パラメーターのサイズと等しくなければならない
- 数値リテラルまたは定数は、引数として渡すことができない
- 1 バイト文字のみを含むリテラルまたは定数は、**CHAR** または **MBCHAR** 型のパラメーターに渡すことができる
- 2 バイト文字のみを含むリテラルまたは定数は、**DBCHAR** 型のパラメーターのみに渡すことができる
- 1 バイト文字と 2 バイト文字の両方を含むリテラルまたは定数は、**MBCHAR** 型のパラメーターに渡すことができる

再帰呼び出しは、すべてのターゲット・システムでサポートされています。

生成時に使用されるリンケージ・オプション・パーツが存在する場合、呼び出しは、それによる影響を受けます。(リンケージ・オプション・パーツを組み込むには、ビルド記述子オプション **linkage** を設定します。)

リンケージの詳細については、『リンケージ・オプション・パーツ』を参照してください。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

93 ページの『EGL ステートメント』

100 ページの『例外処理』

431 ページの『linkage』

37 ページの『プリミティブ型』

case

EGL **case** ステートメントは、文の複数のセットの始まりを示します。この複数のセットの内、実行されるセットは 1 つだけです。**case** ステートメントは、各 case 文節の最後に break がある C または Java 言語の **switch** ステートメントと等価です。



criteria

ConverseVar.eventKey または sysVar.systemType を含む、項目、定数、式、リテラル、またはシステム変数。

criteria を指定する場合、後続の **when** 文節があれば、それぞれに、1 つ以上の *matchExpression* インスタンスを含める必要があります。*criteria* を指定しない場合、後続の **when** 文節があれば、それぞれに、論理式を含める必要があります。

when

これらの場合にのみ呼び出される文節の先頭。

- *criterion* が指定され、when 文節が、*criterion* と等しい *matchExpression* を含む最初の文節である場合。
- *criterion* が指定されず、when 文節が、真と評価される論理式を含む最初の文節である場合。

when 文節が呼び出されても何の効果もないようにしたい場合は、EGL ステートメントを含めずに文節をコーディングします。

case ステートメントには、when 文節がいくつあっても構いません。

matchExpression

次の値のいずれかです。

- 数値またはストリング式
- `ConverseVar.eventKey` または `sysVar.systemType` との比較のためのシンボル

matchExpression 値のプリミティブ型は、*criterion* 値のプリミティブ型と互換性がなければなりません。互換性の詳細については、『論理式』を参照してください。

logicalExpression

論理式。

statement

EGL ステートメント。

otherwise

when 文節が実行していない場合に呼び出される文節の先頭。

when または otherwise 文節中で文が実行された後は、**case** 文の直後の EGL ステートメントに制御が渡されます。どのような状況でも、制御が次の when 文節に渡されることはありません。呼び出されている when 文節がなく、デフォルトの文節が使用されていない場合も、**case** ステートメントが終わった直後に、制御は次の文に渡され、どのエラー状態も有効になりません。

case ステートメントの例は、以下のとおりです。

```
case (myRecord.requestID)
  when (1)
    myFirstFunction();
  when (2, 3, 4)
    try
      call myProgram;
    onException
      myCallFunction(12);
    end
  otherwise
    myDefaultFunction();
end
```

単一の文節に *matchExpression* の複数インスタンス (上の例では 2、3、4) が含まれている場合、これらのインスタンスは左から右に評価され、*criterion* 値と一致する *matchExpression* が 1 つ検出されるとすぐに評価は停止します。

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

93 ページの『EGL ステートメント』

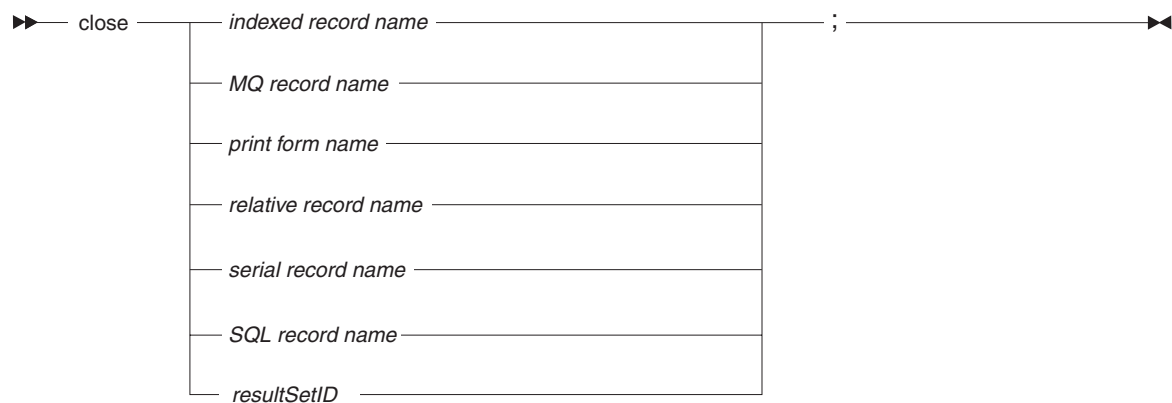
538 ページの『論理式』

987 ページの『eventKey』

1002 ページの『systemType』

close

EGL **close** ステートメントは、プリンターを切断したり、ある特定のレコードに関連付けられたファイルやメッセージ・キューを閉じます。また、SQL レコードの場合は、EGL **open** や **get** ステートメントによってオープンされたカーソルを閉じます。



name

閉じられるリソースに関連付けられた入出力オブジェクトの名前。オブジェクトには、印刷書式や、索引付きレコード、MQ レコード、相対レコード、シリアル・レコード、SQL レコードなどを指定します。

resultSetIdentifier

SQL 処理のみに使用される、同じプログラム内で前に実行された **get** や **open** ステートメントに **close** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

例:

```
if (userRequest == "C")
  try
    close fileA;
  onException
    myErrorHandler(12);
  end
end
```

close ステートメントの振る舞いは、入出力オブジェクトのタイプにより決まります。

索引付きレコード、シリアル・レコード、または相対レコード

close ステートメントで索引付き、シリアル、または相対レコードの名前を使用すると、EGL では、そのレコードに関連付けられたファイルが閉じます。

ファイルが開かれており、そのファイルに関連付けられたリソース名を変更するために `fileAssociation` 項目を使用する場合、EGL ではそのファイルに影響する次の文を実行する前に、そのファイルが自動的に閉じます。詳細については、『*resourceAssociation*』を参照してください。

また、EGL では、プログラムの終了時に開いているすべてのファイルも閉じます。

MQ レコード

`close` ステートメントで MQ レコードの名前を使用すると、EGL ではそのレコードに関連付けられたメッセージ・キューに対して MQSeries コマンド `MQCLOSE` が実行されます。

印刷書式

入出力オブジェクトが印刷書式の場合、`close` ステートメントにより用紙送りが行われ、プリンターが切断されるか、(印刷書式がファイルにスプールされている場合は) ファイルが閉じられます。

`ConverseVar.printerAssociation` を使用して印刷先を変更する前に、`ConverseVar.printerAssociation` の現行値で指定されたプリンターやファイルを閉じてください。複数のプリンターや印刷ファイルが同時にオープンしていることがあるので、それぞれの印刷先ごとに `close` ステートメントオプションを実行します。

EGL ランタイムでは、プログラムの終了時にすべてのプリンターが閉じられていることが確認されます。

SQL レコード

`close` ステートメントで SQL レコードの名前を使用すると、EGL ではそのレコードで開かれた SQL カーソルが閉じられます。

EGL では、以下の場合にカーソルが自動的に閉じます。

- **open** ステートメントの後にカーソル処理ループが続き、No Record Found (NRF) 条件によって、セット内のすべての行が処理されたと示されるまで継続した場合
- 単一行が読み取りで、オプションとして `forUpdate` および `singleRow` のいずれも指定されていない場合に、EGL により SQL レコードに対して **get** ステートメントが実行される場合
- **get** ステートメントにより開かれたカーソルを使用する **replace** または **delete** ステートメントが EGL により実行される場合。この場合、**get** ステートメント中のオプションとして `forUpdate` が指定されます。
- EGL でオープン・カーソルに関連付けられたレコード用の **open** または **get** ステートメントの処理が開始された場合、これらの他の処理をする前にクローズが行われます。
- **sysLib.commit** または **sysLib.rollback** のどちらかがプログラムにより実行される場合。

EGL では、次の場合には開いているすべてのカーソルが閉じます。

- プログラムが `textUI` 型で、書式を会話する前に自動コミットを行う場合。`textUI` プログラムおよび **converse** ステートメントの詳細については、『セグメンテーション』を参照してください。

関連する概念

- 143 ページの『レコード・タイプとプロパティ』
- 799 ページの『resultSetID』
- 170 ページの『テキスト・アプリケーションのセグメンテーション』
- 247 ページの『SQL サポート』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 605 ページの『add』
- 617 ページの『delete』
- 93 ページの『EGL ステートメント』
- 100 ページの『例外処理』
- 619 ページの『execute』
- 631 ページの『get』
- 644 ページの『get next』
- 650 ページの『get previous』
- 579 ページの『入出力エラー値』
- 664 ページの『open』
- 679 ページの『prepare』
- 681 ページの『replace』
- 920 ページの『recordName.resourceAssociation』
- 71 ページの『SQL 項目のプロパティ』
- 956 ページの『commit()』
- 969 ページの『rollback()』
- 988 ページの『printerAssociation』
- 1004 ページの『terminalID』

continue

EGL の **continue** ステートメントは、**for**、**forEach**、**while** のいずれかの文 (これらの文自体も **continue** ステートメントを含んでいる) の末尾に制御を渡します。収容文の実行が続行されるか終了するかは、通常のように、収容文の開始時に実行される論理テストによって決まります。

continue ステートメントは、収容文と同じ関数内に存在する必要があります。



for、forEach、または while

指定したタイプの最も内側の収容文を識別します。これらの文タイプの 1 つを

指定した場合は、そのタイプの文に **continue** ステートメントが含まれている必要があります。文タイプを指定しなかった場合、結果は次のとおりです。

- **continue** ステートメントは、最も内側の収容 **for**、**forEach**、または **while** ステートメントの末尾に制御を渡します。また、
- それらのいずれかのタイプの文に **continue** ステートメントが含まれている必要があります。

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

93 ページの『EGL ステートメント』

converse

EGL **converse** ステートメントは、テキスト・アプリケーションでテキスト書式を表します。

プログラムはユーザーが応答するまで待機し、そのユーザーからテキスト書式を受け取ると、**converse** ステートメントの次の文から処理を続行します。

テキスト書式の処理の概要については、以下のページを順に参照してください。

1. テキスト書式
2. セグメンテーション

►—— converse —— *textFormName* —————; —►

textFormName

プログラムに対して可視になっているテキスト書式の名前。可視性についての詳細は、『パーツの参照』を参照してください。

以下に例を示します。

```
converse myTextForm;
```

以下の考慮事項が該当します。

- テキスト書式との関連でいうと、**converse** ステートメントは常に、呼び出し先プログラムでは有効ですが、セグメント化されているメイン・プログラムを実行している場合は、**converse** ステートメントは以下のようなコードでは無効です。
 - パラメーター、ローカル・ストレージ、または戻り値を持つ関数
 - パラメーター、ローカル・ストレージ、または戻り値を持つ関数によって (直接的または間接的に) 呼び出される関数

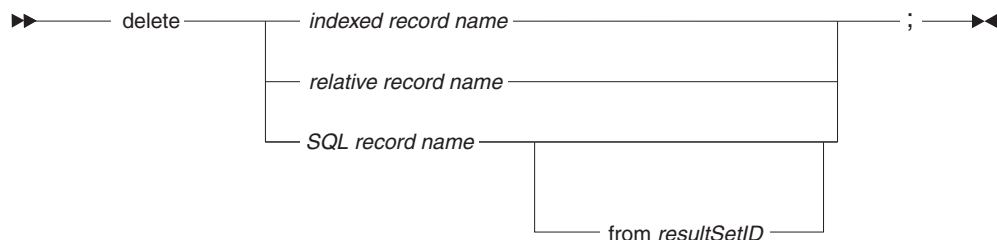
関連する概念

24 ページの『パーツの参照』

170 ページの『テキスト・アプリケーションのセグメンテーション』

delete

EGL delete ステートメントは、ファイルからレコードを除去するか、データベースから行を除去します。



record name

入出力オブジェクトの名前。削除するファイル・レコードまたは SQL 行に関連付けられている索引付きレコード、相対レコード、SQL レコードなどを指定します。

from *resultSetID*

同じプログラム内で前に実行された **get** または **open** ステートメントに **delete** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

以下に例を示します。

```

if (userRequest == "D")
  try
    get myRecord forUpdate;
    onException
      myErrorHandler(12);    // プログラムを終了
  end

  try
    delete myRecord;
    onException
      myErrorHandler(16);
  end
end
end

```

delete の振る舞いは、レコード・タイプによって異なります。SQL 処理の詳細については、『*SOL レコード*』を参照してください。

索引付きレコードまたは相対レコード

索引付きレコードまたは相対レコードを削除するには、以下のようにします。

- レコードに対し **get** ステートメントを発行し、forUpdate オプションを指定します。
- 同じファイルに対する入出力操作を間に挟まずに、**delete** ステートメントを発行します。

get ステートメントを発行した後、同じファイルに対して次に入出力操作を行うと、以下のようになります。

- 次の入出力操作が同じ EGL レコードに対する **replace** ステートメントである場合は、レコードがファイル内で変更される

- 次の入出力操作が同じ EGL レコードに対する **delete** ステートメントである場合は、ファイル内のレコードが削除用にマークされる
- 次の入出力操作が、同じファイルに対する **forUpdate** オプションを含む **get** である場合は、以降の **replace** または **delete** は新しく読み取られるファイル・レコードに対して有効となる
- 次の入出力操作が、同じ EGL レコードに対する **forUpdate** オプションを含まない **get** または同じファイルに対する **close** である場合は、ファイル・レコードは変更なしにリリースされる

forUpdate オプションに関する詳細については、『*get*』を参照してください。

SQL レコード

SQL 処理の場合は、以下のように、次に削除する行を検索するために、EGL **get** または **open** ステートメントに **forUpdate** オプションを使用する 必要があります。

- **get** ステートメントを発行して行を検索する。または
- **open** ステートメントを発行して一連の行を選択してから、**get next** ステートメントを呼び出し、該当する行を検索する。

どちらの場合も、EGL **delete** ステートメントは、カーソル中の現在行を参照する SQL **DELETE** ステートメントにより生成されたコード中に記述されます。この SQL ステートメントは変更できません。これは、次のようなフォーマット設定です。

```
DELETE FROM tableName
WHERE CURRENT OF cursor
```

ユーザー独自の SQL **DELETE** ステートメントを作成するには、EGL **execute** ステートメントを使用します。

単一の EGL **delete** ステートメントを使用して、複数の SQL テーブルから行を除去することはできません。

関連する概念

- 143 ページの『レコード・タイプとプロパティ』
- 799 ページの『*resultSetID*』
- 798 ページの『実行単位』
- 247 ページの『SQL サポート』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 605 ページの『*add*』
- 613 ページの『*close*』
- 93 ページの『EGL ステートメント』
- 100 ページの『例外処理』
- 619 ページの『*execute*』
- 631 ページの『*get*』
- 644 ページの『*get next*』

650 ページの『get previous』
579 ページの『入出力エラー値』
679 ページの『prepare』
664 ページの『open』
681 ページの『replace』
71 ページの『SQL 項目のプロパティ』

display

EGL の **display** ステートメントは、テキスト書式をランタイム・バッファに追加しますが、画面にはデータを表示しません。ランタイムの振る舞いについての詳細は、『テキスト書式』を参照してください。

注: VisualAge Generator との互換性モードで作業している場合は、この文を以下の書式で発行することができます。

```
display printForm;
```

printForm

プログラムに対して可視になっている印刷書式の名前。

その場合、**display** は **print** と等価です。

▶ — display — *textFormName* ————— ; ————— ◀

textFormName

プログラムに対して可視になっているテキスト書式の名前。可視性についての詳細は、『パーツの参照』を参照してください。

関連する概念

24 ページの『パーツの参照』
168 ページの『テキスト書式』

関連する参照項目

681 ページの『print』

execute

EGL **execute** ステートメントでは、SQL ステートメント、特に SQL データ定義ステートメント (CREATE TABLE 型など) やデータ操作ステートメント (INSERT 型、UPDATE 型など) を 1 つ以上作成できます。

ステートメントの例を次に示します (employeeRecord は SQL レコードであることを前提とします)。

```
execute
#sql{
    create table employee (
        empnum decimal(6,0) not null,
        empname char(40) not null,
        empphone char(10) not null)
    };

execute update for employeeRecord;

execute
#sql{
    call aStoredProcedure( :argumentItem)
    };
```

以下の型の SQL ステートメントを発行するために **execute** ステートメントを使用できます。

- ALTER
- CALL
- CREATE ALIAS
- CREATE INDEX
- CREATE SYNONYM
- CREATE TABLE
- CREATE VIEW
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP INDEX
- DROP SYNONYM
- DROP TABLE
- DROP VIEW
- GRANT
- INSERT
- LOCK
- RENAME
- REVOKE
- SAVEPOINT
- SET
- SIGNAL
- UPDATE
- VALUES

以下の型の SQL ステートメントを発行するために **execute** ステートメントを使用することはできません。

- CLOSE
- COMMIT
- CONNECT
- CREATE FUNCTION
- CREATE PROCEDURE
- DECLARE CURSOR
- DESCRIBE

- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- OPEN
- PREPARE
- ROLLBACK WORK
- SELECT
- INCLUDE SQLCA
- INCLUDE SQLDA
- WHENEVER

暗黙の SQL DELETE

暗黙の SQL DELETE ステートメントを指定すると、各 SQL 表キー列の値が SQL レコードの対応するキー項目の値と等しい限り、削除される表の行は、SQL レコード・プロパティ (**defaultSelectCondition**) によって決まります。レコード・キーもデフォルトの選択条件も指定しない場合は、すべての表の行が削除されます。

特定のレコードのデフォルトの暗黙の SQL DELETE ステートメントは、以下のステートメントのようになります。

```
DELETE FROM tableName
WHERE keyColumn01 = :keyItem01
```

単一 EGL ステートメントを使用して、複数のデータベース表から行を削除することはできません。

暗黙の SQL INSERT

暗黙の SQL INSERT ステートメントの効果は、デフォルトでは以下のとおりです。

- テーブル内でのデータの論理位置は、レコード内のキー値によって判別されます。キーを持たないレコードは、SQL 表の定義とデータベースの規則に従って処理されます。
- レコード・パーツにおけるレコード項目と SQL 表の列の関連付けの結果、生成されたコードにより、関連する SQL テーブル列に各レコード項目のデータが格納されます。
- レコード項目が読み取り専用として宣言されている場合、生成された SQL INSERT ステートメントにはそのレコード項目が含まれず、データベース管理システムでは、関連する SQL 表の列の値がデフォルト値 (その列が定義されたときに指定された値) に設定されます。

暗黙の SQL INSERT ステートメントの形式は、以下のとおりです。

```
INSERT INTO tableName
(column01, ... columnNN)
values (:recordItem01, ... :recordItemNN)
```

エラー条件は、以下のとおりです。

- INSERT 以外の型の SQL ステートメントを指定する
- SQL INSERT ステートメントの全部ではなく一部の文節を指定する

- 以下のいずれかの特性を持つ SQL INSERT ステートメントを指定する (または暗黙的な SQL ステートメントを受け入れる)
 - 複数の SQL テーブルに関連付けられている
 - 読み取り専用として宣言したホスト変数のみを組み込む
 - 存在しないか、関連するホスト変数との互換性がない列に関連付けられている

暗黙の SQL UPDATE

暗黙の SQL UPDATE ステートメントの効果は、デフォルトでは以下のとおりです。

- 各 SQL 表キー列の値が SQL レコードの対応するキー項目の値と等しい限り、選択される表の行は、レコード固有の **defaultSelectCondition** と呼ばれる SQL レコード・プロパティによって決まる。レコード・キーもデフォルトの選択条件も指定しない場合は、すべての表の行が更新されます。
- SQL レコード宣言におけるレコード項目と SQL 表の列の関連付けの結果として、特定の SQL 表の列が、関連するレコード項目の内容を受け取ります。ただし、読み取り専用のレコード項目に関連付けられている SQL 表の列は更新されません。

特定のレコードの暗黙の SQL UPDATE ステートメントの形式は、以下のステートメントのようになります。

```
UPDATE tableName
SET   column01 = :recordItem01,
      column02 = :recordItem01, ...
      columnNN = :recordItemNN
WHERE keyColumn01 = :keyItem01
```

以下のいずれかの場合によってエラーが発生します。

- すべての項目が読み取り専用として識別されている
- ステートメントが複数の SQL 表を更新しようとしている
- データベースに書き込まれている項目が実行時に存在しないか、関連する構造体項目との互換性がない列に関連付けられている

関連する概念

143 ページの『レコード・タイプとプロパティ』

247 ページの『SQL サポート』

24 ページの『パーツの参照』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

605 ページの『add』

613 ページの『close』

617 ページの『delete』

93 ページの『EGL ステートメント』

100 ページの『例外処理』

631 ページの『get』

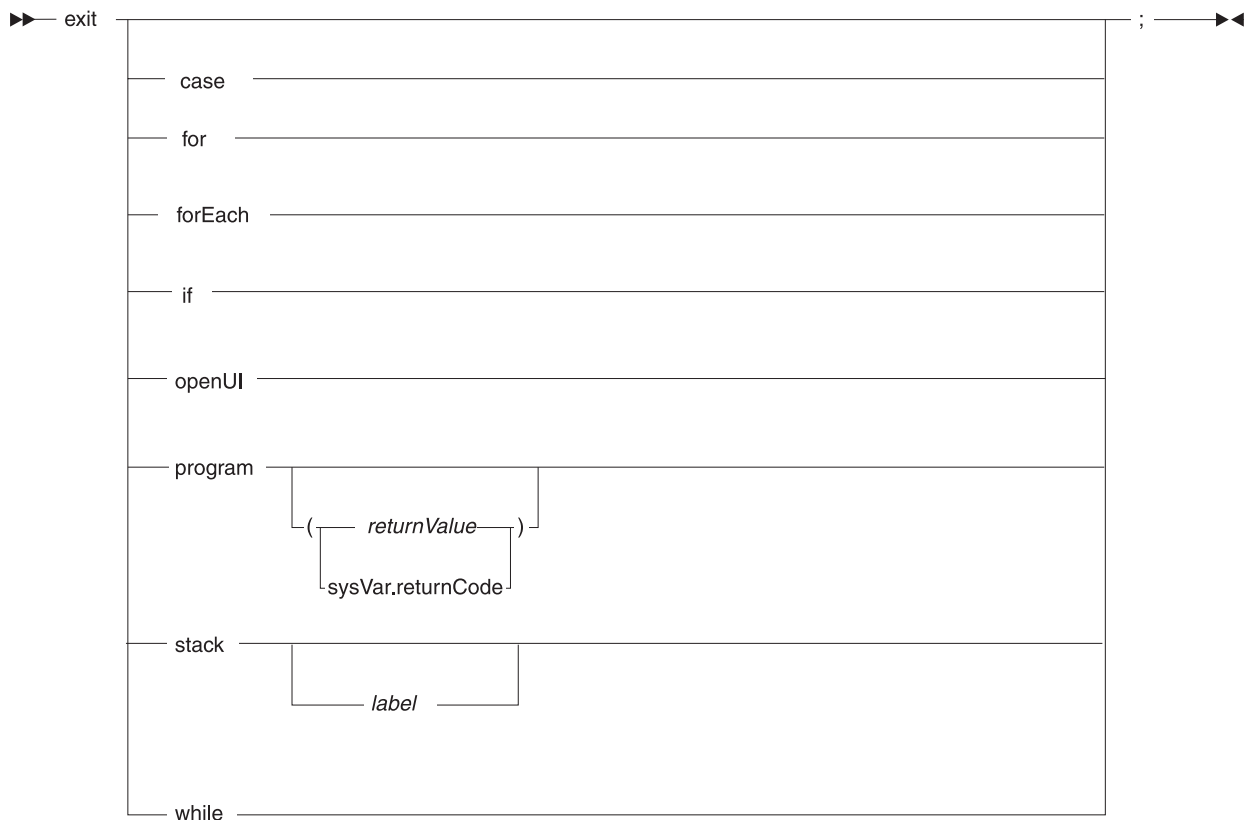
644 ページの『get next』

650 ページの『get previous』

579 ページの『入出力エラー値』
 664 ページの『open』
 679 ページの『prepare』
 681 ページの『replace』
 71 ページの『SQL 項目のプロパティ』
 1004 ページの『terminalID』

exit

EGL **exit** ステートメントは、指定したブロックを出ます。このブロックは、デフォルトでは、即時に **exit** ステートメントを含むブロックです。



case

最新に入力された **exit** ステートメントがある **case** ステートメントを出ます。

case ステートメント後に処理を続けます。

exit ステートメントは、同じ関数で開始される **case** ステートメント内にはありません。

for

exit ステートメントが存在する最新に入力された **for** ステートメントから出ます。**for** ステートメントの後へ処理を進めます。

exit ステートメントが、同じ関数内で始まる **for** ステートメントの内部にない場合は、エラーが発生します。

forEach

exit ステートメントが存在する最新に入力された **forEach** ステートメントから出ます。 **forEach** ステートメントの後へ処理を進めます。

exit ステートメントが、同じ関数内で始まる **forEach** ステートメントの内部にない場合は、エラーが発生します。

if 最新に入力された **exit** ステートメントがある **if** ステートメントを出ます。 **if** ステートメント後に処理を継続します。

exit ステートメントは、同じ関数で開始される **if** ステートメント内にはありません。

program

プログラムを出ます。

以下のいずれかの場合は、システム変数 **sysVar.returnValue** の値がオペレーティング・システムに戻されます。

- プログラムが、戻りコードが含まれていない **exit** ステートメントで終了している
- プログラムが、**sysVar.returnValue** を戻す **exit** ステートメントで終了している
- プログラムが、終了 **exit** ステートメントなしで終了している

プログラムが、**sysVar.returnValue** 以外の戻りコードを含んでいる終了の **exit** ステートメントで終了する場合は、指定された値が、**sysVar.returnValue** 内の値の代わりに使用されます。

returnValue

リテラル整数、または整数へと解決される項目、定数、または数式。戻り値は、オペレーティング・システムから使用可能になり、その範囲は -2147483648 以上 2147483647 以下であることが必要です。

戻り値の詳細については、『**sysVar.returnValue**』を参照してください。

sysVar.returnValue

オペレーティング・システムに戻される値が含まれているシステム変数。

詳細については、『**sysVar.returnValue**』を参照してください。

stack

現在の関数に対する戻り値を設定せずに **main** 関数に制御を戻します。

exit stack という書式の文は、ランタイム *stack* の中間関数に対するすべての参照を除去します。ランタイム *stack* は、関数 (具体的には、現在の関数、および実行すると現在の関数が実行される一連の関数) のリストです。

main 関数は、ある関数 (現在スタック内にある) を呼び出し、その呼び出しに、修飾子 **out** または **inOut** を持つパラメーターが含まれていた可能性があります。そのような場合は、*exit stack* という書式の **exit** ステートメントにより、パラメーターの値を **main** 関数から使用できるようになります。

label (後述) を指定しない場合は、**main** 関数内の最新の **run** 関数呼び出し後に文で処理が継続されます。ラベルを指定すると、**main** 関数のラベルのすぐ後の文で処理が継続されます。ラベルは、**main** 関数内で最新に実行された関数呼び出しの前でも後でもかまいません。

main 関数に *exit stack* の書式の *exit* ステートメントを指定すると、ラベルを指定した場合でも次の文が処理されます。現在の関数の指定したラベルに移動する方法の詳細については、『*goTo*』を参照してください。

label

main 関数、およびブロック以外に表示される以下のような一連の文字。

- if
- else
- **case** ステートメント内
- while
- try

処理が継続されているロケーションに表示されるラベルは、後ろにコロンが付きます。ラベルの有効な文字についての詳細は、『*命名規則*』を参照してください。

関連する参照項目

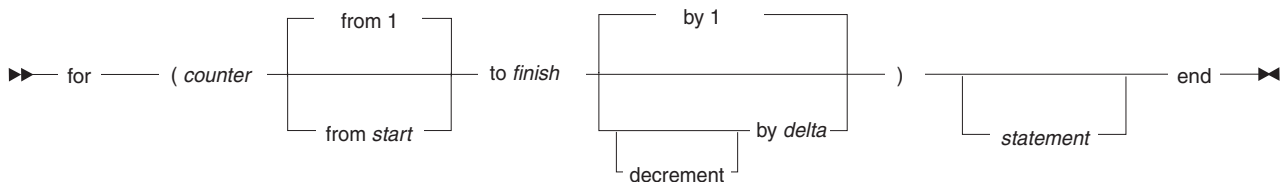
656 ページの『*goTo*』

725 ページの『*命名規則*』

998 ページの『*returnCode*』

for

EGL キーワード **for** は、ループ内でテストによって真と評価された回数だけ実行される文ブロックを開始します。このテストはループの開始時に実行され、カウンターの値が指定された範囲内であるかどうかを示します。キーワード **end** は、**for** ステートメントの終わりを示します。



counter

小数点以下の桁を持たない数値変数。**for** ステートメント内の EGL ステートメントは、*counter* の値を変更できます。

from start

counter の初期値。**from** で始まる文節を指定しなかった場合、初期値は 1 になります。

start は、以下のいずれかにすることができます。

- 整数リテラル
- 小数点以下の桁を持たない数値変数
- 必ず整数へと解決される数式

to finish

decrement を指定しなかった場合は、*finish* が *counter* の上限です。*counter* の

値がその限度を超えると、前に述べたテストが偽へと解決され、文ブロックはそれ以上実行されず、**for** ステートメントは終了します。

decrement を指定した場合は、*finish* が *counter* の下限です。*counter* の値がその限度を下回ると、テストは偽へと解決され、文ブロックはそれ以上実行されず、**for** ステートメントは終了します。

finish は、以下のいずれかにすることができます。

- 整数リテラル
- 小数点以下の桁を持たない数値変数
- 必ず整数へと解決される数式

for ステートメント内の EGL ステートメントは、*finish* の値を変更できます。

by *delta*

decrement を指定しなかった場合は、EGL ステートメントブロックが実行された後、*counter* の値がテストされる前に、*delta* の値が *counter* に加算されます。

decrement を指定した場合は、EGL ステートメントブロックが実行された後、*counter* の値がテストされる前に、*delta* の値が *counter* から減算されます。

delta は、以下のいずれかにすることができます。

- 整数リテラル
- 小数点以下の桁を持たない数値変数
- 必ず整数へと解決される数式

for ステートメント内の EGL ステートメントは、*delta* の値を変更できます。

statement

EGL 言語の文

以下に例を示します。

```
sum = 0;

// sum に 10 個の値を加算する
for (i from 1 to 10 by 1)
    sum = inputArray[i] + sum;
end
```

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

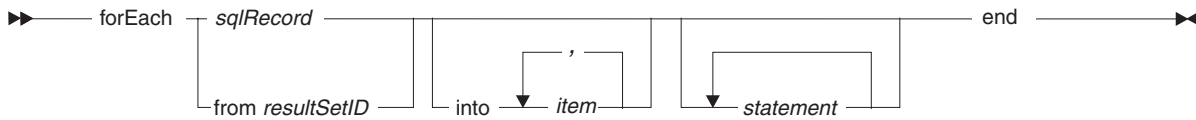
93 ページの『EGL ステートメント』

forEach

EGL キーワード **forEach** は、ループで実行される一連の文の始まりを示します。最初の反復が発生するのは、指定された結果セットが使用可能である場合だけです。（その結果セットが使用可能でない場合、この文はハード・エラーで失敗します。）ループでは、以下のいずれかのイベントが発生するまで、結果セット内の各行の読み取りが続行されます。

- すべての行が取り出される。

- **exit** ステートメントが実行される。または
- ハード・エラーまたはソフト・エラーが発生する。



sqlRecord

直前に実行された **open** ステートメントで使用された SQL レコードの名前。SQL レコードまたは結果セット ID を指定する必要があり、結果セット ID を指定することが推奨されます。

from *resultSetID*

直前に実行された **open** ステートメントで使用された結果セット ID。詳細については、『*resultSetID*』を参照してください。

into ... item

カーソルまたはストアード・プロシージャから値を受け取る EGL ホスト変数を識別する INTO 文節です。 **#sql{ }** ブロックの外側にある、このような文節では、ホスト変数名の前にセミコロンを含めないでください。

このコンテキストで INTO 文節を指定すると、関連する **open** ステートメントで示されている INTO 文節はオーバーライドされます。

statement

EGL 言語の文

ほとんどの場合、EGL ランタイムは **foreach** ステートメントの最後の反復の後に実行される **close** ステートメントを暗黙に発行します。その暗黙のステートメントは、SQL システム変数を変更します。その理由は、ユーザーが SQL 関連システム変数の値を **foreach** ステートメントの本体内に保管したい場合があるからです。

EGL ランタイムは、**foreach** ステートメントが **noRecordFound** 以外のエラーのために終了した場合には、暗黙の **close** ステートメントを発行しません。

以下に例を示します。詳細については、『*SQL 例*』を参照してください。

```
VGVar.handleHardIOErrors = 1;

try
  open selectEmp
    with #sql{
      select empnum, empname
      from employee
      where empnum >= :empnum
      for update of empname
    }
    into empnum, empname;
onException
  myErrorHandler(6);    // プログラムを終了する
end

try
  foreach (from selectEmp)
    empname = empname + " " + "III";

  try
```

```

execute
  #sql{
    update employee
    set empname = :empname
    where current of selectEmp
  };
onException
  myErrorHandler(10); // プログラムを終了する
end
end // end forEach 結果セットの最後の行が読み取られると
// カーソルは自動的にクローズされる

onException
  // 条件が "sqlcode = 100" の場合、forEach に関連した例外ブロックは
  // 実行されない
  // "if (sqlcode != 100)" のテストは避ける。
  myErrorHandler(8); // プログラムを終了する
end

sysLib.commit();

```

関連する概念

799 ページの『resultSetID』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

93 ページの『EGL ステートメント』

624 ページの『exit』

664 ページの『open』

260 ページの『SQL 例』

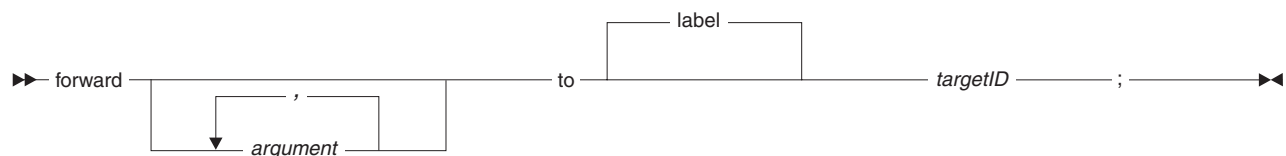
forward

EGL **forward** ステートメントはページ・ハンドラーから呼び出されます。その主な目的は、可変情報を Web ページに表示することですが、この文は、Web アプリケーション・サーバー内で実行されているサーブレットまたは Java プログラムを呼び出すこともできます。

この文の動作は、以下のとおりです。

1. 回復可能リソースのコミット、ファイルのクローズ、ロックの解除
2. 制御権の移動
3. **forward** ステートメントを実行しているコードを終了する。

構文図は以下のとおりです。



argument

呼び出されているコードに渡される項目またはレコード。引数の名前、およびその引数に対応するパラメーターの名前は、すべての場合で同一である必要があります。リテラルは渡せません。

ページ・ハンドラーを呼び出している場合、引数は、ページ・ハンドラーの **onPageLoad** 関数に指定されたパラメーターと互換性がなければなりません。この関数がある場合には、任意の有効な名前を付けることができ、この関数はページ・ハンドラーのプロパティ **OnPageLoadFunction** によって参照されます。プログラムを呼び出している場合は、引数には、そのプログラムのパラメーターとの互換性が必要です。

このテクノロジーの使い方によっては、以下の詳細が役に立つかもしれません。

- 引数は、それに対応するパラメーターと同じ名前である必要があります。それは、この名前は、Web アプリケーション・サーバーに引数値を保管し、それを取り出すときのキーとして使用されるからです。
- 引数を渡す代わりに、呼び出し側は **forward** ステートメントを呼び出す前に、以下を行うことができます。
 - システム関数 `J2EELib.setRequestAttr` を呼び出して、値を要求ブロックに配置する
 - システム関数 `J2EELib.setSessionAttr` を呼び出して、値をセッション・ブロックに配置する

この場合、受信側はこの値を引数として受け取らずに、以下の適切なシステム関数を呼び出します。

- `J2EELib.getRequestAttr` (要求ブロックからデータにアクセスする場合)、または
- `J2EELib.getSessionAttr` (セッション・ブロックからデータにアクセスする場合)
- 文字項目は、Java `String` 型のオブジェクトとして渡されます。
- レコードは、Java `Bean` として渡されます。

to label *targetID*

Java Server Faces (JSF) ラベルを指定します。このラベルは、ランタイムの JSF ベースの構成ファイル内のマッピングを識別します。次に、呼び出すオブジェクトが JSP (通常は、EGL ページ・ハンドラーに関連付けられている JSP)、EGL プログラム、EGL 以外のプログラム、またはサーブレットのいずれであるのかを、このマッピングが識別します。**label** という語はオプションで、*targetID* は引用符で囲まれたストリングです。

関連する参照項目

- 560 ページの『関数呼び出し』
- 862 ページの『`getRequestAttr()`』
- 862 ページの『`getSessionAttr()`』
- 1005 ページの『`transferName`』

freeSQL

EGL の **freeSQL** ステートメントは、動的に準備された SQL ステートメントに関連したリソースを解放し、その SQL ステートメントに関連したすべてのオープン・カーソルをクローズします。

►—freeSQL *preparedStatementID*—————; —◄

preparedStatementID

prepare ステートメントを識別する ID。その文が前に実行されていない場合、エラーは発生しません。

freeSQL ステートメントを発行した後、準備された SQL ステートメント用に **execute**、**open**、**get** ステートメントを実行するには、**prepare** ステートメントを再発行する必要があります。

関連する概念

247 ページの『SQL サポート』

関連する参照項目

619 ページの『execute』

『get』

664 ページの『open』

679 ページの『prepare』

812 ページの『EGL ステートメントおよびコマンドの構文図』

get

EGL **get** ステートメントは、単一ファイル・レコードまたはデータベース行を検索し、コード内で後で格納されたデータを置換または削除することができるオプションを提供します。また、このステートメントを使用してデータベース行の集合を検索し、それぞれの後続の行を動的配列の次の SQL レコードに入れることができます。

get ステートメントは、**get by key value** として識別されることがあり、**get** ワードで始まるその他の文とは区別されます。

#sql{ *sqlStatement* }

『SQL サポート』に説明した明示的な SQL SELECT ステートメント。**#sql** と左中括弧の間にスペースを入れないでください。

into ... *item*

リレーショナル・データベースから値を受け取る EGL ホスト変数を識別する EGL INTO 文節。この文節は、以下のいずれかの場合に SQL を処理するときが必要です。

- SQL レコードが指定されていない。
- SQL レコードと明示的な SQL SELECT ステートメントの両方が指定されているが、SQL SELECT 文節の列がレコード項目と関連付けられていない。(関連は、『SQL 項目プロパティ』に記載されているように SQL レコード・パーツに指定されます。)

このような文節 (**#sql{ }** ブロックの外側にある) では、ホスト変数名の前にセミコロンを含めないでください。

preparedStatementID

実行時に SQL SELECT ステートメントを作成する EGL **prepare** ステートメントの ID。**get** ステートメント SQL SELECT ステートメントを動的に実行します。詳細については、『*prepare*』を参照してください。

using ... *item*

実行時に準備済み SQL SELECT ステートメントに使用可能な EGL ホスト変数を識別する USING 文節。このような文節 (**sql-and-end** ブロックの外側にある) では、ホスト変数名の前にセミコロンを含めないでください。

usingKeys ... *item*

暗黙の SQL ステートメントの WHERE 文節のキー値コンポーネントを作成するために使用するキー項目のリストを識別します。暗黙の SQL ステートメントは、明示的な SQL ステートメントを指定しない場合に実行時に使用されます。

usingKeys 文節を指定しない場合、暗黙のステートメントのキー値コンポーネントは **get** ステートメントで参照される SQL レコード・パーツに基づくか、**get** ステートメントで参照される動的配列の基礎になります。

動的配列の場合、**usingKeys** 文節の項目 (または SQL レコードのホスト変数) は、動的配列の基礎である SQL レコードに存在しない必要があります。

明示的な SQL ステートメントを指定する場合、**usingKeys** 情報は無視されます。

SQL dynamic array

SQL レコードによって構成される動的配列の名前。

以下の例はファイル・レコードを読み取り、置換する方法を示します。

```
emp.empnum = 1;           // sets the key in record emp

try
  get emp forUpdate;
onException
  myErrorHandler(8);      // プログラムを終了する
end

emp.empname = emp.empname + " Smith";
```



```

try
    replace emp;
onException
    myErrorHandler(12);
end

```

次の **get** ステートメントは、データベース行を検索するときに SQL レコード emp を使用し、以降の更新または削除を実行することはできません。

```

try
    get emp singleRow into empname with
        #sql{
            select empname
            from Employee
            where empnum = :empnum
        };
onException
    myErrorHandler(8);
end

```

次の例は、同じ SQL レコードを使用して SQL 行を置換します。

```

try
    get emp forUpdate into empname with
        #sql{
            select empname
            from Employee
            where empnum = :empnum
        };

onException
    myErrorHandler(8);    // プログラムを終了する
end

emp.empname = emp.empname + " Smith";

try
    replace emp;
onException
    myErrorHandler(12);
end

```

get ステートメントの詳細は、レコード・タイプによって異なります。SQL 処理の詳細については、『SQL レコード』を参照してください。

索引付きレコード

索引付きレコードに対して **get** ステートメントを発行するときは、レコードのキー値によって、ファイルから検索されるレコードが決定されます。

索引付き (または相対) レコードを置換または削除する場合は、レコードに対して **get** ステートメントを発行し、同じファイルに対する入出力操作を間に挟まずに、ファイル変更文 (**replace** または **delete**) を発行する必要があります。**get** ステートメントを発行した後、同じファイルに対して次に入出力操作を行うと、以下のようになります。

- 次の入出力操作が同じ EGL レコードに対する **replace** ステートメントである場合は、レコードがファイル内で変更される
- 次の入出力操作が同じ EGL レコードに対する **delete** ステートメントである場合は、ファイル内のレコードが削除用にマークされる

- 次の入出力操作が、同じファイル中のレコードに対する **forUpdate** オプションを含む **get** ステートメントである場合は、以降の **replace** または **delete** ステートメントは新しく読み取られるファイル・レコードに対して有効となる
- 次の入出力操作が、同じ EGL レコードに対する **forUpdate** オプションを含まない **get** ステートメント、または同じファイルに対する **close** ステートメントである場合は、ファイル・レコードは変更なしにリリースされる

ファイルが VSAM ファイルである場合、EGL **get** ステートメント (**forUpdate** オプション指定) の間、他のプログラムからこのレコードを変更することはできません。

相対レコード

相対レコードに対して **get** を発行するときは、レコードに関連付けられたキー項目によって、ファイルから検索されるレコードが決定されます。キー項目は、そのレコードを使用するすべての関数で使用できなければなりません。以下のいずれかをキー項目とすることができます。

- 同じレコード内の項目
- プログラムに対してグローバルであるか、または **get** ステートメントを実行している関数に対してローカルであるレコード内の項目
- プログラムに対してグローバルであるか、または **get** ステートメントを実行している関数に対してローカルであるデータ項目

索引付き (または相対) レコードを置換または削除する場合は、レコードに対して **get** ステートメントを発行し、同じファイルに対する入出力操作を間に挟まずに、ファイル変更文 (**replace** または **delete**) を発行する必要があります。**get** ステートメントを発行した後、同じファイルに対して次に入出力操作を行うと、以下のようになります。

- 次の入出力操作が同じ EGL レコードに対する **replace** ステートメントである場合は、レコードがファイル内で変更される
- 次の入出力操作が同じ EGL レコードに対する **delete** ステートメントである場合は、ファイル内のレコードが削除用にマークされる
- 次の入出力操作が、同じファイルに対する **forUpdate** オプションを含む **get** である場合は、以降の **replace** または **delete** は新しく読み取られるファイル・レコードに対して有効となる
- 次の入出力操作が、同じ EGL レコードに対する **forUpdate** オプションを含まない **get** または同じファイルに対する **close** である場合は、ファイル・レコードは変更なしにリリースされる

SQL レコード

EGL **get** ステートメントにより、生成されるコード内で SQL **SELECT** ステートメントが作成されます。**singleRow** オプションを指定する場合、SQL **SELECT** ステートメントは、スタンドアロン・ステートメントです。または、SQL **SELECT** ステートメントは、『SQL サポート』で説明するようにカーソル内の文節です。

エラー条件: 以下の条件は、**get** ステートメントを使用してリレーショナル・データベースからデータを読み取るときには有効ではありません。

- **SELECT** 以外の型の SQL ステートメントを指定する

- SQL SELECT ステートメントに直接 SQL INTO 文節を指定する
- SQL INTO 文節以外に、SQL SELECT ステートメントの全部ではなく一部の文節を指定する場合
- 存在しないか、関連するホスト変数との互換性がない列に関連付けられている SQL SELECT ステートメントを指定する（または受け入れる）場合

以下のエラー条件は、forUpdate オプションを使用する場合に発生することがあります。

- 複数の表を更新する意図を示す SQL ステートメントを指定する（または受け入れる）場合
- 入出力オブジェクトとして SQL レコードを使用し、すべてのレコード項目が読み取り専用である場合

また、以下の状態でエラーが生じることがあります。

- forUpdate オプションを指定して EGL **get** ステートメントをカスタマイズしたが、特定の SQL 表の列が update で使用可能であることを示していない場合
- **get** ステートメントに関連付けられた replace ステートメントが列を変更しようとした。

この直前の不一致は、以下のいずれかの方法で解決できます。

- EGL **get** ステートメントをカスタマイズする際に、使用する列名を SQL SELECT ステートメントの FOR UPDATE OF 文節に組み込む
- EGL replace ステートメントをカスタマイズする際に、SQL UPDATE ステートメントの SET 文節内にある列への参照を除去する
- **get** と **replace** ステートメントの両方について、デフォルトを受け入れる

暗黙の SQL SELECT ステートメント: **get** ステートメントに SQL レコードを入出力オブジェクトとして指定するが、明示的な SQL ステートメントを指定しない場合、暗黙の SQL SELECT は以下の特性を持ちます。

- 各 SQL 表のキー列の値が SQL レコードの対応するキー項目の値と等しい限り、選択される表の行は、レコード固有の **defaultSelectCondition** と呼ばれるプロパティによって決まる。レコード・キーとデフォルトの選択条件のどちらも指定しない場合は、すべての表の行が選択されます。何らかの理由で複数の表の行が選択された場合は、最初に検索された行がレコードに入られます。
- レコード定義におけるレコード項目と SQL 表の列の関連付けの結果として、特定の項目が、関連する SQL 表の列の内容を受け取る。
- forUpdate オプションを指定した場合、SQL SELECT FOR UPDATE ステートメントには、読み取り専用のレコード項目が含まれない
- 特定のレコード用の SQL SELECT ステートメントは、以下のステートメントのようになります。ただし、FOR UPDATE OF 文節は、**get** ステートメントに forUpdate オプションが含まれる場合にのみ使用します。

```
SELECT column01,
       column02, ...
       columnNN
FROM   tableName
WHERE  keyColumn01 = :keyItem01
```

```
FOR UPDATE OF
    column01,
    column02, ...
    columnNN
```

スタンドアロンの SQL SELECT またはカーソルに関連する FETCH ステートメントの SQL INTO 文節は、以下の文節のようになります。

```
INTO    :recordItem01,
        :recordItem02, ...
        :recordItemNN
```

EGL では、INTO 文節を指定しないと、SQL レコードが 明示的な SQL SELECT ステートメントを伴わない場合、SQL INTO 文節が派生されます。派生した INTO 文節の項目は、SQL ステートメントの SELECT 文節にリストされた列と関連づけられた項目です。(項目と列の関連付けは、『SQL 項目プロパティ』に記載されているように SQL レコード・パーツに指定されます。) 列が項目と関連付けられていない場合は、EGL INTO 文節が必要です。

get ステートメントに SQL レコードの動的配列を入出力オブジェクトとして指定するが、明示的な SQL ステートメントを指定しない場合は、暗黙の SQL SELECT は単一 SQL レコードに関して説明した暗黙の SQL SELECT と類似していますが、以下の相違があります。

- 照会のキー値コンポーネントは、より大か等しい条件に基づく関係の集合です。

```
keyColumn01 >= :keyItem01 &
keyColumn02 >= :keyItem02 &
.
.
.
keyColumnN  >= :keyItemN
```

- **usingKeys** 文節の項目 (SQL レコードのホスト変数) は、動的配列の基礎である SQL レコードに存在しない必要があります。

関連する概念

334 ページの『作業論理単位』
 143 ページの『レコード・タイプとプロパティ』
 24 ページの『パーツの参照』
 799 ページの『resultSetID』
 247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

605 ページの『add』
 613 ページの『close』

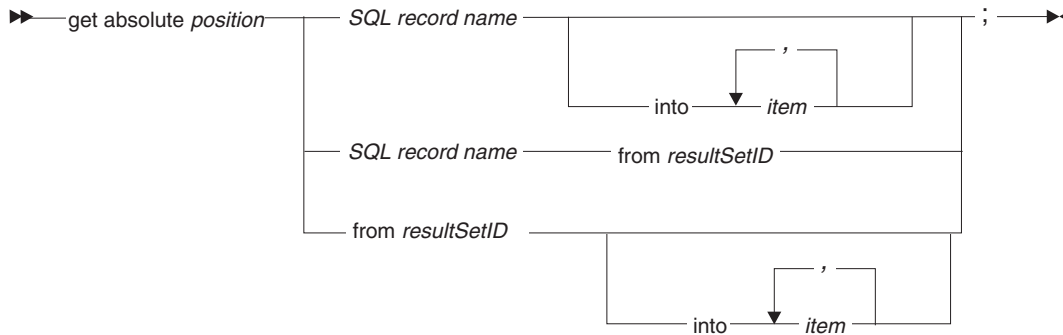
 617 ページの『delete』
 93 ページの『EGL ステートメント』
 100 ページの『例外処理』
 619 ページの『execute』
 644 ページの『get next』
 650 ページの『get previous』

579 ページの『入出力エラー値』
 664 ページの『open』
 679 ページの『prepare』
 681 ページの『replace』
 71 ページの『SQL 項目のプロパティー』
 1004 ページの『terminalID』

get absolute

EGL の **get absolute** ステートメントは、リレーショナル・データベース結果セット内の、番号で指定された行を読み取ります。行は、結果セットの先頭からの相対位置 (正の値を指定した場合) か、結果セットの終わりからの相対位置 (負の値を指定した場合) で、識別されます。

この文を使用できるのは、関連する **open** ステートメントで **scroll** オプションを指定した場合だけです。



position

整数の項目またはリテラル。

position の値が正の場合、行は結果セットの先頭からの相対位置で識別されます。例えば、**get absolute 1** を指定すると最初の行が取り出され、**get first** を指定したのと同じことになります。**get absolute 2** を指定すると、2 番目の行が取り出されます。

position の値が負の場合、行は結果セットの終わりからの相対位置で識別されます。例えば、**get absolute -1** を指定すると最後の行が取り出され、**get last** を指定したのと同じことになります。**get absolute -2** を指定すると、最後から 2 番目の行が取り出されます。

position の値にゼロを指定すると、『例外処理』の説明のように、ハード・エラーになります。

record name

SQL レコードの名前。

from resultSetID

同じプログラム内で前に実行された **open** ステートメントに **get absolute** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

get absolute ステートメントを発行し、forUpdate オプション付きの **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get absolute** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外は変更できません。

結果セットにない行へのアクセスを試みる **get absolute** ステートメントを発行すると、EGL ランタイムは次のように動作します。

- 結果セットからデータをコピーしません。
- カーソルを開いたままに残し、カーソル位置は変更されません。
- SQL レコード (もしあれば) を **noRecordFound** に設定します。

一般に、エラーが発生して処理が続行された場合、カーソルは開いたままに残り、カーソル位置は変更されません。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

799 ページの『*resultSetID*』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

617 ページの『*delete*』

100 ページの『例外処理』

619 ページの『*execute*』

631 ページの『*get*』

640 ページの『*get current*』

641 ページの『*get first*』

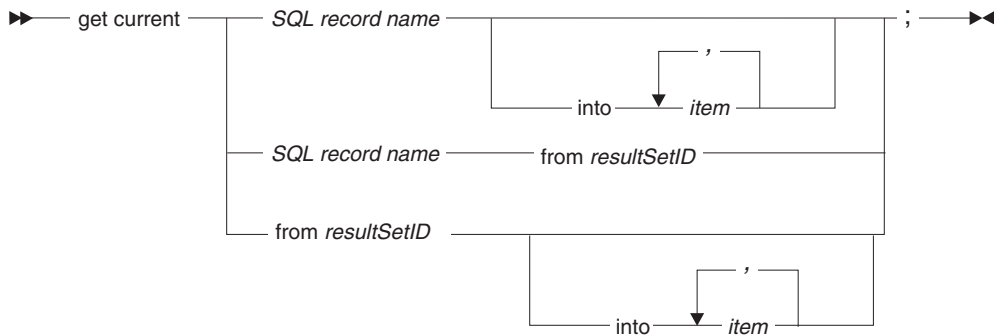
643 ページの『*get last*』

644 ページの『get next』
650 ページの『get previous』
654 ページの『get relative』
93 ページの『EGL ステートメント』
664 ページの『open』
681 ページの『replace』

get current

EGL の **get current** ステートメントは、リレーショナル・データベース結果セット内ですでにカーソルが位置付けられている行を読み取ります。

この文を使用できるのは、関連する **open** ステートメントで **scroll** オプションを指定した場合だけです。



record name

SQL レコードの名前。

from *resultSetID*

同じプログラム内で前に実行された **open** ステートメントに **get current** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

get current ステートメントを発行し、forUpdate オプション付きの **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get current** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外は変更できません。

エラーが発生して処理が続行された場合、カーソルは開いたまま残ります。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

799 ページの『resultSetID』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

617 ページの『delete』

619 ページの『execute』

631 ページの『get』

638 ページの『get absolute』

『get first』

643 ページの『get last』

644 ページの『get next』

650 ページの『get previous』

654 ページの『get relative』

93 ページの『EGL ステートメント』

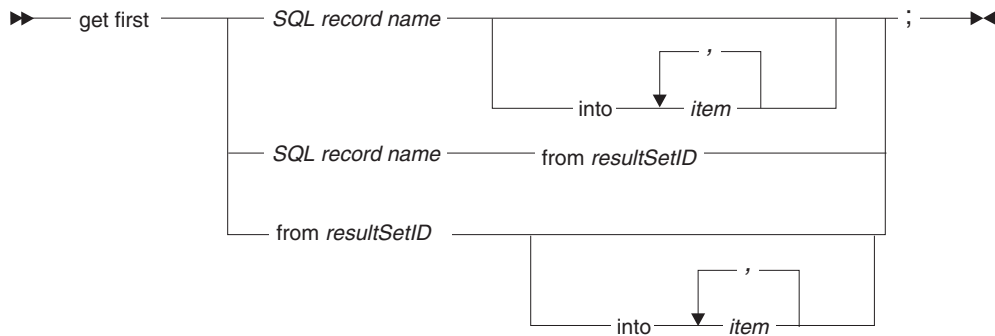
664 ページの『open』

681 ページの『replace』

get first

EGL の **get first** ステートメントは、リレーショナル・データベース結果セット内の最初の行を読み取ります。

この文を使用できるのは、関連する **open** ステートメントで scroll オプションを指定した場合だけです。



record name

SQL レコードの名前。

from *resultSetID*

同じプログラム内で前に実行された **open** ステートメントに **get first** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

get first ステートメントを発行し、forUpdate オプション付きの **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get first** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外の変更できません。

エラーが発生して処理が続行された場合、カーソルは開いたままに残ります。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

799 ページの『*resultSetID*』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

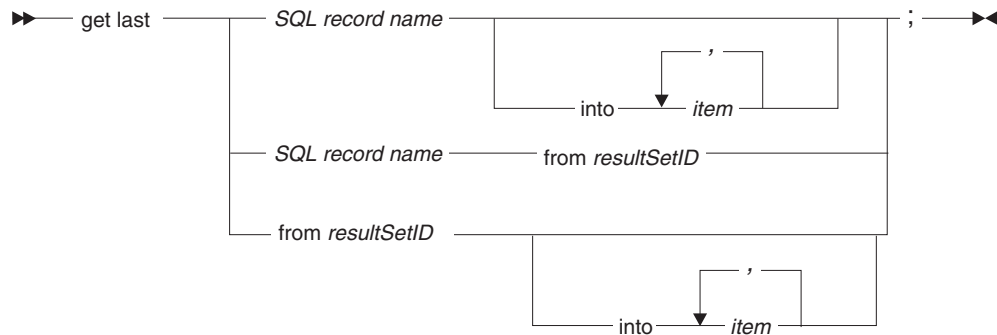
617 ページの『*delete*』

619 ページの『execute』
 631 ページの『get』
 638 ページの『get absolute』
 640 ページの『get current』
 『get last』
 644 ページの『get next』
 650 ページの『get previous』
 654 ページの『get relative』
 93 ページの『EGL ステートメント』
 664 ページの『open』
 681 ページの『replace』

get last

EGL の **get last** ステートメントは、リレーショナル・データベース結果セット内の最後の行を読み取ります。

この文を使用できるのは、関連する **open** ステートメントで **scroll** オプションを指定した場合だけです。



record name

SQL レコードの名前。

from *resultSetID*

同じプログラム内で前に実行された **open** ステートメントに **get last** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

get last ステートメントを発行し、forUpdate オプション付きの **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する

- EGL **execute** ステートメントで行を変更または除去する

SQL **FETCH** ステートメントは、生成されたコードでは EGL **get last** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外は変更できません。

エラーが発生して処理が続行された場合、カーソルは開いたままです。

最終的に、SQL **COMMIT** または `sysLib.commit` が指定された場合、**open** ステートメントで **hold** オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

799 ページの『[resultSetID](#)』

247 ページの『[SQL サポート](#)』

関連するタスク

812 ページの『[EGL ステートメントおよびコマンドの構文図](#)』

関連する参照項目

617 ページの『[delete](#)』

619 ページの『[execute](#)』

631 ページの『[get](#)』

638 ページの『[get absolute](#)』

640 ページの『[get current](#)』

641 ページの『[get first](#)』

『[get next](#)』

650 ページの『[get previous](#)』

654 ページの『[get relative](#)』

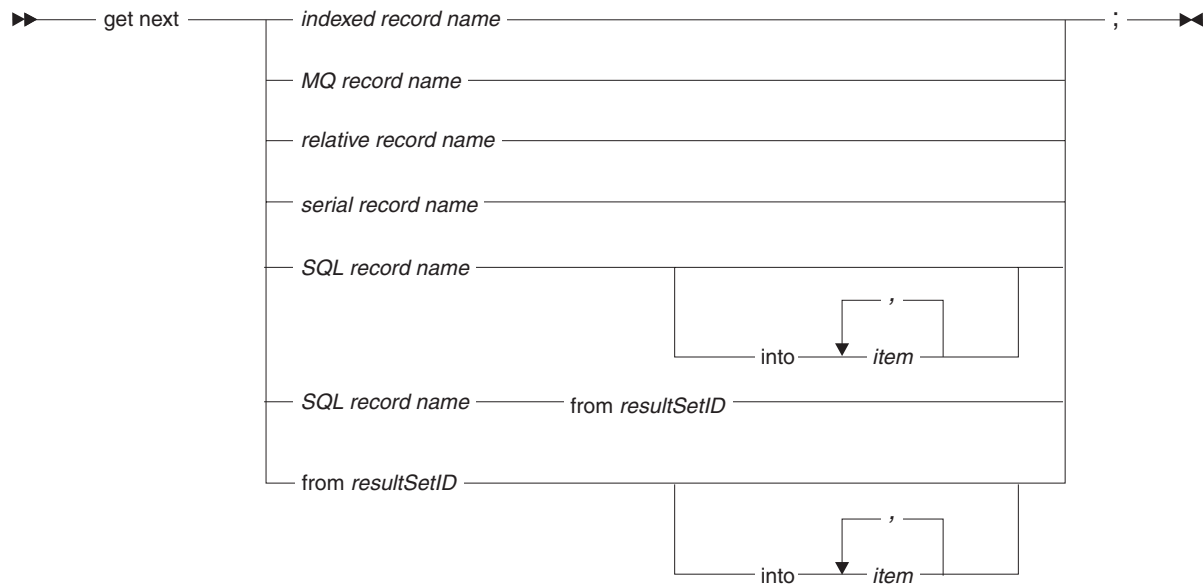
93 ページの『[EGL ステートメント](#)』

664 ページの『[open](#)』

681 ページの『[replace](#)』

get next

EGL **get next** ステートメントは、ファイルまたはメッセージ・キューから次のレコードを読み取るか、データベースから次の行を読み取ります。



record name

入出力オブジェクトの名前。索引付きレコード、MQ レコード、相対レコード、シリアル・レコード、SQL レコードなどを指定します。

from *resultSetID*

SQL 処理のみに使用する、同じプログラム内で前に実行された **open** ステートメントに **get next** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

ファイル・アクセスの例を次に示します。

```

try
    open record1 forUpdate;
    onException
        myErrorHandler(8);
    return;
end
try
    get next record1;
    onException
        myErrorHandler(12);
    return;
end

while (record1 not endOfFile)
    makeChanges(record1); // レコードのプロセス

    try
        replace record1;
    onException
        myErrorHandler(16);
    return;
end

```

```

    try
        get next record1;
    onException
        myErrorHandler(12);
    return;
end
end // end while

sysLib.commit();

```

get next ステートメントの詳細は、レコード・タイプによって異なります。SQL 処理の詳細については、『649 ページの『SQL 処理』』を参照してください。

索引付きレコード

get next ステートメントが索引付きレコードに対して操作を行う場合、効果は現在のファイル位置に基づいたものとなります。この位置は、以下のどちらかの操作によって設定されます。

- **get** ステートメントまたは別の **get next** ステートメントなどの正常な入出力 (I/O) 操作
- *set record position* という書式の **set** ステートメント

規則は以下のとおりです。

- ファイルが開かれていない場合は、**get next** ステートメントはファイル内のキー値が最も低いレコードを読み取る。
- 以降の各 **get next** ステートメントは、現在のファイル位置との関係でキー値が次に高いレコードを読み取る。複写キーの場合の例外については、後で説明します。
- **get next** ステートメントがファイル内のキー値が最も高いレコードを読み取った後、次の **get next** ステートメントの結果は I/O エラー値 **endOfFile** になる。
- 現在のファイル位置は、以下の操作の影響を受ける。
 - *set record position* という書式の EGL **set** ステートメントは、*set value* に基づいてファイル位置を設定する。この値は、**set** ステートメントによって参照される索引付きレコード内のキー値です。同じ索引付きレコードに対する以降の **get next** ステートメントは、キー値が *set value* と等しいかより大きいファイル・レコードを読み取ります。そのようなレコードが存在しない場合は、**get next** の結果は **endOfFile** になります。
 - **get next** ステートメント以外の正常な I/O ステートメントは、新しいファイル位置を設定し、同じ EGL レコードに対して発行される後続の **get next** 文は、次の ファイル・レコードを読み取る。例えば、**get previous** ステートメントがファイル・レコードを読み取った後、**get next** ステートメントは、キーが次に高いレコードを読み取るか **endOfFile** を戻します。
 - **get previous** ステートメントが **endOfFile** を戻した場合、後続の **get next** ステートメントは、ファイルの最初のレコードを検索する。
 - 異常終了した **get**、**get next**、または **get previous** ステートメントの後、ファイル位置は未定義となり、*set record position* 書式の **set** ステートメントによって再設定するか、**get next** または **get previous** ステートメント以外の入出力操作によって再設定する必要がある。

- 代替索引を使用している場合で、ファイル内に複写キーがある場合は、以下の規則が適用される。
 - キーの値がより高いレコードの検索は、**get next** ステートメントが、検索された最新のレコードと同じキーを持つすべてのレコードを読み取った後でのみ行われる。複写キー付きレコードが検索される順序は、VSAM がレコードを戻す順序になります。
 - **get next** が **get next** 以外の正常な入出力操作に続く場合、その **get next** は複写キー付きレコードをスキップオーバーし、キー値が次に高いレコードを検索する。
 - プログラムが同じキーを含むレコード・グループ内の最終レコードを検索した場合、EGL エラー値 **duplicate** は設定されない。

キーが以下のようになっているファイルを考えてみましょう。

1, 2, 2, 2, 3, 4

以下の各表は、同じ索引付きレコードに対して一連の EGL ステートメントを実行する場合の効果を示しています。

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	EGL エラー値
get	2	2 (3 つある内の最初のもの)	duplicate
get next	any	2 (2 番目のもの)	duplicate
get next	any	2 (3 番目のもの)	--
get next	any	3	--

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	EGL エラー値
<i>set record position</i> 書式の set	2	検索されない	duplicate
get next	any	2 (3 つある内の最初のもの)	--
get next	any	2 (2 番目のもの)	duplicate
get next	any	2 (3 番目のもの)	--
get next	any	3	--

メッセージ・キュー

get next が MQ レコードに対して操作を行う場合は、キュー内の最初のレコードが MQ レコードに読み込まれます。この追加処理は、**get next** で、以下に示す 1 つ以上の MQSeries 呼び出しが呼び出されることによって行われます。

- MQCONN は、生成されたコードをデフォルトのキュー・マネージャーに接続する。これは、アクティブな接続が存在しない場合に呼び出されます。
- MQOPEN は、キューとの接続を確立する。これは、接続がアクティブであるがキューがまだ開いていない場合に呼び出されます。
- MQGET は、レコードをキューから除去する。これは、それ以前の MQSeries 呼び出しでエラーが発生していない限り、常に呼び出されます。

相対レコード

get next ステートメントが相対レコードに対して操作を行う場合、効果は現在のファイル位置に基づいたものとなります。この位置は、**get** ステートメントまたは別の **get next** ステートメントなどの正常な入出力操作によって設定されます。規則は以下のとおりです。

- ファイルが開かれていない場合は、**get next** ステートメントはファイル内の最初のレコードを読み取る。
- 以降の各 **get next** は、現在のファイル位置との関係でキー値が次に高いレコードを読み取る。
- 次のレコードが削除されている場合、**get next** は **noRecordFound** を戻さない。代わりに、**get next** は削除されたレコードをスキップしてファイル内の次のレコードを検索します。
- **get next** ステートメントがファイル内のキー値が最も高いレコードを読み取った後、次の **get next** ステートメントの結果は EGL エラー値 **endOfFile** になる。
- 現在のファイル位置は、以下の操作の影響を受ける。
 - **get next** 以外の正常な I/O ステートメントは、新しいファイル位置を設定し、同じ EGL レコードに対する後続の **get next** は次の ファイル・レコードを読み取る。
 - 異常終了した **get**、**get next**、または **get previous** ステートメントの後、ファイル位置は未定義となり、*set record position* 書式の **set** ステートメントによって再設定するか、**get next** ステートメント以外の入出力操作によって再設定する必要がある。
- **get next** ステートメントがファイル内の最終レコードを読み取った後、次の **get next** ステートメントの結果は EGL エラー値 **endOfFile** および **noRecordFound** になる。

シリアル・レコード

get next ステートメントがシリアル・レコードに対して操作を行う場合、効果は現在のファイル位置に基づいたものとなります。この位置は、別の **get next** によって設定されます。規則は以下のとおりです。

- ファイルが開かれていない場合は、**get next** ステートメントはファイル内の最初のレコードを読み取る。
- 以降の各 **get next** は次のレコードを読み取る。
- **get next** ステートメントがファイル内の最終レコードを読み取った後、後続の **get next** ステートメントの結果は EGL エラー値 **endOfFile** になる。
- 生成されたコードによってシリアル・レコードが追加され、同じファイルに **get next** と 同等の文が発行されると、EGL は **get next** ステートメントが実行される前に、ファイルを閉じ、再オープンする。したがって、**add** ステートメントの後に実行される **get next** ステートメントでは、ファイル内の最初のレコードが読み取られます。この振る舞いは、**get next** ステートメントと **add** ステートメントが別々のプログラムにあり、一方のプログラムで他方のプログラムが呼び出される場合にも起こります。

複数のプログラムで同時に同じファイルが開かないようにしておくことをお勧めします。

SQL 処理

get next ステートメントが SQL レコードに対して操作を行う場合は、コードでは **open** ステートメントによって選択された行の中から次の行が読み取られます。 **get next** ステートメントを発行し、forUpdate オプション付きの **open** ステートメントにより選択された行を検索すると、以下のことを実行できます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get next** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外では変更できません。

最後に選択された行を超えた行へのアクセスを試みる **get next** ステートメントを発行すると、次のようになります。

- データは結果セットからコピーされません。
- EGL は SQL レコード (もしあれば) を **noRecordFound** に設定します。
- 関連する **open** ステートメントが scroll オプションを含んでいた場合、カーソルは開かれたまま残り、カーソル位置は変更されません。
- scroll オプションを設定しなかった場合、カーソルは閉じられます。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

143 ページの『レコード・タイプとプロパティ』

799 ページの『resultSetID』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

605 ページの『add』

613 ページの『close』

617 ページの『delete』

100 ページの『例外処理』

619 ページの『execute』

631 ページの『get』

650 ページの『get previous』

579 ページの『入出力エラー値』

93 ページの『EGL ステートメント』

664 ページの『open』

679 ページの『prepare』

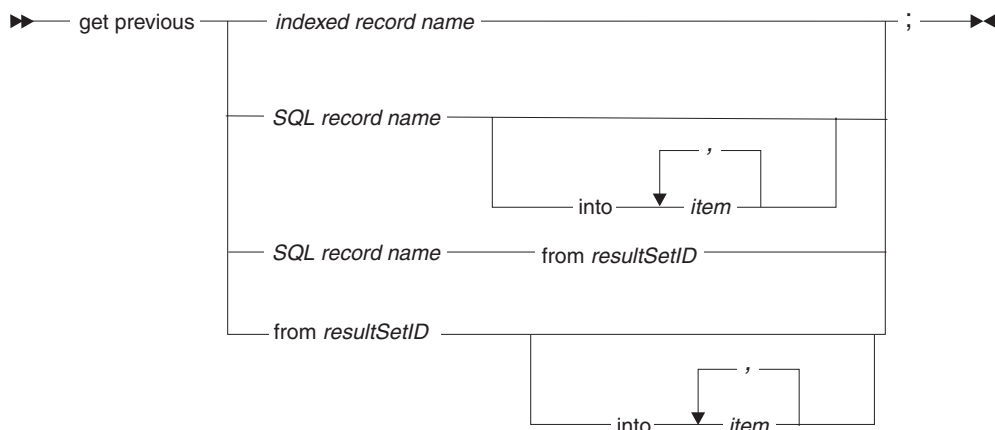
681 ページの『replace』

685 ページの『set』

get previous

EGL **get previous** ステートメントは、リレーショナル・データベース結果セットから直前の行を読み取るか、指定の EGL 索引付きレコードに関連付けられたファイル内の直前のレコードを読み取ります。

この文をリレーショナル・データベース結果セットに使用できるのは、関連する **open** ステートメントで **scroll** オプションを指定した場合だけです。



record name

入出力オブジェクトの名前。索引付きレコード、SQL レコードなどを指定します。

from *resultSetID*

SQL 処理のみに使用する、同じプログラム内で前に実行された **open** ステートメントに **get previous** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

索引付きレコードの例を次に示します。

```
record1.hexKey = "FF";
set record1 position;

try
  get previous record1;
onException
  myErrorHandler(8);
return;
end

while (record1 not endOfFile)
  processRecord(record1); // データの処理

  try
    get previous record1;
  onException
```

```

        myErrorHandler(8);
        return;
    end
end

```

get previous ステートメントの詳細は、索引付きレコードを使用しているか、あるいは、653 ページの『SQL 処理』に関係しているかによって異なります。

索引付きレコード

get previous ステートメントが索引付きレコードに対して操作を行う場合、効果は現在のファイル位置に基づいたものとなります。この位置は、以下のどちらかの操作によって設定されます。

- **get** ステートメントまたは別の **get previous** ステートメントなどの正常な入出力 (I/O) 操作
- *set record position* という書式の **set** ステートメント

索引付きレコードに関する規則は、以下のとおりです。

- ファイルが開かれていない場合は、**get previous** ステートメントはファイル内のキー値が最も高いレコードを読み取る。
- 以降の各 **get previous** は、現在のファイル位置との関係でキー値が次に低いレコードを読み取る。複写キーの場合の例外については、後で説明します。
- **get previous** ステートメントがファイル内のキー値が最も低いレコードを読み取った後、次の **get previous** ステートメントの結果は EGL エラー値 **endOfFile** になる。
- 現在のファイル位置は、以下の操作の影響を受ける。
 - *set record position* という書式の EGL **set** ステートメントは、*set value* に基づいてファイル位置を設定する。この値は、**set** ステートメントによって参照される索引付きレコード内のキー値です。同じ索引付きレコードに対する以降の **get previous** ステートメントは、キー値が *set value* と等しいかより小さいファイル・レコードを読み取ります。そのようなレコードが存在しない場合は、**get previous** ステートメントの結果は **endOfFile** になります。

set value が 16 進数 FF で埋められている場合、*set record position* という形式の **set** ステートメントの結果は以下のようになります。

- **set** ステートメントは、ファイル内の最後のレコードの後ろにファイル位置を設定する
- **get previous** ステートメントが、次の入出力操作の場合、生成されたコードではファイル内の最後のレコードが検索される
- **get previous** ステートメント以外の正常な I/O ステートメントは、新しいファイル位置を設定し、同じ EGL レコードに対する後続の **get previous** ステートメントは直前の ファイル・レコードを読み取る。例えば、**get next** ステートメントがファイル・レコードを読み取った後、**get previous** ステートメントは、キーが次に低いレコードを読み取るか **endOfFile** を戻します。
- **get next** ステートメントが **endOfFile** を戻した場合、後続の **get previous** ステートメントは、ファイルの最後のレコードを検索する。
- 異常終了した **get**、**get next**、または **get previous** ステートメントの後、ファイル位置は未定義となり、*set record position* 書式の **set** ステートメントによ

って再設定するか、**get next** または **get previous** ステートメント以外の入出力操作によって再設定する必要がある。

- 代替索引を使用している場合で、ファイル内に複写キーがある場合は、以下の規則が適用される。
 - キーの値がより低いレコードの検索は、**get previous** ステートメントが、検索された最新のレコードと同じキーを持つすべてのレコードを読み取った後でのみ行われる。複写キー付きレコードが検索される順序は、VSAM がレコードを戻す順序になります。
 - **get previous** ステートメントが **get previous** 以外の正常な入出力操作に続く場合、その **get previous** ステートメントは複写キー付きレコードをスキップオーバーし、キー値が次に低いレコードを検索する。
 - プログラムが同じキーを含むレコード・グループ内の最終レコードを検索した場合、EGL エラー値 **duplicate** は設定されない。

代替索引内のキーが以下のようにになっているファイルについて考えます。

1, 2, 2, 2, 3, 4

以下の各表は、同じ索引付きレコードに対して一連の EGL ステートメントを実行する場合の効果を示しています。

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	COBOL の場合の EGL エラー値
<i>set record position</i> 書式の set	1	--	--
get previous	any	1	--

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	EGL エラー値
get	3	3	--
get previous	any	2 (3 つある内の最初のもの)	duplicate
get previous	any	2 (2 番目のもの)	duplicate
get previous	any	2 (3 番目のもの)	--
get previous	any	1	--

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	EGL エラー値
<i>set record position</i> 書式の set	2	--	duplicate
get next	any	2 (最初のもの)	--
get next	any	2 (2 番目のもの)	duplicate
get previous	any	1	--
get previous	any	--	endOfFile

EGL ステートメント (実行順)	索引付きレコード内のキー	文によって検索されるファイル・レコード内のキー	EGL エラー値
<i>set record position</i> 書式の set	1	--	--
get previous	any	1	--

SQL 処理

get previous ステートメントが SQL レコードに対して操作を行う場合は、コードでは **open** ステートメントによって選択された行の中から前の行が読み取られます。ただし、scroll オプションを指定した場合に限られます。**get previous** ステートメントを発行し、forUpdate オプションも指定されている **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get previous** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外に変更できません。

最初に選択した行より前にある行へのアクセスを試みる **get previous** ステートメントを発行すると、EGL ランタイムは次のように動作します。

- 結果セットからデータをコピーしません。
- カーソルを開いたままに残し、カーソル位置は変更されません。
- SQL レコード (もしあれば) を **noRecordFound** に設定します。

一般に、エラーが発生して処理が続行された場合、カーソルは開いたままに残り、カーソル位置は変更されません。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

143 ページの『レコード・タイプとプロパティ』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

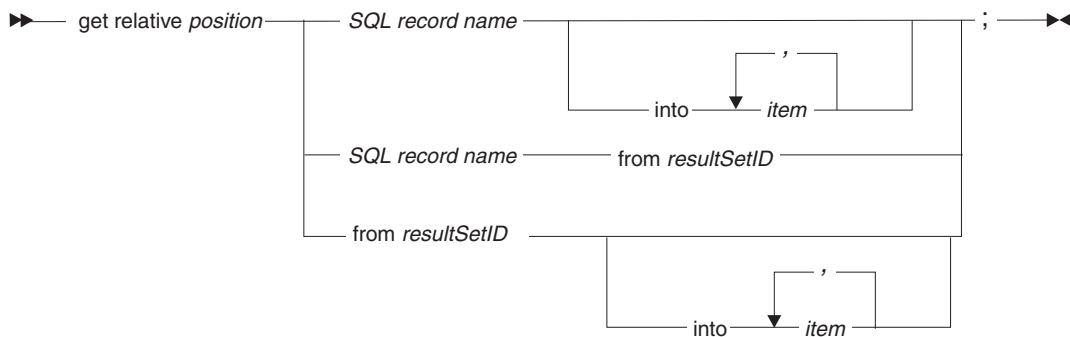
605 ページの『add』
613 ページの『close』
617 ページの『delete』
100 ページの『例外処理』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
579 ページの『入出力エラー値』

664 ページの『open』
679 ページの『prepare』
93 ページの『EGL ステートメント』
681 ページの『replace』
685 ページの『set』

get relative

EGL の **get relative** ステートメントは、リレーショナル・データベース結果セット内の、番号で指定された行を読み取ります。この行は、結果セット内のカーソル位置との相対関係で識別されます。

この文を使用できるのは、関連する **open** ステートメントで **scroll** オプションを指定した場合だけです。



position

整数の項目またはリテラル。

position の値が正の場合、位置は結果セット内の現行番号の位置に対する増分です。例えば、カーソルが 1 行目にあるときに **get relative 2** を指定すると、3 行目を取り出され、**get relative 1** を指定することは、**get next** を指定するのと同じことです。

position の値が負の場合、位置は結果セット内の現行番号の位置に対する減分です。例えば、カーソルが 3 行目にあるときに **get relative -2** を指定すると 1 行目を取り出され、**get relative -1** を指定することは、**get previous** を指定するのと同じことです。

position にゼロの値を指定すると、すでに有効となっているカーソル位置にある行を取り出され、**get current** を指定したのと同じことになります。

record name

SQL レコードの名前。

from resultSetID

同じプログラム内で前に実行された **open** ステートメントに **get relative** ステートメントを結合する ID。詳細については、『*resultSetID*』を参照してください。

into

リレーショナル・データベース表から値を受け取る項目がリストされた文節の中で、EGL を開始します。

item

特定の列の値を受け取る項目。項目名の前にはコロン (:) を付けません。

get relative ステートメントを発行し、forUpdate オプション付きの **open** ステートメントによって選択された行を取得すると、以下のことができます。

- EGL **replace** ステートメントで行を変更する
- EGL **delete** ステートメントで行を除去する
- EGL **execute** ステートメントで行を変更または除去する

SQL FETCH ステートメントは、生成されたコードでは EGL **get relative** ステートメントを表します。生成される SQL ステートメントの形式は、INTO 文節の設定以外は変更できません。

結果セットにない行へのアクセスを試みる **get relative** ステートメントを発行すると、EGL ランタイムは次のように動作します。

- 結果セットからデータをコピーしません。
- カーソルを開いたままに残し、カーソル位置は変更されません。
- SQL レコード (もしあれば) を **noRecordFound** に設定します。

一般に、エラーが発生して処理が続行された場合、カーソルは開いたままに残り、カーソル位置は変更されません。

最終的に、SQL COMMIT または sysLib.commit が指定された場合、**open** ステートメントで hold オプションを使用したときにのみ、**open** ステートメントで宣言されていたカーソル内の位置がコードにより保存されます。

関連する概念

799 ページの『resultSetID』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

617 ページの『delete』

100 ページの『例外処理』

619 ページの『execute』

631 ページの『get』

638 ページの『get absolute』

640 ページの『get current』

641 ページの『get first』

643 ページの『get last』

644 ページの『get next』

650 ページの『get previous』

93 ページの『EGL ステートメント』

664 ページの『open』

681 ページの『replace』

goTo

EGL の **goTo** ステートメントは、指定されたラベルから処理を続行します。このラベルは、この文と同じ関数内にあり、かつブロック外にある必要があります。

▶ goto *label* : ◀

label

関数内のどこかにあり、以下をはじめとするブロックの外にある一続きの文字。

- if
- else
- when (in a **case** statement)
- while
- try

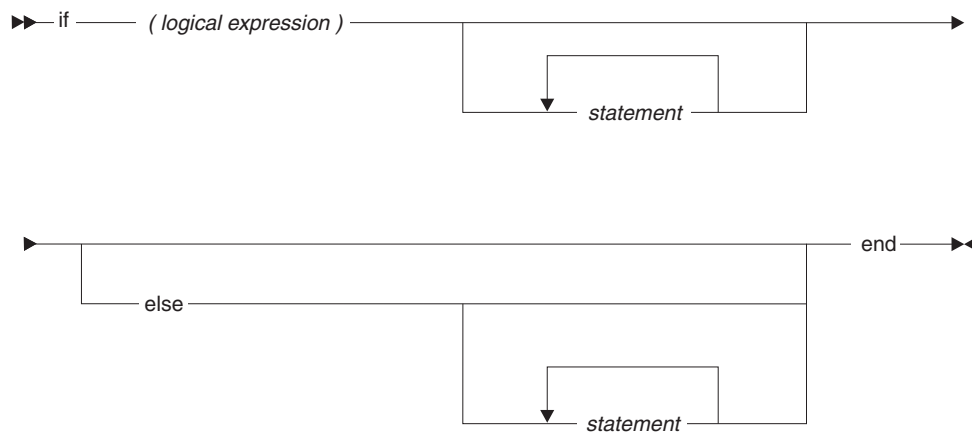
処理が継続されているロケーションに表示されるラベルは、後ろにコロンが付きます。ラベルの有効な文字についての詳細は、『命名規則』を参照してください。

関連する参照項目

725 ページの『命名規則』

if、else

EGL キーワード **if** は、論理式が真に解決される場合にのみ実行される一連の文 (ある場合) の始まりを示します。オプションのキーワード **else** は、論理式が偽に解決される場合にのみ実行される一連の代替文 (ある場合) の始まりを示します。キーワード **end** は、**if** ステートメントの終わりを示します。



logical expression

真または偽に評価される式 (一連のオペランドおよび演算子)

statement

1 つ以上の EGL ステートメント

if および、その他の **end** で終了する文は、任意のレベルまでネストできます。各 **end** キーワードでは、以下のいずれかのキーワードで始まる、終了していない最新の文を参照します。

- **if**
- **case**
- **try**
- **while**

これらの文の後にセミコロンは続きません。

以下に例を示します。

```
if (userRequest == "U")
  try
    update myRecord;
  onException
    myErrorHandler(12); // プログラムを終了
  end
  try
    myRecord.myItem=25;
    replace record1;
  onException
    myErrorHandler(16);
  end
else
  try
    add record2;
  onException
    myErrorHandler(18); // プログラムを終了
  end
  if (sysVar.systemType is WIN)
    myFunction01();
  else
    myFunction02();
  end
end
end
```

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

538 ページの『論理式』

93 ページの『EGL ステートメント』

move

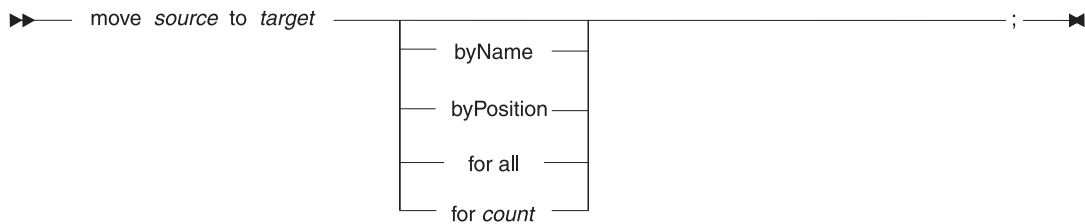
EGL **move** ステートメントにより、データは 3 つの方法のうちいずれかの方法でコピーされます。最初のオプションでは、データは 1 バイトずつコピーされます。2 番目のオプション (「名前順」と呼ばれる) では、1 つの構造体にある指定のフィールドから、別の構造体にある同じ名前のフィールドにデータがコピーされます。3 番目のオプション (「配置順」と呼ばれる) では、1 つの構造体内の各フィールドから、別の構造体内の同じ位置にあるフィールドにデータがコピーされます。

以下の一般的な規則が適用されます。

- ソース値が次のいずれかである場合、デフォルトでデータが 1 バイトずつコピーされます。
 - プリミティブ変数
 - 固定構造体内にあるフィールド
 - リテラル
 - 定数

上記以外の場合、デフォルトでデータは名前順にコピーされます。

- 移動は、フィールド間の互換性をチェックしながら行われます。切り捨て、埋め込み、および型変換の規則は、**assignment** ステートメントで詳細に指定されている規則と同じですが、**move** 文全体の振る舞いは、**assignment** ステートメントの振る舞いとは異なります。
- 動的配列を使用している場合、最後のエレメントは、配列の現行サイズによって決まります。**move** ステートメントは、配列にエレメントを追加しないため、動的配列を拡張するには、その配列固有の関数 `appendElement` または `appendAll` を使用します。詳細は『*Arrays*』を参照してください。



この文を最もよく理解するには、以下のカテゴリーを参照してください。

byName

byName を指定すると、データはソース内の各フィールドから、ターゲット内の同じ名前のフィールドに書き込まれます。操作は、ソースの構造体内での出現順に行われます。

ソースおよびターゲットには、以下を指定することができます。

source

次のいずれか

- 固定レコードの動的配列。ただし、動的配列は、ターゲットがレコードでない場合にのみ有効です。
- レコード
- 固定レコード
- 副構造を持つ構造体フィールド
- 副構造を持つ構造体フィールドの配列。ただし、この配列は、ターゲットがレコードでない場合にのみ有効です。
- データ・テーブル
- 書式

名前にアスタリスク (*) が付いた固定構造体フィールドは、ソース・フィールドとして使用できませんが、そのフィールドの副構造内にある指定のすべてのフィールドは使用できます。

target

次のいずれか

- 固定レコードの動的配列。ただし、この配列は、ソースがレコードでない場合にのみ有効です。
- レコード
- 固定レコード
- 副構造を持つ構造体フィールド
- 副構造を持つ構造体フィールドの配列。ただし、この配列は、ソースがレコードでない場合にのみ有効です。
- データ・テーブル
- 書式
-

例文は次のとおりです。

```
move myRecord01 to myRecord02 byName;
```

以下の場合、操作は無効になります。

- 同一ソース内に同じ名前のフィールドが複数ある。
- 同一宛先内に同じ名前のフィールドが複数ある。
- ソース・フィールドが、多次元の構造体フィールド配列であるか、コンテナが配列である 1 次元の構造体フィールド配列である。
- ターゲット・フィールドが、多次元の構造体フィールド配列であるか、コンテナが配列である 1 次元の構造体フィールド配列である。

この操作は、以下のような場合に有効です。

- 単純なケースでは、ソースが固定構造体であるが、それ自体が配列エレメントではなく、ターゲットについても同様です。以下の規則が適用されます。
 - 関係する配列がない場合は、ソース構造体内の各従属フィールドの値は、ターゲット構造体内の同じ名前の対応するフィールドにコピーされます。
 - 構造体フィールドの配列が別の構造体フィールドにコピーされる場合、以下のような場合では操作は「*move for all*」として扱われます。
 - ソース・フィールドのエレメントが、ターゲット・フィールドの後続エレメントにコピーされる
 - ソース配列に含まれているエレメントがターゲット配列よりも少ない場合は、そのソース配列の最後のエレメントがコピーされると処理が停止する
- 別のケースでは、ソースまたはターゲットがレコードです。ソースのフィールドが、ターゲット内の同じ名前のフィールドに割り当てられます。
- もう 1 つのケースは、それほど単純ではありませんが、次の例で最もよく説明されます。ソースが 10 個の固定レコードからなる配列であり、それぞれに以下の構造体フィールドが含まれます。

```
10 empnum CHAR(3);
10 empname CHAR(20);
```

ターゲットは、以下の構造体フィールドを含む固定構造体です。

```
10 empnum CHAR(3)[10];
10 empname CHAR(20)[10];
```

この操作により、最初の固定レコード内にあるフィールド `empnum` の値が、構造体フィールド配列 `empnum` の最初のエレメントにコピーされます。また最初の固定レコード内にあるフィールド `empname` の値が、構造体フィールド配列 `empname` の最初のエレメントにコピーされ、ソース配列内の固定レコードごとに同様の操作が行われます。

ソースが、以下のような副構造を持つ単一の固定レコードの場合は、同等の操作が発生します。

```
10 mySubStructure[10]
  15 empnum CHAR(3);
  15 empname CHAR(20);
```

- 最後に、ソースが以下の構造体フィールドを含む固定レコードであるケースを考えてみましょう。

```
10 empnum CHAR(3);
10 empname CHAR(20)[10];
```

ターゲットは、以下の副構造を持つ書式、固定レコード、または構造体フィールドです。

```
10 empnum char(3)[10];
10 empname char(20);
```

フィールド `empnum` の値は、ソースからターゲット内の `empnum` の最初のエレメントにコピーされ、`empname` の最初のエレメントの値は、ソースからターゲット内のフィールド `empname` にコピーされます。

byPosition

byPosition は、1 つの構造体の各フィールドから、別の構造体の同じ位置のフィールドにデータをコピーすることを目的としています。

ソースおよびターゲットには、以下を指定することができます。

source

次のいずれか

- 固定レコードの動的配列。ただし、動的配列は、ターゲットがレコードでない場合にのみ有効です。
- レコード
- 固定レコード
- 副構造を持つ構造体フィールド
- 副構造を持つ構造体フィールドの配列。ただし、この配列は、ターゲットがレコードでない場合にのみ有効です。
- データ・テーブル
- 書式

target

次のいずれか

- 固定レコードの動的配列。ただし、この配列は、ソースがレコードでない場合にのみ有効です。
- レコード
- 固定レコード
- 副構造を持つ構造体フィールド
- 副構造を持つ構造体フィールドの配列。ただし、この配列は、ソースがレコードでない場合にのみ有効です。
- データ・テーブル
- 書式

レコードと固定構造体との間でデータを移動する場合は、固定構造体の最上位のフィールドのみが考慮されます。2 つの固定構造体間でデータを移動する場合は、どちらかの構造体の最下位 (リーフ) フィールドのみが考慮されます。

ソースまたはターゲット・フィールドが、多次元の構造体フィールド配列であるか、コンテナが配列である 1 次元の構造体フィールド配列である場合、この操作は無効です。

この操作は、以下のような場合に有効です。

- 単純なケースでは、ソースが固定構造体であるが、それ自体が配列エレメントではなく、ターゲットについても同様です。以下の規則が適用されます。
 - 関係する配列がない場合は、ソース構造体内の各リーフ・フィールドの値は、ターゲット構造体内の対応する位置にあるリーフ・フィールドにコピーされます。
 - 構造体フィールドの配列が別の構造体フィールドにコピーされる場合、以下のような場合では操作は「*move for all*」として扱われます。
 - ソース・フィールドのエレメントが、ターゲット・フィールドの後続エレメントにコピーされる
 - ソース配列に含まれているエレメントがターゲット配列よりも少ない場合は、そのソース配列の最後のエレメントがコピーされると処理が停止する
- 別のケースでは、ソースまたはターゲットがレコードです。ソースの最上位フィールドまたはリーフ・フィールド (ソース・タイプに応じて) は、ターゲットの最上位フィールドまたはリーフ・フィールド (ターゲット・タイプに応じて) に割り当てられます。
- もう 1 つのケースは、それほど単純ではありませんが、次の例で最もよく説明されます。ソースが 10 個の固定レコードからなる配列であり、それぞれに以下の構造体フィールドが含まれます。

```
10 empnum CHAR(3);
10 empname CHAR(20);
```

ターゲットは、以下の構造体フィールドを含む固定構造体です。

```
10 empnum CHAR(3)[10];
10 empname CHAR(20)[10];
```

この操作により、最初の固定レコード内にあるフィールド `empnum` の値が、構造体フィールド配列 `empnum` の最初のエレメントにコピーされます。また

最初の固定レコード内にあるフィールド `empname` の値が、構造体フィールド配列 `empname` の最初のエレメントにコピーされ、ソース配列内の固定レコードごとに同様の操作が行われます。

ソースが、以下のような副構造を持つ単一の固定レコードの場合は、同等の操作が発生します。

```
10 mySubStructure[10]
   15 empnum CHAR(3);
   15 empname CHAR(20);
```

- 最後に、ソースが以下の構造体フィールドを含む固定レコードであるケースを考えてみましょう。

```
10 empnum CHAR(3);
10 empname CHAR(20)[10];
```

ターゲットは、以下の副構造を持つ書式、固定レコード、または構造体フィールドです。

```
10 empnum char(3)[10];
10 empname char(20);
```

フィールド `empnum` の値は、ソースからターゲット内の `empnum` の最初のエレメントにコピーされ、`empname` の最初のエレメントの値は、ソースからターゲット内のフィールド `empname` にコピーされます。

for all

for all は、ターゲット配列内のすべてのエレメントに値を割り当てることを目的としています。

ソースおよびターゲットには、以下を指定することができます。

source

次のいずれか

- レコード、固定レコード、またはプリミティブ変数の動的配列
- レコード
- 固定レコード
- 副構造を持つ構造体フィールド、または持たない構造体フィールド
- 副構造を持つ構造体フィールド配列、または持たない構造体フィールド配列
- プリミティブ変数
- リテラルまたは定数

target

次のいずれか

- レコード、固定レコード、またはプリミティブ変数の動的配列
- 副構造を持つ構造体フィールド配列、または持たない構造体フィールド配列
- 動的または構造体フィールド配列のエレメント

このケースの **move** ステートメントは、複数の **assignment** ステートメント (ターゲット配列エレメントごとに 1 つずつ) と同等であり、割り当ての試行が無効な場合、エラーが発生します。有効性の詳細については、『代入』を参照してください。

ソースまたはターゲット・エレメントに固定構造体がある場合、**move** ステートメントは、構造体のトップレベルが別のプリミティブ型を指定しないかぎり、この構造体を **CHAR** 型のフィールドとして扱います。**for all** が使用中のときは、**move** ステートメントは、副構造を考慮しません。

ソースが配列のエレメントである場合、そのソースは、指定されたエレメントが最初のエレメントである配列として扱われ、前のエレメントは無視されます。

ソースが配列または配列のエレメントである場合は、そのソース配列の後続の各エレメントは、ターゲット配列の次のエレメントに順次コピーされます。ターゲット配列またはソース配列のいずれかが他方より長くても問題ありません。この場合は、他の配列と一致するエレメントを持つ最後のエレメントからデータがコピーされると、操作が終了します。

ソースが配列でも、配列のエレメントでもない場合、この操作はソースの値を使用して、ターゲット配列のすべてのエレメントを初期化します。

for count

for count は、値をターゲット配列内のエレメントの順次サブセットに割り当てることを目的としています。以下に例を示します。

- 次の文では、「abc」をターゲット内のエレメント 7、8、および 9 に移動します。

```
move "abc" to target[7] for 3
```

- 次の文では、ソースのエレメント 2、3、および 4 が、ターゲットのエレメント 7、8、および 9 に移動します。

```
move source[2] to target[7] for 3
```

この操作は、以下のような場合に有効です。

- ソースが配列でも、配列のエレメントでもない場合、この操作はソースの値を使用して、ターゲット配列のエレメントを初期化します。
- ソースが配列である場合、その配列の最初のエレメントが、コピーされる一連のエレメント内で最初のエレメントになります。ソースが配列のエレメントである場合、そのエレメントが、コピーされる一連のエレメント内で最初のエレメントになります。
- ターゲットが配列である場合、その配列の最初のエレメントが、データを受け取る一連のエレメント内で最初のエレメントになります。ターゲットが配列のエレメントである場合、そのエレメントが、データを受け取る一連のエレメント内で最初のエレメントになります。

「*count*」値は、データを受け取るターゲット・エレメントの数を指定します。この値は、以下のいずれかにすることができます。

- 整数リテラル
- 整数に解決する変数
- 数式。ただし、関数呼び出しではありません。

move ステートメントは、複数の **assignment** ステートメント (ターゲット配列エレメントごとに 1 つずつ) と同等であり、割り当ての試行が無効な場合、エラーが発生します。有効性の詳細については、『代入』を参照してください。

ソースまたはターゲット・エレメントに内部構造体がある場合、**move** ステートメントは、構造体のトップレベルが別のプリミティブ型を指定しないかぎり、この構造体を **CHAR** 型のフィールドとして扱います。**for count** が使用中のときは、**move** ステートメントは、副構造を考慮しません。

ソースおよびターゲットの両方が配列であるときは、ターゲットの配列またはソースの配列のいずれかが他方よりも長くてもかまいません。この場合、次の 2 つのイベントの内、最初のイベントが発生した後に操作が終了します。

- 操作要求の対象となる最後のエレメントの間でデータがコピーされたとき
- 他の配列と一致するエレメントを持つ最後のエレメントからデータがコピーされたとき

ソースが配列でない場合は、次の 2 つのイベントのうち、最初のイベントが発生した後にこの操作は終了します。

- 操作要求の対象となる最後のエレメントにデータがコピーされたとき
- 配列内で最後のエレメントにデータがコピーされたとき

ソースがレコード配列 (またはレコード配列のエレメント) である場合、ターゲットはレコード配列でなければなりません。ソースがプリミティブ変数配列 (またはプリミティブ変数配列のエレメント) である場合、ターゲットはプリミティブ変数配列または構造体フィールド配列でなければなりません。ソースが構造体フィールド配列 (または構造体フィールド配列のエレメント) である場合、ターゲットはプリミティブ変数配列または構造体フィールド配列でなければなりません。

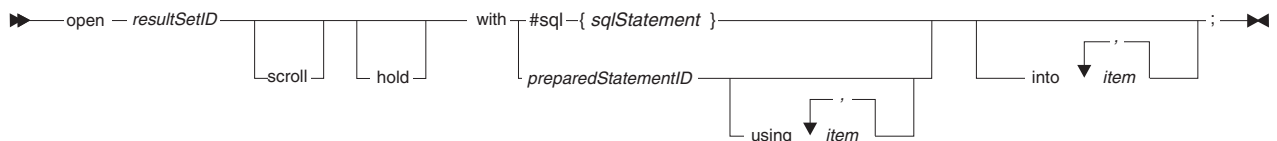
関連する参照項目

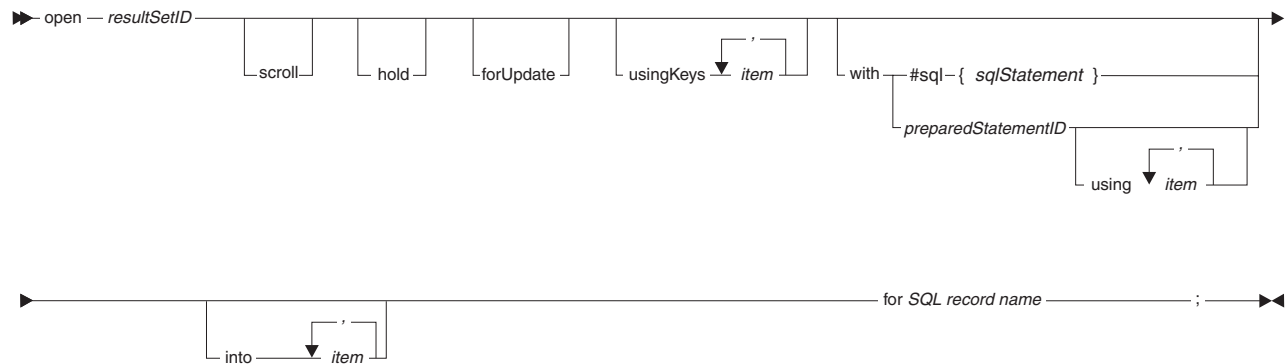
78 ページの『配列』

407 ページの『代入』

open

EGL **open** ステートメントは、後で **get next** ステートメントを使用して検索するために、リレーショナル・データベースから一連の行を選択します。**open** ステートメントは、カーソルまたは呼び出されたプロシージャ上で作動できます。





resultSetID

open ステートメントを、後の **get next**、**replace**、**delete**、および **close** ステートメントに結びつける ID。詳細については、『*resultSetID*』を参照してください。

scroll

結果セット内をさまざまな方法で移動できるオプション。**get next** ステートメントは常時使用できますが、**scroll** を使用すると、以下の文も使用できます。

- **get absolute**
- **get current**
- **get first**
- **get last**
- **get previous**
- **get relative**

hold

コミットが発生したときに、結果セット内の位置を維持します。

注: **hold** オプションを使用できるのは、JDBC ドライバーが JDBC 3.0 以上をサポートしている場合だけです。

hold オプションは、次の場合に適切です。

- ストアード・プロシージャではなく、カーソルをオープンするために EGL **open** ステートメントを使用している場合。
- 結果セット中の位置を失うことなく、定期的に変更内容をコミットする場合。
- 使用しているデータベース管理システムが、SQL カーソル宣言で **WITH HOLD** オプションの使用をサポートしている場合。

例えば、コードは以下のように機能します。

1. EGL **open** ステートメントを実行し、カーソルを宣言してオープンする。
2. EGL **get next** ステートメントを実行して、1 つの行を取り出す。
3. ループの中で以下の操作を行う。
 - a. 何らかの方法でデータを処理する。
 - b. EGL **replace** ステートメントを実行して、行を更新する。
 - c. system 関数 `sysLib.commit` を実行し、変更内容をコミットする。
 - d. EGL **get next** ステートメントを実行して、別の行を取り出す。

hold を指定しない場合は、カーソルが開いていないためにステップ 3 の **d** の最初の実行は失敗します。

hold を指定するカーソルはコミット時には閉じられませんが、ロールバックまたはデータベース接続ではすべてのカーソルが閉じられます。

コミットを通じてカーソル位置を保存する必要がある場合は、**hold** を指定しないでください。

forUpdate

データベースから検索されたデータを置換または削除するために以降の EGL ステートメントを使用することができるオプション

結果セットを検索するためにストアード・プロシージャを呼び出している場合は、**forUpdate** を指定できません。

usingKeys ... item

暗黙の SQL ステートメントの **WHERE** 文節のキー値コンポーネントを作成するために使用するキー項目のリストを識別します。暗黙の SQL ステートメントは、明示的な SQL ステートメントを指定しない場合に実行時に使用されます。

usingKeys 文節を指定しない場合、暗黙のステートメントのキー値コンポーネントは **open** ステートメントで参照される SQL レコード・パーツが基となります。

明示的な SQL ステートメントを指定する場合、**usingKeys** 情報は無視されます。

with #sql{ sqlStatement }

明示的な SQL **SELECT** ステートメント。SQL レコードを指定した場合はオプションです。**#sql** と左中括弧の間にスペースを入れないでください。

into ... item

カーソルまたはストアード・プロシージャから値を受け取る EGL ホスト変数を識別する **INTO** 文節です。 **#sql{ }** ブロックの外側にある、このような文節では、ホスト変数名の前にセミコロンを含めないでください。

with preparedStatementID

実行時に SQL **SELECT** または **CALL** ステートメントを準備する EGL **prepare** ステートメントの ID。**open** ステートメントにより、SQL **SELECT** または **CALL** ステートメントが動的に実行されます。詳細については、『*prepare*』を参照してください。

using ... item

実行時に準備済み SQL **SELECT** または **CALL** ステートメントで使用可能になる EGL ホスト変数を識別する **USING** 文節です。 **#sql{ }** ブロックの外側にある、このような文節では、ホスト変数名の前にセミコロンを含めないでください。

SQL record name

SQLRecord タイプのレコードの名前。レコード名または *sqlStatement* に対する値のどちらかが必要です。*sqlStatement* が省略されると、SQL **SELECT** ステートメントは SQL レコードから引き出されます。

以下に例を示します (SQL レコードにより *emp* が呼び出されたと想定)。

```
open empSetId forUpdate for emp;

open x1 with
  #sql{
    select empnum, empname, empphone
    from employee
    where empnum >= :empnum
    for update of empname, empphone
  }

open x2 with
  #sql{
    select empname, empphone
    from employee
    where empnum = :empnum
  }
for emp;

open x3 with
  #sql{
    call aResultSetStoredProc(:argumentItem)
  }
```

デフォルトの処理

SQL レコードを指定した場合、デフォルトでの **open** ステートメントの影響は、以下のようになります。

- **open** ステートメントは、一連の行を使用可能にする。選択された行の中の各列は構造体項目に関連付けられ、読み取り専用の構造体項目に関連付けられた列を除いて、すべての列は **EGL replace** ステートメントによる以降の更新に使用できます。
- SQL レコードのキー項目を 1 つのみ宣言している場合は、SQL テーブル・キー列の値が SQL レコードのキー項目の値より大きいか等しい限り、**open** ステートメントはレコード固有の **default select condition** を満たす行をすべて選択する。
- SQL レコードに複数のキーが宣言されている場合は、レコード固有の **default select condition** が唯一の検索基準となり、**open** ステートメントはその基準を満たす行をすべて検索する。
- レコード・キーもデフォルトの選択条件も指定しない場合、**open** ステートメントはテーブル内のすべての行を選択する。
- 選択された行はソートされない。

EGL open ステートメントは、生成されたコードでは **SQL SELECT** または **SQL SELECT FOR UPDATE** ステートメントが含まれるカーソル宣言によって表されます。デフォルトでは、以下のことが当てはまります。

- **FOR UPDATE** 文節がある場合、これには読み取り専用の構造体項目が含まれない。
- 特定のレコード用の **SQL SELECT** ステートメントは、以下のステートメントのようになります。

```
SELECT column01,
       column02, ...
       columnNN
INTO  :recordItem01,
      :recordItem02, ...
      :recordItemNN
```

```

FROM   tableName
WHERE  keyColumn01 = :keyItem01
FOR UPDATE OF
      column01,
      column02, ...
      columnNN

```

デフォルトをオーバーライドするには、EGL **open** ステートメント内の SQL ステートメントを指定します。

エラー条件

以下のものを含むさまざまな条件は無効です。

- SELECT に必要な文節が欠けている SQL ステートメントを含める。必要な文節は SELECT、FROM、および **forUpdate** を指定した場合の FOR UPDATE OF です。
- 実行時に存在しないか、関連する構造体項目との互換性がない列に関連付けられている SQL レコード
- **forUpdate** オプションが指定され、次の SQL レコードのいずれかに対して、コードによる **open** ステートメントの実行が 試行される。
 - 読み取り専用の構造体項目しかない SQL レコード
 - 複数の SQL テーブルに関連付けられている SQL レコード

以下の場合でも、問題が生じます。

1. 更新のために EGL **open** ステートメントをカスタマイズしたが、特定の SQL テーブル列が **update** で使用可能であることを示していない場合に、
2. **open** ステートメントに関連付けられた **replace** ステートメントが列を変更しようとした。

この問題は、以下のいずれかの方法で解決できます。

- EGL **open** ステートメントをカスタマイズする際に、使用する列名を SQL SELECT ステートメントの FOR UPDATE OF 文節に組み込む
- EGL **replace** ステートメントをカスタマイズする際に、SQL UPDATE ステートメントの SET 文節内にある列への参照を除去する
- **open** と **replace** ステートメントの両方について、デフォルトを受け入れる

関連する概念

- 143 ページの『レコード・タイプとプロパティ』
- 247 ページの『SQL サポート』
- 799 ページの『resultSetID』
- 24 ページの『パーツの参照』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

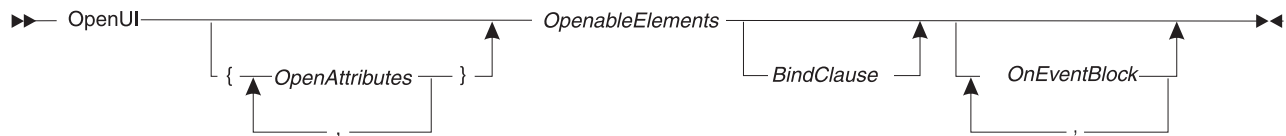
- 605 ページの『add』
- 613 ページの『close』
- 617 ページの『delete』
- 93 ページの『EGL ステートメント』

100 ページの『例外処理』
 619 ページの『execute』
 631 ページの『get』
 644 ページの『get next』
 650 ページの『get previous』
 579 ページの『入出力エラー値』
 679 ページの『prepare』
 681 ページの『replace』
 71 ページの『SQL 項目のプロパティー』
 1004 ページの『terminalID』

openUI

ユーザーは、**OpenUI** ステートメントによって、**consoleUI** に基づくインターフェースのプログラムと対話することができます。この文によってユーザーおよびプログラム・イベントが定義されるとともに、各応答方法が指定されます。

OpenUI ステートメントの構文は、以下のとおりです。



OpenAttributes

OpenAttributes は、次の例のようにそれぞれがコンマで区切られたプロパティーと値のペアのセットを定義します。

```
allowAppend = yes, allowDelete = no
```

各プロパティーはユーザーの相互作用に影響を及ぼし、*OpenableElements* で参照される **consoleUI** 変数のプロパティーを上書きするものもあります。プロパティーには、次のものがあります。

allowAppend

ユーザーが表示中の **arrayDictionary** の最後部にデータを挿入できるようにするかどうかを指定します。「yes」の場合、以下のような影響があります。

- ユーザーは、データを含む最後の行の後ろの **arrayDictionary** 行にカーソルを移動して、データの行を挿入します。
- ユーザーが手順を実行すると、その **arrayDictionary** にバインドされている動的配列に要素が追加されます。

変数型: *ArrayDictionary*

プロパティー・タイプ: *Boolean*

例: *allowAppend = no*

デフォルト: 「yes」。ただし、*openUI* プロパティー **displayOnly** が「yes」に設定されている場合、デフォルトは「no」となります。

allowDelete

ユーザーが表示中の `arrayDictionary` から行を削除できるようにするかどうかを指定します。「yes」の場合、以下のような効果があります。

- ユーザーは、対象の行にカーソルを移動し、**ConsoleLib.key_delete** で参照されるキーを押して、行を削除します。
- ユーザーが手順を実行すると、その `arrayDictionary` にバインドされている動的配列の関連要素が削除されます。

変数型: *ArrayDictionary*

プロパティ・タイプ: *Boolean*

例: `allowDelete = no`

デフォルト: 「yes」。ただし、*openUI* プロパティ **displayOnly** が「yes」に設定されている場合、デフォルトは「no」となります。

allowInsert

ユーザーが表示中の `arrayDictionary` に行を挿入できるようにするかどうかを指定します。「yes」の場合、以下のような影響があります。

- ユーザーは、既存の行にカーソルを移動し、**ConsoleLib.key_insert** で参照されるキーを押して、行を挿入します。
- 新規行は、カーソルが表示される行の前に配置されます。
- ユーザーが手順を実行すると、その `arrayDictionary` にバインドされている動的配列に要素が挿入されます。

変数型: *ArrayDictionary*

プロパティ・タイプ: *Boolean*

例: `allowInsert = no`

デフォルト: 「yes」。ただし、*openUI* プロパティ **displayOnly** が「yes」に設定されている場合、デフォルトは「no」となります。

bindingByName

一連の `ConsoleField` に一連の変数をバインドする方法を示します。特に、各変数名と `ConsoleField` 名を突き合わせるかどうかを示します。変数名は *BindClause* にリストされています。`ConsoleField` 名は `ConsoleField` 名フィールドの値です。

変数型: *ConsoleForm*、*ConsoleField*、または *Dictionary*。*arrayDictionary* は含まれません。

プロパティ・タイプ: *Boolean*

例: `bindByName = yes`

デフォルト: *no*

このオプションの値は、次のとおりです。

no (デフォルト)

次のように位置によって変数と `ConsoleField` を一致させます。

- リスト内の各変数の位置
- `consoleForm` 内の各 `ConsoleField` の位置

`consoleField` が `openUI` ステートメントで明確にリストされる場合、または辞書宣言内にリストされる場合のいずれでも、位置によってバインディングが実行されるため、その順序によって `consoleField` の順序が定義されます (また、その順序によってユーザー入力に対するタブ順序も定義されます。『*ConsoleUI* パーツおよび関連変数』を参照してください。)

yes

名前順に変数と `ConsoleField` を一致させます。

`consoleField` が辞書宣言にリストされているか、または存在しても、一致する変数がバインディング・リストに存在しない場合は、`consoleField` へのユーザーの入力は無視されます。同様に、一致するフィールドがないバインディング変数も無視されます。

実行時には、最低 1 つの `consoleField` および変数がバインドされている必要があります。バインドされていない場合、エラーが発生します。

color

`ConsoleField` のテキストの色を指定します。この値は、`ConsoleField` 宣言で指定された色をオーバーライドします。

変数型: *ConsoleForm*、*ConsoleField*、*ArrayDictionary*、または *Dictionary*

プロパティ・タイプ: *ColorKind*

例: `color = red`

デフォルト: *white*

このオプションの値は、次のとおりです。

defaultColor または white (デフォルト)

白

black

黒

blue

青

cyan

シアン

green

緑

magenta

マゼンタ

red

赤

yellow

黄

currentArrayCount

表示中の `arrayDictionary` がバインドされている動的配列で使用可能な要素の数を指定します。この値が指定されない場合、すべての要素が `arrayDictionary` で使用可能になります。

変数型: *ArrayDictionary*

プロパティー・タイプ: *INT*

例: *currentArrayCount = 4*

デフォルト: *none*

displayOnly

`consoleField` が表示専用で表示されるようにするかどうかを指定します。「yes」の場合、データは更新に対して保護されるため、変更できません。

変数型: *ArrayDictionary*、*Dictionary*、*ConsoleField*、*ConsoleForm*

プロパティー・タイプ: *Boolean*

例: *displayOnly = yes*

デフォルト: *no*

help

ConsoleLib.key_help で識別されたキーをユーザーが押すときに表示されるテキストを指定します。

このヘルプ・テキストは、**openUI** コマンド用です。キーに関連付けられるテキストが、状況に合わせて変化する場合があります。例えば、メニューの各オプションに、それぞれのヘルプ・メッセージが関連付けられることがあります。

変数型: *ConsoleForm*、*ConsoleField*、*ArrayDictionary*、または *Dictionary*

プリミティブ型: *String*

例: *help = "Update the value"*

デフォルト: *Empty string*

helpKey

以下の状態のときに表示されるテキストを含むリソース・バンドルを検索するためのアクセス・キーを指定します。

- カーソルが *OpenableElements* で識別された *ConsoleUI* 変数 (*ConsoleForm*) にあり、しかも
- ユーザーが **ConsoleLib.key_help** で識別されたキーを押す。

help および **helpKey** が両方とも指定されると、**help** が使用可能になります。

変数型: *ConsoleForm*、*ConsoleField*、*ArrayDictionary*、または *Dictionary*

プロパティー・タイプ: *String*

例: *helpKey = "myKey"*

デフォルト: *Empty string*

リソース・バンドルは、システム変数 **ConsoleLib.messageResource** で識別されます。『*messageResource*』を参照してください。

highlight

(必要に応じて) *ConsoleField* の表示時に使用される特殊効果を指定します。この値は、*ConsoleField* 宣言で指定された等価値をオーバーライドします。

変数型: *ConsoleForm*、*ConsoleField*、*ArrayDictionary*、または *Dictionary*

プロパティー・タイプ: *HighlightKind[]*

例: *highlight = [reverse, underline]*

デフォルト: *[noHighLight]*

このオプションの値は、次のとおりです。

noHighlight (デフォルト)

特殊効果は実行されません。この値を使用すると他のすべての値がオーバーライドされます。

blink

効果はありません。

reverse

テキストおよび背景色を反転し、(例えば) 黒色の背景色に白色文字を表示した場合に背景色を黒くしてテキストを白色にします。

アンダーライン

対象となる領域の下にアンダーラインを配置します。アンダーラインの色は、値 **reverse** が指定されている場合でも、テキストの色となります。

intensity

表示フォントを強調する度合いを指定します。

変数型: *ConsoleField*、*ConsoleForm*、*ArrayDictionary*、または *Dictionary*

プロパティ・タイプ: *IntensityKind[]*

例: *intensity = [bold]*

デフォルト: *[normalIntensity]*

このオプションの値は、次のとおりです。

normalIntensity (デフォルト)

特殊効果は実行されません。この値を使用すると他のすべての値がオーバーライドされます。

bold

テキストを太文字で表示します。

dim

入力フィールドが無効の場合に、必要に応じて、低輝度でテキストが表示されます。

invisible

書式上に *ConsoleField* があることを示さないようにします。

isConstruct

openUI ステートメントの目的が **SELECT** などの **SQL** ステートメントで使用する選択基準を作成することであるかどうかを指定します。

変数型: *ConsoleField*、*ConsoleForm*、*Dictionary*

プロパティ・タイプ: *Boolean*

例: *isConstruct = no*

デフォルト: *yes*

このオプションの値は、次のとおりです。

no (デフォルト)

各 *ConsoleField* は通常、変数にバインドされます。

yes

openUI ステートメントは、単一の文字タイプの変数にバインドされる必要があります。この変数によって **ConsoleField** の初期値は取得されませんが、SQL WHERE 文節で使えるようにフォーマット設定されたユーザーの入力を受け取ります。

maxArrayCount

表示中の **arrayDictionary** にバインドされている動的配列に格納可能な最大行数を指定します。最大行数に達すると、それ以上行を挿入することはできません。

変数型: *ArrayDictionary*

プロパティ・タイプ: *INT*

例: *maxArrayCount = 20*

デフォルト: *none*

setInitial

ConsoleField の初期値 (**consoleForm** 宣言で定義) がユーザーによって変更されるまで表示されるようにするかどうかを指定します (初期値は、**ConsoleField** の **initialValue** フィールドを設定して指定します。)。

変数型: *ConsoleField*、*ConsoleForm*、*Dictionary*、または *ArrayDictionary*

プロパティ・タイプ: *Boolean*

例: *setInitial = yes*

デフォルト: *no*

setInitial の値が **no** の場合、最初にバウンド変数の値が取り出され表示されます。

OpenableElements

openUI ステートメントが作用する **ConsoleUI** 変数は以下のとおりです。

- **ConsoleForm**
- **consoleField** または次の項目のいずれか
 - それぞれがコンマで区切られた、**consoleField** のリスト
 - **consoleForm** で宣言され、その **consoleForm** の **consoleField** のセットを参照する辞書
 - **consoleForm** で宣言され、その **consoleForm** の **consoleField** 配列のセットを参照する **arrayDictionary**
- **Menu**
- **Prompt**
- **Window**

BindClause

ConsoleUI 変数にバインドされているプリミティブ変数、レコード、または配列のリスト。バインディング変数の特性は、次のように **openUI** ステートメントの影響を受ける **consoleUI** 変数の特性によって決定します。

- **consoleField** に対しては、プリミティブ変数を指定できます。

- 表示中の `arrayDictionary` に対しては、レコードの配列、すなわち、`arrayDictionary` の行ごとに 1 つの要素を指定できます。`arrayDictionary` の各行が単一値を表している場合、プリミティブ変数の配列を指定できます。
- 辞書または `consoleField` のリストに対しては、プリミティブ変数のリストを指定できます。また、`consoleField` にバインドされている一連のフィールドを含む各要素を使用して、レコードの配列を指定することもできます。後者の方法は、行が 1 つしかない表示中の `arrayDictionary` と動的配列をバインドする方法と同じです。レコードを追加、挿入、または削除して、動的配列を変更でき、レコードは必ず 1 度に 1 つだけ表示されます。
- プロンプトに対しては、ユーザーの応答を受信するプリミティブ・フィールドを指定できます。

バインディングの詳細については、『*OnEventBlock*』(後述) および『*ConsoleUI* パーツおよび関連変数』を参照してください。

OnEventBlock

イベント・ブロック は、特定のイベントに応答するために書き込まれたコードをそれぞれが含む 0 から多数のイベント・ハンドラー によって構成されるプログラミング構造です。イベント・ハンドラーは、次のように `OnEvent` ヘッダーで開始します。

```
OnEvent(eventKind: eventQualifiers)
```

eventKind

幾つかのイベントの 1 つ。有効な値については、676 ページの『イベント・タイプ』を参照してください。

eventQualifier

イベントを詳細に定義するデータ。入力された `ConsoleField` または押されたキー・ストロークの場合があります。

指定のイベントに応答する EGL ステートメントは、後の例が示すように、`OnEvent` ヘッダーと次の `OnEvent` ヘッダー (存在する場合) 間にあります。

ユーザーは継続的にプログラムと対話し、イベント・ハンドラーに関連付けられているイベントが発生すると、プログラムはそのイベント・ハンドラーを実行します。**openUI** ステートメントの目的がプロンプトの表示の場合には、次のようにユーザー・プログラムの相互作用はループとはほとんど似ていません。

1. イベント・ハンドラー (幾つかのうちの 1 つの場合があります) がユーザー・キー・ストロークをトラップし、反応する
2. **openUI** ステートメントは終了する

ウィンドウで使用可能なイベント・ブロックはありません。

ユーザーと `ConsoleForm` 間の相互作用を導出するために以下の例を検討します。

```
openUI {bindingByName=yes}
  activeForm
  bind firstName, lastName, ID
  OnEvent(AFTER_FIELD:"ID")
    if (employeeID == 700)
      firstName = "Angela";
      lastName = "Smith";
    end
end
end
```


コードの動作は、以下のとおりです。

- 有効な *ConsoleForm* (アクティブ・ウィンドウで最後に表示された *consoleForm*) を開きます。
- プリミティブ変数のセットと各 *ConsoleField* をバインドします。
- ユーザーが *employeeID* に値を入力し、*ConsoleField* に対する手順が終了した後で、EGL によって 2 つの他の変数にストリングが配置されます。

先の例について、以下の詳細内容を検討します。

- カーソルはリストされている *consoleField* の先頭から開始します。ただし、他の *consoleField* のユーザーの入力がイベント・ハンドラーによって消去されないように、ID *consoleField* で開始する必要があります。
- *firstName* および *lastName* *consoleField* にバインドされている変数はイベント・ハンドラーによって更新されますが、カーソルがそのフィールドに移動するまで、これらの値は表示されません。値をもっと早い時期に参照することもできます。

フォーム **exit openUI** の **exit** ステートメントを発行して、**openUI** ステートメントを終了することができます。

イベント・タイプ

ConsoleUI は以下のイベントをサポートしています。

BEFORE_OPENUI

EGL ランタイムが開始すると、**OpenUI** ステートメントが実行されます。このイベントは、ウィンドウに基づく変数以外のあらゆる ConsoleUI 変数に対して有効です。

AFTER_OPENUI

EGL ランタイムは、実行中の **OpenUI** ステートメントを停止しようとします。このイベントは、ウィンドウに基づく変数以外のあらゆる ConsoleUI 変数に対して有効です。

ON_KEY:(ListOfStrings)

ユーザーは、"ESC"、"F2"、または "CONTROL_W" などのストリングの指示に従い、特定のキーを押します。以下の例のように、各ストリングを区切って、複数のキーを識別できます。

```
ON_KEY:("a", "ESC")
```

このイベントは、ウィンドウに基づく変数以外のあらゆる ConsoleUI 変数に対して有効です。

BEFORE_FIELD:(ListOfStrings)

ユーザーは、*ConsoleField* 名フィールドの値と一致するストリングの指示に従い、カーソルを指定の *ConsoleField* に移動します。以下の例のように、各ストリングを区切って、同一の *consoleForm* 内で複数の *consoleField* を識別できます。

```
BEFORE_FIELD:("field01", "field02")
```

AFTER_FIELD:(ListOfStrings)

ユーザーは、*ConsoleField* 名フィールドの値と一致するストリングの指示に従

い、カーソルを指定の ConsoleField から移動します。以下の例のように、各ストリングを区切って、同一の consoleForm 内で複数の consoleField を識別できます。

```
AFTER_FIELD:("field01", "field02")
```

BEFORE_DELETE

表示中の arrayDictionary に関連して、ユーザーは **ConsoleLib.key_deleteLine** で指定されたキーを押します。ただし、この時点ではまだ EGL ランタイムによって行は削除されていません。プログラムによって **consoleLib.cancelDelete** を呼び出し、行の削除を回避できます。

BEFORE_INSERT

表示中の arrayDictionary に関連して、ユーザーは **ConsoleLib.key_insertLine** で指定されたキーを押します。ただし、この時点ではまだ EGL ランタイムによって行は挿入されていません。プログラムによって **consoleLib.cancelInsert** を呼び出し、行の挿入を回避できます。

BEFORE_ROW

表示中の arrayDictionary に関連して、ユーザーはカーソルを行に移動します。

AFTER_DELETE

表示中の arrayDictionary に関連して、ユーザーは **ConsoleLib.key_deleteLine** で指定されたキーを押します。EGL ランタイムによって行が削除されます。

AFTER_INSERT

表示中の arrayDictionary に関連して、ユーザーは **ConsoleLib.key_insertLine** で指定されたキーを押します。EGL ランタイムによって行が挿入され、挿入された行からカーソルが抜け出します。

ユーザーは、データベースの変更をコミットする前に行を編集できます。これは、通常 AFTER_INSERT ハンドラーで実行されます。

AFTER_ROW

ユーザーは表示中の arrayDictionary の行からカーソルを移動します。

MENU_ACTION:(ListOfStrings)

ユーザーは menuItem 名フィールドの値と一致するストリングの指示に従い、menuItem を選択します。以下の例のように、各ストリングを区切って、複数の menuItems を識別できます。

```
MENU_ACTION:("item01", "item02")
```

isConstruct

プロパティーが **isConstruct = yes** の場合、**openUI** コマンドにバインドされている変数に配置されているテキストは以下の例のように特別にフォーマットされています。

1. openUI ステートメントは 3 つの ConsoleField (*employee*、*age*、および *city*) の ConsoleForm に影響を及ぼし、各フィールドは同じ名前の SQL テーブル列に関連付けられています。

consoleField プロパティー **SQLColumnName** を設定して、consoleField と SQL テーブル列を関連付けます。また、『*ConsoleField* プロパティーおよびフィールド』の記載に従って、consoleField プロパティー **dataType** を設定する必要があります。

2. ユーザーは以下の手順を実行します。

- *employee* フィールドをブランクにします。
- *age* フィールドに以下の値を入力します。
> 25
- *city* フィールドに以下の値を入力します。
= 'Sarasota'

3. openUI ステートメントが作用する表示中の変数をそのままにすると、バインドされた変数は以下の値を受信します。

AGE > 28 AND CITY = 'Sarasota'

上図のように、EGL によって演算子 AND がユーザーが指定した各文節間に配置されます。

次の表は、有効なユーザー入力およびその文節を示しています。単純な *SQL* 型という用語は構造化型または LOB に似た型のいずれでもない *SQL* 型を表しています。

シンボル	定義	Supported データ型	例	最終文節 (C という名前の文字カラムの場合)	最終文節 (C という名前の数値カラムの場合)
=	等しい	単純 SQL 型	=x, ==x	C = 'x'	C = x
>	より大	単純 SQL 型	>x	C > 'x'	C > x
<	未満	単純 SQL 型	<x	C < 'x'	C < x
>=	以上	単純 SQL 型	>=x	C >= 'x'	C >= x
<=	以下	単純 SQL 型	<=x	C <= 'x'	C <= x
<> または !=	等しくない	単純 SQL 型	<>x または !=x	C != 'x'	C != x
..	範囲	単純 SQL 型	x.y or x..y	C BETWEEN 'x' AND 'y'	C BETWEEN x AND y
*	ストリングのワイルドカード (次の表を参照)	CHAR	*x または x* または *x*	C MATCHES '*x'	適用不可
?	単一文字ワイルドカード (次の表を参照)	CHAR	?x、x?、?x?、x??	C MATCHES '?x'	適用不可
	論理 OR	単純 SQL 型	x y	C IN ('x', 'y')	C IN (x, y)

等号 (=) は IS NULL を意味する場合があります。また、非等号 (!= または <>) は IS NOT NULL を意味する場合があります。

MATCHES 文節は、次の表のワイルドカード文字のいずれかをユーザーが指定することによって生成されます。

シンボル	結果
*	ゼロまたはそれ以上の文字と一致します。
?	すべての単一文字と一致します。
[]	すべての囲み文字と一致します。
- (ハイフン)	ハイフンは、大括弧内の文字間で使用される場合、2 つの文字を含むその範囲の任意の文字に一致します。例えば、[a-z] は、任意の小文字または小文字の範囲の特殊文字に一致します。
^	イニシャル・キャレットは、大括弧内で使用される場合、大括弧内に含まれていない任意の文字に一致します。例えば、[^abc] は a、b、または c 以外の任意の文字に一致します。
¥	デフォルトのエスケープ文字です。次の文字はリテラルです。ワイルドカードの効果を受けることなく、任意のワイルドカード文字をストリングに含めることができます。
大括弧外にあるその他の任意の文字	正確に一致する必要があります。

関連する概念

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

192 ページの『ConsoleUI パーツおよび関連変数』

196 ページの『UNIX 用の ConsoleUI 画面オプション』

786 ページの『EGL ソース形式のテキスト UI プログラム』

関連するタスク

190 ページの『consoleUI を使用したインターフェースの作成』

prepare

EGL **prepare** ステートメントは SQL PREPARE ステートメントを指定します。オプションで、実行時にのみ既知になる詳細が含まれます。 EGL **execute** ステートメントを実行するか、または、SQL ステートメントが結果セットを 戻す場合に、EGL **open** または **get** ステートメントを実行することで、準備済み SQL ステートメントが実行されます。



preparedStatementID

prepare ステートメントを **execute**、**open**、または **get** ステートメントに関連付ける ID。

stringExpression

有効な SQL ステートメントを含むストリング式。

SQL record name

SQL レコードの名前。 この名前の指定には、次の 2 つの利点があります。

- EGL エディターには、ユーザーの指定を基に SQL ステートメントを引き出す ダイアログがある
- **prepare** ステートメントの実行後、次の例のように、レコード名で入出力エラー値を照らし合わせ、その文が成功したかどうかを判別できる

```
try
  prepare prep01 from
    "insert into " + aTableName +
    "(empnum, empname) " +
    "value ?, ?"
  for empRecord;

onException
  if empRecord is unique
    myErrorHandler(8);
  else
    myErrorHandler(12);
  end
end
```

以下に別の例を示します。

```
myString =
  "insert into myTable " + "(empnum, empname) " +
  "value ?, ?";

try
  prepare myStatement
    from myString;
onException
  myErrorHandler(12);  // プログラムを終了
end

try
  execute myStatement
    using :myRecord.empnum,
          :myRecord.empname;
onException
  myErrorHandler(15);
end
```

上記の例で示されるように、開発者はホスト変数を指定する部分に疑問符 (?) を使用できます。その後、実行時に使用されるホスト変数の名前が、準備済み文を実行する **execute**、**open**、または **get** ステートメントを使用している文節に配置されます。

結果セットの行で実行する **prepare** ステートメントには、*resultSetIdentifier* の現行値形式の句を含めることができます。この手法は、次の状況でのみ有効です。

- 句がリテラル中にコード化されている。さらに、
- **prepare** ステートメントが実行されたときに、結果セットがオープンされている。

以下に例を示します。

```
prepare prep02 from
  "update myTable " +
  "set empname = ?, empphone = ? where current of x1" ;

execute prep02 using empname, empphone ; freeSQL prep02;
```

括弧がプラス (+) 記号の使用に与える影響の例については、『式』を参照してください。

関連する概念

- 24 ページの『パーツの参照』
- 143 ページの『レコード・タイプとプロパティ』
- 247 ページの『SQL サポート』

関連する参照項目

- 605 ページの『add』
- 613 ページの『close』
- 617 ページの『delete』
- 100 ページの『例外処理』
- 619 ページの『execute』
- 536 ページの『式』
- 631 ページの『freeSQL』
- 631 ページの『get』
- 644 ページの『get next』
- 650 ページの『get previous』
- 579 ページの『入出力エラー値』
- 93 ページの『EGL ステートメント』
- 664 ページの『open』
- 『replace』
- 71 ページの『SQL 項目のプロパティ』
- 547 ページの『テキスト式』
- 812 ページの『EGL ステートメントおよびコマンドの構文図』

print

EGL の **print** ステートメントは、印刷書式をランタイム・バッファに追加します。これについては、『印刷書式』で説明しています。

► `print printFormName ;` ◀

printFormName

プログラムに対して可視になっている印刷書式の名前。可視性についての詳細は、『パーツの参照』を参照してください。

関連する概念

- 166 ページの『印刷書式』
- 24 ページの『パーツの参照』

replace

EGL **replace** ステートメントは、変更したレコードをファイルまたはデータベースに書き込みます。



- 次の入出力操作が、同じファイル中のレコードに対する **forUpdate** オプションを含む **get** ステートメントである場合は、以降の **replace** または **delete** ステートメントは新しく読み取られるファイル・レコードに対して有効となる
- 次の入出力操作が、同じ EGL レコードに対する **forUpdate** オプションを含まない **get** ステートメント、または同じファイルに対する **close** ステートメントである場合は、ファイル・レコードは変更なしにリリースされる

forUpdate オプションに関する詳細については、『*get*』を参照してください。

SQL レコード

SQL 処理の場合、EGL **replace** ステートメントは、生成されたコードでは SQL **UPDATE** ステートメントになります。

次に置換する行は、以下の 2 つの方法のいずれかで検索する必要があります。

- **forUpdate** オプションを付けて **get** ステートメントを発行し、行を検索する。
- **open** ステートメントを発行して一連の行を選択してから、**get next** ステートメントを呼び出して対象の行を検索する。

エラー条件: **replace** ステートメントを使用した場合、無効となる主な条件には以下のものがあります。

- **UPDATE** 以外の型の SQL ステートメントを指定する
- SQL **UPDATE** ステートメントの全部ではなく一部の文節を指定する
- *resultSetID* 値が必要な場合に指定しなかった場合。詳細については、『*resultSetID*』を参照。
- 次の特性のいずれかを備えた **UPDATE** ステートメントを指定するか、受け入れた場合
 - 複数のテーブルを更新する
 - 存在しないか、関連するホスト変数との互換性がない列に関連付けられている
- 入出力オブジェクトとして SQL レコードを使用し、すべてのレコード項目が読み取り専用である場合

また、以下の状態でもエラーが発生します。

- **forUpdate** オプションを指定して EGL **get** ステートメントをカスタマイズしたが、特定の SQL テーブル列が **update** で使用可能であることを示していない場合
- **get** ステートメントに関連付けられた **replace** ステートメントが列を変更しようとした。

この直前の不一致は、以下のいずれかの方法で解決できます。

- EGL **get** ステートメントをカスタマイズする際に、使用する列名を SQL **SELECT** ステートメントの **FOR UPDATE OF** 文節に組み込む
- EGL **replace** ステートメントをカスタマイズする際に、SQL **UPDATE** ステートメントの **SET** 文節内にある列への参照を除去する
- **get** と **replace** ステートメントの両方について、デフォルトを受け入れる

暗黙的な SQL ステートメント: デフォルトでは、SQL レコードを書き込む **replace** ステートメントの結果は、以下のとおりです。

- レコード宣言におけるレコード項目と SQL テーブル列の関連付けの結果、生成されたコードにより、関連する SQL 表の列に各レコード項目からのデータがコピーされる
- レコード項目を読み取り専用として定義した場合、そのレコード項目に対応する列内の値は影響を受けない

SQL ステートメントの特性は、デフォルトでは以下のとおりです。

- SQL UPDATE ステートメントには、読み取り専用のレコード項目が含まれない
- 特定のレコードの SQL UPDATE ステートメントは、以下のステートメントのようになります。

```
UPDATE tableName
SET column01 = :recordItem01,
    column02 = :recordItem02,
    .
    .
    .
    columnNN = :recordItemNN WHERE CURRENT OF cursor
```

関連する概念

143 ページの『レコード・タイプとプロパティ』
 24 ページの『パーツの参照』
 799 ページの『resultSetID』
 798 ページの『実行単位』
 247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

605 ページの『add』
 613 ページの『close』
 617 ページの『delete』
 93 ページの『EGL ステートメント』
 100 ページの『例外処理』
 619 ページの『execute』
 631 ページの『get』
 644 ページの『get next』
 650 ページの『get previous』
 579 ページの『入出力エラー値』
 664 ページの『open』
 679 ページの『prepare』
 71 ページの『SQL 項目のプロパティ』
 1004 ページの『terminalID』

return

EGL の **return** ステートメントは、関数を終了し、オプションで呼び出し側に値を返します。



returnValue

EGL の関数宣言における **returns** 指定と互換性のある項目、リテラル、または定数。

項目はすべての点で **returns** 指定に対応している必要がありますが、リテラルおよび定数に対する規則は、以下のとおりです。

- 数値リテラルまたは定数を返せるのは、**returns** 指定内のプリミティブ型が数値型の場合だけです。
- 1 バイト文字だけを含むリテラルまたは定数を返せるのは、**returns** 指定内のプリミティブ型が **CHAR** または **MBCHAR** の場合だけです。
- 2 バイト文字だけを含むリテラルまたは定数を返せるのは、**returns** 指定内のプリミティブ型が **DBCHAR** の場合だけです。
- 1 バイト文字と 2 バイト文字の組み合わせを含むリテラルまたは定数を返せるのは、**returns** 指定内のプリミティブ型が **MBCHAR** の場合だけです。
- **returns** 指定内のプリミティブ型が **HEX** の場合は、リテラルまたは定数を返すことはできません。

returns 指定を含む関数は、値を組み込んでいる **return** ステートメントで終了する必要があります。**returns** 指定のない関数も **return** ステートメントで終了することができますが、値を組み込んではいけません。

return ステートメントは、それが **try** ブロック内の **OnException** 文節内にあっても、その関数の呼び出しの後の最初の文に制御を与えます。

set

以下のセクションでは、EGL **set** ステートメントの効果について説明しています。

- 『レコード (または固定レコード) 全体に与える影響』
- 687 ページの『書式全体への影響』
- 689 ページの『コンテキスト内のフィールドへの影響』
- 690 ページの『テキスト書式のフィールドへの影響』

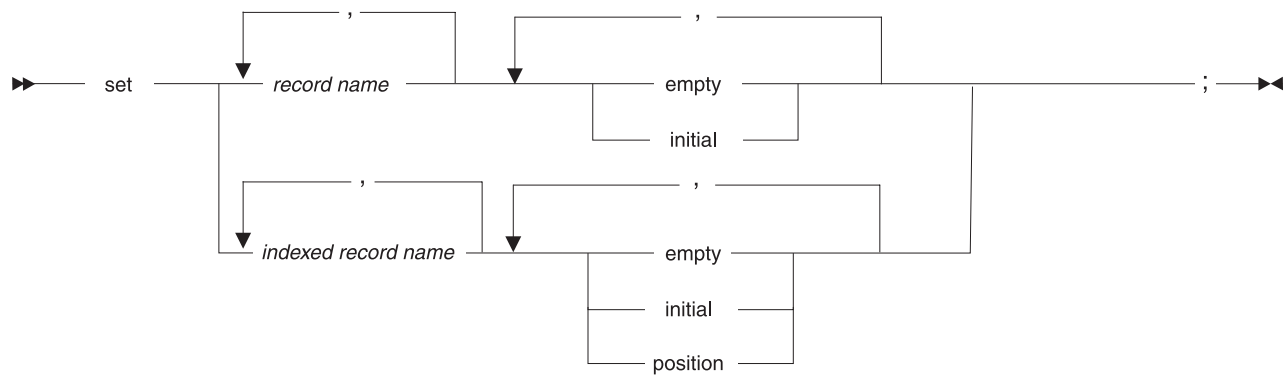
レコード (または固定レコード) 全体に与える影響

次の表は、レコード全体、固定レコード全体、またはレコードの配列に影響を及ぼす **set** ステートメントについて説明しています。

set ステートメントの形式	結果
set record empty	<p>各基本フィールドを空にします。レコードの場合、各従属レコードが空になり、それらの従属レコードの各従属も空になり、以下同様です。固定レコード（固定レコード自体がレコード内に含まれている場合があります）の場合、基本フィールドは固定構造の最下位レベルにあります。</p> <p>各基本フィールドに与える影響は、そのフィールドのプリミティブ型によって以下のように異なります。</p> <ul style="list-style-type: none"> • ANY 型フィールドの場合、set ステートメントにより、そのフィールドの現行型に基づいてフィールドが初期化されます。そのフィールドが ANY 型で、他の型を持たない場合、set ステートメントは影響を与えません。 • その他の型の詳細については、『データの初期化』を参照してください。
set record initial	<p>開発時に value プロパティによって指定された値に、フィールド値を初期化します。これは、ページ・ハンドラーまたは書式内で宣言されたレコードまたは固定レコードの場合、可能です。代入によって設定された値は、復元されません。</p> <p>value プロパティが値を持たないか、レコードがページ・ハンドラーまたは書式内でない場合、<i>set record initial</i> の結果は <i>set record empty</i> の結果と同じですが、1 つだけ例外があります。それはフィールドが ANY 型の場合、set ステートメントにより、ANY 以外のすべての型指定が除去されてしまうことです。</p>
set record position	<p>索引付きレコード型の固定レコードに関連付けられた VSAM ファイル内の位置を設定します。詳細については後述します。</p> <p>この set ステートメントの形式は、配列に対しては使用不可です。</p>

オプションの間にコンマを挿入して区切ることで、文形式を結合できます。レコードの場合、各オプションは **set** ステートメント内で指定された順序で効力を生じます。また、複数のレコードを指定することもでき、その場合は、コンマを挿入してレコード同士を区切ります。

構文図は以下のとおりです。



record name

任意型のレコード名または固定レコード名。配列を指定できます。

indexed record name

索引付きレコード型の固定レコードの名前。配列を指定できるのは、*set record position* を組み込まない場合だけです。

empty

前の表で説明されています。

initial

前の表で説明されています。

position

set value に基づいてファイル位置を設定します。この値は、索引付きレコード内のキー値です。全体的な効果は、以下のように、同じ索引付きレコードに対してコードが実行する次の入出力操作によって異なります。

- 次の操作が EGL **get next** ステートメントの場合、その文は、キー値が *set value* と等しいかより大きい最初のファイル・レコードを読み取る。そのようなレコードが存在しない場合は、**get next** ステートメントの結果は **endOfFile** になります。
- *set record position* の後ろにある次の操作が EGL **get previous** ステートメントの場合、その文は、キー値が *set value* と等しいかより小さい最初のファイル・レコードを読み取る。そのようなレコードが存在しない場合は、**get previous** の結果は **endOfFile** になります。
- *set record position* の後ろにある他のすべての操作は、ファイル位置をリセットする。このとき、*set record position* の効果はなくなります。

set value が 16 進数 FF 値で埋められている場合、以下のようになります。

- *set record position* は、ファイル内の最後のレコードの後ろにファイル位置を設定する
- 次の操作が **get previous** ステートメントの場合、ファイル内の最後のレコードが検索される

書式全体への影響

次の表は、書式全体に影響を及ぼす **set** ステートメントについて説明しています。

set ステートメントの形式	結果
set form alarm	テキスト書式の場合にのみ、 converse ステートメントにより次回書式が出力されたときにアラームが通知されます。
set form empty	書式内の各フィールドの値を空にし、すべての内容を消去します。特定のフィールドへの影響は、次のようにプリミティブ型によって異なります。 <ul style="list-style-type: none"> • ANY 型フィールドの場合、set ステートメントにより、そのフィールドの現行型に基づいてフィールドが初期化されます。そのフィールドが ANY 型で、他の型を持たない場合、set ステートメントは影響を与えません。 • その他の型の詳細については、『データの初期化』を参照してください。
set form initial	各書式フィールドを、書式宣言で表されている、最初に定義された状態に再設定します。プログラムによって行われた変更は取り消されます。ANY 型のフィールドの場合、 set ステートメントにより、ANY 以外のすべての型指定は除去されます。
set form initialAttributes	各書式フィールドを、書式宣言で表されている、最初に定義された状態に再設定します。このフィールドの値は影響を受けず、また (ANY 型のフィールドの場合) 型も影響を受けません。

empty および **alarm** などのオプションの間にコンマを挿入して区切ることで、文形式を結合できます。また、複数の書式を指定することもでき、その場合は、コンマを挿入して書式同士を区切ります。

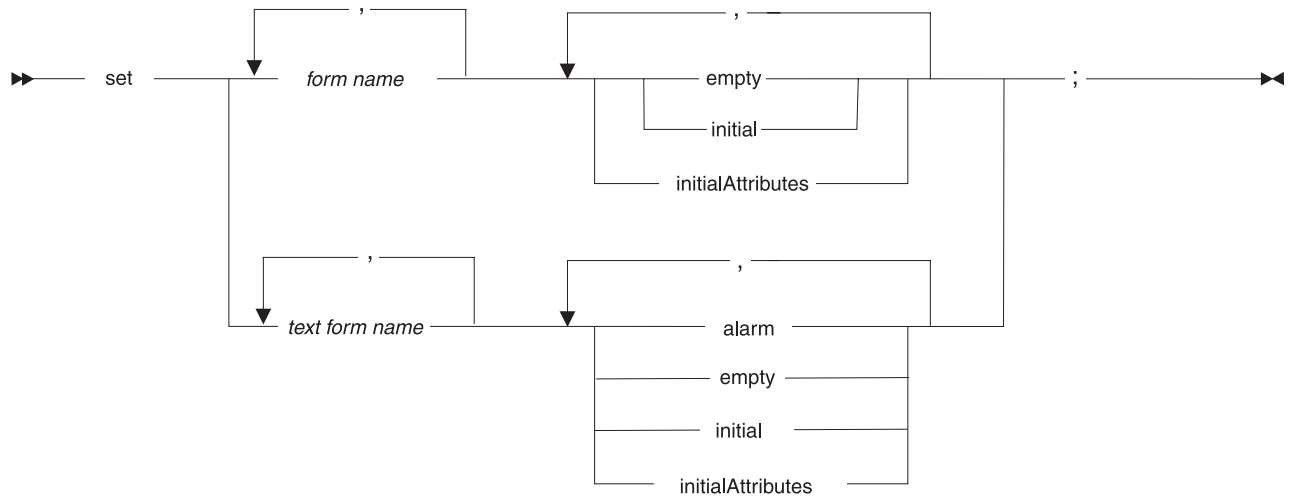
次の形式の 1 つを選択するか、まったく選択しないことができます。

- *set form empty*
- *set form initial*

次の形式の 1 つ、または両方を選択するか、まったく選択しないことができます。

- *set form alarm* (テキスト書式でのみ使用可能)
- *set form initialAttributes*

構文図は以下のとおりです。



form name

text または *print* タイプの書式の名前。これについては、『書式パーツ』で説明されています。

text form name

text タイプの書式の名前。これについては、『書式パーツ』で説明されています。

オプションについては、前の表で説明されています。

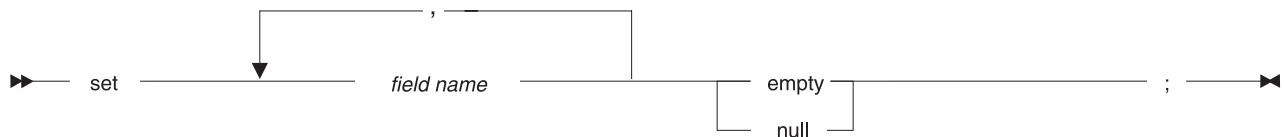
コンテキスト内のフィールドへの影響

次の表は、コンテキスト内のフィールドに影響を及ぼす **set** ステートメントの形式について説明しています。

set ステートメントの形式	結果
set field empty	<p>フィールドを空にし、(副構造を持つ固定フィールドの場合) すべての従属フィールド、基本フィールドを空にします。</p> <p>影響は、フィールドのプリミティブ型によって以下のように異なります。</p> <ul style="list-style-type: none"> • ANY 型フィールドの場合、set ステートメントにより、そのフィールドの現行型に基づいてフィールドが初期化されます。そのフィールドが ANY 型で、他の型を持たない場合、set ステートメントは影響を与えません。 • その他の型の詳細については、『データの初期化』を参照してください。

set ステートメントの形式	結果
set field null	フィールドを Null にすることが有効な場合、フィールドを NULL に設定します。この操作が有効な場合について詳しくは、『itemsNullable』を参照してください。SQL レコードでの NULL 処理の詳細については、『SQL 項目のプロパティ』を参照してください。

構文図は以下のとおりです。



field name

フィールドの名前。

どちらか一方のオプションを選択できます。各オプションについては、前述の表を参照してください。

テキスト書式のフィールドへの影響

次の表は、テキスト書式のフィールドまたはフィールドの配列に影響を及ぼす **set** ステートメントについて説明しています。後で説明されているように、特定の方法的組み合わせでのみ、特定の **set** オプションにより文を結合できます。

注: 説明されている動作の多くは、テキスト書式が表示される装置に依存します。サポートしている各装置上で、出力のテストを行うことをお勧めします。

set ステートメントの形式	結果
set field blink	テキストの点滅を繰り返します。このオプションを使用できるのは、COBOL プログラムに限られます。
set field bold	テキストをボールド体で表示します。
set field cursor	指定されたフィールドにカーソルを置きます。 フィールドが配列であると識別され、occurs 値を持たない場合、カーソルは、デフォルトで最初の配列エレメントに置かれます。 プログラムにより、 <i>set field cursor</i> 形式の複数の文が実行される場合、最後の文は、 converse ステートメントが実行されたときに有効になります。

set ステートメントの形式	結果
set field defaultColor	フィールド固有の color プロパティを <i>defaultColor</i> に設定します。これは、他の条件により表示色が決定されることを意味します。詳細については、『フィールド表示プロパティ』を参照してください。
set field dim	フィールドを通常より低い輝度で表示します。この効果は、フィールド内容の強調解除に使用します。
set field empty	フィールドの値を初期化し、すべての内容を消去します。特定のフィールドへの影響は、『データの初期化』で説明されているように、プリミティブ型によって異なります。
set field full	<p>書式が表示される前に、空、ブランク、または NULL のフィールドに、次のような一連の同一文字を設定します。</p> <ul style="list-style-type: none"> フィールド・プロパティ fillCharacter が次の値の場合、文字はアスタリスク (*) です。また、次の値は fillCharacter のデフォルト値でもあります。 <ul style="list-style-type: none"> HEX 型のフィールドの場合は 0 数値型のフィールドの場合はスペース その他のフィールドの場合は空ストリング fillCharacter が上記のように設定されていない場合、文字は fillCharacter の値と同一に設定されます。 <p>『変更データ・タグとプロパティ』で説明されているように、フィールドに変更データ・タグが設定されている場合にのみ、書式上の文字がプログラムに戻されます。ユーザーがフィールドを変更した場合は、プログラムにフィールド上の文字が戻されないように、それらの文字をすべて除去する必要があります。</p> <p>書式グループが、ビルド記述子オプション <i>setFormItemFull</i> で生成された場合にのみ、<i>set field full</i> の使用が有効になります。</p> <p>MBCHAR 型のフィールドは、そこにすべて単一バイト・スペースが含まれている場合、空であるとみなされます。そのようなフィールドに対して、<i>set field full</i> により一連の単一バイト文字が割り当てられます。</p>
set field initial	プログラムにより行われた変更とは関係なく、フィールドを最初に定義された状態に再設定します。

set ステートメントの形式	結果
set field initialAttributes	value プロパティ（フィールドの現行内容を指定するプロパティ）を使用せずに、フィールドを最初に定義した状態にリセットします。
set field invisible	フィールド・テキストを不可視にします。
set field masked	パスワード・フィールドに使用。Java プログラムによってテキスト書式が提示される場合に、ユーザーが入力フィールドに入力した非ブランク文字の代わりにアスタリスクが表示されます。
set field modified	『変更データ・タグとプロパティ』で説明されているように、変更データ・タグを設定します。
set field noHighlight	点滅、反転、およびアンダースコアの特殊効果を除去します。
set field normal	以下の形式に関連する説明のとおり、フィールドを再設定します。 <ul style="list-style-type: none"> • Set field normalIntensity • Set field unmodified • Set field unprotected 詳細については、次の表を参照してください。
set field normalIntensity	フィールドを太文字を使用せずに表示します。
set field protect	ユーザーがフィールドの値を上書きできないよう設定します。『set field skip』も参照してください。
set field reverse	テキストおよび背景色を反転し、（例えば）濃い背景色に淡色文字を表示した場合に背景色を明るくしてテキストを濃い色にします。
set field <i>selectedColor</i>	フィールド固有の color プロパティを指定した値に設定します。 <i>selectedColor</i> の有効な値は以下のとおりです。 <ul style="list-style-type: none"> • black • blue • green • pink • red • turquoise • white • yellow

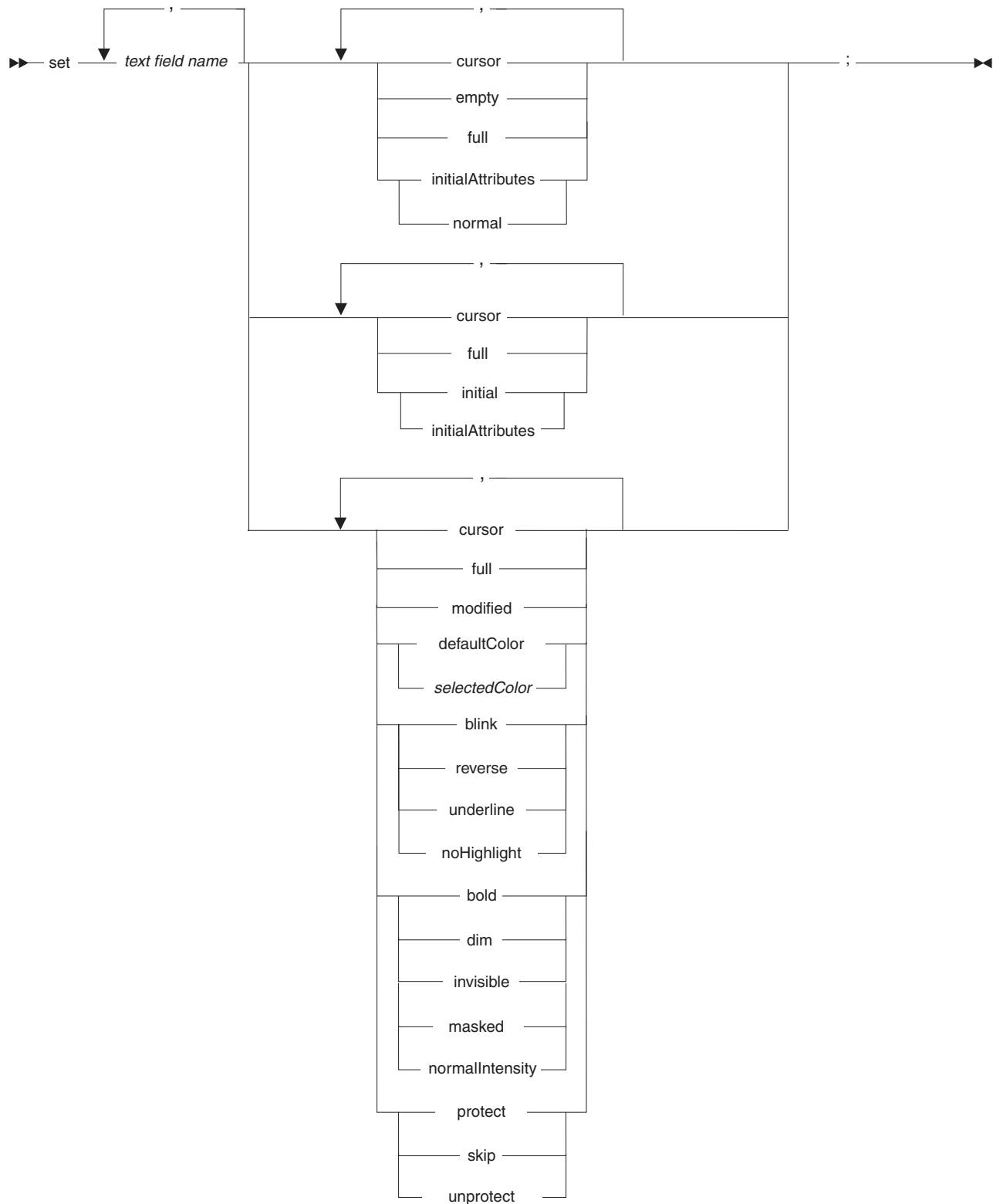
set ステートメントの形式	結果
set field skip	<p>ユーザーがフィールドの値を上書きできないよう設定します。また、次のいずれかの場合には、カーソルがフィールドをスキップします。</p> <ul style="list-style-type: none"> ユーザーがタブ順序の直前のフィールドを操作しているときに、Tab を押したか、そのフィールドに内容を入力した場合。または、 ユーザーがタブ順序の次のフィールドを操作していて、Shift + Tab を押した場合。
set field underline	フィールドの下部にアンダースコアを配置します。
set field unprotect	ユーザーがフィールドの値を上書きできるよう設定します。

次の 3 つの方法により、**cursor** および **full** などのオプションの間にコンマを挿入して区切ることで、文形式を結合できます。

1. **set** ステートメントは以下のように構成できます。
 - 次のフィールド属性形式の 1 つを選択するか、または選択しない。
 - *set field initialAttributes*
 - *set field normal*
 - 任意の数の以下の形式を選択する。
 - *set field cursor*
 - *set field empty*
 - *set field full*
2. 次に、任意の数の以下の形式から **set** ステートメントを構成することができます。
 - *set field cursor*
 - *set field full*
 - *set field initial* または *set field initialAttributes*
3. 最後に、次のように **set** ステートメントを構成できます。
 - 任意の数の以下の形式を選択する。
 - *set field cursor*
 - *set field full*
 - *set field modified*
 - カラー形式を 1 つ選択するか、選択しない。
 - *set field defaultColor*
 - *set field selectedColor*
 - 強調表示形式を 1 つ選択するか、選択しない。
 - *set field blink*
 - *set field reverse*

- *set field underline*
- *set field noHighlight*
- 輝度形式を 1 つ選択するか、選択しない。
 - *set field bold*
 - *set field dim*
 - *set field invisible*
 - *set field masked*
 - *set field normalIntensity*
- 保護形式を 1 つ選択するか、選択しない。
 - *set field protect*
 - *set field skip*
 - *set field unprotect*

構文図は以下のとおりです。



field name

テキスト書式のフィールドの名前名前はフィールドの配列を参照できます。

オプションについては、前の表で説明されています。

関連する概念

- 164 ページの『書式パーツ』
- 171 ページの『変更データ・タグおよびプロパティ』
- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 511 ページの『データの初期化』
- 93 ページの『EGL ステートメント』
- 70 ページの『フィールド表示プロパティ』
- 644 ページの『get next』
- 650 ページの『get previous』
- 430 ページの『itemsNullable』

71 ページの『SQL 項目のプロパティ』

show

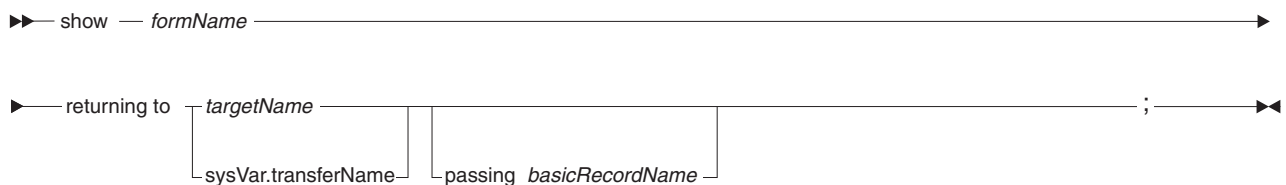
show ステートメントは、以下のようにしてメイン・プログラムからのテキスト書式を表示します。

1. 回復可能リソースのコミット、ファイルのクローズ、ロックの解除
2. オプションで、**show** ステートメントの戻り文節があれば、そこで指定されているプログラムが使用する基本レコードを渡す。
3. 最初のプログラムを終了する。
4. テキスト書式を印刷する。

呼び出し先プログラムでは **show** ステートメントは使用できません。

show ステートメントに戻り文節を組み込んだ場合は、ユーザーがイベント・キーを押したときに、EGL ランタイムが、指定されたプログラムを呼び出します。書式データは、受け取り側プログラムの入力書式に割り当てられます。(ユーザー入力によって変更されていない) 渡されたレコードは、受け取り側プログラムの入力レコードに割り当てられます。

戻り文節を組み込んでいない場合は、テキスト書式が表示されたときに操作が終了します。



formPartName

プログラムに対して可視になっているテキスト書式の名前。可視性についての詳細は、『パーツの参照』を参照してください。この文に戻り文節を組み込んだ場合は、テキスト書式は、呼び出されるプログラムの **inputForm** プロパティで指定されている書式と等価である必要があります。

sysVar.transferName

呼び出されるプログラムの ID を含むシステム変数。実行時に ID を設定する場合は、この変数を使用してください。

basicRecordName

basicRecord タイプのレコードの名前。その内容は、受け取り側プログラムの入力レコードに割り当てられます。

関連する概念

24 ページの『パーツの参照』

関連する参照項目

1005 ページの『transferName』

transfer

EGL の **transfer** ステートメントは、メイン・プログラムから別のプログラムに制御を与えて、転送側のプログラムを終了します。さらにオプションで、受け取り側プログラムの入力レコードが受け取るデータを持っているレコードを渡します。呼び出し先プログラムでは、**transfer** ステートメントは使用することができません。

プログラムは、トランザクションへの移動またはプログラムへの移動という書式の文によって、制御を移動することができます。

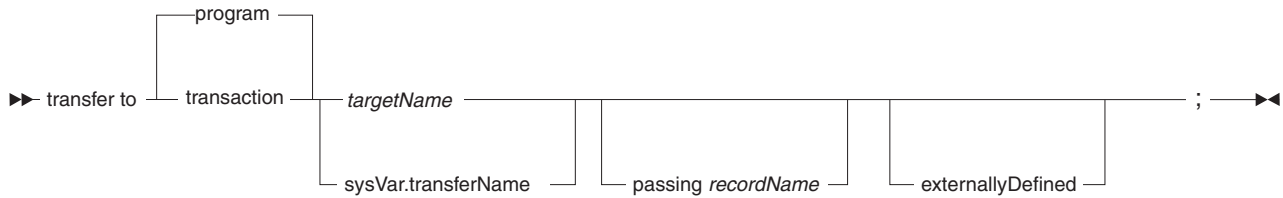
- トランザクションへの転送は、以下のように行われます。
 - Java メイン・テキストまたはメイン・バッチ・プログラムとして実行するプログラムでは、この振る舞いは、ビルド記述子オプション **synchOnTrxTransfer** の設定によって異なります。
 - **synchOnTrxTransfer** の値が YES の場合は、**transfer** ステートメントは回復可能リソースをコミットしてファイルをクローズし、次にカーソルをクローズして、同じ実行単位にあるプログラムを開始します。
 - **synchOnTrxTransfer** の値が NO (デフォルト) の場合も **transfer** ステートメントは同じ実行単位内のプログラムを開始しますが、呼び出し先プログラムで使用可能なリソースは、クローズまたはコミットしません。
 - ページ・ハンドラーでは、トランザクションへの転送は無効です。代わりに、**forward** ステートメントを使用してください。

リンケージ・オプション・パーツである **transferLink** 要素は、Java コードから Java コードへ制御を転送する場合は何の効果も与えませんが、それ以外では意味があります。

EGL または VisualAge Generator を使って作成されていないコードに制御コードを転送する場合は、リンケージ・オプション・パーツである **transferLink** 要素を設定することをお勧めします。 **linkType** プロパティを *externallyDefined* に設定します。

VisualAge Generator との互換性モードで実行している場合は、VisualAge Generator からマイグレーションされたプログラムの場合のように、**transfer** ステートメントで **externallyDefined** オプションを指定することができます。ただし、その代わりにリ

ンケージ・オプション・パーツに等価な値を設定することをお勧めします。
VisualAge Generator との互換性モードについての詳細は、『VisualAge Generator との互換性』を参照してください。



program *targetName* (デフォルト)

制御を受け取るプログラム。

transaction *targetName*

すでに説明した、制御を受け取るプログラム。

sysVar.transferName

実行時に設定可能なターゲット名を含むシステム関数。詳細については、『sysVar.transferName』を参照してください。

passing *recordName*

ターゲット・プログラムで、入力レコードとして受け取られるレコード。渡されるレコードの型は任意ですが、その長さでプリミティブ型は、データを受け取るレコードと互換性がある必要があります。ターゲット・プログラムの入力レコードは、basicRecord 型であることが必要です。

関連する概念

477 ページの『VisualAge Generator との互換性』

720 ページの『別名の割り当て』

関連する参照項目

1005 ページの『transferName』

try

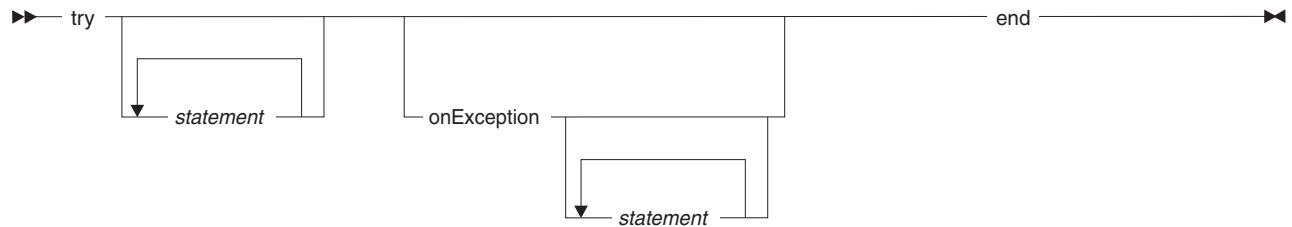
EGL **try** ステートメントは、以下のいずれかの文がエラーになり、**try** 文内にある場合にプログラムの実行が継続されていることを示します。

- input/output (I/O) ステートメント
- システム関数呼び出し
- **call** ステートメント

例外が発生すると、処理は、**onException** ブロック (ある場合) の最初の文または **try** ステートメントの直後の文で再開されます。ただし、入出力ハード・エラーは、システム変数 **VGVar.handleHardIOErrors** が 1 に設定されている場合にのみ処理されます。その他の場合、プログラムは、メッセージを表示 (可能な場合) して終了します。

try ステートメントは、**try** 文内で呼び出された関数やプログラムに例外が発生した場合のランタイムの振る舞いに影響を与えません。

詳細については、『例外処理』を参照してください。



statement

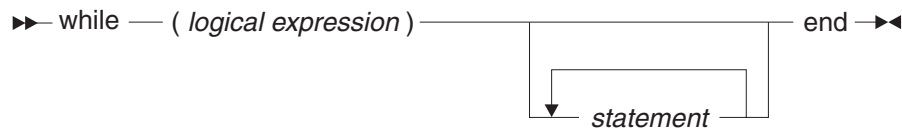
EGL ステートメント。

OnException

例外条件が発生した場合に実行される文のブロック。

while

EGL キーワード **while** は、ループで実行される一連の文の始まりを示します。1 回目のステートメントが実行されるのは、論理式が真になる場合のみであり、2 回目以降が繰り返されるかどうか同じ条件に左右されます。キーワード **end** は、**while** ステートメントの終わりを示します。



logical expression

真または偽に評価される式 (一連のオペランドおよび演算子)

statement

EGL 言語の文

以下に例を示します。

```
sum = 0;
i = 1;
while (i < 4)
    sum = inputArray[i] + sum;
    i = i + 1;
end
```

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

538 ページの『論理式』

93 ページの『EGL ステートメント』

ライブラリー (生成された出力)

Java 出力のライブラリー・パーツは、Java クラスとして生成されます。クラス名はパーツ別名 (別名が指定されていない場合はパーツ名) です。ただし、必要に応じて EGL は文字の置換を行います。詳しくは、『*Java* 名の別名の割り当て方法』を参照してください。

関連する概念

151 ページの『basicLibrary タイプのライブラリー・パーツ』
151 ページの『basicLibrary タイプのライブラリー・パーツ』
798 ページの『実行単位』

関連するタスク

721 ページの『Java 名の別名の割り当て方法』

関連する参照項目

『EGL ソース形式のライブラリー・パーツ』

EGL ソース形式のライブラリー・パーツ

ライブラリー・パーツは EGL ファイルで宣言します。これについては、『*EGL* ソース形式』で説明しています。

ライブラリー・パーツの例は以下のとおりです。

Library CustomerLib3

```
// Use declarations
Use StatusLib;

// Data Declarations
exceptionId ExceptionId ;

// Retrieve one customer for an email
//   In: customer, with emailAddress set
//   Out: customer, status
Function getCustomerByEmail ( customer CustomerForEmail, status int )
status = StatusLib.success;
try
  get customer ;
onException
  exceptionId = "getCustomerByEmail" ;
  status = sqlCode ;
end
commit();
end

// Retrieve one customer for a customer ID
//   In: customer, with customer ID set
//   Out: customer, status
Function getCustomerByCustomerId ( customer Customer, status int )
status = StatusLib.success;
try
  get customer ;
onException
  exceptionId = "getCustomerByCusomerId" ;
  status = sqlCode ;
end
```

```

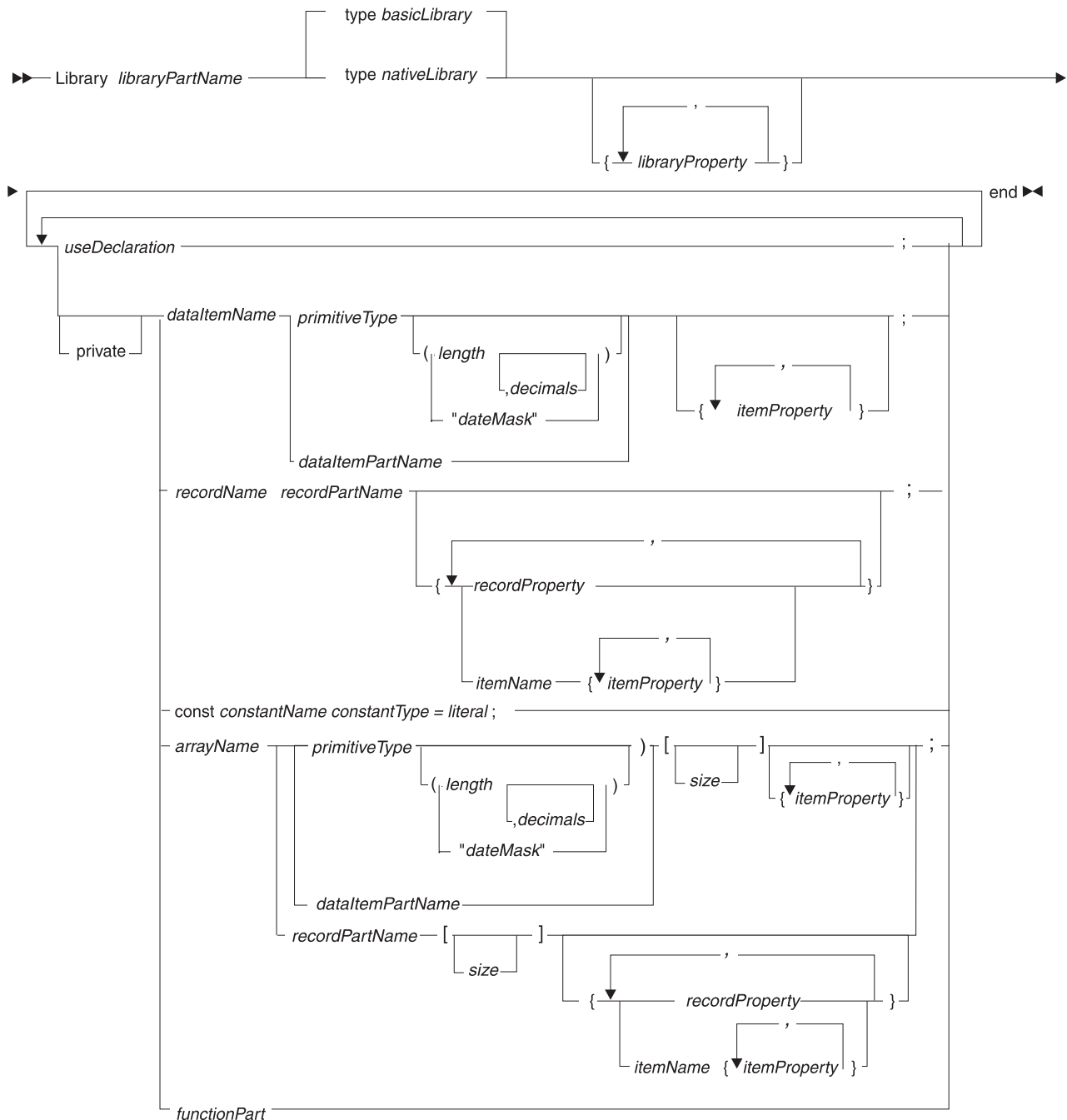
        commit();
    end

    // Retrieve multiple customers for an email
    //   In: startId
    //   Out: customers, status
    Function getCustomersByCustomerId
    ( startId CustomerId, customers Customer[], status int )
        status = StatusLib.success;
        try
            get customers usingKeys startId ;
        onException
            exceptionId = "getCustomerForEmail" ;
            status = sqlCode ;
        end
        commit();
    end

end

```

ライブラリー・パーツのダイアグラムは、次のとおりです。



Library libraryPartName ... end

パーツをライブラリー・パーツとして識別し、その名前を指定します。**alias** プロパティー (後述) を設定していない場合は、生成されるライブラリーの名前は *libraryPartName* になります。

その他の規則については、『命名規則』を参照してください。

basicLibrary 型、nativeLibrary 型

以下のライブラリー型を示します。

- 基本ライブラリー (basicLibrary 型) には、EGL によって記述された関数および他の EGL ロジックでのランタイムに使用される値が含まれます。詳しくは、『*basicLibrary* 型のライブラリー・パーツ』を参照してください。
- ネイティブ・ライブラリー (nativeLibrary 型) は、外部 DLL 用のインターフェースとして機能します。詳細は、『*nativeLibrary* 型のライブラリー・パーツ』を参照してください。

このライブラリーは、デフォルトでは basicLibrary 型です。

libraryProperties

ライブラリー・プロパティは以下のとおりです。

- **alias**
- **allowUnqualifiedItemReferences**
- **callingConvention** (nativeLibrary 型のライブラリー内のみで使用可能)
- **dllName** (nativeLibrary 型のライブラリー内のみで使用可能)
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **messageTablePrefix**
- **throwNrfEofExceptions**

すべてオプションです。

- **alias** = "*alias*" は、生成された出力の名前に組み込まれるストリングを識別します。**alias** プロパティを設定しない場合、プログラム・パーツ名が代わりに使用されます。
- **allowUnqualifiedItemReferences** = no、**allowUnqualifiedItemReferences** = yes は、構造体項目のコンテナ (データ・テーブル、レコード、または構造体項目を保持している書式) の名前を除外しても、コードが構造体項目を参照できるようにするかどうかを指定します。例えば、次のレコード・パーツについて考えましょう。

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

以下の変数はそのパーツに基づいています。

```
myRecord aRecordPart;
```

allowUnqualifiedItemReferences のデフォルト値 (*no*) を受け入れる場合、以下の代入のように myItem01 を参照するときにレコード名を指定する必要があります。

```
myValue = myRecord.myItem01;
```

ただし、**allowUnqualifiedItemReferences** プロパティを *yes* に設定した場合は、次に示すように、レコード名を指定する必要はありません。

```
myValue = myItem01;
```


デフォルト値 (最良実例) を受け入れることをお勧めします。コンテナ名を指定することにより、コードを読み取る人と EGL に対するあいまいさを減らすことができます。

EGL は規則セットを使用して変数名または項目名が参照するメモリー領域を決定します。詳細については、『変数および定数の参照』を参照してください。

- **nativeLibrary** 型のライブラリーで使用されるように、**callingConvention = I4GL** は、EGL ランタイムにより、データが次の 2 種類のコード間でどのように受け渡しされるかを指定します。
 - ライブラリー関数を呼び出す EGL コード、および
 - アクセスされている DLL 内の関数

callingConvention に対して、現在使用可能な値は、*I4GL* のみです。追加情報については、『*nativeLibrary* 型のライブラリー・パーツ』を参照してください。

- **nativeLibrary** 型のライブラリーで使用されるように、**dllName** は、最終の DLL 名を指定します。この名前は、デプロイメント時にはオーバーライドできません。ライブラリー・プロパティー **dllName** に値を指定しない場合、Java ランタイム・プロパティー `vgj.defaultI4GLNativeLibrary` 内で DLL 名を指定する必要があります。

追加情報については、『*nativeLibrary* 型のライブラリー・パーツ』を参照してください。

- **handleHardIOErrors = yes**、**handleHardIOErrors = no** により、システム変数 **VGVar.handleHardIOErrors** にデフォルト値が設定されます。このシステム変数は、try ブロック内の入出力操作でハード・エラーが発生した後に、プログラムを継続して実行するかどうかを制御します。このプロパティーのデフォルト値は *yes* であり、この変数は 1 に設定されます。

詳細については、『*VGVar.handleHardIOErrors*』および『*Exception handling*』を参照してください。

- **includeReferencedFunctions = no**、**includeReferencedFunctions = yes** は、ライブラリーに、そのライブラリー内にない関数、および現行のライブラリーがアクセスするライブラリー内にない関数を含めるかどうかを示します。デフォルト値は *no* です。このライブラリーの一部であるすべての関数がこのライブラリー内にある場合は、このプロパティーを無視することができます。

ライブラリーが、そのライブラリーにない共用関数を使用している場合は、**includeReferencedFunctions** プロパティーを *yes* に設定した場合に限り、生成が可能になります。

- **localSQLScope = yes**、**localSQLScope = no** は、SQL 結果セットおよび作成されたステートメントの識別子が、プログラムまたはページ・ハンドラーによる呼び出し中に、ライブラリー・コードに対してローカルである (これがデフォルトです) かどうかを示します。値 *yes* をそのまま使用すると、異なる複数のプログラムが同じ識別子を独自に使用できます。そのライブラリーを使用するプログラムまたはページ・ハンドラーは、ライブラリー内で使用されているのと同じ識別子を独自に使用することができます。

no を指定すると、識別子はその実行単位全体で共用されます。そのライブラリー内の SQL ステートメントが呼び出されたときに作成された識別子は、そのライブラリーを呼び出すその他のコード内でも使用可能です。ただし、その他のコードは、**localSQLScope = yes** を使用して、これらの識別子へのアクセスをブロックすることができます。またライブラリーは、呼び出し側プログラムまたはページ・ハンドラーで作成される識別子を参照できます。ただし、参照できるのは、SQL 関連ステートメントが既に他のコードで実行された場合、およびその他のコードがアクセスをブロックしなかった場合だけです。

SQL 識別子を共用する効果は、次のとおりです。

- 1 つのコードで結果セットを開き、別のコードでその結果セットから行を取得できる。
- 1 つのコードで SQL ステートメントを作成し、そのステートメントを別のコードで実行できる。

いずれの場合でも、プログラムまたはページ・ハンドラーがライブラリーにアクセスするときに使用可能な識別子は、同じプログラムまたはページ・ハンドラーが、同じライブラリー内の同じ関数または別の関数にアクセスするときに使用可能です。

- **msgTablePrefix = " prefix"** は、メッセージ・テーブルとして使用されるデータ・テーブルの名前内の先頭の 1 から 4 文字を指定します。(メッセージ・テーブルは、ライブラリー関数が出力した書式から使用できます。)この名前のその他の文字は、『EGL ソース形式の DataTable パーツ』にリストされている各国語コードの 1 つに対応します。
- **throwNrfEofExceptions = no**、**throwNrfEofExceptions = yes** は、ソフト・エラーにより例外がスローされるかどうかを指定します。デフォルトは *no* です。背景情報については、『例外処理』を参照してください。

useDeclaration

データ・テーブルまたはライブラリーに簡単にアクセスできるようにします。これは、書式グループ内の書式にアクセスする場合に必要になります。詳細については、『使用宣言』を参照してください。

private

ライブラリー外では、変数、定数、または関数を使用できないことを示します。**private** という用語を省略した場合は、変数、定数、または関数を使用できません。

nativeLibrary 型のライブラリー内の関数には、**private** を指定できません。

dataItemName

データ項目の名前。命名の規則については、『命名規則』を参照してください。

primitiveType

データ項目のプリミティブ型または (配列に関連付けられている) 配列エレメントのプリミティブ型。

length

パラメーターの長さ、または (配列に関して) 配列エレメントの長さ。この長さは、*dataItemName* または (配列の場合は) *dynamicArrayName* のいずれかによって参照されるメモリー領域の文字数または桁数を表す整数です。

decimals

数値型には、*decimals* を指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

"dateTimeMask"

TIMESTAMP 型および INTERVAL 型には、*"dateTimeMask"* を指定できます。これは、日時値の特定の位置に意味（「年の桁」など）を割り当てるものです。このマスクは、データと一緒に格納されません。

dataItemPartName

プログラムに対して可視の *dataItem* パーツ名。可視性についての詳細は、『[パーツの参照](#)』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『[Typedef](#)』を参照してください。

recordName

レコードの名前。命名の規則については、『[命名規則](#)』を参照してください。

recordPartName

プログラムに対して可視のレコード・パーツ名。可視性についての詳細は、『[パーツの参照](#)』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『[Typedef](#)』を参照してください。

constantName literal

定数の名前および値。値は、引用符付きストリングまたは数値となります。命名の規則については、『[命名規則](#)』を参照してください。

itemProperty

項目固有のプロパティと値の組み。『[EGL プロパティとオーバーライドの概要](#)』を参照してください。

recordProperty

レコード固有のプロパティと値の組み。使用可能なプロパティの詳細については、必要なレコード・タイプに関する[参照トピック](#)を参照してください。

基本レコードにはプロパティはありません。

itemName

オーバーライドするプロパティを持つレコード項目の名前。『[EGL プロパティとオーバーライドの概要](#)』を参照してください。

arrayName

レコードまたはデータ項目の動的または静的配列の名前。このオプションを使用する場合、右側の他のシンボル (*dataItemPartName*、*primitiveType* など) は配列の各要素を参照します。

size

配列内の要素数。要素の数を指定すると、その配列は静的になります。指定しない場合、配列は動的です。

functionPart

関数。関数内のどのパラメーターも、緩い型にはできません。詳細については、『[EGL ソース形式の関数パーツ](#)』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 151 ページの『basicLibrary タイプのライブラリー・パーツ』
- 68 ページの『EGL プロパティの概要』
- 24 ページの『パーツの参照』
- 62 ページの『EGL での変数の参照』
- 30 ページの『Typedef』

関連する参照項目

- 413 ページの『EGL ソース形式の基本レコード・パーツ』
 - 513 ページの『EGL ソース形式の DataTable パーツ』
 - 532 ページの『EGL ソース形式』
 - 100 ページの『例外処理』
 - 570 ページの『EGL ソース形式の関数パーツ』
 - 578 ページの『EGL ソース形式の索引付きレコード・パーツ』
 - 792 ページの『入力書式』
 - 792 ページの『入力レコード』
 - 579 ページの『入出力エラー値』
 - 584 ページの『Java ランタイム・プロパティ (詳細)』
 - 714 ページの『EGL ソース形式の MQ レコード・パーツ』
 - 37 ページの『プリミティブ型』
 - 796 ページの『EGL ソース形式の相対レコード・パーツ』
 - 799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
 - 804 ページの『EGL ソース形式の SQL レコード・パーツ』
-
- 1020 ページの『使用宣言』
 - 1012 ページの『handleHardIOErrors』

like 演算子

論理式の中で、テキスト式を別の (*like* 基準 と呼ばれる) スtringと、文字位置ごとに左から右へ比較することができます。この機能の用途は、SQL 照会における SQL キーワード **like** の用途によく似ています。

以下に例を示します。

```
// 変数 myVar01 はString式である
// その内容が like 基準と比較される
myVar01 = "abcdef";

// 次の論理式は「真」へと評価される
if (myVar01 like "a_c%")
;
end
```

like 基準は、リテラル、CHAR 型または MBCHAR 型の項目、UNICODE 型の項目のいずれかにすることができます。like 基準の中には、以下の文字を組み込むことができます。

% ワイルドカードとして機能し、String式内のゼロ個以上の任意の文字に一致します。

_ (アンダースコア)

ワイルドカードとして機能し、ストリング式内の単一の文字に一致します。

¥ 次の文字をストリング式内の単一の文字と比較するよう指示します。円記号 (¥) は、通常の処理から外れるので、エスケープ文字 と呼ばれます。このエスケープ文字はストリング式内の任意の文字とも比較されません。

通常、このエスケープ文字は、% 記号、アンダースコア (_)、またはもう 1 つの円記号の前に置かれます。

円記号を (デフォルトの振る舞いどおりに) エスケープ文字として使用した場合、EGL は同じエスケープ文字を使用して任意のテキスト式内への引用符の組み込みを許可するので、問題が起きます。 like 基準の内容においては、2 つの円記号を指定する必要があります。実行時に使用可能なテキストは、最初の円記号を除いたテキストだからです。

この問題を回避することをお勧めします。後で例を示すように、escape 文節を使用することにより、別の文字をエスケープ文字として指定してください。ただし、二重引用符 (") をエスケープ文字として使用することはできません。

それ以外の *likeCriterion* 内の文字は、すべてリテラルであり、ストリング式 内の単一の文字と比較されます。

次の例は、escape 文節の使用方法を示しています。

```
// 変数 myVar01 はストリング式である
// その内容が like 基準と比較される
myVar01 = "ab%def";

// 次の論理式は「真」へと評価される
if (myVar01 like "ab¥¥%def")
;
end

// 次の論理式は「真」へと評価される
if (myVar01 like "ab+%def" escape "+")
;
end
```

関連する参照項目

93 ページの『EGL ステートメント』

538 ページの『論理式』

547 ページの『テキスト式』

リンケージ・プロパティ・ファイル (詳細)

呼び出し側の Java プログラムまたはラッパーを生成する場合は、実行時にリンケージ情報が必要であるという要求を指定できます。この指定を行うには、呼び出し先プログラムのリンケージ・オプションの値を次のように設定します。

- callLink エLEMENTのプロパティ **type** の値に remoteCall または ejbCall を指定します。
- callLink エLEMENTのプロパティ **remoteBind** に RUNTIME を指定します。

リンケージ・プロパティ・ファイルは手書きしても構いませんが、前述した設定に加えて、ビルド記述子オプション **genProperties** に GLOBAL または PROGRAM を指定して Java プログラムまたはラッパーを生成すると、EGL によってファイルが生成されます。

実行時のリンケージ・プロパティ・ファイルの識別

リンケージ・オプション・パーツ内で、呼び出し先プログラムの **callLink** エレメントのプロパティ **remoteBind** に RUNTIME を設定した場合、リンケージ・プロパティ・ファイルは実行時に検索されますが、ファイル名のソースは、Java プログラムと Java ラッパーとで異なります。

- Java プログラムは、Java ランタイム・プロパティ **cso.linkageOptions.LO** を検査します。ここで、**LO** は、生成で使用するリンケージ・オプション・パーツ名です。プロパティが存在しない場合、EGL ランタイム・コードは、**LO.properties** という名前のリンケージ・プロパティ・ファイルを検索します。繰り返しますが、**LO** は、生成で使用するリンケージ・オプション・パーツ名です。

この場合、EGL ランタイム・コードがリンケージ・オプション・プロパティ・ファイルを検索しても、それが見付からないと、リンケージ・オプション・プロパティ・ファイルの使用を要求する先頭の **call** ステートメントでエラーが発生します。結果の詳細については、『例外処理』を参照してください。

- Java ラッパーは、**callOptions** 型のプログラム・オブジェクト変数 **callOptions** に、リンケージ・プロパティ・ファイルの名前を保管します。生成されるファイル名は、**LO.properties** です。ここで、**LO** は、生成で使用するリンケージ・オプション・パーツ名です。

この場合、Java 仮想マシンがリンケージ・プロパティ・ファイルを検索しても、それが見付からないと、プログラム・オブジェクトは、**CSOException** 型の例外をスローします。

リンケージ・プロパティ・ファイルの形式

実行時に使用するために、リンケージ・プロパティ・ファイルには、デプロイする生成済みの Java プログラムまたはラッパーからの各呼び出しを処理するための一連のエントリーが含まれます。

基本エントリーの型は **cso.serverLinkage** であり、基本エントリーには、リンケージ・オプション・パーツの **callLink** エレメントに設定できるプロパティと値の組みを指定できます。ただし、次の場合は例外です。

- **remoteBind** プロパティは必ず RUNTIME になり、表示されません。
- **type** プロパティを **localCall** にすることはできません。ローカル呼び出しのリンケージは生成時に必ず確立されるからです。

cso.serverLinkage エントリー

基本的な事例では、リンケージ・プロパティ・ファイルの各エントリーの型は、**cso.serverLinkage** になる場合がほとんどです。エントリーのフォーマットは次のとおりです。

```
cso.serverLinkage.programName.property=value
```


programName

呼び出し先プログラムの名前。呼び出し先プログラムが EGL によって生成された場合は、指定した名前はプログラム・パーツ名になります。

property

Java プログラムに適したプロパティ。ただし、**remoteBind** プロパティと **pgmName** プロパティは除きます。詳細については、『*callLink* エlement』を参照してください。

value

指定したプロパティに対して有効な値。

呼び出し先プログラム *xyz* の例を次に示します。*xxx* は、大文字と小文字を区別するストリングを表しています。

```
cso.serverLinkage.Xyz.type=ejbCall
cso.serverLinkage.Xyz.remoteComType=TCPIP
cso.serverLinkage.Xyz.remotePgmType=EGL
cso.serverLinkage.Xyz.externalName=xxx
cso.serverLinkage.Xyz.package=xxx
cso.serverLinkage.Xyz.conversionTable=xxx
cso.serverLinkage.Xyz.location=xxx
cso.serverLinkage.Xyz.serverID=xxx
cso.serverLinkage.Xyz.parmForm=COMMDATA
cso.serverLinkage.Xyz.providerURL=xxx
cso.serverLinkage.Xyz.luwControl=CLIENT
```

リテラル値である TCPIP や EGL など大文字と小文字を区別しないので、有効なデータの例になります。

cso.application エントリー

いくつかの呼び出し先プログラムを参照する一連の *cso.serverLinkage* エントリーを作成する場合は、これらのエントリーを 1 つ以上の *cso.application* 型のエントリーの前に置きます。この場合の目的は、複数のプログラム名を 1 つのアプリケーション名で代用することです。コード後半部分に記述する *cso.serverLinkage* エントリーでは、*programName* の代わりにアプリケーション名を使用します。こうすると、前述の *cso.serverLinkage* エントリーによって、Java ランタイムで複数のプログラムへの呼び出しが処理されます。

cso.application エントリーのフォーマットは次のとおりです。

```
cso.application.wildProgramName.appName
```

wildProgramName

有効なプログラム名、アスタリスク、または有効なプログラム名の先頭の数文字とそれに続くアスタリスク。アスタリスクは、1 つ以上の文字に相当するワイルドカードであり、一連の名前を指定できます。

wildProgramName が指しているものが EGL 生成のプログラムの場合、*wildProgramName* に指定されているプログラム名は、プログラム・パーツ名になります。

appName

EGL 命名規則に適合する一連の文字。*appName* の値は、コードの後半部分に記述する *cso.serverLinkage* エントリーに使用します。

次に示す例では、アスタリスクをワイルドカード文字として使用しています。この例での `cso.serverLinkage` エントリーによる処理対象は、名前が `XYZ` で始まるプログラムの呼び出しです。

```
cso.application.Xyz*=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

次に示す例では、複数のプログラムの先頭の数文字が同じではない場合でも、同じ `cso.serverLinkage` エントリーを使用して複数のプログラムの呼び出しを処理しています。

```
cso.application.Abc=myApp
cso.application.Def=myApp
cso.application.Xyz=myApp
cso.serverLinkage.myApp.type=remoteCall
cso.serverLinkage.myApp.remoteComType=TCPIP
cso.serverLinkage.myApp.remotePgmType=EGL
cso.serverLinkage.myApp.externalName=xxx
cso.serverLinkage.myApp.package=xxx
cso.serverLinkage.myApp.conversionTable=xxx
cso.serverLinkage.myApp.location=xxx
cso.serverLinkage.myApp.serverID=xxx
cso.serverLinkage.myApp.parmForm=COMMDATA
cso.serverLinkage.myApp.luwControl=CLIENT
```

1 つのプログラムに対して複数の `cso.application` エントリーが有効な場合、EGL では、適用される最初のエントリーが使用されます。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

396 ページの『リンケージ・プロパティ・ファイル』

関連するタスク

341 ページの『リンケージ・オプション・パーツの `callLink` エレメントの編集』

384 ページの『EGL 生成コード用の J2EE ランタイム環境のセットアップ』

関連する参照項目

442 ページの『`callLink` エレメント』

100 ページの『例外処理』

584 ページの『Java ランタイム・プロパティ (詳細)』

725 ページの『命名規則』

matches 演算子

論理式の中で、ストリング式を別の (*match* 基準 と呼ばれる) ストリングと、文字位置ごとに左から右へ比較することができます。この機能の用途は、UNIX または Perl における *正規表現* の用途によく似ています。

以下に例を示します。

```
// 変数 myVar01 はストリング式である
// その内容が match 基準と比較される
myVar01 = "abcdef";

// 次の論理式は「真」へと評価される
if (myVar01 matches "a?c*")
;
end
```

match 基準は、リテラル、CHAR 型または MBCHAR 型の項目、UNICODE 型の項目のいずれかにすることができます。 *match* 基準の中には、以下の文字を組み込むことができます。

- * ワイルドカードとして機能し、ストリング式内のゼロ個以上の任意の文字に一致します。
- ? ワイルドカードとして機能し、ストリング式内の単一の文字に一致します。
- [] 区切り文字として機能し、2 つの大括弧の中にある文字のいずれか 1 つが、ストリング式内の次の文字の一致として有効となります。例えば、次の *match* 基準のコンポーネントは、a、b、または c が一致として有効であることを示します。

[abc]

- 大括弧の区切り文字の中で範囲を作成し、その範囲内にあるいずれか 1 文字がストリング式内の次の文字の一致として有効となります。例えば、次の *match* 基準のコンポーネントは、a、b、または c が一致として有効であることを示します。

[a-c]

ハイフン (-) は、大括弧の区切り文字の外部では特別な意味を持ちません。

- ^ ワイルドカードの規則を作成し、キャレット (^) が大括弧の区切り文字内で最初の文字ならば、区切り文字以外の任意の文字が、ストリング式内の次の文字の一致として有効となります。例えば、次の *match* 基準のコンポーネントは、a、b、c を除く任意の文字が一致として有効であることを示します。

[^abc]

キャレットは、次の場合には特別な意味を持ちません。

- 大括弧の区切り文字の外部にある場合
- 大括弧の区切り文字の内部にあっても、先頭の文字でない場合

- ¥ 次の文字をストリング式内の単一の文字と比較するよう指示します。円記号 (¥) は、通常の処理から外れるので、*エスケープ文字* と呼ばれます。このエスケープ文字はストリング式内の任意の文字とも比較されません。

通常、このエスケープ文字は、例えばアスタリスク (*) や疑問符 (?) など、エスケープ文字がないと `match` 基準内で別の意味になる文字の前に置かれます。 .

円記号を (デフォルトの振る舞いどおりに) エスケープ文字として使用した場合、EGL は同じエスケープ文字を使用して任意のテキスト内への引用符の組み込みを許可するので、問題が起きます。 `match` 基準の内容においては、2 つの円記号を指定する必要があります。実行時に使用可能なテキストは、最初の円記号を除いたテキストだからです。

この問題を回避することをお勧めします。後で例を示すように、`escape` 文節を使用することにより、別の文字をエスケープ文字として指定してください。ただし、二重引用符 (") をエスケープ文字として使用することはできません。

それ以外の `matchCriterion` 内の文字は、すべてリテラルであり、ストリング式 内の単一の文字と比較されます。

次の例は、`escape` 文節の使用方法を示しています。

```
// 変数 myVar01 はストリング式である
// その内容が match 基準と比較される
myVar01 = "ab*def";

// 次の論理式は「真」へと評価される
if (myVar01 matches "ab¥¥*[abcd][abcde][^a-e]")
;
end

// 次の論理式は「真」へと評価される
if (myVar01 matches "ab+*def" escape "+")
;
end
```

関連する参照項目

93 ページの『EGL ステートメント』

538 ページの『論理式』

547 ページの『テキスト式』

EGL Java ランタイム用のメッセージのカスタマイズ

Java ランタイムにエラーが発生すると、デフォルトで EGL システム・メッセージが表示されます。しかし、これらのシステム・メッセージごとに、またはサブセットに対して、カスタマイズされたメッセージを指定することができます。

メッセージが必要な場合、EGL はまず、Java ランタイム・プロパティー `vgj.messages.file` で識別するプロパティー・ファイルを検索します。参照されるファイルの形式は、『プログラム・プロパティー・ファイル』および現行のトピックに説明されているように、任意の Java プロパティー・ファイルと同じです。

多くの場合、システム・メッセージには、EGL が実行時に検索するメッセージ挿入用のプレースホルダーが含まれています。たとえば、ご使用のコードがシステム関数に無効な日付マスクを提示する場合、メッセージには 2 つのプレースホルダーがあります。1 つは、日付マスク自体用のプレースホルダー 0 であり、もう 1 つ

は、システム関数の名前用のプレースホルダー 1 です。プロパティー・ファイル形式では、デフォルト・メッセージのエントリは次のとおりです。

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

メッセージの語句を変更して、すべてまたは一部のプレースホルダーを任意の順序で組み込むことができますが、プレースホルダーを追加することはできません。有効な例は、次のとおりです。

```
VGJ0216E = Function {1} was given invalid date mask {0}.
```

```
VGJ0216E = Function {1} was given an invalid date mask.
```

プロパティー `vgj.messages.file` で識別されるファイルを開けない場合は、致命的エラーが発生します。

メッセージ番号とその意味の詳細については、『*EGL Java ランタイム・エラー・コード*』を参照してください。

その他の詳細情報は、Java 言語資料で入手できます。

- メッセージの処理方法、および有効なコンテンツに関する詳細については、Java クラス `java.text.MessageFormat` の資料を参照してください。
- (常にプロパティー・ファイルで使用される) ISO 8859-1 文字エンコードで直接表すことができない文字の処理に関する詳細については、Java クラス `java.util.Properties` の資料を参照してください。

関連する概念

380 ページの『プログラム・プロパティー・ファイル』

関連する参照項目

1027 ページの『*EGL Java ランタイム・エラー・コード*』

584 ページの『Java ランタイム・プロパティー (詳細)』

EGL ソース形式の MQ レコード・パーツ

MQ レコード・パーツは、EGL ソース・ファイル内で宣言できます。このファイルの概要については、『*EGL ソース形式*』を参照してください。EGL と MQSeries の対話方法については、『*MQSeries サポート*』を参照してください。

MQ レコード・パーツの例を次に示します。

```
Record myMQRecordPart type mqRecord
{
    queueName = "myQueue"
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

MQ レコード・パーツの構文図は、以下のとおりです。



Record recordPartName mqRecord

パーツをタイプ **mqRecord** として識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

queueName = "msgQueueName"

メッセージ・キュー名を指定します。通常これは、論理キューの名前であり、物理キューの名前ではありません。入力の形式については、『MQ レコードのプロパティー』を参照してください。

getOptionsRecord = "getRecordName"

get オプション・レコードとして使用されるプログラム変数 (基本レコード) を識別します。詳細については、『MQ レコード用のオプション・レコード』を参照してください。このプロパティーは、以前の **getOptions** プロパティーです。

putOptionsRecord = "putRecordName"

put オプション・レコードとして使用されるプログラム変数 (基本レコード) を識別します。詳細については、『MQ レコード用のオプション・レコード』を参照してください。このプロパティーは、以前の **putOptions** プロパティーです。

openOptionsRecord = "openRecordName"

open オプション・レコードとして使用されるプログラム変数 (基本レコード) を識別します。詳細については、『MQ レコード用のオプション・レコード』を参照してください。このプロパティーは、以前の **openOptions** プロパティーです。

msgDescriptorRecord = "msgDRecordName"

メッセージ記述子として使用されるプログラム変数 (基本レコード) を識別します。詳細については、『MQ レコード用のオプション・レコード』を参照してください。このプロパティは、以前の **msgDescriptor** プロパティです。

queueDescriptorRecord = "QDRecordName"

キュー記述子として使用されるプログラム変数 (基本レコード) を識別します。詳細については、『MQ レコード用のオプション・レコード』を参照してください。このプロパティは、以前の **queueDescriptor** プロパティです。

includeMsgInTransaction = yes、**includeMsgInTransaction** = no

このプロパティを *yes* (デフォルト) に設定すると、レコード固有のメッセージがそれぞれトランザクションに組み込まれ、自分のコードでそのトランザクションをコミットしたり、ロールバックしたりできるようになります。選択の意味については、『MQSeries のサポート』を参照してください。

openQueueExclusive = no、**openQueueExclusive** = yes

このプロパティを *yes* に設定すると、コードは、メッセージ・キューから排他的に読み取りを行うことができます。このプロパティを *yes* に設定しないと、他のプログラムもこのメッセージ・キューから読み取りを行うことができます。デフォルトは *no* です。このプロパティは、MQSeries のオプション MQOO_INPUT_EXCLUSIVE に相当します。

lengthItem = "lengthField"

長さフィールド。詳細については、『MQ レコードのプロパティ』を参照してください。

numElementsItem = "numElementsField"

要素フィールドの数。詳細については、『MQ レコードのプロパティ』を参照してください。

structureField

構造体フィールド。詳細については、『EGL ソース形式の構造体項目』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 24 ページの『パーツの参照』
- 285 ページの『MQSeries のサポート』
- 19 ページの『パーツ』
- 141 ページの『レコード・パーツ』
- 62 ページの『EGL での変数の参照』
- 30 ページの『Typedef』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 78 ページの『配列』
- 512 ページの『EGL ソース形式の DataItem パーツ』
- 532 ページの『EGL ソース形式』
- 570 ページの『EGL ソース形式の関数パーツ』
- 578 ページの『EGL ソース形式の索引付きレコード・パーツ』

『MQ レコードのプロパティ』
725 ページの『命名規則』
718 ページの『MQ レコード用のオプション・レコード』
37 ページの『プリミティブ型』
783 ページの『EGL ソース形式のプログラム・パーツ』
796 ページの『EGL ソース形式の相対レコード・パーツ』
799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』
808 ページの『EGL ソース形式の構造体フィールド』

MQ レコードのプロパティ

このページでは、次の MQ レコード・プロパティについて説明します。

- キュー名
- トランザクションにメッセージを組み込む
- 入力キューを排他使用で開く

その他のプロパティについては、次のページを参照してください。

- MQ レコード用のオプション・レコード
- 可変長レコードをサポートするプロパティ

キュー名

Queue name は必須プロパティであり、論理的なキュー名を意味します。キュー名は、8 文字以内で指定します。入力の意味については、『*MQSeries 関連の EGL キーワード*』を参照してください。

トランザクションにメッセージを組み込む

Include message in transaction を指定した場合、トランザクション内にレコード固有の個々のメッセージを埋め込み、コードで該当のトランザクションをコミットまたはロールバックできます。

選択の意味については、『*MQSeries のサポート*』を参照してください。

入力キューを排他使用で開く

Open input queue for exclusive use を指定した場合、コードはメッセージ・キューから読み取りを排他使用で行うことができます。指定しない場合、ほかのプログラムも同じキューから読み取りを行うことがあります。このプロパティは、MQSeries のオプション `MQOO_INPUT_EXCLUSIVE` に相当します。

関連する概念

- 289 ページの『MQSeries 関連の EGL キーワード』
- 285 ページの『MQSeries のサポート』
- 143 ページの『レコード・タイプとプロパティ』

関連する参照項目

- 718 ページの『MQ レコード用のオプション・レコード』
- 793 ページの『可変長レコードをサポートするプロパティ』

MQ レコード用のオプション・レコード

各 MQ レコードは、5 つのオプション・レコード が関連付けられています。これらのレコードは、MQSeries への隠れた呼び出しの引数として、EGL で使用されます。

- get オプション用レコード (MQGMO)
- put オプション用レコード (MQPMO)
- open オプション用レコード (MQOO: 1 つの構造体の項目を持つレコード)
- メッセージ記述子レコード (MQMD)
- キュー記述子レコード (MQOD)

MQ レコードのプロパティとしてオプション・レコードを指定すると、作業用ストレージ・レコード・パーツ (MQOD など) を `typeDef` として使用する変数を参照することになります。このパーツは、プロダクトで提供される EGL ファイルの中にあります (『MQSeries のサポート』を参照)。レコード・パーツは、そのまま使用することもできますが、ご自分の EGL ファイルにコピーして、カスタマイズすることもできます。

あるオプション・レコードについて、それが使用中であることを示さないと、EGL は、デフォルトのレコードを作成し、値を割り当てます。これについては、以降のセクションを参照してください。ただし、MQ レコードを使用しないで MQSeries にアクセスしたときは、デフォルトのオプション・レコードは提供されません。

get オプション用レコード

get オプション用レコードは、MQSeries MQGET 呼び出しの引数である、MQSeries Get Message Options (MQGMO) に基づいて作成できます。get オプション用レコードを宣言しない場合、EGL は自動的に MQGMO という名前のデフォルトのレコードを作成します。生成されたプログラムでは、以下のことを行います。

- 『データの初期化』の冒頭に記載した値を使用して、get オプション用レコードを初期化する。
- MQ レコードのプロパティ *Include message in transaction* を設定したかどうかに応じて、OPTIONS を MQGMO_SYNCPOINT または MQGMO_NO_SYNCPOINT のいずれかに設定する。

put オプション用レコード

put オプション用レコードは、MQSeries MQPUT 呼び出しの引数である、MQSeries Put Message Options (MQPMO) に基づいて作成できます。put オプション用レコードを宣言しない場合、EGL は自動的に MQPMO という名前のデフォルトのレコードを作成します。生成されたプログラムでは、以下のことを行います。

- 『データの初期化』の冒頭に記載した値を使用して、put オプション用レコードを初期化する。
- MQ レコードのプロパティ *Include message in transaction* を設定したかどうかに応じて、OPTIONS を MQPMO_SYNCPOINT または MQPMO_NO_SYNCPOINT のいずれかに設定する。

open オプション用レコード

open オプション用レコードの内容によって、MQSeries のコマンド MQOPEN または MQCLOSE の呼び出しに使用する Options パラメーターの値が決まります。

open オプション用レコード・パーツ (MQOO) が使用可能ですが、このパーツに基づいてレコードを宣言しないと、EGL は、以下のようにして、MQOO という名前のデフォルトのレコードを自動的に作成します。

- EGL の add ステートメントによって呼び出された MQOPEN で、生成されたプログラムが MQOO.OPTIONS を次のように設定します。

```
MQOO_OUTPUT + MQOO_FAIL_IF QUIESCING
```

- メッセージ・キュー・レコードのプロパティのオプション *Open input queue for exclusive use* が有効な場合、EGL の scan ステートメントによって呼び出された MQOPEN で、生成されたプログラムが MQOO.OPTIONS を次のように設定します。

```
MQOO_INPUT_EXCLUSIVE + MQOO_FAIL_IF QUIESCING
```

- メッセージ・キュー・レコードのプロパティのオプション *Open input queue for exclusive use* が有効でない場合、EGL の scan ステートメントによって呼び出された MQOPEN で、生成されたプログラムが MQOO.OPTIONS を次のように設定します。

```
MQOO_INPUT_SHARED + MQOO_FAIL_IF QUIESCING
```

- EGL の close ステートメントによって呼び出された MQCLOSE で、生成されたプログラムは MQOO.OPTIONS を次のように設定します。

```
MQCO_NONE
```

メッセージ記述子レコード

メッセージ記述子レコードは、MQGET 呼び出しと MQPUT 呼び出しのパラメーターである、MQSeries Message Descriptor (MQMD) に基づいて作成できます。メッセージ記述子レコードを宣言しない場合、EGL は自動的に MQMD という名前のデフォルトのレコードを作成し、そのレコードを『データの初期化』に記載した値で初期化します。

キュー記述子レコード

キュー記述子レコードは、MQSeries の MQOPEN 呼び出しと MQCLOSE 呼び出しの引数である、MQSeries Object Descriptor (MQOD) に基づいて作成できます。キュー記述子レコードを宣言しない場合、EGL は自動的に MQOD という名前のデフォルトのレコードを作成し、生成されたプログラムは以下のことを行います。

- 『データの初期化』の冒頭に記載した値を使用して、キュー記述子レコードを初期化する。
- 該当レコード内の OBJECTTYPE を MQOT_Q に設定する。
- システム・ワード **record.resourceAssociation** で指定されるキュー・マネージャー名を OBJECTMGRNAME に設定する。ただし、**record.resourceAssociation** がキュー・マネージャー名を参照しない場合、OBJECTQMGRNAME は値を持ちません。
- **record.resourceAssociation** 内のキュー名を OBJECTNAME に設定する

関連する概念

292 ページの『直接 MQSeries 呼び出し』

289 ページの『MQSeries 関連の EGL キーワード』

285 ページの『MQSeries のサポート』

関連する参照項目

- 511 ページの『データの初期化』
- 920 ページの『recordName.resourceAssociation』
- 717 ページの『MQ レコードのプロパティ』

別名の割り当て

Java 出力で無効な名前を使用すると、生成プログラムによって名前の別名が作成され、生成コードで使用されます。

- 許可される ID 文字の違い
- 長さ制限の違い
- 大文字と小文字に対するサポートの違い
- 生成言語において予約語であるワードの使用
- 名前の別名構文と一致するワードの使用 (例えば、**class\$** には別名が割り当てられますが、これは **class\$** が Java 生成の **class** の別名である ためです。)

別名の生成には、無効文字を有効な文字セットで置換する方法、長すぎる名前を切り捨てる方法、名前に接頭部または接尾部を追加する方法、または完全に異なる名前 (**EZE00123** など) を作成する方法があります。

関連する概念

関連するタスク

- 146 ページの『EGL プログラム・パーツの作成』

関連する参照項目

- 721 ページの『Java 名の別名の割り当て方法』
- 722 ページの『Java ラッパーの別名の割り当て方法』
- 725 ページの『命名規則』

JSP ファイル内の EGL ID の変更と Java Bean の生成

ページ・ハンドラーの関数、レコード、および項目には、『命名規則』で詳しく説明されている規則に従って名前を割り当てます。ただし、EGL は、JSP ファイル内、およびページ・ハンドラーから派生する Java Bean 内に Java ID を作成するときは、それらの名前のバリエーションを使用します。ソース・タブを使用して JSP ファイルを編集する場合、プロパティ・ビューを使用する場合、または EGL 対応のツールの外部で作業する場合は、それらのバリエーションを知っておく必要があります。

バリエーションは、以下のとおりです。

- ページ・ハンドラーのレコード、項目、および関数の名前の前に、英字の **EGL** が付きます。このバリエーションの目的は、Java Bean の仕様と EGL の命名規則の違いによって、Java ランタイム環境で起きる可能性があるエラーからユーザーを保護することです。
- いくつかの状況では、特定の種類の出力コントロールへバインドされた変数の名前に接尾部が追加されます。

- 項目を「ブール値 (Boolean)」チェック・ボックスにバインドした場合、Java ID に接尾部 *AsBoolean* が組み込まれます。
- 項目を選択コントロール (リスト・ボックス、コンボ・ボックス、ラジオ・ボタン・グループ、またはチェック・ボックス・グループ) にバインドし、JavaServer Faces の *selectItems* タグでその項目を参照した場合、Java ID に接尾部 *AsSelectItemsList* が組み込まれます。
- 項目を JavaServer Faces のデータ・テーブル内でチェック・ボックスにバインドした場合 (特に、その項目が *inputRowSelect* タグの中で参照されている場合)、Java ID に接尾部 *AsIntegerArray* が組み込まれます。

上記のバリエーションを除くと、EGL はページ・ハンドラー内の名前に完全に一致する ID を作成しようとします。

変数 *myItem* を含んでいる、ページ・ハンドラー *myJSP* について考慮してください。この変数を「ブール値 (Boolean)」チェック・ボックスにバインドすると、JSP ファイルは Java Bean プロパティ *myJSP.EGLmyItemAsBoolean* を参照し、Java Bean の getter および setter 関数には、次のような名前が付きます。

- *getEGLmyItemAsBoolean*
- *setEGLmyItemAsBoolean*

JSP ファイル内の「ブール値 (Boolean)」チェック・ボックス・タグのソースは、次のとおりです。

```
<h:selectBooleanCheckbox styleClass="selectBooleanCheckbox"
  id="checkbox1" value="#{myJSP.EGLmyItemAsBoolean}">
</h:selectBooleanCheckbox>
```

EGL は、Java で無効となるような名前の生成を回避します。詳細については、『Java 名の別名の割り当て方法』を参照してください。

関連する概念

207 ページの『ページ・ハンドラー』

関連するタスク

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

214 ページの『EGL レコードと Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

『Java 名の別名の割り当て方法』

725 ページの『命名規則』

205 ページの『EGL の Page Designer サポート』

Java 名の別名の割り当て方法

パーツに名前を付ける場合、その名前は有効な Java ID である必要がありますが、パーツ名にはハイフンまたは負符号 (-) を使用できます。ただし、ハイフンをパーツ名の先頭文字にすることはできません。

Java キーワードである名前またはドル記号 (\$) やハイフンや負符号を含む名前を選択すると、そのパーツ名は、生成される出力での名前と一致しくなくなります。Java

キーワードである各パーツ名には、別名割り当てメカニズムによって、自動的にドル記号が付加されます。1 つ以上のドル記号またはハイフンを含む名前を指定すると、別名割り当てメカニズムによって、各記号が以下のようなユニコード値に置換されます。

```
$ $0024
- $002d
```

例えば、**class** という項目には **class\$** という別名が割り当てられ、**class\$** という項目には **class\$0024** という別名が割り当てられます。

パーツ名の宣言に使用した大文字小文字は保存されます。プログラム XYZ および xyz は、それぞれ XYZ.java および xyz.java に生成されます。Windows 2000/NT/XP では、大/小文字のみが違う名前と同じディレクトリー・パーツ内に生成すると、古いファイルが上書きされます。

EGL パッケージ名は、常に小文字の Java パッケージ名に変換されます。

最後に、プログラム、ページ・ハンドラー、またはライブラリーの名前が Java システム・パッケージ java.lang の 1 つのクラスの名前に一致する場合は、このクラス名にドル記号が付加されます。例えば、Object は Object\$、Error は Error\$ のようになります。

EGL が、JSP ファイル内、およびページ・ハンドラーから派生する Java bean 内に Java ID を作成する方法の詳細については、『JSP ファイルおよび生成された Java beans 内の EGL ID の変更』を参照してください。

関連する概念

720 ページの『別名の割り当て』

関連する参照項目

720 ページの『JSP ファイル内の EGL ID の変更と Java Bean の生成』

Java ラッパーの別名の割り当て方法

EGL 生成プログラムは、Java ラッパーの名前に別名を割り当てる際に以下の規則を適用します。

1. EGL の名前がすべて大文字の場合は、EGL 名を小文字に変換する。
2. 名前がクラス名またはメソッド名の場合は、先頭の文字を大文字にする。(例えば、**x** の getter メソッドは **getX()** であって、**getx()** ではありません。)
3. アンダースコア (_) およびハイフン (-) をすべて削除する。(VisualAge Generator との互換性モードを使用する場合は、EGL 名のハイフンは有効です。) アンダースコアまたはハイフンの後に文字が続く場合は、その文字を大文字に変える。
4. 名前が、区切り文字としてピリオド (.) を使用する修飾名である場合は、すべてのピリオドをアンダースコアで置き換え、名前の先頭にアンダースコアを追加する。
5. 名前にドル記号 (\$) が含まれる場合は、ドル記号を 2 つのアンダースコアで置き換え、名前の先頭にアンダースコアを追加する。
6. 名前が Java キーワードである場合は、名前の先頭にアンダースコアを追加する。

7. 名前が * (アスタリスク。充てん文字項目を表す) である場合は、先頭のアスタリスクを **Filler1**、2 番目のアスタリスクを **Filler2**、(以下同様...) にリネームする。

また、プログラム・ラッパー、レコード・ラッパー、および副構造配列項目の Java ラッパー・クラス名には特別な規則が適用されます。以降の項では、これらの規則について説明し、例を 1 つ示します。一般に、生成されるラッパー・クラスのフィールドで名前が競合する場合は、クラス名と変数名を区別するために修飾名が使用されます。それでも競合が解決されない場合は、生成時に例外がスローされます。

プログラム・ラッパー・クラス

レコード・パラメーター・ラッパーには、型定義名に適用される上記の規則を使用して名前が付けられる。レコード・ラッパー・クラス名がプログラム・クラス名またはプログラム・ラッパー・クラス名と競合する場合は、レコード・ラッパー・クラス名の最後に **Record** が追加される。

変数名の規則は以下のとおりです。

1. レコード・パラメーター変数には、パラメーター名に適用される上記の規則を使用して名前が付けられる。したがって、**get()** メソッドと **set()** メソッドには、クラス名ではなくこれらの名前が組み込まれる。
2. **get** メソッドおよび **set** メソッドの名前は、**get** または **set** の後に、上記の規則が適用されたパラメーター名を続けたものである。

レコードのラッパー・クラス

副構造配列項目クラス名の規則は以下のとおりです。

1. 副構造配列項目は、レコードのラッパー・クラスのインナー・クラスになり、クラス名は項目名に対する上記の規則を適用することによって導き出される。このクラス名が、組み込まれているレコードのクラス名と競合する場合は、項目クラス名に **Structure** が付加される。
2. 項目クラス名が互いに競合する場合は、修飾項目名が使用される。

get メソッド名および **set** メソッド名の規則は以下のとおりです。

1. メソッドの名前は、**get** または **set** の後に、上記の規則が適用された項目名を続けたものである。
2. 項目名が互いに競合する場合は、修飾項目名が使用される。

副構造配列項目クラス

副構造配列項目クラス名の規則は以下のとおりです。

1. 副構造配列項目は、組み込まれている副構造配列項目のために生成されるラッパー・クラスのインナー・クラスになり、クラス名は項目名に対する上記の規則を適用することによって導き出される。
2. このクラス名が、組み込まれている副構造配列項目のクラス名と競合する場合は、項目クラス名に **Structure** が付加される。

get メソッド名および **set** メソッド名の規則は以下のとおりです。

1. メソッドの名前は、**get** または **set** の後に、上記の規則が適用された項目名を続けたものである。

2. 項目名が互いに競合する場合は、修飾項目名が使用される。

例

以下のサンプル・プログラムおよび生成される出力には、ラッパー生成中に予期されるものが示されています。

サンプル・プログラム:

```
Program WrapperAlias(param1 RecordA)

end

Record RecordA type basicRecord
  10 itemA CHAR(10)[1];
  10 item_b CHAR(10)[1];
  10 item$C CHAR(10)[1];
  10 static CHAR(10)[1];
  10 itemC CHAR(20)[1];
  15 item CHAR(10)[1];
  15 itemD CHAR(10)[1];
  10 arrayItem CHAR(20)[5];
  15 innerItem1 CHAR(10)[1];
  15 innerItem2 CHAR(10)[1];
end
```

生成される出力:

生成される出力の名前

出力	名前
プログラム・ラッパー・クラス	WrapperaliasWrapper 。レコード・ラッパー・クラス RecordA のインスタンスである フィールド param1 が含まれています。
パラメーター・ラッパー・クラス	RecordA 。次のメソッドからアクセスできます。 <ul style="list-style-type: none">• getItemA (itemA から)• getItemB (先頭の item-b から)• get_Item__C (item\$C から)• get_Static (static から)• get_ItemC_itemB (itemC 内の itemB から)• getItemD (itemD から)• getArrayItem (arrayItem から) ArrayItem は、 getInnerItem1 および getInnerItem2 を通じてアクセス可能なフィールドを含む RecordA のインナー・クラスです。

関連する概念

- 477 ページの『VisualAge Generator との互換性』
- 327 ページの『Java ラッパー』
- 720 ページの『別名の割り当て』

関連するタスク

- 327 ページの『Java ラッパーの生成』

関連する参照項目

- 595 ページの『Java ラッパー・クラス』

命名規則

このページでは、パーツや変数を命名する際の規則、およびファイル名などのプロパティに値を割り当てる際の規則について説明します。ロジック・パーツのメモリー領域の参照方法については、『変数および定数の参照』および『配列』を参照してください。

ID の 3 つのカテゴリは EGL にあります。

- 以降で説明する EGL パーツ名および変数名
- パーツ宣言または変数宣言のプロパティ値として指定される外部リソース名。
これらの名前は特殊ケースを表し、命名規則はランタイム・システムの規則に従います。
- EGL パッケージ名 (例: com.mycom.mypack)。このケースでは、各文字シーケンスがピリオドで区切られ、各シーケンスの名前は EGL パーツ名の命名規則に従っています。パッケージ名とファイル構造の関係について詳しくは、『EGL プロジェクト、パッケージ、およびファイル』を参照してください。

EGL パーツ名または変数名は 1 から 128 字の連続する文字です。特に断りがない限り、名前は、ユニコード文字またはアンダースコアで始める必要があります。名前には、追加のユニコード文字、数字、および通貨記号を含めることができます。そのほかに、以下のような制限事項があります。

- 先頭文字を、大文字と小文字のいずれの組み合わせでも EZE にすることはできません。
- 名前にブランクを埋め込んだり、EGL 予約語を名前に指定したりすることはできません。

パーツには、次のような特殊な考慮事項が適用されます。

- レコード・パーツでは、論理ファイルまたはキューの名前が 8 文字を超えてはなりません。
- さまざまなパーツで、生成された出力ファイルおよび Java クラスの名前に別名が組み込まれます。外部名を指定しない場合は、プログラム・パーツ名が使用されます。ただし、プログラム・パーツ名は、ランタイム環境で許される最大文字数に (必要に応じて) 切り捨てられます。

コードが VisualAge Generator と互換性がある場合、パーツ名および変数名には以下の規則も適用されます。ただし、パッケージ名には適用されません。

- 名前の先頭の文字はアットマーク (@)
- 後続の文字に使用できるのは、アットマーク (@)、ハイフン (-)、およびポンド記号 (#)

関連する概念

477 ページの『VisualAge Generator との互換性』

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

720 ページの『別名の割り当て』
62 ページの『EGL での変数の参照』

関連する参照項目

78 ページの『配列』
720 ページの『JSP ファイル内の EGL ID の変更と Java Bean の生成』
527 ページの『EGL 予約語』
535 ページの『EGL システム制限』

演算子と優先順位

次の表は、EGL 演算子を降順にリストしたものです。単項のプラス (+)、マイナス (-)、および not (!) を除き、各演算子では 2 つのオペランドを使用します。

演算子 (コンマ区切り)	演算子の型	意味
+, -	数値、単項	単項のプラス (+) やマイナス (-) は、オペランドや括弧で囲まれた式の前に付く符号であり、2 つの式の間に使用される演算子ではありません。
**	数値	** は <i>toThePowerOfInteger</i> 演算子で、指定された累乗の数値を受け入れます。例えば、 <code>c = a**b</code> と指定すると、 <code>c</code> に <code>(a^b)</code> の値が代入されます。第 1 オペランド (上の例では <code>a</code>) を負の値にすることはできません。第 2 オペランド (上の例では <code>b</code>) は整数または精度 0 の数値フィールドであることが必要です。第 2 オペランドは、正、負、ゼロのいずれでもかまいません。
, /, %	数値	乗算 () と整数除算 (/) は、等しい優先順位を持ちます。整数の除算では、小数値があれば、その値も保持されます。例えば、 <code>7/5</code> は <code>1.4</code> になります。 % は、剰余 演算子です。この演算子は、前者のオペランドまたは数式を後者で割ったときのモジュラスに解決されます。例えば、 <code>7%5</code> は <code>2</code> になります。
+, -	数値	加算 (+) と減算 (-) は、同じ優先順位を持ちます。
=	数値または ストリング	= は、代入 演算子で、式やオペランドからオペランドに数値や文字値をコピーします。
!	論理、単項	! は、not 演算子で、直後の論理式の値を反転したブール値 (真または偽) に解決されます。この演算子の後ろにくる式は、括弧で囲む必要があります。
==, !=, <, >, <=, >=, in, is, not	論理 (比較 用)	比較のために用いられる論理演算子は、いずれも等しい優先順位を持ちます。これらの演算子は、論理式のページで説明されています。各演算子は真または偽に解決されます。

演算子 (コンマ区切り)	演算子の型	意味
&&	論理	&& は、 <i>and</i> 演算子で、「両方が真でなければならない」ことを意味します。この演算子は、演算子の前にある論理式が真で、演算子の後ろにある論理式も真である場合に、真に解決されます。それ以外の場合、&& は偽に解決されます。
	論理	は、 <i>or</i> 演算子で、「どちらか一方、または両方」を意味します。この演算子は、演算子の前にある論理式が真であるか、演算子の後ろにある論理式が真であるか、あるいはその両方が真である場合に、真に解決されます。それ以外の場合、 は偽に解決されます。

通常の優先順位 (演算の順番 と呼ばれる) は、括弧を使用してある式を他の式から分離することによってオーバーライドできます。式内に同じ優先順位を持つ演算子がある演算では、左から右の順番で実行されます。

関連する参照項目

- 575 ページの『in 演算子』
- 538 ページの『論理式』
- 546 ページの『数式』
- 37 ページの『プリミティブ型』
- 547 ページの『テキスト式』

Java プログラム生成の出力

Java サーバー・プログラム生成の出力は、以下のとおりです。

- ビルド記述子オプション **genProject** が省略されている場合は、ビルド計画
- Java ソース・コード (『Java プログラム、ページ・ハンドラー、およびライブラリー』を参照)
- プログラムの作成と実行に必要な関連オブジェクト (『Java プログラム、ページ・ハンドラー、およびライブラリー』を参照)
- J2EE 環境ファイル
- プログラム・プロパティ・ファイル
- **genProject** が省略されている場合は、結果ファイル

EGL 生成プログラムを使用して、Java プログラム全体を生成することができます。プログラムとレコードは、個別の Java クラスとして生成されます。関数は、プログラム内のメソッドとして生成されます。データ項目と構造体項目は、それぞれが属するレコードまたはプログラムのクラスのフィールドとして生成されます。

以下の表に、生成される Java パーツのさまざまな型の名前を示します。

生成される Java パーツ名

パーツ型と名前	生成されるもの
P という名前のプログラム	P.java 内の P というクラス

生成される Java パーツ名

パーツ型と名前	生成されるもの
プログラム P 内の F という関数	P.java 内の \$funcF という P クラスのメソッド
R という名前のレコード	EzeR.java 内の EzeR というクラス
R という名前の基本レコード、関数 F へのパラメーター	Eze\$paramR.java 内の Eze\$paramR というクラス
L という名前のリンケージ・オプション・パーツ	L.properties という名前のリンケージ・プロパティ・ファイル
Lib という名前のライブラリー	Lib.java 内の Lib という名前のクラス
DT という名前の DataTable	EzeDT.java 内の EzeDT という名前のクラス
F という名前の書式	EzeF.java 内の EzeF という名前のクラス
FG という名前の FormGroup	FG.java 内の FG という名前のクラス

1. 指示されたパーツ型の場合は、複数のパーツが同じ名前で存在することも可能です。その場合、2 番目のパーツ名には追加の接尾部 \$v2 が付けられます。3 番目のパーツ名には \$v3 という接尾部、4 番目のパーツ名には \$v4 という接尾部が付けられます。

命名の形式によって同じ名前が 2 つできる場合は、EGL によって、2 番目以降に生成されたファイルに接尾部が追加されます。接尾部は以下のとおりです。

\$vn

ここで、

n 2 から始まって順番に割り当てられる整数です。

関連する概念

356 ページの『ビルド計画』

388 ページの『J2EE 環境ファイル』

357 ページの『Java プログラム、ページ・ハンドラー、およびライブラリー』

396 ページの『リンケージ・プロパティ・ファイル』

380 ページの『プログラム・プロパティ・ファイル』

357 ページの『結果ファイル』

関連する参照項目

442 ページの『callLink エlement』

Java ラッパー生成の出力

Java ラッパー生成の出力は、以下のとおりです。

- ビルド記述子オプション **genProject** が省略されている場合は、ビルド計画
- Java サーバー・プログラムへの呼び出しを折り返すための **JavaBeans™** (『Java ラッパー』を参照)。
- 特定の状況下での **EJB セッション Bean**。詳細については、『リンケージ・オプション・パーツ』の **callLink** エlementの説明を参照してください。
- **genProject** が省略されている場合は、結果ファイル

生成された Bean を使用して、サーブレット、EJB、または Java アプリケーションなどの非 EGL Java クラスからサーバー・プログラムへの呼び出しを折り返すことができます。以下のクラスの型が生成されます。

- サーバー用の Bean
- レコード・パラメーター用の Bean
- レコード配列行用の Bean

次の表に、生成される Java ラッパー・パーツのさまざまな型の名前を示します。

生成される Java ラッパー・パーツ名

パーツ型と名前	生成されるもの
P という名前のプログラム	PWrapper.java 内の PWrapper というクラス
パラメーターとして使用される R という名前のレコード	R.java 内の R というクラス
パラメーターとして使用されるレコード R 内の副構造領域 S	R.java 内の R.S というクラス
L という名前のリンケージ・オプション・パーツ	L.properties という名前のリンケージ・プロパティー・ファイル

1. 指示されたパーツ型の場合は、複数のパーツが同じ名前で存在することも可能です。その場合、2 番目のパーツ名には追加の接尾部 \$v2 が付けられます。3 番目のパーツ名には \$v3 という接尾部、4 番目のパーツ名には \$v4 という接尾部が付けられます。

プログラム・パーツを Java ラッパーとして生成するよう要求すると、EGL では、以下の実行可能プログラムについてそれぞれ Java クラスが作成されます。

- プログラム・パーツ
- プログラム・パラメーターとして宣言される各レコード
- リンケージ・オプション・パーツを指定し、生成されるプログラムの **callLink** 要素に **ejbCall** というリンク・タイプがある場合は、セッション Bean

また、レコードごとに生成されるクラスには、以下の特性を持つ各構造体項目のインナー・クラス (またはインナー・クラス内のクラス) が組み込まれます。

- それらのレコードのいずれかの内部構造体内にある構造体項目
- 少なくとも 1 つの従属構造体項目を持つ構造体項目 (つまり、副構造がある構造体項目)
- 配列である構造体項目 (この場合は、副構造配列)

生成される各クラスはファイルに格納されます。EGL 生成プログラムは、Java ラッパーで使用する名前を以下のように作成します。

- 名前は小文字に変換される。
- ハイフンや負符号 (-) やアンダースコア () はそれぞれ削除される。ハイフンまたはアンダースコアの後に続く文字は大文字に変換される。
- 名前がクラス名として使用されている場合やメソッド名の中で使用されている場合は、最初の文字を大文字に戻す。

プログラムのパラメーターの 1 つがレコードである場合、EGL では、その変数のラッパー・クラスも生成されます。プログラム Prog に Rec という型定義のレコード・パラメーターがある場合、そのパラメーターのラッパー・クラスは Rec と呼ばれます。パラメーターの型定義の名前がプログラムと同じ場合は、そのパラメーターのラッパー・クラスに、「Record」という接尾部が付けられます。

レコードのパラメーターに配列項目がありその項目の下に他の項目がある場合は、生成プログラムによってラッパーも生成されます。この副構造配列ラッパーは、レコード・ラッパーのインナー・クラスになります。ほとんどの場合、AItem in Rec という副構造配列項目は、Rec.AItem というクラスによってラップされます。同じ名前を持つ 2 つの副構造配列項目をレコードに含めることもできますが、その場合は、項目の修飾名を使用して項目ラッパーに名前が付けられます。最初の AItem の修飾名が Top1.AItem で 2 番目の修飾名が Top2.Middle2.AItem である場合、それらのクラスの名前は Rec.Top1\$_aItem および Rec.Top2\$_middle2\$_aItem になります。副構造配列の名前がプログラムの名前と同じである場合は、副構造配列のラッパー・クラスに Structure という接尾部が付けられます。

低レベル項目の値を設定および取得するメソッドは、各レコード・ラッパーおよび副構造配列ラッパーに生成されます。レコードまたは副構造配列内に同じ名前の低レベル項目が 2 つある場合は、生成プログラムにより、前の段落で説明した修飾名方式が使用されます。

その他のメソッドは、SQL レコード変数のラッパーに生成されます。レコード変数内の各項目に対して、生成プログラムは、NULL 標識値を取得および設定するメソッドとその SQL 長さ標識を取得および設定するメソッドを作成します。

クラスのコンパイルが済んだら、Javadoc ツールを使用して *classname.html* ファイルをビルドすることができます。HTML ファイルで、クラスのパブリック・インターフェースが説明されます。Javadoc によって作成された HTML ファイルを使用する場合は、それが EGL Java ラッパーであることを確認してください。

VisualAge Generator Java ラッパーから生成される HTML ファイルは、Java ラッパーから生成される HTML ファイルとは異なります。

例

副構造配列を持つレコード・パーツの例を次に示します。

```
Record myRecord type basicRecord
  10 MyTopStructure[3];
  15 MyStructureItem01 CHAR(3);
  15 MyStructureItem02 CHAR(3);
end
```

プログラム・パーツに関しては、出力ファイルの名前は以下のようになります。

aliasWrapper.java

ここで、

alias

プログラム・パーツで指定される別名です (存在する場合)。外部名が指定されていない場合はプログラム・パーツ名が使用されます。

プログラム・パラメーターとして宣言される各レコードに関しては、出力ファイルの名前は以下のようになります。

recordName.java

ここで、

recordName

レコード・パーツ名です。

副構造配列に関しては、インナー・クラスの名前と位置は、配列名がレコード内で固有であるかどうかによって変わります。

- 配列名がレコード内で固有である場合、インナー・クラスはレコード・クラス内に置かれ、以下のように名付けられます。

recordName.siName

ここで、

recordName

レコード・パーツ名です。

siName

配列の名前です。

- 配列名がレコード内で固有ではない場合、インナー・クラスの名前は配列の完全修飾名に基づいて付けられますが、ドル記号 (\$) とアンダースコア (_) の組み合わせで各修飾子が区切られます。例えば、配列がレコードの 3 次レベルである場合、生成されるクラスはレコード・クラスのインナー・クラスになり、以下のように命名されます。

Topname\$_Secondname\$_Siname

ここで、

Topname

トップレベル構造体項目の名前です。

Secondname

2 次レベル構造体項目の名前です。

Siname

副構造配列項目の名前です。

同じ名前を持つ別の配列がレコードの最高レベルの直属である場合は、インナー・クラスもレコード・クラス内に置かれ、以下のように命名されます。

Topname\$_Siname

ここで、

Topname

最高レベル構造体項目の名前です。

Siname

副構造配列項目の名前です。

最後に、以下のような場合を考えてみましょう。すなわち、レコード内で固有でない名前を持つある副構造配列が、レコード内で固有でない名前を持つ別の副構造配列に従属している場合です。この従属配列のクラスは、インナー・クラスのインナー・クラスとして生成されます。

リンケージ・オプションを実行時に設定するよう要求する場合は、Java ラッパーを生成するときに、Java プロパティー・ファイルとリンケージ・プロパティー・ファイルも生成します。

関連する概念

356 ページの『ビルド計画』
343 ページの『Enterprise JavaBean (EJB) セッション Bean』
327 ページの『Java ラッパー』
338 ページの『リンケージ・オプション・パーツ』
396 ページの『リンケージ・プロパティ・ファイル』
357 ページの『結果ファイル』

関連するタスク

327 ページの『Java ラッパーの生成』

関連する参照項目

442 ページの『callLink エlement』
595 ページの『Java ラッパー・クラス』

EGL ソース形式のページ・ハンドラー・パーツ

ページ・ハンドラーパーツは EGL ファイルで宣言します。これについては、『EGL プロジェクト、パッケージ、およびファイル』で説明しています。このパーツは生成可能なパーツです。つまり、このパーツは、ファイルの最上位にあり、ファイルと同じ名前を持つ必要があります。

ページ・ハンドラーパーツの例は以下のとおりです。

```
// Page designer requires that all pageHandlers
// be in a package named "pagehandlers".
package pagehandlers ;

PageHandler ListCustomers
{onPageLoadFunction="onPageLoad"}

// Library for customer table access
use CustomerLib3;

// List of customers
customerList Customer[] {maxSize=100};

Function onPageLoad()

// Starting key to retrieve customers
startkey CustomerId;

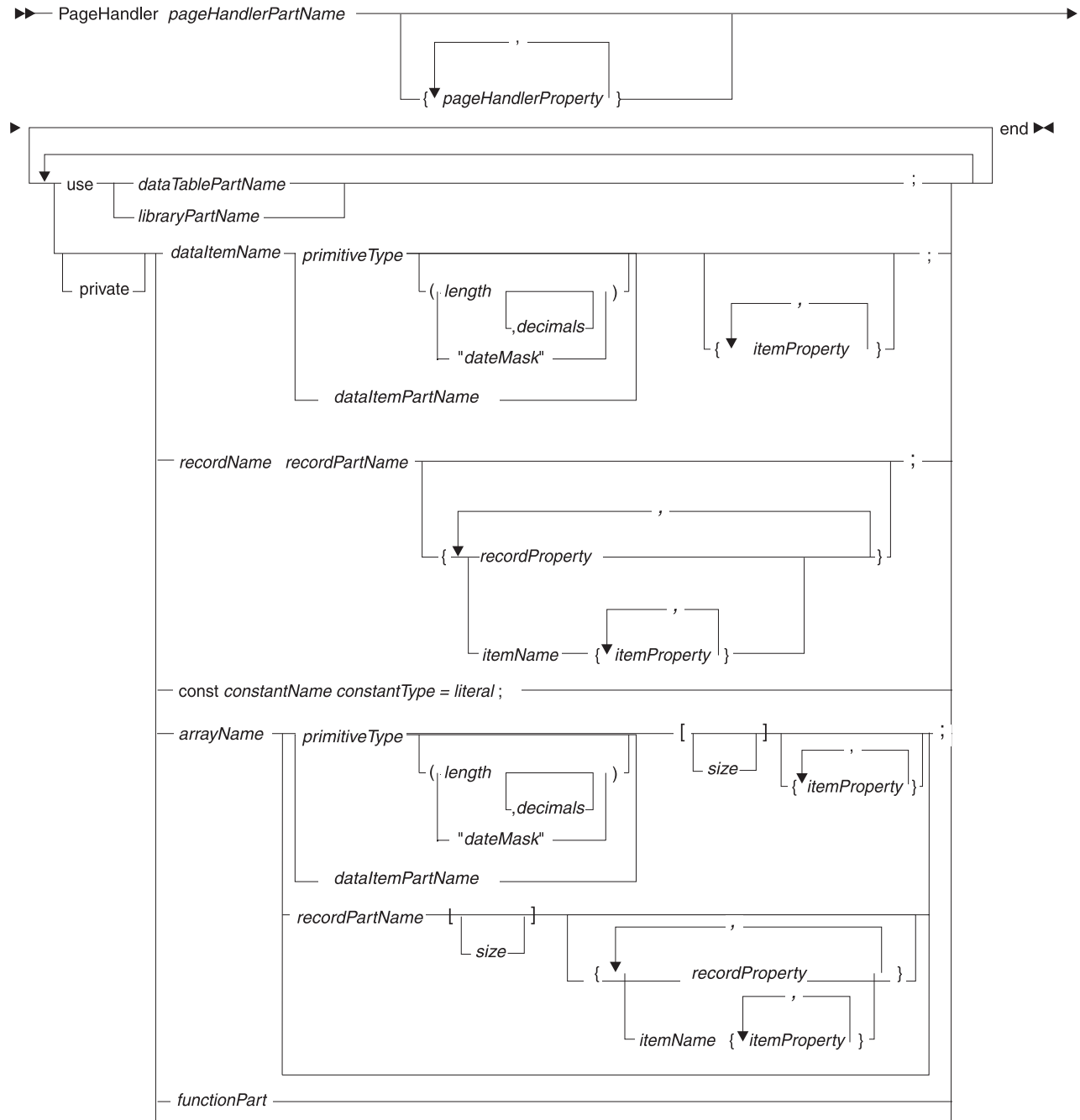
// Result from library call
status int;

// Retrieve up to 100 customer records
startKey = 0;
CustomerLib3.getCustomersByCustomerId(startKey,
    customerList, status);

    if ( status != 0 && status != 100 )
        setError("Retrieval of Customers Failed.");
    end
end

Function returnToIntroductionClicked()
    forward to "Introduction";
end
End
```

ページ・ハンドラーパーツのダイアグラムは、次のとおりです。



PageHandler pageHandlerPartName ... end

パーツをページ・ハンドラーとして識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

pageHandlerProperty

ページ・ハンドラー・パーツのプロパティにリストされているページ・ハンドラー・パーツのプロパティ。

use *dataTablePartName*, **use** *libraryPartName*

データ・テーブルまたはライブラリーのアクセスを簡略化する使用宣言。詳細については、『使用宣言』を参照してください。

private

Web ページを表示する JSP からは、変数、定数、または関数を使用できないことを示します。**private** という用語を省略すると、変数、定数、または関数を、Web ページ上のコントロールにバインドできます。

dataItemName

データ項目 (変数) の名前。命名の規則については、『命名規則』を参照してください。

primitiveType

データ項目に割り当てられるプリミティブ型。

length

構造体項目の長さ (整数)。構造体項目に基づくメモリー領域の値には、指定された数の文字または数字が含まれます。

decimals

decimals は、数値タイプ (BIN、DECIMAL、NUM、NUMC、または PACF) に対して指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

dataItemPartName

データ項目の形式のモデルになっている *dataItem* パーツ名。これについては、『*typeDef*』で説明しています。*dataItem* パーツは、ページ・ハンドラーパーツに対して可視になっている必要があります。これについては、『パーツの参照』で説明しています。

itemProperty

項目のプロパティ。詳細については、『ページ項目のプロパティ』を参照してください。

recordName

レコードの名前 (変数)。命名の規則については、『命名規則』を参照してください。

recordPartName

レコードの形式のモデルになっているレコード・パーツ名。これについては、『*typeDef*』で説明しています。レコード・パーツは、ページ・ハンドラーパーツに対して可視になっている必要があります。これについては、『パーツの参照』で説明しています。

recordProperty

レコード・プロパティのオーバーライド。レコード・プロパティについての詳細は、*recordPartName* のレコードのタイプに応じて、以下の説明のいずれかを参照してください。

- EGL ソース形式の基本レコード・パーツ
- EGL ソース形式の索引付きレコード・パーツ
- EGL ソース形式の MQ レコード・パーツ
- EGL ソース形式の相対レコード・パーツ

- EGL ソース形式のシリアル・レコード・パーツ
- EGL ソース形式の SQL レコード・パーツ

itemName

オーバーライドしようとしているプロパティを持つレコード項目の名前。

itemProperty

項目プロパティのオーバーライド。詳細については、『EGL プロパティとオーバーライドの概要』を参照してください。

constantName literal

定数の名前および値。命名の規則については、『命名規則』を参照してください。

arrayName

レコードまたはデータ項目の動的または静的配列の名前。このオプションを使用する場合、右側の他のシンボル (*dataItemPartName*、*primitiveType* など) は配列の各要素を参照します。

functionPart

埋め込み関数。構文についての詳細は、『EGL ソース形式の関数パーツ』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 68 ページの『EGL プロパティの概要』
- 207 ページの『ページ・ハンドラー』
- 24 ページの『パーツの参照』
- 62 ページの『EGL での変数の参照』
- 30 ページの『Typedef』

関連する参照項目

- 100 ページの『例外処理』
- 570 ページの『EGL ソース形式の関数パーツ』
- 725 ページの『命名規則』
- 738 ページの『ページ・ハンドラー・フィールドのプロパティ』
- 『ページ・ハンドラー・パーツ・プロパティ』
- 37 ページの『プリミティブ型』
- 795 ページの『EGL での参照の互換性』
- 970 ページの『setError()』
- 1020 ページの『使用宣言』

ページ・ハンドラー・パーツ・プロパティ

ページ・ハンドラー・フィールド に特有のプロパティを除いて、ページ・ハンドラーのプロパティは以下のとおりであり、これらはオプションです。

alias = "alias"

生成された出力の名前に取り込まれるストリング。別名 (alias) を指定しなかった場合は、ページ・ハンドラー・パーツ名が代わりに使用されます。

allowUnqualifiedItemReferences = no、**allowUnqualifiedItemReferences** = yes

構造体項目のコンテナ (構造体項目を保持しているデータ・テーブル、レコー

ド、または書式) の名前を除外しても、コードが構造体項目を参照できるようにするかどうかを指定します。例えば、次のレコード・パーツについて考えましょう。

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

以下の変数はそのパーツに基づいています。

```
myRecord aRecordPart;
```

allowUnqualifiedItemReferences のデフォルト値 (*no*) を受け入れる場合、以下の代入のように `myItem01` を参照するときにレコード名を指定する必要があります。

```
myValue = myRecord.myItem01;
```

ただし、**allowUnqualifiedItemReferences** プロパティを *yes* に設定した場合は、次に示すように、レコード名を指定する必要はありません。

```
myValue = myItem01;
```

デフォルト値 (最良実例) を受け入れることをお勧めします。コンテナ名を指定することにより、コードを読み取る人と EGL に対するあいまいさを減らすことができます。

EGL は規則セットを使用して変数名または項目名が参照するメモリー領域を決定します。詳細については、『変数および定数の参照』を参照してください。

handleHardIOErrors = yes, handleHardIOErrors = no

システム変数 **VGVar.handleHardIOErrors** にデフォルト値を設定します。このシステム変数は、try ブロック内の入出力操作でハード・エラーが発生した後に、プログラムを継続して実行するかどうかを制御します。このプロパティのデフォルト値は *yes* であり、この変数は 1 に設定されます。

詳細については、『VGVar.handleHardIOErrors』および『Exception handling』を参照してください。

includeReferencedFunctions = no, includeReferencedFunctions = yes

ページ・ハンドラー内またはページ・ハンドラーによってアクセスされるライブラリー内にない各関数のコピーがページ・ハンドラー Bean に含まれるかどうかを示します。デフォルト値は *no* です。これは、推奨されるように開発時に以下の規則に従う場合、このプロパティを無視することができることを意味します。

- 共用関数をライブラリーに収める
- 非共用関数をページ・ハンドラーに収める

ライブラリーに収められていない共用関数を使用している場合は、プロパティ **includeReferencedFunctions** を *yes* に設定する場合にのみ生成が可能です。

localSQLScope = no, localSQLScope = yes

SQL 結果セットおよび作成されたステートメントの識別子が、ページ・ハンドラーに対してローカルである (これがデフォルトです) かどうかを示します。

値 *yes* をそのまま使用すると、ページ・ハンドラーから呼び出される異なる複数のプログラムが、同じ識別子を独自に使用することができます。

no を指定すると、識別子はその実行単位全体で共用されます。現行のコード内で作成された識別子は、その他のコード内でも使用可能です。ただし、その他のコードは、**localSQLScope = yes** を使用して、これらの識別子の使用をブロックすることができます。また現行のコードは、他のコードで作成された識別子を参照できます。ただし、参照できるのは、既にその他のコードが実行され、アクセスをブロックしなかった場合だけです。

SQL 識別子を共用する効果は、次のとおりです。

- ページ・ハンドラーまたは呼び出し先プログラムで結果セットを開き、別のコードでその結果セットから行を取得できる。
- 1 つのコードで SQL ステートメントを作成し、そのステートメントを別のコードで実行できる。

msgResource = "logicalName"

エラー・メッセージ表示で使用する Java リソース・バンドルまたはプロパティー・ファイルを識別します。リソース・バンドルまたはプロパティー・ファイルの内容は、キーと関連値のセットで構成されています。

プログラムによる EGL システム関数 `sysLib.setError` の呼び出しに応答して、特定の値が表示されます (呼び出しにその値のキーの使用が含まれる場合)。

onPageLoadFunction = "functionName"

関連 JSP が最初に Web ページを表示するときに制御を受け取るページ・ハンドラー機能の名前。この機能は、ページに表示されるデータの初期値をセットアップする場合に使用できます。このプロパティーは、以前の **onPageLoad** プロパティーです。

関数に渡される引数には、『EGL における参照互換性』で説明されているように参照互換性が必要です。

scope = session, scope = request

ページ・ハンドラーデータが Web ページに送信された後に発生することを指定します。

- 有効範囲が *session* (デフォルト) に設定されている場合、ページ・ハンドラー変数値は、そのユーザー・セッションの間保持されますが、ユーザーが後で同じページ・ハンドラーにアクセスしても、**OnPageLoad** 関数は再度呼び出されません。
- 有効範囲が *request* に設定されている場合、ページ・ハンドラー変数値が失われ、ユーザーが同じページ・ハンドラーにアクセスすると、**OnPageLoad** 関数が再度呼び出されます。

このプロパティーを明示的に設定して、ユーザーの決定を文書化することをお勧めします。この決定は、Web アプリケーションの設計および操作に大きな影響を及ぼします。

throwNrfEofExceptions = no, throwNrfEofExceptions = yes

ソフト・エラーにより例外がスローされるかどうかを指定します。デフォルトは *no* です。背景情報については、『例外処理』を参照してください。

title = "literal"

title プロパティーは、バインド・プロパティーです。つまり、Page Designer で作業をしている場合は、割り当て済みの値がデフォルトとして使用されます。このプロパティーは、ページのタイトルを指定します。

literal は、引用符付きストリングです。

validationBypassFunctions = ["functionNames"]

1 つ以上の *event handler* を示します。イベント・ハンドラーは、JSP 内のボタン制御に関連付けられたページ・ハンドラー関数です。各関数名はコンマで区切られます。

このコンテキストでイベント・ハンドラーを指定すると、イベント・ハンドラーであるか、またはそれに関係するボタンまたはハイパーテキスト・リンクをクリックしたときに、EGL ランタイムは、入力フィールドおよびページの検証をスキップします。このプロパティは、現在のページ・ハンドラー処理を終わらせて別の Web リソースに制御を転送するユーザー・アクションを受信する場合に役立ちます。

validatorFunction = "functionName"

ページ・ハンドラーのバリデーター関数を示します。これは、すべての項目バリデーターが呼び出された後に呼び出されます。これについては、『EGL を使用してビルドされた Web アプリケーションの検証』で説明します。このプロパティは、以前の **validator** プロパティです。

view = "JSPFileName"

ページ・ハンドラーにバインドされている Java Server Page (JSP) の名前、およびその JSP に至るサブディレクトリー・パスを示します。*JSPFileName* は、引用符付きストリングです。

デフォルト値は、ファイル拡張子 **.jsp** が付いたページ・ハンドラーの名前です。このプロパティを指定する場合、ファイル拡張子がある場合には、それを含めてください。

ページ・ハンドラーを保管または生成する場合、同じ名前の JSP ファイル (**view** プロパティで指定した名前) が適切な フォルダー (フォルダー `WebContent\WEB-INF`) にない限り、EGL は後続のカスタマイズのために、プロジェクトに JSP ファイルを追加します。EGL は JSP を上書きしません。

ページ・ハンドラー・フィールドのプロパティ

ページ・ハンドラー・フィールドのプロパティは、ページ・ハンドラー・パーツでフィールドが宣言された場合に有効になる特性を指定します。

プロパティには、次のものがあります。

- 743 ページの『action』
- 745 ページの『byPassValidation』
- 751 ページの『displayName』
- 751 ページの『displayUse』
- 753 ページの『help』
- 762 ページの『newWindow』
- 763 ページの『numElementsItem』
- 766 ページの『selectFromListItem』
- 767 ページの『selectType』
- 772 ページの『validationOrder』

- 778 ページの『value』

関連する概念

- 68 ページの『EGL プロパティの概要』
- 207 ページの『ページ・ハンドラー』

関連するタスク

- 204 ページの『EGL ページ・ハンドラーパーツの作成』
- 216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 735 ページの『ページ・ハンドラー・パーツ・プロパティ』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』

pfKeyEquate

テキスト書式を参照する書式グループを宣言する場合、プロパティ *pfKeyEquate* は、ユーザーが大きい番号のファンクション・キー (PF13 から PF24) を押したときに登録されるキー・ストロークが、ユーザーが 12 よりも小さいファンクション・キーを押したときに登録されるキー・ストロークと同じかどうかを指定します。

pfKeyEquate のデフォルト値 *yes* を受け入れた場合、例えば、PF2 は PF14 と同じであるため、論理式は 12 個のファンクション・キーのみを参照できます。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

関連する概念

- 163 ページの『FormGroup パーツ』

関連する参照項目

- 550 ページの『EGL ソース形式の FormGroup パーツ』

プリミティブ・フィールド・レベル・プロパティ

次の表は、EGL 内のプリミティブ・フィールド・レベル・プロパティをリストしています。

プロパティ	説明
action	ユーザーがボタンまたはリンクをクリックしたときに呼び出されるコードを示します。
align	変数フィールドにおいてデータの長さがフィールドの長さよりも短い場合にデータの位置を指定します。

プロパティ	説明
byPassValidation	ユーザーがボタンまたはリンクをクリックしたときに、EGL ベースの検証がバイパスされるかどうかを表します。
color	テキスト書式のフィールドの色を指定します。
column	項目に関連付けられているデータベース表の列の名前を指します。デフォルトは、項目の名前です。
currency	数値フィールドの値の前に通貨記号を含めるかどうかを指定し、 zeroFormat プロパティにより決められた記号の正確な位置を指定します。
currencySymbol	currency プロパティが有効である場合、どの通貨記号を使用するかを指定します。
dateFormat	日付のフォーマットを識別します。
	フィールドをライト・ペンまたは (エミュレーター・セッションの場合) カーソル・クリックで選択した場合、フィールドの変更されたデータ・タグを設定するかどうかを指定します。
displayName	フィールドの次に表示されるラベルを指定します。
displayUse	EGL フィールドとユーザー・インターフェース制御を関連付けます。
fieldLen	テキスト書式フィールドに表示できる 1 バイト文字の数を指定します。
fill	ユーザーが各フィールド位置にデータを入力する必要があるかどうかを示します。
fillCharacter	テキストや印刷書式、またはページ・ハンドラー・データの未使用箇所を充てんするための文字を指定します。
help	ユーザーが入力フィールドの上にカーソルを置いたときに表示される吹き出しヘルプ・テキストを指定します。
highlight	フィールドを表示するための特殊効果がある場合にこれを指定します。
inputRequired	ユーザーがフィールドにデータを配置する必要があるかどうかを示します。
inputRequiredMsgKey	フィールド・プロパティ inputRequired が yes に設定されており、ユーザーがフィールドにデータを配置できない場合に示されるメッセージを示します。
intensity	表示フォントを強調する度合いを指定します。
isBoolean	フィールドがブール値を表していることを示します。

プロパティ	説明
isDecimalDigit	入力値に 10 進数のみが含まれていることをチェックするかどうかを決定します
isHexDigit	入力値に 16 進数のみが含まれていることをチェックするかどうかを決定します
isNullable	項目を NULL に設定できるかどうかを指定します。これは、項目に関連したテーブル列が NULL に設定できる場合に適切です。
isReadOnly	データベースに書き込みを行うデフォルト SQL ステートメント、または FOR UPDATE OF 文節を含むデフォルト SQL ステートメントから、項目および関連する列を省略するかどうかを示します。
lineWrap	テキストの切り捨てを回避するために折り返しが必要であるときに、改行できるかどうかを示します。
lowerCase	ユーザーの 1 バイト文字入力の英字を小文字に設定するかどうかを示します。
masked	ユーザーが入力した文字を表示するかどうかを指します。
maxLen	データベース列に書き込まれるフィールド・テキストの最大長を指定します。
minimumInput	ユーザーがフィールドにデータを配置する場合に、フィールドに配置する必要のある文字の最小数を指します。
minimumInputMsgKey	ユーザーが以下の操作を行った場合に表示するメッセージを示します。 <ul style="list-style-type: none"> フィールドにデータを配置し、 プロパティ minimumInputRequired に指定された値より少ない文字を配置した場合。
modified	ユーザーによる値の変更とは関係なく、プログラムがフィールドを変更されているとみなすかどうかを指示します。
needsSOSI	ユーザーが ASCII デバイスで MBCHAR 型のデータを入力した場合に、EGL が特別な検査を行うかどうかを示します。
newWindow	EGL ランタイムが action プロパティで識別されるアクティビティに対する応答として Web ページを表示する場合に、新しいブラウザー・ウィンドウを使用するかどうかを指定します。
numElementsItem	ランタイム値が表示する配列エレメントの数を指定するページ・ハンドラー・フィールドを示します。
numericSeparator	3 桁を超える整数部を持つ数値に文字を組み込むかどうかを示します。

プロパティ	説明
outline	このプロパティを使用すると、2 バイト文字をサポートする装置上で、フィールドの端に線を描くことができます。
pattern	検証のために、ユーザー入力テキストを指定されたパターンと突き合わせます。
persistent	フィールドが、SQL レコード用に生成された暗黙の SQL ステートメントに含まれているかどうかを示します。
protect	ユーザーがフィールドにアクセスできるようにするかを指定します。
selectFromListItem	ユーザーが、配列または宣言されているプリミティブ・フィールドに転送される値 (単数または複数) を選択した元の、配列または DataTable 列を識別します。
selectType	宣言されている配列またはプリミティブ・フィールドに取り込む値のタイプを指示します。
sign	ユーザー入力やプログラムによりフィールドに数値を入れた場合に、正 (+) または負 (-) の符号が表示される位置を示します。
sqlDataCode	レコード項目に関連付けられている SQL データ型を示します。
sqlVariableLen	EGL ランタイムがデータを SQL データベースに書き込む前に文字フィールドの末尾ブランクと NULL が切り捨てられるかどうかを示します。
timeFormat	時刻のフォーマットを識別します。
timeStampFormat	書式に表示されるかまたはページ・ハンドラーで維持される、タイム・スタンプの形式を識別します。
typeChkMsgKey	入力データがフィールド・タイプに適さない場合に表示されるメッセージを示します。
upperCase	ユーザーの 1 バイト文字入力の英字を大文字に設定するかどうかを指します。
validationOrder	フィールドのバリデーター機能が、その他のフィールドのバリデーター機能に関連している実行されるのかを示します。
validatorDataTable	ユーザー入力との比較の基本となる dataTable パーツ validator table を示します。
validatorDataTableMsgKey	プロパティ validatorDataTable で指定されたテーブルであるバリデーター・テーブルの要件に合わないデータをユーザーが入力した場合に表示されるメッセージを示します。

プロパティ	説明
validatorFunction	EGL ランタイムが基本検証チェックを行った後で実行されるロジックであるバリデータ関数 (存在する場合) を示します。
validatorFunctionMsgKey	表示されるメッセージを示します
validValues	ユーザー入力として有効な値のセットを示します。
validValuesMsgKey	フィールド・プロパティ validValues が設定されており、ユーザーが範囲外のデータをフィールドに配置した場合に表示されるメッセージを示します。
value	Web ページが表示されるときに、フィールド・コンテンツとして表示される文字列リテラルを示します。
zeroFormat	数値フィールドにゼロ値を表示する方法を指定しますが、 MONEY 型フィールドには指定しません。

action

EGL プロパティ **displayUse** が、*button* または *hyperlink* である場合、プロパティ **action** は、ユーザーがボタンまたはリンクをクリックしたときに呼び出されるコードを示します。**action** に割り当てた値は、ユーザーが Page Designer の Web ページ上のフィールド (または、フィールドを含むレコード) を置く場合のデフォルトとして使用されます。

action の値は、以下の文字列リテラルの種類のうちの 1 つです。

- ページ・ハンドラー内のイベント処理関数の名前
- Web リソース (JSP など) にマップされ、JSF アプリケーション構成リソース・ファイルの navigation-rule エントリーの from-outcome 属性に関連付けられているラベル
- 以下の規則が適用される場合の Java Bean のメソッド名
 - 形式が Bean 名、ピリオド、およびメソッド名である
 - Bean 名が JSF アプリケーション構成リソース・ファイルの管理対象 bean-name エントリーに関連付けられている

action の値を指定しない場合、ユーザーがフィールドをクリックすると、以下のようになります。

- プロパティ **displayUse** の値が *button* の場合は、JSF が同じ Web ページを再表示した後、検証が行われる。
- プロパティ **displayUse** の値が *hyperlink* の場合、検証が行われませんが、JSF は同じ Web ページを再表示する。

関連する概念

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

- 215 ページの『EGL ページ・ハンドラーへのJavaServer Faces コマンド・コンポーネントのバインディング』
- 204 ページの『EGL ページ・ハンドラーパーツの作成』
- 216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 738 ページの『ページ・ハンドラー・フィールドのプロパティ』
- 735 ページの『ページ・ハンドラー・パーツ・プロパティ』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』

align

align プロパティは、変数フィールドについてデータの長さがフィールドの長さよりも短い場合にデータの位置を指定します。

値は、列挙型 **alignKind** です。

left

データをフィールド内の左側に配置します (文字データのデフォルト)。開始スペースは除去され、フィールドの末尾に置かれます。

none

データを位置調整しません。この設定は、文字データについてのみ有効です。

right

データをフィールド内の右側に配置します (数値データのデフォルト)。末尾スペースは除去され、フィールドの先頭に置かれます。この設定は、小数点位または符号を持つ数値データの場合に必須です。

このプロパティは、**DataItem** パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- コンソール書式
- 印刷書式
- テキスト書式
- Web ページ

出力では、文字および数値データがこのプロパティによる影響を受けます。入力の場合、文字データはこのプロパティの影響を受けますが、数値データは常に右そろえにします。

関連する概念

- 524 ページの『EGL での列挙型』
- 68 ページの『EGL プロパティの概要』

関連する参照項目

- 71 ページの『フォーマット設定プロパティ』

byPassValidation

EGL プロパティ **displayUse** が、*button* または *hyperlink* である場合、プロパティ **byPassValidation** は、ユーザーがボタンまたはリンクをクリックしたときに、EGL ベースの検証がバイパスされるかどうかを示します。例えば、ユーザーが「終了」ボタンをクリックしたときなどに、パフォーマンスを向上させるために検証をバイパスしたい場合があります。

byPassValidation に割り当てた値は、ユーザーが Page Designer の Web ページ上のフィールド（または、フィールドを含むレコード）を置く場合のデフォルトとして使用されます。

プロパティは、EGL ベースの検証にのみ影響し、JSF タグで指定されたものには影響しません。詳しくは、『ページ・ハンドラー』を参照してください。

値は、列挙型 **Boolean** です。

no (デフォルト)

入力フィールドは、通常通り検証されます

yes

EGL ランタイムは、ユーザー・データをページ・ハンドラーに戻しません。

関連する概念

- 524 ページの『EGL での列挙型』
- 68 ページの『EGL プロパティの概要』
- 207 ページの『ページ・ハンドラー』

関連するタスク

- 215 ページの『EGL ページ・ハンドラーへのJavaServer Faces コマンド・コンポーネントのバインディング』
- 204 ページの『EGL ページ・ハンドラーパーツの作成』
- 216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 738 ページの『ページ・ハンドラー・フィールドのプロパティ』
- 735 ページの『ページ・ハンドラー・パーツ・プロパティ』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』

color

color プロパティは、テキスト書式内のフィールドの色を指定します。以下のいずれかを選択します。

- black
- blue
- cyan
- defaultColor (デフォルト)
- green
- magenta
- red

- white
- yellow

値 *defaultColor* を割り当てた場合、次の表に示すように他の条件により表示色が決まります。

書式のすべてのフィールドに値 <i>defaultColor</i> が割り当てられているか	<i>protect</i> の値	<i>intensity</i> の値	値 <i>defaultColor</i> が割り当てられたフィールドの表示色
yes	yes または <i>skip</i>	not <i>bold</i>	blue
yes	yes または <i>skip</i>	<i>bold</i>	white
yes	<i>no</i>	not <i>bold</i>	green
yes	<i>no</i>	<i>bold</i>	red
no	いずれかの値	not <i>bold</i>	green
no	いずれかの値	<i>bold</i>	white

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

70 ページの『フィールド表示プロパティ』

column

column プロパティ。項目に関連付けられているデータベース表の列の名前を示します。デフォルトは、項目の名前です。列および関連する項目は、デフォルトの SQL ステートメントに影響を与えます。詳細については、『SQL サポート』を参照してください。

"*columnName*" を、以下の例のように引用符付きストリング、文字型の変数、または連結に置き換えます。

```
column = "Column" + "01"
```

列名が以下の SQL 予約語のいずれかである場合、特別な構文が適用されます。

- CALL
- COLUMNS
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE

- VALUES
- WHERE

以下の例に示すように、これらの名前は二重引用符で囲み、内部の各引用符の前にエスケープ文字 (¥) を付ける必要があります。

```
column = "¥\"SELECT¥\""
```

(これらの予約語をテーブル名として使用する場合も同様です。)

関連する概念

- 477 ページの『VisualAge Generator との互換性』
- 143 ページの『レコード・タイプとプロパティ』
- 247 ページの『SQL サポート』
- 28 ページの『固定構造体』
- 30 ページの『Typedef』

関連するタスク

- 272 ページの『SQL テーブル・データの検索』

関連する参照項目

- 70 ページの『フィールド表示プロパティ』
- 605 ページの『add』
- 613 ページの『close』
- 511 ページの『データの初期化』
- 617 ページの『delete』
- 619 ページの『execute』
- 631 ページの『get』
- 644 ページの『get next』
- 664 ページの『open』
- 679 ページの『prepare』
- 37 ページの『プリミティブ型』
- 793 ページの『レコードとファイル・タイプの相互参照』
- 681 ページの『replace』
- 685 ページの『set』
- 801 ページの『SQL データ・コードおよび EGL ホスト変数』
- 1004 ページの『terminalID』
- 441 ページの『VAGCompatibility』

currency

currency プロパティは、数値フィールドの値の前に通貨記号を含めるかどうかを指定し、**zeroFormat** プロパティにより決められた記号の正確な位置を指定します。MONEY 型のフィールドのフォーマット設定は、**strLib.defaultMoneyFormat** の値に応じて異なり、**currency** プロパティには影響されません。

currency プロパティの値は以下のとおりです。

No (デフォルト)

通貨記号を使用しません。

Yes

currencySymbol で指定された記号を使用します。ここで値が指定されていない場合は、デフォルトの通貨記号を使用します。

デフォルトの通貨記号は、マシンのロケールにより決まります。

このプロパティは、**DataItem** パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- 印刷書式
- テキスト書式
- Web ページ

このプロパティは、入出力時に使用されます。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

71 ページの『フォーマット設定プロパティ』

currencySymbol

currencySymbol プロパティは、**currency** プロパティが有効である場合にどの通貨記号を使用するかを指定します。値は文字列リテラルです。

このプロパティは、**DataItem** パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- 印刷書式
- テキスト書式
- Web ページ

このプロパティは、入出力時に使用されます。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

71 ページの『フォーマット設定プロパティ』

dateFormat

dateFormat プロパティは、日付の形式を識別します。

有効な値は以下のとおりです。

"pattern"

pattern の値は、『日付、時刻、およびタイム・スタンプ・フォーマット指定子』に示す一連の文字から構成されます。

文字は、完全な日付指定の先頭または末尾から除去することが出来ますが、途中の部分を除去することはできません。

defaultDateFormat

ページ・フィールドに指定された場合、**defaultDateFormat** の値は、ランタイム Java ロケールで指定された日付形式です。書式フィールドに指定された場合、デフォルトのパターンは、**systemGregorianCalendar** を選択した場合に等しくなります。

eurDateFormat

「dd.MM.yyyy」パターン (IBM 欧州標準規格の日付形式)。

isoDateFormat

「yyyy-MM-dd」パターン (International Standards Organization (ISO) により指定された日付形式)。

jisDateFormat

「yyyy-MM-dd」パターン (日本工業規格 (JIS) 日付形式)。

usaDateFormat

「MM/dd/yyyy」パターン (IBM USA 標準規格の日付形式)。

systemGregorianCalendar

dd (日数)、MM (月数)、および yy か yyyy (年数) を、d、M、y または数字以外の、区切り文字として使用される文字と一緒に含む、8 または 10 文字のパターン。

Java ランタイム・プロパティに形式が指定されています。

```
vgj.datemask.gregorian.long.NLS
```

NLS

Java ランタイム・プロパティ **vgj.nls.code** に指定されている NLS (各国語サポート) コード。このコードは、『targetNLS』にリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

systemJulianDateFormat

DDD (日数)、および yy か yyyy (年数) を、D、y、または数字以外の、区切り文字として使用される文字と共に含む、6 または 8 文字のパターン。

Java ランタイム・プロパティに形式が指定されています。

```
vgj.datemask.julian.long.NLS
```

NLS

Java ランタイム・プロパティ **vgj.nls.code** に指定されている NLS (各国語サポート) コード。このコードは、『targetNLS』にリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

このプロパティは、DataItem パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- コンソール書式
- 印刷書式
- テキスト書式

- Web ページ

このプロパティは入出力の両方に使用されますが、以下の場合は使用しません。

- フィールドに小数位、通貨記号、数字区切り文字、または符号が含まれる
- フィールドが DBCHAR、MBCHAR、または HEX タイプになっている
- フィールドにマスクを反映する値を含むだけの長さが無いその他の詳細については、『日付の長さに関する考慮事項』を参照してください。

内部日付形式

ユーザーが有効なデータを入力すると、フィールドに指定された形式から以降の検証で使用される内部形式にデータが変換されます。

文字データの内部形式は、システムのデフォルトの形式と同じで、区切り文字を含みます。

数値データの場合、内部形式は以下のとおりです。

- グレゴリオ暦 (短) 00yyMMdd
- グレゴリオ暦 (長) 00yyyyMMdd
- ユリウス暦 (短) 0yyDDD
- ユリウス暦 (長) 0yyyyDDD

日付の長さに関する考慮事項

書式内で、書式上のフィールド長は、指定するフィールド・マスクの長さ以上であることが必要です。フィールドの長さは、日付の内部形式を保持するのに十分な長さであることが必要です。

ページ・フィールドにおける規則は以下のとおりです。

- フィールド長が、指定された日付マスクに十分な長さにする必要があり、これより長くてもかまわない
- 数値フィールドの場合、区切り文字は長さの計算から除外される

次の表に例を示します。

形式タイプ	例	書式フィールドの長さ	ページ・フィールドの最小長さ (文字型)	ページ・フィールドの有効長さ (数値型)
短いグレゴリオ暦	yy/MM/dd	8	8	6
長いグレゴリオ暦	yyyy/MM/dd	10	10	8
短いユリウス暦	DDD-yy	6	6	5
長いユリウス暦	DDD-yyyy	8	8	7

日付の入出力に関する考慮事項

変数フィールドに入力されたデータは、指定された形式で日付が入力されたかチェックされます。ユーザーは、日付および月の先行ゼロを入力する必要はなく、(例え

ば) 08/05/1996 でなく 8/5/1996 を指定することができます。ただし、区切り文字を省略した場合は、先行ゼロをすべて入力する必要があります。

関連する概念

377 ページの『Java ランタイム・プロパティー』

68 ページの『EGL プロパティーの概要』

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

71 ページの『フォーマット設定プロパティー』

584 ページの『Java ランタイム・プロパティー (詳細)』

displayName

displayName プロパティーは、フィールドの次に表示されるラベルを指定します。ユーザーが割り当てた値は、Page Designer の Web ページ上のフィールド (または、フィールドを含むレコード) を置く場合のデフォルトとして使用されます。

このプロパティーの値は、文字列リテラルです。

関連する概念

68 ページの『EGL プロパティーの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

738 ページの『ページ・ハンドラー・フィールドのプロパティー』

735 ページの『ページ・ハンドラー・パーツ・プロパティー』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

displayUse

displayUse プロパティーは、EGL フィールドとユーザー・インターフェース制御を関連付けます。ユーザーが割り当てた値は、Page Designer の Web ページ上のフィールド (または、フィールドを含むレコード) を置く場合のデフォルトとして使用されます。

値は、列挙型 **displayUseKind** です。

button

コントロールには、ボタン・コマンド・タグが含まれます。

secret

データは、ユーザーから可視ではありません。この値は、パスワードに適しています。

hyperlink

action プロパティがイベント処理関数の名前である場合、コントロールにハイパーリンク・コマンド・タグが含まれます。**action** プロパティがラベルである場合、コントロールにリンク・タグが含まれます。どちらのケースでも、ユーザーがリンクをクリックしたときに検証は行われず、入力データは戻されません。

input

コントロールはユーザー入力を受け入れます。コントロールは、最初にページ・ハンドラーによって提供された値を表示します。

table

データはテーブル・タグ内にあります。

output

ページ・ハンドラー・フィールド出力がある場合は、コントロールで可視です。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

215 ページの『EGL ページ・ハンドラーへのJavaServer Faces コマンド・コンポーネントのバインディング』

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

738 ページの『ページ・ハンドラー・フィールドのプロパティ』

735 ページの『ページ・ハンドラー・パーツ・プロパティ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

fieldLen

プロパティ **fieldLen** は、テキスト書式フィールドに表示できる 1 バイト文字の数を指定します。この値には先行する属性バイトは含まれません。

数値フィールドに対する **fieldLen** の値は、フィールドに格納可能な最大の数値を表示できる長さに、(数値に小数点以下の桁がある場合は) 小数点を足した大きさでなければなりません。CHAR、DBCHAR、MBCHAR、または UNICODE 型のフィールドに対する **fieldLen** の値は、2 バイト文字と SI/SO 文字を考慮した大きさでなければなりません。

デフォルトの **fieldLen** は、すべてのフォーマット設定文字を含めて、可能な最大のプリミティブ型の数字を表示するために必要なバイト数です。

関連する概念

68 ページの『EGL プロパティの概要』

関連する参照項目

553 ページの『EGL ソース形式の書式パーツ』

fill

fill プロパティは、ユーザーが各フィールド位置にデータを入力する必要があるかどうかを示します。有効な値は *no* (デフォルト) と *yes* です。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

fillCharacter

fillCharacter プロパティは、テキストや印刷書式、またはページ・ハンドラー・データの未使用箇所を充てんする文字を指示します。また、このプロパティは、*set field full* (*set* を参照) の効果も変更します。このプロパティは、出力にのみ影響します。

デフォルトは、数値の場合にスペース、16 進項目の場合に 0 です。文字型のデフォルトはメディアに応じて異なります。

- テキストまたは印刷書式の場合、デフォルトは空ストリングです。
- ページ・ハンドラー・データの場合、タイプ CHAR または MBCHAR のデータに関するデフォルトはブランクになります。

ページ・ハンドラーでは、**fillCharacter** の値は、DBCHAR または UNICODE 型の項目に関してはスペース (デフォルト) になっている必要があります。

help

help プロパティは、ユーザーが入力フィールドの上にカーソルを置いたときに表示される吹き出しヘルプ・テキストを指定します。ユーザーが割り当てた値は、Page Designer の Web ページ上の EGL フィールド (または、EGL フィールドを含むレコード) を置く場合のデフォルトとして使用されます。

このプロパティの値は、文字列リテラルです。

関連する概念

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 738 ページの『ページ・ハンドラー・フィールドのプロパティー』
- 735 ページの『ページ・ハンドラー・パーツ・プロパティー』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』

highlight

highlight プロパティーは、フィールドを表示するための特殊効果がある場合にこれを指定します。有効な値は以下のとおりです。

noHighlight (デフォルト)

特殊効果を表示しないよう指定し、blink、reverse、または underline を抑止します。

アンダーライン

フィールドの下部にアンダースコアを配置します。

関連する概念

- 524 ページの『EGL での列挙型』
- 68 ページの『EGL プロパティーの概要』

関連する参照項目

- 70 ページの『フィールド表示プロパティー』

inputRequired

inputRequired プロパティーは、ユーザーがフィールドにデータを配置する必要があるかどうかを指示します。有効な値は *no* (デフォルト) と *yes* です。

プロパティー値が *yes* の場合にユーザーがフィールドにデータを配置しないと、EGL ランタイムは、フィールド・プロパティー **inputRequiredMsgKey** に関連して記述されたメッセージを表示します。

関連する概念

- 168 ページの『テキスト書式』

関連する参照項目

- 72 ページの『検証プロパティー』
- 849 ページの『validationFailed()』
- 513 ページの『EGL ソース形式の DataTable パーツ』
- 977 ページの『verifyChkDigitMod10()』
- 978 ページの『verifyChkDigitMod11()』

inputRequiredMsgKey

inputRequiredMsgKey プロパティーは、フィールド・プロパティー **inputRequired** が *yes* に設定されており、ユーザーがフィールドにデータを配置しない場合に表示されるメッセージを示します。

メッセージ・テーブル (メッセージを含むデータ・テーブル) は、プログラム・プロパティー **msgTablePrefix** に示されます。データ・テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

inputRequiredMsgKey の値は、メッセージ・テーブルの最初の列のエントリーに一致するストリングまたはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字ストリングに変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、ストリング内の値は、符号付きまたは符号なし整数である必要があります。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

intensity

intensity プロパティは、表示フォントを強調する度合いを指定します。有効な値は以下のとおりです。

normalIntensity (デフォルト)

フィールドを太文字を使用せずに表示します。

bold

テキストを太文字で表示します。

dim

入力フィールドが使用不可になっている場合などに、テキストを表示する輝度を低くします。

invisible

書式上にフィールドがあることを示さないようにします。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

70 ページの『フィールド表示プロパティ』

isBoolean

isBoolean (以前の **boolean**) プロパティは、フィールドがブール値を表すことを指定します。このプロパティは、有効なフィールド値を制限するもので、入力または出力時のテキストおよび印刷書式やページ・ハンドラーで役立ちます。

EGL ページ・ハンドラーに関連した Web ページでは、ブール項目はチェック・ボックスで示されます。書式上では、次のようになります。

- 数値フィールドの値は 0 (false) または 1 (true)。

- 文字フィールドの値は、各国語に依存した語または語のサブセットにより示されます。例えば英語では、3 つ以上の文字のブール値フィールドは、値 *yes* (*true*) または *no* (*false*) をとり、1 文字のブール値フィールドは *y* または *n* のように値が切り捨てられます。

yes および *no* に対する固有の文字は、ロケールにより決まります。

isDecimalDigit

isDecimalDigit プロパティは、入力値が次のような 10 進数のみを含むことを、チェックするかどうか決定します。

0123456789

有効な値は *no* (デフォルト) と *yes* です。

このプロパティは、文字フィールドにのみ適用されます。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

isHexDigit

isHexDigit プロパティは、入力値が次のような 16 進数文字のみを含むことを、チェックするかどうか決定します。

0123456789abcdefABCDEF

有効な値は *no* (デフォルト) と *yes* です。

このプロパティは、文字フィールドにのみ適用されます。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

isNullable

プロパティ **isNullable** は、項目と関連付けられたテーブル列が NULL に設定できる場合に適切のように NULL 設定可能かどうかを示します。有効な値は *yes* (デフォルト) と *no* です。

isNullable が *yes* に設定された場合にのみ、SQL レコードの任意の項目に対して以下の機能が使用可能です。

- プログラムがデータベースから項目へ NULL 値を受け入れることができる。
- プログラムが **set** ステートメントを使用して、『*set*』に説明しているように項目を NULL に設定することができる。『データ初期化』に説明しているように項目が初期化されます。
- プログラムが **if** ステートメントを使用して、項目が NULL に設定されているかどうかをテストすることができる。

関連する概念

477 ページの『VisualAge Generator との互換性』
143 ページの『レコード・タイプとプロパティ』
247 ページの『SQL サポート』
28 ページの『固定構造体』
30 ページの『Typedef』

関連するタスク

272 ページの『SQL テーブル・データの検索』

関連する参照項目

70 ページの『フィールド表示プロパティ』
605 ページの『add』
613 ページの『close』
511 ページの『データの初期化』
617 ページの『delete』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
664 ページの『open』
679 ページの『prepare』
37 ページの『プリミティブ型』
793 ページの『レコードとファイル・タイプの相互参照』
681 ページの『replace』
685 ページの『set』
801 ページの『SQL データ・コードおよび EGL ホスト変数』
1004 ページの『terminalID』
441 ページの『VAGCompatibility』

isReadOnly

isReadOnly プロパティ。データベースに書き込みを行うデフォルト SQL ステートメント、または FOR UPDATE OF 文節を含むデフォルト SQL ステートメントから、項目および関連する列を削除するかどうかを示します。デフォルト値は *no* です。ただし、次のいずれかの場合、EGL では、構造体項目が「読み取り専用」として扱われます。

- SQL レコードのプロパティ **key** は、構造体項目と関連付けられている列がキー欄であることを示す。
- SQL レコード・パーツが複数のテーブルに関連付けられている。

- SQL 列名が式である。

関連する概念

- 477 ページの『VisualAge Generator との互換性』
- 143 ページの『レコード・タイプとプロパティ』
- 247 ページの『SQL サポート』
- 28 ページの『固定構造体』
- 30 ページの『Typedef』

関連するタスク

- 272 ページの『SQL テーブル・データの検索』

関連する参照項目

- 70 ページの『フィールド表示プロパティ』
- 605 ページの『add』
- 613 ページの『close』
- 511 ページの『データの初期化』
- 617 ページの『delete』
- 619 ページの『execute』
- 631 ページの『get』
- 644 ページの『get next』
- 664 ページの『open』
- 679 ページの『prepare』
- 37 ページの『プリミティブ型』
- 793 ページの『レコードとファイル・タイプの相互参照』
- 681 ページの『replace』
- 685 ページの『set』
- 801 ページの『SQL データ・コードおよび EGL ホスト変数』
- 1004 ページの『terminalID』
- 441 ページの『VAGCompatibility』

lineWrap

EGL プロパティ **lineWrap** は、テキストの切り捨てを回避するためにテキストを改行する必要がある場合に、折り返しができるかどうかを指定します。

有効な値は、列挙型 **lineWrapType** の値です。

character (デフォルト)

フィールド内のテキストは、ホワイト・スペースで分割されません。

compress

ConsoleField 型のフィールド内のテキストは、ホワイト・スペースで分割されますが、ユーザーが (別のフィールドに移動するか、Esc キーを押すことにより) そのフィールドから離れると、テキストの折り返しに使用される余分なスペースはすべて除去されます。この値は、コンソール・フィールドでのみ有効です。

word

可能な場合、フィールド内のテキストはホワイト・スペースで分割されます。

プロパティ **lineWrap** は、DataItem パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- コンソール書式
- 印刷書式
- テキスト書式
- Web ページ

このプロパティは、入出力に影響します。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

71 ページの『フォーマット設定プロパティ』

lowerCase

lowerCase プロパティは、ユーザーの 1 バイト文字入力の英字を小文字に設定するかどうかを指示します。このオプションの値は、次のとおりです。

no (デフォルト)

ユーザーの入力を小文字に設定しません。

yes

ユーザーの入力を小文字に設定します。

masked

masked プロパティは、ユーザーが入力した文字を表示するかどうかを示します。このプロパティは、パスワードの入力に使用されます。このオプションの値は、次のとおりです。

no (デフォルト)

ユーザーが入力した文字は表示されます。

yes

ユーザーが入力した文字は表示されません。

maxLen

プロパティ **maxLen** は、データベース列に書き込まれるフィールド・テキストの最大長を指定します。可能な場合、このプロパティのデフォルト値がフィールド長になりますが、フィールドが **STRING** 型である場合、デフォルト値は存在しません。

関連する概念

477 ページの『VisualAge Generator との互換性』

143 ページの『レコード・タイプとプロパティ』

247 ページの『SQL サポート』

28 ページの『固定構造体』

30 ページの『Typedef』

関連するタスク

272 ページの『SQL テーブル・データの検索』

関連する参照項目

70 ページの『フィールド表示プロパティ』
605 ページの『add』
613 ページの『close』
511 ページの『データの初期化』
617 ページの『delete』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
664 ページの『open』
679 ページの『prepare』
37 ページの『プリミティブ型』
793 ページの『レコードとファイル・タイプの相互参照』
681 ページの『replace』
685 ページの『set』
801 ページの『SQL データ・コードおよび EGL ホスト変数』
1004 ページの『terminalID』
441 ページの『VAGCompatibility』

minimumInput

minimumInput プロパティは、ユーザーがフィールドにデータを配置する場合に、フィールドに配置する必要のある文字の最小数を指示します。デフォルトは 0 です。

ユーザーが配置した文字が最小文字数より少ない場合、EGL ランタイムは、フィールド・プロパティ **minimumInputMsgKey** に関連して記述されたメッセージを表示します。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』
849 ページの『validationFailed()』
513 ページの『EGL ソース形式の DataTable パーツ』
977 ページの『verifyChkDigitMod10()』
978 ページの『verifyChkDigitMod11()』

minimumInputMsgKey

minimumInputMsgKey プロパティは、ユーザーが以下の操作を行った場合に表示するメッセージを示します。

- フィールドにデータを配置し、
- プロパティ **minimumInputRequired** に指定された値より少ない文字を配置した場合。

メッセージ・テーブル (メッセージを含むテーブル) は、プログラム・プロパティ **msgTablePrefix** に示されます。テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

minimumInputMsgKey の値は、メッセージ・テーブルの最初の列のエントリーに一致するストリングまたはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字ストリングに変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、ストリング内の値は、符号付きまたは符号なし整数である必要があります。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティー』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

modified

ユーザーによる値の変更とは関係なく、プログラムがフィールドを変更されているとみなすかどうかを指示します。詳しくは、『変更データ・タグおよびプロパティー』を参照してください。

デフォルトは *no* です。

関連する概念

171 ページの『変更データ・タグおよびプロパティー』

68 ページの『EGL プロパティーの概要』

関連する参照項目

553 ページの『EGL ソース形式の書式パーツ』

needsSOSI

needsSOSI プロパティーは、マルチバイト・フィールド (MBCHAR 型のフィールド) のみに使用され、ユーザーが ASCII デバイスで MBCHAR 型のデータを入力した場合に特別な検査を行うかどうかを指定します。有効な値は *yes* (デフォルト) と *no* です。チェックでは、入力をホスト SO/SI 形式に正しく変換できるかが判別されます。

このプロパティーは、変換中、マルチバイト・ストリングの最後から末尾空白を削除して、2 バイト文字の各サブストリングの間に SO/SI 区切り文字を挿入できるため、役立ちます。正しく変換を行うため、書式フィールドにはマルチバイト値の 2 バイト・ストリングごとに少なくとも 2 つの空白が必要です。

needsSOSI を *no* に設定した場合、ユーザーによる入力フィールドの入力が可能であり、この場合、変換により警告が出ずにデータが切り捨てられます。

needsSOSI を *yes* に設定すると、ユーザーがマルチバイト・データを入力した場合の結果は、次のようになります。

- 十分な空白がある場合には、値はそのまま受け入れられる。または、

- 値が切り捨てられ、ユーザーに警告メッセージが出される。

ユーザーによるマルチバイト・マルチバイトの ASCII 入力を z/OS または iSeries システムで使用する場合は、**needsSOSI** を *yes* に設定します。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティー』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

newWindow

プロパティー **newWindow** は、EGL ランタイムが **action** プロパティーで識別されるアクティビティーに対する応答として Web ページを表示する場合に、新しいブラウザー・ウィンドウを使用するかどうかを指示します。

値は、列挙型 **Boolean** です。

No (デフォルト)

ページの表示には、現行ブラウザー・ウィンドウが使用されます。

Yes

新規ブラウザー・ウィンドウが使用されます。

action プロパティーが指定されていない場合、ページの表示には現行ブラウザー・ウィンドウが使用されます。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティーの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

215 ページの『EGL ページ・ハンドラーへのJavaServer Faces コマンド・コンポーネントのバインディング』

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

743 ページの『action』

738 ページの『ページ・ハンドラー・フィールドのプロパティー』

735 ページの『ページ・ハンドラー・パーツ・プロパティー』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

numElementsItem

構造体フィールド配列で設定された場合、プロパティ **numElementsItem** は、ランタイム値が表示する配列エレメントの数を指定するページ・ハンドラー・フィールドを示します。プロパティは、出力のみに使用され、1 より大きい **occurs** 値を持つ固定レコード構造化フィールドにセットされた場合にのみ意味を持ちます。

numElementsItem の値は、ページ・ハンドラー・フィールドの名前を識別する文字列リテラルです。このプロパティは、使用中の要素の数の標識が含まれている動的配列については有効ではありません。詳しくは、*Arrays* を参照してください。

関連する概念

- 142 ページの『固定レコード・パーツ』
- 68 ページの『EGL プロパティの概要』
- 207 ページの『ページ・ハンドラー』

関連するタスク

- 214 ページの『EGL レコードと Faces JSP との関連付け』
- 204 ページの『EGL ページ・ハンドラーパーツの作成』
- 212 ページの『EGL フィールドの作成と Faces JSP との関連付け』
- 216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 78 ページの『配列』
- 738 ページの『ページ・ハンドラー・フィールドのプロパティ』
- 735 ページの『ページ・ハンドラー・パーツ・プロパティ』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』

numericSeparator

numericSeparator プロパティは、3 桁を超える整数部を持つ数値に文字を組み込むかどうかを示します。数字区切り文字がコンマの場合、例えば千は *1,000* と表示され、百万は *1,000,000* と表示されます。このオプションの値は、次のとおりです。

no (デフォルト)

数字区切り文字を使用しません。

yes

数字区切り文字を使用します。

デフォルトは、マシンのロケールにより決まります。

outline

outline プロパティを使用すると、2 バイト文字をサポートする装置上で、フィールドの端に線を描くことができます。有効な値は以下のとおりです。

box

フィールド内容の周囲にボックスを描きます。

noOutline (デフォルト)

線を描きません。

この他、ボックスのいずれかまたはすべての要素を指定することもできます。この場合、次の例のように 1 つ以上の値を大括弧で囲み、それぞれの値をコンマで区切ります。

```
outline = [left, over, right, under]
```

それぞれの値は、以下のとおりです。

left

フィールドの左端に縦線を描きます。

over

フィールドの上端に横線を描きます。

right

フィールドの右端に縦線を描きます。

under

フィールドの下端に横線を描きます。

各書式フィールドの内容は、属性バイトで始まります。属性バイトを書式の最後の列に置き、**outline** 値を次の列に入れることはできません。この場合、書式の端を超えてしまいます。(フィールドは、次の行に折り返しはしません。) 同様に、属性バイトを書式の最初の列に置いて、**outline** 値をその列内に入れることもできません。**outline** 値は次の列にのみ現れます。

関連する概念

524 ページの『EGL での列挙型』

68 ページの『EGL プロパティの概要』

関連する参照項目

70 ページの『フィールド表示プロパティ』

pattern

検証のために、ユーザー入力テキストを指定されたパターンと突き合わせます。

関連する概念

68 ページの『EGL プロパティの概要』

関連する参照項目

553 ページの『EGL ソース形式の書式パーツ』

persistent

プロパティ **persistent** は、フィールドが SQL レコード用に生成された暗黙の SQL ステートメントに含まれているかどうかを示します。この値が *yes* の場合、以下のようなケースでの実行時にエラーが発生します。

- ユーザーのコードが暗黙の SQL ステートメントに依存し、
- どの列もフィールド固有の **column** プロパティの値に一致しない(デフォルト値はフィールド名)。

一時プログラム変数を、データベース内の変数に対応する列を持たない SQL 行に関連付けたい場合は、**persistent** を *no* に設定します。例えば、プログラムがその行を変更したかどうかを示す変数が必要な場合です。

暗黙の SQL ステートメントの詳細については、『SQL サポート』を参照してください。

関連する概念

477 ページの『VisualAge Generator との互換性』
143 ページの『レコード・タイプとプロパティ』
247 ページの『SQL サポート』
28 ページの『固定構造体』
30 ページの『Typedef』

関連するタスク

272 ページの『SQL テーブル・データの検索』

関連する参照項目

70 ページの『フィールド表示プロパティ』
605 ページの『add』
613 ページの『close』
511 ページの『データの初期化』
617 ページの『delete』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
664 ページの『open』
679 ページの『prepare』
37 ページの『プリミティブ型』
793 ページの『レコードとファイル・タイプの相互参照』
681 ページの『replace』
685 ページの『set』
801 ページの『SQL データ・コードおよび EGL ホスト変数』
1004 ページの『terminalID』
441 ページの『VAGCompatibility』

protect

ユーザーがフィールドにアクセスできるようにするかを指定します。有効な値は以下のとおりです。

no (変数フィールドのデフォルト)

ユーザーがフィールドの値を上書きできるよう設定します。

skip (定数フィールドのデフォルト)

ユーザーがフィールドの値を上書きできないよう設定します。また、次のいずれかの場合には、カーソルがフィールドをスキップします。

- ユーザーがタブ順序の直前のフィールドを操作しているときに、**Tab** を押したか、そのフィールドに内容を入力した場合。または、
- ユーザーがタブ順序の次のフィールドを操作していて、**Shift + Tab** を押した場合。

yes

ユーザーがフィールドの値を上書きできないよう設定します。

関連する概念

68 ページの『EGL プロパティの概要』

関連する参照項目

553 ページの『EGL ソース形式の書式パーツ』

selectFromListItem

プロパティ **selectFromListItem** は、ユーザーが、配列または宣言されているプリミティブ・フィールドに転送される値 (単数または複数) を選択した元の、配列または DataTable 列を識別します。 **selectFromListItem** に割り当てた値は、ユーザーが Page Designer の Web ページに配列またはプリミティブ・フィールドを置く場合のデフォルトとして使用されます。

プロパティ **selectFromListItem** の値は、ソース配列または DataTable 列を識別する文字列リテラルです。

配列の宣言時にこのプロパティを指定すると、ユーザーは複数の値を選択できるようになります。プリミティブ・フィールドの宣言時にこのプロパティを指定すると、ユーザーはただ 1 つの値を選択できるようになります。

ユーザーから受け取る値は、次のいずれかのタイプに対応している必要があります。

- ユーザーが選択した配列エレメントまたは DataTable 列の内容。または、
- 配列または DataTable 索引 (選択された要素または列を識別する整数)。索引の範囲は、1 から使用可能な要素の数まで。

プロパティ **selectType** は、受け取る値のタイプ (ユーザーにより選択された内容、または配列や列の索引) を指示します。

関連する概念

155 ページの『DataTable』

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

78 ページの『配列』

738 ページの『ページ・ハンドラー・フィールドのプロパティ』

735 ページの『ページ・ハンドラー・パーツ・プロパティ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

767 ページの『selectType』

selectType

プロパティ **selectType** は、宣言されている配列またはプリミティブ・フィールドに取り込む値のタイプを指示します。割り当てた値は、ユーザーが Page Designer の Web ページに配列またはプリミティブ・フィールドを置く場合のデフォルトとして使用されます。

値は、列挙型 **selectTypeKind** です。

index (デフォルト)

宣言されている配列またはプリミティブ・フィールドは、ユーザー選択に応答して索引を受け取ります。この場合、配列またはプリミティブ・フィールドは数値型になっている必要があります。

value

宣言されている配列またはプリミティブ・フィールドは、ユーザー選択値を受け取ります。この場合、項目はいずれの型でもかまいません。

背景情報については、**selectFromListItem** プロパティを参照してください。

関連する概念

- 155 ページの『DataTable』
- 68 ページの『EGL プロパティの概要』
- 207 ページの『ページ・ハンドラー』

関連するタスク

- 214 ページの『EGL レコードと Faces JSP との関連付け』
- 204 ページの『EGL ページ・ハンドラーパーツの作成』
- 212 ページの『EGL フィールドの作成と Faces JSP との関連付け』
- 216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

- 78 ページの『配列』
- 738 ページの『ページ・ハンドラー・フィールドのプロパティ』
- 735 ページの『ページ・ハンドラー・パーツ・プロパティ』
- 732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』
- 205 ページの『EGL の Page Designer サポート』
- 766 ページの『selectFromListItem』

sign

sign プロパティは、ユーザー入力やプログラムによりフィールドに数値を入れた場合に、正 (+) または負 (-) の符号を表示する位置を示します。このオプションの値は、次のとおりです。

none

符号を表示しません。

leading

デフォルトです。数値の最初の桁の左に符号を表示します。符号は、後述する **zeroFormat** プロパティにより決められた符号の正確な位置に置かれます。

trailing

符号は、数値の最後の桁の右に表示されます。

sqlDataCode

sqlDataCode プロパティの値。レコード項目に関連付けられている SQL データ型を示す数字。SQL データ・コードは、宣言時、検証時、または生成したプログラムの実行時に、データベース管理システムにアクセスした際に使用されます。

プロパティ **sqlDataCode** は、VisualAge Generator 互換性のための環境をセットアップした場合にのみ使用可能です。詳細については、『*VisualAge Generator との互換性*』を参照してください。

デフォルト値は、次の表に示すように、レコード項目のプリミティブ型と長さによって決まります。詳細については、『*SQL データ・コード*』を参照してください。

EGL プリミティブ型	長さ	SQL データ・コード
BIN	4	501
	9	497
CHAR	<=254	453
	>254 および <=4000	449
	>4000	457
DBCHAR	<=127	469
	>127 および <=2000	465
	>2000	473
DECIMAL	any	485
HEX	any	481
UNICODE	<=127	469
	>127 および <=2000	465
	>2000	473

関連する概念

477 ページの『*VisualAge Generator との互換性*』
143 ページの『*レコード・タイプとプロパティ*』
247 ページの『*SQL サポート*』
28 ページの『*固定構造体*』
30 ページの『*Typedef*』

関連するタスク

272 ページの『*SQL テーブル・データの検索*』

関連する参照項目

70 ページの『*フィールド表示プロパティ*』
605 ページの『*add*』
613 ページの『*close*』
511 ページの『*データの初期化*』
617 ページの『*delete*』
619 ページの『*execute*』
631 ページの『*get*』
644 ページの『*get next*』

664 ページの『open』
679 ページの『prepare』
37 ページの『プリミティブ型』
793 ページの『レコードとファイル・タイプの相互参照』
681 ページの『replace』
685 ページの『set』
801 ページの『SQL データ・コードおよび EGL ホスト変数』
1004 ページの『terminalID』
441 ページの『VAGCompatibility』

sqlVariableLen

プロパティ **sqlVariableLen** (以前の **sqlVar** プロパティ) の値は、EGL ランタイムがデータを SQL データベースに書き込む前に文字フィールドの末尾ブランクと NULL が切り捨てられるかどうかを示します。このプロパティは非文字データに影響を与えません。

対応する SQL テーブル列が `varchar` または `varcharic` SQL データ型である場合は `yes` を指定します。

関連する概念

477 ページの『VisualAge Generator との互換性』
143 ページの『レコード・タイプとプロパティ』
247 ページの『SQL サポート』
28 ページの『固定構造体』
30 ページの『Typedef』

関連するタスク

272 ページの『SQL テーブル・データの検索』

関連する参照項目

70 ページの『フィールド表示プロパティ』
605 ページの『add』
613 ページの『close』
511 ページの『データの初期化』
617 ページの『delete』
619 ページの『execute』
631 ページの『get』
644 ページの『get next』
664 ページの『open』
679 ページの『prepare』
37 ページの『プリミティブ型』
793 ページの『レコードとファイル・タイプの相互参照』
681 ページの『replace』
685 ページの『set』
801 ページの『SQL データ・コードおよび EGL ホスト変数』
1004 ページの『terminalID』
441 ページの『VAGCompatibility』

timeFormat

timeFormat プロパティは、時刻の形式を識別します。

有効な値は以下のとおりです。

"pattern"

pattern の値は、『日付、時刻、およびタイム・スタンプ・フォーマット指定子』に示す一連の文字から構成されます。

文字は、完全な時刻指定の先頭または末尾から除去することが出来ますが、途中の部分を除去することはできません。

defaultTimeFormat

Java 環境のデフォルト値は、Java ロケールによって設定されます。

eurTimeFormat

HH:mm:ss パターン (IBM 欧州標準規格の時刻形式)

isoTimeFormat

HH:mm:ss パターン (International Standards Organization (ISO)) により指定された時刻形式)

jisTimeFormat

HH:mm:ss パターン (日本工業規格の時刻形式)。

usaTimeFormat

hh:mm AM パターン (IBM USA 標準規格の時刻形式)。

このプロパティは、*DataItem* パーツで使用可能であり、次のコンテキストで表示されるフィールドにとって意味があります。

- コンソール書式
- 印刷書式
- テキスト書式
- Web ページ

このプロパティは入出力の両方に使用されますが、以下の場合は使用しません。

- フィールドに小数位、通貨記号、数字区切り文字、または符号が含まれる
- フィールドが *DBCHAR*、*MBCHAR*、または *HEX* タイプになっている
- フィールドにマスクを反映する値を含むだけの長さが無いその他の詳細は、『時刻の長さに関する考慮事項』を参照してください。

時刻の長さに関する考慮事項

書式中、フィールド長は指定された時刻マスクの長さに一致する必要があります。ページ・フィールドにおける規則は以下のとおりです。

- 項目の長さが、指定された時刻マスクに十分な長さで、これを超えることがない
- 数値項目の場合、区切り文字は長さの計算から除外される

時刻の入出力に関する考慮事項

変数フィールドに入力されたデータは、指定された形式で時刻が入力されたかチェックされます。ユーザーは、時間、分、および秒の先行ゼロを入力する必要はな

く、(例えば) 08:15 でなく 8:15 を指定することができます。ただし、区切り文字を省略した場合は、先行ゼロをすべて入力する必要があります。

内部形式で保管された時刻は、時刻として認識されず、単にデータとして認識されます。6 文字の時刻フィールドを、例えば長さ 10 の文字項目に移動した場合、EGL では宛先フィールドにブランクが埋め込まれます。ただし、時刻フィールドを書式上に表示した場合、時刻は必要に応じて内部形式から変換されます。

関連する概念

377 ページの『Java ランタイム・プロパティー』

68 ページの『EGL プロパティーの概要』

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

71 ページの『フォーマット設定プロパティー』

584 ページの『Java ランタイム・プロパティー (詳細)』

timestampFormat

timestampFormat プロパティーは、書式上に表示されるかまたはページ・ハンドラー内で保守される、タイム・スタンプ形式を識別します。

有効な値は以下のとおりです。

"pattern"

pattern の値は、『日付、時刻、およびタイム・スタンプ・フォーマット指定子』に示す一連の文字から構成されます。

文字は、完全なタイム・スタンプ指定の先頭または末尾から除去することが出来ますが、途中の部分の除去することはできません。

defaultTimestampFormat

Java 環境では、デフォルトは Java ロケールにより設定されます。

db2TimestampFormat

yyyy-MM-dd-HH.mm.ss.ffffff パターン (IBM DB2 のデフォルトのタイム・スタンプ・フォーマット)。

odbcTimestampFormat

yyyy-MM-dd HH:mm:ss.ffffff パターン (ODBC のタイム・スタンプ・フォーマット)。

関連する概念

377 ページの『Java ランタイム・プロパティー』

68 ページの『EGL プロパティーの概要』

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

584 ページの『Java ランタイム・プロパティー (詳細)』

typeChkMsgKey

typeChkMsgKey プロパティーは、入力データがフィールド・タイプに適さない場合に表示されるメッセージを示します。

メッセージ・テーブル (メッセージを含むテーブル) は、プログラム・プロパティ **msgTablePrefix** に示されます。テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

typeChkMsgKey の値は、メッセージ・テーブルの最初の列のエントリーに一致する文字列またはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字列に変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、文字列内の値は、符号付きまたは符号なし整数である必要があります。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

upperCase

upperCase プロパティは、ユーザーの 1 バイト文字入力の英字を大文字に設定するかどうかを指示します。

このプロパティは、書式やページ・ハンドラーで役立ちます。

upperCase の値は、以下のとおりです。

No (デフォルト)

ユーザーの入力を大文字に設定しません。

Yes

ユーザーの入力を大文字に設定します。

validationOrder

プロパティ **validationOrder** は、フィールドのバリデーター機能が、その他のフィールドのバリデーター機能に関連していつ実行されるのかを示します。このプロパティは、あるフィールドの検証が前に他の項目が検証されるのに応じて行われる場合に意味を持ちます。

値は、リテラル整数です。

検証は最初に、プロパティ **validationOrder** の値が指定されたフィールドについて行われ、最下位の数値を持つ項目から順に検証されます。続いて、**validationOrder** の値が指定されていない項目について検証が行われます。この場合の検証は、ページ・ハンドラーでのフィールドの定義順に行われます。

関連する概念

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

204 ページの『EGL ページ・ハンドラーパーツの作成』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

738 ページの『ページ・ハンドラー・フィールドのプロパティ』

735 ページの『ページ・ハンドラー・パーツ・プロパティ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

validatorDataTable

validatorDataTable プロパティ (以前の **validatorTable** プロパティ) は、ユーザー入力との比較の基本となる **dataTable** パーツ **validator table** を示します。バリデーター・テーブルは、EGL ランタイムが基本検証チェックを行った後で使用されます (存在する場合)。この基本チェックは、以下のプロパティと関連して記述されます。

- **inputRequired**
- **isDecimalDigit**
- **isHexDigit**
- **minimumInput**
- **needsSOSI**
- **validValues**

チェックはすべて、値間の検証を行う検証関数を指定する **validatorFunction** プロパティを使用する前に行われます。

『EGL ソース形式の *DataTable* パーツ』に記載された、以下のいずれかのタイプのバリデーター・テーブルを指定できます。

matchInvalidTable

ユーザーの入力が、データ・テーブルの最初の列のいずれかの値と異なる必要があることを示します。

matchValidTable

ユーザーの入力が、データ・テーブルの最初の列の値に一致する必要があることを示します。

rangeChkTable

ユーザーの入力が、少なくとも 1 つのデータ・テーブル行の最初の列と 2 番目の列の値の間の値に一致する必要があることを示します。(範囲は包括的です。ユーザーの入力は、行の最初の列または 2 番目の値に一致した場合も有効です。)

検証が失敗した場合、プロパティ **validatorDataTableMsgKey** の値に基づくメッセージが表示されます。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

validatorDataTableMsgKey

validatorDataTableMsgKey プロパティ (以前の **validatorTableMsgKey** プロパティ) は、プロパティ **validatorDataTable** で指定されたテーブルであるバリデーター・テーブルの要件に合わないデータをユーザーが入力した場合に表示されるメッセージを示します。

メッセージ・テーブル (メッセージを含むテーブル) は、プログラム・プロパティ **msgTablePrefix** に示されます。メッセージ・テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

validatorDataTableMsgKey の値は、メッセージ・テーブルの最初の列のエントリと一致するストリングまたはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字ストリングに変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、ストリング内の値は、符号付きまたは符号なし整数である必要があります。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

validatorFunction

validatorFunction プロパティ (以前の **validator** プロパティ) は、EGL ランタイムが基本検証チェックを行った後で実行されるロジックであるバリデーター関数 (存在する場合) を指定します。このチェックは、以下のプロパティと関連して記述されます。

- inputRequired
- isDecimalDigit
- isHexDigit
- minimumInput
- needsSOSI

- `validValues`

(**validatorDataTable** プロパティーに関連して記述されているように) バリデーター・テーブルを使用する前に基本チェックが行われ、**validatorFunction** プロパティーを使用する前にすべてのチェックが行われます。このイベント順序は、バリデーター関数でフィールド間のチェックが可能であり、このようなチェックでは有効なフィールド値を必要とするため、重要です。

validatorFunction の値は、作成したバリデーター関数です。この関数は、エラーを検出した場合、`ConverseLib.validationFailed` を呼び出して書式の再表示を要求するように、パラメーターなしでコーディングしてください。

この 2 つのシステム関数の一方を指定すると検証が失敗する場合、プロパティー **validatorFunctionMsgKey** の値に基づくメッセージが表示されます。ただし、独自のバリデーター関数を指定すると検証が失敗する場合、その関数は **validatorFunctionMsgKey** を使用するのではなく、`ConverseLib.validationFailed` を呼び出してメッセージを表示します。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティー』

849 ページの『`validationFailed()`』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『`verifyChkDigitMod10()`』

978 ページの『`verifyChkDigitMod11()`』

validatorFunctionMsgKey

validatorFunctionMsgKey プロパティー (以前の **validatorMsgKey** プロパティー) は、次の場合に表示されるメッセージを示します。

- **validatorFunction** プロパティーが、`sysLib.verifyChkDigitMod10` または `sysLib.verifyChkDigitMod11` の使用を指定し、
- 指定された関数が、ユーザーの入力にエラーがあることを示した場合。

メッセージ・テーブル (メッセージを含むテーブル) は、プログラム・プロパティー **msgTablePrefix** に示されます。テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

validatorFunctionMsgKey の値は、メッセージ・テーブルの最初の列のエントリーと一致するストリングまたはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字ストリングに変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、ストリング内の値は、符号付きまたは符号なし整数である必要があります。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティー』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

validValues

validValues プロパティー (以前の **range** プロパティー) は、ユーザーが入力できる有効な一連の値を示します。このプロパティーは、数値フィールドまたは文字フィールドに使用されます。プロパティーの形式は、次のとおりです。

```
validValues = arrayLiteral
```

arrayLiteral

単一および 2 つの値の要素からなる配列リテラルで、次の例のようになります。

```
validValues = [ [1,3], 5, 12 ]  
validValues = [ "a", ["bbb", "i"]]
```

各単一要素には、1 つの有効な値が含まれ、2 つの値の各要素には、以下のように範囲が含まれます。

- 数値の場合、左端の値が有効な最下位の値で、右端の値が有効な最上位の値です。前述の例では、値 1、2、および 3 が INT 型のフィールドに有効です。
- 文字フィールドでは、比較が可能な文字数である場合、ユーザー入力は値の範囲に照らして比較されます。例えば、範囲 ["a", "c"] には、先頭文字が「a」、「b」、または「c」であるすべての入力データが (有効として) 含まれます。照合シーケンスでは、ストリング「cat」は「c」より大きくなっていますが、「cat」は有効な入力データです。

一般的な規則では、範囲内の最初の値は *lowValue* と呼ばれ、2 番目の値は *highValue* と呼ばれます。以下のすべてのテストが実行された場合、ユーザーの入力データは有効です。

- ユーザー入力が、*lowValue* または *highValue* に等しい。
- ユーザー入力が *lowValue* より大きく、*highValue* より小さい。
- 比較が可能であるかぎり、最初の一連の入力文字は、*lowValue* 内の最初の一連の文字と一致する。
- 比較が可能であるかぎり、最初の一連の入力文字は、*highValue* 内の最初の一連の文字と一致する。

追加の例は、次のとおりです。

```
// valid values are 1, 2, 3, 5, 7, 9, and 11  
validValues = [[1, 3], 5, 7, 11]  
  
// valid values are the letters "a" and "z"  
validValues = ["a", "z"]  
  
// valid values are any string beginning with "a"  
validValues = ["a", "a"]]
```

```
// valid values are any string
// beginning with a lowercase letter
validValues = [{"a", "z"}]
```

ユーザー入力指定された範囲外の場合、EGL ランタイムは、フィールド・プロパティ **validValuesMsgKey** に関連して記述されているように、メッセージを表示します。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

72 ページの『検証プロパティ』

849 ページの『validationFailed()』

513 ページの『EGL ソース形式の DataTable パーツ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

validValuesMsgKey

validValuesMsgKey プロパティ (以前の **rangeMsgKey** プロパティ) は、フィールド・プロパティ **validValues** が設定されているときに、ユーザーが範囲外のデータをフィールドに配置した場合に表示されるメッセージを示します。

メッセージ・テーブル (メッセージを含むテーブル) は、プログラム・プロパティ **msgTablePrefix** に示されます。テーブル名についての詳細は、『EGL ソース形式の DataTable パーツ』を参照してください。

validValuesMsgKey の値は、メッセージ・テーブルの最初の列のエントリと一致する文字列またはリテラルです。

文字キーが想定されるメッセージ・テーブルで数字キーが使用された場合、数値は文字列に変換されます。数字キーが想定されるメッセージ・テーブルで文字列リテラルが使用された場合、文字列内の値は、符号付きまたは符号なし整数である必要があります。

このプロパティは、数値フィールドにのみ適用されます。

関連する概念

168 ページの『テキスト書式』

関連する参照項目

513 ページの『EGL ソース形式の DataTable パーツ』

849 ページの『validationFailed()』

72 ページの『検証プロパティ』

977 ページの『verifyChkDigitMod10()』

978 ページの『verifyChkDigitMod11()』

value

プロパティ **value** は、Web ページが表示されるときに、フィールド・コンテンツとして表示される文字列リテラルを示します。このリテラルは、Page Designer の Web ページに EGL フィールドを置く場合のデフォルトとして使用されます。

関連する概念

68 ページの『EGL プロパティの概要』

207 ページの『ページ・ハンドラー』

関連するタスク

214 ページの『EGL レコードと Faces JSP との関連付け』

212 ページの『EGL フィールドの作成と Faces JSP との関連付け』

204 ページの『EGL ページ・ハンドラーパーツの作成』

216 ページの『ページ・ハンドラー・コード用の「クイック編集」ビューの使用』

関連する参照項目

738 ページの『ページ・ハンドラー・フィールドのプロパティ』

735 ページの『ページ・ハンドラー・パーツ・プロパティ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

205 ページの『EGL の Page Designer サポート』

zeroFormat

zeroFormat プロパティは、MONEY 型フィールドではなく、数値フィールドにゼロ値を表示する方法を指定します。このプロパティは、**numeric separator**、**currency**、および **fillCharacter** プロパティにより影響を受けます。 **zeroFormat** の値は、以下のとおりです。

Yes

ゼロ値は数 0 として表示されます。これは、小数点 (例えば 2 桁の小数部により項目を定義した場合に 0.00) および通貨記号と区切り文字 (\$000,000.00 など。 **currency** および **numericSeparator** プロパティの値により異なる) を使用して表すことができます。プロパティ **zeroFormat** の値が **yes** の場合、以下の規則が適用されます。

- 充てん文字 (**fillCharacter** プロパティの値) が 0 の場合、データは文字 0 によりフォーマット設定されます。
- 充てん文字が NULL の場合、データは左そろえされます。
- 充てん文字がブランクの場合、データは右そろえされます。
- 充てん文字がアスタリスク (*) の場合、ブランクの代わりにアスタリスクが左側の充てん文字に使用されます。

No 一連の充てん文字にゼロ値が表示されます。

関連する参照項目

584 ページの『Java ランタイム・プロパティ (詳細)』

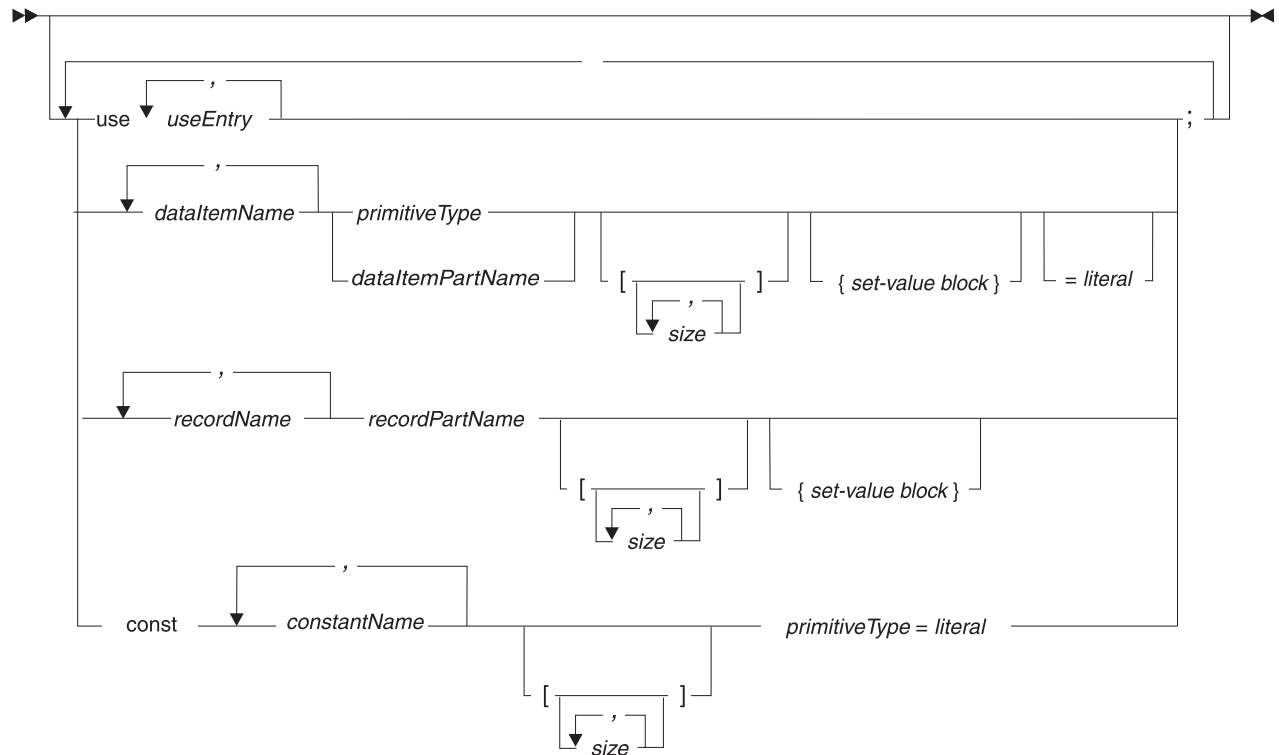
685 ページの『set』

420 ページの『currencySymbol』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

パラメーター以外のプログラム・データ

プログラム・データの構文図は、以下のとおりです。



use useEntry

dataTable またはライブラリーへのより簡単なアクセスを提供し、formGroup の書式にアクセスする場合に必要となります。詳細については、『使用宣言』を参照してください。

dataItemName

プリミティブ・フィールドの名前。命名の規則については、『命名規則』を参照してください。

primitiveType

プリミティブ・フィールドの型または、(配列の場合は) 配列エレメントのプリミティブ型。この型に応じて、以下の情報が必要になります。

- パラメーターの長さまたは (配列に関連付けられている) 配列エレメントの長さ。この長さは、メモリー領域内の文字数または桁数を表す整数です。
- 一部の数値型には、小数点以下の桁数を表す整数を指定できます。小数点は、データとともに保管されません。
- INTERVAL 型または TIMESTAMP 型の項目には、日時マスクを指定できます。これは、項目の値の特定の位置に意味 (「年の桁」など) を割り当てるものです。

詳しくは、『プリミティブ型』および各型のトピックを参照してください。

dataItemPartName

プログラムに対して可視の `dataItem` パーツ名。可視性についての詳細は、『[パーツの参照](#)』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『[Typedef](#)』を参照してください。

size

配列内の要素数。要素の数を指定すると、その配列は指定した要素数で初期化されます。

set-value block

詳細については、『[EGL プロパティと設定値のブロック](#)』を参照してください。

recordName

レコードの名前。命名の規則については、『[命名規則](#)』を参照してください。

recordPartName

プログラムに対して可視のレコード・パーツ名。可視性についての詳細は、『[パーツの参照](#)』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『[Typedef](#)』を参照してください。

const *constantName primitiveType=literal*

定数の名前、型および値。引用符付きストリング (文字型の場合)、数値 (数値型の場合)、または適切な型付きの値の配列 (配列の場合) を指定します。以下に例を示します。

```
const myString String = "Great software!";  
const myArray BIN[] = [36, 49, 64];  
const myArray02 BIN[][] = [[1,2,3],[5,6,7]];
```

命名の規則については、『[命名規則](#)』を参照してください。

関連する概念

- 15 ページの『[EGL プロジェクト、パッケージ、およびファイル](#)』
- 68 ページの『[EGL プロパティの概要](#)』
- 19 ページの『[パーツ](#)』
- 147 ページの『[プログラム・パーツ](#)』
- 62 ページの『[EGL での変数の参照](#)』
- 170 ページの『[テキスト・アプリケーションのセグメンテーション](#)』
- 72 ページの『[値の設定ブロック](#)』
- 812 ページの『[EGL ステートメントおよびコマンドの構文図](#)』
- 30 ページの『[Typedef](#)』

関連する参照項目

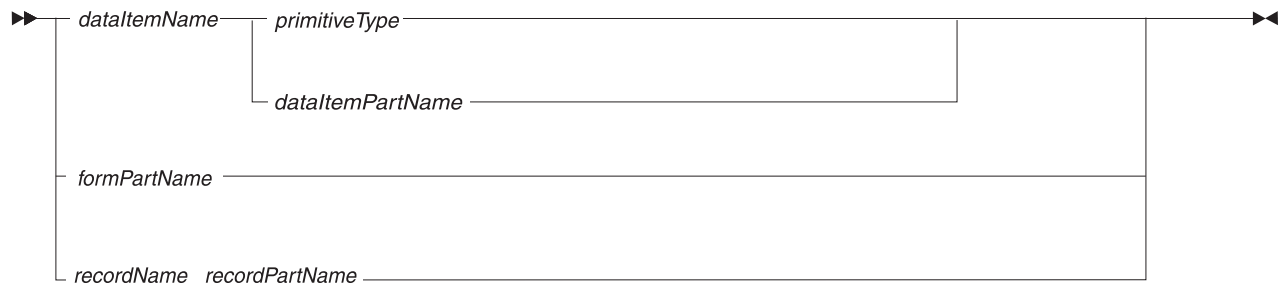
- 78 ページの『[配列](#)』
- 511 ページの『[データの初期化](#)』
- 512 ページの『[EGL ソース形式の DataItem パーツ](#)』
- 513 ページの『[EGL ソース形式の DataTable パーツ](#)』
- 532 ページの『[EGL ソース形式](#)』
- 93 ページの『[EGL ステートメント](#)』
- 629 ページの『[forward](#)』

570 ページの『EGL ソース形式の関数パーツ』
 578 ページの『EGL ソース形式の索引付きレコード・パーツ』
 792 ページの『入力書式』
 792 ページの『入力レコード』
 45 ページの『INTERVAL』
 579 ページの『入出力エラー値』
 714 ページの『EGL ソース形式の MQ レコード・パーツ』
 725 ページの『命名規則』
 37 ページの『プリミティブ型』
 796 ページの『EGL ソース形式の相対レコード・パーツ』
 799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
 804 ページの『EGL ソース形式の SQL レコード・パーツ』
 47 ページの『TIMESTAMP』

1020 ページの『使用宣言』

プログラム・パラメーター

プログラム・パラメーターの構文図は、以下のとおりです。



dataItemName

プリミティブ・フィールドの名前。命名の規則については、『命名規則』を参照してください。

primitiveType

プリミティブ・フィールドの型。この型に応じて、以下の情報が必要になります。

- パラメーターの長さ。メモリ領域の文字数または桁数を表す整数です。
- 一部の数値型には、小数点以下の桁数を表す整数を指定できます。小数点は、データとともに保管されません。
- INTERVAL 型または TIMESTAMP 型の項目には、日時マスクを指定できます。これは、項目の値の特定の位置に意味（「年の桁」など）を割り当てるものです。

dataItemPartName

プログラムに対して可視の `dataItem` パーツ名。可視性についての詳細は、『パーツの参照』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『Typedef』を参照してください。

formPartName

書式の名前。

書式には、プログラムの使用宣言の 1 つを示す `formGroup` を経由してアクセスする必要があります。パラメーターとしてアクセスされる書式は、ユーザーには表示されませんが、他のプログラムから受け渡されるフィールド値にアクセスできるようにします。

命名の規則については、『命名規則』を参照してください。

recordName

レコードまたは固定レコードの名前。命名の規則については、『命名規則』を参照してください。

recordPartName

プログラムから可視のレコード・パーツ名 (固定レコード・パーツ)。可視性についての詳細は、『パーツの参照』を参照してください。

パーツは、形式のモデルとして機能します。詳しくは、『*Typedef*』を参照してください。

レコード・パラメーターに対する入力または出力 (I/O) には、以下のことが当てはまります。

- 他のプログラムから渡されるレコードには、入出力エラー値 `endOfFile` などのレコードの状態は含まれていません。同様に、レコードの状態の変更はいずれも呼び出し元に戻されません。このため、レコード・パラメーターの I/O を実行する場合には、レコードのテストは、プログラムの終了前に行う必要があります。
- レコードに対して行われる入出力操作は、引数に指定されたレコード・プロパティではなく、パラメーターに指定されたレコード・プロパティを使用します。
- `indexedRecord`、`mqRecord`、`relativeRecord`、または `serialRecord` タイプのレコードの場合、レコード宣言と関連付けられたファイルまたはメッセージ・キューは、プログラム・リソースではなく実行単位リソースとして扱われます。レコード・プロパティ `fileName` (または `queueName`) が同じ値を持つ場合、ローカル・レコード宣言は同じファイル (またはキュー) を共用します。実行単位のファイルまたはキューと関連付けるレコード数にかかわらず、物理ファイルを一度に 1 つずつファイルまたはキュー名に関連付けることができ、EGL は、ファイルをクローズし、再オープンすることによりこの規則を実行します。

別の EGL プログラムから送信された引数は、関連するパラメーターと参照互換性があることが必要です。詳細については、『EGL での参照の互換性』を参照してください。

関連する概念

147 ページの『プログラム・パーツ』

24 ページの『パーツの参照』

62 ページの『EGL での変数の参照』

812 ページの『EGL ステートメントおよびコマンドの構文図』

30 ページの『*Typedef*』

関連する参照項目

78 ページの『配列』

413 ページの『EGL ソース形式の基本レコード・パーツ』
512 ページの『EGL ソース形式の DataItem パーツ』
532 ページの『EGL ソース形式』
578 ページの『EGL ソース形式の索引付きレコード・パーツ』
45 ページの『INTERVAL』
725 ページの『命名規則』
37 ページの『プリミティブ型』
795 ページの『EGL での参照の互換性』
796 ページの『EGL ソース形式の相対レコード・パーツ』
799 ページの『EGL ソース形式のシリアル・レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』
47 ページの『TIMESTAMP』

EGL ソース形式のプログラム・パーツ

EGL ファイルでプログラム・パーツを宣言します。これについては、『EGL ソース形式』で説明しています。このファイルを作成する場合、以下の事項に従ってください。

- プログラムによって外部的に使用されるパーツのみを組み込みます。
- 他の基本パーツ (dataTable、ライブラリー、プログラム、またはページ・ハンドラー) を組み込まないようにします。

次の例では、2 つの組み込み関数を含む呼び出し先プログラム・パーツ、スタンドアロンの関数、およびスタンドアロンのレコード・パーツを示します。

```

Program myProgram type basicProgram (employeeNum INT)
{
    includeReferencedFunctions = yes
}

// プログラムグローバル変数
employees record_ws;
employeeName char(20);

// 必要組み込み関数
Function main()
    // 従業員名の初期化
    recd_init();

    // 渡された employeeNum に基づいて
    // 目的の従業員名を取得
    employeeName = getEmployeeName(employeeNum);
end

// 別の組み込み関数
Function recd_init()
    employees.name[1] = "Employee 1";
    employees.name[2] = "Employee 2";
end

// スタンドアロンの関数
Function getEmployeeName(employeeNum INT) returns (CHAR(20))

    // ローカル変数
    index BIN(4);
    index = 2;

```

```

        if (employeeNum > index)
            return("Error");
        else
            return(employees.name[employeeNum]);
        end
    end

    // 従業員の typeDef として機能するレコード・パーツ
    Record record_ws type basicRecord
    10 name CHAR(20)[2];
    end

```

その他の詳細については、プログラムの特定タイプごとのトピックを参照してください。

関連する概念

19 ページの『パーツ』
147 ページの『プログラム・パーツ』

関連する参照項目

『EGL ソース形式の基本プログラム』
532 ページの『EGL ソース形式』
570 ページの『EGL ソース形式の関数パーツ』
786 ページの『EGL ソース形式のテキスト UI プログラム』

EGL ソース形式の基本プログラム

基本プログラムの例を次に示します。

```

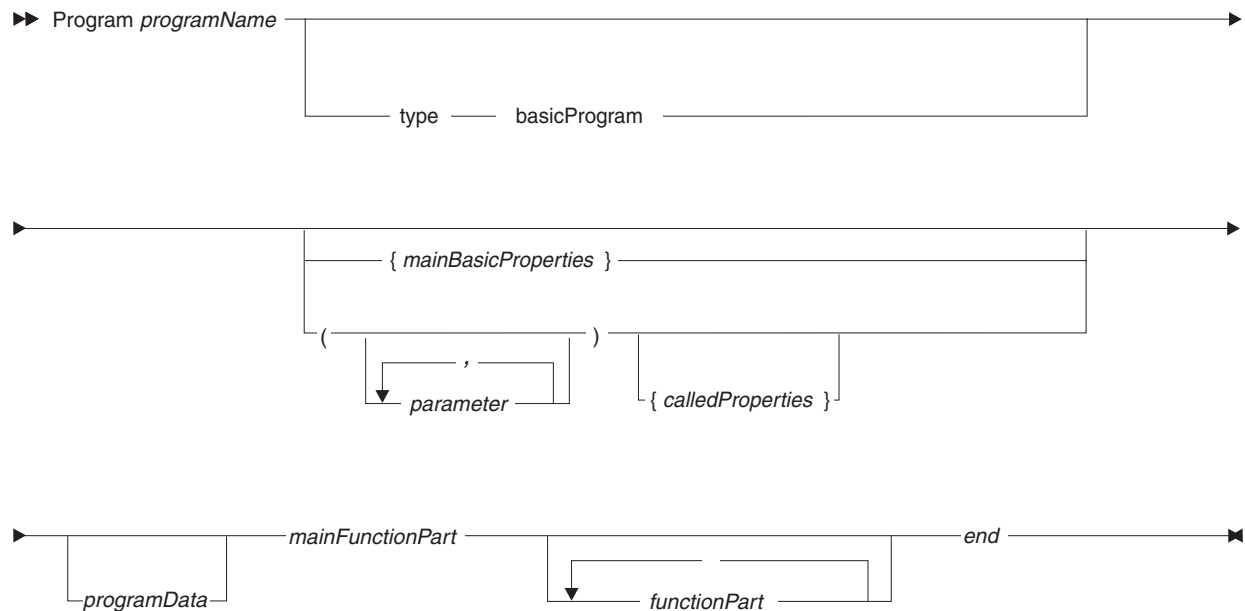
program myCalledProgram type basicProgram
    (buttonPressed int, returnMessage char(25))

    function main()
        returnMessage = "";
        if (buttonPressed == 1)
            returnMessage = "Message1";
        end

        if (buttonPressed == 2)
            returnMessage = "Message2";
        end
    end
end

```

basicProgram タイプのプログラム・パーツの構文図は、以下のとおりです。



Program *programPartName* ... **end**

パーツをプログラム・パーツとして識別し、その名前およびタイプを指定します。プログラム名の後ろに左括弧が付いている場合、そのプログラムは呼び出し側基本プログラムです。

alias プロパティ（後述）を設定しない場合、生成されるプログラムの名前は、*programPartName* になります。

その他の規則については、『命名規則』を参照してください。

mainBasicProperties

メイン基本プログラムのプロパティには以下のオプションがあります。

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEofExceptions**

詳細については、『プログラムのプロパティ』を参照してください。

parameter

パラメーター名を指定します。これは、データ項目、レコード、または書式、つまりレコードまたはデータ項目の動的配列です。命名の規則については、『命名規則』を参照してください。

呼び出し元の引数が変数（定数またはリテラルではない）の場合、パラメーターの変更により、呼び出し元で使用可能なメモリー領域が変更されます。

各パラメーターはコンマにより区切られます。その他の詳細については、『プログラム・パラメーター』を参照してください。

calledProperties

以下の呼び出し先のプロパティは、オプションです。

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **localSQLScope**
- **msgTablePrefix**
- **throwNrfEofExceptions**

詳細については、『プログラムのプロパティ』を参照してください。

programData

変数および使用宣言については、『パラメーター以外のプログラム・データ』に記載されています。

mainFunctionPart

main と名付けられた必要関数です。パラメーターは取りません。(パラメーターを取ることのできるプログラム・コードは、プログラム自体か、*main* 以外の関数のみです。)

関数のコーディングの詳細については、『EGL ソース形式の関数パーツ』を参照してください。

functionPart

このプログラムに対してプライベートである埋め込み関数。関数のコーディングの詳細については、『EGL ソース形式の関数パーツ』を参照してください。

関連する概念

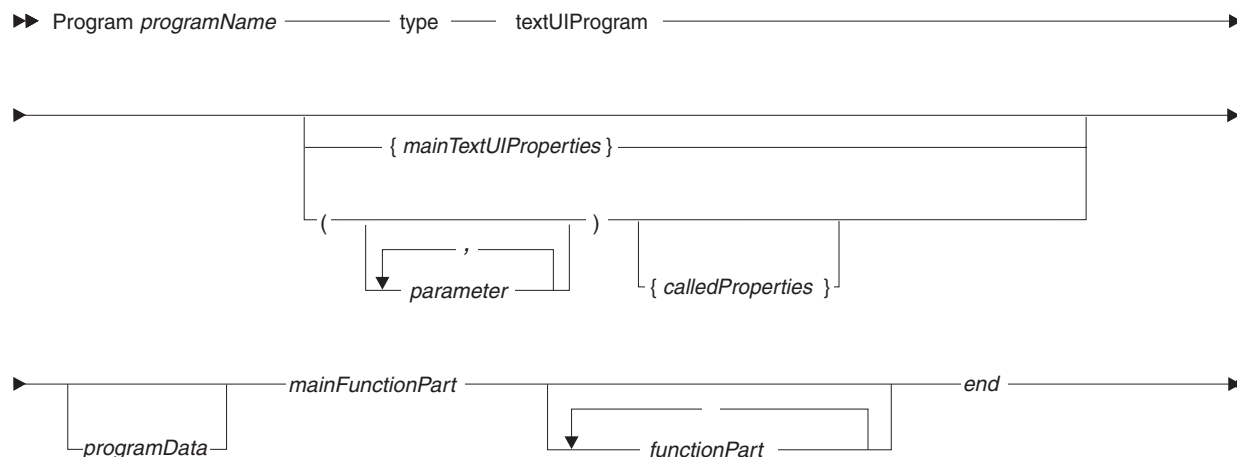
- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 68 ページの『EGL プロパティの概要』
- 19 ページの『パーツ』
- 147 ページの『プログラム・パーツ』
- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 532 ページの『EGL ソース形式』
- 570 ページの『EGL ソース形式の関数パーツ』
- 725 ページの『命名規則』
- 779 ページの『パラメーター以外のプログラム・データ』
- 781 ページの『プログラム・パラメーター』
- 783 ページの『EGL ソース形式のプログラム・パーツ』
- 789 ページの『プログラム・パーツ・プロパティ』
- 1020 ページの『使用宣言』

EGL ソース形式のテキスト UI プログラム

textUIProgram タイプのプログラム・パーツの構文図は、以下のとおりです。



Program *programPartName* ... end

パーツをプログラム・パーツとして識別し、その名前およびタイプを指定します。プログラム名の後ろに左括弧が付いている場合、そのプログラムは呼び出し側基本プログラムです。

alias プロパティ (後述) を設定しない場合、生成されるプログラムの名前は、*programPartName* になります。**alias** プロパティ (後述) を設定しない場合、生成されるプログラムの名前は、*programPartName* になるか、COBOL プログラムを生成している場合は、*programPartName* の先頭の 8 文字になります。

その他の規則については、『命名規則』を参照してください。

mainTextUIProperties

以下のメイン・テキスト UI プログラムのプロパティはオプションです。

- **alias**
- **allowUnqualifiedItemReferences**
- **handleHardIOErrors**
- **includeReferencedFunctions**
- **inputForm**
- **inputRecord**
- **localSQLScope**
- **msgTablePrefix**
- **segmented**
- **throwNrfEofExceptions**

詳細については、『プログラムのプロパティ』を参照してください。

parameter

パラメーター名を指定します。これは、データ項目、レコード、または書式、つまりレコードまたはデータ項目の動的配列です。命名の規則については、『命名規則』を参照してください。

呼び出し元の引数に変数 (定数またはリテラルではない) の場合、パラメーターの変更により、呼び出し元で使用可能なメモリー領域が変更されます。

各パラメーターはコンマにより区切られます。その他の詳細については、『プログラム・パラメーター』を参照してください。

calledProperties

以下の呼び出し先のプロパティは、オプションです。

- **alias**
- **allowUnqualifiedItemReferences**
- **includeReferencedFunctions**
- **msgTablePrefix**

詳細については、『プログラムのプロパティ』を参照してください。

programData

変数および使用宣言については、『パラメーター以外のプログラム・データ』に記載されています。

mainFunctionPart

main と名付けられた必要関数です。パラメーターは取りません。(パラメーターを取ることのできるプログラム・コードは、プログラム自体か、*main* 以外の関数のみです。)

関数のコーディングの詳細については、『EGL ソース形式の関数パーツ』を参照してください。

functionPart

組み込み関数です。プログラム以外の論理パーツには使用不可です。関数のコーディングの詳細については、『EGL ソース形式の関数パーツ』を参照してください。

テキスト UI プログラムの例を次に示します。

```
Program HelloWorld type textUIprogram
{
  use myFormgroup;
  myMessage char(25);

  function main()
  while (ConverseVar.eventKey not pf3)
    myTextForm.msgField = "          ";
    myTextForm.msgField="myMessage";
    converse myTextForm;
    if (ConverseVar.eventKey is pf3)
      exit program;
    end
    if (ConverseVar.eventKey is pf1)
      myMessage = "Hello Word";
    end
  end
end
end
```

関連する概念

15 ページの『EGL プロジェクト、パッケージ、およびファイル』

68 ページの『EGL プロパティの概要』

19 ページの『パーツ』

147 ページの『プログラム・パーツ』

170 ページの『テキスト・アプリケーションのセグメンテーション』

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

532 ページの『EGL ソース形式』
570 ページの『EGL ソース形式の関数パーツ』
725 ページの『命名規則』
779 ページの『パラメーター以外のプログラム・データ』
781 ページの『プログラム・パラメーター』
783 ページの『EGL ソース形式のプログラム・パーツ』
『プログラム・パーツ・プロパティー』
1020 ページの『使用宣言』

プログラム・パーツ・プロパティー

プログラム・パーツ・プロパティーは、プログラムが呼び出し先かメインかによって異なり、メインの場合、プログラムのタイプが、テキスト UI かによって異なります。プロパティーには、次のものがあります。

alias = *"alias"*

生成された出力の名前に取り込まれるストリング。**alias** プロパティーを設定しない場合、プログラム・パーツ名が代わりに使用されます。

alias プロパティーはすべてのプログラムで使用可能です。

allowUnqualifiedItemReferences = no, **allowUnqualifiedItemReferences** = yes

コードが構造内の項目を参照するときに、コンテナおよびサブストラクチャー修飾子を省略できるかどうかを指定します。

allowUnqualifiedItemReferences プロパティーはすべてのプログラムで使用可能です。

例えば、次のレコード・パーツについて考えましょう。

```
Record aRecordPart type basicRecord
  10 myItem01 CHAR(5);
  10 myItem02 CHAR(5);
end
```

以下の変数はそのパーツに基づいています。

```
myRecord aRecordPart;
```

allowUnqualifiedItemReferences のデフォルト値 (*no*) を受け入れる場合、以下の代入のように `myItem01` を参照するときにレコード名を指定する必要があります。

```
myValue = myRecord.myItem01;
```

ただし、**allowUnqualifiedItemReferences** プロパティーを *yes* に設定した場合は、次に示すように、レコード名を指定する必要はありません。

```
myValue = myItem01;
```

デフォルト値 (最良実例) を受け入れることをお勧めします。コンテナ名を指定することにより、コードを読み取る人と EGL に対するあいまいさを減らすことができます。

EGL は規則セットを使用して変数名または項目名が参照するメモリー領域を決定します。詳細については、『変数および定数の参照』を参照してください。

handleHardIOErrors = yes, handleHardIOErrors = no

システム変数 **VGVar.handleHardIOErrors** にデフォルト値を設定します。このシステム変数は、try ブロック内の入出力操作でハード・エラーが発生した後に、プログラムを継続して実行するかどうかを制御します。このプロパティのデフォルト値は *yes* であり、この変数は 1 に設定されます。

handleHardIOErrors を *no* に設定して変数を 0 に設定していない場合、VisualAge Generator から移行されたコードが、以前のように機能しないことがあります。

このプロパティはすべてのプログラムで使用可能です。詳細については、『VGVar.handleHardIOErrors』および『Exception handling』を参照してください。

includeReferencedFunctions = no, includeReferencedFunctions = yes

プログラム内またはプログラムによってアクセスされるライブラリー内にない各関数のコピーがプログラムに含まれるかどうかを示します。

includeReferencedFunctions プロパティはすべてのプログラムで使用可能です。

デフォルト値は *no* です。これは、推奨されるように開発時に以下の規則に従う場合、このプロパティを無視することができることを意味します。

- 共用関数をライブラリーに収める
- 非共用関数をプログラムに収める

ライブラリーに収められていない共用関数を使用している場合は、プロパティ **includeReferencedFunctions** を *yes* に設定する場合にのみ生成が可能です。

inputForm = "formName"

プログラム・ロジックが実行される前にユーザーに提示される書式を識別します。詳細については、『入力書式』に説明します。

inputForm プロパティは、メイン・テキスト UI プログラムでのみ使用可能です。

inputRecord = "inputRecord"

プログラムが自動的に初期化され、**transfer** ステートメントを使用して制御権を移動するプログラムからデータを受け取ることができるグローバルな基本レコードを識別します。追加情報については、『入力レコード』を参照してください。

inputRecord プロパティはすべてのメイン・プログラムで使用可能です。

localSQLScope = yes, localSQLScope = no

SQL 結果セットおよび作成されたステートメントの識別子が、プログラムに対してローカルである (これがデフォルトです) かどうかを示します。値 *yes* をそのまま使用すると、異なる複数のプログラムが同じ識別子を独自に使用することができます。

no を指定すると、識別子はその実行単位全体で共用されます。現行のコード内で作成された識別子は、その他のコード内でも使用可能です。ただし、その他のコードは、**localSQLScope = yes** を使用して、これらの識別子の使用をブロックすることができます。また現行のコードは、他のコードで作成された識別子を参照できます。ただし、参照できるのは、既にその他のコードが実行され、アクセスをブロックしなかった場合だけです。

SQL 識別子を共用する効果は、次のとおりです。

- 1 つのプログラムで結果セットを開き、別のプログラムでその結果セットから行を取得できる。
- 1 つのプログラムで SQL ステートメントを作成し、そのステートメントを別のプログラムで実行できる。

localSQLScope プロパティはすべてのプログラムで使用可能です。

msgTablePrefix = "prefix"

プログラムのメッセージ・テーブルとして使用されるデータ・テーブルの名前の最初の 1 文字から 4 文字を指定します。この名前のその他の文字は、『EGL ソース形式の DataTable パーツ』にリストされている各国語コードの 1 つに対応します。

msgTablePrefix プロパティは、すべての基本またはテキスト UI プログラムで使用可能です。

Web アプリケーションで実行されるプログラムは、メッセージ・テーブルを使用しませんが、JavaServer Faces メッセージ・リソースを使用します。そのリソースの詳細については、**msgResource** プロパティの説明を参照してください。

- EGL ソース形式のページ・ハンドラー・パーツ

segmented = no, **segmented** = yes

『セグメンテーション』で説明しているように、プログラムがセグメント化されているかどうかを示します。デフォルトは、メイン・テキスト UI プログラムで *no* です。その他のタイプのプログラムではプロパティは有効ではありません。

throwNrfEofExceptions = no, **throwNrfEofExceptions** = yes

ソフト・エラーにより例外がスローされるかどうかを指定します。デフォルトは *no* です。背景情報については、『例外処理』を参照してください。

関連する概念

147 ページの『プログラム・パーツ』

62 ページの『EGL での変数の参照』

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

513 ページの『EGL ソース形式の DataTable パーツ』

100 ページの『例外処理』

629 ページの『forward』

792 ページの『入力書式』

792 ページの『入力レコード』

725 ページの『命名規則』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

812 ページの『EGL ステートメントおよびコマンドの構文図』

1012 ページの『handleHardIOErrors』

入力書式

テキスト・アプリケーションで実行するメインプログラムを宣言する場合、入力書式を指定できます。入力書式は、プログラム・ロジック実行前にユーザーに表示される書式です。

2 つのシナリオが考えられます。

- プログラムが EGL 生成プログラムの `show-form-returning-to` ステートメントのターゲットである場合、送信側プログラムがユーザーに書式を示します。その書式は受信側プログラムの入力書式と同一である必要があります。受信側プログラムは、ユーザーが書式を処理依頼した後に限って呼び出されます。ユーザーが書式を処理依頼した後は、受信側プログラムが入力書式を 2 度表示することではなく、代わりに初期ロジック (実行関数) が実行されます。
- プログラムが別のプログラム (EGL または非 EGL) の転送文のターゲットである場合、またはプログラムがユーザーやオペレーティング・システムによって起動される場合、受信側プログラムは入力書式を変換します(この場合、その書式の入力フィールドは表示前に初期化されます)。ユーザーが書式を処理依頼した後、初期ロジック (実行関数) が実行されます。

入力書式は、プログラム・パーツの宣言で指定した書式グループになければなりません。

関連する参照項目

511 ページの『データの初期化』

入力レコード

入力レコードは任意のメインプログラムにあります。入力レコードは、EGL で生成されたプログラムによって自動的に初期化されるグローバル・レコードです。レコードはタイプ `basicRecord` である必要があります。

プログラムがレコード付き転送の結果として開始される場合、プログラムは入力レコード (そのプログラムの内部にある) を初期化してから、転送されたデータをレコードに割り当てます。

受信したデータよりも入力レコードの方が長い場合は、入力レコードの余分の領域に、レコード初期化時に割り当てられた値が保持されます。受信したデータよりも入力レコードの方が短い場合は、余分のデータは切り捨てられます。

転送されたデータ内のプリミティブ型が、入力レコードの同等の位置にあるプリミティブ型と非互換である場合は、受信側プログラムが異常終了する可能性があります。

関連する概念

68 ページの『EGL プロパティの概要』

19 ページの『パーツ』

477 ページの『VisualAge Generator との互換性』

関連する参照項目

511 ページの『データの初期化』

レコードとファイル・タイプの相互参照

次の表は、レコード・タイプとファイル・タイプの関連を、ターゲット・プラットフォームごとに示しています。

関連する概念

143 ページの『レコード・タイプとプロパティ』

331 ページの『リソース関連とファイル・タイプ』

関連タスク

336 ページの『EGL ビルド・ファイルへのリソース関連パーツの追加』

336 ページの『EGL ビルド・ファイル内のリソース関連パーツの編集』

338 ページの『EGL ビルド・ファイルからのリソース関連パーツの除去』

関連する参照項目

432 ページの『resourceAssociations』

可変長レコードをサポートするプロパティ

レコード・パーツを宣言すると、可変長レコードの使用をサポートするプロパティを含めることができます。順次ファイルへのアクセスには可変長シリアル・レコード、VSAM ファイルへのアクセスには可変長シリアル・レコードまたは可変長索引付きレコード、MQSeries メッセージ・キューへのアクセスには可変長 MQ レコードをそれぞれ使用できます。

lengthItem プロパティを持つ可変長レコード

lengthItem プロパティがある場合、以下の場合に使用される項目を示します。

- コードがファイルまたはキューからレコードを読み取るとき。長さ項目を使用して可変長レコードに読み取られたバイト数を受け取ります。
- コードがレコードを書き込むとき。長さ項目を使用してファイルまたはキューに追加されたバイト数を指定します。

長さ項目は、次のいずれかになります。

- 同じレコード内の構造項目
- プログラムではグローバル、またはレコードにアクセスする関数ではローカルなレコード内の構造項目 (長さ項目はプログラムまたは関数で宣言されたレコード変数となります)。
- プログラムにとってグローバルなデータ項目、またはレコードにアクセスする関数にとってローカルなデータ項目

長さ項目には以下のような特性があります。

- BIN、DECIMAL、INT、NUM、または SMALLINT のプリミティブ型を持つ
- 小数部を含まない
- 桁数は最大 9 桁まで

lengthItem プロパティを持つ可変長レコードの例は以下のとおりです。

```

Record mySerialRecordPart1 type serialRecord
{
  fileName = "myFile",
  lengthItem = "myOtherField"
}
10 myField01 BIN(4);    // 2 バイト長
10 myField02 NUM(3);    // 3 バイト長
10 myField03 CHAR(20);  // 20 バイト長
end

```

長さ項目が文字項目でない場合、レコードの書き込み時、長さ項目の値は項目の境界を越える必要があります。例えば、mySerialRecordPart1 タイプのレコードには長さ項目 myOtherField があり、それには 2、5、6、7、...、24、25 を設定できます。myOtherField に 2 が設定されているレコードには、myField01 の値のみが含まれます。myOtherField に 5 が設定されているレコードには、myField01 および myField02 の値が含まれます。myOtherField に 6 から 24 が設定されているレコードには、myField03 のパーツの値も含まれます。

numElementsItem プロパティを持つ可変長レコード

NumElementsItem プロパティがある場合、コードがファイルまたはキューへの追加または更新を行う場合に使用する項目を示します。可変長レコードには、最終の最上位構造体項目として配列がある必要があります。要素項目数の値は、書き込まれる配列要素の実際の数を表します。値の範囲は、0 から最大値までです。この最大値は、レコード内の最後のトップレベル構造体項目の宣言で指定された *occurs* 値です。

書き出されたバイト数は、以下の合計となります。

- レコードの固定長部分のバイト数
- 要素項目の数の値に、最後の配列の各要素内のバイト数を乗算した値

要素項目数には以下のような特性があります。

- BIN、DECIMAL、INT、NUM、SMALLINT のプリミティブ型を持つ
- 小数部を含まない
- 桁数は最大 9 桁まで

numElementsItem プロパティを持つ可変長レコード・パーツの例は以下のとおりです。

```

Record mySerialRecordPart2 type serialRecord
{
  fileName = "myFile",
  numElementsItem = "myField02"
}
10 myField01 BIN(4);    // 2 バイト長
10 myField02 NUM(3);    // 3 バイト長
10 myField03 CHAR(20)[3]; // 60 バイト長
  20 mySubField01 CHAR(10);
  20 mySubField02 CHAR(10);
end

```

要素項目 myField02 に 2 が設定されている mySerialRecordPart2 タイプのレコードを書き込みすると、myField01、myField02、および 2 つのオカレンス myField03 を持つ可変長レコードがファイルまたはキューに書き込まれます。

要素項目数は、可変長レコードの固定長パーツ内の項目の値と同じである必要があります。非修飾参照を使用して、要素項目数に名前を付けます。例えば、`myRecord.myField02` ではなく `myField02` を使用します。

要素項目の数は、ファイルからレコードを読み取っているときには無効です。

lengthItem プロパティと numElementsItem プロパティの両方を持つ可変長レコード

可変長レコードに `lengthItem` プロパティと `numElementsItem` プロパティの両方が指定されている場合、要素項目数を使用してレコード長が計算されます。計算されたレコード長は、レコードがファイルに書き込みされる前にレコード長項目に移動します。

呼び出しまたは転送で渡される可変長レコード

呼び出しで可変長レコードが渡される場合は、次のことが適用されます。

- レコードに対して指定された最大長のスペースが予約されます。
- `callLink` エLEMENTの値としてプロパティ `type` が `remoteCall` または `ejbCall` である場合、可変長項目は、レコード内部にある必要があります (長さ項目が存在する場合)。詳細については、『`callLink` エLEMENT』を参照してください。

同様に、可変長レコードが転送時に渡される場合は、レコードに指定された最大長に対するスペースが予約されます。

関連する概念

285 ページの『MQSeries のサポート』

143 ページの『レコード・タイプとプロパティ』

関連する参照項目

442 ページの『`callLink` エLEMENT』

717 ページの『MQ レコードのプロパティ』

EGL での参照の互換性

パラメーターまたは変数はメモリーの一領域を表します。場合によっては、変数はパラメーター名や従業員 ID などのインタレストのビジネス・データを含みます。あるいは、変数は参照変数 で、実行時にビジネス・データにアクセスする時に使用する値 (特にメモリー・アドレス) を含みます。

ある非参照変数を別の非参照変数に割り当てると、同一のビジネス・データが 2 つ作成されます。例えば、`assignment` ステートメントに含まれるソース変数が特定の従業員 ID を含んでいる場合は、この `assignment` ステートメントによってターゲット変数にもその従業員 ID が含まれることになります。ただし、ある参照変数を別の参照変数に割り当てると、同じメモリー領域へのアクセスに使用される値がソース変数とターゲット変数の両方に含まれます。

以下の場合には、参照の互換性規則 (後で説明します) が適用されます。

- ある参照変数を別の参照変数に割り当てると、または

- EGL が引数と関連したパラメーターとの間でデータを転送し、しかも以下のいずれかに該当する場合
 - 受信関数内のパラメーターが修飾子 INOUT を含む。
 - パラメーターがページ・ハンドラーの onPageLoad 関数に含まれる。
 - パラメーターが、別の EGL プログラムによって呼び出された EGL プログラムに含まれる。

これらの場合は、引数がソース、パラメーターがターゲットになります。

参照の互換性の規則は以下のとおりです。

- 参照変数は、同じ型の参照変数にのみ割り当てたり、渡したりすることができます。
- ソース (または引数) がプリミティブ型または DataItems 配列を参照する場合は、次のステートメントが適用されます。
 - プリミティブ特性 (存在する場合) が同一でなければなりません。例えば、CHAR(6) 型の引数は CHAR(7) 型のパラメーターと互換性がありません。
 - NULL 可能の引数は、NULL 可能のパラメーターまたは NULL 可能でないパラメーターと互換性があります。NULL 可能でない引数は、NULL 可能でないパラメーターとのみ互換性があります。
- 異なるパッケージに含まれるパーツは、DataItem パーツを除いて、型が異なるものと見なされます。
- 固定レコードまたは構造化フィールドの場合、引数の長さはパラメーターの長さ以上でなければなりません。この規則により、受信コードによるメモリーへの無効なアクセスが防止されます。

関連する概念

207 ページの『ページ・ハンドラー』

関連する参照項目

564 ページの『関数パラメーター』

570 ページの『EGL ソース形式の関数パーツ』

732 ページの『EGL ソース形式のページ・ハンドラー・パーツ』

781 ページの『プログラム・パラメーター』

783 ページの『EGL ソース形式のプログラム・パーツ』

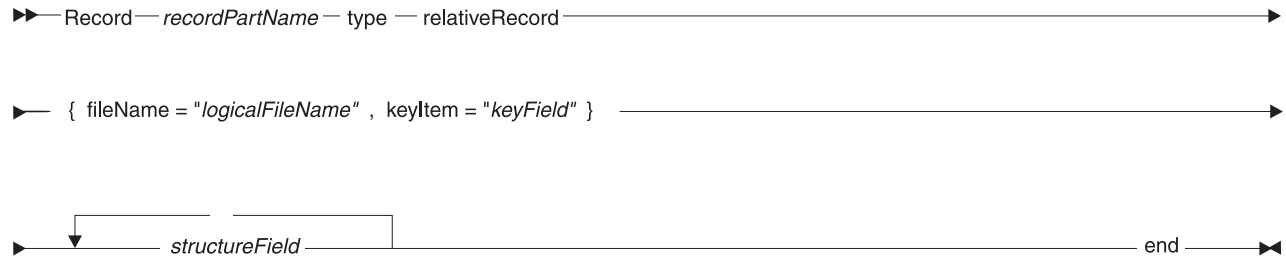
EGL ソース形式の相対レコード・パーツ

EGL ファイルで relativeRecord 型の固定レコード・パーツを宣言します。これについては、『EGL ソース形式』で説明します。

相対レコード・パーツの例を次に示します。

```
Record myRelativeRecordPart type relativeRecord
{
  fileName = "myFile",
  keyItem  = "myKeyItem"
}
10 myKeyItem NUM(4);
10 myContent CHAR(76);
end
```

相対レコード・パーツの構文図は、次のとおりです。



Record *recordPartName* **relativeRecord**

パーツをタイプ **relativeRecord** として識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

fileName = *"logicalFileName"*

論理ファイル名。入力の意味については、『リソース関連 (概説)』を参照してください。命名の規則については、『命名規則』を参照してください。

keyItem = *"keyField"*

キー・フィールド。次のメモリー領域を指定できます。

- 同じ固定レコード内にあるフィールド
- プログラムに対してグローバルな変数またはフィールド、あるいは固定レコードにアクセスする関数に対してローカルな変数またはフィールド

キー・フィールドを指定するには、非修飾の参照を使用する必要があります。例えば、*myRecord.myField* ではなく *myField* を使用します。(ただし、関数では、他のフィールドを参照するのと同じようにキー・フィールドを参照できます。) キー・フィールドは、レコードにアクセスする関数のローカルな有効範囲内で固有でなければなりません。または、ローカルな有効範囲に存在せず、かつグローバルな有効範囲内で固有でなければなりません。

キー・フィールドには、次の特性があります。

- NUM、BIN、DECIMAL、INT、または SMALLINT のプリミティブ型を持つ
- 小数部を含まない
- 桁数は最大 9 桁まで

相対レコード・キー・フィールドを使用するのは **get** ステートメントと **add** ステートメントのみですが、ファイル・アクセスに固定レコードを使用するすべての関数がキー・フィールドを使用できる必要があります。

structureField

構造体フィールド。詳細については、『EGL ソース形式の構造体フィールド』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 24 ページの『パーツの参照』
- 19 ページの『パーツ』
- 141 ページの『レコード・パーツ』
- 62 ページの『EGL での変数の参照』
- 30 ページの『Typedef』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

78 ページの『配列』

512 ページの『EGL ソース形式の DataItem パーツ』

532 ページの『EGL ソース形式』

570 ページの『EGL ソース形式の関数パーツ』

578 ページの『EGL ソース形式の索引付きレコード・パーツ』

714 ページの『EGL ソース形式の MQ レコード・パーツ』

725 ページの『命名規則』

37 ページの『プリミティブ型』

783 ページの『EGL ソース形式のプログラム・パーツ』

331 ページの『リソース関連とファイル・タイプ』

799 ページの『EGL ソース形式のシリアル・レコード・パーツ』

804 ページの『EGL ソース形式の SQL レコード・パーツ』

808 ページの『EGL ソース形式の構造体フィールド』

実行単位

実行単位 は、ローカル呼び出しまたは (場合によっては) 転送により関連付けられているプログラムのセットです。各実行単位には以下の特性があります。

- プログラムは 1 つのグループとして実行される。処理できないハード・エラーが発生すると、実行単位のすべてのプログラムがメモリーから除去される。
- プログラムは同じランタイム・プロパティを共有する。実行単位では同一のデータベースおよびファイルが使用される。例えば `sysLib.connect` または `VGLib.connectionService` を呼び出してデータベースに動的に接続すると、同一実行単位で制御を受け取るすべてのプログラムでこの接続が使用される。

Java 実行単位は、単一のスレッドで実行されるプログラムで構成されています。新規の実行単位は、ユーザーがプログラムを起動したときのように、メイン・プログラムで開始することができます。**transfer** ステートメントもメインプログラムを起動しますが、同じ実行単位を継続します。

以下の場合には、呼び出されたプログラムが、実行単位の初期プログラムとなります。

- 呼び出しが、EJB セッション Bean からの呼び出しである。または、
- 呼び出しがリモート呼び出しである。ただし、以下の場合に同一の実行単位が継続している場合を除く。
 - 呼び出し先プログラムが EGL または VisualAge Generator によって生成された。または
 - TCP/IP リスナーが呼び出しに関係していない。

Java 実行単位のすべてのプログラムには、同一の Java ランタイム・プロパティが適用されます。

関連する概念

377 ページの『Java ランタイム・プロパティ』
338 ページの『リンケージ・オプション・パーツ』

関連する参照項目

270 ページの『デフォルト・データベース』

957 ページの『connect()』
980 ページの『connectionService()』

resultSetID

結果セット ID は、EGL 構文内にあり、リレーショナル・データベースにアクセスして以下の種類の文に関連付ける必要がある場合に使用されます。

- 最初に、結果セットを選択する **open** ステートメントまたは **get** ステートメント、あるいは結果セットを戻すストアード・プロシージャを呼び出す **open** ステートメント
- 次に、結果セットにアクセスするステートメント

SQL レコードを入出力オブジェクトとして使用する場合、レコード名は、1 種類のステートメントを別の種類のステートメントに関連付けるには十分です。ただし、異なるステートメントにおいて更新のために異なる列セットを取得するように、レコードに関連付けられた SQL ステートメントを変更した場合はこの限りではありません。この場合、結果セット ID を使用して、EGL **replace** ステートメントに関連付けられた結果セットを識別してください。

関連する概念

247 ページの『SQL サポート』

関連する参照項目

681 ページの『replace』
664 ページの『open』
631 ページの『get』

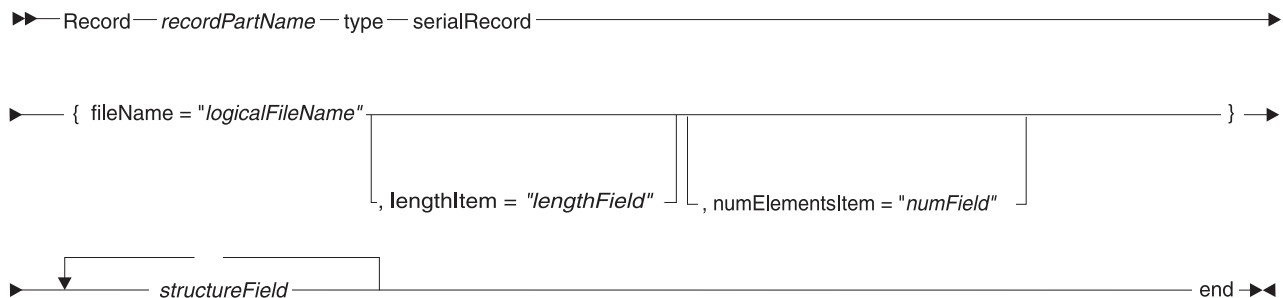
EGL ソース形式のシリアル・レコード・パーツ

EGL ファイルでタイプ `serialRecord` のレコード・パーツを宣言します。これについては、『EGL ソース形式』で説明しています。

シリアル・レコード・パーツの例を次に示します。

```
Record mySerialRecordPart type serialRecord
{
    fileName = "myFile"
}
10 myField01 CHAR(2);
10 myField02 CHAR(78);
end
```

シリアル・レコード・パーツの構文図は、以下のとおりです。



Record *recordPartName* **serialRecord**

パーツをタイプ **serialRecord** として識別し、パーツ名を指定します。命名の規則については、『命名規則』を参照してください。

fileName = *"logicalFileName"*

論理ファイル名。入力の意味については、『リソース関連 (概説)』を参照してください。命名の規則については、『命名規則』を参照してください。

lengthItem = *"lengthField"*

長さフィールド。詳細については、『可変長レコードをサポートするプロパティ』を参照してください。

numElementsItem = *"numField"*

要素フィールドの数。詳細については、『可変長レコードをサポートするプロパティ』を参照してください。

structureField

構造体フィールド。詳細については、『EGL ソース形式の構造体フィールド』を参照してください。

関連する概念

- 15 ページの『EGL プロジェクト、パッケージ、およびファイル』
- 24 ページの『パーツの参照』
- 19 ページの『パーツ』
- 141 ページの『レコード・パーツ』
- 62 ページの『EGL での変数の参照』
- 331 ページの『リソース関連とファイル・タイプ』
- 30 ページの『Typedef』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

- 78 ページの『配列』
- 512 ページの『EGL ソース形式の DataItem パーツ』
- 532 ページの『EGL ソース形式』
- 570 ページの『EGL ソース形式の関数パーツ』
- 578 ページの『EGL ソース形式の索引付きレコード・パーツ』
- 714 ページの『EGL ソース形式の MQ レコード・パーツ』
- 725 ページの『命名規則』
- 37 ページの『プリミティブ型』
- 783 ページの『EGL ソース形式のプログラム・パーツ』

793 ページの『可変長レコードをサポートするプロパティ』
796 ページの『EGL ソース形式の相対レコード・パーツ』
804 ページの『EGL ソース形式の SQL レコード・パーツ』
808 ページの『EGL ソース形式の構造体フィールド』

SQL データ・コードおよび EGL ホスト変数

SQL データ・コードというプロパティは、EGL ホスト変数と関連付ける SQL データ型を示しています。SQL データ・コードは、宣言時、検証時、または生成したプログラムの実行時に、データベース管理システムによって使用されます。

プリミティブ型である CHAR、DBCHAR、HEX、または UNICODE に該当するホスト変数の SQL データ・コードは、変更することができます。ただし、ホスト変数がこれら以外のいずれかのプリミティブ型である場合、SQL データ・コードは固定です。

EGL によってデータベース管理システムから列定義を検索した場合、検索結果に SQL データ・コードがあった場合は、これに変更を加えないでください。

これ以降の説明内容を次に示します。

- 『変数と固定長の列』
- 『SQL データ型と EGL プリミティブ型との互換性』
- 803 ページの『VARCHAR、VARGRAPHIC、および関連する LONG データ型』
- 803 ページの『DATE、TIME、および TIMESTAMP』

変数と固定長の列

テーブル列が可変長か固定長かを指定するには、次の表に示すように、対応するホスト変数の SQL データ・コードを該当する値に設定します。

EGL 基本型	SQL データ型	可変長または固定長	SQL データ・コード
CHAR	CHAR (デフォルト)	固定長	453
	VARCHAR、長さ < 255	可変長	449
	VARCHAR、長さ > 254	可変長	457
DBCHAR、 UNICODE	GRAPHIC (デフォルト)	固定長	469
	VARGRAPHIC、長さ < 128	可変長	465
	VARGRAPHIC、長さ > 127	可変長	473

注: SQL データ型では null 標識の使用が必要な場合がありますが、この要件が EGL プログラムのコード記述方法に影響を与えることはありません。null の詳細については、『SQL サポート』を参照してください。

SQL データ型と EGL プリミティブ型との互換性

EGL ホスト変数とこれに対応する SQL テーブル列に互換性のある状態は、次のいずれかです。

- SQL 列の書式が文字データの書式であり、EGL ホスト変数の型が CHAR で、その長さが SQL 列の長さ以下である場合。
- SQL 列の書式が DBCHAR データの書式であり、EGL ホスト変数の型が DBCHAR で、その長さが SQL 列の長さ以下である場合。
- SQL 列の書式が数値の書式であり、EGL ホスト変数の型が次のいずれかである場合。
 - BIN (2 バイトまたは 4 バイトで、小数点以下の桁がないもの)。
 - DECIMAL (小数点以下の桁を含み、最大 18 桁)。DECIMAL 変数の桁数は、EGL ホスト変数や SQL 列の桁数と同じであること。
 - SMALLINT
- SQL 列のデータ型に制限がなく、EGL ホスト変数の型は HEX であり、SQL 列とホスト変数のバイト数が同じ場合。データはデータ転送時に変換されない。

型が HEX の EGL ホスト変数では、EGL プリミティブ型に対応しないデータ型の SQL 列であれば、どのような SQL 列へのアクセスでもサポートされています。

文字データを SQL テーブル列から読み取って、長さの短いホスト変数に変換する場合、その内容は右端から切り捨てられます。切り捨てるテストを行うには、EGL の **if** ステートメントで予約語 **trunc** を使用します。

数値データを SQL テーブル列から読み取って長さの短いホスト変数に変換する場合、先頭のゼロの桁は左端から切り捨てられます。ゼロの桁を切り捨てても桁数がホスト変数より大きい場合は、(10 進数の) 小数点以下が右端から削除されます。このときエラーは表示されません。小数点以下が削除されても桁数が多すぎる場合は、負の SQL コードが戻り、オーバーフロー条件になっていることが表示されます。

次の表に、EGL エディターの検索機能によってデータベース管理システムから情報が抽出されたときに割り当てられる EGL ホスト変数特性を示します。

SQL データ型	EGL ホスト変数特性			SQL データ・コード (SQLTYPE)
	基本型	長さ	バイト数	
BIGINT	HEX	16	8	493
CHAR	CHAR	1?32767	1?32767	453
DATE	CHAR	10	10	453
DECIMAL	DECIMAL	1 から 18	1?10	485
DOUBLE	HEX	16	8	481
FLOAT	HEX	16	8	481
GRAPHIC	DBCHAR	1?16383	2?32766	469
INTEGER	BIN	9	4	497
LONG VARBINARY	HEX	65534	32767	481
LONG VARCHAR	CHAR	>4000	>4000	457
LONG VARGRAPHIC	DBCHAR	>2000	>4000	473

SQL データ型	EGL ホスト変数特性			SQL データ・コード (SQLTYPE)
	基本型	長さ	バイト数	
NUMERIC	DECIMAL	1 から 18	1?10	485
REAL	HEX	8	4	481
SMALLINT	BIN	4	2	501
TIME	CHAR	8	8	453
TIMESTAMP	CHAR	26	26	453
VARBINARY	HEX	2?65534	1?32767	481
VARCHAR	CHAR	?4000	?4000	449
VARGRAPHIC	DBCHAR	?2000	?4000	465

VARCHAR、VARGRAPHIC、および関連する LONG データ型

VARCHAR 型または VARGRAPHIC 型の SQL テーブル列には最大長が定義されており、retrieve コマンドでは、この最大長を使用して EGL ホスト変数に長さを割り当てます。ただし、LONG VARCHAR 型または LONG VARGRAPHIC 型の SQL テーブル列には最大長が定義されていないので、retrieve コマンドでは、SQL データ型の最大長を使用して長さを割り当てます。

DATE、TIME、および TIMESTAMP

EGL システム・デフォルトの長いグレゴリオ形式で使用される形式が、SQL データベース・マネージャーで指定される日付形式と同じであることを確認してください。EGL 形式の設定方法については、『VGVar.currentFormattedGregorianDate』を参照してください。

システム変数 *VGVar.currentFormattedGregorianDate* で提供される日付が SQL データベース・マネージャーで使用される形式になるように、この 2 つの形式を一致させることができます。

関連する概念

247 ページの『SQL サポート』

関連する参照項目

71 ページの『SQL 項目のプロパティ』

1007 ページの『currentFormattedGregorianDate』

SQL レコードの内部レイアウト

次のような場合は、SQL レコードの内部レイアウトに注意する必要があります。

- EGL の assignment ステートメントを使用して SQL レコードを別の種類のレコードにコピーするか、またはその逆方向のコピーをする場合。
- EGL プログラムに渡されるランタイムの引数が SQL レコードであるのに、プログラム・パラメーターが SQL レコードではない場合。

- EGL 関数に渡されるランタイムの引数が SQL レコードである。この場合、パラメーターは、作業用ストレージ・レコードでなければならない場合。
- SQL レコードを EGL 以外のプログラムのパラメーターとして受け取る場合。

SQL レコードの各構造体項目の前には、4 バイトのデータがあります。最初の 2 バイトは NULL 標識で、ここが NULL の場合は負の値と解釈されます。2 番目の 2 バイトは長さフィールド用に予約されているので、このフィールドは決して書き換ええない てください。

関連する概念

- 149 ページの『関数パーツ』
- 147 ページの『プログラム・パーツ』
- 247 ページの『SQL サポート』

関連する参照項目

- 407 ページの『代入』

EGL ソース形式の SQL レコード・パーツ

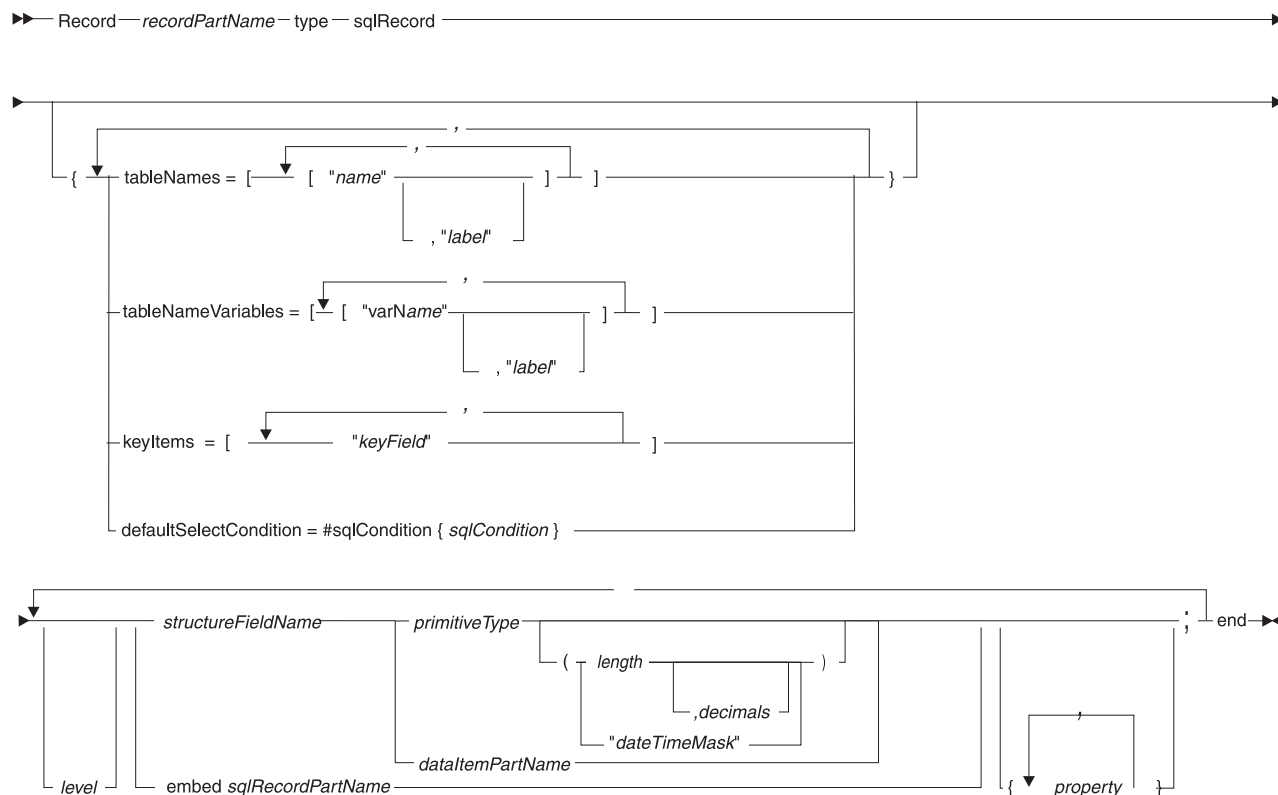
sqlRecord タイプのレコード・パーツを EGL ファイルに宣言します。詳細については、『EGL ソース形式』に説明します。EGL とリレーショナル・データベースの対話方法の概要については、『SQL サポート』を参照してください。

SQL レコード・パーツの例を次に示します。

```
Record mySQLRecordPart type sqlRecord
{
  tableNames = [["mySQLTable", "T1"]],
  keyItems = ["myHostVar01"],
  defaultSelectCondition =
    #sqlCondition{ // #sqlCondition と中括弧の間にスペースを入れない
      myHostVar02 = 4 -- start each SQL comment
                    -- with a double hyphen
    }
}

// SQL レコードの構造には階層がありません。
10 myHostVar01 myDataItemPart01
{
  column = "column01",
  isNullable = no,
  isReadOnly = no
};
10 myHostVar02 myDataItemPart02
{
  column = "column02",
  isNullable = yes,
  isReadOnly = no
};
end
```

SQL レコード・パーツの構文図は、以下のとおりです。



Record *recordPartName* **sqlRecord**

パーツを **sqlRecord** タイプのレコード・パーツとして識別し、名前を指定します。命名の規則については、『命名規則』を参照してください。

tableNames = [{"name", "label"}, ..., {"name", "label"}]

テーブルまたは SQL レコードによってアクセスされるテーブルをリストします。特定のテーブル名変数にラベルを指定する場合、そのラベルは、レコードに関連付けられているデフォルトの SQL ステートメントに含まれます。

テーブル名変数には二重引用符 (") を含めることができますが、その場合は、二重引用符の前にエスケープ文字 (¥) を付ける必要があります。その規則は、例えば、テーブル名が以下の SQL 予約語のいずれかである場合に必要です。

- CALL
- FROM
- GROUP
- HAVING
- INSERT
- ORDER
- SELECT
- SET
- UPDATE
- UNION

- VALUES
- WHERE

これらの名前は、二重引用符で囲む必要があります。唯一のテーブル名が *SELECT* である場合は、例えば、*tableNames* 文節は次のとおりです。

```
tableNames=[["¥SELECT¥"]]
```

SQL 予約語の 1 つが列名として使用される場合にも同様な状態が当てはまります。

tableNameVariables = [{"varName", "label"}, ..., {"varName", "label"}]

1 つ以上のテーブル名変数をリストします。それぞれのテーブル名変数には SQL レコードによってアクセスされるテーブルの名前が含まれます。テーブルの名前は実行時にのみ決定されます。

変数は、ライブラリー名によって修飾することや、添え字を付けることができます。

特定のテーブル名変数にラベルを指定する場合、そのラベルは、レコードに関連付けられているデフォルトの SQL ステートメントに含まれます。

テーブル名変数を単独で、またはテーブル名とともに使用することができますが、テーブル名変数を使用すると、SQL ステートメントの特性は実行時にのみ決定される特性が保証されます。

テーブル名変数には二重引用符 (") を含めることができますが、その場合は、二重引用符の前にエスケープ文字 (¥) を付ける必要があります。

keyItems = ["item", ..., "item"]

特定のレコード項目に関連付けられている列が、データベース表内のキーの一部であるかどうかを指定します。データベース表が複合キーを持つ場合、キーとして定義されたレコード項目の順序は、データベース表のキーとなる列の順序と同じでなければなりません。

defaultSelectCondition = #sqlCondition { sqlCondition }

暗黙の SQL ステートメントの WHERE 文節内の検索基準の一部を定義します。*defaultSelectCondition* の値には、SQL キーワードの WHERE は含まれません。

次の EGL ステートメントのいずれかをコーディングすると、EGL により WHERE 文節が指定された暗黙の SQL ステートメントが提供されます。

- **get**
- **open**
- **execute** (暗黙の SQL DELETE または UPDATE ステートメントを要求する場合に限る)

暗黙の SQL ステートメントは、EGL ソース・コード内に保管されません。これらのステートメントの概要については『SQL サポート』を参照してください。

level

構造体フィールドの階層位置を示す整数。この値を除外する場合、パーツはレコード・パーツです。この値をインクルードする場合、パーツは固定レコード・パーツです。

structureFieldName

構造体フィールドの名前。命名の規則については、『命名規則』を参照してください。

primitiveType

構造体フィールドに割り当てられるプリミティブ型。

length

構造体フィールドの長さ (整数)。構造体項目に基づくメモリー領域の値には、指定された数の文字または数字が含まれます。

decimals

decimals は、数値タイプ (BIN、DECIMAL、NUM、NUMC、または PACF) に対して指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

"dateTimeMask"

INTERVAL 型または TIMESTAMP 型の項目には、*"dateTimeMask"* を指定できます。これは、項目の値の特定の位置に意味 (「年の桁」など) を割り当てるものです。このマスクは、データと一緒に格納されません。

dataItemPartName

宣言されている構造体項目の形式のモデルとして機能する *dataItem* パーツ名を指定します。詳細については、『*typeDef*』を参照してください。

embed *sqlRecordPartName*

sqlRecord タイプのレコード・パーツ名を指定し、そのレコード・パーツの構造体を現在のレコードに組み込みます。組み込まれた構造体項目は一定レベルの階層を現在のレコードに追加しません。詳細については、『*typeDef*』を参照してください。

property

項目のプロパティ。詳細については、『*EGL* プロパティとオーバーライドの概要』に説明します。SQL レコードでは、SQL フィールドのプロパティが特に重要です。

関連する概念

15 ページの『*EGL* プロジェクト、パッケージ、およびファイル』

68 ページの『*EGL* プロパティの概要』

19 ページの『パーツ』

24 ページの『パーツの参照』

141 ページの『レコード・パーツ』

247 ページの『SQL サポート』

30 ページの『*Typedef*』

関連するタスク

812 ページの『*EGL* ステートメントおよびコマンドの構文図』

関連する参照項目

78 ページの『配列』

512 ページの『EGL ソース形式の DataItem パーツ』

532 ページの『EGL ソース形式』

570 ページの『EGL ソース形式の関数パーツ』

578 ページの『EGL ソース形式の索引付きレコード・パーツ』

714 ページの『EGL ソース形式の MQ レコード・パーツ』

725 ページの『命名規則』

37 ページの『プリミティブ型』

783 ページの『EGL ソース形式のプログラム・パーツ』

62 ページの『EGL での変数の参照』

796 ページの『EGL ソース形式の相対レコード・パーツ』

799 ページの『EGL ソース形式のシリアル・レコード・パーツ』

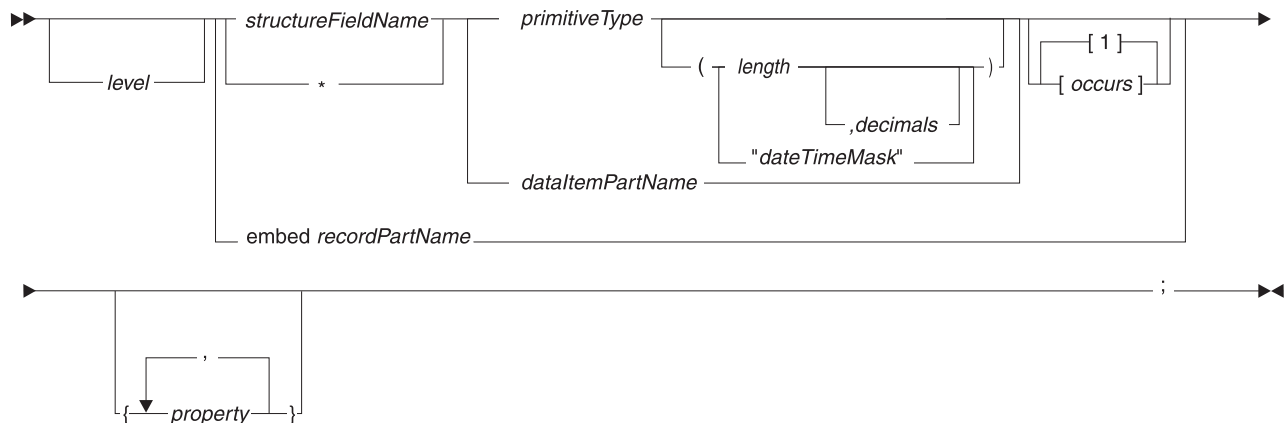
71 ページの『SQL 項目のプロパティ』

EGL ソース形式の構造体フィールド

構造体フィールドの例を次に示します。

```
10 address;  
20 street01 CHAR(20);  
20 street02 CHAR(20);
```

構造体フィールドの構文図は、以下のとおりです。



level

構造体フィールドの階層位置を示す整数。

structureFieldName

構造体フィールドの名前。命名の規則については、『命名規則』を参照してください。

- * 構造体フィールドで充てん文字 を記述することを示します (充てん文字は、名前が重要でないメモリ領域です)。アスタリスクは、メモリ領域を参照する際に有効ではありません。詳細については、『変数および定数の参照』を参照してください。

primitiveType

構造体フィールドに割り当てられるプリミティブ型。

length

構造体フィールドの長さ (整数)。構造体フィールドに基づくメモリー領域の値には、指定された数の文字または数字が含まれます。

decimals

decimals は、数値タイプ (BIN、DECIMAL、NUM、NUMC、または PACF) に対して指定できます。これは、小数点以下の桁数を表す整数です。小数部の桁の最大数は、18 または *length* で宣言された桁数の小さいほうです。小数点は、データとともに保管されません。

"dateTimeMask"

INTERVAL 型または TIMESTAMP 型の項目には、「*dateTimeMask*」を指定できます。これは、フィールド値の特定の位置に意味（「年の桁」など）を割り当てるものです。このマスクは、データと一緒に格納されません。

dataItemPartName

宣言されている構造体フィールドの形式のモデルとして機能する *dataItem* パーツ名を指定します。詳細については、『*typeDef*』を参照してください。

embed *recordPartName*

レコード・パーツ名を指定し、そのレコード・パーツの構造体を現在のレコードに埋め込みます。組み込まれた構造体項目は一定レベルの階層を現在のレコードに追加しません。詳細については、『*typeDef*』を参照してください。

recordPartName

レコード・パーツ名を指定し、そのレコード・パーツの構造体を現在のレコードに組み込みます。ワード *embed* が存在しない場合、レコード構造体は、宣言する構造体フィールドの副構造として組み込まれます。詳細については、『*typeDef*』を参照してください。

occurs

構造体項目の配列内の要素数。デフォルトは 1 で、異なる指定をしない限り、構造体フィールドが配列ではないことを意味します。詳細については、『配列』を参照してください。

property

フィールドのプロパティ。詳細については、『EGL プロパティとオーバーライドの概要』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

68 ページの『EGL プロパティの概要』

関連する参照項目

78 ページの『配列』

725 ページの『命名規則』

37 ページの『プリミティブ型』

62 ページの『EGL での変数の参照』

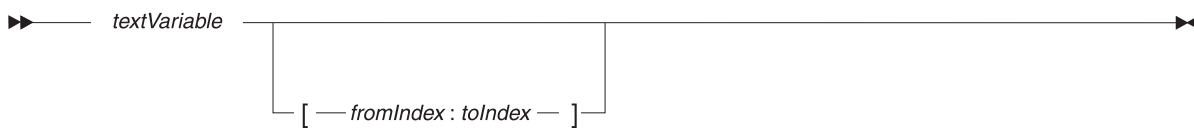
30 ページの『Typedef』

サブストリング

文字フィールドを参照するコンテキストでは、そのフィールド内の文字の順次サブセットであるサブストリングを参照できます。例えば、フィールドの値が *ABCD* である場合に、その 2 番目と 3 番目の文字である *BC* を参照できます。

また、ターゲット・フィールドが CHAR 型、DBCHAR 型、または UNICODE 型の場合には、assignment ステートメントの左側にサブストリングを指定できます。サブストリング領域には値が入力されますが (必要に応じてブランクが埋め込まれます)、割り当てられたテキストはサブストリング領域外に拡張されません (必要に応じて切り捨てられます)。

サブストリング参照の構文は、以下のとおりです。



itemReference

文字または HEX フィールドで、リテラルでないもの。この項目は、システム変数または配列エレメントでもかまいません。

fromIndex

項目内の求める最初の文字。ここで、1 は文字項目の最初の文字を表し、2 は 2 番目の文字を表し、以下同様です。整数に変換される任意の数式を使用できますが、数式に関数呼び出しを含めることはできません。

fromIndex の値はバイト位置を表します。ただし、*itemReference* が DBCHAR 型または UNICODE 型の項目である場合は除きます。その場合、値は 2 バイト文字の位置を表します。

アラビア語やヘブライ語などの双方向言語で作業している場合でも、左端の文字からカウントしてください。

toIndex

項目内の求める最後の文字。ここで、1 は文字項目の最初の文字を表し、2 は 2 番目の文字を表し、以下同様です。整数に変換される任意の数式を使用できますが、数式に関数呼び出しを含めることはできません。

toIndex の値はバイト位置を表します。ただし、*itemReference* が DBCHAR 型または UNICODE 型の項目である場合は除きます。その場合、値は 2 バイト文字の位置を表します。

アラビア語やヘブライ語などの双方向言語で作業している場合でも、左端の文字からカウントしてください。

関連する概念

62 ページの『EGL での変数の参照』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

EGL 関数の構文図

特定の EGL システム関数について説明しているトピックに示される構文図で、各関数仮パラメーターのタイプおよび戻り値があればそのタイプについての詳細を読者に提供しています。関数ライブラリーの名前は、そのトピックの前の個所で指定されています。

以下にサンプル図を示します。

```
StrLib.clip(text STRING in)  
returns (result STRING)
```

この図は、最初に関数の名前を示し、パラメーター仕様のリストを表示しています。それぞれのパラメーター仕様には次の詳細が含まれています。

- パラメーター名、これはユーザーが自由に指定します。この例では、あるパラメーターの名前は *text* となっています。
- パラメーター・タイプ、これは EGL 言語のタイプまたはタイプの組み合わせです。(そのタイプが EGL 言語にない場合は、トピック内でさらに説明が行われています。)この例では、タイプは **STRING** です。
- 『関数パラメーター』で説明されている、修飾子 **in**、**out**、または **inOut**。

パラメーター仕様が中括弧 ([]) で囲まれている場合、そのパラメーターに関連付けられた引数はオプションとなります。仕様が花括弧 ({ }) で囲まれている場合、引数もオプションです。ただし、この場合はすべてが同じタイプの複数の引数を含めることができます。

関数が値を戻す場合、この図には戻り値 というワード、および名前とタイプが括弧付きで表示されます。該当トピックは戻り値を説明する際にその名前を参照しますが、この名前はそれ以外には意味を持ちません。

戻り値文節が中括弧 ([]) で囲まれている場合、戻り値はオプションとなります。

関連する参照項目 560 ページの『関数呼び出し』

564 ページの『関数パラメーター』

813 ページの『EGL ライブラリー ConsoleLib』

860 ページの『EGL ライブラリー J2EELib』

864 ページの『EGL ライブラリー JavaLib』

891 ページの『EGL ライブラリー LobLib』

899 ページの『EGL ライブラリー MathLib』

922 ページの『EGL ライブラリー ReportLib』

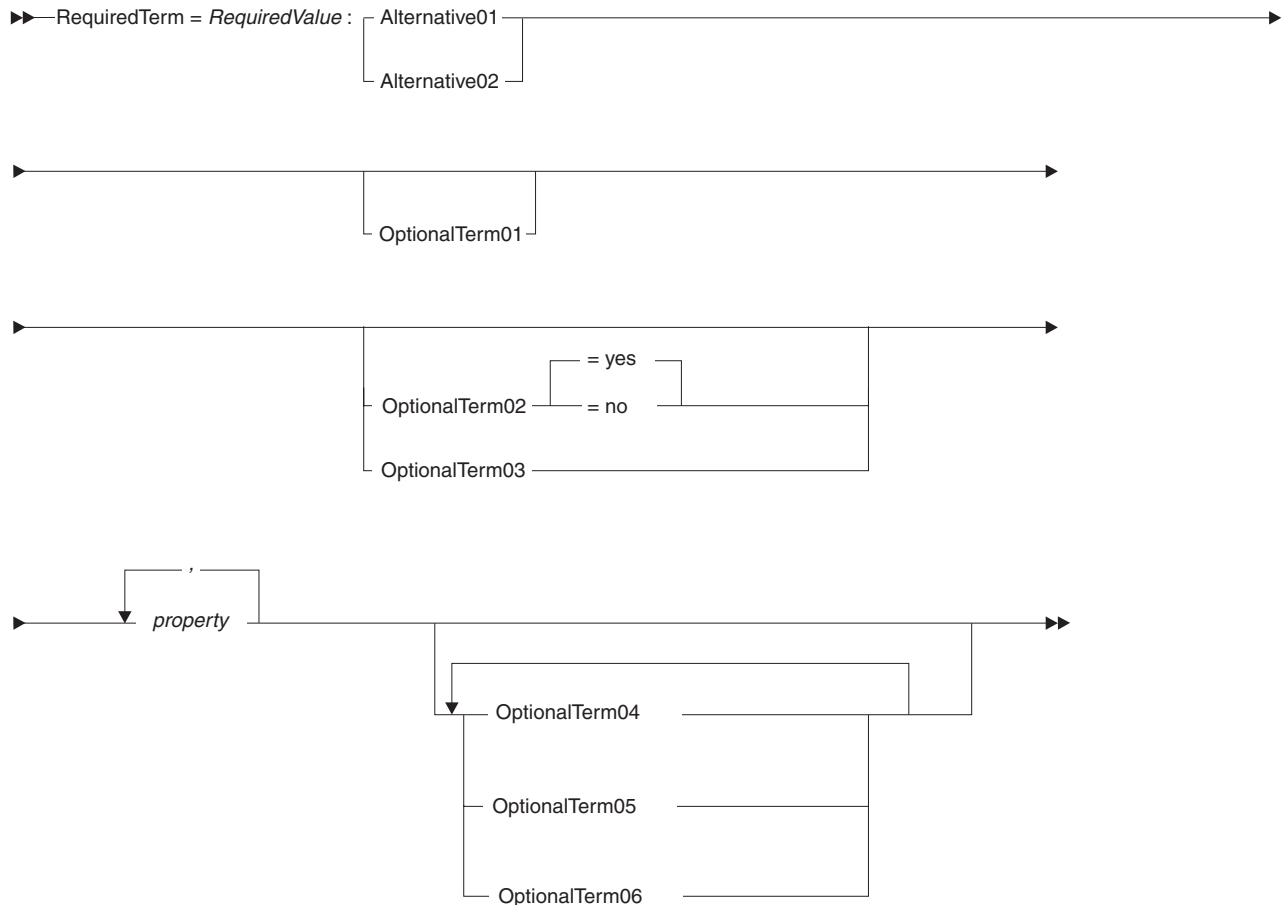
928 ページの『EGL ライブラリー StrLib』

949 ページの『EGL ライブラリー SysLib』

980 ページの『EGL ライブラリー VGLib』

EGL ステートメントおよびコマンドの構文図

IBM 構文図を使用すると、EGL ステートメントまたはビルド・コマンドの構成方法を簡単に理解できます。図の例を以下に示します。



図は、左側の二重矢印 (➤➤) で始まるメインパスの線に従って左から右へ、上から下へ読みます。メインパスについていくときに、従属パスでエントリーを選択することができます。この場合、従属パスに従って左から右へ読み取りを続行します。

この例では、メインパスは 4 つの線分で構成されています。これを理解するのは大切です。メインパスの 2 番目と 3 番目の線分は、矢印 (➤) で始まり、従属情報が含まれています。メインパスの 4 番目の線分も矢印 (➤) で始まり、戻り矢印と従属情報が含まれ、向かい合う矢印 (➤<) で終了しています。

斜体ではない語句 (またはシンボル) は、表示されているとおりに指定する必要があります。この例では、語句 **RequiredTerm** を指定します。一方、斜体の語句は、指定する値のプレースホルダーです。この例では、*RequiredValue* の代わりに以下のいずれかのシンボルを含めます。

```
myVariable  
50  
"Oh!"
```

斜体の語句に関する特定の要件（ストリングまたは数値のいずれが適切かなど）は、構文図自体ではなく、構文図の後のテキストで説明されています。

図に非英数字がある場合は、その文字を構文の一部として入力します。たとえば、*RequiredValue* の値を指定した後、 コロン (:) およびブランクを入力します。

いくつかの語句から選択することが許されている場合、それらの語句はスタックに表示されます。この例では、語句 **Alternative01** または **Alternative02** を指定することができます。

(この場合) スタックにリストされている語句からいずれかを選択しなければならない 場合は、選択肢のいずれか (任意で指定される) がスタックの上の行に表示されます。語句を選択する必要がない場合は、**OptionalTerm01** のように、スタックの最上部の行の下にすべての語句が表示されます。

パス上にあるが、上方に表示される (= **yes** の場合) 値は、値が表示されるスタックのデフォルト値です。この例では、以下のストリングを指定でき、 最初の 2 つは同等であることを示しています。

```
optionalTerm01 = yes

optionalTerm01

optionalTerm01 = no

OptionalTerm02
```

語句の上で左へ戻る矢印は、その語句を繰り返し使用できることを示しています。この例では、*property* の値をコンマで区切って指定します。

垂直スタックの上で左へ戻る矢印は、そのエントリーのリストから任意の順序で選択できることを示しています。この例では、以下の各ストリングは有効 (他のバリエーションでも同様) ですが、必須ではありません。

```
OptionalTerm04 OptionalTerm05
OptionalTerm06
OptionalTerm04 OptionalTerm06 OptionalTerm05
```

システム・ライブラリー

EGL ライブラリー ConsoleLib

コンソール・ライブラリーは、EGL プログラムに consoleUI 機能を提供します。**ConsoleLib** 修飾子 (たとえば、**ConsoleLib.activateWindow**) の使用はオプションです。

関数	説明
activateWindow (<i>window</i>)	指定されたウィンドウをアクティブ・ウィンドウにし、それに応じて ConsoleLib 変数 <i>activeWindow</i> を更新します。
activateWindowByName (<i>name</i>)	指定されたウィンドウをアクティブ・ウィンドウにし、それに応じて ConsoleLib 変数 <i>activeWindow</i> を更新します。

関数	説明
<code>cancelArrayDelete ()</code>	BEFORE_DELETE OpenUI イベント・コード・ブロックの実行時に進行中の現行 <i>delete</i> 操作を終了させます。
<code>cancelArrayInsert ()</code>	BEFORE_INSERT OpenUI イベント・コード・ブロックの実行時に進行中の現行 <i>insert</i> 操作を終了させます。
<code>clearActiveForm ()</code>	すべてのフィールドのディスプレイ・バッファをクリアします。
<code>clearActiveWindow ()</code>	表示されているすべての素材をアクティブ・ウィンドウから除去します。
<code>clearFields ([<i>consoleField</i>{, <i>consoleField</i>})</code>	アクティブな書式で指定されたフィールドのディスプレイ・バッファをクリアします。フィールドが指定されていない場合、その書式のすべてのフィールドがクリアされます。
<code>clearFieldsByName (<i>fieldName</i>{, <i>fieldName</i>})</code>	アクティブな書式にある、指定されたフィールドのディスプレイ・バッファをクリアします。フィールドが指定されていない場合、その書式のすべてのフィールドがクリアされます。
<code>clearForm (<i>consoleForm</i>)</code>	すべてのフィールドのディスプレイ・バッファをクリアします。
<code>clearWindow (<i>window</i>)</code>	表示されているすべての素材を指定のウィンドウから除去します。
<code>clearWindowByName (<i>name</i>)</code>	表示されているすべての素材を指定のウィンドウから除去します。
<code>closeActiveWindow ()</code>	画面からウィンドウをクリアし、そのウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。
<code>closeWindow (<i>window</i>)</code>	画面からウィンドウをクリアし、そのウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。
<code>closeWindowByName (<i>name</i>)</code>	画面からウィンドウをクリアし、そのウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。
<code><i>result</i> = currentArrayCount ()</code>	現在、アクティブな書式と関連付けられている動的配列内の要素数を戻します。
<code><i>result</i> = currentArrayDataLine ()</code>	OpenUI ステートメントの間またはその直後に画面配列の現在行に表示される、プログラム配列内のプログラム・レコードの番号を戻します。
<code><i>result</i> = currentArrayScreenLine ()</code>	OpenUI ステートメントの間に画面配列内の現在の画面レコードの番号を戻します。

関数	説明
<code>displayAtLine (text, line)</code>	アクティブ・ウィンドウ内の指定場所にストリングを表示します。
<code>displayAtPosition (text, line, column)</code>	アクティブ・ウィンドウ内の指定場所にストリングを表示します。
<code>displayError (msg)</code>	エラー・ウィンドウを作成し、表示し、そのウィンドウ内にエラー・メッセージを表示します。
<code>displayFields ([consoleField{, consoleField}])</code>	コンソールに対する書式フィールド値を表示します。
<code>displayFieldsByName (consoleFieldName{, consoleFieldName})</code>	コンソールに対する書式フィールド値を表示します。
<code>displayForm (consoleForm)</code>	アクティブ・ウィンドウに書式を表示します。
<code>displayFormByName (formName)</code>	アクティブ・ウィンドウに書式を表示します。
<code>displayLineMode (text)</code>	書式ウィンドウ・モードではなく、行モードでストリングを表示します。
<code>displayMessage (msg)</code>	アクティブ・ウィンドウ内の指定場所にストリングを表示し、アクティブ・ウィンドウの <i>messageLine</i> 設定を使用して、ストリングを表示する場所を識別します。
<code>drawBox (row, column, depth, width)</code>	指定のロケーションと寸法で、アクティブ・ウィンドウに長方形を描画します。
<code>drawBoxWithColor (row, column, depth, width, Color)</code>	指定のロケーション、寸法、および色で、アクティブ・ウィンドウに長方形を描画します。
<code>result = getKey ()</code>	入力からキーを読み取り、そのキーの整数コードを戻します。
<code>result = getKeyCode (keyName)</code>	ストリング内の指定のキーのキー整数コードを戻します。
<code>result = getKeyName (keyCode)</code>	整数キー・コードを表す名前を戻します。
<code>gotoField (consoleField)</code>	指定の書式フィールドにカーソルを移動します。
<code>gotoFieldByName (name)</code>	指定の書式フィールドにカーソルを移動します。
<code>gotoMenuItem (item)</code>	指定のメニュー項目にメニュー・カーソルを移動します。
<code>gotoMenuItemByName (name)</code>	指定のメニュー項目にメニュー・カーソルを移動します。
<code>hideAllMenuItems ()</code>	現在、表示されているメニューのメニュー項目をすべて隠します。
<code>hideErrorWindow ()</code>	エラー・ウィンドウを非表示にします。
<code>hideMenuItem (item)</code>	ユーザーが選択できないように指定のメニュー項目を非表示にします。

関数	説明
<code>hideMenuItemByName (name)</code>	ユーザーが選択できないように指定のメニュー項目を非表示にします。
<code>result = isCurrentField (consoleField)</code>	カーソルが指定の書式フィールド内にある場合は true を返し、ない場合は false を返します。
<code>result = isCurrentFieldByName (name)</code>	カーソルが指定の書式フィールド内にある場合は true を返し、ない場合は false を返します。
<code>result = isFieldModified (consoleField)</code>	指定された書式フィールドの内容をユーザーが変更した場合は true を、このフィールドが編集されていない場合は false を返します。
<code>result = isFieldModifiedByName (name)</code>	指定された書式フィールドの内容をユーザーが変更した場合は true を、このフィールドが編集されていない場合は false を返します。
<code>result = lastKeyTyped ()</code>	キーボードで最後に押された物理キーの整数コードを返します。
<code>nextField ()</code>	定義されたフィールド移動順序にしたがって、次の書式フィールドにカーソルを移動します。
<code>openWindow (window)</code>	ウィンドウを可視にし、そのウィンドウをウィンドウ・スタックの一番上に追加します。書式がウィンドウに表示されます。
<code>openWindowByName (name)</code>	ウィンドウを可視にし、そのウィンドウをウィンドウ・スタックの一番上に追加します。
<code>openWindowWithForm (Window, form)</code>	ウィンドウを可視にし、そのウィンドウをウィンドウ・スタックの一番上に追加します。ウィンドウが宣言されたときにウィンドウ・サイズが定義されていない場合、指定の書式を保持するために、ウィンドウのサイズが変更されます。
<code>openWindowWithFormByName (windowName, formName)</code>	ウィンドウを可視にし、そのウィンドウをウィンドウ・スタックの一番上に追加します。
<code>previousField ()</code>	定義されたフィールド移動順序にしたがって、前の書式フィールドにカーソルを移動します。
<code>result = promptLineMode (prompt)</code>	行モード 環境でユーザーに対するプロンプト・メッセージを表示します。
<code>scrollDownLines (numLines)</code>	データの先頭方向 (つまり、より小さいレコード索引方向) にデータ・テーブルをスクロールします。
<code>scrollDownPage ()</code>	データの先頭方向 (つまり、より小さいレコード索引方向) にデータ・テーブルをスクロールします。

関数	説明
scrollUpLines (<i>numLines</i>)	データの末尾方向 (つまり、より大きいレコード索引方向) にデータ・テーブルをスクロールします。
scrollUpPage ()	データの末尾方向 (つまり、より大きいレコード索引方向) にデータ・テーブルをスクロールします。
setArrayLine (<i>recordNumber</i>)	指定のプログラム・レコードに選択を移動します。必要に応じて、データ・テーブルがディスプレイ内でスクロールされ、選択されたレコードが見えるようになります。
setCurrentArrayCount (<i>count</i>)	プログラム配列内に存在するレコード数を設定します。 OpenUI ステートメントの前に呼び出す必要があります。
showAllMenuItems ()	ユーザー選択用に、メニュー項目をすべて表示します。
showHelp (<i>helpkey</i>)	EGL プログラムの実行時に ConsoleUI ヘルプ画面を表示します。
showMenuItem (<i>item</i>)	ユーザー選択用に、指定されたメニュー項目を表示します。
showMenuItemByName(<i>name</i>)	ユーザー選択用に、指定されたメニュー項目を表示します。

変数	説明
activeForm	アクティブ・ウィンドウに最も新しく表示される書式。
activeWindow	最上位ウィンドウであり、ウィンドウ名が指定されない場合のウィンドウ操作のターゲットです。
commentLine	コメント・メッセージが表示されるウィンドウ行。
CurrentDisplayAttrs	表示関数を介して表示されるエレメントに適用される設定値。
currentRowAttrs	現在行に適用される属性を強調表示します。
cursorWrap	true の場合、カーソルが折り返して書式の先頭フィールドに進みます。 false の場合、カーソルが書式の最後の入力フィールドから前に移動すると文が終了します。
defaultDisplayAttributes	新規オブジェクトの表示属性 のデフォルト設定値。
defaultInputAttributes	入力操作の表示属性 のデフォルト設定値。

変数	説明
deferInterrupt	true の場合、プログラムは INTR シグナルをキャッチし、 <i>interruptRequested</i> 変数に記録します。その後、プログラムがモニターを担当します。Windows では、論理 INTERUPT キー (デフォルトで CONTROL_C) が押されると、シグナルがシミュレートされます。
deferQuit	true の場合、プログラムは QUIT シグナルをキャッチし、それを <i>interruptRequested</i> 変数に記録します。その後、プログラムがモニターを担当します。Windows では、論理 QUIT キー (デフォルトで CONTROL_¥) が押されると、シグナルがシミュレートされます。
definedFieldOrder	true の場合、上下矢印キーが、トラバースル順序で前または次のフィールドに移動します。 false の場合、画面上で物理的にその方向のフィールドへ上下に移動します。
errorLine	エラー・メッセージが表示されるウィンドウ。
errorWindow	ConsoleUI 画面でエラー・メッセージが表示されるウィンドウ・ロケーション。
errorWindowVisible	true の場合、画面にエラー・ウィンドウが現在表示されています。
formLine	書式が表示されるウィンドウ行。
interruptRequested	INTR シグナルが受信 (またはシミュレート) されたことを示します。
key_accept	OpenUI ステートメントの正常終了のキー。デフォルトのキーは ESCAPE です。
key_deleteLine	画面配列から現在の行を削除するためのキー。デフォルトのキーは F2 です。
key_help	OpenUI ステートメントの間にコンテキスト・ヘルプを表示するためのキー。デフォルトのキーは CTRL_W です。
key_insertLine	画面配列に行を挿入するためのキー。デフォルトのキーは F1 です。
key_interrupt	INTR シグナルをシミュレートするためのキー。デフォルトのキーは CTRL_C です。
key_pageDown	画面配列 (データ・テーブル) 内で順方向にページングするためのキー。デフォルトのキーは F3 です。
key_pageUp	画面配列 (データ・テーブル) 内で逆方向にページングするためのキー。デフォルトのキーは F4 です。

変数	説明
key_quit	QUIT シグナルをシミュレートするためのキー。デフォルトのキーは CTRL_¥ です。
menuLine	メニューが表示されるウィンドウ行。
messageLine	メッセージが表示されるウィンドウ行。
messageResource	リソース・バンドルのファイル名。
promptLine	エラー・メッセージが表示されるウィンドウ行。
quitRequested	QUIT シグナルが受信 (またはシミュレート) されたことを示します。
screen	自動的に定義される、境界線のないデフォルト・ウィンドウ。寸法は、使用可能な表示面の寸法と同じです。
sqlInterrupt	true の場合、ユーザーは処理中の SQL ステートメントに割り込むことができます。 false の場合、ユーザーは OpenUI ステートメントのみに割り込むことができます。 <i>deferInterrupt</i> および <i>deferQuit</i> 変数と組み合わせて使用されます。

activeForm

システム変数 **ConsoleLib.activeForm** は、アクティブ・ウィンドウに最も新しく表示された書式です。

データ型: ConsoleForm

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

activateWindow()

システム関数 **ConsoleLib.activateWindow** は、指定されたウィンドウをアクティブ・ウィンドウにし、変数 *activeWindow* を更新します。

ConsoleLib.activateWindow(*window1* **Window** inOut)

window1

アクティブ化されるウィンドウ。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

activeWindow

システム変数 **ConsoleLib.activeWindow** は、最上位ウィンドウまたは最近アクティブにされたウィンドウです。 **ConsoleLib.activeWindow** は、ウィンドウが指定されない場合のウィンドウ操作のターゲットです。

データ型: Window

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

activateWindowByName()

システム関数 **ConsoleLib.activateWindowByName** は、指定されたウィンドウをアクティブ・ウィンドウにし、それに応じて **consoleLib** 変数 *activeWindow* を更新します。

```
ConsoleLib.activateWindowByName(name STRING in)
```

name

ウィンドウの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

cancelArrayDelete()

システム関数 **ConsoleLib.cancelArrayDelete** は、**BEFORE_DELETE OpenUI** イベント・コード・ブロックの実行時に進行中の現行 *delete* 操作を終了させます。

実行時にこの関数が **OpenUI** ステートメントの有効範囲外で実行される場合、その結果は NULL 操作です。

```
ConsoleLib.cancelArrayDelete( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

cancelArrayInsert()

システム関数 **ConsoleLib.cancelArrayInsert** は、**BEFORE_INSERT OpenUI** イベント・コード・ブロックの実行時に進行中の現行 *insert* 操作を終了させます。 実行時にこの関数が **OpenUI** ステートメントの有効範囲外で実行される場合、その結果は NULL 操作です。

```
ConsoleLib.cancelArrayInsert( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearActiveForm()

システム関数 **ConsoleLib.clearActiveForm** は、すべてのフィールドのディスプレイ・バッファをクリアします。この関数は、バインド済みデータ・エレメントには無効です。つまり、バインド済みデータ・エレメントに保管されているデータはクリアされません。

```
ConsoleLib.clearActiveForm( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearActiveWindow

システム関数 **ConsoleLib.clearActiveWindow** は、表示されているすべての素材をアクティブ・ウィンドウから除去します。これには、現在の書式で表示されている定数情報の消去が含まれます。アクティブ・ウィンドウに境界線がある場合、その境界線は消去されません。この文は、ウィンドウ・スタックの順序にも、またウィンドウ・スタック内でそのウィンドウより上にあるウィンドウにも影響を与えません。

```
ConsoleLib.clearActiveWindow( )
```

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearFields()

システム関数 **ConsoleLib.clearFields** は、指定されたフィールドのディスプレイ・バッファをクリアします。フィールドが指定されていない場合、すべてのフィールドがクリアされます。この関数は、バインド済みデータ・エレメントには無効です。つまり、バインド済みデータ・エレメントに保管されているすべてのデータはクリアされません。

```
ConsoleLib.clearFields(  
  [consoleField1 ConsoleField inOut  
  {, consoleField1 ConsoleField inOut}  
  ] )
```

consoleField1

ConsoleField タイプの変数名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearFieldsByName()

システム関数 **ConsoleLib.clearFieldsByName** は、指定した表示中のフィールドをクリアします。フィールドが指定されていない場合は、すべてのフィールドをクリアします。表示中のフィールドにバインドされている変数は影響を受けません。

```
ConsoleLib.clearFieldsByName(  
    [fieldName STRING in  
    { , fieldName STRING in }])
```

fieldName

ConsoleField の名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearForm()

システム関数 **ConsoleLib.clearForm** は、指定した書式のすべてのフィールドをクリアします。それらのフィールドにバインドされている変数は影響を受けません。

```
ConsoleLib.clearForm(consoleForm ConsoleForm inOut)
```

consoleForm

ConsoleForm 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearWindow()

システム関数 **ConsoleLib.clearWindow** は、表示されているすべての素材を指定のウィンドウから除去します。これには、現在の書式で表示されている定数情報の消去が含まれます。ウィンドウに境界線がある場合、その境界線は消去されません。この文は、ウィンドウ・スタックの順序にも、またウィンドウ・スタック内でそのウィンドウより上にあるウィンドウにも影響を与えません。

```
ConsoleLib.clearWindow(window1 Window inOut)
```

window1

クリアされるウィンドウ。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

clearWindowByName()

システム関数 **ConsoleLib.clearWindowByName** は、表示されているすべての素材を指定のウィンドウから除去します。これには、現在の書式で表示されている定数情報の消去が含まれます。ウィンドウに境界線がある場合、その境界線は消去されません。この文は、ウィンドウ・スタックの順序に影響を与えません。

ActiveWindow 変数は、表示スタック内の最上位ウィンドウを指します。

```
ConsoleLib.cleaWindowByName(name STRING in)
```

name

ウィンドウの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

closeActiveWindow()

システム関数 **ConsoleLib.closeActiveWindow** は、画面からウィンドウをクリアし、クリアされたウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。

ConsoleLib.closeActiveWindow が呼び出された後、**ConsoleLib.openWindow** または **ConsoleLib.openWindowByName** によってウィンドウを再オープンすることはできません。さらに、SCREEN ウィンドウを閉じることは許可されません。

```
ConsoleLib.closeActiveWindow( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

closeWindow()

コンソール・ライブラリー関数 **ConsoleLib.closeWindow** は、画面から指定のウィンドウをクリアし、クリアされたウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。

ConsoleLib.closeWindow が呼び出された後、**ConsoleLib.openWindow** または **ConsoleLib.openWindowByName** によってウィンドウを再オープンすることはできません。さらに、SCREEN ウィンドウを閉じることは許可されません。

```
ConsoleLib.closeWindow(window1 Window inOut)
```

window1

画面上の指定されたウィンドウ・オブジェクト。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

closeWindowByName()

システム関数 **ConsoleLib.closeWindowByName** は、画面から指定のウィンドウをクリアし、閉じたウィンドウに関連したリソースを解放し、以前にアクティブであったウィンドウをアクティブにします。

ConsoleLib.closeWindowByName が呼び出された後、**ConsoleLib.openWindow** または **ConsoleLib.openWindowByName** によってウィンドウを再オープンすることはありません。コンソール・ウィンドウは開いたままになります。

```
ConsoleLib.closeWindowByName(name STRING in)
```

name

ウィンドウの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

commentLine

システム変数 **ConsoleLib.commentLine** は、コメント・メッセージが表示されるウィンドウ行です。

データ型: Integer

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

currentArrayCount()

システム関数 **ConsoleLib.currentArrayCount** は、現行のアクティブ書式と関連する動的配列内のエレメント数を戻します。

この関数は、Informix 4GL で書かれたアプリケーションの移行を支援するためのものであるため、使用しないことをお勧めします。代わりに、『配列』で説明されているように、配列固有の関数 **getSize** を使用します。

```
ConsoleLib.currentArrayCount( )  
returns (result INT)
```

result

エレメントの数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

78 ページの『配列』

813 ページの『EGL ライブラリー ConsoleLib』

currentArrayDataLine()

システム関数 **ConsoleLib.currentArrayDataLine** は、**openUI** 文の間またはその直後に画面配列の現在行に表示される、プログラム配列内のプログラム・レコードの番号を戻します。

```
ConsoleLib.currentArrayDataLine( )  
returns (result INT)
```

result

任意の整数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

currentArrayScreenLine()

システム関数 **ConsoleLib.currentArrayScreenLine** は、**openUI** 文の処理中に画面配列内の現在の画面レコードの番号を戻します。

```
ConsoleLib.currentArrayScreenLine( )  
returns (result INT)
```

result

任意の整数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

currentDisplayAttrs

システム変数 **ConsoleLib.currentDisplayAttrs** は、変数が設定された後に表示されるテキストの表示特性を指定します。

PresentationAttributes 型の変数は、フィールド色、輝度、および強調表示を含みます。詳細については、『ConsoleUI パーツおよび関連した変数』を参照してください。

データ型: PresentationAttributes

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

192 ページの『ConsoleUI パーツおよび関連変数』

currentRowAttrs

システム変数 **ConsoleLib.currentRowAttrs** は、画面配列の現在行に適用される強調表示属性です。

`PresentationAttributes` 型の変数には、フィールド色、輝度、および強調表示が含まれます。詳細については、『*ConsoleUI* パーツおよび関連した変数』を参照してください。

データ型: `PresentationAttributes`

関連する参照項目

`currentDisplayAttrs`

813 ページの『EGL ライブラリー `ConsoleLib`』

cursorWrap

システム変数 `ConsoleLib.cursorWrap` は、カーソルが折り返して書式の先頭フィールドに進むかどうかを指定します。カーソルが折り返して書式の先頭フィールドに進む場合、この関数は **true** を返します。カーソルが書式の最後の入力フィールドから前に移動すると文が終了する場合は、**false** を返します。

データ型: `Boolean`

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

defaultDisplayAttributes

システム変数 `ConsoleLib.defaultDisplayAttributes` は、変数内の `PresentationAttributes` に対して使用される設定を含みます。

`PresentationAttributes` 型の変数には、フィールド色、輝度、および強調表示が含まれます。詳細については、『*ConsoleUI* パーツおよび関連した変数』を参照してください。

データ型: `PresentationAttributes`

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

defaultInputAttributes

システム変数 `ConsoleLib.defaultInputAttributes` には、入力操作の表示属性のデフォルト設定値が入っています。

`PresentationAttributes` 型の変数には、フィールド色、輝度、および強調表示が含まれます。詳細については、『*ConsoleUI* パーツおよび関連した変数』を参照してください。

データ型: `PresentationAttributes`

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

deferInterrupt

コンソール UI ライブラリー変数 **ConsoleLib.deferInterrupt** は、**INTERRUPT** シグナルを受け取るときのアプリケーションの振る舞いを識別します。結果が **true** である場合、プログラムは **INTR** シグナルをキャッチし、*interruptRequested* 変数に記録します。その後、プログラムがモニターを担当します。Windows では、論理 **INTERRUPT** キー (デフォルトで **CONTROL_C**) が押されると、シグナルがシミュレートされます。結果が **false** である場合は、**interrupt** キーが押されるとプログラムが終了します。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

deferQuit

システム変数 **ConsoleLib.deferQuit** が **true** である場合、プログラムは **QUIT** シグナルをキャッチし、*quitRequested* 変数に記録します。その後、プログラムがモニターを担当します。Windows では、論理 **QUIT** キー (デフォルトで **CONTROL_¥**) が押されると、シグナルがシミュレートされます。**false** の場合、quit シグナルを受信すると、アプリケーションが終了します。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

definedFieldOrder

コンソール UI 変数 **ConsoleLib.definedFieldOrder** は、書式を使用した入力時の上下矢印キーの振る舞いを決定します。**true** の場合、上下矢印キーを使用すると、カーソルは、定義された順にフィールドを横断します。**false** の場合、カーソルは、画面上のフィールドの物理的な配置にしたがって上下に移動します。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayAtLine()

システム関数 **ConsoleLib.displayAtLine** は、アクティブ・ウィンドウ内の指定の場所にストリングを表示します。

```
ConsoleLib.displayAtLine(  
    text STRING in  
    line INT in)
```

text

表示するストリング。

line

ストリングの表示に使用される行番号。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayAtPosition()

システム関数 **ConsoleLib.displayAtPosition** は、アクティブ・ウィンドウ内の指定の場所にストリングを表示します。

```
ConsoleLib.displayAtPosition(  
    text STRING in,  
    line INT in,  
    column INT in)
```

text

表示するストリング。

line

ストリングが表示される位置の行番号。

column

ストリングの表示に使用される列の番号。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayError()

システム関数 **ConsoleLib.displayError** は、エラー・ウィンドウを作成および表示させ、そのウィンドウ内にエラー・メッセージを表示します。このエラー・ウィンドウは、*hideErrorWindow()* を呼び出すか、キーを押して閉じるまで、他のすべてのウィンドウ上で浮動します。該当する場合は、ターミナル・ベルがアクティブになります。

```
ConsoleLib.displayError(msg STRING in)
```

msg

表示するエラー・メッセージ・

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayFields()

システム関数 **ConsoleLib.displayFields** は、コンソールに書式フィールド値を表示します。データ・エレメントがフィールドにバインドされる場合、データはそれらのエレメントから検索され、書式フィールドで指定された規則にしたがってフォーマ

ット設定されます。アンバインドされた書式フィールドの場合、**ConsoleField.value** フィールドにアクセスすると、フィールドに対してデータを直接設定することができます。

```
ConsoleLib.displayFields(  
  [consoleField1 ConsoleField in  
  {, consoleField1 ConsoleField in}  
  ])
```

consoleField1

ConsoleField タイプの変数名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayFieldsByName()

システム関数 **ConsoleLib.displayFieldsByName** は、コンソールに書式フィールド値を表示します。データ・エレメントがフィールドにバインドされる場合、データはそれらのエレメントから検索され、書式フィールドで指定された規則にしたがってフォーマット設定されます。アンバインドされた書式フィールドの場合、**ConsoleField.value** フィールドにアクセスすると、これらのフィールドに対してデータを直接設定することができます。

```
ConsoleLib.displayFieldsByName(  
  consoleFieldName1 ConsoleFieldName in  
  {, consoleFieldName1 ConsoleFieldName in})
```

consoleFieldName1

表示するフィールドの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayForm()

システム関数 **ConsoleLib.displayForm** は、アクティブ・ウィンドウに指定した書式を表示します。

```
ConsoleLib.displayForm(consoleForm ConsoleForm in)
```

consoleForm

ConsoleForm 型の変数の名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayFormByName()

システム関数 **ConsoleLib.displayFormByName** は、アクティブ・ウィンドウに指定された書式を表示します。

ConsoleLib.displayFormByName(*formName* **STRING** in)

formName

ConsoleForm の名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayLineMode()

システム関数 **ConsoleLib.displayLineMode** は、書式/ウィンドウ・モードではなく、**行モード**で指定のストリングを表示します。ストリング値は、実行中のシステム上の標準出力 ロケーションに送信されます。折り返しおよびスクロールなどの表示特性はすべて、標準出力インターフェースの責任で扱われます。

ConsoleLib.displayLine(*text* **STRING** in)

text

表示するストリング。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

displayMessage()

システム関数 **ConsoleLib.displayMessage** は、アクティブ・ウィンドウのメッセージ行にストリングを表示します。この関数は、アクティブ・ウィンドウの *MessageLine* 設定を使用して、ストリングを表示する場所を認識します。

ConsoleLib.displayMessage(*msg* **STRING** in)

msg

表示するメッセージ・

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

drawBox()

システム関数 **ConsoleLib.drawBox** は、最初の 2 つの整数を行、列、次の 2 つの整数を深さ、幅 として、アクティブ・ウィンドウの左上隅に長方形を描画します。行と列は、現行ウィンドウの左上隅を基準にします。

```
ConsoleLib.drawBox(  
    row INT in,  
    column INT in,  
    depth INT in,  
    width INT in)
```

row

ウィンドウの左上隅に対応する行番号。

column

ウィンドウの左上隅に対応する列番号。

depth

ボックスの深さ、または高さ。

width

ボックスの幅。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

drawBoxWithColor()

システム関数 **ConsoleLib.drawBoxWithColor** は、最初の 2 つの整数を行、列、次の 2 つの整数を深さ、幅として、アクティブ・ウィンドウの左上隅に長方形を描画します。行と列は、現行ウィンドウの左上隅を基準にします。長方形は、指定の色で描画されます。

```
ConsoleLib.drawBoxWithColor(  
    row INT in,  
    column INT in,  
    depth INT in,  
    width INT in,  
    color enumerationColorKind in)
```

row

ウィンドウの左上隅に対応する行番号。

column

ウィンドウの左上隅に対応する列番号。

depth

ボックスの深さ、または高さ。

width

ボックスの幅。

color

ボックスの色。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

errorLine

コンソール UI 変数 **ConsoleLib.errorLine** は、エラー・メッセージが **ConsoleUI** 画面に表示される行ロケーションを制御します。

データ型: INT

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

errorWindow

システム変数 **ConsoleLib.errorWindow** は、**ConsoleLib.displayError()** からのエラー・メッセージが表示されるウィンドウです。

データ型: Window

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

828 ページの『displayError()』

errorWindowVisible

コンソール UI 変数 **ConsoleLib.errorWindowVisible** は、エラー・メッセージ・ウィンドウの状況を識別します。 **true** の場合、ウィンドウは可視です。 **false** の場合、ウィンドウは不可視です。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

formLine

システム変数 **ConsoleLib.formLine** は、ウィンドウに書式が表示されるデフォルトの行ロケーションです。ウィンドウが開いたときのウィンドウのプロパティに影響を与えます。

データ型: INT

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

getKey()

システム関数 **ConsoleLib.getKey** は、キーが押されるのを待って、押された物理キーの整数コードを返します。この関数は、入力からキーを読み取ります。コードは、通常文字用の単純なものであり、ユニコード・コード・ポイントです。結果を **getKeyCode(String keyname)** によって戻された値と比較することによって、結果は移植可能な方法で解釈されます。

```
ConsoleLib.getKey( )  
returns (result INT)
```

result

押されたキーを表す整数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

832 ページの『getKey()』

getKeyCode()

システム関数 **ConsoleLib.getKeyCode** は、指定されたキー名のキー整数コードを戻します。

```
ConsoleLib.getKeyCode(keyName STRING in)  
returns (result INT)
```

result

キー名を表す整数。

keyName

対応するキー・コードが計算される論理キーまたは物理キーの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

getKeyName()

システム関数 **ConsoleLib.getKeyName** は、整数キー・コードを表すキーの名前を戻します。

```
ConsoleLib.getKeyName(keyCode INT in)  
returns (result STRING)
```

result

整数キー・コードのキーの名前。

keyCode

キーの整数コード。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

gotoField()

システム関数 **ConsoleLib.gotoField** は、指定の書式フィールドにカーソルを移動します。この関数は、コンソール書式で動作する **OpenUI** ステートメントで有効です。

```
ConsoleLib.gotoField(consoleField1 ConsoleField in)
```

consoleField1

カーソルの移動先である `ConsoleField` タイプの変数名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

gotoFieldByName()

システム関数 **`ConsoleLib.gotoFieldByName`** は、指定の書式フィールドにカーソルを移動します。この関数は、コンソール書式で動作する **`openUI`** ステートメントで有効です。

```
ConsoleLib.gotoFieldByName(name STRING in)
```

name

カーソルを移動する先のフィールドの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

gotoMenuItem()

システム関数 **`ConsoleLib.gotoMenuItem`** は、指定のメニュー項目にメニュー・カーソルを移動します。この関数が呼び出されると、指定のメニュー項目が選択されます。

```
ConsoleLib.gotoMenuItem(item MenuItem in)
```

item

カーソルを移動する先のメニュー項目。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー `ConsoleLib`』

gotoMenuItemByName()

システム関数 **`ConsoleLib.gotoMenuItemByName`** は、指定のメニュー項目にメニュー・カーソルを移動します。この関数が呼び出されると、指定のメニュー項目が選択されます。

```
ConsoleLib.gotoMenuItemByName(name STRING in)
```

name

カーソルを移動する先のメニュー項目名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

hideAllMenuItems()

システム関数 **ConsoleLib.hideAllMenuItems** は現在表示されているメニューのすべてのメニュー項目を非表示にします。

```
ConsoleLib.hideAllMenuItems( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

hideErrorWindow()

システム関数 **ConsoleLib.hideErrorWindow** は、エラー・ウィンドウを非表示にします。

```
ConsoleLib.hideErrorWindow( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

hideMenuItem()

システム関数 **ConsoleLib.hideMenuItem** は、ユーザーが選択できないように指定のメニュー項目を非表示にします。デフォルトでは、すべてのメニュー項目が表示されます。非表示の項目は、キー・ストロークによってアクティブになりません。

```
ConsoleLib.hideMenuItem(item MenuItem in)
```

item

非表示にされるメニュー項目。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

hideMenuItemByName()

システム関数 **ConsoleLib.hideMenuItemByName** は、ユーザーが選択できないように指定のメニュー項目を非表示にします。デフォルトでは、すべてのメニュー項目が表示されます。非表示の項目は、キー・ストロークによってアクティブになりません。

```
ConsoleLib.hideMenuItemByName(name STRING in)
```

name

非表示にされるメニュー項目の名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

interruptRequested

コンソール UI 変数 **ConsoleLib.interruptRequested** は、**INTR** シグナルが受信またはシミュレートされたかどうかを指定します。 **true** の場合、**INTR** シグナルは受信されました。 **false** の場合、**INTR** シグナルは受信されていません。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

isCurrentField()

システム関数 **ConsoleLib.isCurrentField** は、カーソルがフィールド内にある場合は **yes** を返し、カーソルがフィールド内にはない場合は **no** を返します。 この関数は、arrayDictionary で動作する **OpenUI** ステートメントで有効です。

```
ConsoleLib.isCurrentField(consoleField1 ConsoleField in)
returns (result BOOLEAN)
```

result

カーソルが指定の書式フィールド内にある場合は **true** を返し、ない場合は **false** を返します。

consoleField1

ConsoleField タイプの変数名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

isCurrentFieldByName()

システム関数 **ConsoleLib.isCurrentFieldByName** は、カーソルがフィールド内にある場合は **yes** を返し、ない場合は **no** を返します。

この関数は、コンソール書式で動作する **OpenUI** ステートメントで有効です。

```
ConsoleLib.isCurrentFieldByName(name STRING in)
returns (result BOOLEAN)
```

result

カーソルが指定の書式フィールド内にある場合は **true** を返し、ない場合は **false** を返します。

name

ConsoleField 名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

isFieldModified()

システム関数 **ConsoleLib.isFieldModified** は、**OpenUI** 書式/フィールドについて、現在の **OpenUI** 文の処理中にフィールドが変更されたかどうかを識別します。**OpenUI** screenarray (arrayDictionary) の場合、現在の行内のフィールドが、カーソルがその行に入ってから以降変更されたかどうかを戻します。

この関数は、フィールドを変更するコマンドで有効であり、**BEFORE_OPENUI** 文節に表示される文の効果を登録しません。フィールドにタッチ済みのマークを付けることなく、これらの文節でフィールドに値の割り当てを行うことができます。

```
ConsoleLib.isFieldModified(consoleFiled1 ConsoleField in)
returns (result BOOLEAN)
```

result

指定された書式フィールドが変更されている場合は **true** を戻し、変更されていない場合には **false** を戻します。

consoleFiled1

ConsoleField タイプの変数名。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

isFieldModifiedByName()

システム関数 **ConsoleLib.isFieldModifiedByName** は、指定されたフィールドの内容が変更されたかどうかを識別します。ユーザーがフィールドの内容を変更した場合、**ConsoleLib.isFieldModifiedByName** は **yes** を戻し、ユーザーがフィールドの内容を変更しなかった場合は **no** を戻します。

この関数は、フィールドを変更するコマンドで有効であり、**BEFORE_OPENUI** 文節に表示される文の効果を登録しません。フィールドにタッチ済みのマークを付けることなく、これらの文節でフィールドに値の割り当てを行うことができます。

```
ConsoleLib.isFieldModifiedByName(name STRING in)
returns (result BOOLEAN)
```

result

指定された書式フィールドが変更されている場合は **true** を戻し、変更されていない場合には **false** を戻します。

name

ConsoleField 名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_accept

システム変数 **ConsoleLib.key_accept** は、**OpenUI** ステートメントを正常終了させるキーです。デフォルトのキーは **Esc** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_deleteLine

システム変数 **ConsoleLib.key_deleteLine** は、コンソール書式の arrayDictionary から現在の行を削除するためのキーです。デフォルトのキーは **F2** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_help

システム変数 **ConsoleLib.key_help** は、**OpenUI** ステートメントの間にコンテキスト・ヘルプを表示するためのキーです。デフォルトのキーは **CRTL_W** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_insertLine

システム変数 **ConsoleLib.key_insertLine** は、consoleForm 上の arrayDictionary に行を挿入するために使用するキー・ストロークを識別します。デフォルトのキーは **F1** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_interrupt

システム変数 **ConsoleLib.key_interrupt** は、割り込みをシミュレートするためのキーです。デフォルトのキーは **CTRL_C** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_pageDown

システム変数 **ConsoleLib.key_pageDown** は、コンソール書式の arrayDictionary で順方向にページ付けをするキーです。デフォルトのキーは **F3** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_pageUp

システム変数 **ConsoleLib.key_pageUp** は、コンソール書式の arrayDictionary で逆方向にページ付けをするキーです。デフォルトのキーは **F4** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

key_quit

システム変数 **ConsoleLib.key_quit** は、ユーザー入力の検証を行わずにプログラムを終了するためのキーです。デフォルトのキーは **CTRL_¥** です。

データ型: CHAR(32)

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

lastKeyTyped()

システム関数 **ConsoleLib.lastKeyTyped** は、キーボードで最後に押された物理キーの整数コードを戻します。

```
ConsoleLib.lastKeyTyped( )  
returns (result INT)
```

result

最後に押されたキーを表す整数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

menuLine

システム変数 **ConsoleLib.menuLine** には、ウィンドウにメニューが表示される行ロケーションが入っています。ウィンドウが開いたときのウィンドウのプロパティーに影響を与えます。

データ型: INT

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

messageLine

システム変数 **ConsoleLib.messageLine** は、メッセージが表示されるウィンドウ・ロケーションです。

データ型: INT

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

messageResource

システム変数 **ConsoleLib.messageResource** は、ヘルプおよびその他のメッセージがロードされる元のリソース・バンドルのファイル名です。この変数が値を持たない場合、EGL ランタイムは Java ランタイムのプロパティー **vgj.messages.file** で識別されるファイルを検査します。

データ型: CHAR(255)

関連する概念

811 ページの『EGL 関数の構文図』

189 ページの『コンソール・ユーザー・インターフェース』

関連する参照項目

192 ページの『ConsoleUI パーツおよび関連変数』

813 ページの『EGL ライブラリー ConsoleLib』

584 ページの『Java ランタイム・プロパティー (詳細)』

nextField()

システム関数 **ConsoleLib.nextField** は、定義されたフィールド移動順序にしたがって、次の書式フィールドにカーソルを移動します。この関数は、コンソール書式で動作する **openUI** ステートメントで有効です。

```
ConsoleLib.nextField( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

openWindow()

システム関数 **ConsoleLib.openWindow** は、ウィンドウを可視にして、そのウィンドウをウィンドウ・スタックの一番上に追加します。

```
ConsoleLib.openWindow(window1 Window inOut)
```

window1

Window 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

openWindowByName()

システム関数 **ConsoleLib.openWindowByName** は、ウィンドウを可視にして、そのウィンドウをウィンドウ・スタックの一番上に追加します。

```
ConsoleLib.openWindowByName(name STRING in)
```

name

ウィンドウの名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

openWindowWithForm()

システム関数 **ConsoleLib.openWindowWithForm** は、ウィンドウを可視にして、ウィンドウ・スタックの一番上に追加し、そのウィンドウに書式を表示します。このウィンドウは、書式に合わせてサイズ変更されます。

```
ConsoleLib.openWindowWithForm(  
    window1 Window inOut,  
    form ConsoleForm in)
```

window1

Window 型の変数。

form

ConsoleForm 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

openWindowWithFormByName()

システム関数 **ConsoleLib.openWindowWithFormByName** は、ウィンドウをアクティブにして、可視にし、指定したコンソール書式を表示します。このウィンドウは、書式に合わせてサイズ変更されます。

```
ConsoleLib.openWindowWithFormByName(  
    windowName STRING in,  
    formName STRING in)
```

windowName

ウィンドウの名前フィールドの値。

formName

ConsoleForm の名前フィールドの値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

previousField()

システム関数 **ConsoleLib.previousField** は、定義されたフィールド・タブ順序にしたがって、前の書式フィールドにカーソルを移動します。

```
ConsoleLib.previousField( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

promptLine

システム変数 **ConsoleLib.promptLine** は、ウィンドウでプロンプトが表示されるデフォルト行です。ウィンドウが開いたときのウィンドウのプロパティに影響を与えます。

データ型: INT

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

promptLineMode()

システム関数 **ConsoleLib.promptLineMode** は、行モードでストリングを表示してユーザー入力を待ち、ユーザーが **Enter** を押したときに実行されます。

```
ConsoleLib.promptLineMode(message String in)  
returns (result STRING)
```

result

ユーザー入力。

message

表示する語句。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

quitRequested

システム変数 **ConsoleLib.quitRequested** は、**QUIT** シグナルが受信 (またはシミュレート) されたことを示します。**true** の場合、**QUIT** シグナルが受信されています。**false** の場合、**QUIT** シグナルは受信されませんでした。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

screen

システム変数 **ConsoleLib.screen** は、境界線のないデフォルト・ウィンドウを自動的に定義します。画面の寸法は、使用可能な表示面の寸法と同じです。

データ型: Window

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

scrollDownLines()

システム関数 **ConsoleLib.scrollDownLines** は、表示中のデータをデータの下部に向かってスクロールします。

```
ConsoleLib.scrollDownLines(numLines INT in)
```

numLines

下方にスクロールする行数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

scrollDownPage()

システム関数 **ConsoleLib.scrollDownPage** は、表示中のデータをデータの下部に 1 ページ分スクロールします。

```
ConsoleLib.scrollDownPage( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

scrollUpLines()

システム関数 **ConsoleLib.scrollUpLines** は、表示中のデータをデータの上部に向かってスクロールします。

ConsoleLib.scrollUpLines(*numLines* INT in)

numLines

スクロールアップする行数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

scrollUpPage()

システム関数 **ConsoleLib.scrollUpPage** は、表示中のデータをデータの上部に向かって 1 ページ分スクロールします。

ConsoleLib.scrollUpPage()

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

setArrayLine()

システム関数 **ConsoleLib.setArrayLine** は、指定のプログラム・レコードに選択を移動します。必要に応じて、データがスクロールされ、選択されたレコードが見えるようになります。

ConsoleLib.setArrayLine(*recordNumber* INT を含む)

recordNumber

選択するレコード。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

setCurrentArrayCount()

システム関数 **ConsoleLib.setCurrentArrayCount** は、表示中の *arrayDictionary* にバウンドする動的配列の行数を指定します。この機能は、*arrayDictionary* を使用する **openUI**ステートメントを実行する前に起動した場合にのみ有効です。

ConsoleLib.setCurrentArrayCount(*count* INT in)

count

openUI ステートメント開始時の配列エントリーの数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

669 ページの『openUI』

showAllMenuItems()

システム関数 **ConsoleLib.showAllMenuItems** は現在表示されているメニューのすべてのメニュー項目を表示します。

```
ConsoleLib.showAllMenuItems( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

showHelp()

システム関数 **ConsoleLib.showHelp** は、ヘルプ・メッセージを表示します。ストリング引数は、**ConsoleLib.messageResource** フィールドで構成されるリソース・バンドル内のメッセージのキーです。

```
ConsoleLib.showHelp(helpKey STRING in)
```

helpKey

ヘルプ・メッセージを見つけるためにテキストをルックアップするキー。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

showMenuItem()

システム関数 **ConsoleLib.showMenuItem** は、ユーザーが選択できるように、指定のメニュー項目を表示します。デフォルトでは、すべてのメニュー項目が表示されます。

```
ConsoleLib.showMenuItem(item MenuItem in)
```

item

表示するメニュー項目。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

showMenuItemByName

システム関数 **ConsoleLib.showMenuItemByName** は、ユーザーが選択できるように、指定のメニュー項目を表示します。デフォルトでは、すべてのメニュー項目が表示されます。

記号 **ConsoleLib.showMenuItemByName**(*name* **STRING** *in*)

name

MenuItem の名前フィールドの値。

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

sqlInterrupt

システム変数 **ConsoleLib.sqlInterrupt** が **yes** である場合、ユーザーは処理中の SQL ステートメントに割り込むことができます。 **no** の場合、ユーザーは **OpenUI** ステートメントのみに割り込むことができます。変数 **sqlInterrupt** は、*deferInterrupt* および *deferQuit* 変数と組み合わせて使用されます。

データ型: Boolean

関連する参照項目

813 ページの『EGL ライブラリー ConsoleLib』

EGL ライブラリー ConverseLib

Converse ライブラリーは、以下の表に示す関数を提供します。

関数	説明
<code>clearScreen ()</code>	画面をクリアする。これは、プログラムがテキスト・アプリケーションで <code>converse</code> ステートメントを発行する前に役立ちます。
<code>displayMsgNum (msgNumber)</code>	プログラムのメッセージ・テーブルから値を取り出す。 converse 、 display 、 print 、または show ステートメントによって次に書式が表示されるときに、メッセージが表示されます。
<code>result = fieldInputLength (textField)</code>	最後にテキスト書式が提示されたときに入力フィールドにユーザーが入力した文字数を返す。この数には、文頭および末尾の空白または NULL は含まれません。
<code>pageEject ()</code>	印刷書式出力を次ページの先頭に進める。これは、プログラムが <code>print</code> ステートメントを実行する前に役立ちます。

関数	説明
validationFailed (<i>msgNumber</i>)	<ul style="list-style-type: none"> テキスト・アプリケーションのフィールド妥当性検査関数で呼び出された場合、ConverseLib.validationFailed は、すべての検証関数が処理された後、受け取ったテキスト書式を再表示する。最後に呼び出された ConverseLib.validationFailed は、表示するメッセージを決定します。 検証関数の外部で呼び出された場合、ConverseLib.validationFailed は、converse、display、print、または show ステートメントによって次に書式が表示されるときに、指定されたメッセージを表示する。この場合の振る舞いは、ConverseLib.displayMsgNum の振る舞いに似ています。

clearScreen()

システム関数 **ConverseLib.clearScreen** は画面を消去します。この関数は、プログラムがテキスト・アプリケーション内で **converse** ステートメントを発行する前に役立ちます。

ConverseLib.clearScreen()

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

616 ページの『converse』

846 ページの『EGL ライブラリー ConverseLib』

displayMsgNum()

システム関数 **ConverseLib.displayMsgNum** は、プログラムのメッセージ・テーブルから値を検索します。 **converse**、**display**、**print**、または **show** ステートメントによって次に書式が表示されるときに、検索されたメッセージが表示されます。

可能な場合、メッセージは書式自体に (書式プロパティ **msgField** が参照しているフィールドに) 表示されます。 書式プロパティ **msgField** に値が存在しない場合は、メッセージが別のモーダル画面または印刷可能ページに表示されてから、書式が表示されます。

ConverseLib.displayMsgNum が唯一の引数として取る値は、プログラムのメッセージ・テーブル の最初の列にある各セルと比較されます。メッセージ・テーブルは、プログラムの **msgTablePrefix** プロパティが参照しているデータ・テーブルです。その関数によって検索されたメッセージは同じ行の 2 番目の列に格納されます。

ConverseLib.displayMsgNum(msgNumber INT in)

msgNumber

メッセージは番号別にメッセージ・テーブルから検索されます。引数は、整数リテラル、またはプリミティブ型の `SMALLINT` か `INT`、または `BIN` に相当する型の項目でなければなりません。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

846 ページの『EGL ライブラリー `ConverseLib`』

fieldInputLength()

システム関数 **`ConverseLib.fieldInputLength`** は、テキスト書式が最後に表示された時の入力フィールドにユーザーが入力した文字数を返します。この数には、文頭および末尾のブランクまたは `NULL` は含まれません。

フィールドが当初定義された状態である場合、この機能は長さ 0 を返します。例えば、フィールドに *value* プロパティが含まれていて、実行中にこれが変更されなかった場合、長さは 0 として計算されます。*set form initial* ステートメントは、このフィールドを当初の定義状態にリセットします。フィールドが当初の定義状態でない場合、長さは最後の `converse` 文での表示または入力に従って計算されます。

```
ConverseLib.fieldInputLength(textField TestFormField in)  
returns(result INT)
```

result

ユーザーが入力した文字数。

textField

テキスト・フィールドの名前。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

846 ページの『EGL ライブラリー `ConverseLib`』

pageEject()

システム関数 **`ConverseLib.pageEject`** は、印刷書式出力を次ページの先頭に進めます。これは、プログラムが `print` ステートメントを発行する前に役に立ちます。

```
ConverseLib.pageEject( )
```

印刷についてのその他の詳細は、『印刷書式』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

166 ページの『印刷書式』

関連する参照項目

846 ページの『EGL ライブラリー `ConverseLib`』

681 ページの『`print`』

validationFailed()

システム関数 **ConverseLib.validationFailed** は、次の 2 つの方法のいずれかで使用します。

- テキスト・アプリケーションのフィールド妥当性検査関数で呼び出された場合、**ConverseLib.validationFailed** は、すべての検証関数が処理された後、受け取ったテキスト書式を再表示します。最後に呼び出された **ConverseLib.validationFailed** は、表示するメッセージを決定します。

可能な場合、メッセージは書式自体に (書式プロパティ **msgField** が参照しているフィールドに) 表示されます。書式プロパティ **msgField** に値が存在しない場合は、メッセージが別のモーダル画面に表示された後、書式が表示されます。

- 検証関数の外部で呼び出された場合、**ConverseLib.validationFailed** は、**converse**、**display**、**print**、または **show** ステートメントによって次に書式が表示されるときに、指定されたメッセージを表示します。この場合の振る舞いは、**ConverseLib.displayMsgNum** の振る舞いに似ています。

いずれの場合も、**ConverseLib.validationFailed** に割り当てられた値は システム変数 **ConverseVar.validationMsgNum** に格納されます。

ConverseLib.validationFailed(*[msgNumber* INT *in]*)

msgNumber

表示するメッセージの数。引数は、整数リテラル、またはプリミティブ型の SMALLINT か INT、または BIN に相当する型の項目でなければなりません。この数字は、プログラムのメッセージ・テーブルの最初の列にある各セルと比較されます。メッセージ・テーブルは、プログラムの **msgTablePrefix** プロパティが参照しているデータ・テーブルです。検索されたメッセージは同じ行の 2 番目の列に格納されます。

デフォルトのメッセージ番号は 9999 です。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

847 ページの『displayMsgNum()』

991 ページの『validationMsgNum』

846 ページの『EGL ライブラリー ConverseLib』

EGL ライブラリー DateTimeLib

日時システム変数を使用すると、次の表に示すさまざまな形式でシステム日時または日付/時刻を取得できます。

システム変数	説明
<i>result</i> = currentDate ()	現在のシステム日付を、8 桁のグレゴリオ暦形式 (yyyyMMdd) で格納します。このシステム変数は、DATE 型の変数に割り当てることができます。

システム変数	説明
<i>result</i> = currentTime ()	現在のシステム時刻を、6 桁のユリウス日付形式 (HHmmss) で格納します。このシステム変数は、TIME 型の変数に割り当てることができます。
<i>result</i> = currentTimeStamp ()	現在のシステム時刻と日付を、20 桁のユリウス日付形式 (yyyyMMddHHmmssffffff) で格納します。このシステム変数は、TIMESTAMP 型の変数に割り当てることができます。
<i>result</i> = dateOf (aTimeStamp)	TIMESTAMP 型の変数から派生した日付を戻します。
<i>result</i> = dateValue (dateAsString)	入力ストリングに対応する DATE 値を戻します。
<i>result</i> = dateValueFromGregorian (gregorianIntegerDate)	グレゴリオ日付の整数表記に対応する DATE 値を戻します。
<i>result</i> = dateValueFromJulian (julianIntegerDate)	ユリウス日付の整数表記に対応する DATE 値を戻します。
<i>result</i> = dayOf (aTimeStamp)	TIMESTAMP 型の変数から派生した、日にちを表す正の整数を戻します。
<i>result</i> = extend (extensionField [, mask])	タイム・スタンプ、時刻、または日付を長形式または短形式のタイム・スタンプ値に変換します。
<i>result</i> = intervalValue (intervalAsString)	ストリング定数またはリテラルを表す INTERVAL 値を戻します。
<i>result</i> = intervalValueWithPattern (intervalAsString[, intervalMask])	ストリング定数またはリテラルを表す INTERVAL 値を戻し、指定した間隔マスクに基づいて構築されます。
<i>result</i> = mdy (month, day, year)	カレンダー日付の月、その月の日、および年を表す 3 つの整数から派生した DATE 値を戻します。
<i>result</i> = monthOf (aTimeStamp)	TIMESTAMP 型の変数から派生した、月を表す正の整数を戻します。
<i>result</i> = timeOf ([aTimeStamp])	TIMESTAMP 変数またはシステム・クロックのいずれかから派生した時刻を表すストリングを戻します。
<i>result</i> = timeStampFrom (tsDate tsTime)	現在のシステム時刻と日付を、20 桁のユリウス日付形式 (yyyyMMddHHmmssffffff) で格納します。このシステム変数は、TIMESTAMP 型の変数に割り当てることができます。
<i>result</i> = timeStampValue (timeStampAsString)	ストリング定数またはリテラルを表す TIMESTAMP 値を戻します。
<i>result</i> = timeStampValueWithPattern (timeStampAsString[, timeStampMask])	ストリングを表す TIMESTAMP 値を戻し、指定したタイム・スタンプ・マスクに基づいて構築されます。

システム変数	説明
<i>result</i> = <i>timeValue</i> (<i>timeAsString</i>)	ストリング定数またはリテラルを表す TIME 値を戻します。
<i>result</i> = <i>weekdayOf</i> (<i>aTimeStamp</i>)	TIMESTAMP 型の変数から派生した、曜日 を表す正の整数 (0 - 6) を戻します。
<i>result</i> = <i>yearOf</i> (<i>aTimeStamp</i>)	TIMESTAMP 型の変数から派生した、年を 表す正の整数を戻します。

日付、時刻、またはタイム・スタンプ変数を設定するには、それぞれ、
VGVar.currentGregorianCalendar、DateTimeLib.currentTime、および
DateTimeLib.currentTimeStamp を代入できます。書式制御文字テキストを戻す関数を
この目的に使用することはできません。

関連する参照項目

93 ページの『EGL ステートメント』

currentTime()

システム関数 **DateTimeLib.currentTime** は、システム・クロックを読み取り、現在の
のカレンダー日付を表す DATE 値を戻します。この関数は、現在の日付のみを戻
し、時刻は戻しません。

```
DateTimeLib.currentTime( )
returns (result DATE)
```

result

現在のカレンダー日付を表す DATE 値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

currentTime()

システム関数 **DateTimeLib.currentTime** は、現在のシステム時刻を 6 桁のユリウ
ス形式 (HHMMSS) で取得します。この値は、プログラムで参照されるたびに自動
的に更新されます。

```
DateTimeLib.currentTime( )
returns (result TIME)
```

result

現在のシステム時間を表す TIME 値。

DateTimeLib.currentTime は、以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースとして使用
- **return** ステートメントの引数として使用

DateTimeLib.currentTime には、以下のような特性があります。

プリミティブ型

TIME

データ長

6

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myTime = DateTimeLib.currentTime;
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

currentTimeStamp()

システム関数 **DateTimeLib.currentTimeStamp** は、現在のシステム時刻と日付を 20 桁の形式 (yyyyMMddHHmmssffffff) のタイム・スタンプとして取得します。この値は、プログラムで参照されるたびに自動的に更新されます。

```
DateTimeLib.currentTimeStamp( )  
returns (result TIMESTAMP)
```

result

現在のシステム時間および日付を表す TIMESTAMP 値。

DateTimeLib.currentTimeStamp は、以下のように使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースとして使用
- **return** ステートメントの引数として使用

DateTimeLib.currentTimeStamp には、以下のような特性があります。

プリミティブ型

TIMESTAMP

データ長

20

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myTimeStamp = DateTimeLib.currentTimeStamp;
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

dateOf()

システム関数 **DateTimeLib.dateOf** は、TIMESTAMP 型の変数から派生した DATE 値を戻します。

```
DateTimeLib.dateOf(aTimeStamp TIMESTAMP in)  
returns (result DATE)
```

result

DATE 値。

aTimeStamp

日付の派生元の値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

dateValue()

関数 **DateTimeLib.dateValue** は、ストリングに対応する DATE 値を戻します。

```
DateTimeLib.dateValue(dateAsString STRING in)  
returns (result DATE)
```

result

DATE 型の変数。

dateAsString

マスク「yyyyMMdd」を表す数字を含むストリング定数またはリテラル。詳細については、『DATE』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

536 ページの『日時式』

dateValueFromGregorian()

関数 **DateTimeLib.dateValueFromGregorian** は、グレゴリオ日付の整数表記に対応する DATE 値を戻します。

```
DateTimeLib.dateValueFromGregorian(  
  gregorianIntegerDate INT in)  
returns (result DATE)
```

result

DATE 型の変数。

gregorianIntegerDate

グレゴリオ日付を 00YYMMDD または 00YYMMDD 形式で表す VisualAge Generator の数値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

dateValueFromJulian()

関数 **DateTimeLib.dateValueFromJulian** は、ユリウス日付の整数表記に対応する DATE 値を返します。

```
DateTimeLib.dateValueFromJulian(  
    julianIntegerDate INT in)  
returns (result DATE)
```

result

DATE 型の変数。

julianIntegerDate

ユリウス日付を 00YYYYDDD または 00YYDD 形式で表す VisualAge Generator の数値。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

dayOf()

システム関数 **DateTimeLib.dayOf** は、TIMESTAMP 型の変数から派生した 日 (1 から 7 まで) を表す正整数を返します。

```
DateTimeLib.dayOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

result

月の日に対応する正整数。

aTimeStamp

日の派生元の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

extend()

システム関数 **DateTimeLib.extend** は、タイム・スタンプ、時刻、または日付を長形式または短形式のタイム・スタンプ値に変換します。以下に例を示します。

- 「ddHH」(日および時間) として定義された入力タイム・スタンプがあり、
「ddHHmm」(日、時間、および分) のタイム・スタンプ・マスクを提供する場合、**DateTimeLib.extend** は、そのマスクと一致する拡張値を返します。

- 「yyyyMMddHHmmss」(年、月、日、時間、分、および秒)として定義された入力タイム・スタンプがあり、「yyyy」(年)のタイム・スタンプ・マスクを提供する場合、**DateTimeLib.extend** は、そのマスクと一致する短縮値を返します。

```
DateTimeLib.extend(  
  extensionField dateOrTimeOrTimeStamp in in  
  [, mask outputTimeStampMask in in  
  ]  
)  
returns (result TIMESTAMP)
```

result

TIMESTAMP 型の変数。

extensionField

TIMESTAMP 型、**TIME** 型または **DATE** 型のフィールドの名前。このフィールドには、拡張または短縮される値が含まれます。

mask

関数が戻すタイム・スタンプ値のマスクを定義する文字列リテラルまたは定数。詳細については、『**TIMESTAMP**』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

intervalValue()

日時値関数 **DateTimeLib.intervalValue** は、ストリング定数またはリテラルを表す **INTERVAL** 値を返すもので、デフォルトの間隔マスク **yyyyMM** に基づいて構築されます。

入力ストリングは 6 桁の数字で構成されます。最初の 4 桁の数字は間隔の年数を表し、最後の 2 桁の数字は月数を表します。

yyyyMM 以外の形式でマスクを指定する場合は、**DateTimeLib.intervalValueWithPattern** を呼び出します。

```
DateTimeLib.intervalValue(intervalAsString STRING in)  
returns (result INTERVAL)
```

result

INTERVAL 型の変数

intervalAsString

意味が間隔マスク **yyyyMM** で示される 6 桁の数字を含むストリング定数またはリテラル

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

45 ページの『**INTERVAL**』

856 ページの『**intervalValueWithPattern()**』

intervalValueWithPattern()

日時値関数 **DateTimeLib.intervalValueWithPattern** は、ストリング定数またはリテラルを表す **INTERVAL** 値を戻すもので、(オプションとして) 指定した間隔マスクに基づいて構築されます。例えば、マスクが *yyyy* の場合、入力ストリングには 4 桁の数字が含まれている必要があり、これらの数値は、間隔で表される年数を表します。

```
DateTimeLib.intervalValueWithPattern(  
  intervalAsString STRING in  
  [, intervalMask STRING in  
  ]  
)  
returns (result INTERVAL)
```

result

INTERVAL 型の変数。

intervalAsString

意味が間隔マスクによって示される数字を含む、ストリング定数またはリテラル。

intervalMask

最初のパラメーターの各数字に意味付けをする間隔マスクを指定します。デフォルトのマスクは *yyyyMM* です。その他の詳細については、『*INTERVAL*』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

45 ページの『*INTERVAL*』

mdy()

mdy 演算子は、3 つの整数から派生した **DATE** 値を戻します。これらの整数は、カレンダー日付の月、その月の日、および年を表します。

```
DateTimeLib.mdy(  
  month INT in,  
  day INT in,  
  year INT in)  
returns (result DATE)
```

result

DATE 値。

month

1 から 12 の範囲の整数で、月を表します。

day

1 か月のうちの日を表す整数 (月によって 1 から 28、29、30、または 31)。

year

年を表す 4 桁の数字。

カレンダーの日および月の範囲外の値を指定した場合、またはオペランドが 3 つでない場合、エラーになります。 **MDY**() が関数である場合と同じように、コマで

区切られた 3 つの整数式オペランドを括弧で囲む必要があります。3 番目の式は、年の略語にすることはできません。例えば、年に 99 を指定すると、1 世紀の 99 年 (約 1,900 年前) が指定されます。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

monthOf()

システム関数 **DateTimeLib.monthOf** は、TIMESTAMP 型の変数から派生した月を表す正整数を戻します。

```
DateTimeLib.monthOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

result

月を表す正整数。

aTimeStamp

月の派生元の **TIMESTAMP** 変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

timeOf()

システム関数 **DateTimeLib.timeOf** は、TIMESTAMP 変数またはシステム・クロックから派生した時刻を表す文字列を戻します。

```
DateTimeLib.timeOf([aTimeStamp TIMESTAMP in])  
returns (result STRING)
```

result

24 時間クロックおよび以下のフォーマットに基づいた *aTimeStamp* 引数の時刻部分。

hh:mm:ss

hh 時間 (2 桁の文字列)

mm

分 (2 桁の文字列)

ss 秒 (2 桁の文字列)

aTimeStamp

DATEIME 値。値が指定されない場合、この演算子は現在時刻を表す文字列を戻します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

849 ページの『EGL ライブラリー DateTimeLib』

timeStampFrom()

関数 **DateTimeLib.timeStampFrom** は、指定した DATE および TIME に基づいて構築された TIMESTAMP 値を返します。

```
DateTimeLib.timeStampFrom(  
    tsDate DATE in,  
    tsTime TIME in)  
returns (result TIMESTAMP)
```

result

TIMESTAMP 型の値。

tsDate

DATE 型の変数。

tsTime

TIME 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

849 ページの『EGL ライブラリー DateTimeLib』

47 ページの『TIMESTAMP』

timeStampValue()

関数 **DateTimeLib.timeStampValue** は、ストリング定数またはリテラルを表す TIMESTAMP 値を返し、デフォルトのタイム・スタンプ・マスク *yyyyMMddHHmmss* に基づいて構築されます。

入力ストリングは次のように 14 桁の数字で構成されます。

- 最初の 4 桁の数字は年を表します。
- 次の 2 桁の数字は月 (数字) を表します。
- 次の 2 桁の数字はその月の日付を表します。
- 次の 2 桁の数字は時間数を表します (00 から 24 まで)。
- 次の 2 桁の数字はその時間内の分数を表します。
- 最後の 2 桁の数字はその分内の秒数を表します。

yyyyMMddHHmmss 以外の形式でマスクを指定する場合は、**DateTimeLib.timestampValueWithPattern** を呼び出します。

```
DateTimeLib.timeStampValue(timestampAsString STRING in)  
returns (result TIMESTAMP)
```

result

TIMESTAMP 型の変数。

timeStampAsString

意味がタイム・スタンプ・マスク *yyyyMMddHHmmss* で示される 14 桁の数字を含む、ストリングまたはリテラル。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

47 ページの『TIMESTAMP』

『timeStampValueWithPattern()』

timeStampValueWithPattern()

関数 **DateTimeLib.timeStampValueWithPattern** は、ストリング定数またはリテラルを表す **TIMESTAMP** 値を戻すもので、(オプションとして) 指定したタイム・スタンプ・マスクに基づいて構築されます。例えば、マスクが「yyyy」の場合、入力ストリングには 4 桁の数値が含まれている必要があり、これらの数値は、年の値をタイム・スタンプで表します。

```
DateTimeLib.timeStampValueWithPattern(  
    timeStampAsString STRING in  
    [, timeStampMask STRING in  
    ]  
)  
returns (result TIMESTAMP)
```

result

TIMESTAMP 型の変数。

timeStampAsString

意味がタイム・スタンプ・マスクによって示される数字を含む、ストリング定数またはリテラル。

timeStampMask

最初のパラメーターの各数字に意味付けをするタイム・スタンプ・マスクを指定します。デフォルトのマスクは *yyyyMMddHHmmss* です。詳細については、『**TIMESTAMP**』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

47 ページの『TIMESTAMP』

timeValue()

日時値関数 **DateTimeLib.timeValue** は、ストリング定数またはリテラルを表す **TIME** 値を戻します。

```
DateTimeLib.timeValue(timeAsString STRING in)  
returns (result TIME)
```

result

TIME 型の変数。

timeAsString

マスク「HHmmss」を表す数字を含むストリング定数またはリテラル。詳細については、『*TIME*』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

536 ページの『日時式』

47 ページの『*TIME*』

weekdayOf()

システム関数 **DateTimeLib.weekdayOf** は *TIMESTAMP* 型の変数から派生した曜日を表す正整数を返します。数値 0 が日曜日を表し、1 が月曜日を表し、以下同様です。

```
DateTimeLib.weekdayOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

result

0 から 6 までの正整数。

aTimeStamp

日の派生元の *TIMESTAMP* 変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『*DATE*』

849 ページの『EGL ライブラリー *DateTimeLib*』

yearOf()

システム関数 **DateTimeLib.yearOf** は、変数から *TIMESTAMP* 型に派生した年を表す 4 桁の整数を返します。

```
DateTimeLib.yearOf(aTimeStamp TIMESTAMP in)  
returns (result INT)
```

result

年を表す整数。

aTimeStamp

年の派生元の *TIMESTAMP* 変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『*DATE*』

849 ページの『EGL ライブラリー *DateTimeLib*』

EGL ライブラリー *J2EELib*

ライブラリー *J2EELib* のシステム関数は、以下の表のとおりです。

関数	説明
<code>clearRequestAttr (key)</code>	要求オブジェクト内で、指定されたキーに関連付けられている引数を除去する。
<code>clearSessionAttr (key)</code>	セッション・オブジェクト内で、指定されたキーに関連付けられている引数を除去する。
<code>getRequestAttr (key, argument)</code>	指定されたキーを使用して、要求オブジェクトから指定された変数に引数を取り出す。
<code>getSessionAttr (key, argument)</code>	指定されたキーを使用して、セッション・オブジェクトから指定された変数に引数を取り出す。
<code>setRequestAttr (key, argument)</code>	指定されたキーを使用して、要求オブジェクトに指定された引数を置く。
<code>setSessionAttr (key, argument)</code>	指定されたキーを使用して、セッション・オブジェクトに指定された引数を置く。

clearRequestAttr()

システム関数 **J2EELib.clearRequestAttr** は、要求オブジェクト内で、指定されたキーに関連付けられている引数を除去します。この関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。

システム関数 **J2EELib.setRequestAttr** を使用すると、要求オブジェクト内に引数を設定することができます。システム関数 **J2EELib.getRequestAttr** を使用すると、その引数を取り出すことができます。

J2EELib.clearRequestAttr(key STRING in)

key

文字列リテラル、または STRING 型の式

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

862 ページの『getRequestAttr()』

863 ページの『setRequestAttr()』

clearSessionAttr()

システム関数 **J2EELib.clearSessionAttr** は、セッション・オブジェクト内で、指定されたキーに関連付けられている引数を除去します。この関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。

システム関数 **J2EELib.setSessionAttr** を使用すると、セッション・オブジェクトに引数を設定することができます。システム関数 **J2EELib.getSessionAttr** を使用すると、後でその引数を取り出すことができます。

J2EELib.clearSessionAttr(key STRING in)

key

文字列リテラル、または `STRING` 型の式

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

『getSessionAttr()』

864 ページの『setSessionAttr()』

getRequestAttr()

システム関数 **J2EELib.getRequestAttr** は、指定されたキーを使用して、要求オブジェクトから引数を取り出し、指定された変数に格納しますこの関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。

指定されたキーを使用してオブジェクトが見つからない場合は、ターゲット変数は変更されていません。検索されたオブジェクトの型が間違っている場合は、例外がスローされ、プログラムまたはページ・ハンドラーは終了します。

システム関数 **J2EELib.setRequestAttr** を使用することによって、要求オブジェクトに引数を設定できます。サープレットの要求のコレクション内に配置された引数オブジェクトは、サープレットの要求が有効であれば、アクセスすることができます。ページから書式を処理依頼すると、新しい要求が作成されます。

```
J2EELib.getRequestAttr(  
    key STRING in,  
    argument attribute inOut)
```

key

文字リテラル、または任意の文字型の項目。

argument

項目、レコード、または配列。

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

863 ページの『setRequestAttr()』

getSessionAttr()

システム関数 **J2EELib.getSessionAttr** は、指定されたキーを使用して、セッション・オブジェクトから引数を取り出し、指定された変数に格納します。この関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。

指定されたキーを使用してオブジェクトが見つからない場合は、ターゲット変数は変更されていません。検索されたオブジェクトの型が間違っている場合は、例外がスローされ、プログラムまたはページ・ハンドラーは終了します。

システム関数 `J2EELib.setSessionAttr` を使用すると、セッション・オブジェクトに引数を配置することができます。

```
J2EELib.getSessionAttr(  
    key STRING in,  
    argument attribute in)
```

key

文字リテラル、または任意の文字型の項目。

argument

項目、レコード、または配列。

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

864 ページの『setSessionAttr()』

setRequestAttr()

システム関数 **J2EELib.setRequestAttr** は、指定されたキーを使用して要求オブジェクトに指定された引数を配置します。この関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。システム関数 **J2EELib.getRequestAttr** を使用することによって、後で引数を検索できます。

```
J2EELib.setRequestAttr(  
    key STRING in,  
    argument attribute in)
```

key

文字リテラル、または任意の文字型の項目。

argument

項目、レコード、または配列。

生成済みの Java 出力では、項目引数は、プリミティブ Java オブジェクト (String、Integer、Decimal など) として渡されます。レコード引数は、レコード Bean として渡されます。配列は、関連付けられている型の配列リストとして渡されます。引数オブジェクトは、サーブレットの要求のコレクション内に配置され、サーブレットの要求が有効であれば、アクセスすることができます。ページから書式を処理依頼すると、新しい要求が作成されます。

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

862 ページの『getRequestAttr()』

setSessionAttr()

システム関数 **J2EELib.setSessionAttr** は、指定されたキーを使用して、指定された引数をセッション・オブジェクトに配置します。この関数は、ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで役に立ちます。システム関数 **J2EELib.getSessionAttr** を使用すると、後でその引数を取り出すことができます。

```
J2EELib.setSessionAttr(  
    key STRING in,  
    argument attribute in)
```

key

文字リテラル、または任意の文字型の項目。

argument

項目、レコード、または配列。

生成済みの Java 出力では、項目引数は、プリミティブ Java オブジェクト (String、Integer、Decimal など) として渡されます。レコード引数は、レコード Bean として渡されます。配列は、関連付けられている型の配列リストとして渡されます。

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

860 ページの『EGL ライブラリー J2EELib』

862 ページの『getSessionAttr()』

EGL ライブラリー JavaLib

次の表は、Java アクセス関数をまとめたものです。

関数	説明
<i>result</i> = getField (<i>identifierOrClass</i> , <i>field</i>)	指定したオブジェクトまたはクラスの、指定したフィールドの値を戻す
<i>result</i> = invoke (<i>identifierOrClass</i> , <i>method</i> [, <i>argument</i>])	Java オブジェクトまたはクラス上のメソッドを呼び出す。値が戻されることがある。
<i>result</i> = isNull (<i>identifier</i>)	指定した ID が NULL オブジェクトを参照しているかどうかを示す値を戻す (真の場合は 1、偽の場合は 0)
<i>result</i> = isObjID (<i>identifier</i>)	指定した ID がオブジェクト・スペース内に存在しているかどうかを示す値を戻す (真の場合は 1、偽の場合は 0)
<i>result</i> = qualifiedTypeName(<i>identifier</i>)	オブジェクト・スペース内にあるオブジェクトのクラスの完全修飾名を戻す
remove (<i>identifier</i>)	指定した ID をオブジェクト・スペースから除去する。オブジェクトを参照している ID が他にない場合は、そのオブジェクトを除去する。

関数	説明
<code>removeAll ()</code>	オブジェクト・スペースから ID とオブジェクトをすべて除去する
<code>setField (identifierOrClass, field, value)</code>	Java オブジェクトまたはクラスにフィールド値を設定する
<code>store (storeId, identifierOrClass, method{,argument})</code>	メソッドを呼び出し、戻されたオブジェクト (または <code>null</code>) を、指定した ID とともにオブジェクト・スペースに格納する
<code>storeCopy (sourceId, targetID)</code>	オブジェクト・スペース内の別の ID に基づいて、新しい ID を作成して、両方の ID が同じオブジェクトを参照するようにする。
<code>storeField (storeId, identifierOrClass, field)</code>	クラス・フィールドまたはオブジェクト・フィールドの値をオブジェクト・スペースに格納する
<code>storeNew (storeId, class{,argument})</code>	クラスのコンストラクターを呼び出し、新しいオブジェクトをオブジェクト・スペースに格納する

Java アクセス関数

Java アクセス関数は、EGL システム関数です。これによって、ユーザーが作成した Java コードで、ネイティブ Java オブジェクトおよびクラス (特にネイティブ・コードの `public` メソッド、コンストラクター、およびフィールド) にアクセスできます。

EGL フィーチャーは、実行時に *EGL Java* オブジェクト・スペース が存在する場合に使用可能になります。EGL Java ネームスペースは、一連の名前とそれらの名前が参照するオブジェクトです。生成したプログラムとそれらのプログラムがローカルで呼び出すすべての生成済み Java コードで、単一のオブジェクト・スペースを使用できます。呼び出しが直接呼び出しであるか別のローカル生成された Java プログラムを介しているかにかかわらず、任意のレベルの呼び出しで使用できます。オブジェクト・スペースは任意のネイティブ Java コードで利用できるわけではありません。

オブジェクトをネームスペースに格納したりオブジェクト・スペースで検索したりするには、Java アクセス関数を呼び出します。呼び出しでは ID も使用します。各 ID は、オブジェクトをオブジェクト・スペースに格納したり、すでに存在する名前と一致させるために使用されるストリングです。ID が名前と一致すると、コードからその名前に関連付けられたオブジェクトにアクセスできます。

以下に次の各セクションがあります。

- 『EGL および Java の型のマッピング』
- 867 ページの『例』
- 870 ページの『エラー処理』

EGL および Java の型のマッピング: メソッドに渡すそれぞれの引数 (およびフィールドに割り当てるそれぞれの値) は、Java オブジェクトまたはプリミティブ型へマップされます。例えば、EGL プリミティブ型 `CHAR` の項目は、Java `String` クラ

スのオブジェクトとして渡されます。 EGL の型から Java の型へのマッピングが不十分な場合のために、キャスト演算子が提供されています。

Java 名を指定すると、EGL は、値の先頭と最後から、単一バイトのブランクと 2 バイトのブランクを削除します。この値では、大/小文字が区別されます。切り捨ては、どのキャストよりも先に行われます。この規則は、ストリング・リテラル、および CHAR 型、DBCHAR 型、MBCHAR 型、または UNICODE 型の項目に適用されます。値を objID または null にキャストしない限り、メソッド引数またはフィールド値の指定時に、このような切り捨ては行われません (例えば、ストリング "my data" は、そのままの形でメソッドに渡されます)。

次の表は、有効なマッピングをすべてまとめたものです。

引数のカテゴリー		例	Java の型
ストリング・リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目	キャストなし	"myString"	java.lang.String
	ID を示す objId でキャスト	(objId)"myId" x = "myId"; (objId)x	ID が参照するオブジェクトのクラス
	null でキャスト。これは、完全修飾クラスに null 参照を提供するのに適しています	(null)"java.lang.Thread" x = "java.util.HashMap"; (null)x	指定したクラス 注: (null)"int[]" などの null でキャストされた配列に渡すことはできません
	char でキャスト。これは、値の先頭の文字が渡されることを意味します (次の列の例ではそれぞれ "a" が渡されます)	(char)"abc" x = "abc"; (char)x	char
FLOAT 型の項目または浮動小数点数リテラル	キャストなし	myFloatValue	double
HEX 型の項目	キャストなし	myHexValue	バイト配列
SMALLFLOAT 型の項目	キャストなし	mySmallFloat	float
DATE 型の項目	キャストなし	myDate	java.sql.Date
TIME 型の項目	キャストなし	myTime	java.sql.Time
TIMESTAMP 型の項目	キャストなし	myTimeStamp	java.sql.Timestamp
INTERVAL 型の項目	キャストなし	myInterval	java.lang.String
浮動小数点数リテラル	キャストなし	-6.5231E96	double

引数のカテゴリー		例	Java の型
小数部を含まない数値項目（または非浮動小数点数リテラル）。先行ゼロは、リテラルの桁数に含まれます。	キャストなし、1 - 4 桁	0100	short
	キャストなし、5 から 9 桁	00100	int
	キャストなし、9 から 18 桁	1234567890	long
	キャストなし、18 桁より大	1234567890123456789	java.math.BigInteger
小数部を含む数値項目（または非浮動小数点数リテラル）。先行ゼロおよび後続ゼロは、リテラルの桁数に含まれます。	キャストなし、1 から 6 桁	3.14159	float
	キャストなし、7 から 18 桁	3.14159265	double
	キャストなし、18 桁より大	56789543.222	java.math.BigDecimal
小数部付き、または小数部なしの数値項目または非浮動小数点数リテラル	bigdecimal、biginteger、byte、double、float、short、int、long でキャスト	X = 42; (byte)X (long)X	指定した基本型。ただし、値がその型の範囲を超えると、精度が失われ、符号が変わることがあります。
	ブール値へのキャスト。これは、ゼロ以外が真、ゼロが偽であることを意味します。	X = 1; (boolean)X	boolean

注: 精度が失われるのを回避するために、Java の double には EGL の float 項目を使用し、Java の float には EGL の smallfloat 項目を使用してください。それ以外の EGL の型を使用すると、多くの場合、結果としての値が丸められます。

EGL 内の項目の内部形式については、『基本型』のページを参照してください。

例: このセクションでは、Java アクセス関数の使用方法について例を示します。

日付ストリングの出力: 次の例では、日付ストリングが出力されます。

```
// Java Date クラスのコンストラクターを呼び出し
// 新規オブジェクトを ID "date" に割り当てます。
JavaLib.storeNew( (objId)"date", "java.util.Date" );

// 新規 Date オブジェクトの toString メソッドを呼び出し
// 出力（本日の日付）を charItem に割り当てます。
// キャスト (objId) が存在しない場合、"date" は
// オブジェクトではなくクラスを参照します。
charItem = JavaLib.invoke( (objId)"date", "toString" );

// Java System クラスの標準出力ストリームを
// ID "systemOut" に割り当てます。
JavaLib.storeField( (objId)"systemOut",
    "java.lang.System", "out" );

// 出力ストリームの println メソッドを呼び出し
// 今日の日付を出力します。
```

```

JavaLib.invoke( (objID)"systemOut","println",charItem );

// "java.lang.System.out" を
// 前の行の第 1 引数として使用すると、
// 無効です。この引数は、すでにオブジェクト・スペース内にある
// 識別子であるか、クラス名であることが必要だからです。
// この引数は static フィールドを参照できません。

```

システム・プロパティのテスト: 次の例では、システム・プロパティが検索され、値の有無がテストされます。

```

// ID の名前を CHAR 型の項目に割り当てます
valueID = "osNameProperty"

// プロパティ os.name の値をオブジェクト・スペースに格納し
// この値 (Java の String) を以下に
// 関連付けます
JavaLib.store((objId)valueId, "java.lang.System",
    "getProperty", "os.name");

// プロパティ値が存在しないかどうかをテストし、
// それに応じて処理を実行します
myNullFlag = JavaLib.isNull( (objId)valueId );

if( myNullFlag == 1 )
    error = 27;
end

```

配列の使用: EGL で Java 配列を扱うときは、Java クラス `java.lang.reflect.Array` を使用します (詳細については、以降の例や Java API 文書を参照してください)。Java 配列はコンストラクターを持たないため、**JavaLib.storeNew** を使用して Java 配列を作成することはできません。

オブジェクト・スペースに配列を作成するには、`java.lang.reflect.Array` の静的メソッド `newInstance` を使用します。配列を作成したら、そのクラスの他のメソッドを使用して、エレメントにアクセスします。

メソッド `newInstance` は、次の 2 つの引数を取ります。

- 作成される配列の型を決定する Class オブジェクト
- 配列内のエレメント数を指定する数字

Class オブジェクトを識別するコードは、オブジェクト配列、または基本配列のどちらを作成するのかに応じて変化します。配列と対話する後続のコードも、これと同じ基準で変化します。

オブジェクトの配列の使用: 次の例は、ID `"myArray"` を使用してアクセス可能な、5 つのエレメントを持つオブジェクト配列の作成方法を示しています。

```

// newInstance で使用するためにクラスへの参照を取得します
JavaLib.store( (objId)"objectClass", "java.lang.Class",
    "forName", "java.lang.Object" );

// オブジェクト・スペース内に配列を作成します
JavaLib.store( (objId)"myArray", "java.lang.reflect.Array",
    "newInstance", (objId)"objectClass", 5 );

```

異なる型のオブジェクトを保持する配列を作成したい場合は、**JavaLib.store** の最初の呼び出しに渡すクラス名を変更してください。String オブジェクトの配列を作成するには、例えば、`"java.lang.Object"` の代わりに `"java.lang.String"` を渡します。

オブジェクト配列の要素にアクセスするには、`java.lang.reflect.Array` の `get` メソッドと `set` メソッドを使用します。次の例では、`i` と `length` が数値項目となっています。

```
length = JavaLib.invoke( "java.lang.reflect.Array",
    "getLength", (objId)"myArray" );
i = 0;

while ( i < length )
    JavaLib.store( (objId)"element", "java.lang.reflect.Array",
        "get", (objId)"myArray", i );

    // ここで、要素を適宜処理します
    JavaLib.invoke( "java.lang.reflect.Array", "set",
        (objId)"myArray", i, (objId)"element" );
    i = i + 1;
end
```

上記の例は、次の Java コードと同等です。

```
int length = myArray.length;

for ( int i = 0; i < length; i++ )
{
    Object element = myArray[i];

    // ここで、要素を適宜処理します

    myArray[i] = element;
}
```

Java プリミティブの配列の操作: オブジェクトではなく、Java プリミティブを格納する配列を作成するには、`java.lang.reflect.Array` を使用するより前のステップで、別のメカニズムを使用します。特に、基本型クラスの静的フィールド `TYPE` にアクセスして、`newInstance` の `Class` 引数を取得する必要があります。

次の例では、30 の要素を持つ整数配列、`myArray2` を作成します。

```
// newInstance で使用するためにクラスへの参照を取得します
JavaLib.storeField( (objId)"intClass",
    "java.lang.Integer", "TYPE");

// オブジェクト・スペース内に配列を作成します
JavaLib.store( (objId)"myArray2", "java.lang.reflect.Array",
    "newInstance", (objId)"intClass", 30 );
```

異なる型のプリミティブを保持する配列を作成したい場合は、**`JavaLib.storeField`** の呼び出しに渡すクラス名を変更してください。文字配列を作成するには、例えば、`"java.lang.Integer"` の代わりに `"java.lang.Character"` を渡します。

基本配列の要素にアクセスするには、基本型に固有の `java.lang.reflect.Array` メソッドを使用します。このメソッドには、`getInt`、`setInt`、`getFloat`、`setFloat` などが含まれています。次の例では、`length`、`element`、および `i` が数値項目となっています。

```
length = JavaLib.invoke( "java.lang.reflect.Array",
    "getLength", (objId)"myArray2" );
i = 0;

while ( i < length )
    element = JavaLib.invoke( "java.lang.reflect.Array",
        "getDouble", (objId)"myArray2", i );
```

```
// ここで、エレメントを適宜処理します

JavaLib.invoke( "java.lang.reflect.Array", "setDouble",
  (objId)"myArray2", i, element );
i = i + 1;
end
```

上記の例は、次の Java コードと同等です。

```
int length = myArray2.length;

for ( int i = 0; i < length; i++ )
{
  double element = myArray2[i];

  // ここで、エレメントを適宜処理します

  myArray2[i] = element;
}
```

コレクションの使用: *list* と呼ばれる変数が参照するコレクションを繰り返すには、Java プログラムで次を実行します。

```
Iterator contents = list.iterator();

while( contents.hasNext() )
{
  Object myObject = contents.next();
  // myObject を処理します
}
```

hasNext は数値データであり、プログラムでは *list* と呼ばれる ID とコレクションが関連付けられているものと想定します。この場合、次の EGL コードは、前述した Java コードと同じ意味を持ちます。

```
JavaLib.store( (objId)"contents", (objId)"list", "iterator" );
hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );

while ( hasNext == 1 )
  JavaLib.store( (objId)"myObject", (objId)"contents", "next");

  // myObject を処理します
  hasNext = JavaLib.invoke( (objId)"contents", "hasNext" );
end
```

配列をコレクションに変換: オブジェクト配列からコレクションを作成するには、`java.util.Arrays` の `asList` メソッドを使用します。例を次に示します。

```
// 配列 myArray からコレクションを作成し、
// そのコレクションと ID "list" を関連付けます
JavaLib.store( (objId)"list", "java.util.Arrays",
  "asList", (objId)"myArray" );
```

次に、前のセクションで示したように、*list* を繰り返します。

配列をコレクションに変換できるのは、オブジェクトの配列の場合に限られます。Java プリミティブの配列をコレクションに変換することはできません。`java.util.Arrays` と `java.lang.reflect.Array` を混同しないように気を付けてください。

エラー処理: 多くの場合、Java アクセスメソッドは、エラー・コードに関連付けられています (関数固有のヘルプのページを参照してください)。リストされているエラーのいずれかが発生したときに、システム変数 `VGVar.handleSysLibraryErrors` の値

が 1 であると、EGL は、システム変数 **sysVar.errorCode** にゼロ以外の値を設定します。エラーのいずれかが発生したときに **VGVar.handleSysLibraryErrors** の値が 0 の場合は、プログラムが終了します。

sysVar.errorCode の値 "00001000" は、特に重要です。この値は、呼び出されたメソッドにより、またはクラス初期化の結果として、例外がスローされたことを示します。

例外がスローされると、EGL は、それをオブジェクト・スペースに格納します。別の例外が発生すると、2 番目の例外が 1 番目の例外に取って代わります。発生した最後の例外にアクセスするには、ID *caughtException* を使用します。

特別な状況では、呼び出されたメソッドは、例外だけではなく、**OutOfMemoryError** や **StackOverflowError** などのエラーもスローします。この場合、プログラムは、システム変数 **VGVar.handleSysLibraryErrors** の値に関係なく終了します。

次の Java コードは、Java プログラムに複数の catch ブロックを記述して、各種の例外を処理する方法を示しています。このコードでは、**FileOutputStream** オブジェクトの生成を試みています。コードにエラーが発生すると、**errorType** 変数が設定され、スローされた例外が格納されます。

```
int errorType = 0;
Exception ex = null;

try
{
    java.io.FileOutputStream fOut =
        new java.io.FileOutputStream( "out.txt" );
}
catch ( java.io.IOException iox )
{
    errorType = 1;
    ex = iox;
}
catch ( java.lang.SecurityException sx )
{
    errorType = 2;
    ex = sx;
}
```

次の EGL コードは、前述した Java コードと同等です。

```
VGVar.handleSysLibraryErrors = 1;
errorType = 0;

JavaLib.storeNew( (objId)"fOut",
    "java.io.FileOutputStream", "out.txt" );

if ( sysVar.errorCode == "00001000" )
    exType = JavaLib.qualifiedTypeName( (objId)"caughtException" );

if ( exType == "java.io.IOException" )
    errorType = 1;
    JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
else
    if ( exType == "java.lang.SecurityException" )
        errorType = 2;
        JavaLib.storeCopy( (objId)"caughtException", (objId)"ex" );
    end
end
end
```


関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

100 ページの『例外処理』

37 ページの『プリミティブ型』

『getField()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

getField()

システム関数 **JavaLib.getField** は、指定されたオブジェクトまたはクラスの、指定されたフィールドの値を返します。 **JavaLib.getField** は、Java アクセス関数の 1 つです。

```
JavaLib.getField(  
    identifierOrClass javaObjIdOrClass in,  
    field STRING in)  
returns (result anyJavaPrimitive)
```

result

結果フィールドは、必須であり、2 番目の引数で指定されたフィールドの値を受け取ります。次のことが該当します。

- 受け取る値が `BigDecimal`、`BigInteger`、`byte`、`short`、`int`、`long`、`float`、または `double` の場合、結果フィールドは数値データ型でなければなりません。特性は値と同じである必要はありません。例えば、`float` は、小数桁数なしで宣言された戻り変数に保管できます。オーバーフロー処理の詳細については、『`VGVar.handleOverflow`』および『`sysVar.overflowIndicator`』を参照してください。
- 受け取る値がブール値の場合、結果フィールドは、数値プリミティブ型でなければなりません。値は、真の場合は 1、偽の場合は 0 です。
- 受け取る値がバイト配列の場合、結果フィールドは、`HEX` 型でなければなりません。長さが一致しない場合については、『代入』を参照してください。
- 受け取る値が `String` または `char` の場合、結果フィールドは、`CHAR` 型、`DBCHAR` 型、`MBCHAR` 型、`STRING` 型、または `UNICODE` 型でなければなりません。
 - 結果フィールドが `MBCHAR` 型、`STRING` 型、または `UNICODE` 型である場合、受け取る値は常に適切な値になります。
 - 結果フィールドが `CHAR` 型の場合、受け取る値に `DBCHAR` 文字に相当する文字が含まれていると、問題が発生することがあります。
 - 結果フィールドが `DBCHAR` 型の場合、受け取る値に 1 バイト文字に相当するユニコード文字が含まれていると、問題が発生することがあります。

長さが一致しない場合については、『代入』を参照してください。

- ネイティブ Java メソッドが値を戻さないか、または NULL を戻す場合は、エラー 00001004 (後述) が発生します。

identifierOrClass

この引数は、次のエンティティのいずれかです。

- オブジェクト・スペース内のオブジェクトを参照する ID
- Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。オブジェクトの ID を指定する場合は、後述の例のように、その ID を objID にキャストする必要があります。次の引数に static フィールドを指定する場合は、この引数にクラスを指定することをお勧めします。

EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

field

読み取るフィールドの名前。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。大文字小文字の区別があるストリングの先頭と最後から、1 バイトのブランクと 2 バイトのブランクが削除されます。

以下に例を示します。

```
myVar = JavaLib.getField( (objId)"myID", "myField" );
```

JavaLib.getField の処理中にエラーが発生すると、**sysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001004	メソッドが NULL を戻したか、メソッドが値を戻さないか、またはフィールドの値が NULL でした。
00001005	戻り値が、戻り変数の型と一致しません。
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとしていました。

sysVar.errorCode の値	説明
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する概念

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

407 ページの『代入』

53 ページの『BIN および整数型』

864 ページの『EGL ライブラリー JavaLib』

100 ページの『例外処理』

『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

invoke()

システム関数 **JavaLib.invoke** は、ネイティブ Java オブジェクトまたはクラスのメソッドを呼び出し、値を戻すことができます。**JavaLib.invoke** は、Java アクセス関数の 1 つです。

```
JavaLib.invoke(
    identifierOrClass javaObjIdOrClass in,
    method STRING in
    {, argument anyEglPrimitive in})
returns (result anyJavaPrimitive)
```

result

結果フィールドが存在する場合は、結果フィールドがネイティブ Java メソッドからの値を受け取ります。

ネイティブ Java メソッドが値を戻す場合、結果フィールドはオプションです。

次のことが該当します。

- 戻り値が BigDecimal、BigInteger、byte、short、int、long、float、または double の場合、結果フィールドは数値データ型でなければなりません。特性は値と同じである必要はありません。例えば、float は、小数桁数なしで宣言された結果フィールドに保管できます。オーバーフロー処理の詳細については、『VGVar.handleOverflow』および『SysVar.overflowIndicator』を参照してください。

- 戻り値がブール値の場合、結果フィールドは、数値プリミティブ型でなければなりません。値は、真の場合は 1、偽の場合は 0 です。
- 戻り値がバイト配列の場合、結果フィールドは、HEX 型でなければなりません。長さが一致しない場合については、『代入』を参照してください。
- 戻り値が String または char の場合、結果フィールドは、CHAR 型、DBCHAR 型、MBCHAR 型、STRING 型、または UNICODE 型でなければなりません。
 - 結果フィールドが MBCHAR 型、STRING 型、または UNICODE 型である場合、戻り値は常に適切な値になります。
 - 結果フィールドが CHAR 型の場合、戻り値に DBCHAR 文字に相当する文字が含まれていると、問題が発生することがあります。
 - 結果フィールドが DBCHAR 型の場合、戻り値に 1 バイト文字に相当するユニコード文字が含まれていると、問題が発生することがあります。

長さが一致しない場合については、『代入』を参照してください。

- ネイティブ Java メソッドが値を戻さないか、または NULL を戻す場合は、次のことが該当します。
 - 結果フィールドが存在しない場合は、エラーが発生しません。
 - 結果フィールドが存在する場合は、実行時にエラーが発生します。エラーは、00001004 (後述) です。

identifierOrClass

この引数は、次のエンティティのいずれかです。

- オブジェクト・スペース内のオブジェクトを参照する ID
- Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。オブジェクトの ID を指定する場合は、後述の例のように、その ID を objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

オブジェクト ID を作成するまでは、オブジェクト上のメソッドをコードで呼び出すことはできません。後述の例では、PrintStream オブジェクトを参照する java.lang.System.out を使用して、この点を説明します。

method

呼び出すメソッドの名前。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。大文字小文字の区別があるストリングの先頭と最後から、1 バイトのブランクと 2 バイトのブランクが削除されます。

argument

メソッドに渡される値。

『Java アクセス (システム・ワード)』に記述したキャストが必要な場合があります。

Java の型変換の規則が適用されます。例えば、int として宣言されたメソッド・パラメーターに short を渡しても、エラーは発生しません。

精度が失われるのを避けるために、Java の double には EGL の float 変数を使用し、Java の float には EGL の smallfloat 変数を使用してください。それ以外の EGL の型を使用すると、多くの場合、結果としての値が丸められます。

呼び出し側プログラムのメモリー領域は、メソッドの実行内容にかかわらず、変更されません。

次の例では、特に注記がない限り、キャスト (objId) が必要です。

```
// Java Date クラスのコンストラクターを呼び出し
// 新規オブジェクトを ID "date" に割り当てます。
JavaLib.storeNew( (objId)"date", "java.util.Date");

// 新規 Date オブジェクトの toString メソッドを呼び出し
// 出力 (本日の日付) を chaItem に割り当てます。
// キャスト (objId) が存在しない場合、"date" は
// オブジェクトではなくクラスを参照します。
chaItem = JavaLib.invoke( (objId)"date", "toString" );

// Java System クラスの標準出力ストリームを
// ID "systemOut" に割り当てます。
JavaLib.storeField( (objId)"systemOut", "java.lang.System", "out" );

// 出力ストリームの println メソッドを呼び出し
// 今日の日付を出力します。
JavaLib.invoke( (objID)"systemOut", "println", chaItem );

// "java.lang.System.out" を
// 前の行の第 1 引数として使用すると、
// 無効です。この引数は、すでにオブジェクト・スペース内にある
// 識別子であるか、クラス名であることが必要だからです。
// この引数は static フィールドを参照できません。
```

JavaLib.invoke の処理中にエラーが発生すると、**SysVar.errorCode** は以下の表の値に設定されます。

SysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001003	EGL プリミティブ型が、Java で想定される型と同じではありません。
00001004	メソッドが NULL を戻したか、メソッドが値を戻さないか、またはフィールドの値が NULL でした。
00001005	戻り値が、戻り変数の型と一致しません。
00001006	NULL へキャストする引数のクラスをロードできませんでした。

SysVar.errorCode の値	説明
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとした。
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する概念

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

407 ページの『代入』

53 ページの『BIN および整数型』

864 ページの『EGL ライブラリー JavaLib』

100 ページの『例外処理』

872 ページの『getField()』

『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

37 ページの『プリミティブ型』

997 ページの『overflowIndicator』

1012 ページの『handleOverflow』

isNull()

システム関数 **JavaLib.isNull** は、指定した ID が NULL オブジェクトを参照しているかどうかを示す値 (真の場合は 1、偽の場合は 0) を戻します。 **JavaLib.isNull** は、Java アクセス関数の 1 つです。

```
JavaLib.isNull(identifier javaObjId in)
returns (result INT)
```

result

数値フィールド。真の場合は 1、偽の場合は 0 を受け取ります。非数値フィールドを使用すると、検証時にエラーが発生します。

identifier

オブジェクト・スペース内のオブジェクトを参照する ID

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、STRING 型、または UNICODE 型の変数のいずれかです。ID は、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

以下に例を示します。

```
// オブジェクトが NULL であるかどうかをテストし、
// それに応じてプロセスを実行します
isNull = JavaLib.isNull( (objId)valueId );

if( isNull == 1 )
    error = 12;
end
```

JavaLib.isNull の処理中にエラーが発生すると、**SysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001001	指定した ID がオブジェクト・スペース内にありませんでした。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

- 864 ページの『EGL ライブラリー JavaLib』
- 872 ページの『getField()』
- 874 ページの『invoke()』
- 『isObjID()』
- 879 ページの『qualifiedTypeName()』
- 880 ページの『remove()』
- 881 ページの『removeAll()』
- 882 ページの『setField()』
- 884 ページの『store()』
- 886 ページの『storeCopy()』
- 887 ページの『storeField()』
- 889 ページの『storeNew()』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

isObjID()

システム関数 **JavaLib.isObjID** は、指定した ID がオブジェクト・スペース内に存在しているかどうかを示す値 (真の場合は 1、偽の場合は 0) を戻します。

JavaLib.isObjID は、Java アクセス関数の 1 つです。

```
JavaLib.isObjID(identifier javaObjId in)
returns (result INT)
```

result

数値項目。真の場合は 1、偽の場合は 0 を受け取ります。非数値項目を使用すると、検証時にエラーが発生します。

identifier

オブジェクト・スペース内のオブジェクトを参照する ID

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。ID は、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのblankと 2 バイトのblankを削除します。

以下に例を示します。

```
// オブジェクトが存在していないかどうかをテストし、  
// それに応じてプロセスを実行します  
isPresent = JavaLib.isObjID( (objId)valueId );  
  
if( isPresent == 0 )  
    error = 27;  
end
```

JavaLib.isObjID に関連付けられているランタイム・エラーはありません。

関連する概念

811 ページの『EGL 関数の構文図』

865 ページの『Java アクセス関数』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

qualifiedTypeName()

システム関数 **JavaLib.qualifiedTypeName** は、EGL Java オブジェクト・スペース内にあるオブジェクトのクラスの完全修飾名を戻します。

JavaLib.qualifiedTypeName は、Java アクセス関数の 1 つです。

```
JavaLib.qualifiedTypeName(identifier javaObjId in)  
returns (result STRING)
```

result

結果フィールドは必ず指定する必要があり、結果フィールドの型は、CHAR、MBCHAR または UNICODE でなければなりません。

- 結果フィールドの型が MBCHAR または UNICODE の場合は、常に適切な値を受け取ります。

- 結果フィールドが CHAR 型の場合、受け取る値に DBCHAR 文字に相当する文字が含まれていると、問題が発生することがあります。

長さが一致しない場合については、『代入』を参照してください。

identifier

オブジェクト・スペース内のオブジェクトを参照する ID

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。ID は、後述の例のように、objId にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

以下に例を示します。

```
myItem = JavaLib.qualifiedTypeName( (objId)"myId" );
```

JavaLib.qualifiedTypeName の処理中にエラーが発生すると、**sysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

remove()

システム関数 **JavaLib.remove** は、指定した ID を EGL Java オブジェクト・スペースから除去します。指定した ID が、オブジェクトを参照している唯一の ID である場合は、その ID に関連するオブジェクトも除去されます。別の ID がそのオブジェクトを参照している場合、オブジェクトはオブジェクト・スペース内に残ります。このオブジェクトは、その ID 経由でアクセスできます。

JavaLib.remove は、Java アクセス関数の 1 つです。

```
JavaLib.remove(identifier javaObjId in)
```

identifier

オブジェクトを参照する ID。ID が見付からなくても、エラーは発生しません。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。ID は、後述の例のように、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

以下に例を示します。

```
JavaLib.remove( (objId)myStoredObject );
```

JavaLib.remove に関連付けられているランタイム・エラーはありません。

注: システム関数 **JavaLib.remove** と **JavaLib.removeAll** を呼び出すと、Java 仮想マシンで EGL Java オブジェクト・スペース内のガーベッジ・コレクションを処理できます。システム関数を呼び出してオブジェクト・スペースからオブジェクトを除去しないと、オブジェクト・スペースにアクセスするプログラムの実行時にメモリーがリカバリーされません。

関連する概念

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

864 ページの『EGL ライブラリー **JavaLib**』

872 ページの『**getField()**』

874 ページの『**invoke()**』

877 ページの『**isNull()**』

878 ページの『**isObjID()**』

879 ページの『**qualifiedTypeName()**』

『**removeAll()**』

882 ページの『**setField()**』

884 ページの『**store()**』

886 ページの『**storeCopy()**』

887 ページの『**storeField()**』

889 ページの『**storeNew()**』

removeAll()

システム関数 **JavaLib.removeAll** は、EGL Java オブジェクト・スペースからすべての ID とオブジェクトを除去します。 **JavaLib.removeAll** は、Java アクセス関数の 1 つです。

```
JavaLib.removeAll( )
```

JavaLib.removeAll に関連付けられているランタイム・エラーはありません。

注: システム関数 **JavaLib.remove** と **JavaLib.removeAll** を呼び出すと、Java 仮想マシンで EGL Java オブジェクト・スペース内のガーベッジ・コレクションを処理できます。システム関数を呼び出してオブジェクト・スペースからオブジェクトを除去しないと、オブジェクト・スペースにアクセスするプログラムの実行時にメモリーがリカバリーされません。

関連する概念

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

setField()

システム関数 **JavaLib.setField** は、ネイティブ Java オブジェクトまたはクラス内のフィールドに値を設定します。**JavaLib.setField** は、Java アクセス関数の 1 つです。

```
JavaLib.setField(  
    identifierOrClass javaObjId in,  
    field STRING in,  
    value anyEglPrimitive in)
```

identifierOrClass

この引数は、次のエンティティのいずれかです。

- オブジェクト・スペース内のオブジェクトを参照する ID
- Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。オブジェクトの ID を指定する場合は、後述の例のように、その ID を `objID` にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

field

変更するフィールドの名前。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。大文字小文字の区別があるストリングの先頭と最後から、1 バイトのブランクと 2 バイトのブランクが削除されます。

value
値そのもの。

『Java アクセス (システム・ワード)』に記述したキャストが必要な場合があります。

Java の型変換の規則が適用されます。例えば、int として宣言されたフィールドに short を割り当てても、エラーは発生しません。

以下に例を示します。

```
JavaLib.setField( (objID)"myId", "myField",  
                (short)myNumItem );
```

JavaLib.setField の処理中にエラーが発生すると、SysVar.errorCode は以下の表の値に設定されます。

SysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001003	EGL プリミティブ型が、Java で想定される型と同じではありません。
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとしてしました。
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する概念

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの 『isNull()』
878 ページの 『isObjID()』
879 ページの 『qualifiedTypeName()』
880 ページの 『remove()』
881 ページの 『removeAll()』
『store()』
886 ページの 『storeCopy()』
887 ページの 『storeField()』
889 ページの 『storeNew()』

store()

システム関数 **JavaLib.store** は、メソッドを呼び出し、戻りオブジェクト（または NULL）を、指定した ID とともに EGL Java オブジェクト・スペースに格納します。ID がすでにオブジェクト・スペース内に存在している場合は、次のステップと同等のアクションが実行されます。

- ID を指定して **JavaLib.remove** を実行し、その ID に関連付けられたオブジェクトを除去します。
- **JavaLib.store** で戻されたオブジェクトをターゲット ID に関連付けます。

メソッドにより、オブジェクトではなく Java プリミティブが戻された場合、EGL は、プリミティブを表すオブジェクトを保管します。例えば、メソッドで `int` が戻された場合、EGL は、`java.lang.Integer` 型のオブジェクトを保管します。

JavaLib.store は、Java アクセス関数の 1 つです。

```
JavaLib.store(  
    storeId javaObjId in,  
    identifierOrClass javaObjId in,  
    method STRING in  
    {, argument anyEglPrimitive in} )
```

storeId

戻されたオブジェクトとともに保管する ID。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。ID は、後述の例のように、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

identifierOrClass

この引数は、次のエンティティのいずれかです。

- オブジェクト・スペース内のオブジェクトを参照する ID
- Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、STRING 型、または UNICODE 型の変数のいずれかです。オブジェクトの ID を指定する場合は、後述の例のように、その ID を objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

method

呼び出すメソッド。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。オブジェクトの ID を指定する場合は、後述の例のように、その ID を objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

argument

メソッドに渡される値。

『Java アクセス (システム・ワード)』に記述したキャストが必要な場合があります。

Java の型変換の規則が適用されます。例えば、int として宣言されたメソッド・パラメーターに short を渡しても、エラーは発生しません。

精度が失われるのを回避するために、Java の double には EGL の float 項目を使用し、Java の float には EGL の smallfloat 項目を使用してください。それ以外の EGL の型を使用すると、多くの場合、結果としての値が丸められます。

呼び出し側プログラムのメモリー領域は、メソッドの実行内容にかかわらず、変更されません。

以下に例を示します。

```
JavaLib.store( (objId)"storeId", (objId)"myId",  
              "myMethod", 36 );
```

JavaLib.store の処理中にエラーが発生すると、**sysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001003	EGL プリミティブ型が、Java で想定される型と同じではありません。
00001006	NULL へキャストする引数のクラスをロードできませんでした。
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとしていました。
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

『storeCopy()』

887 ページの『storeField()』

889 ページの『storeNew()』

storeCopy()

システム関数 **JavaLib.storeCopy** は、オブジェクト・スペース内の別の ID に基づいて新しい ID を作成します。これにより、同じオブジェクトを 2 つの ID で参照できるようになります。オブジェクト・スペース内にソース ID が存在しない場合は、ターゲット ID に NULL が保管されます。エラーは発生しません。ターゲット ID がすでにオブジェクト・スペース内に存在している場合は、次のステップと同等のアクションが実行されます。

- ターゲット ID を指定して **JavaLib.remove** を実行し、その ID に関連付けられたオブジェクトを除去します。
- ソース・オブジェクトをターゲット ID に関連付けます。

JavaLib.storeCopy は、Java アクセス関数の 1 つです。

```
JavaLib.storeCopy(  
    sourceId javaObjId in,  
    targetId javaObjId in)
```

sourceId

オブジェクト・スペース内のオブジェクト、または NULL を参照する ID。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の変数のいずれかです。ID は、後述の例のように、objId にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

targetId

同じオブジェクトを参照する新しい ID。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、STRING 型、または UNICODE 型の変数のいずれかです。ID は、後述の例のように、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

以下に例を示します。

```
JavaLib.storeCopy( (objId)"sourceId", (objId)"targetId" );
```

JavaLib.storeCopy に関連付けられているランタイム・エラーはありません。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

『storeField()』

889 ページの『storeNew()』

storeField()

システム関数 **JavaLib.storeField** は、クラス・フィールドまたはオブジェクト・フィールドの値を EGL Java オブジェクト・スペースに格納します。オブジェクトの保管に使用する ID がすでにオブジェクト・スペース内に存在している場合は、次のステップと同等のアクションが実行されます。

- ID を指定して **JavaLib.remove** を実行し、その ID に関連付けられたオブジェクトを除去します。
- 新しいオブジェクトを ID に関連付けます。

クラス・フィールドまたはオブジェクト・フィールドに、オブジェクトではなく Java プリミティブが含まれる場合、EGL は、プリミティブを表すオブジェクトを保管します。例えば、フィールドに `int` が含まれる場合、EGL は、`java.lang.Integer` 型のオブジェクトを保管します。

```
JavaLib.storeField(  
    storeId javaObjId in,  
    identifierOrClass javaObjIdOrClass in,  
    field STRING in)
```

storeId

オブジェクトとともに保管する ID。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。ID は、後述の例のように、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

identifierOrClass

この引数は、次のエンティティのいずれかです。

- オブジェクト・スペース内のオブジェクトを参照する ID
- Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。オブジェクトの ID を指定する場合は、後述の例のように、その ID を objID にキャストする必要があります。次の引数に static フィールドを指定する場合は、この引数にクラスを指定することをお勧めします。

EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

field

オブジェクトを参照するフィールドの名前。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。大文字小文字の区別があるストリングの先頭と最後から、1 バイトのブランクと 2 バイトのブランクが削除されます。

以下に例を示します。

```
JavaLib.storeField( (objId)"myStoreId",
    (objId)"myId", "myField");
```

JavaLib.storeField の処理中にエラーが発生すると、**sysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとしてしました。
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

『storeNew()』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

storeNew()

システム関数 **JavaLib.storeNew** は、クラスのコンストラクターを呼び出し、新しいオブジェクトを EGL Java オブジェクト・スペースに格納します。ID がすでにオブジェクト・スペース内に存在している場合は、次のステップと同等のアクションが実行されます。

- ID を指定して **JavaLib.remove** を実行し、その ID に関連付けられたオブジェクトを除去します。
- 新しいオブジェクトを ID に関連付けます。

JavaLib.storeNew は、Java アクセス関数の 1 つです。

```
JavaLib.storeNew(  
    storeId javaObjId in,  
    class STRING in  
    {, argument anyEglPrimitive in})
```

storeId

新しいオブジェクトとともに保管する ID。

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。ID は、後述の例のように、objID にキャストする必要があります。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

class

Java クラスの完全修飾名

この引数は、文字列リテラル、または CHAR 型、DBCHAR 型、MBCHAR 型、UNICODE 型の項目です。EGL は、大文字小文字の区別がある引数値の先頭と最後から、1 バイトのブランクと 2 バイトのブランクを削除します。

argument

コンストラクターに渡される値。

『Java アクセス (システム・ワード)』に記述したキャストが必要な場合があります。

Java の型変換の規則が適用されます。例えば、int として宣言されたコンストラクター・パラメーターに short を渡しても、エラーは発生しません。

精度が失われるのを回避するために、Java の double には EGL の float 項目を使用し、Java の float には EGL の smallfloat 項目を使用してください。それ以外の EGL の型を使用すると、多くの場合、結果としての値が丸められます。

呼び出し側プログラムのメモリー領域は、コンストラクターの実行内容にかかわらず、変更されません。

以下に例を示します。

```
JavaLib.storeNew( (objId)"storeId", "myClass", 36 );
```

JavaLib.storeNew の処理中にエラーが発生すると、**sysVar.errorCode** は以下の表の値に設定されます。

sysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定された名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001003	EGL プリミティブ型が、Java で想定される型と同じではありません。
00001006	NULL ヘキャストする引数のクラスをロードできませんでした。
00001007	メソッドまたはフィールドに関する情報の取得中に、SecurityException または IllegalAccessException がスローされました。または、final 宣言されたフィールドの値を設定しようとしていました。
00001008	コンストラクターを呼び出すことができません。クラス名はインターフェースまたは抽象クラスを参照しています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

864 ページの『EGL ライブラリー JavaLib』

872 ページの『getField()』

874 ページの『invoke()』

877 ページの『isNull()』

878 ページの『isObjID()』

879 ページの『qualifiedTypeName()』

880 ページの『remove()』

881 ページの『removeAll()』

882 ページの『setField()』

884 ページの『store()』

886 ページの『storeCopy()』

887 ページの『storeField()』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

EGL ライブラリー LobLib

ライブラリー LobLib の関数は以下の表のとおりです。

システム関数/呼び出し	説明
<code>attachBlobToFile(blobVariable, fileName)</code>	BLOB 型の変数が参照するデータを、指定したファイルにコピーします。
<code>attachBlobToTempFile(blobVariable)</code>	BLOB 型の変数が参照するデータを、固有の一時システム・ファイルにコピーします。
<code>attachClobToFile(clobVariable, fileName)</code>	CLOB 型の変数が参照するデータを、指定したファイルにコピーします。
<code>attachClobToTempFile(clobVariable)</code>	CLOB 型の変数が参照するデータを、固有の一時システム・ファイルにコピーします。
<code>freeBlob(blobVariable)</code>	BLOB 型の変数により使用されるリソースをリリースします。
<code>freeClob(clobVariable)</code>	CLOB 型の変数により使用されるリソースをリリースします。
<code>result = getBlobLen(blobVariable)</code>	BLOB 型の変数により参照される値のバイト数を戻します。
<code>result = getClobLen(clobVariable)</code>	CLOB 型の変数により参照される文字数を戻します。
<code>result = getStrFromClob(clobVariable)</code>	CLOB の変数により参照される値と一致する文字列を戻します。
<code>result = getSubStrFromClob(clobVariable, pos, length)</code>	CLOB 型の変数により参照される値からのサブストリングを戻します。
<code>loadBlobFromFile(blobVariable, fileName)</code>	BLOB 型の変数によって参照されるメモリー・エリアに、指定したファイルからのデータをコピーします。
<code>loadClobFromFile(blobVariable, fileName)</code>	CLOB 型の変数によって参照されるメモリー・エリアに、指定したファイルからのデータをコピーします。
<code>setClobFromString(clobVariable, str)</code>	CLOB 型の変数によって参照されるメモリー・エリアに、文字列をコピーします。
<code>setClobFromStringAtPosition(clobVariable, pos, str)</code>	CLOB 型の変数によって参照されるメモリー・エリアに、文字列をコピーします (メモリー・エリア内の指定した位置から始めます)。
<code>truncateBlob(blobVariable, length)</code>	BLOB 型の変数により参照される値を切り捨てます。
<code>truncateClob(clobVariable, length)</code>	CLOB 型の変数により参照される値を切り捨てます。
<code>updateBlobToFile(blobVariable, fileName)</code>	BLOB 型の変数が参照するデータを、指定したファイルにコピーします。

システム関数/呼び出し	説明
<code>updateClobToFile(blobVariable, fileName)</code>	CLOB 型の変数が参照するデータを、指定したファイルにコピーします。

attachBlobToFile()

システム関数 **LobLib.attachBlobToFile** は、BLOB 型の変数が参照するデータを、指定したファイルにコピーします。

```
LobLib.attachBlobToFile(
    blobVariable BLOB inOut,
    fileName STRING in)
```

blobVariable

BLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

attachBlobToTempFile()

システム関数 **LobLib.attachBlobToTempFile** は、BLOB 型の変数が参照するデータを、固有の一時システム・ファイルにコピーします。この関数は、実行時に使用されるメモリーを最小化します。

```
LobLib.attachBlobToTempFile(blobVariable BLOB in)
```

blobVariable

BLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

attachClobToFile()

システム関数 **LobLib.attachClobToFile** は、CLOB 型の変数が参照するデータを、指定したファイルにコピーします。

```
LobLib.attachClobToFile(
    clobVariable CLOB inOut,
    fileName STRING in)
```

clobVariable

CLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

attachClobToTempFile()

システム関数 **LobLib.attachClobToTempFile** は、CLOB 型の変数が参照するデータを、固有の一時システム・ファイルにコピーします。この関数は、実行時に使用されるメモリーを最小化します。

LobLib.attachClobToTempFile(*clobVariable* CLOB in)

clobVariable

CLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

freeBlob()

システム関数 **LobLib.freeBlob** は、BLOB 型の変数により使用された任意のリソースをリリースします。

LobLib.freeBlob(*blobVariable* BLOB inOut)

blobVariable

BLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

freeClob()

システム関数 **LobLib.freeClob** は、CLOB 型の変数により使用されたリソースをリリースします。

LobLib.freeClob(*clobVariable* CLOB inOut)

clobVariable

CLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

getBlobLen()

システム関数 **LobLib.getBlobLen** は、BLOB 型の変数により参照された値のバイト数を返します。

```
LobLib.getBlobLen(blobVariable BLOB in)  
returns (result BIGINT)
```

result

バイト数。

blobVariable

BLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

getClobLen()

システム関数 **LobLib.getClobLen** は、CLOB 型の変数により参照された文字数を返します。

```
LobLib.getClobLen(clobVariable CLOB in)  
returns (result BIGINT)
```

result

文字数。

blobVariable

CLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

getStrFromClob()

システム関数 **LobLib.getStrFromClob** は、CLOB 型の変数により参照された値と一致する文字列を返します。

```
LobLib.getStrFromClob(clobVariable CLOB in)  
returns (result STRING)
```

result

戻された文字列。

clobVariable

CLOB 型の変数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

getSubStrFromClob()

システム関数 **LobLib.getSubStrFromClob** は、CLOB 型の変数により参照される値からのサブストリングを戻します。

```
LobLib.getSubStrFromClob(  
  clobVariable CLOB in,  
  pos BIGINT in,  
  length BIGINT in)  
returns (result STRING)"
```

result

STRING 型の値。

clobVariable

CLOB 型の変数。

pos

サブストリングを開始する文字の数値位置を識別します。 CLOB 変数の先頭文字は位置 1 にあります。

length

サブストリングの文字数を識別します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

loadBlobFromFile()

システム関数 **LobLib.loadBlobFromFile** は、BLOB 型の変数によって参照されるメモリー・エリアに、指定したファイルからのデータをコピーします。

```
LobLib.loadBlobFromFile(  
  blobVariable BLOB inOut,  
  fileName STRING in)
```

blobVariable

BLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

loadClobFromFile()

システム関数 **LobLib.loadClobFromFile** は、CLOB 型の変数によって参照されるメモリー・エリアに、指定したファイルからのデータをコピーします。

```
LobLib.loadClobFromFile(  
    clobVariable CLOB inOut,  
    fileName STRING in)
```

clobVariable

CLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

setClobFromString()

システム関数 **LobLib.setClobFromString** は、CLOB 型の変数によって参照されるメモリー・エリアに、文字列をコピーします。

```
LobLib.setClobFromString(  
    clobVariable CLOB inOut,  
    str STRING in)
```

clobVariable

CLOB 型の変数。

str コピーされる文字列。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

setClobFromStringAtPosition()

システム関数 **LobLib.setClobFromStringAtPosition** は、CLOB 型の変数によって参照されるメモリー・エリアに、文字列をコピーします (メモリー・エリアの指定した位置から始めます)。

```
LobLib.setClobFromStringAtPosition(  
  clobVariable CLOB inOut,  
  pos BIGINT in  
  str STRING in)
```

clobVariable

CLOB 型の変数。

pos

clobVariable により参照される値の文字位置。 CLOB 変数の先頭文字は位置 1 にあります。

str コピーされる文字列。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

truncateBlob()

システム関数 **LobLib.truncateBlob** は、BLOB 型の変数により参照された値を切り捨てます。

```
LobLib.truncateBlob(  
  blobVariable BLOB inOut,  
  length BIGINT in)
```

blobVariable

BLOB 型の変数。

length

出力のバイト数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

truncateClob()

システム関数 **LobLib.truncateClob** は、CLOB 型の変数により参照された値を切り捨てます。

```
LobLib.truncateClob(  
  clobVariable CLOB inOut,  
  length BIGINT in)
```

clobVariable

CLOB 型の変数。

length

出力の (文字数ではなく) バイト数。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

updateBlobToFile()

システム関数 **LobLib.updateBlobToFile** は、BLOB 型の変数が参照したデータを、指定したファイルにコピーします。ファイルが存在する場合、この関数はまずファイルの内容を消去します。存在しない場合、この関数はファイルを作成します。

```
LobLib.updateBlobToFile(  
    blobVariable BLOB inOut,  
    fileName STRING in)
```

blobVariable

BLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

52 ページの『BLOB』

891 ページの『EGL ライブラリー LobLib』

updateClobToFile()

システム関数 **LobLib.updateClobToFile** は、CLOB 型の変数が参照したデータを、指定したファイルにコピーします。ファイルが存在する場合、この関数はまずファイルの内容を消去します。存在しない場合、この関数はファイルを作成します。

```
LobLib.updateClobToFile(  
    clobVariable CLOB inOut,  
    fileName STRING in)
```

clobVariable

CLOB 型の変数。

fileName

ファイルの名前。名前は完全に修飾されているか、またはプログラムが起動されたディレクトリーを基準にしています。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

51 ページの『CLOB』

891 ページの『EGL ライブラリー LobLib』

EGL ライブラリー MathLib

次の表は、システム・ライブラリー MathLib 内の関数を示しています。

注: *numericField* フィールドの型は、

BIGINT、BIN、DECIMAL、HEX、INT、NUM、

NUMC、PACF、SMALLINT、FLOAT、または SMALLFLOAT です。

HEX 型 (長さ 8) のフィールドは、ランタイム環境に固有な単精度の 4 バイト浮動小数点数であると想定されています。HEX 型 (長さ 16) のフィールドは、ランタイム環境に固有な倍精度の 8 バイト浮動小数点数であると想定されています。

システム関数/呼び出し	説明
<i>result</i> = abs (<i>numericField</i>)	<i>numericField</i> の絶対値を返す
<i>result</i> = acos (<i>numericField</i>)	<i>numericField</i> のアークコサインを返す
<i>result</i> = asin (<i>numericField</i>)	<i>numericField</i> のアークサインを返す
<i>result</i> = atan (<i>numericField</i>)	<i>numericField</i> のアークタンジェントを返す
<i>result</i> = atan2 (<i>numericField1</i> , <i>numericField2</i>)	戻り値の四分円を判別するために、両方の引数の符号を使用して、 <i>numericField1/numericField2</i> のアークタンジェントの主値を計算する
<i>result</i> = ceiling (<i>numericField</i>)	<i>numericField</i> 以上の最も小さい整数を返す
<i>result</i> = compareNum (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> が <i>numericField2</i> より小さいか、同等か、大きいかを示す結果 (-1、0、または 1) を返す
<i>result</i> = cos (<i>numericField</i>)	<i>numericField</i> のコサインを返す
<i>result</i> = cosh (<i>numericField</i>)	<i>numericField</i> の双曲線コサインを返す
<i>result</i> = exp (<i>numericField</i>)	<i>numericField</i> の指数関数値を返す
<i>result</i> = floatingAssign (<i>numericField</i>)	<i>numericField</i> を倍精度浮動小数点数として返す
<i>result</i> = floatingDifference (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> と <i>numericField2</i> の差を返す
<i>result</i> = floatingMod (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> を <i>numericField2</i> で割った余りを浮動小数点で計算し、 <i>numericField1</i> と同じ符号を付ける
<i>result</i> = floatingProduct (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> と <i>numericField2</i> の積を返す
<i>result</i> = floatingQuotient (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> を <i>numericField2</i> で割った商を返す
<i>result</i> = floatingSum (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> と <i>numericField2</i> の合計を返す
<i>result</i> = floor (<i>numericField</i>)	<i>numericField</i> 以下の最も大きい整数を返す

システム関数/呼び出し	説明
<i>result</i> = frexp (<i>numericField</i> , <i>integer</i>)	数値を、.5 から 1 (戻り値) までの範囲の正規化された部分と 2 (<i>integer</i> で戻される) の累乗に分割する
<i>result</i> = Ldexp (<i>numericField</i> , <i>integer</i>)	<i>numericField</i> に 2 の <i>integer</i> 乗を掛けた数値を返す
<i>result</i> = log (<i>numericField</i>)	<i>numericField</i> の自然対数を返す
<i>result</i> = log10 (<i>numericField</i>)	<i>numericField</i> の、底を 10 とする対数を返す
<i>result</i> = maximum (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> と <i>numericField2</i> のうちの大きい方を返す
<i>result</i> = minimum (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> と <i>numericField2</i> のうちの小さい方を返す
<i>result</i> = modf (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> を整数部分と小数部分に分け、両方に <i>numericField1</i> と同じ符号を付けて、整数部分を <i>numericField2</i> に配置し、小数部分を返す
<i>result</i> = pow (<i>numericField1</i> , <i>numericField2</i>)	<i>numericField1</i> の <i>numericField2</i> 乗を返す
<i>result</i> = precision (<i>numericField</i>)	<i>numericField</i> の最大精度 (10 進数) を返す
<i>result</i> = round (<i>numericField</i> [, <i>integer</i>]) <i>result</i> = mathLib.round(<i>numericExpression</i>)	数値または式を最も近い値 (最も近い 1,000 の倍数など) に丸めて、結果を返す
<i>result</i> = sin (<i>numericField</i>)	<i>numericField</i> のサインを返す
<i>result</i> = sinh (<i>numericField</i>)	<i>numericField</i> の双曲線サインを返す
<i>result</i> = sqrt (<i>numericField</i>)	<i>numericField</i> がゼロ以上の場合、 <i>numericField</i> の平方根を返す
<i>result</i> = stringAsDecimal (<i>numberAsText</i>)	文字の値 ("98.6" など) を受け入れ、DECIMAL 型と同等の値を返す
<i>result</i> = stringAsFloat (<i>numberAsText</i>)	文字の値 ("98.6" など) を受け入れ、FLOAT 型と同等の値を返す
<i>result</i> = stringAsInt (<i>numberAsText</i>)	文字の値 ("98.6" など) を受け入れ、BIGINT 型と同等の値を返す
<i>result</i> = tan (<i>numericField</i>)	<i>numericField</i> のタンジェントを返す
<i>result</i> = tanh (<i>numericField</i>)	<i>numericField</i> の双曲線タンジェントを返す

abs()

システム関数 **MathLib.abs** は、数値の絶対値を返します。

```
MathLib.abs(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 *numericItem* の絶対値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

MathLib.abs は、すべてのターゲット・システムで機能します。Java プログラムとの関係で、EGL は Java StrictMath クラスで `abs()` メソッドを使用して、ランタイムの振る舞いがすべての Java 仮想マシンで同じになるようにします。

例:

```
myItem = -5;
result = MathLib.abs(myItem); // result = 5
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

acos()

システム関数 **MathLib.acos** は、引数のアークコサインをラジアンで戻します。

```
MathLib.acos(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。戻り値 (0.0 からパイの間) はラジアン単位で、*result* の形式に変換されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、計算が行われる前に倍精度の浮動小数点数に変換されます。値が -1 から 1 の範囲でない場合は、エラーが発生します。

MathLib.acos は、すべてのターゲット・システムで機能します。Java プログラムとの関係で、EGL は Java StrictMath クラスで `acos()` メソッドを使用して、ランタイムの振る舞いがすべての Java 仮想マシンで同じになるようにします。

例:

```
result = MathLib.acos(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

asin()

システム関数 **MathLib.asin** は、-1 から 1 までの範囲の数値のアークサインを戻します。結果は、ラジアンで戻され、-pi/2 から pi/2 までの範囲になります。

```
MathLib.asin(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。MathLib.asin 関数から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、`mathLib.asin` 関数が呼び出される前に倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.asin(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

atan()

システム関数 **MathLib.atan** は、数値のアーктanジェントを戻します。結果は、ラジアンで戻され、 $-\pi/2$ から $\pi/2$ までの範囲にあります。

```
MathLib.atan(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.atan** から戻された値は、*result* の形式に変換されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、**MathLib.atan** が呼び出される前に倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.atan(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

atan2()

システム関数 **MathLib.atan2** は、戻り値の四分円を判別するために、両方の引数の符号を使用して、 y/x のアーктanジェントの基本値を計算します。結果は、ラジアンで戻され、 $-\pi$ から π までの範囲にあります。

```
MathLib.atan2(
  numericField1 mathLibNumber in,
  numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.atan2** から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、**MathLib.atan2** が呼び出される前に倍精度の浮動小数点数に変換されます。*numericField1* は y 値です。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、**MathLib.atan2** が呼び出される前に倍精度の浮動小数点数に変換されます。*numericField2* は x 値です。

例:

```
myItemY = 1;
myItemX = 5;

// returns pi/2
result = MathLib.atan2(myItemY, myItemX);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

ceiling()

システム関数 **MathLib.ceiling** は指定された数以上の整数のなかで最小の整数を戻します。

```
MathLib.ceiling(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。*numericItem* 以上の最短整数は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
myItem = 4.5;
result = MathLib.ceiling(myItem); // result = 5
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

compareNum()

システム関数 **MathLib.compareNum** は、2 つの値のうちの最初の値が 2 番目の値より小さいか、同等か、あるいは大きいかを示す結果 (-1、0、または 1) を戻します。

```
MathLib.compareNum(
    numericField1 mathLibNumber in,
    numericField2 mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

タイプ INT として、または、長さ 9 で小数点以下の桁のないタイプ BIN と同等のものとして定義されます。この項目は、次の値のうちの 1 つを受信します。

- 1 *numericField1* は *numericField2* より小さい
- 0 *numericField1* は *numericField2* と同等
- 1 *numericField1* は *numericField2* より大きい

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
myItem01 = 4
myItem02 = 7

result = MathLib.compareNum(myItem01,myItem02);

// result = -1
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

cos()

システム関数 **MathLib.cos** は、数値のコサインを戻します。戻り値は、-1 から 1 の範囲になります。

```
MathLib.cos(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.cos** から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、**MathLib.cos** が呼び出される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.cos(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

cosh()

システム関数 **MathLib.cosh** は、数値の双曲線コサインを戻します。

```
MathLib.cosh(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **mathLib.cosh** から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 この項目は、**mathLib.cosh** が呼び出される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.cosh(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

exp()

システム関数 **MathLib.exp** は、 e をある数値で累乗した値を戻します。

```
MathLib.exp(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『*MathLib*』で説明されている任意の数値項目または HEX 項目。 **MathLib.exp** から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『*MathLib*』で説明されている任意の数値項目または HEX 項目。 この項目は、**MathLib.exp** が呼び出される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.exp(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingAssign()

システム関数 **MathLib.floatingAssign** は、*numericItem* を倍精度浮動小数点数として戻します。この関数は、BIN、DECIMAL、NUM、NUMC、または PACKF 項目の値を HEX 項目として定義された浮動小数点数に代入したり、その逆に、HEX 項目の浮動小数点数をこれらの値に代入したりします。

```
MathLib.floatingAssign(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。浮動小数点数は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、結果に割り当てられる前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingAssign(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingDifference()

システム関数 **MathLib.floatingDifference** は、2 つの数値の最初の数値から 2 番目の数値を減算して、その差を戻します。この関数は、倍精度の浮動小数点算術計算を使用して実装されます。

```
MathLib.floatingDifference(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。差は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、差が計算される前に、倍精度の浮動小数点数に変換されます。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、差が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingDifference(myItem01,myItem02);
```


関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingMod()

システム関数 **MathLib.floatingMod** は、もう 1 つの値で除算された値の浮動小数点の剰余を返します。結果には、分子と同じ符号が付きます。分母が 0 であると、ドメイン例外が発生します。

```
MathLib.floatingMod(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。浮動小数点の剰余は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingMod(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingProduct()

システム関数 **MathLib.floatingProduct** は、2 つの数値の積を返します。この関数は、倍精度の浮動小数点算術計算を使用して実装されます。

```
MathLib.floatingProduct(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。積は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingProduct(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingQuotient()

システム関数 **MathLib.floatingQuotient** は、もう 1 つの値で除算された値の商を戻します。分母が 0 であると、ドメイン例外が発生します。この関数は、倍精度の浮動小数点算術計算を使用して実装されます。

```
MathLib.floatingQuotient(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。商は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、商が計算される前に、倍精度の浮動小数点数に変換されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、商が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingQuotient(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floatingSum()

システム関数 **MathLib.floatingSum** は、2 つの数値の合計を戻します。この関数は、倍精度の浮動小数点算術計算を使用して実装されます。

```
MathLib.floatingSum(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。合計は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、合計が計算される前に、倍精度の浮動小数点数に変換されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、合計が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.floatingSum(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

floor()

システム関数 **MathLib.floor** は、指定された数以下の整数のなかで最大の整数を戻します。

```
MathLib.floor(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 *numericField* 以下の最長整数は、*result* の形式に変換され、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
myItem = 4.6;  
result = MathLib.floor(myItem); // result = 4
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

frexp()

システム関数 **MathLib.frexp** は、数値を .5 から 1 までの範囲の正規化された小数部 (*result* として戻されます) と 2 の累乗 (*exponent* で戻されます) に分割します。

```
MathLib.frexp(  
    numericField mathLibNumber in,  
    exponent mathLibInteger inOut)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。浮動小数点の小数部は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

exponent

タイプ INT として、または、長さ 9 で小数点以下の桁のないタイプ BIN と同等のものとして定義されます。

例:

```
result = MathLib.frexp(myItem,myInteger);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

Ldexp()

システム関数 **MathLib.Ldexp** は、指定された値に 2 の *exponent* 乗を乗算した値を戻します。

```
MathLib.Ldexp(  
    numericField mathLibNumber in,  
    exponent mathLibInteger in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。計算値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

exponent

タイプ INT として、または、長さ 9 で小数点以下の桁のないタイプ BIN と同等のものとして定義されます。

例:

```
result = MathLib.Ldexp(myItem,myInteger);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

log()

システム関数 **MathLib.log** は、数値の自然対数を返します。

```
MathLib.log(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **mathLib.log** 関数から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.log(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

log10()

システム関数 **MathLib.log10** は、数値の底 10 の対数を返します。

```
MathLib.log10(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **log10** 関数から戻された値は、*result* の形式に変換されて、*result* に戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.log10(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

maximum()

システム関数 **MathLib.maximum** は、2 つの数値のうちの大きいほうの値を返します。

```
MathLib.maximum(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。2 つの数値のうちの大きいほうの値は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
result = MathLib.maximum(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

minimum()

システム関数 **MathLib.minimum** は、2 つの数値のうちの小さいほうの値を返します。

```
MathLib.minimum(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。2 つの数値のうちの小さいほうの値は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
result = MathLib.minimum(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

modf()

システム関数 **MathLib.modf** は、数値を整数部分と小数部分に分け、その両方に数値と同じ符号を付けます。小数部分は *result* に戻され、整数部分は *numericField2* に戻されます。

```
MathLib.modf(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber inOut)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 *numericField1* の小数部分は *result* の形式に変換され、*result* で戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 *numericField1* の整数部分は、*numericField2* の形式に変換されて、*numericField2* で戻されます。

例:

```
result = MathLib.modf(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

pow()

システム関数 **MathLib.pow** は、ある数値を 2 番目の数値で累乗した数に戻します。pow(x,y) の x の値が負で y が整数でない場合、または、x の値が 0.0 で y が負の場合は、ドメイン例外が発生します。

```
MathLib.pow(  
    numericField1 mathLibNumber in,  
    numericField2 mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **mathLib.pow** 関数の結果値は、*result* の形式に変換されて、*result* に戻されます。

numericField1

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

numericField2

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.pow(myItem01,myItem02);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

precision()

システム関数 **MathLib.precision** は、数値の最大精度 (10 進数) を戻します。浮動小数点数 (標準精度浮動小数点数なら 8 桁 HEX 項目、倍精度浮動小数点数なら 16 桁 HEX 項目) の場合、精度は、10 進数の最大数で、この数字はプログラムが稼働しているシステムの番号で表すことができます。

```
MathLib.precision(numericField mathLibNumber in)  
returns (result INT)
```

result

numericItem の精度を受信する項目。 *result* 項目は、タイプ INT として、または、長さ 9 で小数点以下の桁のないタイプ BIN と同等のものとして定義されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

例:

```
result = MathLib.precision(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

round()

システム関数 **MathLib.round** は、数値または式を任意の単位 (例えば、千単位) に丸め、結果を戻します。

```
MathLib.round(  
  numericField mathLibNumber in  
  [, powerOf10 mathLibInteger in  
  ]  
)  
returns (result mathLibTypeDependentResult)
```

MathLib.round(*numericExpression* **anyNumericExpression** **in**
returns (*result* **mathLibTypeDependentResult**)

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。丸め操作によって生成された値は、*result* の形式に変換されて、*result* に戻されます。

この場合、丸め操作は以下のように行われるため、サポートされる最大長は、32 ではなく 31 です。

- 結果の数字の精度より 1 高い精度で、*result* の数字に 5 を加算する
- 結果を切り捨てる

計算に 31 を超える桁数が使用され、EGL が開発時に違反を判別できないと、実行時に数値のオーバーフローが発生します。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。

numericExpression

単純な数値項目以外の数値表現。演算子を指定すると、*integer* の値は指定できません。

剰余演算子 (%) とともに **MathLib.round** を使用することはできません。

powerOf10

数値をどの値に切り上げるかを決める整数

- その整数が正の場合、その数値は 10 の *powerOf10* 乗単位で丸められます。例えば、整数が 3 の場合、その数値は千単位で丸められます。
- 整数がゼロまたは負の場合も同様になります。その場合、数値は、指定された小数点以下の桁数に丸められます。

powerOf10 を指定しないと、**MathLib.round** は *result* の小数点以下の桁数に丸められます。

その整数は、INT 型か、長さが 9 で小数部がない BIN 型と同等と定義されます。

例: 次の例では、*balance* を 1000 の位の値に丸めます。

```
balance = 12345.6789;  
rounder = 3;  
balance = MathLib.round(balance, rounder);  
// balance の値は 12000.0000 です
```

次の例では、*rounder* 値を -2 にして、*balance* を小数点以下 2 桁に丸めます。

```
balance = 12345.6789;  
rounder = -2;  
balance = mathLib.round(balance, rounder);  
// balance の値は 12345.6800 です
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

sin()

システム関数 **MathLib.sin** は、数値のサインを戻します。結果は、-1 から 1 の範囲になります。

```
MathLib.sin(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.sin** 関数から戻された値は、*result* の形式に変換されて、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.sin(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

sinh()

システム関数 **MathLib.sinh** は、数値の双曲線サインを戻します。

```
MathLib.sinh(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.sinh** 関数から戻された値は、*result* の形式に変換されて、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.sinh(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

sqrt()

数学関数 **MathLib.sqrt** は、数値の平方根を戻します。この関数は、ゼロ以上の数値に対して機能します。

```
MathLib.sqrt(numericField mathLibNumber in)  
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.sqrt** 関数から戻された値は、*result* の形式に変換されて、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.sqrt(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

stringAsDecimal()

システム関数 **MathLib.stringAsDecimal** は、文字値 ("98.6" など) を受け入れ、タイプ DECIMAL と等価の値を戻します。

```
MathLib.stringAsDecimal(numberAsText STRING in)  
returns (result DECIMAL)
```

result

タイプ DECIMAL の値。受信フィールドは、任意の小数部の桁および長さにすることができます。

EGL では、小数点の両側を最大 32 桁にすることができます。小数点 (ある場合) は、Java ロケールに固有のものです。

異なるタイプのフィールドへの数値の割り当ての影響については、『割り当て』を参照してください。

numberAsText

イニシャル符号文字を含むことが出来る文字フィールドまたはリテラル・ストリング。

例:

```
myField = "-5.243";  
  
// result = -5.243  
result = MathLib.stringAsDecimal(myField);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

407 ページの『代入』

899 ページの『EGL ライブラリー MathLib』

stringAsFloat()

システム関数 **MathLib.stringAsFloat** は、文字値 ("98.6" など) を受け入れ、タイプ **FLOAT** と等価の値を戻します。

```
MathLib.stringAsFloat(numberAsText STRING in)  
returns (result FLOAT)
```

result

タイプ **FLOAT** の値。受信フィールドは、任意の小数部の桁および長さにすることができます。小数点 (ある場合) は、Java ロケールに固有のものです。

異なるタイプのフィールドへの数値の割り当ての影響については、『割り当て』を参照してください。

numberAsText

イニシャル符号文字を含むことができる文字フィールドまたはリテラル・ストリング。

例:

```
myField = "-5.243";  
  
// result = -5.243  
result = MathLib.stringAsFloat(myField);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

407 ページの『代入』

899 ページの『EGL ライブラリー MathLib』

stringAsInt()

システム関数 **MathLib.stringAsInt** は、文字値 ("98" など) を受け入れ、タイプ **BIGINT** と等価の値を戻します。

```
MathLib.stringAsInt(numberAsText STRING in)  
returns (result BIGINT)
```

result

タイプ **BIGINT** の値。

異なるタイプのフィールドへの数値の割り当ての影響については、『割り当て』を参照してください。

numberAsText

イニシャル符号文字を含むことが出来る文字フィールドまたはリテラル・ストリング。

例:

```
myField = "-5";  
  
// result = -5  
result = MathLib.stringAsInt(myField);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

tan()

システム関数 **MathLib.tan** は、数値のタンジェントを戻します。

```
MathLib.tan(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.tan** 関数から戻された値は、*result* の形式に変換されて、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.tan(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

tanh()

システム関数 **MathLib.tanh** は、数値の双曲線タンジェントを戻します。結果は、-1 から 1 の範囲になります。

```
MathLib.tanh(numericField mathLibNumber in)
returns (result mathLibTypeDependentResult)
```

result

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。 **MathLib.tanh** 関数から戻された値は、*result* の形式に変換されて、*result* で戻されます。

numericField

『数学的 (システム・ワード)』で説明されている任意の数値項目または HEX 項目。この項目は、*result* が計算される前に、倍精度の浮動小数点数に変換されます。

例:

```
result = MathLib.tanh(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

899 ページの『EGL ライブラリー MathLib』

recordName.resourceAssociation

プログラムがレコードに対する入出力操作を行わない場合、入出力はレコード特定変数 **recordName.resourceAssociation** 内に名前がある物理ファイル上で行われます。この変数は、生成時に使用される **resourceAssociation** パーツに基づいて初期化されます。詳しくは、『リソース関連とファイル・タイプ』を参照してください。システム・リソース名は、実行時に、異なる値を **resourceAssociation** に配置することによって変更できます。

大抵の場合、構文 **recordName.resourceAssociation** を使用する必要があります。ただし、以下のそれぞれの場合が真であるため、EGL がユーザーが意図したレコードを判別できる場合は、レコード名を指定する必要はありません。

- 入出力は、プログラム内の 1 つのレコードに対してのみ実行される。
- **resourceAssociation** は、1 つのレコードに対してのみ入出力を実行する関数で使用される。
- 入出力は、ターゲットのレコードがすべて同じファイル名である場合に、プログラム内の複数のレコードに対して実行される。この場合、入出力オブジェクトとして表示される最初のレコードは暗黙の修飾子として使用されます。

resourceAssociation は、以下のいずれかとして使用できます。

- **assignment** ステートメントのソースまたはターゲット・オペランド
- **case**、**if**、または **while** ステートメントでの論理式における項目
- **return** または **exit** 文内の引数

resourceAssociation には、以下のような特性があります。

プリミティブ型

CHAR

データ長

ファイル・タイプによって異なる

複数のセグメントにわたって保管されるかどうか

はい

定義に関する考慮事項

recordName.resourceAssociation に移される値は、プログラムの生成時に指定されたシステムとファイル・タイプに有効なシステム・リソース名でなければなりません。同じファイル名を指定したレコードが複数存在する場合、そのファイル名を持ついずれかのレコードで **resourceAssociation** が変更されると、同じファイル名を持つプログラム内のすべてのレコードで **resourceAssociation** の設定が変更されます。

resourceAssociation の設定で識別されたシステム・リソースがレコード特定変数が変更されるときにオープンしていると、その変数内に存在したシステム・リソースは後続の環境ではクローズされます。I/O オプションは、**resourceAssociation** を修飾するレコードと同じ EGL ファイル名を持つレコードに対して実行されます。

2 つのプログラムで同じ EGL ファイル名が使用されている場合は、レコード固有の **resourceAssociation** 変数に同じ値が含まれていなければなりません。それ以外の場合は、新しいシステム・リソースがオープンするときに、以前にオープンされていたシステム・リソースがクローズされます。

resourceAssociation の別の値との比較は、突き合わせが全く一致する場合にのみ真と判別されます。例えば、小文字を使用して **resourceAssociation** を初期化する場合、その小文字の値は小文字にのみ突き合わせします。

プログラム間でのファイル共用: システム・リソース名は、生成時またはランタイムのいずれかに設定することができます。

生成時に設定する場合

同じ実行単位内の 2 つのプログラムで同じ論理ファイルがアクセスされる場合は、生成時にその論理ファイルに対して同じシステム・リソース名を指定し、ランタイムに両方のプログラムで同じ物理ファイル物理ファイルがアクセスされるようにする必要があります。

ランタイムに設定する場合

recordName.resourceAssociation を使用する場合は、ファイルにアクセスする各プログラムに、そのファイルの **resourceAssociation** を設定する必要があります。同じ実行単位内の 2 つのプログラムで同じ論理ファイルがアクセスされる場合は、各プログラムで **resourceAssociation** を同じシステム・リソース名に設定して、ランタイムに両方のプログラムで同じ物理ファイル物理ファイルがアクセスされるようにする必要があります。

複数のプログラムでシステム・リソースを共用する場合は、そのシステム・リソースにアクセスする各プログラムで、**resourceAssociation** が同じリソースを参照するように設定する必要があります。また、同じ実行単位内の 2 つのプログラムで同じ論理ファイルがアクセスされる場合は、プログラムの生成時に各プログラムで **resourceAssociation** を同じシステム・リソース名に設定して、ランタイムに両方のプログラムで同じシステム・リソースがアクセスされるようにする必要があります。

MQ レコード: MQ レコードのシステム・リソース名は、キュー・マネージャー名とキュー名を定義します。名前は、以下の形式で指定してください。

queueManagerName:queueName

queueManagerName

キュー・マネージャーの名前。

queueName

キューの名前。

表示されているように、複数の名前はコロンで区切ります。ただし、*queueManagerName* とコロンは省略できます。システム・リソース名は、**resourceAssociation** 項目で初期値として使用され、レコードに関連付けられているデフォルトのキューを識別します。詳細については、『*MQSeries* のサポート』を参照してください。

例

```
if (process == 1)
  myrec.resourceAssociation = "myFile.txt";
else
  myrec.resourceAssociation = "myFile02.txt";
end
```

関連する概念

285 ページの『MQSeries のサポート』

331 ページの『リソース関連とファイル・タイプ』

関連する参照項目

EGL ライブラリー ReportLib

EGL Report ライブラリー *ReportLib* は、JasperReports ライブラリーとの相互作用に必要なすべてのコンポーネントが含まれているフレームワークを確立するシステム・ライブラリーです。EGL Report ライブラリーには、次のコンポーネントが含まれています。

- 次の目的で使用される関数、変数、および定数:
 - JasperReports ライブラリー関数と相互作用する
 - レポートのデータ・ソースを定義、設定、および検索する
 - 埋め込まれたレポートを、さまざまなファイル形式にエクスポートする
 - レポートのコンテンツを操作し、レポート・データを処理する
- レポート設計、埋め込まれたレポート、およびエクスポートされたレポートを保管するファイルの名前を含むレコード
- レポート

Report ライブラリーには、次の関数が含まれています。

システム関数/呼び出し	説明
<code>addReportParameter(report, parameterString, parameterValue)</code>	指定されたデータ・ソースを使用してレポートを埋め込みます。
<code>fillReport(report, source)</code>	埋め込まれたレポートを指定の形式でエクスポートします。
<code>exportReport(report, format)</code>	レポートのパラメーター・リストに値を追加します。
<code>resetReportParameters(report)</code>	特定のレポートに使用されるパラメーターをすべて除去します。

以下の関数はレポート・ハンドラー中でのみ起動されます。

システム関数/呼び出し	説明
<code>addReportData(rd, dataSetName)</code>	指定された名前を持つレポート・データ・オブジェクトを、現在のレポート・ハンドラーに追加します。
<code>result = getReportData(dataSetName)</code>	指定された名前を持つレポート・データ・レコードを検索します。戻り値は <code>ReportData</code> 型です。
<code>result = getReportParameter(parameter)</code>	埋め込み中のレポートから、指定のパラメーターの値を戻します。
<code>result = getFieldValue(fieldName)</code>	現在処理中の行について、指定のフィールド値を戻します。戻り値は <code>ANY</code> 型です。

システム関数/呼び出し	説明
<i>result</i> = <code>getReportVariableValue(variable)</code>	埋め込み中のレポートから、指定の変数の値を返します。戻り値は ANY 型です。
<code>setReportVariableValue(variable, value)</code>	指定の変数の値を、提供された値に設定します。

注: EGL レポートを削除する場合、そのレポートのすべての参照を除去する必要があります。

関連する概念

228 ページの『データ・ソース』

226 ページの『EGL レポート作成プロセスの概要』

225 ページの『EGL レポートの概要』

addReportData()

システム関数 **ReportLib.addReportData** は、指定された名前を持つレポート・データ・オブジェクトを現在のレポート・ハンドラーに追加します。

```
ReportLib.addReportData(  
    rd ReportData in,  
    dataSetName STRING in)
```

rd

dataSetName

関連する概念

811 ページの『EGL 関数の構文図』

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

922 ページの『EGL ライブラリー ReportLib』

『addReportParameter()』

925 ページの『fillReport()』

924 ページの『exportReport()』

addReportParameter()

ReportLib.addReportParameter 関数の構文図は、次のとおりです。

```
ReportLib.addReportParameter(  
    report Report in,  
    parameterString STRING in,  
    parameterValue any in)
```

report

レポートの名前。

parameterString

レポートで使用されるパラメーター。

parameterValue

parameterString で指定されたパラメーターの値。

レポートに埋め込む前に、EGL は、 レポートで使用される値を確立するか、XML レポート設計で指定されるパラメーターをオーバーライドする、1 組のパラメーターを渡すことができます。 **ReportLib.addReportParameter** 関数は、指定されたパラメーターの値を、レポートのパラメーター・リストに追加します。

注: JasperReports パラメーターおよびデータ・タイプに関する情報については、JasperReports の資料を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

EGL レポートの概要

EGL レポート作成プロセスの概要

関連する参照項目

EGL レポート・ライブラリー

ReportLib.fillReport 関数

ReportLib.exportReport 関数

ReportLib.resetReportParameters 関数

exportReport()

システム関数 **ReportLib.exportReport** は埋め込まれたレポートを指定の形式でエクスポートします。

次の図は、この関数の構文を示します。

```
ReportLib.exportReport(  
  report Report in,  
  format ExportFormat in)
```

report

エクスポートされるレポート。

format

エクスポートされるレポートのフォーマットおよびファイル拡張子。

値は **ExportFormat** の列挙型の値です。

csv

出力が示すように、値は次の値とコンマで区切られ、**csv** はコンマで区切った値を表します。

html

出力は HTML 形式です。

pdf

出力は Adobe Acrobat PDF 形式です。

text

出力は ASCII テキスト形式です。

関連する概念

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

524 ページの『EGL での列挙型』

811 ページの『EGL 関数の構文図』

関連するタスク

245 ページの『レポートのエクスポート』

関連する参照項目

923 ページの『addReportParameter()』

922 ページの『EGL ライブラリー ReportLib』

『fillReport()』

927 ページの『resetReportParameters()』

fillReport()

ReportLib.fillReport 関数の構文図は、次のとおりです。

```
ReportLib.fillReport(  
    report Report in,  
    source DataSource in)
```

report

データが埋め込まれるレポート。

source

レポートに埋め込むために使用されるデータ・ソース。

reportData タイプの変数のレポートへの関連付けの方法を示した次の例を参照してください。

```
eglReport    Report;  
eglReportData ReportData;  
eglReport.reportData = eglReportData;
```

source は、**ReportData** タイプの変数で使用するフィールドを示します。*source* の各値はフィールド名ではなく、列挙型 **DataSource** の値です。

databaseConnection

次の例のように、**reportData** 変数の **connectionName** フィールドで参照される変数を使用します。

```
eglReportData.connectionName = "mycon";
```

ここでは、データにアクセスする SQL ステートメントはレポート設計ファイルにあり、これは EGL の外部で作成されます。

reportData

次の例のように、**reportData** 変数の **data** フィールドで参照される変数を使用します。

```
//データ付きレコード配列  
myRecords customerRecord[];  
  
eglReportData.data = myRecords;
```

sqlStatement

次の例のように、**reportData** 変数の **sqlStatement** フィールドで識別される SQL ステートメントを使用します。

```
mySQLString = "Select * From MyTable";  
eglReportData.sqlStatement = mySQLString;
```

呼び出し例を次に示します。

```
ReportLib.fillReport (eglReport, DataSource.sqlStatement);
```

関連する概念

- 225 ページの『EGL レポートの概要』
- 226 ページの『EGL レポート作成プロセスの概要』
- 524 ページの『EGL での列挙型』
- 811 ページの『EGL 関数の構文図』

関連する参照項目

- 228 ページの『データ・ソース』
- 922 ページの『EGL ライブラリー ReportLib』
- 923 ページの『addReportParameter()』
- 924 ページの『exportReport()』
- 927 ページの『resetReportParameters()』

getFieldValue()

ReportLib.getFieldValue 関数は、現在処理中の行について、指定のフィールド値を返します。

```
ReportLib.getFieldValue(fieldName STRING in)  
returns (result ANY)
```

result

fieldName

関連する概念

- 811 ページの『EGL 関数の構文図』
- 225 ページの『EGL レポートの概要』
- 226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

- 923 ページの『addReportParameter()』
- 925 ページの『fillReport()』
- 922 ページの『EGL ライブラリー ReportLib』
- 924 ページの『exportReport()』

getReportData()

システム関数 **ReportLib.getReportData** は、指定された名前を持つレポート・データ・レコードを検索します。

```
ReportLib.getReportData(dataSetName STRING in)  
returns (result ReportData)
```

result

dataSetName

関連する概念

- 811 ページの『EGL 関数の構文図』
- 225 ページの『EGL レポートの概要』
- 226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

- 923 ページの『addReportParameter()』

922 ページの『EGL ライブラリー ReportLib』

924 ページの『exportReport()』

925 ページの『fillReport()』

getReportParameter()

ReportLib.getReportParameter 関数は、記入中のレポートから、指定のパラメータの値を戻します。

```
ReportLib.getReportParameter(parameter STRING in)  
returns (result ANY)
```

result

parameter

関連する概念

811 ページの『EGL 関数の構文図』

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

922 ページの『EGL ライブラリー ReportLib』

923 ページの『addReportParameter()』

925 ページの『fillReport()』

924 ページの『exportReport()』

getReportVariableValue()

システム関数 **ReportLib.getReportVariableValue** は、記入中のレポートから指定の変数の値を戻します。

```
ReportLib.getReportVariableValue(variable STRING in)  
returns (result ANY)
```

result

variable

関連する概念

811 ページの『EGL 関数の構文図』

225 ページの『EGL レポートの概要』

226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

923 ページの『addReportParameter()』

922 ページの『EGL ライブラリー ReportLib』

924 ページの『exportReport()』

925 ページの『fillReport()』

resetReportParameters()

ReportLib.resetReportParameters 関数の構文図は、次のとおりです。

```
ReportLib.resetReportParameters(report Report in)
```


report

除去したいパラメーターが含まれるレポートの名前。

ReportLib.resetReportParameters 関数は、特定のレポートに使用される EGL パラメーターをすべて除去します。

関連する概念

- 811 ページの『EGL 関数の構文図』
- 225 ページの『EGL レポートの概要』
- 226 ページの『EGL レポート作成プロセスの概要』

関連する参照項目

- 922 ページの『EGL ライブラリー ReportLib』
- 923 ページの『addReportParameter()』
- 924 ページの『exportReport()』
- 925 ページの『fillReport()』

setReportVariableValue()

ReportLib.setReportVariableValue 関数は、指定の変数の値を、この関数に提供された値に設定します。

```
ReportLib.setReportVariableValue(  
    variable STRING in,  
    value Any in)
```

variable

value

関連する概念

- 811 ページの『EGL 関数の構文図』
- 226 ページの『EGL レポート作成プロセスの概要』
- 225 ページの『EGL レポートの概要』

関連する参照項目

- 923 ページの『addReportParameter()』
- 922 ページの『EGL ライブラリー ReportLib』
- 925 ページの『fillReport()』
- 924 ページの『exportReport()』

EGL ライブラリー StrLib

次の表は、ライブラリー **StrLib** におけるシステム関数を示しています。それに続く表では、そのライブラリーにおける変数および定数を示しています。

システム関数およびシステム呼び出し	説明
<i>result</i> = characterAsInt (<i>text</i>)	文字ストリングを、文字式内の最初の文字に対応する整数ストリングに変換する
<i>result</i> = clip (<i>text</i>)	戻された文字ストリングの末尾から、blank・スペースと NULL を削除する

システム関数およびシステム呼び出し	説明
<i>result</i> = <i>compareStr</i> (<i>target</i> , <i>targetSubstringIndex</i> , <i>targetSubstringLength</i> , <i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceSubstringLength</i>)	実行時に 2 つのサブストリングを ASCII または EBCDIC 順に比較し、どちらが大きい かを示す値 (-1、0、または 1) を返す
<i>result</i> = <i>concatenate</i> (<i>target</i> , <i>source</i>)	<i>target</i> と <i>source</i> を連結して、新規ストリン グを <i>target</i> に配置し、 <i>target</i> が新規ストリ ングを含めるだけの長さがあったかどうかを 示す整数を返す
<i>result</i> = <i>concatenateWithSeparator</i> (<i>target</i> , <i>source</i> , <i>separator</i>)	<i>target</i> と <i>source</i> を連結し、それらの間に <i>separator</i> を挿入して、新規ストリングを <i>target</i> に配置し、 <i>target</i> が新規ストリングを 含めるだけの長さがあったかどうかを示す整 数を返す
<i>copyStr</i> (<i>target</i> , <i>targetSubstringIndex</i> , <i>targetSubstringLength</i> , <i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceSubstringLength</i>)	1 つのサブストリングを別のサブストリング にコピーする
<i>result</i> = <i>findStr</i> (<i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceSubstringLength</i> , <i>searchString</i>)	ストリング内で、あるサブストリングの最初 のオカレンスを検索する
<i>result</i> = <i>formatDate</i> (<i>dateValue</i> [, <i>dateFormat</i>])	日付値をフォーマット設定し、 STRING 型の 値を返す。デフォルトのフォーマットは、現 行ロケールで指定済みのフォーマットです。
<i>result</i> = <i>formatNumber</i> (<i>numericExpression</i> , <i>numericFormat</i>)	数値をフォーマット設定されたストリングと して返す
<i>result</i> = <i>formatTime</i> (<i>timeValue</i> [, <i>timeFormat</i>])	パラメーターを時刻値にフォーマット設定 し、 STRING 型の値を返す。デフォルトのフ ォーマットは、現行ロケールで指定済みのフ ォーマットです。
<i>result</i> = <i>formatTimeStamp</i> (<i>timeStampValue</i> [, <i>timeStampFormat</i>])	パラメーターをタイム・スタンプ値にフォー マット設定し、 STRING 型の値を返す。 DB2 形式がデフォルト・フォーマットで す。
<i>result</i> = <i>getNextToken</i> (<i>target</i> , <i>source</i> , <i>sourceSubstringIndex</i> , <i>sourceStringLength</i> , <i>characterDelimiter</i>)	ストリング内にある次のトークンを検索し て、そのトークンを <i>target</i> にコピーする
<i>result</i> = <i>integerAsChar</i> (<i>integer</i>)	整数ストリングを文字ストリングに変換する
<i>result</i> = <i>lowerCase</i> (<i>text</i>)	文字ストリング内のすべての大文字値を小文 字値に変換する。数値および既存の小文字値 は影響を受けません。
<i>setBlankTerminator</i> (<i>target</i>)	C または C++ プログラムから戻ったストリ ング値が EGL 生成プログラムで正しく作動 することができるように、ストリング内の NULL 終了文字およびそれに続くすべての文 字を、スペースに置換する
<i>setNullTerminator</i> (<i>target</i>)	ストリングの末尾のスペースをすべて NULL に変更する
<i>setSubStr</i> (<i>target</i> , <i>targetSubstringIndex</i> , <i>targetSubstringLength</i> , <i>source</i>)	サブストリング内の各文字を、指定した 1 文字に置き換える

システム関数およびシステム呼び出し	説明
<i>result</i> =spaces (<i>characterCount</i>)	指定された長さの文字列を返す
<i>result</i> = strlen (<i>source</i>)	項目内のバイト数を返す (末尾のスペースまたは NULL を除く)
<i>result</i> = textLen (<i>source</i>)	テキスト式内のバイト数を返す (末尾のスペースまたは NULL を除く)
<i>result</i> = upperCase (<i>characterItem</i>)	文字文字列内のすべての小文字値を大文字値に変換する。数値および既存の大文字値は影響を受けません。

ライブラリー **StrLib** におけるシステム変数は、以下の表のとおりです。

システム変数	説明
defaultDateFormat	関数 StrLib.formatDate によって戻される文字列を作成するために使用することが出来る、いくつかのマスクのうちの 1 つである defaultDateFormat の値を指定する。
defaultMoneyFormat	関数 StrLib.formatNumber によって戻される文字列を作成するために使用することが出来る、いくつかのマスクのうちの 1 つである defaultMoneyFormat の値を指定する。
defaultNumericFormat	関数 StrLib.formatNumber によって戻される文字列を作成するために使用することが出来る、いくつかのマスクのうちの 1 つである defaultNumericFormat の値を指定する。
defaultTimeFormat	関数 StrLib.formatTime によって戻される文字列を作成するために使用することが出来る、いくつかのマスクのうちの 1 つである defaultTimeFormat の値を指定する。
defaultTimestampFormat	関数 StrLib.formatTimestamp によって戻される文字列を作成するために使用することが出来る、いくつかのマスクのうちの 1 つである defaultTimestampFormat の値を指定する。

ライブラリー **StrLib** におけるシステム定数は、以下の表のとおりです。これらは、すべて **STRING** 型です。

システム変数	説明
db2TimestampFormat	yyyy-MM-dd-HH.mm.ss.fffff パターン (IBM DB2 のデフォルトのタイム・スタンプ・フォーマット)。
eurDateFormat	dd.MM.yyyy パターン (IBM 欧州標準規格の日付形式)

システム変数	説明
eurTimeFormat	<i>HH:mm:ss</i> パターン (IBM 欧州標準規格の時刻形式)
isoDateFormat	<i>yyyy-MM-dd</i> パターン (International Standards Organization (ISO) により指定された日付形式)。
isoTimeFormat	<i>HH:mm:ss</i> パターン (International Standards Organization (ISO)) により指定された時刻形式)
jisDateFormat	<i>yyyy-MM-dd</i> パターン (日本工業規格 (JIS) 日付形式)。
jisTimeFormat	<i>HH:mm:ss</i> パターン (日本工業規格の時刻形式)。
odbcTimestampFormat	<i>yyyy-MM-dd HH:mm:ss.ffffff</i> パターン (ODBC のタイム・スタンプ・フォーマット)。
usaDateFormat	<i>MM/dd/yyyy</i> パターン (IBM USA 標準規格の日付形式)
usaTimeFormat	<i>hh:mm AM</i> パターン (IBM USA 標準規格の時刻形式)。

関連する参照項目

939 ページの『formatDate()』
940 ページの『formatNumber()』
941 ページの『formatTime()』
942 ページの『formatTimeStamp()』

characterAsInt()

文字列・フォーマット設定関数 **StrLib.characterAsInt** は、文字列を、文字列内の最初の文字に対応する整数文字列に変換します。

```
StrLib.characterAsInt(text STRING in)
returns (result INT)
```

result

INT 型の変数。

text

CHAR 型の文字列を戻すリテラル、変数、または式。

整数文字列を文字列に変換するには、**StrLib.integerAsChar** 文字列・フォーマット設定関数を使用します。

関連する参照項目

928 ページの『EGL ライブラリー StrLib』
945 ページの『integerAsChar()』

clip()

文字列・フォーマット設定関数 **StrLib.clip** は、戻された文字列の末尾から、空白・スペースと NULL を削除します。

```
StrLib.clip(text STRING in)  
returns (result STRING)
```

result

文字ストリング。

text

CHAR 型の文字ストリングを戻すリテラル、変数、または式。

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

compareStr()

システム関数 **StrLib.compareStr** は、実行時に 2 つのサブストリングを ASCII または EBCDIC 順に比較します。

```
StrLib.compareStr(  
  target VagText in,  
  targetSubStringIndex INT in,  
  targetSubStringLength INT in,  
  source VagText in,  
  sourceSubStringIndex INT in,  
  sourceSubStringLength INT in )  
returns (result INT)
```

result

関数から戻される (INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された) 次のいずれかの値を受け取る数値項目。

- 1** *target* を基にしたサブストリングは、 *source* を基にしたサブストリングより小さい
- 0** *target* を基にしたサブストリングは、 *source* を基にしたサブストリングと等しい
- 1** *target* を基にしたサブストリングは、 *source* を基にしたサブストリングより大きい

target

ターゲット・サブストリングの派生元のストリング。項目またはリテラルを使用できます。

targetSubStringIndex

target の先頭バイトの指標値が 1 である場合に、*target* のサブストリングの開始バイトを識別する。この指標には、整数リテラルを使用できます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

targetSubStringLength

target から派生するサブストリングのバイト数を識別する。長さは整数リテラルとすることができます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

source

ソース・サブストリングの派生元のストリング。項目またはリテラルを使用できます。

sourceSubStringIndex

source の先頭バイトの指標値が 1 である場合に、*source* のサブストリングの開

始バイトを識別する。この指標には、整数リテラルを使用できます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

sourceSubStringLength

source から派生するサブストリングのバイト数を識別する。長さは整数リテラルとすることができます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

サブストリング値に関して、バイト対バイトのバイナリー比較が実行されます。サブストリングの長さが一致しない場合は、比較する前に、短い方のサブストリングにスペースが埋め込まれます。

定義に関する考慮事項: 以下の値が **sysVar.errorCode** で戻されます。

- 8** 指標が 1 より小さいか、ストリングの長さよりも大きい。
- 12** 長さが 1 より小さい。
- 20** 無効な 2 バイト指標である。DBCHAR ストリングや UNICODE ストリングの指標が、2 バイト文字の中間を指しています。
- 24** 無効な 2 バイトの長さである。DBCHAR ストリングや UNICODE ストリングの長さを示すバイト数が、奇数になっています (2 バイト文字の長さは常に偶数になります)。

例:

```
target = "123456";
source = "34";
result =
  StrLib.compareStr(target,3,2,source,1,2);
// result = 0
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

concatenate()

システム関数 **StrLib.concatenate** は、2 つのストリングを連結します。

```
StrLib.concatenate(
  target VagText inOut,
  source VagText in)
returns (result INT)
```

result

関数から戻される (INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された) 次のいずれかの値を受け取る数値項目。

- 1** 連結されたストリングが長すぎてターゲット項目と適合しないため、ストリングが切り捨てられた (後述)
- 0** 連結されたストリングがターゲット項目に適合する

target

ターゲット項目

source

ソース項目またはリテラル

2 つのストリングを連結すると、以下のようになります。

1. ターゲット・ストリングの末尾のスペースや NULL がすべて削除される。
2. ステップ 1 で生成されたストリングにソース・ストリングが付加される。
3. ステップ 2 で生成されたストリングがターゲット・ストリング項目よりも長い場合は、ストリングが切り捨てられる。ターゲット・ストリング項目よりも短い場合は、ブランクが埋め込まれる。

例:

```
phrase = "and/ "; // CHAR(7)
or      = "or";
result =
  StrLib.concatenate(phrase,or);
if (result == 0)
  print phrase; // phrase = "and/or "
end
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

concatenateWithSeparator()

システム関数 **StrLib.concatenateWithSeparator** は、ストリングとストリングの間に区切りストリングを挿入して 2 つのストリングを連結します。ターゲット・ストリングの最初の長さがゼロ (末尾のブランクや NULL は数えない) である場合、区切り文字は省略され、ソース・ストリングがターゲット・ストリングにコピーされます。

```
StrLib.concatenateWithSeparator(
  target VagText inOut,
  source VagText in,
  separator VagText in)
returns (result INT)
```

result

関数から戻される (INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された) 次のいずれかの値を受け取る数値項目。

- 0 連結されたストリングがターゲット項目に適合する。
- 1 連結されたストリングが長すぎてターゲット項目と適合しないため、ストリングが切り捨てられた (後述)

target

ターゲット項目。

source

ソース項目またはリテラル。

separator

区切り項目またはリテラル。

末尾のスペースや NULL が *target* から切り捨てられ、区切り ストリングおよび *source* が切り捨てられた値に付加されます。連結した結果がターゲットで許可される長さより長い場合は、切り捨てが実行されます。連結した結果がターゲットで許可される長さより短い場合は、連結された値にスペースが埋め込まれます。

例:

```
phrase = "and"; // CHAR(7)
or      = "or";
result =
  StrLib.concatenateWithSeparator(phrase,or,"/");
if (result == 0)
  print phrase; // phrase = "and/or "
end
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

copyStr()

システム関数 **StrLib.copyStr** は、あるサブストリングを別のサブストリングにコピーします。

```
StrLib.copyStr(
  target VagText inOut,
  targetSubstringIndex INT in,
  targetSubstringLength INT in,
  source VagText in,
  sourceSubstringIndex INT in,
  sourceSubstringLength INT in)
```

target

ターゲット・サブストリングの派生元のストリング。項目またはリテラルを使用できます。

targetSubstringIndex

target の先頭バイトの値が 1 である場合に、*target* の開始バイトを識別する。この指標には、整数リテラルを使用できます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

targetSubstringLength

target から派生するサブストリングのバイト数を識別する。長さは整数リテラルとすることができます。または、長さは、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目とすることができます。

source

ソース・サブストリングの派生元のストリング。項目またはリテラルを使用できます。

sourceSubstringIndex

source の先頭バイトの値が 1 である場合に、*source* のサブストリングの開始バイトを識別する。この指標には、整数リテラルを使用できます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

sourceSubstringLength

source から派生するサブストリングのバイト数を識別する。長さは整数リテラルとすることができます。または、長さは、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目とすることができます。

ソースがターゲットよりも長い場合は、ソースが切り捨てられます。ソースがターゲットよりも短い場合は、ソースの値の右側にスペースが埋め込まれます。

定義に関する考慮事項: 以下の値が **sysVar.errorCode** で戻されます。

- 8 指標が 1 より小さいか、ストリングの長さよりも大きい。
- 12 長さが 1 より小さい。
- 20 無効な 2 バイト指標である。DBCHAR ストリングや UNICODE ストリングの指標が、2 バイト文字の中間を指しています。
- 24 無効な 2 バイトの長さである。DBCS ストリングや UNICODE ストリングの長さを示すバイト数が、奇数になっています (2 バイト文字の長さは常に偶数になります)。

例:

```
target = "120056";
source = "34";
StrLib.copyStr(target,3,2,source,1,2);
// target = "123456"
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

defaultDateFormat

システム変数 **StrLib.defaultDateFormat** は、関数 **StrLib.formatDate** によって戻されるストリングを作成するために使用できる、いくつかのマスクのうちの 1 つである **defaultDateFormat** の値を指定します。

StrLib.defaultDateFormat の初期値は、Java ランタイム・プロパティー **vgj.default.dateFormat** の値です。このプロパティーが設定されていない場合、**StrLib.defaultDateFormat** の初期値は *MM/dd/yyyy* です。

時刻マスクの特性についての詳細は、『日付、時刻、およびタイム・スタンプ指定子 (*Date, time, and timestamp specifiers*)』を参照してください。

データ型: STRING

関連する参照項目

- 49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』
- 928 ページの『EGL ライブラリー StrLib』
- 939 ページの『formatDate()』
- 584 ページの『Java ランタイム・プロパティー (詳細)』

defaultMoneyFormat

システム変数 **StrLib.defaultMoneyFormat** は、関数 **StrLib.formatNumber** によって戻されるSTRINGを作成するために使用できるいくつかのマスクのうちの 1 つである、**defaultMoneyFormat** の値を指定します。

StrLib.defaultMoneyFormat の初期値は、Java ランタイム・プロパティー **vgj.default.moneyFormat** の値です。このプロパティーが設定されていない場合、**StrLib.defaultMoneyFormat** の初期値は空STRINGです。

数値マスクの特性についての詳細は、『*formatNumber()*』を参照してください。

データ型: STRING

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

940 ページの『*formatNumber()*』

584 ページの『Java ランタイム・プロパティー (詳細)』

defaultNumericFormat

システム変数 **StrLib.defaultNumericFormat** は、関数 **StrLib.formatNumber** によって戻されるSTRINGを作成するために使用できるいくつかのマスクのうちの 1 つである、**defaultNumericFormat** の値を指定します。

StrLib.defaultNumericFormat の初期値は、Java ランタイム・プロパティー **vgj.default.numericFormat** の値です。このプロパティーが設定されていない場合、**StrLib.defaultNumericFormat** の初期値は空STRINGです。

数値マスクの特性についての詳細は、『*formatNumber()*』を参照してください。

データ型: STRING

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

940 ページの『*formatNumber()*』

584 ページの『Java ランタイム・プロパティー (詳細)』

defaultTimeFormat

システム変数 **StrLib.defaultTimeFormat** は、関数 **StrLib.formatTime** によって戻されるSTRINGを作成するために使用できるいくつかのマスクのうちの 1 つである、**defaultTimeFormat** の値を指定します。この変数は、他のコンテキストでは使用されません。

StrLib.defaultTimeFormat の初期値は、Java ランタイム・プロパティー **vgj.default.timeFormat** の値です。このプロパティーが設定されていない場合、**StrLib.defaultTimeFormat** の初期値は *HH:mm:ss* です。

時刻マスクの特性についての詳細は、『*日付、時刻、およびタイム・スタンプ指定子 (Date, time, and timestamp specifiers)*』を参照してください。

データ型: STRING

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

928 ページの『EGL ライブラリー StrLib』

941 ページの『formatTime()』

584 ページの『Java ランタイム・プロパティー (詳細)』

defaultTimestampFormat

システム変数 **StrLib.defaultTimestampFormat** は、関数 **StrLib.formatTimestamp** によって戻されるストリングを作成するために使用できるいくつかのマスクのうちの 1 つである、**defaultTimestampFormat** の値を指定します。

StrLib.defaultTimestampFormat の初期値は、Java ランタイム・プロパティー **vgj.default.timestampFormat** の値です。このプロパティーが設定されていない場合、**StrLib.defaultTimestampFormat** の初期値は空ストリングです。

タイム・スタンプ・マスクの特性についての詳細は、『日付、時刻、およびタイム・スタンプ指定子 (*Date, time, and timestamp specifiers*)』を参照してください。

データ型: STRING

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

928 ページの『EGL ライブラリー StrLib』

942 ページの『formatTimeStamp()』

584 ページの『Java ランタイム・プロパティー (詳細)』

findStr()

システム関数 **StrLib.findStr** は、あるサブストリングがストリング内で最初に現れる位置を検索します。

```
StrLib.findStr(  
    source VagText in,  
    sourceSubstringIndex INT inOut,  
    sourceSubstringLength INT in,  
    searchString VagText in)  
returns (result INT)
```

result

関数から戻される (INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された) 次のいずれかの値を受け取る数値項目。

-1 検索ストリングが見付からない

0 検索ストリングが見付かった

source

ソース・サブストリングの派生元のストリング。項目またはリテラルを使用できます。

sourceSubstringIndex

source の先頭バイトの指標値が 1 である場合に、*source* のサブストリングの開

始バイトを識別する。この指標は、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目とすることができます。

sourceStringLength

source から派生するサブストリングのバイト数を識別する。この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

searchString

ソース・サブストリングの中で検索するストリング項目またはリテラル。検索ストリングの末尾のブランクや NULL は、検索が開始される前に切り捨てられます。

ソース・サブストリングの中で *searchString* が検出されると、そのロケーション (一致するサブストリングが始まるソース内のバイト) を示すように *sourceSubstringIndex* が設定されます。検出されない場合は、*sourceSubstringIndex* は変更されません。

定義に関する考慮事項: 以下の値が `sysVar.errorCode` で戻されます。

- 8 指標が 1 より小さいか、ストリングの長さよりも大きい。
- 12 長さが 1 より小さい。
- 20 無効な 2 バイト指標である。DBCHAR ストリングや UNICODE ストリングの指標が、2 バイト文字の中間を指しています。
- 24 無効な 2 バイトの長さである。DBCHAR ストリングや UNICODE ストリングの長さを示すバイト数が、奇数になっています (2 バイト文字の長さは常に偶数になります)。

例:

```
source = "123456";
sourceIndex = 1
sourceLength = 6
search = "34";
result =
  StrLib.findStr(source,sourceIndex,sourceLength,"34");
// result = 0, sourceIndex = 3
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

formatDate()

システム関数 **StrLib.formatDate** は、日付値をフォーマット設定し、STRING 型の値を戻します。

```
StrLib.formatDate(
  dateValue DATE in
  [, dateFormat STRING in])
returns (result STRING)
```

result

STRING 型の変数。

dateValue

フォーマット設定される値。システム変数 **VGVar.currentGregorianCalendar** のように日付値に解決される任意の式が可能です。

dateFormat

『日付、時間、およびタイム・スタンプ指定子』の説明に従って、日付形式を識別します。*dateFormat* に対して値を指定しない場合、EGL ランタイムは Java ロケールの日付形式を使用します。

ストリング、システム変数 **StrLib.dateFormat** (『*dateFormat*』を参照)、または以下の任意の定数を使用できます。

StrLib.eurDateFormat

「*dd.MM.yyyy*」パターン (IBM 欧州標準規格の日付形式)。

StrLib.isoDateFormat

「*yyyy-MM-dd*」パターン (International Standards Organization (ISO) により指定された日付形式)。

StrLib.jisDateFormat

「*yyyy-MM-dd*」パターン (日本工業規格 (JIS) 日付形式)。

StrLib.usaDateFormat

「*MM/dd/yyyy*」パターン (IBM USA 標準規格の日付形式)。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

44 ページの『DATE』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

936 ページの『*dateFormat*』

928 ページの『EGL ライブラリー **StrLib**』

formatNumber()

ストリング関数 **StrLib.formatNumber** は、数値をフォーマット設定されたストリングとして戻します。

```
StrLib.formatNumber(  
    numericExpression anyNumericExpression in,  
    numericFormat STRING in)  
returns (result STRING)
```

result

STRING 型の変数。

numericExpression

フォーマット設定される数値。数値に解決される任意の式を入れることができます。

numericFormat

数値のフォーマット設定の方法を定義するストリング。詳細については、次の表を参照してください。ストリングは必須ですが、システム変数

StrLib.defaultMoneyFormat または **StrLib.defaultNumericFormat** を使用できます。これらの変数の詳細については、『*defaultMoneyFormat*』および『*defaultNumericFormat*』を参照してください。

有効な文字は以下のとおりです。

- # 数字のプレースホルダー。
- * 先行ゼロの充てん文字としてアスタリスク (*) を使用します。
- & 先行ゼロの充てん文字としてゼロを使用します。
- # 先行ゼロの充てん文字としてスペースを使用します。
- < 数値を左そろえにします。
- , その位置に先行ゼロが含まれていない場合、ロケールに依存する数字分離記号を使用します。
- . ロケールに依存する小数点を使用します。
- 0 未満の値に対しては負符号 (-) を使用し、0 以上の値に対してはスペースを使用します。
- + 0 未満の値に対しては負符号 (-) を使用し、0 以上の値に対しては正符号 (+) を使用します。
- (会計処理などで、必要に応じて、左括弧の前に負の値を配置します。
-) 会計処理などで、必要に応じて、負の値の後に右括弧を配置します。
- \$ ロケールに依存する通貨記号の前に値を配置します。
- @ 値の後にロケールに依存する通貨記号を配置します。

関連する参照項目

- 937 ページの『defaultMoneyFormat』
- 937 ページの『defaultNumericFormat』
- 928 ページの『EGL ライブラリー StrLib』

formatTime()

日時関数 **StrLib.formatTime** は、時刻値をフォーマット設定し、STRING 型の値を返します。

```
StrLib.formatTime(  
    aTime Time in  
    [, timeFormat STRING in  
    ]  
)  
returns (result STRING)
```

result

STRING 型の変数。

aTime

フォーマット設定される値。システム変数 **DateTimeLib.currentTime** のように時刻値に解決される任意の式が可能です。

timeFormat

『日付、時間、およびタイム・スタンプ指定子』の説明に従って、時刻形式を識別します。*timeFormat* の値を指定しない場合、EGL ランタイムは Java ロケールの時刻形式を使用します。

ストリング、システム変数 **StrLib.defaultTimeFormat** (『defaultTimeFormat』を参照)、または以下の任意の定数を使用できます。

eurTimeFormat

「*HH:mm:ss*」パターン (IBM 欧州標準規格の時刻形式)

isoTimeFormat

「*HH:mm:ss*」パターン (International Standards Organization (ISO) により指定された時刻形式)

jisTimeFormat

「*HH:mm:ss*」パターン (日本工業規格の時刻形式)

usaTimeFormat

「*hh:mm AM*」パターン (IBM USA 標準規格の時刻形式)。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

937 ページの『defaultTimeFormat』

928 ページの『EGL ライブラリー StrLib』

47 ページの『TIME』

formatTimeStamp()

日時フォーマット設定関数 **StrLib.formatTimeStamp** は、パラメーターをタイム・スタンプ値にフォーマット設定し、STRING 型の値を戻します。

```
StrLib.formatTimeStamp(  
    aTimeStamp TimeStamp in  
    [, timeStampFormat STRING in  
    ]  
)  
returns (result STRING)
```

result

STRING 型の変数。

aTimeStamp

フォーマット設定される **TIMESTAMP** 値。システム変数

DateTimeLib.currentTimeStamp のように **TIMESTAMP** 値に解決される任意の式が可能です。

timeStampFormat

『日付、時間、およびタイム・スタンプ指定子』の説明に従って、日付形式を識別します。

ストリング、システム変数 **StrLib.defaultTimestampFormat**

(『defaultTimestampFormat』を参照) または以下の定数のいずれかを使用できます。

db2TimeStampFormat

パターン「*yyyy-MM-dd-HH:mm:ss.ffffff*」パターン (IBM DB2 のデフォルトのタイム・スタンプ・フォーマット)。

odbcTimeStampFormat

「*yyyy-MM-dd HH:mm:ss.ffffff*」パターン (ODBC のタイム・スタンプ・フォーマット)

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

852 ページの『currentTimeStamp()』

49 ページの『日付、時刻、およびタイム・スタンプのフォーマット指定子』

938 ページの『defaultTimestampFormat』

928 ページの『EGL ライブラリー StrLib』

47 ページの『TIMESTAMP』

getNextToken()

システム関数 **StrLib.getNextToken** は、あるトークンをサブストリングの中で検索し、そのトークンをターゲット項目にコピーします。

トークンとは、区切り文字で区切られたストリングのことです。例えば、スペース (「 」) とコンマ (「,」) の文字が区切り文字として定義されている場合、ストリング「CALL PROGRAM ARG1,ARG2,ARG3」は、「CALL」、「PROGRAM」、「ARG1」、「ARG2」、および「ARG3」という 5 つのトークンに分解できます。

```
StrLib.getNextToken(  
    target VagText inOut,  
    source VagText in,  
    sourceSubstringIndex INT inOut,  
    sourceSubstringLength INT inOut,  
    characterDelimiter VagText in)  
returns (result INT)
```

result

INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された項目。値は以下のいずれかです。

- +n** トークン内の文字数。トークンは、調査されているサブストリングからターゲット項目にコピーされます。
- 0** 調査されているサブストリングの中にトークンがなかった。
- 1** ターゲット項目へのコピー時にトークンが切り捨てられた。

target

CHAR 型、DBCHAR 型、HEX 型、MBCHAR 型、または UNICODE 型のターゲット項目。

source

CHAR 型、DBCHAR 型、HEX 型、MBCHAR 型、または UNICODE 型のソース項目。UNICODE 以外の型のリテラルである場合もあります。

sourceSubstringIndex

source の先頭バイトの値が 1 である場合に、区切り文字の検索を開始する開始バイトを識別する。 *sourceSubstringIndex* には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。トークンが検出されると、*sourceSubstringIndex* の値は、トークンの後ろにある最初の文字の指標に変更されます。

sourceSubstringLength

調査されているサブストリング内のバイト数を示す。 *sourceSubstringLength* には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を

使用できます。トークンが検出されると、*sourceSubstringLength* の値は、戻されたトークンの後ろで始まるサブストリング内のバイト数に変更されます。

characterDelimiter

1 つ以上の区切り文字 (区切り文字と区切り文字を区切る文字はない)。 CHAR 型、DBCHAR 型、HEX 型、MBCHAR 型、または UNICODE 型の項目とすることができます。UNICODE 以外の型のリテラルである場合もあります。

後述の例に示されているとおり、*sourceSubstringIndex* および *sourceSubstringLength* の値をリセットすることなく、サブストリング内の各トークンを検索する一連の呼び出しが可能です。

エラー条件: 以下の値が **SysVar.errorCode** で戻されます。

- 8 *sourceSubstringIndex* が 1 より小さいか、調査されているサブストリング内のバイト数より大きい。
- 12 *sourceSubstringLength* が 1 より小さい。
- 20 DBCHAR または UNICODE ストリングの *sourceSubstringIndex* の値が、2 バイト文字の 2 番目のバイトを参照する。
- 24 DBCHAR ストリングまたは UNICODE ストリングの *sourceSubstringLength* の値が、奇数になっています (2 バイト文字の長さは常に偶数になります)。

例:

```
Function myFunction()
  myVar myStructurePart;
  myRecord myRecordPart;

  i = 1;
  myVar.mySourceSubstringIndex = 1;
  myVar.mySourceSubstringLength = 29;

  while (myVar.mySourceSubstringLength > 0)
    myVar.myResult = StrLib.getNextToken( myVar.myTarget[i],
      "CALL PROGRAM arg1, arg2, arg3",
      myVar.mySourceSubstringIndex,
      myVar.mySourceSubstringLength, " ," );

    if (myVar.myResult > 0)
      myRecord.outToken = myVar.myTarget[i];
      add myRecord;
      set myRecord empty;
      i = i + 1;
    end
  end
end

Record myStructurePart
  01 myTarget CHAR(80)[5];
  01 mySource CHAR(80);
  01 myResult myBinPart;
  01 mySourceSubstringIndex INT;
  01 mySourceSubstringLength BIN(9,0);
  01 i myBinPart;
end

Record myRecordPart
  serialRecord:
```

```

        fileName="Output"
    end
    01 outToken CHAR(80);
end

```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

integerAsChar()

文字列・フォーマット設定関数 **StrLib.integerAsChar** は、整数文字列を文字列に変換します。

```

StrLib.integerAsChar(integer INT in)
returns (result STRING)

```

result

STRING 型の変数。

integer

BIGINT、INT、SMALLINT のいずれかの型の整数を戻すリテラル、変数、または式。

文字列を整数文字列に変換するには、**StrLib.characterAsInt** 文字列・フォーマット設定関数を使用します。

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

931 ページの『characterAsInt()』

lowerCase()

文字列・フォーマット設定関数 **StrLib.lowerCase** は、文字列内のすべての大文字値を小文字値に変換します。数値および既存の小文字値は影響を受けません。

```

StrLib.lowerCase(text STRING in)
returns (result STRING)

```

result

STRING 型の変数。

text

CHAR 型の文字列を戻すリテラル、変数、または式。

StrLib.lowerCase 関数は、DBCHAR 型または MBCHAR 型の項目内の 2 バイト文字には影響を及ぼしません。

小文字値を大文字値に変換するには、**StrLib.upperCase** 文字列・フォーマット設定関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

949 ページの『upperCase()』

setBlankTerminator()

システム関数 **StrLib.setBlankTerminator** は、NULL 終了文字およびそれに続くすべての文字をスペースに変更します。**StrLib.setBlankTerminator** は、C または C++ プログラムから戻されたストリング値を、EGL プログラムで適正に操作できる文字値に変更します。

```
StrLib.setBlankTerminator(target VagText inOut)
```

target

ターゲット・ストリング項目。*targetString* で NULL が検出されない場合は、この関数は影響を与えません。

例:

```
StrLib.setBlankTerminator(target);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

setNullTerminator()

システム関数 **StrLib.setNullTerminator** は、ストリングの末尾のスペースをすべて NULL に変更します。**StrLib.setNullTerminator** を使用すると、引数として NULL 終了ストリングを予想する C または C++ プログラムに項目を渡す前に、その項目を変換することができます。

```
StrLib.setNullTerminator(target VagText inOut)
```

target

変換するストリング

ターゲット・ストリングの中で末尾のスペースや NULL が検索されます。スペースが見付かった場合は NULL に変換されます。

定義に関する考慮事項: 以下の値が **sysVar.errorCode** で戻されます。

16 ストリングの最後のバイトがスペースや NULL でない。

例:

```
StrLib.setNullTerminator(myItem01);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

setSubStr()

システム関数 **StrLib.setSubStr** は、サブストリング内の各文字を指定された文字に置き換えます。

```
StrLib.setSubStr(  
    target VagText inOut,  
    targetSubstringIndex INT in,  
    targetSubstringLength INT in,  
    source)
```

target

変更される項目。

targetSubstringIndex

target の先頭バイトの指標値が 1 である場合に、*target* のサブストリングの開始バイトを識別する。この指標には、整数リテラルを使用できます。または、この指標には、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目を使用できます。

targetSubstringLength

target から派生するサブストリングのバイト数を識別する。長さは整数リテラルとすることができます。または、長さは、INT 型または長さが 9 で小数部がない BIN 型と同等と定義された項目とすることができます。

source

ターゲット項目が CHAR、MBCHAR、または HEX である場合、ソース項目は 1 バイトの CHAR、MBCHAR、または HEX 項目、あるいは CHAR リテラルでなければなりません。ターゲットが DBCHAR 項目または UNICODE 項目である場合、ソースは、1 文字の DBCHAR 項目または UNICODE 項目でなければなりません。

定義に関する考慮事項: 以下の値が SysVar.errorCode で戻されます。

- 8 指標が 1 より小さいか、ストリングの長さよりも大きい
- 12 長さが 1 より小さい
- 20 無効な 2 バイト指標である。DBCHAR ストリングや UNICODE ストリングの指標が、2 バイト文字の中間を指しています。
- 24 無効な 2 バイトの長さである。DBCHAR ストリングや UNICODE ストリングの長さを示すバイト数が、奇数になっています (2 バイト文字の長さは常に偶数になります)。

例:

```
StrLib.setSubStr(target,12,5," ");
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

spaces()

システム関数 **StrLib.spaces** は、指定された数のスペースで構成されたストリングを返します。

```
StrLib.spaces(characterCount INT in)  
returns (result STRING)
```

result

STRING 型の変数。

characterCount

戻されるスペースのストリングの長さ。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

strLen()

システム関数 **StrLib.strLen** は、末尾のスペースまたは NULL を除外した、項目内のバイト数を返します。

```
StrLib.strLen(source VagText in)  
returns (result INT)
```

result

INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された項目。

source

長さを測るストリング項目またはリテラル。

例:

```
length = StrLib.strLen(source);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

textLen()

システム関数 **StrLib.textLen** は、末尾のスペースまたは NULL を除外した、テキスト式内の文字数を返します。

```
StrLib.textLen(source STRING in)  
returns (result INT)
```

result

INT 型か、長さが 9 で小数部がない BIN 型と同等と定義された項目。

source

該当するテキスト式。

例:

```
length = StrLib.textLen(source);
```


関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

547 ページの『テキスト式』

upperCase()

文字列・フォーマット設定関数 **StrLib.upperCase** は、文字列内のすべての小文字値を大文字値に変換します。数値および既存の大文字値は影響を受けません。

```
StrLib.upperCase(text STRING in)
returns (result STRING)
```

result

STRING 型の変数。

text

CHAR 型の文字列を戻すリテラル、変数、または式。

StrLib.upperCase 関数は、DBCHAR 型または MBCHAR 型の項目内の 2 バイト文字には影響を及ぼしません。

文字列を小文字に変換するには、**StrLib.lowerCase** 文字列・フォーマット設定関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

928 ページの『EGL ライブラリー StrLib』

945 ページの『lowerCase()』

EGL ライブラリー SysLib

関数	説明
beginDatabaseTransaction([database])	EGL ランタイムが変更を自動的にコミットしていないときに限り、リレーショナル・データベースのトランザクションを開始する。
result = bytes(field)	名前付きメモリ領域内のバイト数を戻す
calculateChkDigitMod10 (text, checkLength, result)	modulus-10 チェック・ディジットを、連続した整数で始まる文字項目内に配置する
calculateChkDigitMod11 (text, checkLength, result)	modulus-11 チェック・ディジットを、連続した整数で始まる文字項目内に配置する
callCmd (commandString[, modeString])	システム・コマンドを実行し、コマンドが終了するまで待つ

関数	説明
<code>commit()</code>	最後のコミット以降にデータベース、MQSeries メッセージ・キュー、および CICS リカバリー可能ファイルに行われた更新内容を保管する。生成された Java プログラムまたはラッパーは、リモートの CICS ベース COBOL プログラムによって加えられた更新も保管します (CICS リカバリー可能ファイルへの更新も含まれます)。ただし、リモート COBOL プログラムの呼び出しに、『 <i>callLink</i> エレメントの <i>luwControl</i> 』で説明されているようなクライアント制御の作業単位が含まれる場合に限りです。
<code>result = conditionAsInt (booleanExpression)</code>	論理式 (<i>myVar == 6</i> のような) を受け入れ、式が真である場合には "1" を、偽である場合には "0" を返す。
<code>connect (database, userID, password[, commitScope[, disconnectOption[, isolationLevel[, commitControl]]]])</code>	すべてのカーソルを閉じ、ロックを解放し、既存の接続を終了して、データベースに接続する
<code>convert (target, direction, conversionTable)</code>	EBCDIC フォーマット (ホスト) と ASCII フォーマット (ワークステーション) の間でデータを変換するか、単一フォーマット内でコード・ページを変換する
<code>defineDatabaseAlias (alias, database)</code>	ユーザーのコードが既に接続されているデータベースとの新規接続を確立するために使用可能な別名を作成する。
<code>disconnect ([database])</code>	指定したデータベースから、または (データベースが指定されていない場合は) 現行データベースから切断する
<code>disconnectAll ()</code>	現在接続されているすべてのデータベースから切断する
<code>errorLog ()</code>	システム関数 SysLib.startLog によって開始されたエラー・ログにテキストをコピーする。
<code>result = getCmdLineArg (index)</code>	EGL プログラムの呼び出しに使用された引数リストから、指定された引数を返す。指定された引数は、ストリング値として戻されます。
<code>result = getCmdLineArgCount ()</code>	メイン EGL プログラムを開始するために使用された引数の数を返す。
<code>result = getMessage (key [, insertArray])</code>	Java ランタイム・プロパティー <code>vgj.message.file</code> 内で参照されるファイルからのメッセージを返す
<code>result = getProperty(propertyName)</code>	Java ランタイム・プロパティーの値を検索する。指定されたプロパティーが検出されない場合、この関数により <code>NULL</code> ストリング ("") が戻されます。

関数	説明
<code>loadTable (filename, insertintoClause[, delimiter])</code>	ファイルからリレーショナル・データベースヘデータをロードする。
<code>result = maximumSize (arrayName)</code>	データ項目またはレコードの動的配列に含めることができる最大行数を戻す。具体的には、この関数は、配列プロパティ maxSize の値を戻します。
<code>queryCurrentDatabase (product, release)</code>	現在接続されているデータベースの製品およびリリース番号を戻す
<code>rollback ()</code>	最後のコミット以来、データベースおよび MQSeries メッセージ・キューに行われた更新内容を取り消す。取り消しは、すべての EGL 生成アプリケーションで生じます。
<code>setCurrentDatabase (database)</code>	指定したデータベースを現行のアクティブ・データベースにする
<code>setError (itemInError, msgKey[, itemInsert])</code> <code>setError (this, msgKey[, itemInsert])</code> <code>setError (msgText)</code>	メッセージを、ページ・ハンドラーまたは UI レコード内の項目と関連付けるか、またはページ・ハンドラーまたは UI レコード自体に関連付ける。メッセージは、JSP の JSF メッセージまたはメッセージ・タグのロケーションに配置され、関連する Web ページが表示されると表示されます。
<code>setLocale (languageCode, countryCode[, variant])</code>	ページ・ハンドラー、および Web アプリケーションで実行されるプログラムで使用される
<code>setRemoteUser (userID, passWord)</code>	Java プログラムからリモート・プログラムへの呼び出しに使用するユーザー ID およびパスワードを設定する
<code>result = size (arrayName)</code>	指定されたデータ・テーブルの行数、または指定された配列のエレメント数を戻す。この配列は、構造体項目配列、データ項目またはレコードの静的配列、またはデータ項目またはレコードの動的配列です。
<code>startCmd (commandString[, modeString])</code>	システム・コマンドを実行し、コマンドが終了するまで待たない
<code>startLog (logFile)</code>	エラー・ログを開く。プログラムが SysLib.errorLog を呼び出すたびに、そのログにテキストが書き込まれます。
<code>startTransaction (termID[, prID[, termID]])</code>	メインプログラムを非同期に呼び出し、そのプログラムをプリンターまたは端末と関連付けて、レコードの受け渡しを行う。受信プログラムが EGL によって生成されている場合には、そのレコードは入力レコードの初期化で使います。受信側が VisualAge Generator によって生成されている場合には、そのレコードは作業用ストレージの初期化で使います。

関数	説明
unloadTable (<i>filename</i> , <i>selectStatement</i> [, <i>delimiter</i>])	リレーショナル・データベースからファイルにデータをアンロードする。
verifyChkDigitMod10 (<i>input</i> , <i>checkLength</i> , <i>result</i>)	連続した整数で始まる文字項目内の、modulus-10 チェック・ディジットを検査する
verifyChkDigitMod11 (<i>input</i> , <i>checkLength</i> , <i>result</i>)	連続した整数で始まる文字項目内の、modulus-11 チェック・ディジットを検査する
wait (<i>timeInSeconds</i> 0)	指定した秒数の間、実行を中断する

beginTransaction()

システム関数 **SysLib.beginTransaction** は、リレーショナル・データベースのトランザクションを開始します。ただし、EGL ランタイムが変更を自動的にコミットしていないときに限ります。変更が自動的にコミットされている場合、この関数は無効です。

```
SysLib.beginTransaction(  
    [database STRING in])
```

database

SysLib.connect または VGLib.connectionService で指定されたデータベース名。文字型のリテラルまたは変数を使用します。

接続を指定しなかった場合、この関数は現行接続に影響を及ぼします。

SysLib.beginTransaction を呼び出すと、指定の接続を使用する次回の入出力操作で、トランザクションが開始されます。またこのトランザクションは、コミットまたはロールバックが発生すると、終了します。詳細については、『作業論理単位』を参照してください。コミットまたはロールバック後に、EGL ランタイムにより変更のコミットが自動的に再開されます。

自動コミットの詳細については、『SysLib.connect』および『sqlCommitControl』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

334 ページの『作業論理単位』

247 ページの『SQL サポート』

関連する参照項目

435 ページの『sqlCommitControl』

957 ページの『connect()』

980 ページの『connectionService()』

bytes()

システム関数 **SysLib.bytes** は、名前付きメモリー領域内のバイト数を戻します。

```
SysLib.bytes(field fixedFieldOrArray in)  
returns (result INT)
```

result

field のバイト数が戻される数値項目。次の 2 つの状況に注意する必要があります。

- *field* が配列の場合、この関数は 1 つのエレメントのバイト数を返します。
- *field* が SQL レコードの場合、この関数は追加のバイトを含むレコードのバイト数を返します。詳しくは、『SQL レコード内部』を参照してください。

field

配列、項目、またはレコード

Example():

```
result = SysLib.bytes(myItem);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

37 ページの『プリミティブ型』

803 ページの『SQL レコードの内部レイアウト』

calculateChkDigitMod10()

システム関数 **SysLib.calculateChkDigitMod10** は、一連の整数で始まる文字項目にモジュラス 10 チェック・ディジットを配置します。

```
SysLib.calculateChkDigitMod10(
    text anyChar inOut,
    checkLength SMALLINT in,
    result SMALLINT inOut)
```

text

一連の整数で始まる文字項目。項目には、他の整数のすぐ右にチェック・ディジット用の追加の位置が含まれています。

checkLength

text 項目から使用する文字の数を含む項目。チェック・ディジットに使用される桁も含みます。この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

result

次の 2 つの値のいずれかを受け取る項目です。

- チェック・ディジットが作成された場合は 0
- チェック・ディジットが作成されなかった場合は 1

この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

SysLib.calculateChkDigitMod10 は関数呼び出し文で使用できます。

例: 次の例では、myInput は CHAR 型の項目であり、値 1734289 を含みます。myLength は SMALLINT 型の項目であり、値 7 を含みます。myResult は SMALLINT 型の項目です。

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

モジュラス 10 チェック・ディジットはアルゴリズムを使用して派生され、すべての場合に、チェック・ディジットの位置に格納されている数値は考慮されません。アルゴリズムについて、例の値を使用して説明します。

1. 入力数の最小桁に 2 を乗算し、右から左の順で 1 桁おきに 2 を乗算します。

$$\begin{aligned}8 \times 2 &= 16 \\4 \times 2 &= 8 \\7 \times 2 &= 14\end{aligned}$$

2. 2 を乗算していない入力数字 (132) に、生成された数字 (16814) を加算します。

$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$

3. チェック・ディジットを算出するには、0 で終わる次に大きい数値から合計を減算します。

$$30 - 26 = 4$$

減算の結果が 10 ならば、チェック・ディジットは 0 です。

この例では、myInput の元の文字は次のようになります。

1734284

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

calculateChkDigitMod11()

システム関数 **SysLib.calculateChkDigitMod11** は、一連の整数で始まる文字項目にモジュラス 11 チェック・ディジットを配置します。

```
SysLib.calculateChkDigitMod11(  
    text anyChar inOut,  
    checkLength SMALLINT in,  
    result SMALLINT inOut)
```

text

一連の整数で始まる文字項目。項目には、他の整数のすぐ右にチェック・ディジット用の追加の位置が含まれています。

checkLength

text 項目から使用する文字の数を含む項目。チェック・ディジットに使用される桁も含みます。この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

result

次の 2 つの値のいずれかを受け取る項目です。

- チェック・ディジットが作成された場合は 0
- チェック・ディジットが作成されなかった場合は 1

この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

SysLib.calculateChkDigitMod11 は関数呼び出しステートメントで使用できます。

例: 次の例では、myInput は CHAR 型の項目であり、値 56621869 を含みます。myLength は SMALLINT 型の項目であり、値 8 を含みます。myResult は SMALLINT 型の項目です。

```
SysLib.verifyChkDigitMod (myInput, myLength, myResult);
```

モジュラス 11 チェック・ディジットはアルゴリズムを使用して派生され、すべての場合に、チェック・ディジットの位置に格納されている数値は考慮されません。アルゴリズムについて、例の値を使用して説明します。

1. 入力数値の単位の位にある数字に 2 を乗算し、10 の位に 3 を、100 の位に 4 を乗算する、というように繰り返します。ただし、乗数として使用する数字は `myLength " 1` を最大数とします。入力数値にそれ以上の桁がある場合は、2 を乗数としてこの手順をもう一度最初から実行します。

```
6 x 2 = 12
8 x 3 = 24
1 x 4 = 4
2 x 5 = 10
6 x 6 = 36
6 x 7 = 42
5 x 2 = 10
```

2. 最初のステップの積を加算し、合計を 11 で除算します。

```
(12 + 24 + 4 + 10 + 36 + 42 + 10) / 11
= 138 / 11
= 12 余り 6
```

3. チェック・ディジットを算出するには、11 から余りを減算して自己検査ディジットを得ます。

```
11 - 6 = 5
```

余りが 0 または 1 である場合、チェック・ディジットは 0 です。

この例では、`myInput` の元の文字は次のようになります。

```
56621865
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

callCmd()

システム関数 **SysLib.callCmd** は、システム・コマンドを実行し、コマンドが終了するまで待ちます。

```
SysLib.callCmd(
    commandString STRING in
    [, modeString STRING in
    ])
```

commandString

呼び出すオペレーティング・システム・コマンドを識別します。

modeString

modeString は、任意の文字項目またはストリング項目にすることができます。この項目は、次の 2 つのモードのどちらでもかまいません。

- *form*: 入力の各文字が、入力されるたびにプログラムから使用可能になります。つまり、すべてのキー・ストロークが直接、指定したコマンドへ渡されます。

- *line*: 入力、改行文字のキーが使用されるまで使用可能になりません。つまり、ENTER キーが押されるまで、指定したコマンドへ情報が渡されず、押された後、入力された行全体がコマンドへ送られます。

実行されるシステム・コマンドは、実行中のプログラムから可視であることが必要です。例えば、**callCmd**("mySpecialProgram.exe") を実行する場合、プログラム "mySpecialProgram.exe" は環境変数 PATH によってポイントされたディレクトリーに入っている必要があります。完全なディレクトリー・ロケーションを指定して、例えば、**callCmd**("program files/myWork/mySpecialProgram.exe") とすることもできます。

SysLib.callCmd 関数は、Java 環境でのみサポートされます。

コマンドが終了するまで待たないシステム・コマンドを実行するには、**SysLib.startCmd** 関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

974 ページの『startCmd()』

commit()

システム関数 **SysLib.commit** は、最終コミット以降にデータベース、および MQSeries メッセージ・キューに対して加えられた更新を保管します。生成された Java プログラムまたはラッパーは、リモートの CICS ベース COBOL プログラムによって行われた更新内容も保管します (CICS リカバリー可能ファイルへの更新も含みます)。ただし、リモート COBOL プログラムの呼び出しに、『*callLink* エレメントの *luwControl*』で説明されているようなクライアント制御の作業単位が含まれる場合に限りです。

SysLib.commit()

たいていの場合、EGL は、リカバリー可能なマネージャーに順番に適用されるシングル・フェーズ・コミットを実行します。ただし、CICS for z/OS では、

SysLib.commit によって、すべてのリソース・マネージャーに適用される 2 フェーズ・コミットを実行する CICS SYNCPOINT が実行されます。

SysLib.commit により、ファイルまたはデータベースのスキャン位置と更新ロックが解放されます。

MQ レコードとともに **SysLib.commit** を使用する場合は、以下の点に注意してください。

- メッセージ・キューの更新は、MQ レコードのパーツで *Include message in transaction* オプションが選択されている場合にのみリカバリー可能です。
- メッセージ **get** および **add** は、いずれも、リカバリー可能メッセージに関する **commit** および **rollback** の影響を受けます。リカバリー可能メッセージに対して、**get** の後に **rollback** が出された場合は、メッセージが入力キューに戻されるため、トランザクションが正常に完了できなくても入力メッセージは失われませ

ん。また、**rollback** がリカバリー可能メッセージ用の **add** に続いて実行される場合、メッセージはキューから削除されます。

パフォーマンスを向上させるには、不必要に **SysLib.commit** を使用しないようにしてください。暗黙的なコミットがいつ実行されるかについて詳しくは『作業論理単位』を参照してください。

例:

```
sysLib.commit();
```

関連する概念

811 ページの『EGL 関数の構文図』
334 ページの『作業論理単位』
285 ページの『MQSeries のサポート』
798 ページの『実行単位』
247 ページの『SQL サポート』

関連する参照項目

986 ページの『commitOnConverse』
990 ページの『segmentedMode』
949 ページの『EGL ライブラリー SysLib』
450 ページの『callLink エレメントの luwControl』
664 ページの『open』
679 ページの『prepare』

conditionAsInt()

システム関数 **SysLib.conditionAsInt** は、論理式 (*myVar* == 6 など) を受け入れ、式が真である場合には "1" を、偽である場合には "0" を戻します。

```
SysLib.conditionAsInt(logicalExpression AnyLogicalExpression in)  
returns (result SMALLINT)
```

result

タイプ **SMALLINT** の値。

logicalExpression

『論理式』で説明されている論理式。

例:

```
myField = -5;  
  
// result = 0  result = SysLib.conditionAsInt(myField == 6);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
538 ページの『論理式』

connect()

システム関数 **SysLib.connect** を使用すると、プログラムを実行時にデータベースに接続することができます。この関数は、値を戻しません。

```

SysLib.connect(
    database STRING in,
    userID STRING in,
    password STRING in
    [, commitScope enumerationCommitScope in
    [, disconnectOption enumerationDisconnectOption in
    [, isolationLevel enumerationIsolationLevel in
    [, commitControl enumerationCommitControlOption in
    ]]] )

```

database

データベースを識別します。

- *database* を RESET に設定すると、デフォルト・データベースに再接続します。ただし、デフォルト・データベースが使用できない場合には、接続状況は変更されません。詳しくは、『デフォルト・データベース』を参照してください。
- それ以外の場合、物理データベース名を確認するには、プロパティ **vgj.jdbc.database.server** (*server* は **SysLib.connect** 呼び出しで指定したデータベースの名前) を調べます。このプロパティが定義されていない場合は、**SysLib.connect** 呼び出しで指定したデータベース名がそのまま使用されます。
- J2EE 接続と非 J2EE 接続ではデータベース名の形式が異なります。
 - J2EE 環境で実行されるプログラムを生成した場合には、JNDI レジストリでのデータ・ソースのバインド先となる名前 (jdbc/MyDB など) を使用します。ビルド記述子オプション **J2EE** が YES に設定されている場合に、この状態が発生します。
 - 非 J2EE JDBC 環境のためのプログラムを生成した場合 には、接続 URL (jdbc:db2:MyDB など) を使用します。オプション **J2EE** が NO に設定されている場合に、この状態が発生します。

userID

データベースへアクセスするときに使用されるユーザー ID。CHAR 型で長さが 8 の項目を引数として使用する必要があります。リテラルは有効です。引数を指定する必要があります。背景情報については、『データベースの権限とテーブル名』を参照してください。

password

データベースへアクセスするときに使用されるパスワード。CHAR 型で長さが 8 の項目を引数として使用する必要があります。リテラルは有効です。引数を指定する必要があります。

commitScope

値は以下のワードのいずれかであり、引用符および変数は使用できません。

type1 (デフォルト)

シングル・フェーズ・コミットのみがサポートされます。新規接続によって、すべてのカーソルが閉じられ、ロックが解放され、既存の接続が終了します。ただし、type1 接続を行う前に、**SysLib.commit** または **SysLib.rollback** を呼び出します。

type1 を *commitScope* の値として使用する場合、パラメーター *disconnectOption* の値は、デフォルトであるワード *explicit* である必要があります。

type2

データベースへ接続しても、カーソルの終了、ロックの解放、既存の接続の終了は行われません。複数のデータベースから読み込むために複数の接続を使用できますが、1つの作業単位に1つのデータベースのみを更新する必要があります。これは、シングル・フェーズ・コミットしか使用できないためです。

twophase

type2 と同じ。

disconnectOption

このパラメーターは、Java 出力を生成する場合にのみ有効です。値は以下のワードのいずれかであり、引用符および変数は使用できません。

explicit (デフォルト)

プログラムが **SysLib.commit** または **SysLib.rollback** を呼び出した後、接続はアクティブのまま残ります。接続リソースを解放するには、プログラムで **SysLib.disconnect** を実行する必要があります。

type1 を *commitScope* の値として使用する場合、パラメーター *disconnectOption* の値に、ワード *explicit* を設定 (またはデフォルトに設定) する必要があります。

automatic

コミットまたはロールバックによって、既存の接続が終了します。

conditional

コミットまたはロールバックによって既存の接続が自動的に終了します (カーソルが開いていて保留オプションがそのカーソルに対して有効である場合を除く)。保留オプションに関する詳細については、『オープン』を参照してください。

isolationLevel

このパラメーターは、別のデータベース・トランザクションからのデータベース・トランザクションの独立性レベルを示します。

以下のワードは厳密性の少ない順であり、ここでも、引用符および変数は使用できません。

- **readUncommitted**
- **readCommitted**
- **repeatableRead**
- **serializableTransaction** (デフォルト値)

詳しくは、Sun Microsystems, Inc の JDBC 資料を参照してください。

commitControl

このパラメーターは、データベースに変更を加えるたびに、コミットを行うかどうかを指定します。

有効な値は以下のとおりです。

- **noAutoCommit** (デフォルト) は、コミットを自動実行しないことを意味します。通常は、その方が実行速度が速くなります。その場合のコミットとロールバックの詳しい規則については、『作業論理単位』を参照してください。

- **autoCommit** は、更新が即時に有効になることを意味します。

一時的に **autoCommit** から **noAutoCommit** に切り替えることもできます。詳細については、『*SysLib.beginDatabaseTransaction*』を参照してください。

定義に関する考慮事項: **SysLib.connect** は、以下のシステム変数を設定します。

- **VGVar.sqlerrd**[3]
- **SysVar.sqlca**
- **SysVar.sqlcode**
- **VGVar.sqlWarn**[2]

例:

```
SysLib.connect(myDatabase, myUserid, myPassword);
```

関連する概念

334 ページの『作業論理単位』

798 ページの『実行単位』

247 ページの『SQL サポート』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

394 ページの『J2EE JDBC 接続のセットアップ』

283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

506 ページの『データベースの権限とテーブル名』

270 ページの『デフォルト・データベース』

949 ページの『EGL ライブラリー SysLib』

584 ページの『Java ランタイム・プロパティ (詳細)』

664 ページの『open』

435 ページの『sqlDB』

952 ページの『beginDatabaseTransaction()』

963 ページの『disconnect()』

1000 ページの『sqlca』

1001 ページの『sqlcode』

1015 ページの『sqlerrd』

1016 ページの『sqlerrmc』

1017 ページの『sqlWarn』

convert()

システム関数 **SysLib.convert** は、EBCDIC 形式 (ホスト) と ASCII 形式 (ワークステーション) の間でデータを変換するか、単一形式内でコード・ページを変換します。**SysLib.convert** は、関数呼び出し文の中で関数名として使用することができます

```
SysLib.convert(  
    target anyFixedItemOrRecordOrFormVariable inout,  
    direction enumerationConversionDirection in,  
    conversionTable CHAR(8) in)
```

target

変換対象の形式を持つレコード、データ項目、または書式の名前。データは、ターゲット・オブジェクト内の最低レベルの項目（副構造のない項目）に関する項目定義に基づいて変換されます。

可変長レコードは、現行レコードの長さの部分のみが変換されます。現行レコードの長さは、レコードの `numElementsItem` を使用して計算されるか、またはレコードの `lengthItem` から設定されます。可変長レコードが数値フィールドまたは DBCHAR 文字の途中で終わっている場合には、変換エラーが発生してプログラムは終了します。

direction

変換の方向。有効な値は "R" および "L" のみです（引用符を含む）。

conversionTable が指定されている場合には必須、それ以外の場合はオプションです。

"R" デフォルト値。データはリモート形式であると見なされ、ローカル形式に変換されます。

"L" データはローカル形式であると見なされ、(変換テーブルでの定義に従って) リモート形式に変換されます。

conversionTable

データ変換に使われる変換テーブルの名前を指定するデータ項目またはリテラル（8 文字、オプション）。デフォルト値は、プログラム生成時に指定された各国語コードに関連付けられた変換テーブルです。

定義に関する考慮事項: リンケージ・オプション・パーツを使用すると、リモート呼び出し、リモート非同期トランザクションの始動、またはリモート・ファイル・アクセスに関してデータの自動変換を行うよう要求することができます。自動変換は、変換対象の引数に定義されているデータ構造を常に使用して行われます。引数に複数の形式が存在する場合は、自動変換を要求しないでください。この場合は、引数の現行値を正しくマップするようにレコード宣言を再定義して、**SysLib.convert** を明示的に呼び出してください。

例:

```
Record RecordA
  record_type char(3);
  item1 char(20);
end

Record RecordB
  record_type char(3);
  item2 bigint;
  item3 decimal(7);
  item4 char(8);
end

Program ProgramX type basicProgram
  myRecordA RecordA;
  myRecordB RecordB {redefines = "myRecordA"};
  myConvTable char(8);

  function main();
    myConvTable = "ELACNENU"; // conversion table for US English
    if (myRecordA.record_type == "00A")
      SysLib.convert(myRecordA, "L", myConvTable);
    else;
      SysLib.convert(myRecordB, "L", myConvTable);
    end;
  end;
```

```

        end
    call ProgramY myRecordA;
end
end

```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

507 ページの『データ変換』

949 ページの『EGL ライブラリー SysLib』

995 ページの『callConversionTable』

defineDatabaseAlias()

システム関数 **SysLib.defineDatabaseAlias** は、ユーザーのコードが既に接続されているデータベースとの新規接続を確立するのに使用可能な別名を作成します。接続が確立されると、この別名は以下のいずれかの関数で使用できます。

- SysLib.connect
- SysLib.disconnect
- SysLib.beginDatabaseTransaction
- SysLib.setCurrentDatabase
- VGLib.connectionService

この別名は、**ReportData** 型の変数の **connectionName** フィールドでも使用できます。

```

SysLib.defineDatabaseAlias(
    alias STRING in,
    database STRING in)

```

alias

2 次パラメーター内で識別された接続の別名として機能する文字列リテラルまたは変数。この別名は、大/小文字を区別しません。

database

SysLib.connect または VGLib.connectionService で指定されたデータベース名。文字型のリテラルまたは変数を使用します。

接続を指定しなかった場合、この関数は現行接続に影響を及ぼします。

以下に例を示します。

```

// データベースに「alias」という別名で接続します。
// これが現行接続になります。
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );

// 同じデータベースに 2 つの接続を確立します。
String db = "database";
defineDatabaseAlias( "alias1", db );
defineDatabaseAlias( "alias2", db );
connect( "alias1", "user", "pwd" );
connect( "alias2", "user", "pwd" );

// 同じデータベースに 2 つの接続を
// 確立する別の方法。
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );

```



```

connect( "database", "user", "pwd" );

// 別名が定義されていますが、使用されていません。
// 2 番目の connect() では、新規の接続は作成されません。
defineDatabaseAlias( "alias", "database" );
connect( "database", "user", "pwd" );
connect( "database", "user", "pwd" );

// 接続を切断するときの別名
// (大/小文字を区別しない) の使用。    defineDatabaseAlias( "alias", "database" );
connect( "aLiAs", "user", "pwd" );
disconnect( "ALIAS" );

// 次の切断呼び出しは、接続が「database」ではなく、
// 「alias」として呼び出されているために失敗しました。
defineDatabaseAlias( "alias", "database" );
connect( "alias", "user", "pwd" );
disconnect( "database" );

// 別名を変更できます。次の呼び出し後、
// 「alias」は「firstDatabase」として呼び出されます。
defineDatabaseAlias( "alias", "firstDatabase" );

次の呼び出し後、
// 「alias」は「secondDatabase」として呼び出されます。
defineDatabaseAlias( "alias", "secondDatabase" );

// 接続が「alias」として残っている場合、
// 最後の呼び出しは失敗します。

```

関連する概念

811 ページの『EGL 関数の構文図』

247 ページの『SQL サポート』

関連する参照項目

952 ページの『beginDatabaseTransaction()』

957 ページの『connect()』

『disconnect()』

970 ページの『setCurrentDatabase()』

980 ページの『connectionService()』

disconnect()

システム関数 **SysLib.disconnect** は、指定したデータベース、または (データベースが指定されていない場合には) 現行データベースからの切断を行います。

```

SysLib.disconnect(
    [database STRING in
    ])

```

database

SysLib.connect または VGLib.connectionService で指定されたデータベース名。
文字型のリテラルまたは変数を使用します。

切断の前に、SysLib.commit または SysLib.rollback を呼び出します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
956 ページの『commit()』
957 ページの『connect()』
969 ページの『rollback()』
980 ページの『connectionService()』

disconnectAll()

システム関数 **SysLib.disconnectAll** は、現在接続中のすべてのデータベースから切断します。

切断の前に、**SysLib.commit** または **SysLib.rollback** を呼び出します。

```
SysLib.disconnectAll( )
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
957 ページの『connect()』
980 ページの『connectionService()』

errorLog()

システム関数 **SysLib.errorLog** は、システム関数 **SysLib.startLog** によって開始されたエラー・ログにテキストをコピーします。

```
SysLib.errorLog(text STRING in)
```

text

エラー・ログで置き換えられることになる値。

エントリーが書き込まれた日時を含むログ・エントリー。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
975 ページの『startLog()』

getCmdLineArg()

システム関数 **SysLib.getCmdLineArg** は EGL プログラムの呼び出しに使用された引数リストから、指定された引数を戻します。指定された引数は、ストリング値として戻されます。

```
SysLib.getCmdLineArg(index INT in)  
returns (result STRING)
```

result

result は、任意の文字項目にすることができます。

index

index は任意の整数項目にすることができます。

- `index = 0` の場合は、コマンド名が戻されます。
- `index = n` の場合は、`n` 番目の引数名が戻されます。
- `n` が引数カウントより大きい場合は、ブランクが戻されます。

次のコード例は、引数リスト内をループします。

```
count int;
argument char(20);

count = 0;
argumentCount = SysLib.getCmdLineArgCount();

while (count < argumentCount)
    argument = SysLib.getCmdLineArg(count)
    count = count + 1;
end
```

SysLib.getCmdLineArg 関数は、Java 環境でのみサポートされます。

メイン EGL プログラムの呼び出し時に、そのプログラムに渡された引数またはパラメーターの数を取得するには、**SysLib.getCmdLineArgCount** 関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
『getCmdLineArgCount()』

getCmdLineArgCount()

システム関数 **SysLib.getCmdLineArgCount** は、メイン EGL プログラムを開始するために使用された引数の数を戻します。

```
SysLib.getCmdLineArgCount( )
returns (result INT)
```

result

result は引数の数です。

次のコード例は、引数リスト内をループします。

```
count int;
argument char(20);

count = 0;
argumentCount = SysLib.getCmdLineArgCount();

while (count < argumentCount)
    argument = SysLib.getCmdLineArg(count)
    count = count + 1;
end
```

SysLib.getCmdLineArgCount 関数は、Java 環境でのみサポートされます。

EGL プログラムの呼び出しに使用された引数リストから、引数を指定して取得するには、**SysLib.getCmdLineArg** 関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

964 ページの『getCmdLineArg()』

getMessage()

システム関数 **SysLib.getMessage** は、Java ランタイム・プロパティー `vgj.message.file` の中で参照されているファイルから、メッセージを戻します。そのメッセージ内への組み込みには、挿入を指定できます。メッセージを取り出した後、そのメッセージをテキスト書式、印刷書式、コンソール書式、Web ページ、またはログ・ファイル内に表示できます。

```
SysLib.getMessage(  
    key STRING in  
    [, insertArray STRING[] in])  
returns (result STRING)
```

result

STRING 型のフィールド。

key

STRING 型の文字フィールドまたはリテラル。このパラメーターは、実行時に使用されるプロパティー・ファイルへのキーを提供します。キーがブランクの場合、メッセージはメッセージ挿入の連結となります。

insertArray

STRING 型の配列。各エレメントには、取り出したメッセージに組み込む挿入が格納されています。

メッセージ・テキスト内の置換シンボルは、次のプロパティー・ファイルからの例に示すように、中括弧で囲まれた整数です。

```
VGJ0216E = {0} is not a valid date mask for {1}.
```

insertArray 内の最初のエレメントは、ゼロの番号が付いたプレースホルダーへ代入され、2 番目のエレメントは 1 の番号が付いたプレースホルダーへ代入され、それ以降も同様に代入されます。

Java ランタイム・プロパティー `vgj.messages.file` によって参照されるファイルのフォーマットは、通常の Java プロパティー・ファイルと同じです。そのフォーマットの詳細については、『プログラム・プロパティー・ファイル』を参照してください。

関連する概念

811 ページの『EGL 関数の構文図』

377 ページの『Java ランタイム・プロパティー』

380 ページの『プログラム・プロパティー・ファイル』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

584 ページの『Java ランタイム・プロパティー (詳細)』

getProperty()

システム関数 **SysLib.getProperty** により、Java ランタイム・プロパティーの値が検索されます。指定されたプロパティーが検出されない場合、この関数により NULL ストリング ("") が戻されます。

```
SysLib.getProperty(propertyName STRING in)  
returns (result STRING)
```

result

STRING 型のフィールド

propertyName

文字変数または定数、あるいは文字列リテラル

関連する概念

811 ページの『EGL 関数の構文図』

377 ページの『Java ランタイム・プロパティー』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

584 ページの『Java ランタイム・プロパティー (詳細)』

loadTable()

システム関数 **SysLib.loadTable** は、ファイルからリレーショナル・データベースにデータをロードします。

```
SysLib.loadTable(  
  fileName STRING in,  
  insertIntoClause STRING in  
  [, delimiter STRING in  
  ]  
)
```

fileName

ファイルの名前。ファイルの名前は完全に限定的な名前か、またはプログラムが起動されるディレクトリーに関連した名前となります。

insertIntoClause

データを提供する表および行を指定します。SQL INSERT ステートメントでは以下の例のように INSERT 文節の構文を使用します。

```
"INSERT INTO myTable(column1, column2)"
```

すべてのテーブル列に関する値が列の順に従ってファイルに含まれている場合には、次のような文節で十分です。

```
"INSERT INTO myTable"
```

delimiter

ファイル内である値と次の値を分離するためのシンボルを指定します。(データの 1 つの行は、改行文字で次の行と分離される必要があります。)

delimiter のデフォルト・シンボルは、Java ランタイム・プロパティーの値である **vgj.default.databaseDelimiter** です。このプロパティーのデフォルト値はパイプ (|) です。

以下のシンボルは使用できません。

- 16 進文字 (0 から 9 まで、a から f まで、および A から F までの範囲)

- 円記号 (¥)
- 改行文字または *CONTROL-J*

リレーショナル・データベース表から情報をアンロードし、ファイルに挿入するには、**SysLib.unloadTable** 関数を使用します。

関連する参照項目

949 ページの『EGL ライブラリー SysLib』
 584 ページの『Java ランタイム・プロパティ (詳細)』
 976 ページの『unloadTable()』

maximumSize()

システム関数 **SysLib.maximumSize** は、動的配列に含めることができる最大行数を返します。具体的には、配列プロパティ **maxSize** の値を返します。

```
SysLib.maximumSize(arrayName anyArray in)
returns (result INT)
```

result 最大行数。

arrayName

動的配列の名前。

定義に関する考慮事項: 値の戻り先の項目は、INT 型か、長さが 9 で小数部がない BIN 型と同等でなければなりません。

配列名は、パッケージ名かライブラリー名、またはその両方で修飾できます。

動的配列以外の項目またはレコードを参照すると、エラーが発生します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

78 ページの『配列』
 949 ページの『EGL ライブラリー SysLib』

queryCurrentDatabase()

システム関数 **SysLib.queryCurrentDatabase** は、現在接続されているデータベースの製品番号およびリリース番号を返します。

```
SysLib.queryCurrentDatabase(
  product CHAR(8) inOut,
  release CHAR(8) inOut)
```

product

データベースのプロダクト名を受け取ります。CHAR 型で長さが 8 の項目を引数として使用する必要があります。

コードで特定のデータベースへ接続するときに受け取るストリングを確認するには、データベースまたはドライバの製品資料を参照するか、またはテスト環境でコードを実行し、受け取る値をファイルへ書き込んでください。

release

データベースのリリース・レベルを受け取ります。 CHAR 型で長さが 8 の項目を引数として使用する必要があります。

コードで特定のデータベースへ接続するときに受け取るストリングを確認するには、データベースまたはドライバの製品資料を参照するか、またはテスト環境でコードを実行し、受け取る値をファイルへ書き込んでください。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

rollback()

システム関数 **SysLib.rollback** は、最後のコミット以降、データベースおよび MQSeries メッセージ・キューに行われた更新内容をリバースします。取り消しは、すべての EGL 生成アプリケーションで生じます。

SysLib.rollback()

エラー状態によりプログラムが終了した場合には、ロールバックが自動的に実行されます。

定義に関する考慮事項: MQ レコードとともに **SysLib.rollback** を使用する場合は、以下の点に注意してください。

- メッセージ・キューの更新は、MQ レコードのパーツで *Include message in transaction* オプションが選択されている場合にのみリカバリー可能です。
- メッセージ **scans** および **add** は、いずれも、リカバリー可能メッセージに関する **commit** および **rollback** の影響を受けます。リカバリー可能メッセージに対して、**scan** の後に **rollback** が出された場合は、メッセージが入力キューに戻されるため、トランザクションが正常に完了できなくても入力メッセージは失われません。また、**rollback** がリカバリー可能メッセージ用の **add** に続いて実行される場合、メッセージはキューから削除されます。

ターゲット・プラットフォーム:

プラットフォーム	互換性に関する考慮事項
iSeries、USS、Windows 2000、Windows NT	リレーショナル・データベースや MQSeries メッセージ・キューに加えられた変更や、クライアント制御の作業単位を使用して呼び出されたリモート・サーバー・プログラムに対する変更が反転されます。

例:

```
SysLib.rollback();
```

関連する概念

811 ページの『EGL 関数の構文図』

334 ページの『作業論理単位』

285 ページの『MQSeries のサポート』

247 ページの『SQL サポート』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

setCurrentDatabase()

システム関数 **SysLib.setCurrentDatabase** は、指定されたデータベースを現行のアクティブ・データベースにします。

```
SysLib.setCurrentDatabase(database STRING in)
```

database

SysLib.connect または VGLib.connectionService で指定されたデータベース名。
文字型のリテラルまたは変数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

957 ページの『connect()』

980 ページの『connectionService()』

setError()

システム関数 **SysLib.setError** は、メッセージをページ・ハンドラー内の項目、またはページ・ハンドラー全体と関連付けます。メッセージは、JSP の JSF メッセージまたはメッセージ・タグのロケーションに配置され、関連する Web ページが表示されると表示されます。

検証機能が **SysLib.setError** を呼び出す場合、この機能の終了時に Web ページが自動的に再表示されます。

```
SysLib.setError(  
    itemInError anyPageItem in,  
    msgKey STRING in  
    {, itemInsert sysLibItemInsert in})
```

```
SysLib.setError(  
    this enumerationThis in,  
    msgKey STRING in  
    {, itemInsert sysLibItemInsert in})
```

```
SysLib.setError(msgText STRING in)
```

itemInError

エラーのあるページ・ハンドラー項目の名前。

this

SysLib.setError を発行するページ・ハンドラーです。この場合、メッセージは項目固有ではなく、ページ・ハンドラー全体に関連付けられています。**this** の詳細については、『変数および定数の参照』を参照してください。

msgKey

実行時に使用されるメッセージ・リソース・バンドルまたはプロパティ・ファイルにキーを提供する文字項目またはリテラル (型 **CHAR** または **MBCHAR**) です。キーがブランクの場合、メッセージはメッセージ挿入の連結となります。

itemInsert

出力メッセージへの挿入として含まれる文字項目またはリテラルです。メッセージ・テキスト内の置換シンボルは、次の例のような中括弧で囲まれた整数です。

```
Invalid file name {0}
```

msgText

ほかの引数を指定しない場合に指定可能な文字項目またはリテラルです。テキストは、ページ全体に関連付けられます。

複数のメッセージを 1 つの項目またはページ・ハンドラーに関連付けられます。EGL ランタイムは、ページが再表示されるときにメッセージを表示します。コントロールが転送される (特に、ページ・ハンドラーが **forward** ステートメントを実行する) と、それらのメッセージは消去されます。

関連する概念

207 ページの『ページ・ハンドラー』

62 ページの『EGL での変数の参照』

関連するタスク

812 ページの『EGL ステートメントおよびコマンドの構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

629 ページの『forward』

setLocale()

システム関数 **SysLib.setLocale** はページ・ハンドラー内で使用します。この関数は、Java ロケールを設定します。Java ロケールは、ランタイムの振る舞いに関する以下の特徴を決定します。

- ラベルおよびメッセージで使用する国の言語
- デフォルトの日時形式

例えば、Web ページに言語リストを表示して、ユーザーの選択に基づいて Java ロケールを設定します。新規 Java ロケールは、以下のいずれかが発生するまで使用中になります。

- **SysLib.setLocale** を再び呼び出した場合、または
- ブラウザー・セッションが終了した場合、または
- その他、新規 Web ページが表示された場合。

前述の場合には、次の Web ページが、ブラウザで指定した Java ロケールに復帰します (デフォルト)。

ユーザーが書式を処理依頼した場合、または新規ウィンドウをオープンするリンクをクリックした場合に、元のウィンドウの Java ロケールは、新規ウィンドウのロケールの影響を受けません。

SysLib.setLocale は、JDK 1.1 および 1.2 API 文書の場合の `java.util.Locale` クラスに準拠します。言語コード ISO 639 および国コード ISO 3166 を参照してください。

```
SysLib.setLocale(  
    languageCode CHAR(2) in,  
    countryCode CHAR(2) in  
    [, variant CHAR(2) in])
```

languageCode

リテラルとして指定された 2 文字言語コード、または CHAR タイプの項目に含まれている 2 文字言語コードです。ISO 639 で定義されている言語コードのみ有効です。

countryCode

リテラルとして指定された 2 文字国別コード、または CHAR タイプの項目に含まれている 2 文字国別コードです。ISO 3166 で定義されている国別コードのみ有効です。

variant

バリエーション (リテラルとして指定されたコード、または CHAR タイプの項目に含まれているコード) です。このコードは、Java 仕様の一部ではなく、ブラウザーやユーザー環境の他の状況に依存します。

関連する概念

811 ページの『EGL 関数の構文図』

207 ページの『ページ・ハンドラー』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

setRemoteUser()

システム関数 **SysLib.setRemoteUser** は、リモート・プログラムを呼び出すときに使用されるユーザー ID およびパスワードを設定します。

```
SysLib.setRemoteUser(  
    userID STRING in,  
    password STRING in)
```

userID

リモート・システムのユーザー ID。

password

リモート・システムのパスワード。

リンケージ・オプション・パーツの CallLink エLEMENTのプロパティー remoteComType がリモート呼び出しの CICSJ2C、CICSECI、または JAVA400 である場合、許可は、**SysLib.setRemoteUser** に渡された値 (非ブランクの場合) に基づきます。値がブランク、または指定されていない場合は、CSOUID (ユーザー ID) および CSOPWD (パスワード) のプロパティーを含むファイル **csouidpwd.properties** から値が検索されます。いずれの方法も使用しない場合は、EGL ランタイムは、ユーザー名およびパスワードを使用しないで呼び出しを実行します。

SysLib.setRemoteUser を呼び出す前に、コードにより、ユーザーにユーザー ID とパスワードの入力を求めるダイアログ・ボックスを表示する、Java アクセス関数を発行できます。ユーザーが情報を提供しない場合に有効となるデフォルトとして、**csouidpwd.properties** 内の値のいずれかまたは両方を使用できます。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

411 ページの『リモート呼び出し用 csouidpwd.properties ファイル』

949 ページの『EGL ライブラリー SysLib』

456 ページの『callLink エLEMENTの remoteComType』

size()

システム関数 **SysLib.size** は、指定されたデータ・テーブルの行数、または、指定された配列のエLEMENT数を返します。この配列は、構造体項目配列、データ項目またはレコードの動的配列です。

```
SysLib.size(arrayName anyArray in)
returns (result INT)
```

result

指定されたデータ・テーブルの行数、または指定された配列のエLEMENT数。

arrayName

配列またはデータ・テーブルの名前。

定義に関する考慮事項: 値の戻り先の項目は、INT 型か、長さが 9 で小数部がない BIN 型と同等でなければなりません。

配列名 (*arrayName*) が別の配列の副構造体ELEMENTに入っている場合、戻される値は、その項目が入っている構造体における出現総数ではなく、その構造体項目そのものの出現回数になります (『例』セクションを参照)。

配列名は、パッケージ名かライブラリー名、またはその両方で修飾できます。

配列以外の項目またはレコードを参照する場合は、エラーが発生します。

例: この例では、**SysLib.size** によって戻された値を使用してループを制御します。

```
// 数の配列の合計を計算
sum = 0;
i = 1;
myArraySize = SysLib.size(myArray);

while (i <= myArraySize)
    sum = myArray[i] + sum;
    i = i + 1;
end
```

次に以下のレコード・パーツを検討します。

```
Record myRecordPart
    10 siTop CHAR(40)[3];
    20 siNext CHAR(20)[2];
end
```

myRecordPart を基にしたレコードを作成する場合は、**SysLib.size(siNext)** を使用して、従属配列の occurs 値を判別できます。

```
// count を 2 に設定
count = SysLib.size(myRecord.siTop.siNext);
```

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

78 ページの『配列』

949 ページの『EGL ライブラリー SysLib』

startCmd()

システム関数 **SysLib.startCmd** は、システム・コマンドを実行し、コマンドが終了するまで待ちません。

```
SysLib.startCmd(  
    commandString STRING in  
    [, modeString STRING in  
    ])
```

commandString

呼び出すオペレーティング・システム・コマンドを識別します。

modeString

modeString は、任意の文字項目またはストリング項目にすることができます。この項目は、次の 2 つのモードのどちらでもかまいません。

- *form*: 入力各文字が、入力されるたびにプログラムから使用可能になります。つまり、すべてのキー・ストロークが直接、指定したコマンドへ渡されます。
- *line*: 入力は、改行文字が使用されるまで使用可能になりません。つまり、ENTER キーが押されるまで、指定したコマンドへ情報が渡されず、押された後、入力された行全体がコマンドへ送られます。

The following code example

example from Arlan here...

実行されるシステム・コマンドは、実行中のプログラムから可視であることが必要です。例えば、**callCmd**("mySpecialProgram.exe") を実行する場合、プログラム "mySpecialProgram.exe" は環境変数 PATH によってポイントされたディレクトリーに入っている必要があります。完全なディレクトリー・ロケーションを指定して、例えば、**callCmd**("program files/myWork/mySpecialProgram.exe") とすることもできます。

SysLib.startCmd 関数は、Java 環境でのみサポートされます。

コマンドが終了するまで待つシステム・コマンドを実行するには、**SysLib.callCmd** 関数を使用します。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

dummy change as necessary

startLog()

システム関数 **SysLib.startLog** は、エラー・ログを開きます。プログラムが **SysLib.errorLog** を呼び出すたびに、そのログにテキストが書き込まれます。

```
SysLib.startLog(logFile STRING in)
```

logFile

エラー・ログ。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

964 ページの『errorLog()』

startTransaction()

システム関数 **SysLib.startTransaction** は、メインプログラムを非同期に起動し、そのプログラムをプリンターまたは端末に関連付け、レコードを渡します。受信プログラムが EGL によって生成されている場合には、そのレコードは入力レコードの初期化で使います。受信側が VisualAge Generator によって生成されている場合には、そのレコードは作業用ストレージの初期化で使います。

この関数のデフォルトの振る舞いは、同一 Java パッケージ内に存在するプログラムを起動することです。この振る舞いを変更するには、呼び出し側プログラムの生成に使用されるリンケージ・オプション・パーツに **asynchLink** エレメントを指定します。

Java プログラムは、同一マシン上の他の Java プログラムにのみ転送できます。

```
SysLib.startTransaction(  
  request anyBasicRecord in  
  [, prID startTransactionPrId in  
  [, termID CHAR(4) in ]])
```

request

基本レコードの名前。以下の形式になっている必要があります。

- 最初の 2 バイト (SMALLINT 型または小数部のない BIN 型) に、開始するトランザクションに渡されるデータの長さが格納され、その後にトランザクションには渡されない 2 つのフィールド (このフィールドを含む) 用の 10 バイトが続きます。
- 次の 8 バイト (CHAR 型) も渡されませんが、このバイトには始動するプログラムの名前が入ります。
- 要求レコードの残りの部分は渡されます。

prID

この 4 バイト項目は、指定しても無視されます。

termID

この CHAR 型 4 バイト項目は、指定しても無視されます。*termID* を指定する場合は、*prID* を指定する必要があります。

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

410 ページの『asynchLink エlement』

949 ページの『EGL ライブラリー SysLib』

996 ページの『errorCode』

988 ページの『printerAssociation』

697 ページの『transfer』

unloadTable()

システム関数 **SysLib.unloadTable** はリレーショナル・データベースからファイルにデータをアンロードします。

```
SysLib.unloadTable(  
    fileName STRING in,  
    selectStatement STRING in  
    [, delimiter STRING in  
    ])
```

fileName

ファイルの名前。ファイルの名前は完全に限定的な名前か、またはプログラムが起動されるディレクトリーに関連した名前となります。

selectStatement

リレーショナル・データベースからデータを選択するための基準を指定します。以下の例のように、ホスト変数を含まない SQL SELECT ステートメントの構文を使用します。

```
"SELECT column1, column2 FROM myTABLE  
WHERE column3 > 10"
```

delimiter

ファイル内である値と次の値を分離するためのシンボルを指定します。(データの 1 つの行は、改行文字で次の行と分離される必要があります。)

delimiter のデフォルト・シンボルは、Java ランタイム・プロパティーの値である **vgj.default.databaseDelimiter** です。このプロパティーのデフォルト値はパイプ (|) です。

以下のシンボルは使用できません。

- 16 進文字 (0 から 9 まで、a から f まで、および A から F までの範囲)
- 円記号 (¥)
- 改行文字または *CONTROL-J*

ファイルから情報をロードし、リレーショナル・データベース表に挿入するには、**SysLib.loadTable** 関数を使用します。

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

967 ページの『loadTable()』

584 ページの『Java ランタイム・プロパティー (詳細)』

verifyChkDigitMod10()

システム関数 **SysLib.verifyChkDigitMod10** は、一連の整数で始まる文字項目のモジュラス 10 チェック・ディジットを検証します。

```
SysLib.verifyChkDigitMod10(  
    text anyChar in,  
    checkLength SMALLINT in,  
    result SMALLINT inOut)
```

text

一連の整数で始まる文字項目。この項目には、チェック・ディジット用に追加の 1 桁があり、位置は他の整数の右隣になります。

checkLength

text 項目から使用する文字の数を含む項目。チェック・ディジットに使用される桁も含みます。この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

result

次の 2 つの値のいずれかを受け取る項目です。

- 計算されたチェック・ディジットが *text* の値と一致する場合は、0
- 計算されたチェック・ディジットがその値と一致しない場合は、1

この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

SysLib.verifyChkDigitMod10 は、関数呼び出しの文で使用できます。つまり、テキスト書式で項目バリデーターとして使用できます。

例: 次の例では、myInput は CHAR 型の項目であり、値は 1734284 です。myLength は SMALLINT 型の項目であり、値は 7 です。myResult は SMALLINT 型の項目です。

```
SysLib.verifyChkDigitMod10 (myInput, myLength, myResult);
```

モジュラス 10 チェック・ディジットを導き出すためにアルゴリズムが使用され、すべての場合に、チェック・ディジット位置にある数は考慮されません。ただし、アルゴリズムが完了したときに、計算値がチェック・ディジット位置にある数と比較されます。

アルゴリズムについて、例の値を使用して説明します。

1. 入力数の最小桁に 2 を乗算し、右から左の順で 1 桁おきに 2 を乗算します。

$$\begin{aligned}8 \times 2 &= 16 \\4 \times 2 &= 8 \\7 \times 2 &= 14\end{aligned}$$

2. 2 を乗算していない入力数字 (132) に、生成された数字 (16814) を加算します。

$$1 + 6 + 8 + 1 + 4 + 1 + 3 + 2 = 26$$

3. チェック・ディジットを算出するには、0 で終わる次に大きい数値から合計を減算します。

$$30 - 26 = 4$$

減算の結果が 10 ならば、チェック・ディジットは 0 です。

この例では、計算されたチェック・ディジットがチェック・ディジット位置と一致し、myResult の値は 0 です。

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

72 ページの『検証プロパティー』

verifyChkDigitMod11()

システム関数 **SysLib.verifyChkDigitMod11** は、一連の整数で始まる文字項目のモジュラス 11 チェック・ディジットを検証します。

```
SysLib.verifyChkDigitMod11(  
    text anyChar in,  
    checkLength SMALLINT in,  
    result SMALLINT inOut)
```

text

一連の整数で始まる文字項目。この項目には、チェック・ディジット用に追加の 1 桁があり、位置は他の整数の右隣になります。

checkLength

text 項目から使用する文字の数を含む項目。チェック・ディジットに使用される桁も含まれます。この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

result

次の 2 つの値のいずれかを受け取る項目です。

- 計算されたチェック・ディジットが *text* の値と一致する場合は、0
- 計算されたチェック・ディジットがその値と一致しない場合は、1

この項目は、4 桁の SMALLINT 型または BIN 型で、小数点以下の桁はありません。

SysLib.verifyChkDigitMod11 は、関数呼び出しのステートメントで使用できます。つまり、テキスト書式で項目バリデーターとして使用できます。

例: 次の例では、myInput は CHAR 型の項目であり、値 56621869 を含みます。myLength は SMALLINT 型の項目であり、値 8 を含みます。myResult は SMALLINT 型の項目です。

```
sysLib.verifyChkDigitMod11 (myInput, myLength, myResult);
```

モジュラス 11 チェック・ディジットはアルゴリズムを使用して派生され、すべての場合に、チェック・ディジットの位置に格納されている数値は考慮されません。ただし、アルゴリズムが完了した時点で、計算された値がチェック・ディジット位置の数値と比較されます。アルゴリズムについて、例の値を使用して説明します。

1. 入力数値の単位の位にある数字に 2 を乗算し、10 の位に 3 を、100 の位に 4 を乗算する、というように繰り返します。ただし、乗数として使用する数字は myLength " 1 を最大数とします。入力数値にそれ以上の桁がある場合は、2 を乗数としてこの手順をもう一度最初から実行します。

```
6 x 2 = 12  
8 x 3 = 24  
1 x 4 = 4
```

2 x 5 = 10
6 x 6 = 36
6 x 7 = 42
5 x 2 = 10

2. 最初のステップの積を加算し、合計を 11 で除算します。

(12 + 24 + 4 + 10 + 36 + 42 + 10) / 11
= 138 / 11
= 12 余り 6

3. チェック・ディジットを算出するには、11 から余りを減算して自己検査ディジットを得ます。

11 - 6 = 5

余りが 0 または 1 である場合、チェック・ディジットは 0 です。

この例では、計算されたチェック・ディジットがチェック・ディジット位置と一致し、myResult の値は 0 です。

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

72 ページの『検証プロパティ』

wait()

システム関数 **SysLib.wait** は、指定した秒数だけ実行を中断します。

SysLib.wait(*timeInSeconds* **BIN(9,2)** in)

timeInSeconds

時間は、任意の数値項目またはリテラルです。100 分の 1 秒の小数部は、数字が整数でない場合に使用します。

2 つの非同期実行プログラムが、共用ファイルまたはデータベース内のレコードを介して通信する必要がある場合には、**SysLib.wait** を使用できます。1 つのプログラムが、他のプログラムが共用レコードの情報を更新するまで、処理を中断する必要がある場合があります。

例

SysLib.wait(15); // 15 秒間待機する

関連する概念

811 ページの『EGL 関数の構文図』

関連する参照項目

949 ページの『EGL ライブラリー SysLib』

EGL ライブラリー VGLib

以下の表に VGLib 関数を示します。

システム関数/呼び出し	説明
<code>connectionService(userID, password, serverName [, product, release [, connectionOption]])</code>	これには、次の 2 つの利点があります。 <ul style="list-style-type: none">プログラム実行時にデータベースへの接続または切断を可能にします。データベースのプロダクト名とリリース・レベルを受け取ります (オプション)。実行時処理がデータベースの特性に依存して実行されるように、受け取った情報を case、if、または while ステートメントで使用できます。
<code>result = getVAGSysType()</code>	プログラムを実行するターゲット・システムを識別します。

connectionService()

システム関数 **VGLib.connectionService** には、以下の 2 つの利点があります。

- プログラム実行時にデータベースへの接続または切断を可能にします。
- データベースのプロダクト名とリリース・レベルを受け取ります (オプション)。実行時処理がデータベースの特性に依存して実行されるように、受け取った情報を **case**、**if**、または **while** ステートメントで使用できます。

VGLib.connectionService を使用して Java プログラムから新規接続を作成する場合、システム変数 **VGVar.sqlIsolationLevel** を設定することによって分離レベルを指定します。

VGLib.connectionService は、VisualAge Generator および EGL 5.0 からマイグレーションされたプログラム専用です。関数は、(開発時間に) EGL 設定 **VisualAge Generator** との互換性 が選択されている場合、または (生成時間に) ビルド記述子オプション **VAGCompatibility** が *yes* に選択されている場合にサポートされます。

新規プログラムの場合、代わりに以下のシステム関数を使用します。

- `SysLib.connect`
- `SysLib.disconnect`
- `SysLib.disconnectAll`
- `SysLib.queryCurrentDatabase`
- `SysLib.setCurrentDatabase`

VGLib.connectionService は値を戻しません。

```
VGLib.connectionService(  
  userID CHAR(8) in,  
  password CHAR(8) in,  
  serverName CHAR(18) in  
  [, product CHAR(8) inOut,  
    release CHAR(8) inOut  
  [, connectionOption STRING in  
  ]])
```

userID

データベースへアクセスするときに使用されるユーザー ID。CHAR 型で長さが 8 の項目を引数として使用する必要があります。リテラルは無効です。引数を指定する必要があります。背景情報については、『データベースの権限とテーブル名』を参照してください。

password

データベースへアクセスするときに使用されるパスワード。CHAR 型で長さが 8 の項目を引数として使用する必要があります。リテラルは無効です。引数を指定する必要があります。

serverName

接続を指定し、引数 *product* と *release* が **VGLib.connectionService** の呼び出しに含まれている場合には、指定した接続を使用してこれらの引数に値を代入します。

CHAR 型で長さが 18 の項目を引数 *serverName* として使用する必要があります。有効な値を以下に示します。

ブランク (内容なし)

接続が残っている場合、**VGLib.connectionService** はその接続を維持します。接続が残っていない場合には、実行単位の開始時に有効な接続状況へ、結果 (値代入以外の 結果) が戻されます。詳細については、『デフォルト・データベース』を参照してください。

RESET

RESET を指定すると、デフォルト・データベースに再接続します。ただしデフォルト・データベースが使用できない場合には、接続状況は変更されません。

詳細については、『デフォルト・データベース』を参照してください。

serverName

データベースを識別します。

- 物理データベース名を検出するには、プロパティ

vgj.jdbc.database.server (*server* は **VGLib.connectionService** 呼び出しで指定した名前) を検索します。このプロパティが定義されていない場合は、**VGLib.connectionService** 呼び出しで指定したサーバー名がそのまま使用されます。

- J2EE 接続と非 J2EE 接続ではデータベース名の形式が異なります。

- J2EE 環境で実行されるプログラムを生成した場合には、JNDI レジストリーでのデータ・ソースのバインド先となる名前 (jdbc/MyDB など) を使用します。ビルド記述子オプション **J2EE** が YES に設定されている場合に、この状態が発生します。
- 非 J2EE JDBC 環境のためのプログラムを生成した場合には、接続 URL (jdbc:db2:MyDB など) を使用します。オプション **J2EE** が NO に設定されている場合に、この状態が発生します。

product

データベースのプロダクト名を受け取ります。引数を指定する場合には、CHAR 型で長さが 8 の項目を引数として使用する必要があります。

コードで特定のデータベースへ接続するときに受け取るストリングを確認するには、データベースまたはドライバの製品資料を参照するか、またはテスト環境でコードを実行し、受け取る値をファイルへ書き込んでください。

release

データベースのリリース・レベルを受け取ります。引数を指定する場合には、CHAR 型で長さが 8 の項目を引数として使用する必要があります。

コードで特定のデータベースへ接続するときに受け取るストリングを確認するには、データベースまたはドライバの製品資料を参照するか、またはテスト環境でコードを実行し、受け取る値をファイルへ書き込んでください。

connectionOption

有効な値は以下のとおりです。

DIE

デフォルトは、「DIE」です。オプション名の *I* は、シングル・フェーズ・コミットがサポートされていることを示し、*E* は、切断が明示的でないことを示します。この場合、コミットまたはロールバックは既存の接続で有効ではありません。

データベースへ接続しても、カーソルの終了、ロックの解放、既存の接続の終了は行われません。ただし、実行単位がすでに同一データベースへ接続している場合には、DIE を指定すると、DISC を指定してから DIE を指定した場合と同様の効果が得られます。

複数のデータベースから読み込むために複数の接続を使用している場合には、作業単位では 1 つのデータベースのみを更新する必要があります。これは、シングル・フェーズ・コミットのみが実行可能であるためです。

D1A

オプション名の *I* は、シングル・フェーズ・コミットがサポートされていることを示し、*A* は、切断が自動的であることを示します。このオプションの特性は、以下のとおりです。

- 1 度に 1 つのデータベースにしか接続できない
- データベースへのコミット、ロールバック、または接続によって、既存の接続が終了する

DISC

指定されているデータベースから切断します。データベースから切断すると、このデータベースについてのみロールバックが実行され、ロックが解放されます。

DCURRENT

現在接続されているデータベースから切断します。データベースから切断すると、このデータベースについてのみロールバックが実行され、ロックが解放されます。

DALL

すべてのデータベースから切断します。すべてのデータベースから切断すると、これらのデータベースでロールバックが実行されます。ただし、他のリカバリー可能リソースではロールバックは実行されません。

SET

接続を現行接続に設定します。(デフォルトでは、実行単位内の最新の接続が現行接続になります。)

VisualAge Generator との互換性を維持するため、値 R、D1C、D2A、D2C、D2E がサポートされています。ただしこれらの値は D1E と同等です。

定義に関する考慮事項: **VGLib.connectionService** は、以下のシステム変数を設定します。

- VGVar.sqlerrd
- SysVar.sqlca
- SysVar.sqlcode
- VGVar.sqlWarn

例:

```
VGLib.connectionService(myUserId, myPassword,  
myServerName, myProduct, myRelease, "D1E");
```

関連する概念

- 811 ページの『EGL 関数の構文図』
- 334 ページの『作業論理単位』
- 798 ページの『実行単位』
- 247 ページの『SQL サポート』

関連するタスク

- 812 ページの『EGL ステートメントおよびコマンドの構文図』
- 394 ページの『J2EE JDBC 接続のセットアップ』
- 283 ページの『標準 JDBC 接続の作成方法について』

関連する参照項目

- 506 ページの『データベースの権限とテーブル名』
- 270 ページの『デフォルト・データベース』
- 980 ページの『EGL ライブラリー VGLib』

- 584 ページの『Java ランタイム・プロパティー (詳細)』
- 435 ページの『sqlDB』
- 1000 ページの『sqlca』
- 1001 ページの『sqlcode』
- 1015 ページの『sqlerrd』
- 1016 ページの『sqlerrmc』
- 1016 ページの『sqlIsolationLevel』
- 1017 ページの『sqlWarn』

getVAGSysType()

システム関数 **VGLib.getVAGSysType** は、プログラムを実行するターゲット・システムを識別します。この関数がサポートされるのは、(開発時に) プログラム・プロパティー **VAGCompatibility** が選択された場合、または (生成時に) ビルド記述子オプション **VAGCompatibility** が *yes* に設定された場合です。

生成された出力が Java ラッパーの場合、**VGLib.getVAGSysType** は使用できません。それ以外の場合、この関数は、VisualAge Generator EZESYS 特殊関数ワードによって戻された文字値を返します。現在のシステムが VisualAge Generator によってサポートされていない場合、この関数は **SysVar.systemType** によって戻されたコードに相当する大文字の文字列を返します。

```
VGLib.getVAGSysType( )
returns (result CHAR(8))
```

result

次の表に示される、システム・タイプ・コードを含む文字ストリング。

VGLib.getVAGSysType は、**SysVar.systemType** 内の値に 相当する VisualAge Generator の値を返します。

sysVar.systemType の値	VGLib.getVAGSysType から戻される値
AIX	"AIX"
DEBUG	"ITF"
ISERIESC	"OS400"
ISERIESJ	"OS400"
LINUX	"LINUX"
USS	"OS390"
WIN	"WINNT"

VGLib.getVAGSysType から戻された値は、文字ストリングとしてのみ使用できます。戻された値をオペランド *is* または *not* とともに論理式で使用することはできません。**sysVar.systemType** では使用できます。

```
// sysVar.systemType に対してのみ有効
if sysVar.systemType is AIX
  call myProgram;
end
```

VGLib.getVAGSysType は、assignment または **move** ステートメントのソースとしてのみ使用できます。

VGLib.getVAGSysType には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8 (ブランクが埋め込まれる)

converse 後、値は常にリストアされますか

はい

VGLib.getVAGSysType の代わりに **sysVar.systemType** を使用することをお勧めします。

定義に関する考慮事項: **VGLib.getVAGSysType** の値は、生成時にどのコードが検証されるかには影響しません。例えば、以下の **add** ステートメントは、Windows 用に生成を行う場合でも検証されます。

```

mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();
if (mySystem == "AIX")
    add myRecord;
end

```

ターゲット・システムで実行できないコードを有効化することを避けるために、有効化しない文を別のプログラムに移動してください。その後、元のプログラムで新しいプログラムを条件付きで呼び出します。

```

mySystem CHAR(8);
mySystem = VGLib.getVAGSysType();

if (mySystem == "AIX")
    call myAddProgram myRecord;
end

```

別の問題解決方法もありますが、**VGLib.getVAGSysType** の代わりに **sysVar.systemType** を使用することが必要です。詳細については、『*eliminateSystemDependentCode*』を参照してください。

関連する参照項目

980 ページの『EGL ライブラリー VGLib』
 424 ページの『*eliminateSystemDependentCode*』
 1002 ページの『systemType』

EGL ライブラリー外部のシステム変数

EGL ライブラリーに含まれる変数は、実行単位に対してグローバルです。他のシステム変数は、異なるスコープ特性を持ち、次のように分類されます。

ConverseVar

textUI アプリケーションで主に役立つ変数。

SysVar

汎用に役立つ変数。

VGVar

VisualAge Generator からマイグレーションされたアプリケーションで主に役立つ変数。

別の、同じ名前の識別子がスコープにある時に、システム変数を参照している場合は、修飾子としてカテゴリー名を含める必要があります。例えば、2 番目の **eventKey** という名前の変数がスコープにある場合、**eventKey** ではなく、**ConverseVar.eventKey** と指定する必要があります。同じ名前の識別子がスコープにない場合は、修飾子はオプションです。

関連する概念

62 ページの『EGL での変数の参照』
 60 ページの『有効範囲指定の規則と EGL での「this」』

関連する参照項目

986 ページの『ConverseVar』
 991 ページの『SysVar』
 1006 ページの『VGVar』

ConverseVar

修飾子 **ConverseVar** は、次の表にリストされた各 EGL システム変数の名前の前に付きます。これらの変数は、textUI アプリケーションで主に役立ちます。

システム変数	説明
commitOnConverse	セグメント化されていないプログラムが converse を発行する前に、リソースのコミットおよびリリースがテキスト・アプリケーションで起こるかどうかを指定します。デフォルト値は、非セグメント化プログラムの場合が 0 (いいえ の意味)、セグメント化プログラムの場合は 1 (はい の意味) です。
eventKey	書式を EGL テキストまたはプログラムに戻すためにユーザーが押したキーを識別します。
printerAssociation	実行時に、印刷書式を印刷するときの出力の宛先を指定できるようにします。
segmentedMode	テキスト・アプリケーションで使用され、 converse ステートメントの効果を変更します。ただし、呼び出し先プログラムでは変数のこの目的は無視されます。
validationMsgNum	テキスト・アプリケーション内の ConverseLib.validationFailed によって割り当てられた値を含むため、検証機能がエラーを報告したかどうかを判別するために使用できます。

関連する概念

62 ページの『EGL での変数の参照』

60 ページの『有効範囲指定の規則と EGL での「this」』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

commitOnConverse

システム変数 **ConverseVar.commitOnConverse** は、セグメント化されていないプログラムが **converse** を発行する前に、テキスト・アプリケーション内のリソースのコミットおよびリリースが行われるかどうかを指定します。デフォルト値は、非セグメント化プログラムの場合が 0 (いいえ)、セグメント化プログラムの場合は 1 (はい) です。

ConverseVar.commitOnConverse は以下のいずれかの方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- **case**、**if**、または **while** ステートメントで使用される論理式での変数として使用
- **return** または **exit** 文内の引数として使用

ConverseVar.commitOnConverse には、ほかに以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

はい

この変数の使用の詳細については、『セグメンテーション』を参照してください。

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

616 ページの『converse』

985 ページの『EGL ライブラリー外部のシステム変数』

eventKey

システム変数 **ConverseVar.eventKey** は、ユーザーが EGL テキスト・プログラムに書式を戻すために押したキーを識別します。この値は、プログラムが **converse** ステートメントを実行するごとにリセットされます。

EGL コードに入力書式がない場合、**ConverseVar.eventKey** の初期値は **ENTER** になります。

以下の値が有効です (大文字、小文字、大文字と小文字の両方が含まれていても構いません)。

- **ENTER**
- **BYPASS** (書式のバイパス・キーとして指定されたキーを参照するか、書式にキーが指定されていない場合は **formGroup** のバイパス・キーとして指定されたキーを参照します。**formGroup** にキーが指定されていない場合、プログラムのバイパス・キーとして指定されたキーを参照します)
- **PA1** から **PA3**
- **PF1** から **PF24** (F1 から F24 にも使用される)
- **PAKEY** (PA キーの場合)
- **PFKEY** (PF または F キーの場合)

注: **PA** キーは常時バイパス・キーとして扱われます。

ConverseVar.eventKey は、**if** または **while** ステートメントのオペランドとして使用できます。

このシステム変数には、以下のような特性があります。

プリミティブ型

CHAR

データ長

1

複数のセグメントにわたって値が保管されるかどうか

いいえ

ConverseVar.eventKey は、バッチ・プログラムでは無効です。

例: **ConverseVar.eventKey** の比較演算子は、この例のように *is* または *not* です。

```
if (ConverseVar.eventKey IS PF3)
  exit program(0);
end
```

関連する参照項目

538 ページの『論理式』

985 ページの『EGL ライブラリー外部のシステム変数』

printerAssociation

システム変数 **ConverseVar.printerAssociation** を使用すると、実行時に、印刷書式を印刷するときの出力の宛先を指定することができます。

この変数は、以下のいずれかの方法で使用できます。

- assignment ステートメントまたは move ステートメントのソースまたはターゲットとして使用
- 論理式の比較値として
- return ステートメントの値として

ConverseVar.printerAssociation には以下のような特性があります。

プリミティブ型

CHAR

データ長

ファイル・タイプによって異なる

converse 後、値は常にリストアされますか

はい

ConverseVar.printerAssociation は、生成時に指定された、またはデバッグの目的で指定されたシステム・リソース名に初期設定されます。プログラムが別のプログラムに制御を渡すと、**ConverseVar.printerAssociation** の値は受け取り側のプログラムのデフォルト値に設定されます。

特定の印刷書式について、複数の印刷ジョブが可能であったとしても、close ステートメントは、**ConverseVar.printerAssociation** の現在の値に関連するファイルのみをクローズします。

Java 出力用の詳細情報: Java 出力では、**ConverseVar.printerAssociation** を、次のように 2 つの部分から成るストリングをコロンで区切って設定します。

jobID:destination

jobID 各印刷ジョブを一意的に識別する一続きの (コロンが含まれていない) 文字。文字の大/小文字が区別され (*job01* と *JOB01* が区別されます)、印刷ジョブがクローズした後、*jobID* を再利用することができます。

コード内のイベントの流れに応じて、異なる種類の出力や異なる出力順序をプロモートするのに別々のジョブを使用することができます。例えば、次の EGL ステートメントの順序を見てください。

```
ConverseVar.printerAssociation = "job1";
print form1;
ConverseVar.printerAssociation = "job2";
print form2;
ConverseVar.printerAssociation = "job1";
print form3;
```

プログラムの終了時に、次の 2 つの印刷ジョブが作成されます。

- form1 (その後に form3 が続く)
- form2 (単独)

destination

出力を受け取るプリンターまたはファイル。

ストリング *destination* の指定はオプションです。印刷ジョブがオープンしている場合は無視されます。ストリングを指定しなかった場合、以下の記述が当てはまります。

- *destination* の前のコロンを省略できる。
- ほとんどの場合、プログラムにより印刷プレビュー・ダイアログが表示され、ここからユーザーが出力用のプリンターまたはファイルを指定できる。UNIX で `curses` ライブラリーが使用される場合は、例外が発生します。この場合は、印刷ジョブはデフォルト・プリンターに出力されます。

Windows 2000/NT/XP 用の出力を生成する場合は、*destination* の設定には以下の記述が当てはまります。

- 出力をデフォルト・プリンターに送信する場合は、以下のようにする。
 - リソース関連パーツの **fileName** プロパティーと一致する値を指定する。
 - Java ランタイム・プロパティーを変更して、関連するファイル・タイプの値を *spool* (*seqws* ではなく) にする。例えば、リソース関連パーツで **fileName** プロパティーの値が *myFile* で、**systemName** の値が *printer* の場合、`vgj.ra.myFile.fileType` が *spool* (*seqws* ではなく) に設定されるように Java ランタイム・プロパティーの設定を変更する必要があります。変更後のプロパティーは、以下のようになります。

```
vgj.ra.myFile.systemName=printer
vgj.ra.myFile.fileType=spool
```

- 出力をファイルに送信する場合は、リソース関連パーツの関連する **fileType** プロパティーの値が *seqws* のとき、リソース関連パーツの **fileName** プロパティーと一致する値を指定する。**systemName** プロパティーは、出力を受け取るオペレーティング・システム・ファイルの名前を含むリソース関連パーツです。
- *destination* の値として、値 *printer* は指定しない。これを指定した場合、ユーザーに対して印刷プレビュー・ダイアログが表示されますが、今後のバージョンの EGL では振る舞いに変更される可能性があります。

UNIX 用の出力を生成する場合は、*destination* の設定には以下の記述が当てはまります。

- 出力をデフォルト・プリンターに送信する場合は (`curses` ライブラリーを使用しているかどうかに関係なく)、リソース関連パーツの関連する

fileType プロパティの値が *spool* のとき、リソース関連パーツの
fileName プロパティと一致する値を指定する。

- 出力をファイルに送信する場合は、リソース関連パーツの関連する
fileType プロパティの値が *seqws* のとき、リソース関連パーツの
fileName プロパティと一致する値を指定する。リソース関連パーツの
systemName プロパティは、出力を受け取るオペレーティング・システム・ファイルの名前を含みます。
- *destination* の値として、値 *printer* は指定しない。これを指定した場合 (および *curses* ライブラリーを使用していない場合)、ユーザーに対して印刷プレビュー・ダイアログが表示されますが、今後のバージョンの EGL では振る舞いに変更される可能性があります。

segmentedMode

システム変数 **ConverseVar.segmentedMode** は、テキスト・アプリケーションで使用され、*converse* ステートメントの効果を変更します。ただし、呼び出し先プログラムでは、変数のこの目的は無視されます。背景情報については、『セグメンテーション』を参照してください。

ConverseVar.segmentedMode の値は以下のとおりです。

1 次の **converse** ステートメントは、セグメント化モードで実行します。

0 次の **converse** ステートメントは、非セグメント化モードで実行します。

デフォルト値は、非セグメント化プログラムの場合が 0、セグメント化プログラムの場合が 1 です。変数は、**converse** ステートメントの実行後にデフォルトに再設定されます。

この変数は、以下のいずれかの方法で使用できます。

- *assignment* ステートメントまたは *move* ステートメントのソースまたは宛先として
- **move...for count** ステートメントのカウント値として
- 論理式の比較値として
- *return* ステートメントの値として

ConverseVar.segmentedMode には以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後に値がリストアされるかどうか

いいえ

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

validationMsgNum

システム変数 **ConverseVar.validationMsgNum** には、テキスト・アプリケーションで **ConverseLib.validationFailed** によって割り当てられた値が含まれているため、検証機能によってエラーが報告されたかどうかを判断できます。この値は、以下の場合にゼロにリセットされます。

- プログラムが初期化された
- プログラムが `converse`、`display`、または `print` ステートメントを発行した
- プログラムが、検証エラーの結果としてテキスト書式を表示するため、`converse` ステートメントを再発行した

ConverseVar.validationMsgNum は以下の方法で使用できます。

- `assignment` ステートメントまたは `move` ステートメントのソースまたはターゲットとして使用 (`move` ステートメントの「`for count`」でも使用可能)
- 論理式の変数として使用
- `return` または `exit` 文内の引数として使用

ConverseVar.validationMsgNum には、以下のような特性があります。

プリミティブ型

INT

converse 後、値は常にリストアされますか

いいえ

例

```
/* 検証ルーチンの実行中に設定された最初のメッセージ番号を保存しておきます。
*/
if (ConverseVar.validationMsgNum > 0)
    ConverseLib.validationFailed(10);
end
```

関連する参照項目

616 ページの『`converse`』

849 ページの『`validationFailed()`』

619 ページの『`display`』

681 ページの『`print`』

985 ページの『EGL ライブラリー外部のシステム変数』

SysVar

修飾子 **SysVar** は、次の表にリストされた各 EGL システム変数の名前の前に付きます。これらの変数は、汎用に役立ちます。

システム変数	説明
arrayIndex	<p>以下の番号を含んでいます。</p> <ul style="list-style-type: none"> • in 演算子を使用した単一論理式の検索条件に一致する、配列内の最初のエレメントの番号 • 検索条件に一致する配列エレメントが存在しない場合には、ゼロ • move ... for count ステートメントの後のターゲット配列内で最後に変更されたエレメントの番号
callConversionTable	<p>プログラムが実行時に以下を行うときのデータ変換に使用される変換テーブルの名前を含んでいます。</p> <ul style="list-style-type: none"> • リモート・システム上のプログラムへの呼び出しで引数を受け渡す • リモート・プログラムをシステム関数 <code>sysLib.startTransaction</code> 経由で呼び出すときに引数を受け渡す • リモート・ロケーションにあるファイルにアクセスする
errorCode	<p>以下のいずれかが発生した場合に状況コードを受け取る。</p> <ul style="list-style-type: none"> • call ステートメントの呼び出し (その文が試行ブロックにある場合) • 索引ファイル、MQ ファイル、相対ファイル、またはシリアル・ファイルに対する入出力操作 • 次の場合における、ほとんどすべてのシステム関数の呼び出し <ul style="list-style-type: none"> – 呼び出しが試行ブロック内にある場合、または – プログラムが VisualAge Generator との互換性モードで実行中であり、VGVar.handleSysLibraryErrors が 1 に設定されている場合
formConversionTable	<p>EGL 生成 Java プログラムが以下のように動作する場合に双方向テキスト変換で使用される変換テーブルの名前を含んでいます。</p> <ul style="list-style-type: none"> • ヘブライ語またはアラビア語の文字を含むテキスト書式または印刷書式を表示する。 • ヘブライ語またはアラビア語の文字を受け入れるテキスト書式を表示する。
overflowIndicator	<p>算術オーバーフローが発生すると 1 に設定されます。この変数の値を検査することによって、オーバーフロー条件が発生したかどうかを確認できます。</p>

システム変数	説明
returnCode	プログラムによって設定され、オペレーティング・システムで使用可能な外部戻りコードを含んでいます。
sessionID	Web アプリケーション・サーバー・セッションに固有の ID を含んでいます。
sqlca	SQL 通信域 (SQLCA) 全体を含んでいます。
sqlcode	最後に完了した SQL 入出力操作の戻りコードを含んでいます。コードは、SQL 通信域 (SQLCA) から取得され、リレーショナル・データベース・マネージャーによって異なります。
sqlState	最後に完了した SQL 入出力操作の SQL 状態値を含んでいます。コードは、SQL 通信域 (SQLCA) から取得され、リレーショナル・データベース・マネージャーによって異なります。
systemType	プログラムを実行するターゲット・システムを識別します。
terminalID	Java 仮想マシン・システム・プロパティ <i>user.name</i> から初期化され、このプロパティを検索できない場合は、ブランクになります。
transactionID	トピック <i>transactionID</i> で説明されています。
transferName	実行時に、転送先のプログラムまたはトランザクションの名前の指定を許可します。
userID	ユーザー ID が使用可能な環境にユーザー ID を含みます。

関連する概念

62 ページの『EGL での変数の参照』

60 ページの『有効範囲指定の規則と EGL での「this」』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

arrayIndex

システム変数 **SysVar.arrayIndex** の数値は、以下のとおりです。

- **in** 演算子を使用した単純な論理式の検索条件に一致する、配列内の最初のエレメントの番号 (以下の例を参照)。
- 検索条件に一致する配列エレメントが存在しない場合には、ゼロ。
- **move ... for count** ステートメントの後に変更されたターゲット配列内の最終エレメントの番号

SysVar.arrayIndex は、以下のいずれとしても使用できます。

- 一致する行や配列エレメントにアクセスするための配列の添え字として
- `assignment` ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- **move ... for count** ステートメントのカウント値として使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SysVar.arrayIndex には、以下のような特性があります。

プリミティブ型

BIN

データ長

4

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例: レコード *myRecord* は以下のパーツを基にしていると想定します。

```
Record mySerialRecPart
  serialRecord:
    fileName = "myFile"
  end
  10 zipCodeArray CHAR(9)[100];
  10 cityStateArray CHAR(30)[100];
end
```

また、配列は郵便番号と市および県の組み合わせで初期化されていると想定します。

以下のコードは、変数 *currentCityState* を、指定された郵便番号に対応する市および県に設定します。

```
currentZipCode = "27540";
if (currentZipCode in myRecord.zipCodeArray)
  currentCityState = myRecord.cityStateArray[SysVar.arrayIndex];
end
```

if ステートメントの後、**SysVar.arrayIndex** には、値 "27540" を含む最初の *zipCodeArray* エレメントの指標が入ります。*zipCodeArray* で "27540" が検出されない場合、**SysVar.arrayIndex** の値は 0 になります。

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

78 ページの『配列』

575 ページの『in 演算子』

538 ページの『論理式』

985 ページの『EGL ライブラリー外部のシステム変数』

callConversionTable

システム変数 **SysVar.callConversionTable** には、プログラムが実行時に以下のことを行うときにデータ変換のために使用される、変換テーブルの名前が含まれます。

- リモート・システム上のプログラムへの呼び出しで引数を受け渡す
- リモート・プログラムをシステム関数 **SysLib.startTransaction** 経由で呼び出すときに引数を受け渡す
- リモート・ロケーションにあるファイルにアクセスする

変換が行われるのは、EBCDIC ベースのシステムと ASCII ベースのシステムとの間、または互いに異なるコード・ページを使用するシステムの間でデータが送受信されるときです。変換は、生成時に使用されるリンケージ・オプション・パーツが、**callLink** または **asynchLink** エlement内のプロパティ **conversionTable** の値に **PROGRAMCONTROLLED** を指定している場合にのみ可能です。ただし、**PROGRAMCONTROLLED** が指定されていても **SysVar.callConversionTable** がブランクである場合は、変換は行われません。

特性: **SysVar.callConversionTable** には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8

複数のセグメントにわたって値が保管されるかどうか

はい

定義に関する考慮事項: **SysVar.callConversionTable** は、プログラムの中で変換テーブルを切り替えるか、データ変換をオン/オフにするために使用します。

SysVar.callConversionTable の初期値はブランクです。変換が行われるようにするには、以前に説明したとおり、リンケージ・オプション・パーツに値 **PROGRAMCONTROLLED** が含まれていることを確認して、変換テーブルの名前をシステム変数に移動します。**SysVar.callConversionTable** をアスタリスク (*) に設定して、デフォルトの各国語コード用のデフォルトの変換テーブルを使用することができます。この設定は、ロケールが **targetNLS** ビルド記述子オプションに指定できる言語のいずれかにマップされているときは、ターゲット・システム上のデフォルトのロケールを参照します。

変換は、呼び出し、起動、ファイル・アクセスを発信するシステム上で行われます。1 つのレコード構造を複数のレベルで定義した場合、変換は最低レベルの項目 (副構造が存在しない項目) で行われます。

SysVar.callConversionTable は以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットのオペランドとして使用
- 論理式の変数として使用
- **return** または **exit** 文内の引数として使用。

SysVar.callConversionTable の別の値との比較は、突き合わせが完全に一致する場合にのみ真と判別されます。例えば、小文字を使用して **SysVar.callConversionTable** を初期化する場合、その小文字の値は小文字にのみ突き合わせされます。

SysVar.callConversionTable に配置する値は、比較目的のために未変更のまま残されます。上の特定の変換テーブルをシークするときに大文字変換されます。

例:

```
SysVar.callConversionTable = "ELACNENU";  
// US English COBOL 生成用の変換テーブル
```

関連する参照項目

507 ページの『データ変換』

975 ページの『startTransaction()』

985 ページの『EGL ライブラリー外部のシステム変数』

440 ページの『targetNLS』

errorCode

システム変数 **SysVar.errorCode** は、以下のいずれかが発生した場合に状況コードを受け取ります。

- call ステートメントの呼び出し (その文が試行ブロックにある場合)
- 索引ファイル、MQ ファイル、相対ファイル、またはシリアル・ファイルに対する入出力操作
- 次の場合における、ほとんどすべてのシステム関数の呼び出し
 - 呼び出しが試行ブロック内にある場合、または
 - プログラムが VisualAge Generator との互換性モードで実行されており、**VGVar.handleSysLibraryErrors** が 1 に設置されている場合

特定のシステム関数に関連付けられた **SysVar.errorCode** の値については、このトピックではなく該当するシステム関数に関する説明を参照してください。

SysVar.errorCode は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- 論理式の変数として使用
- 関数呼び出しで、in、out、または inOut パラメーターに関連付けされた引数として使用

call、I/O、またはシステム関数呼び出しが成功すると、**SysVar.errorCode** が 0 に設定されます。

SysVar.errorCode には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8

converse 後、値は常にリストアされますか

はい

SysVar.errorCode の詳細な概要については、『例外処理』を参照してください。考えられる **SysVar.errorCode** 値のリストが *EGL Java* ランタイム・エラー・コードに示されます。

例:

```
if (SysVar.errorCode == "00000008")
    exit program;
end
```

関連する参照項目

1027 ページの『EGL Java ランタイム・エラー・コード』

100 ページの『例外処理』

985 ページの『EGL ライブラリー外部のシステム変数』

698 ページの『try』

1013 ページの『handleSysLibraryErrors』

formConversionTable

システム変数 **SysVar.formConversionTable** には、EGL により生成された Java プログラムが以下のように動作する場合に、双方向テキスト変換で使用される変換テーブルの名前が含まれています。

- ヘブライ語またはアラビア語の文字を含むテキスト書式または印刷書式を表示する。
- ヘブライ語またはアラビア語の文字を受け入れるテキスト書式を表示する。

特性: **SysVar.formConversionTable** には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8

複数のセグメントにわたって値が保管されるかどうか

はい

関連する参照項目

510 ページの『双方向言語テキスト』

507 ページの『データ変換』

985 ページの『EGL ライブラリー外部のシステム変数』

overflowIndicator

システム変数 **SysVar.overflowIndicator** は、算術オーバーフローが発生したときに 1 に設定されます。この変数の値を検査することによって、オーバーフロー条件が発生したかどうかを確認できます。

オーバーフロー条件が検出された後で、**SysVar.overflowIndicator** は自動的にリセットされません。オーバーフロー検査が起動される可能性のある演算を実行する前に **SysVar.overflowIndicator** を 0 にリセットするコードを、プログラムに含めなければなりません。

SysVar.overflowIndicator は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SysVar.overflowIndicator には、以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

はい

例:

```
SysVar.overflowIndicator = 0;
VGVar.handleOverflow = 2;
a = b;
if (SysVar.overflowIndicator == 1)
  add errorrecord;
end
```

関連する参照項目

407 ページの『代入』

985 ページの『EGL ライブラリー外部のシステム変数』

1012 ページの『handleOverflow』

returnCode

システム変数 **SysVar.returnCode** には、プログラムによって設定され、オペレーティング・システムで使用可能な外部戻りコードが含まれています。1 つの EGL プログラムから別の EGL プログラムに戻りコードを渡すことはできません。ゼロ以外の戻りコードによって、例えば EGL が **onException** ブロックを実行することはありません。

SysVar.returnCode の初期値はゼロで、値は -2147483648 以上 2147483647 以下の範囲内にいることが必要です。

SysVar.returnCode は、メインのテキスト・プログラム (J2EE の外部で稼働します) またはメインのバッチ・プログラム (J2EE の外部または J2EE アプリケーション・クライアント内のいずれかで稼働します) に関してのみ意味があります。このコンテキストでの **SysVar.returnCode** は、プログラムを呼び出すコマンド・ファイルまたはコマンド実行用のコードを提供することを目的にしています。プログラムが、

プログラムが制御できないエラーで終了した場合、EGL ランタイムは **SysVar.returnValue** の設定を無視して、 値 693 を戻そうと試みます。

SysVar.returnValue は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SysVar.returnValue には、以下のような特性があります。

プリミティブ型

BIN

データ長

9

converse 後、値は常にリストアされますか

はい

例:

```
SysVar.returnValue = 6;
```

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

sessionID

Web アプリケーションでは、システム変数 **SysVar.sessionID** には、Web アプリケーション・サーバー・セッションに固有の ID が入ります。**SysVar.sessionID** 値は、プログラム間で共用されるファイルまたはデータベース情報にアクセスするためのキー値として使用できます。

Web アプリケーションの外部では、以下のことが該当します。

- システム変数 **SysVar.sessionID** には、プログラムで使用されるシステム依存のユーザー ID または端末 ID が入ります。
- **SysVar.sessionID** は、この目的で EGL より前の製品 (特に CSP 370AD バージョン 4 リリース 1 より前の CSP リリース) との互換性を確保するためにのみサポートされています。代わりに **SysVar.userID** または **SysVar.terminalID** を使用することをお勧めします。

SysVar.sessionID は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースとして使用
- 論理式の変数として使用
- **return** ステートメントの引数として使用

SysVar.sessionID には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8 (値が 8 文字より少ない場合はブランクが埋め込まれる)

converse 後、値は常にリストアされますか

はい

SysVar.sessionID は Java 仮想マシンの システム・プロパティ *user.name* から初期化されます。このプロパティを検索できない場合は、**SysVar.sessionID** はブランクになります。

例:

```
myItem = SysVar.sessionID;
```

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

1004 ページの『terminalID』

1005 ページの『userID』

sqlca

システム変数 **SysVar.sqlca** には、SQL 連絡域 (SQLCA) 全体が含まれます。後述するように、コードがリレーショナル・データベースにアクセスすると、SQLCA にあるフィールドのサブセットの現行値が使用可能になります。

SysVar.sqlca は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- 関数呼び出しで、in、out、または inOut パラメーターに関連付けされた引数として使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SQLCA 内の特定のフィールドを参照するためには、**SysVar.sqlca** を基底レコードに移動する必要があります。レコードは、データベース管理システムの SQLCA の説明で指定されているとおりの構造を持つ必要があります。SQLCA の内容をリモート・プログラムに渡す際には、基底レコードを使用し、SQLCA の内容がリモート・システムのデータ形式に正しく変換されるようにしてください。

SysVar.sqlca で使用可能なフィールドに関する固有の情報については、以下のトピックを参照してください。

- VGVVar.sqlerrd
- SysVar.sqlcode
- SysVar.sqlState
- VGVVar.sqlWarn

SysVar.sqlca には、以下のような特性があります。

プリミティブ型

HEX

データ長

272 (136 バイト)

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
myItem = SysVar.sqlca;
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

『sqlcode』

1002 ページの『sqlState』

1015 ページの『sqlerrd』

1017 ページの『sqlWarn』

sqlcode

システム変数 **SysVar.sqlcode** には、最新に完了した SQL 入出力操作の戻りコードが入ります。コードは、SQL 通信域 (SQLCA) から取得され、リレーショナル・データベース・マネージャーによって異なります。

SysVar.sqlcode は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- 関数呼び出しで、in、out、または inOut パラメーターに関連付けされた引数として使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SysVar.sqlcode には、以下のような特性があります。

プリミティブ型

BIN

データ長

9

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
rcitem = SysVar.sqlcode;
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

sqlState

システム変数 **SysVar.sqlState** には、最後に完了した SQL 入出力操作の SQL 状態値が含まれます。コードは、SQL 通信域 (SQLCA) から取得され、リレーショナル・データベース・マネージャーによって異なります。

SysVar.sqlState は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- 関数呼び出しで、パラメーター in、out、または inOut に関連付けられた引数として使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

SysVar.sqlState には、以下のような特性があります。

プリミティブ型

CHAR

データ長

5

converse 後、値は常にもリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
rcitem = SysVar.sqlState;
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

systemType

システム変数 **SysVar.systemType** は、プログラムが稼動しているターゲット・システムを識別します。生成された出力が Java ラッパーの場合、**SysVar.systemType** は使用できません。それ以外の場合は、有効な値は以下のとおりです。

aix AIX の場合

debug EGL デバッガーの場合

hp HP-UX の場合

iseriesj iSeries の場合

linux Linux (Intel ベースのハードウェア上) の場合

solaris Solaris の場合

win Windows 2000/NT/XP の場合

SysVar.systemType は以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースとして使用
- 論理式の変数として使用
- **return** ステートメントの引数として使用

SysVar.systemType には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8 (ブランクが埋め込まれる)

converse 後、値は常にリストアされますか

はい

VGLib.getVAGSysType でなく、**SysVar.systemType** を使用してください。

定義に関する考慮事項: **SysVar.systemType** の値は、生成時にどのコードが検証されるかに影響を与えることはありません。例えば、以下の **add** ステートメントは、Windows 用に生成を行う場合でも検証されます。

```
if (sysVar.systemType IS AIX)
  add myRecord;
end
```

ターゲット・システムで実行されないコードを検証しないようにするには、以下のいずれかの処置をとります。

- ビルド記述子オプション **EliminateSystemDependentCode** を YES に設定します。現在の例では、このビルド記述子オプションを YES に設定すると、**add** ステートメントは検証されません。ただし、生成プログラムがシステム依存コードを除去できるのは、論理式 (この場合は **SysVar.systemType IS AIX**) が単純で生成時に評価できる場合のみであることに注意してください。
- または、検証しない文を 2 番目のプログラムに移動し、元のプログラムが新規プログラムを以下のように条件付きで呼び出すようにします。

```
if (SysVar.systemType IS AIX)
  call myAddProgram myRecord;
end
```

例:

```
if (SysVar.systemType is WIN)
  call myAddProgram myRecord;
end
```

関連する参照項目

424 ページの『`eliminateSystemDependentCode`』
985 ページの『EGL ライブラリー外部のシステム変数』
983 ページの『`getVAGSysType()`』

terminalID

SysVar.terminalID は (**SysVar.sessionID** のように) Java 仮想マシンのシステム・プロパティー `user.name` から初期化されます。このプロパティーを検索できない場合は、**SysVar.terminalID** はブランクになります。

SysVar.terminalID は以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースとして使用
- 論理式の変数として使用
- **return** ステートメントの引数として使用

SysVar.terminalID には、以下のような特性があります。

プリミティブ型

CHAR

データ長

iSeries COBOL の場合は 8 で、値が最大文字数よりも小さい場合はブランクが挿入されます。

converse 後、値は常にリストアされますか

はい

例:

```
myItem10 = SysVar.terminalID;
```

transactionID

変数は使用されません。ただし、プログラムが *transfer to program* 書式の **transfer** ステートメントによって呼び出されている場合には、変数には移行元のプログラムの名前が含まれます。

この変数は、以下のいずれかの方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたは宛先として
- 論理式の比較値として
- **return** ステートメントの値として

SysVar.transactionID には以下のような特性があります。

プリミティブ型

CHAR

データ長

8

converse 後、値は常にリストアされますか

はい

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

transferName

システム変数 **SysVar.transferName** を使用することにより、実行時に、転送先のプログラムまたはトランザクションの名前を指定できます。

この変数は、以下のいずれかの方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- **transfer** ステートメントのプログラムまたはトランザクション名として
- 論理式の比較値として
- **return** ステートメントの値として

SysVar.transferName には以下のような特性があります。

プリミティブ型

CHAR

データ長

8

converse 後、値は常にリストアされますか

はい

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

697 ページの『transfer』

userID

システム変数 **SysVar.userID** には、ユーザー ID が入手できる場合にはユーザー ID が入ります。

SysVar.userID は以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースとして使用
- 論理式の変数として使用
- **return** ステートメントの引数として使用

SysVar.userID には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8 (値が 8 文字より少ない場合はブランクが埋め込まれる)

converse 後、値は常にリストアされますか

はい

SysVar.userID は Java 仮想マシンのシステム・プロパティー *user.name* から初期化されます。このプロパティーを検索できない場合は、**SysVar.userID** は空白になります。

例:

```
myItem = SysVar.userID;
```

VGVar

修飾子 **VGVar** は、次の表にリストされた各 EGL システム変数の名前の前に付きます。これらの変数は、VisualAge Generator からマイグレーションされたアプリケーションで主に役立ちます。

システム変数	説明
currentFormattedGregorianDate	現在のシステム日付を、長いグレゴリオ暦形式で格納します。
currentFormattedJulianDate	現在のシステム日付を、長いユリウス暦形式で格納します。
currentFormattedTime	現在のシステム時刻を HH:mm:ss 形式で格納します。
currentGregorianDate	現在のシステム日付を、8 桁のグレゴリオ暦形式 (yyyyMMdd) で格納します。
currentJulianDate	現在のシステム日付を、7 桁のユリウス暦形式 (yyyyDDD) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するものなので、なるべく使用しないでください。
currentShortGregorianDate	現在のシステム日付を、6 桁のグレゴリオ暦形式 (yyMMdd) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するものなので、なるべく使用しないでください。
currentShortJulianDate	現在のシステム日付を、5 桁のユリウス暦形式 (yyDDD) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するものなので、なるべく使用しないでください。
handleHardIOErrors	try ブロック内の入出力操作でハード・エラーが発生した後に、プログラムを継続して実行するかどうかを制御します。
handleOverflow	算術オーバーフロー後のエラー処理を制御します。
handleSysLibraryErrors	システム変数 >SysVar.errorCode の値がシステム関数の呼び出しの影響を受けるかどうかを指定します。

システム変数	説明
mqConditionCode	MQ レコードの add または scan 入出力操作の後に行われる MQSeries API 呼び出しの完了コードを含んでいます。
sqlerrd	6 エlementからなる配列であり、各Elementには、最後の SQL I/O オプションから戻された、対応する SQL 通信域 (SQLCA) 値が入ります。
sqlerrmc	SysVar.sqlcode の戻りコードに関連したエラー・メッセージの置換変数を含んでいます。
sqlIsolationLevel	データベース・トランザクション間の独立性レベルを示します。。
sqlWarn	最後の SQL 入出力操作に関して SQL 通信域 (SQLCA) で戻された警告バイトが各Elementに含まれ、インデックスが SQL SQLCA の説明内の警告番号よりも 1 大きい、11 個のElementからなる配列。

関連する概念

62 ページの『EGL での変数の参照』

60 ページの『有効範囲指定の規則と EGL での「this」』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

currentFormattedGregorianCalendar

システム変数 **VGVar.currentFormattedGregorianCalendar** は、長いグレゴリオ暦形式で現在のシステム日付を格納します。この値は、システム変数がプログラムで参照されるたびに自動的に更新されます。

Java ランタイム・プロパティに形式が指定されています。

```
vgj.datemask.gregorian.long.NLS
```

NLS

Java ランタイム・プロパティ **vgj.nls.code** に指定されている NLS (各国語サポート)。このコードは、**targetNLS** ビルド記述子オプションにリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティ (詳細)』を参照してください。

vgj.datemask.gregorian.long.NLS に指定される形式 には、dd (日数)、MM (月数)、yyyy (年数) のほか、区切り文字として使用される文字 (d、M、y、および数字を除く) を含んでいます。形式は、**dateMask** ビルド記述子オプション内に指定できます。デフォルトの形式は、ロケールごとに異なります。

VGVar.currentFormattedGregorianCalendar は、**assignment** または **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

この長いグレゴリオ暦の日付形式が、SQL データベース・マネージャーに対して指定された日付形式と同一であることを確認してください。この 2 つの形式が一致していれば、**VGVar.currentFormattedGregorianDate** を指定することにより、データベース・マネージャーが想定する形式で日付が生成されます。

VGVar.currentFormattedGregorianDate には、以下のような特性があります。

プリミティブ型

CHAR

データ長

10

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myDate = VGVar.currentFormattedGregorianDate;
```

関連する概念

319 ページの『ビルド記述子パーツ』

377 ページの『Java ランタイム・プロパティー』

関連するタスク

328 ページの『ビルド記述子での Java ランタイム・プロパティーの編集』

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

584 ページの『Java ランタイム・プロパティー (詳細)』

985 ページの『EGL ライブラリー外部のシステム変数』

440 ページの『targetNLS』

currentFormattedJulianDate

システム変数 **VGVar.currentFormattedJulianDate** は、長いユリウス日付形式で現在のシステム日付を格納します。この値は、システム変数がプログラムで参照されるたびに自動的に更新されます。

Java ランタイム・プロパティーに形式が指定されています。

```
vgj.datemask.julian.long.NLS
```

NLS

Java ランタイム・プロパティー **vgj.nls.code** に指定されている NLS (各国語サポート)。このコードは、**targetNLS** ビルド記述子オプションにリストされているコードの 1 つです。大文字の英語 (コード ENP) はサポートされていません。

vgj.nls.code の詳細については、『Java ランタイム・プロパティー (詳細)』を参照してください。

vgj.datemask.julian.long.NLS に指定される形式は、DDD (日数) と yyyy (年数) のほか、区切り文字として使用される文字 (D、y、および数字を除く) を含んでいます。形式は、**dateMask** ビルド記述子オプション内に指定できます。デフォルトの形式は、ロケールごとに異なります。

VGVar.currentFormattedJulianDate は、assignment ステートメントまたは **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

VGVar.currentFormattedJulianDate には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8

複数のセグメントにわたって値が保管されるかどうか

いいえ

大文字の英語 (NLS コード ENP) はサポートされていない。

例:

```
myDate = VGVar.currentFormattedJulianDate;
```

関連する概念

319 ページの『ビルド記述子パーツ』

377 ページの『Java ランタイム・プロパティー』

関連するタスク

328 ページの『ビルド記述子での Java ランタイム・プロパティーの編集』

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

584 ページの『Java ランタイム・プロパティー (詳細)』

985 ページの『EGL ライブラリー外部のシステム変数』

440 ページの『targetNLS』

currentFormattedTime

システム変数 **VGVar.currentFormattedTime** には、現在のシステム時刻が HH:mm:ss 形式で入ります。この値は、プログラムで参照されるたびに自動的に更新されます。

VGVar.currentFormattedTime は、以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースとして使用
- **exit** または **return** ステートメントの引数として使用

VGVar.currentFormattedTime には、以下のような特性があります。

プリミティブ型

CHAR

データ長

8

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
timeField = VGVar.currentFormattedTime;
```

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

985 ページの『EGL ライブラリー外部のシステム変数』

currentGregorianDate

システム変数 **VGVar.currentGregorianDate** は、現在のシステム日付を 8 桁のグレゴリオ暦形式 (yyyyMMdd) で格納します。

VGVar.currentGregorianDate の値は、それぞれの参照の前に自動的に更新されます。値は数値で、区切り文字は含まれません。

VGVar.currentGregorianDate は、**assignment** または **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

VGVar.currentGregorianDate には、以下のような特性があります。

プリミティブ型

DATE

データ長

8

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myDate = VGVar.currentGregorianDate
```

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

985 ページの『EGL ライブラリー外部のシステム変数』

currentJulianDate

システム変数 **VGVar.currentJulianDate** は、現在のシステム日付を 7 桁のユリウス日付形式 (yyyyDDD) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するものなので、なるべく使用しないでください。

この値は数値であり、区切り文字を含まず、参照されるたびに、前もって自動的に更新されます。

VGVar.currentJulianDate は、**assignment** または **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

VGVar.currentJulianDate には、以下のような特性があります。

プリミティブ型

NUM

データ長

7

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myDay = VGVar.currentJulianDate;
```

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

985 ページの『EGL ライブラリー外部のシステム変数』

currentShortGregorianDate

システム変数 **VGVar.currentShortGregorianDate** は、現在のシステム日付を 6 桁のグレゴリオ暦形式 (yyMMdd) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するもので、なるべく使用しないでください。

VGVar.currentShortGregorianDate 値は、プログラムで参照されるたびに自動的に更新されます。戻り値は数値で、区切り文字は含まれません。

VGVar.currentShortGregorianDate は、**assignment** または **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

VGVar.currentShortGregorianDate には、以下のような特性があります。

プリミティブ型

NUM

データ長

6

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myDay = VGVar.currentShortGregorianDate;
```

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

985 ページの『EGL ライブラリー外部のシステム変数』

currentShortJulianDate

システム変数 **VGVar.currentShortJulianDate** は、現在のシステム日付を 5 桁のユリウス日付形式 (yyDDD) で格納します。この変数は、VisualAge Generator から EGL へのコード・マイグレーションをサポートするために存在するもので、なるべく使用しないでください。

この値は数値であり、区切り文字を含まず、プログラムで参照されるたびに自動的に更新されます。

VGVar.currentShortJulianDate は、**assignment** または **move** 文内のソースとして、または、**return** または **exit** 文内の引数として使用できます。

VGVar.currentShortJulianDate には、以下のような特性があります。

プリミティブ型

NUM

データ長

5

複数のセグメントにわたって値が保管されるかどうか

いいえ

例:

```
myDay = VGVar.currentShortJulianDate;
```

関連する参照項目

849 ページの『EGL ライブラリー DateTimeLib』

985 ページの『EGL ライブラリー外部のシステム変数』

handleHardIOErrors

システム変数 **VGVar.handleHardIOErrors** は、try ブロック内の入出力操作でハード・エラーが発生した後に、プログラムを継続して実行するかどうかを制御します。デフォルト値は 1 です。ただし、プログラム・プロパティ **handleHardIOErrors** を *no* に設定した場合は除きます。その場合、この変数は 0 に設定されます。(このプロパティは、他の生成可能ロジック・パーツにも使用できます。) 背景情報については、『例外処理』を参照してください。

VGVar.handleHardIOErrors は、以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- **case**、**if**、または **while** ステートメントで使用される論理式での変数として使用
- **return** または **exit** 文内の引数として使用

VGVar.handleHardIOErrors には、以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

はい

例

```
VGVar.handleHardIOErrors = 1;
```

関連する参照項目

100 ページの『例外処理』

985 ページの『EGL ライブラリー外部のシステム変数』

handleOverflow

システム変数 **VGVar.handleOverflow** は、算術オーバーフロー後のエラー処理を制御します。以下の 2 種類のオーバーフロー条件が検出されます。

- ユーザー変数のオーバーフロー。算術演算または数値項目への代入の結果、項目の長さが原因で (小数点位でない) 有効値が失われたときに発生する。

- 最大値オーバーフロー。算術演算の結果が 18 桁を超えた場合に発生する。

VGVar.handleOverflow は以下のいずれかの値に設定できます。(デフォルトの設定は 0 です。)

値	ユーザー・オーバーフローへの影響	最大値オーバーフローへの影響
0	プログラムは、システム変数 SysVar.overflowIndicator を 1 に設定して処理を続行する	プログラムは終了し、エラー・メッセージが出される
1	プログラムは終了し、エラー・メッセージが出される	プログラムは終了し、エラー・メッセージが出される
2	プログラムは、システム変数 SysVar.overflowIndicator を 1 に設定して処理を続行する	プログラムは、システム変数 SysVar.overflowIndicator を 1 に設定して処理を続行する

VGVar.handleOverflow は以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.handleOverflow には、以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

はい

例:

```
VGVar.handleOverflow = 2;
```

関連する参照項目

407 ページの『代入』

985 ページの『EGL ライブラリー外部のシステム変数』

997 ページの『overflowIndicator』

handleSysLibraryErrors

システム変数 **VGVar.handleSysLibraryErrors** は、システム変数 **SysVar.errorCode** の値がシステム関数の呼び出しによって影響を受けるかどうかを指定します。ただし、**VGVar.handleSysLibraryErrors** を使用できるのは、VisualAge Generator 互換性が有効な場合のみです (『VisualAge Generator との互換性』を参照)。

詳細および制限事項については、『例外処理』を参照してください。

VGVar.handleSysLibraryErrors は、以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.handleSysLibraryErrors には、以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
VGVar.handleSysLibraryErrors = 1;
```

関連する概念

477 ページの『VisualAge Generator との互換性』

170 ページの『テキスト・アプリケーションのセグメンテーション』

関連する参照項目

100 ページの『例外処理』

985 ページの『EGL ライブラリー外部のシステム変数』

996 ページの『errorCode』

mqConditionCode

システム変数 **VGVar.mqConditionCode** は、MQ レコードの **add** または **scan** 入出力操作の後に行われる MQSeries API 呼び出しの完了コードを含んでいます。有効な値と関連の意味は以下のとおりです。

00 OK

01 WARNING

02 FAILED

VGVar.mqConditionCode は、以下の方法で使用できます。

- **assignment** ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用 (**move** ステートメントの「for count」でも使用可能)
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.mqConditionCode には、以下のような特性があります。

プリミティブ型

NUM

データ長

2

converse 後、値は常にリストアされますか

はい

例:

```
add MQRecord;
if (VGVar.mqConditionCode == 0)
  // 続行
else
  exit program;
end
```

関連する概念

285 ページの『MQSeries のサポート』

関連する参照項目

100 ページの『例外処理』

985 ページの『EGL ライブラリー外部のシステム変数』

sqlerrd

システム配列 **VGVar.sqlerrd** は 6 つの要素からなる配列で、それぞれの要素には、最後の SQL 入出力オプションから戻された、対応する SQL 通信域 (SQLCA) の値が入っています。例えば、**VGVar.sqlerrd[3]** の値は 3 番目の値であり、一部の SQL 要求用に処理された行数を示します。

VGVar.sqlerrd の要素のうち、**VGVar.sqlerrd[3]** のみが、データベース管理システムによって、Java コード用に、またはデバッグ時に更新されます。

VGVar.sqlerrd エレメントは、以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- **move** ステートメントの **for count** 文節の値として使用
- 関数呼び出しで、in、out、または inOut パラメーターに関連付けされた引数として使用
- 論理式の変数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.sqlerrd 配列の各要素の特性は、以下のとおりです。

プリミティブ型

BIN

データ長

9

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
myItem = VGVar.sqlerrd[3];
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

985 ページの『EGL ライブラリー外部のシステム変数』

sqlerrmc

システム変数 **VGVar.sqlerrmc** には、**SysVar.sqlcode** の戻りコードに関連したエラー・メッセージのが格納されます。**VGVar.sqlerrmc** は、SQL 通信域 (SQLCA) から取得され、リレーショナル・データベース・マネージャーによって異なります。

VGVar.sqlerrmc は、JDBC 環境では意味を持ちません。

VGVar.sqlerrmc は、以下の方法で使用できます。

- assignment ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- 論理式の変数として使用
- 関数呼び出しで、in、out、または inOut パラメーターに関連付けされた引数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.sqlerrmc には、以下のような特性があります。

プリミティブ型

CHAR

データ長

70

converse 後、値は常に戻アされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

例:

```
myItem = VGVar.sqlerrmc;
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

1000 ページの『sqlca』

985 ページの『EGL ライブラリー外部のシステム変数』

sqlIsolationLevel

システム変数 **VGVar.sqlIsolationLevel** は、データベース・トランザクション間の独立性のレベルを示します。

分離レベルと句 *repeatable read* と *serializable transaction* の概要については、Sun Microsystems, Inc の JDBC 資料を参照してください。

VGVar.sqlIsolationLevel は、VisualAge Generator および EGL 5.0 から移行されたプログラムのみで使用されます。関数は、(開発時に) EGL 設定 **VisualAge Generator との互換性** が選択されている場合、または (生成時に) ビルド記述子オプション **VAGCompatibility** が *yes* に設定されている場合にサポートされます。

新規開発の場合は、**SysLib.connect** に SQL 分離レベルを設定します。

以下の **VGVar.sqlIsolationLevel** の値は、厳密性の低い順になっています。

0 (デフォルト)

反復可能読み取り

1 順序付け可能トランザクション

この変数は、以下のいずれかの方法で使用できます。

- assignment ステートメントまたは move ステートメントのソースまたはターゲットとして使用
- 関数呼び出しで、パラメーター in、out、または inOut に関連付けられた引数として
- 論理式の比較値として
- return ステートメントの値として

SysVar.transactionID には以下のような特性があります。

プリミティブ型

NUM

データ長

1

converse 後、値は常にリストアされますか

はい

関連する参照項目

957 ページの『connect()』

985 ページの『EGL ライブラリー外部のシステム変数』

sqlWarn

システム配列 **VGVar.sqlWarn** は 11 エLEMENTの配列で、各ELEMENTは直前の SQL 入出力操作の SQL 通信域 (SQLCA) 内に戻された警告バイトを含み、インデックスは、SQL SQLCA 記述の警告番号より 1 だけ大きくなります。例えば、システム変数 **VGVar.sqlWarn[2]** は SQLWARN1 を参照し、これは項目内の文字に入出力操作で切り捨てが行われたかどうかを示します。

VGVar.sqlWarn のELEMENTのうち、システム変数 **VGVar.sqlWarn[2]** のみが、データベース管理システムによって、Java コード用に、またはデバッグ時に更新されます。

VGVar.sqlWarn は、以下の方法で使用できます。

- `assignment` ステートメントまたは **move** ステートメントのソースまたはターゲットとして使用
- **move** ステートメントの **for count** 文節の値として使用
- 論理式の変数として使用
- 関数呼び出しで、`in`、`out`、または `inOut` パラメーターに関連付けされた引数として使用
- **exit** または **return** ステートメントの引数として使用

VGVar.sqlWarn 配列の各エレメントの特性は、以下のとおりです。

プリミティブ型

CHAR

データ長

1

converse 後、値は常にリストアされますか

非セグメント化テキスト・プログラムでのみ。『セグメンテーション』を参照してください。

定義に関する考慮事項: 直前の SQL 入出力操作で、プログラムのホスト変数内に十分なスペースがないためにデータベース・マネージャーが文字データ項目に切り捨てを行った場合、**VGVar.sqlWarn[2]** には *W* が入ります。論理式を使用して、特定のホスト変数内の値が切り捨てされたかどうかをテストすることができます。詳しくは、『論理式』の『**trunc**』の説明を参照してください。

ホスト変数が数値である場合は、切り捨ての警告は出されません。数値の端数部分は、何の通知もなく切り捨てられます。

例: 以下の例では、*my-char-field* が、直前に処理された SQL 行レコード内のフィールドを表し、*lost-data* が、*my-char-field* の切り捨てに関する情報のエラー・メッセージを設定する関数を表します。

```
if (VGVar.sqlWarn[2] == 'W')
  if (my-char-field is trunc)
    lost-data();
  end
end
```

関連する概念

170 ページの『テキスト・アプリケーションのセグメンテーション』

247 ページの『SQL サポート』

関連する参照項目

538 ページの『論理式』

985 ページの『EGL ライブラリー外部のシステム変数』

transferToTransaction エレメント

リンケージ・オプション・パーツの *transferToTransaction* 要素では、生成されたプログラムがトランザクションおよび終了処理に制御を転送する方法を指定します。この要素には、プロパティー *toPgm* が含まれ、以下のプロパティーが含まれることもあります。

- *alias*。コードが、ランタイム名が関連プログラム・パーツ名とは異なるプログラムに転送している場合に必要です。
- *externallyDefined*。コードが、EGL または VisualAge Generator を使用して作成されていないプログラムに転送している場合に必要です。

ターゲット・プログラムが VisualAge Generator または (別名が存在しない場合) EGL で生成されている場合は、**transferToTransaction** 要素を指定する必要がありません。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

344 ページの『リンケージ・オプション・パーツのtransfer 関連エレメントの編集』

関連する参照項目

『transfer 関連リンケージ要素の別名』

1020 ページの『transferToTransaction エレメントの *externallyDefined*』

transfer 関連リンケージ要素の別名

リンケージ・オプション・パーツの *transfer* 関連要素では、プロパティー *alias* は、プロパティー *toPgm* で識別されるプログラムの実行時の名前を指定します。

このプロパティーの値は、転送先のプログラムの宣言時に指定した別名がある場合、その別名と同じにする必要があります。プログラム宣言時に別名を指定していない場合は、プロパティー *alias* にプログラム・パーツ名を設定するか、または何も設定しないようにします。

関連する概念

338 ページの『リンケージ・オプション・パーツ』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

344 ページの『リンケージ・オプション・パーツのtransfer 関連エレメントの編集』

関連する参照項目

『transferToTransaction エレメント』

transferToTransaction エLEMENTの externallyDefined

リンケージ・オプション・パーツの transferToTransaction 要素の **externallyDefined** プロパティは、EGL または VisualAge Generator 以外のソフトウェアによって生成されたプログラムに転送しようとしているかどうかを示します。有効な値は **NO** (デフォルト) と **YES** です。

yes と指定した場合には、XCTL はすべての COBOL ターゲット・システムに transfer ステートメントを実装します。

プログラム・プロパティ **VAGCompatibility** が **yes** に設定されている場合には、transfer ステートメントに externallyDefined を指定できます。『VisualAge Generator との互換性』の注を参照してください。代わりに transferToTransaction 要素に値を指定することをお勧めします。値は、どちらかの場所で指定されている場合に有効になります。

関連する概念

477 ページの『VisualAge Generator との互換性』

関連するタスク

340 ページの『EGL ビルド・ファイルへのリンケージ・オプション・パーツの追加』

344 ページの『リンケージ・オプション・パーツのtransfer 関連ELEMENTの編集』

関連する参照項目

697 ページの『transfer』

使用宣言

このセクションでは、まず使用宣言について説明し、その後で使用宣言の書き方についての詳細を説明します。

- 1021 ページの『プログラムまたはライブラリー・パーツ内』
- 1024 ページの『formGroup パーツ内』
- 1024 ページの『ページ・ハンドラーパーツ内』

背景

使用宣言を行うと、個別に生成される、パーツ内のデータ域および関数を簡単に参照することができます。例えば、プログラムで使用宣言を発行すると、データ・テーブル、ライブラリー、または書式グループを簡単に参照することができます。ただし、これは、そのようなパーツがプログラム・パーツに対して可視になっている場合だけです。可視性についての詳細は、『パーツの参照』を参照してください。

ほとんどの場合、使用宣言が行われている、いないに関係なく、データ域および機能は別のパーツから参照することができます。例えば、作成しているプログラムで myLib というライブラリー・パーツに対して使用宣言をしていなくても、次のようにすると myVar というライブラリー変数にアクセスできます。

```
myLib.myVar
```

しかし、使用宣言にライブラリー名を組み込んでいれば、この変数は次のように参照することができます。

`myVar`

前の短い書式の参照は、シンボル `myVar` がプログラムに対してグローバルな、すべての変数および構造体項目に対して一意である場合だけ有効になります。(シンボルが一意でない場合は、エラーが発生します。) また、シンボル `myVar` は、ローカル変数またはパラメーターが同じ名前を持たない場合だけ、ライブラリー内の項目を参照します。(ローカル・データ域は、同じ名前で、プログラムでグローバルになっているデータ域よりも優先されます。)

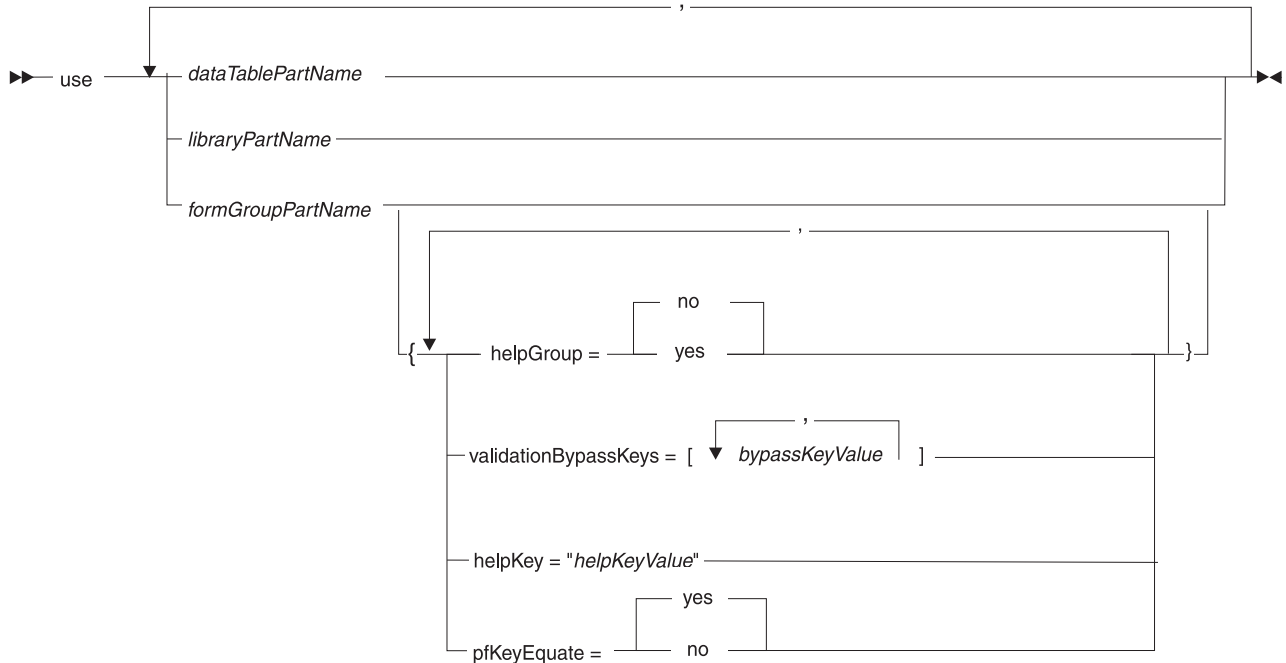
使用宣言が必要になるのは、以下の状態の場合です。

- ある与えられた `formGroup` パーツ内のいずれかの書式を使用するプログラムまたはライブラリーで、その `formGroup` パーツに対して使用宣言を行う必要がある。
- プログラムまたはライブラリーに必要ではあるけれども、`formGroup` パーツに組み込まれていない書式に対して、`formGroup` パーツで使用宣言を行う必要がある。
- 関数をコンテナ (プログラム、ページ・ハンドラー、またはライブラリー) の物理的な内部でなく、EGL ソース・ファイルのトップレベルで宣言した場合、その関数がライブラリー関数を呼び出すことができるのは、以下の状態の場合のみです。
 - コンテナに、ライブラリーを参照する `use` ステートメントが含まれている
 - 呼び出し側関数内で、`containerContextDependent` プロパティーが `yes` に設定されている

使用宣言で指定されている各名前は、パッケージ名、ライブラリー名、またはその両方で修飾することができます。

プログラムまたはライブラリー・パーツ内

プログラムまたはライブラリー内の各使用宣言は、すべての関数の外側になければなりません。使用宣言の構文は、以下のとおりです。



dataTablePartName

プログラムまたはライブラリーに対して可視になっている **dataTable** パーツ名。

プログラム・プロパティ **msgTablePrefix** で参照されている **dataTable** パーツに対しては、使用宣言内での参照は不要です。

使用宣言内では、**dataTable** パーツのプロパティをオーバーライドすることはできません。

dataTable パーツの概要については、『*DataTable* パーツ』を参照してください。

libraryPartName

プログラムまたはライブラリーに対して可視になっているライブラリー・パーツ名。

使用宣言内では、ライブラリー・パーツのプロパティをオーバーライドすることはできません。

ライブラリー・パーツの概要については、『*basicLibrary* 型のライブラリー・パーツ』および『*nativeLibrary* 型のライブラリー・パーツ』を参照してください。

formGroupName

プログラムまたはライブラリーに対して可視になっている **formGroup** パーツ名。書式グループの概要については、『*FormGroup* パーツ』を参照してください。

ある指定された `formGroup` パーツ内のいずれかの書式を使用するプログラムでは、その `formGroup` パーツに対して使用宣言を行う必要があります。

書式レベルのプロパティでは、オーバーライドは発生しません。例えば、**`validationBypassKeys`** のようなプロパティが書式内で指定された場合、その書式内の値は、実行時に有効になります。しかし、書式レベルのプロパティが書式で指定されない場合は、以下のような状態になります。

- EGL ランタイムでは、プログラムの使用宣言の値が使用されます。
- プログラムの使用宣言内に値が指定されていない場合は、ランタイムは書式グループ内の値を使用する (そのような値がある場合)。

その後に指定するプロパティで、特定のプログラムが書式グループにアクセスしたときの振る舞いを変更することができます。

`helpGroup = no, helpGroup = yes`

`formGroup` パーツをヘルプ・グループとして使用するかどうかを指定します。デフォルトは `no` です。

`validationBypassKeys = [bypassKeyValue]`

EGL ランタイムに入力フィールドの検証をスキップさせるユーザーのキー・ストロークを示します。このプロパティは、プログラムを即座に終了させるキー・ストロークを予約するときに便利です。各 `bypassKeyValue` オプションは以下のとおりです。

`pfn`

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

複数のキーを指定する場合、キーを 1 つずつコンマで区切ってください。

`helpKey = "helpKeyValue"`

EGL ランタイムにヘルプ書式をユーザーに表示させる、ユーザー・キー・ストロークを示します。`helpKeyValue` オプションは以下のとおりです。

`pfn`

F または PF キーの名前。1 から 24 の数を含みます (両端を含む)。

注: PC キーボードのファンクション・キーは、F1 などのように *F* キーの場合もありますが、EGL では IBM *PF* 用語を使用し、例えば、F1 は PF1 と呼ばれます。

`pfKeyEquate = yes, pfKeyEquate = no`

ユーザーが大きな番号のファンクション・キー (PF13 から PF24) を押したときに登録されるキー・ストロークが、ユーザーがそのキーよりも 12 だけ小さなファンクション・キーを押したときに登録されるキー・ストロークと同じであるかどうかを指定します。デフォルトは、`yes` です。詳細については、『*pfKeyEquate*』を参照してください。

formGroup パーツ内

formGroup パーツでは、使用宣言は書式グループの外側で指定された書式を参照します。この種の宣言を使用すると、複数の書式グループで同一の書式を共有することができます。

formGroup パーツにおける使用宣言の構文は、以下のとおりです。

```
▶▶ use formPartName ;
```

formPartName

書式グループに対して可視になっている書式の名前。書式の概要については、『Form パーツ』を参照してください。

formGroup パーツ内の使用宣言の書式パーツのプロパティは、オーバーライドできません。

ページ・ハンドラーパーツ内

ページ・ハンドラーパーツ内の各使用宣言は、すべての関数の外側になければなりません。使用宣言の構文は、以下のとおりです。

```
▶▶ use dataTablePartName ;
    libraryPartName
```

dataTablePartName

ページ・ハンドラーパーツに対して可視になっている dataTable パーツ名。

使用宣言内では、dataTable パーツのプロパティをオーバーライドすることはできません。

dataTable パーツの概要については、『DataTable パーツ』を参照してください。

libraryPartName

ページ・ハンドラーパーツに対して可視になっているライブラリー・パーツ名。

使用宣言内では、ライブラリー・パーツのプロパティをオーバーライドすることはできません。

ライブラリー・パーツの概要については、『Library パーツ』を参照してください。

関連する概念

155 ページの『DataTable』

163 ページの『FormGroup パーツ』

164 ページの『書式パーツ』

151 ページの『basicLibrary タイプのライブラリー・パーツ』

151 ページの『basicLibrary タイプのライブラリー・パーツ』

24 ページの『パーツの参照』

関連する参照項目

739 ページの『pfKeyEquate』

EGL Java ランタイム・エラー・コード

Java ランタイム時にエラーが発生すると、EGL は、システム変数 `sysVar.errorCode` にエラー・コードを入れ、大部分の場合、そのエラー・コードと同じ ID を持つメッセージを提示します。この EGL メッセージの代わりに、カスタマイズしたメッセージを表示することができます。詳細については、『*EGL Java ランタイムのメッセージ・カスタマイズ*』を参照してください。

エラー状況は次のとおりです

- リモート呼び出し中、EJB 呼び出し中、コミット中、またはロールバック中に失敗が起きました。この場合、メッセージ ID は CSO で始まります。
- Web アプリケーションでエラーが発生しました。この場合のサブセットでは、メッセージ ID は EGL で始まります。
- ローカル呼び出し中、ファイルまたはデータベースへのアクセス中、または以下のシステム関数の実行中にエラーが発生しました。
 - 数学関数
 - ストリング関数
 - `sysLib.convert`

この場合、メッセージ ID は VGJ で始まります。

- Java アクセス関数でエラーが発生しました。その場合、エラー・コードには数値だけが含まれ、メッセージは表示されません。

次の表に、Java アクセス関数で割り当てられたエラー・コードを示します。他のエラー・コードは、以下のセクションで示します。

sysVar.errorCode の値	説明
00001000	呼び出されたメソッドによって、またはクラスの初期化の結果として、例外がスローされました。
00001001	オブジェクトが NULL でした。または指定された ID がオブジェクト・スペース内にありませんでした。
00001002	指定した名前の public メソッド、フィールド、またはクラスが存在しないか、ロードできません。
00001003	EGL プリミティブ型が、Java で予期される型と一致しません。
00001004	メソッドが NULL を戻したか、メソッドが値を戻さないか、またはフィールドの値が NULL でした。
00001005	戻り値が戻り項目の型に一致しません。
00001006	NULL ヘキャストする引数のクラスをロードできませんでした。
00001007	メソッドまたはフィールドに関する情報取得の試行中に、 <code>SecurityException</code> または <code>IllegalAccessException</code> がスローされました。または、final 宣言されたフィールドの値を設定しようとする試みがなされました。
00001008	コンストラクターを呼び出すことができません。クラス名はインターフェースまたは抽象クラスを参照しています。

sysVar.errorCode の値	説明
00001009	クラス名ではなく、ID を指定する必要があります。メソッドまたはフィールドが静的ではありません。

関連する参照項目

579 ページの『入出力エラー値』

713 ページの『EGL Java ランタイム用のメッセージのカスタマイズ』

996 ページの『errorCode』

EGL Java ランタイム・エラー・コード CSO7000E

CSO7000E: 指定した呼び出し先プログラム %1 のエントリーがリンケージ・プロパティー・ファイル %2 にありません。

説明

このメッセージは以下の場合に発生します。

- 呼び出し側プログラムの生成時に、呼び出し先プログラムの **callLink** エレメントのリンケージ・オプション・パーツでプロパティー **remoteBind** が **RUNTIME** に設定された。また、
- 指定した呼び出し先プログラムのエントリーが、実行時にリンケージ・プロパティー・ファイルに見つからない。理由として以下のいずれかが考えられます。
 - リンケージ・プロパティー・ファイルが見つからない。
 - ファイルは見つかったが、そのファイルに呼び出し先プログラムのためのエントリーがない。
 - 誤ったリンケージ・プロパティー・ファイルが指定された。

ユーザー応答

以下のようにします。

- このプログラムを Java ラッパーから呼び出している場合は、リンケージ・プロパティー・ファイルの名前を **link.properties** にする必要があります。ここで、**link** は、生成時に使用したリンケージ・オプション・パーツの名前です。ファイルが存在し、呼び出し先プログラムのエントリーがあり、**CLASSPATH** 変数で指定されているディレクトリーまたはアーカイブにあることを確認してください。
- J2EE 環境で実行されているプログラムからプログラムを呼び出す場合は、リンケージ・プロパティー・ファイルをデプロイメント記述子の **cso.linkageOptions.link** 環境変数で指定できます。ここで、**link** は生成時に使用したリンケージ・オプション・パーツの名前です。環境変数が設定されていない場合は、リンケージ・プロパティー・ファイルの名前は **link.properties** でなければなりません。ここで、**link** は生成時に使用したリンケージ・オプション・パーツの名前です。ファイルが存在し、呼び出し先プログラムのエントリーがあり、**CLASSPATH** で指定されているディレクトリーまたはアーカイブにあることを確認してください。
- J2EE 環境で実行されていないプログラムからプログラムを呼び出す場合は、状況は以下のとおりです。

- リンケージ・プロパティ・ファイルを `cso.linkageOptions.link` プロパティで識別できます。ここで、`link` は生成時に使用したリンケージ・オプション・パーツの名前です。プロパティが設定されていない場合は、リンケージ・プロパティ・ファイルの名前は `link.properties` にすることができます。ここで、`link` は生成時に使用したリンケージ・オプション・パーツの名前です。これら 2 つのどちらの場合も、ファイルが存在し、呼び出し先プログラムのエントリーがあり、`CLASSPATH` で指定されているディレクトリまたはアーカイブにあることを確認してください。
- リンケージ・プロパティ・ファイルが見つからない場合は、リンケージ・プロパティがプログラム・プロパティ・ファイルになければなりません。その場合、プログラム・プロパティ・ファイルに呼び出し先プログラムのエントリーがあり、プログラム・プロパティ・ファイルが `CLASSPATH` で指定されたディレクトリまたはアーカイブにあることを確認してください。

その他の詳細については、**callLink** エレメント、Java ランタイム・プロパティ、環境のセットアップに関する EGL ヘルプ・ページを参照してください。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。

3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード CSO7015E

CSO7015E: リンケージ・プロパティ・ファイル %1 をオープンできません。

説明

ファイルがロックされているかファイルが見つからないため、リンケージ・プロパティ・ファイルをオープンできません。

ユーザー応答

リンケージ・プロパティ・ファイルが他のプロセスによってロックされておらず、ファイルが `CLASSPATH` で指定されたディレクトリまたはアーカイブに存在することを確認してください。

EGL Java ランタイム・エラー・コード CSO7016E

CSO7016E: プロパティ・ファイル `csouidpwd.properties` を読み取れませんでした。 エラー: %1

説明

ファイルは見つかりましたが、そのファイルの読み取り中にエラーが検出されました。

ユーザー応答

メッセージの「エラー」部分を使用して問題を診断および訂正してください。

EGL Java ランタイム・エラー・コード CSO7020E

CSO7020E: 変換テーブル %1 は無効です。

説明

双方向テキストを処理する変換テーブルが無効であるか、ロードできません。

ユーザー応答

変換テーブルが CLASSPATH で指定されたディレクトリまたはアーカイブに存在している必要があります。変換テーブルの開発方法の詳細については、双方向テキストに関するヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード CSO7021E

CSO7021E: 変換テーブル %1 のクライアント・テキスト属性タグ %2 が無効です。

説明

変換テーブル・ファイルは無効です。

ユーザー応答

ファイルを訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7022E

CSO7022E: 変換テーブル %1 のサーバー・テキスト属性タグ %2 が無効です。

説明

変換テーブル・ファイルは無効です。

ユーザー応答

ファイルを訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7023E

CSO7023E: 変換テーブル %1 の Arabic オプション・タグ %2 の値 %3 が無効です。

説明

変換テーブル・ファイルは無効です。

ユーザー応答

ファイルを訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7024E

CSO7024E: 変換テーブル %1 の Wordbreak オプション・タグ %2 の値 %3 が無効です。

説明

変換テーブル・ファイルは無効です。

ユーザー応答

ファイルを訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7026E

CSO7026E: 変換テーブル %1 の Roundtrip オプション・タグ %2 の値 %3 が無効です。

説明

変換テーブル・ファイルは無効です。

ユーザー応答

ファイルを訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7045E

CSO7045E: 共用ライブラリー %2 内のエントリー・ポイント %1 のアドレスを取得するときのエラー。RC = %3。

説明

共用ライブラリー内のエントリー・ポイントのアドレスを取得するときにエラーが発生しました。

ユーザー応答

参照している共用ライブラリーが、ロードする正しい共用ライブラリーであることを確認してください。正しい場合は、共用ライブラリーが正しくビルドされていることを確認してください。

EGL Java ランタイム・エラー・コード CSO7050E

CSO7050E: 日付 %2、時刻 %3 にリモート・プログラム %1 でエラーが発生しました。

説明

呼び出し先プログラムでエラーが発生し、プログラムが実行を停止しました。

ユーザー応答

このメッセージの日付と時刻のスタンプを使用し、このメッセージをリモート・ロケーションでログで記録された診断メッセージと対応付けてください。それらの診断メッセージを検査して詳細を調べてください。

EGL Java ランタイム・エラー・コード CSO7060E

CSO7060E: 共用ライブラリー %1 のロード中にエラーが発生しました。 戻りコードは %2 です。

説明

共用ライブラリーのロード中にエラーが発生しました。

ユーザー応答

PATH または LIBPATH 環境変数で指定されたディレクトリーに共用ライブラリーがあることを確認してください。共用ライブラリーが正しくビルドされていることを確認してください。

EGL Java ランタイム・エラー・コード CSO7080E

CSO7080E: 指定されたプロトコル %1 は無効です。

説明

リンケージで指定されたプロトコルを認識できません。

ユーザー応答

資料を参照して有効なプロトコルを指定してください。

EGL Java ランタイム・エラー・コード CSO7160E

CSO7160E: リモート・プログラム %1、日付 %2、時刻 %3、 システム %4 でエラーが発生しました。

説明

実行している Java プログラムが、指定されたシステムでリモート・プログラムを呼び出しますが、指定された日時に実行に失敗しました。

ユーザー応答

問題分析の詳細な説明については、リモート・サーバーのログを検査してください。

EGL Java ランタイム・エラー・コード CSO7161E

CSO7161E: プログラム %2 を呼び出そうと試みたときにシステム %1 でアプリケーション・エラーが発生したため、実行単位は終了しました。%3

説明

リモート・プログラムの実行中に、リモート・サーバーで、リモート実行単位を異常終了させるエラーが発生しました。このエラーの性質は、サーバーのジョブ・ログにおいて、このメッセージに先行する診断メッセージで説明されます。追加情報がある場合には、メッセージ・テキストに組み込まれることがあります。

ユーザー応答

サーバー・システムで記録されたエラー・メッセージを検査し、元の問題の修正方法を判別してください。

EGL Java ランタイム・エラー・コード CSO7162E

CSO7162E: システム %1 に接続するために提供された、無効なパスワードまたはユーザー ID。Java 例外メッセージを受け取りました: %2

説明

リモート・システムに接続するために提供されたパスワードまたはユーザー ID が設定されていないか無効です。

ユーザー応答

接続が設定されていることを検証してください。リモート・システムに提供されたユーザー ID およびパスワード正しいことを検証して再試行してください。

EGL Java ランタイム・エラー・コード CSO7163E

CSO7163E: システム %1、ユーザー %2 に対するリモート・アクセス・セキュリティ・エラー。Java 例外メッセージを受け取りました: %3

説明

現在システムに接続している指定されたユーザーに十分な権限がないか、指定されたシステムのリモート・リソースにアクセスできません。

ユーザー応答

リモート・マシンに接続しているユーザーに、リモート・マシンに接続する適切な権限およびリモート・サーバー・プログラムを実行する適切な権限があることを検証してください。

EGL Java ランタイム・エラー・コード CSO7164E

CSO7164E: システム %1 へのリモート接続エラー。Java 例外メッセージを受け取りました: %2

説明

リモート・システムとの通信中または接続中にエラーが発生しました。

ユーザー応答

リモート・サーバーが使用可能かどうかを検査し、再試行してください。これでもうまくいかない場合は、リモート・ホストのシステム管理者に連絡し、実際の問題を判別してください。

EGL Java ランタイム・エラー・コード CSO7165E

CSO7165E: システム %1 でコミットが失敗しました。%2

説明

リモート・システムでコミット操作が失敗しました。

ユーザー応答

ここでは %2 として示されている詳細なメッセージを検討して問題を診断してください。

EGL Java ランタイム・エラー・コード CSO7166E

CSO7166E: システム %1 でロールバックが失敗しました。%2

説明

リモート・システムでロールバック操作が失敗しました。

ユーザー応答

ここでは %2 として示されている詳細なメッセージを検討して問題を診断してください。

EGL Java ランタイム・エラー・コード CSO7360E

CSO7360E: システム %4 でプログラム %3 の呼び出し中の AS400Toolbox 実行エラー %1、%2。

説明

実行している Java プログラムまたはアプレットは、Java400 プロトコルを使用してリモート・サーバー・プログラムを呼び出します。サーバー・プログラムの呼び出しを試みているときに予期しない例外をキャッチしました。メッセージ・テキスト

は、AS400 Toolbox 例外の名前と、それに続く、例外とともに戻されたメッセージで成り立っています。

ユーザー応答

提供された AS400 Toolbox エラー・メッセージを使用して問題の原因を分析してください。

EGL Java ランタイム・エラー・コード CSO7361E

CSO7361E: EGL OS/400® ホスト・サービス・エラー。 必要なファイルがシステム %1 に見つかりません。

説明

実行している Java プログラムまたはアプレットは、Java400 プロトコルを使用してリモート・サーバー・プログラムを呼び出します。リモート・キャッチャーが見つからないか、サーバーの適切なライブラリーにないときに例外が発生しました。

ユーザー応答

リモート・システムに EGL OS/400 ホスト・サービスが適切にインストールされていることを検査してください。使用可能な場合は最新の PTF を適用してください。

EGL Java ランタイム・エラー・コード CSO7488E

CSO7488E: 不明な TCP/IP ホスト名: %1

説明

リモート TCP/IP リスナー・プログラムへの接続を試みているときに `UnknownHostException` がスローされました。

ユーザー応答

以下のようにします。

- プロパティー `cso.serverLinkage.xxx.location` をランタイム・リンケージ・プロパティー・ファイルに追加する。ここで、`xxx` は、リンケージ・プロパティー・ファイルに関する EGL 参照型のヘルプ・ページに説明されているように、呼び出し先プログラム名またはアプリケーション名です。プロパティーの値は有効な TCP/IP ホスト名です。
- 代わりに、以下のように生成時に TCP/IP ホスト名を設定し、プログラムを再生成することもできる。
 - リンケージ・オプション・パーツで、呼び出し先プログラムの `CallLink` エレメントのプロパティー **location** を TCP/IP ホスト名に設定する。
 - リンケージ・オプションを実行時にのみファイナライズ (決定) する場合は、プロパティー **remoteBind** を `RUNTIME` に設定し、ビルド記述子オプション **genProperties** を `YES` に設定して生成する。

その他の詳細については、CallLink エlement、リンケージ・プロパティ・ファイル、および環境のセットアップに関する EGL ヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード CSO7489E

CSO7489E: プログラムの呼び出しに使用するリンケージ情報が矛盾しているか欠落しています。

説明

プログラムが、プログラムの呼び出し方法を判別できませんでした。

ユーザー応答

必要なリンケージ情報をすべて提供してください。必要な情報は呼び出しの種類によって異なります。リンケージ・オプション・パーツに関する (特に CallLink Element に関する) ヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード CSO7610E

CSO7610E: CICS ECI を呼び出して作業単位をコミットするときにエラーが発生しました。CICS 戻りコードは %1 です。

説明

クライアントによってコミット要求が出されましたが、成功しませんでした。CICS 外部呼び出しインターフェースを呼び出して作業論理単位をコミットするときにエラーが発生しました。

ユーザー応答

指定されたエラーに対する修正アクションについては、該当する CICS の資料を参照してください。

EGL Java ランタイム・エラー・コード CSO7620E

CSO7620E: CICS ECI を呼び出して作業単位をロールバックするときにエラーが発生しました。CICS 戻りコードは %1 です。

説明

クライアントによってロールバック要求が出されましたが、成功しませんでした。CICS 外部呼び出しインターフェースを呼び出して作業論理単位をロールバックするときにエラーが発生しました。

ユーザー応答

指定されたエラーに対する修正アクションについては、該当する CICS の資料を参照してください。

EGL Java ランタイム・エラー・コード CSO7630E

CSO7630E: CICS サーバーに対するリモート・プロシージャ・コールを終了するときにエラーが発生しました。CICS 戻りコードは %1 です。

説明

CICS サーバーに対する EGL リモート・プロシージャ・コールを終了する前にオープンされている作業論理単位をすべてコミットしようとしたましたが、成功しませんでした。この要求は、CICS 外部呼び出しインターフェースを経由して行われました。

ユーザー応答

指定されたエラーに対する修正アクションについては、該当する CICS の資料を参照してください。

EGL Java ランタイム・エラー・コード CSO7640E

CSO7640E: %1 は ctgport エントリーに対して無効な値です。

説明

ctgport の値は整数でなければなりません。

ユーザー応答

正しい ctgport 番号を使用してください。

EGL Java ランタイム・エラー・コード CSO7650E

CSO7650E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。戻りコードは %2 です。CICS システム ID は %3 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7651E

CSO7651E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコードは -3 (ECI_ERR_NO_CICS) です。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -3 - ECI_ERR_NO_CICS

クライアントまたはサーバー・システムが使用できません

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7652E

CSO7652E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -4 (ECI_ERR_CICS_DIED)。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -4 - ECI_ERR_CICS_DIED

サーバー・システムが使用できなくなりました

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7653E

CSO7653E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -6 (ECI_ERR_RESPONSE_TIMEOUT)。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -6 - ECI_ERR_RESPONSE_TIMEOUT

応答タイムアウト。 時間制限が環境変数 CSOTIMEOUT で指定されています。

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7654E

CSO7654E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -7 (ECI_ERR_TRANSACTION_ABEND)。 CICS システム ID は %2 です。異常終了コードは: %3。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -7 - ECI_ERR_TRANSACTION_ABEND

サーバーでの異常終了。 共通の ABEND コードは以下のとおりです。

- AEI0: サーバー・プログラムが定義されていません
- AEI1: サーバー・トランザクションが定義されていません

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7655E

CSO7655E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -22 (ECI_ERR_UNKNOWN_SERVER)。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -22 - ECI_ERR_UNKNOWN_SERVER

サーバー・システムが定義されていません

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7656E

CSO7656E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -27 (ECI_ERR_SECURITY_ERROR)。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -27 - ECI_ERR_SECURITY_ERROR

ユーザー ID またはパスワードが無効です

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7657E

CSO7657E: CICS ECI を使用してプログラム %1 を呼び出すときにエラーが発生しました。 戻りコード: -28 (ECI_ERR_MAX_SYSTEMS)。 CICS システム ID は %2 です。

説明

リモート・サーバー・プログラムの呼び出しを試みていたときに、CICS 外部呼び出しインターフェース (ECI) 関数呼び出しからエラーが戻されました。

システム ID は、サーバー・プログラムを実行しようとしていた CICS システムの名前です。ブランクの場合は、プログラムの CICS プログラム定義または CICS クライアント初期化ファイルでシステムが指定されています。戻りコードは CICS の戻りコードです。

CICS 戻りコードには、以下の意味があります。

- -28 - ECI_ERR_MAX_SYSTEMS

サーバーが最大数に達しました

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明を参照する場合または戻りコードが上で説明されていない場合は、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7658E

CSO7658E: ユーザー %3 のシステム %2 でプログラム %1 を呼び出すときにエラーが発生しました。CICS ECI 呼び出しの戻り RC は %4、異常終了コードは %5 です。

説明

メッセージで示されたユーザーに代わってゲートウェイから指定されたシステムに行われた CICS ECI 呼び出しで、ゼロ以外の戻りコードが戻されました。

ユーザー応答

戻りコードによって示される問題を訂正してください。

戻りコードの詳細な説明については、使用しているシステムに応じた CICS ECI の資料を参照して修正アクションに関する情報を調べてください。

戻りコードの値は、CICS ECI インクルード・ファイル faaecih.h または cics_eci.h のシンボルに関連付けられています。

EGL Java ランタイム・エラー・コード CSO7659E

CSO7659E: CICS システム %1 への ECI 要求のフローで例外が発生しました。
例外: %2

説明

ゲートウェイからメッセージで識別された CICS システムに ECI 要求を送信しようと試みたときに、フロー・メソッドで予期しない例外が発生しました。

ユーザー応答

戻された例外ストリングを調べてください。問題の原因を例外から判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7669E

CSO7669E: CTG への接続中にエラーが発生しました。 CTG ロケーション: %1、CTG ポート: %2、例外: %3

説明

CICS Transaction Gateway への接続中に予期しない例外が発生しました。

ユーザー応答

戻された例外ストリングを調べてください。問題の原因を例外から判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7670E

CSO7670E: CTG からの切断中にエラーが発生しました。 CTG ロケーション: %1、CTG ポート: %2、例外: %3

説明

CICS Transaction Gateway からの切断中に予期しない例外が発生しました。

ユーザー応答

戻された例外ストリングを調べてください。問題の原因を例外から判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7671E

CSO7671E: CICSSSL プロトコルを使用する場合は、ctgKeyStore と ctgKeyStorePassword の両方を指定する必要があります。

説明

必要な値が指定されなかったため、呼び出しを完了できません。

ユーザー応答

ctgKeyStore と ctgKeyStorePassword の両方を指定するよう確認してください。

EGL Java ランタイム・エラー・コード CSO7816E

CSO7816E: ゲートウェイが、ユーザー ID %4 のためにホスト名 %1 およびポート %2 でサーバーに接続しようと試みたときにソケット例外が発生しました。例外は %3 でした。

説明

ゲートウェイからメッセージで識別されたサーバー・システムへのソケットの作成および接続のためのソケット呼び出しが、示された例外で失敗しました。

EGL ゲートウェイが、サーバー呼び出しのための TCP/IP ソケットの作成と接続を行うために、ソケット呼び出しを試行しました。ソケット呼び出しが、メッセージで示された例外で失敗しました。

ユーザー応答

例外情報を調べ、ゲートウェイからのソケット呼び出しが失敗した理由を判別してください。例外情報を調べても問題の原因を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7819E

CSO7819E: 関数 %2 で予期しない例外が発生しました。例外は %1 です。

説明

EGL ゲートウェイが、メッセージで識別された関数から予期しない例外を受け取りました。内部エラーが発生した可能性があります。

ユーザー応答

例外情報を調べても問題の原因を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7831E

CSO7831E: クライアントのバッファが、呼び出しで渡されたデータの量には小さすぎました。渡されるパラメーターの累計サイズが最大許容サイズの 32567 バイトを超えないことを確認してください。

説明

クライアントによって確立されたバッファは、リモートの呼び出し先プログラムに渡すパラメーターの累積サイズまで大きくすることはできません。

ユーザー応答

渡されるパラメーターの累計サイズが最大許容サイズの 32567 バイトを超えないことを確認してください。最大値を超えていないがこのエラーが発生する場合は、このエラーを IBM サポートに報告してください。

EGL Java ランタイム・エラー・コード CSO7836E

CSO7836E: クライアントは、サーバーがリモートの呼び出し先プログラムを開始できないという通知を受け取りました。理由コード: %1。

説明

サーバーがリモート呼び出し先プログラムを実行できず、問題判別のための理由コードを戻しました。

ユーザー応答

理由コードは以下のとおりです。

- 2: サーバーが呼び出し先プログラムのクラスをロードできませんでした。サーバー・トレース・ファイルが、より詳細な情報を示す場合があります。クラスがサーバーから使用可能なことを確認してください。

この問題の原因として、サーバーに渡されたクラス名が適切に変換されていないことが考えられます。データ変換に関するヘルプ・ページを検討し、プロパティ `conversionTable` の呼び出し先プログラムに対する `CallLink` エLEMENTのリンク・オプション・パーツに正しい変換テーブルが指定されていることを検証してください。

- 3: エラーのため、呼び出し先プログラムが終了しました。サーバー・トレース・ファイルが、より詳細な情報を示す場合があります。

上記に記載されていない理由コードの場合、または失敗の原因を判別できない場合は、IBM サポートに連絡してください。

EGL Java ランタイム・エラー・コード CSO7840E

CSO7840E: クライアントは、リモートの呼び出し先プログラムに戻りコード %1 で失敗したという通知をサーバーから受け取りました。

説明

リモートの呼び出し先プログラムは実行されましたが、ゼロ以外の戻りコードで終了しました。問題は通信ではなく、プログラムに存在します。

ユーザー応答

呼び出し先プログラムを検査またはトレースし、ゼロ以外の戻りコードで完了した理由を調べてください。

EGL Java ランタイム・エラー・コード CSO7885E

CSO7885E: TCP/IP 読み取り関数が、ユーザー ID %2、ホスト名 %1 の呼び出しで失敗しました。戻された例外: %3

説明

TCP/IP 読み取り関数を試行するときに EGL ゲートウェイが例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。失敗した理由を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7886E

CSO7886E: TCP/IP 書き込み関数が、ユーザー ID %2、ホスト名 %1 の呼び出しで失敗しました。 戻された例外: %3

説明

TCP/IP 書き込み関数を試行するときに EGL ゲートウェイが例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。失敗した理由を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO7955E

CSO7955E: %1、%2

説明

予期しない Java 例外をキャッチしました。

メッセージ・テキストには、Java 例外名の後に例外でスローされた Java メッセージが示されます。

ユーザー応答

メッセージを検討し、適切に対処してください。

EGL Java ランタイム・エラー・コード CSO7957E

CSO7957E: 変換テーブル名 %1 は Java データ変換に対して無効です。

説明

生成された Java クラスを使用してプログラムを呼び出している場合に、呼び出し先プログラムが使用する形式に Java データを変換するための変換テーブルを誤って指定しました。

ユーザー応答

データ変換に関するヘルプ・ページを検討し、プロパティ `conversionTable` の呼び出し先プログラムに対する `CallLink` エレメントのリンケージ・オプション・パーツに指定する変換テーブル名を判別してください。

EGL Java ランタイム・エラー・コード CSO7958E

CSO7958E: ネイティブ・コードは CSOPowerServer という型のオブジェクトを Java ラッパーに提供していませんが、このオブジェクトは、Java ラッパーと EGL 生成プログラムとの間でデータを変換するときが必要です。

説明

ネイティブ Java コードが、最初にクラス CSOPowerServer のオブジェクトをインスタンス化してそのオブジェクトをラッパーに提供することなく、Java ラッパーの call または execute メソッドを呼び出しました。

ユーザー応答

EGL ミドルウェアにアクセスするときの詳細について、Java ラッパーに関するヘルプ・ページを検討してください (これはデータ変換のときに常に要求されます)。

EGL Java ランタイム・エラー・コード CSO7966E

CSO7966E: 変換テーブル %2 にコード・ページ・エンコード %1 はありません。

説明

リンケージ・オプションで指定した変換テーブルが、使用する Java 仮想マシン (JVM) で使用できないエンコードを必要としています。

ユーザー応答

データ変換に関するヘルプ・ページを検討し、プロパティ conversionTable の呼び出し先プログラムに対する CallLink エlementのリンケージ・オプション・パーツに指定する正しい変換テーブル名を判別してください。正しい変換テーブル名を指定した場合は、使用している JVM が EGL の Java ランタイム環境によってサポートされていることを確認してください。

前記の手順で問題が判明しない場合は、インストールした JVM が壊れているか、使用する Java 仮想マシンがすべてのエンコードをサポートしていない可能性があります。これらの場合は、JVM ベンダーの資料を参照するか、JVM ベンダーに連絡して支援を求めてください。

ブラウザーでのアプレット・クライアントの実行中にエラーが発生した場合は、エラーは、クライアント・アプレットによって使用された PowerServer SessionManager で発生しています。この場合は、SessionManager を実行している JVM の資料を参照するか、JVM ベンダーに連絡してください。

EGL Java ランタイム・エラー・コード CSO7968E

CSO7968E: ホスト %1 が認識されていないか見つかりません。

説明

リンケージでリモート・システムが指定されていません。

ユーザー応答

リンケージ・パーツのロケーション・フィールドでリモート・システムを指定する必要があります。

EGL Java ランタイム・エラー・コード CSO7970E

CSO7970E: 必要な EGL 共用ライブラリー %1 をロードできませんでした。理由: %2

説明

操作を完了するには共用ライブラリーが必要ですが、ロードできませんでした。

ユーザー応答

共用ライブラリーがシステムにあることを確認してください。共用ライブラリー・パス PATH または LIBPATH を指定する環境変数に含める必要があります。

EGL Java ランタイム・エラー・コード CSO7975E

CSO7975E: プロパティー・ファイル %1 をオープンできませんでした。

説明

プログラムで必要なプロパティー・ファイルをオープンできませんでした。プロパティー・ファイルの名前は、プログラムの開始時にコマンド行で指定できます。プログラムの開始時に名前を指定しない場合は、デフォルトで次の名前が使用されます。

```
tcpiplistener.properties
```

プロパティー・ファイルが存在しないか、存在していてもオープンできませんでした。

ユーザー応答

プロパティー・ファイルが存在し、プログラムがそれを読み取るための適切なアクセス権を持っていることを確認し、再度プログラムを実行してください。

EGL Java ランタイム・エラー・コード CSO7976E

CSO7976E: トレース・ファイル %1 をオープンできませんでした。例外は %2 です。メッセージは次のとおりです: %3

説明

プログラムがトレース出力ファイルをオープンしようと試みたときに例外が発生しました。

ユーザー応答

問題を訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7977E

CSO7977E: プログラムのプロパティ・ファイルには、%1 プロパティに有効な設定が入っていませんが、これは必要です。

説明

プロパティがプログラム・プロパティ・ファイルで定義されていません。

ユーザー応答

プロパティをプログラム・プロパティ・ファイルに追加し、プログラムを再実行してください。詳細については、Java ランタイム・プロパティに関するヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード CSO7978E

CSO7978E: 予期しない例外が発生しました。 例外は %1 です。メッセージは次のとおりです: %2

説明

プログラムでエラーが発生しました。

ユーザー応答

問題を訂正してプログラムを再実行してください。

EGL Java ランタイム・エラー・コード CSO7979E

CSO7979E: InitialContext を作成できません。 例外は %1 です。

説明

javax.naming.InitialContext のコンストラクターから例外がスローされました。プログラムが J2EE 環境設定にアクセスするには、InitialContext オブジェクトを作成する必要があります。

ユーザー応答

例外のテキストおよび J2EE 環境の資料を使用して、問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8000E

CSO8000E: ゲートウェイに入力されたパスワードが失効しています。 %1

説明

EGL GatewayServlet が、提供されたパスワードでユーザーを認証しようと試みたときに、パスワード失効例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。新規パスワードを提供して問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8001E

CSO8001E: ゲートウェイに入力されたパスワードが無効です。%1

説明

EGL GatewayServlet が、提供されたパスワードでユーザーを認証しようと試みたときに、無効なパスワード例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。新規パスワードを提供して問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8002E

CSO8002E: ゲートウェイに入力されたユーザー ID が無効です。%1

説明

EGL GatewayServlet が、提供されたユーザー ID でユーザーを認証しようと試みたときに、無効なユーザー ID 例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。新規ユーザー ID を提供して問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8003E

CSO8003E: %1 に NULL エントリーがあります

説明

NULL エントリーを検出しました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。必要なエントリーを提供して問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8004E

CSO8004E: ゲートウェイが不明なセキュリティー・エラーを受け取りました。

説明

EGL GatewayServlet が、提供されたユーザー情報でユーザーを認証しようと試みたときに、不明なセキュリティ例外を受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。新しいユーザー情報を提供して問題を訂正してください。失敗した理由を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO8005E

CSO8005E: パスワードの変更中にエラーが発生しました。 %1

説明

EGL GatewayServlet が、提供されたパスワードを変更しようと試みたときにエラーを受け取りました。

ユーザー応答

戻された例外情報を調べ、問題の原因を判別してください。新規パスワードを提供して問題を訂正してください。失敗した理由を判別できない場合は、IBM サポートに連絡して援助を受けてください。

EGL Java ランタイム・エラー・コード CSO8100E

CSO8100E: 接続ファクトリーを取得できません。 例外は %1 です。

説明

remoteComType プロパティの値が CICSJ2C の場合に呼び出しで使用する接続ファクトリーのルックアップ時に例外がスローされました。 remoteComType プロパティは、呼び出し先プログラムに対する CallLink エレメントのリンケージ・オプション・パーツにあります。

接続ファクトリーの名前は、java:comp/env/ の後に、 同じ CallLink エレメントの location プロパティで設定した値を続けたものです。

ユーザー応答

接続ファクトリーが J2EE 環境で適切に定義されており、location プロパティの値が呼び出し先プログラムに対する CallLink エレメントで正しく設定されていることを確認してください。

EGL Java ランタイム・エラー・コード CSO8101E

CSO8101E: 接続を取得できません。 例外: %1

説明

remoteComType プロパティの値が CICSJ2C の場合に呼び出しに使用された ConnectionFactory オブジェクトの getConnection メソッドによって例外がスローされました。 remoteComType プロパティは、呼び出し先プログラムに対する CallLink エlementのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を参照し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8102E

CSO8102E: Interaction を取得できません。 例外: %1

説明

remoteComType プロパティの値が CICSJ2C の場合に呼び出しに使用される Connection オブジェクトの createInteraction メソッドによって例外がスローされました。 remoteComType プロパティは、呼び出し先プログラムに対する CallLink エlementのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8103E

CSO8103E: 対話動詞を設定できません。 例外は %1 です。

説明

remoteComType プロパティの値が CICSJ2C の場合に呼び出しに使用される ECIIInteractionSpec オブジェクトの setInteractionVerb メソッドによって例外がスローされました。 remoteComType プロパティは、呼び出し先プログラムに対する CallLink エlementのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8104E

CSO8104E: CICS との通信の試行中にエラーが発生しました。 例外は %1 です。

説明

`remoteComType` プロパティの値が `CICSJ2C` の場合に呼び出しに使用される `Interaction` オブジェクトの `execute` メソッドによって例外がスローされました。
`remoteComType` プロパティは、呼び出し先プログラムに対する `CallLink` エレメントのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。追加情報がゲートウェイ・ログまたはリモート・システムのログ・ファイルにある場合があります。

EGL Java ランタイム・エラー・コード CSO8105E

CSO8105E: `Interaction` または `Connection` を終了できません。例外は %1 です。

説明

`remoteComType` プロパティの値が `CICSJ2C` の場合に呼び出しに使用される `Connection` または `Interaction` オブジェクトの `close` メソッドによって例外がスローされました。
`remoteComType` プロパティは、呼び出し先プログラムに対する `CallLink` エレメントのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8106E

CSO8106E: クライアント作業単位の `LocalTransaction` を取得できません。例外は %1 です。

説明

以下の状況で呼び出しに使用される `Connection` オブジェクトの `getLocalTransaction` メソッドによって例外がスローされました。

- `remoteComType` プロパティの値が `CICSJ2C` である
- `luwControl` プロパティの値が `CLIENT` である

これらのプロパティは、呼び出し先プログラムに対する `CallLink` エレメントのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8107E

CSO8107E: CICSJ2C 呼び出しのタイムアウト値を設定できません。例外は %1 です。

説明

remoteComType プロパティの値が CICSJ2C の場合に呼び出しに使用される ECIInteractionSpec オブジェクトの setExecuteTimeout メソッドによって例外がスローされました。remoteComType プロパティは、呼び出し先プログラムに対する CallLink エレメントのリンケージ・オプション・パーツにあります。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。

EGL Java ランタイム・エラー・コード CSO8108E

CSO8108E: CICS との通信の試行中にエラーが発生しました。

説明

呼び出しに使用される Interaction オブジェクトの execute メソッドが false を返しました。呼び出しは正常に完了しませんでした。

ユーザー応答

接続ファクトリーまたはリソース・アダプターが定義されていないか、適切に構成されていない可能性があります。例外のテキスト、リソース・アダプターの資料、および J2EE 環境の資料を使用し、問題を診断してください。追加情報がゲートウェイ・ログまたはリモート・システムのログ・ファイルにある場合があります。

EGL Java ランタイム・エラー・コード CSO8109E

CSO8109E: タイムアウト値 %1 は無効です。これは数値でなければなりません。

説明

タイムアウトに無効な値が指定されました。

ユーザー応答

タイムアウト値を指定しないようにするか、数値を指定してください。

EGL Java ランタイム・エラー・コード CSO8110E

CSO8110E: 少なくとも 1 つのパラメーターが動的配列であるため、parmForm リンケージ・プロパティを COMMPTR に設定してプログラム %1 を呼び出す必要があります。

説明

パラメーターのうち 1 つが動的配列であるため、parmForm を COMMPTR にする必要があります。

ユーザー応答

parmForm を COMMPTR に変更してください。

EGL Java ランタイム・エラー・コード CSO8180E

CSO8180E: リンケージが J2EE サーバー内で DEBUG 呼び出しを指定しました。呼び出しが J2EE サーバーで行われなかったか、J2EE サーバーがデバッグ・モードでないか、または J2EE サーバーが EGL デバッグに対して使用可能になっていませんでした。

説明

DEBUG 呼び出しを完了できません。

ユーザー応答

呼び出しを J2EE サーバーで行わない場合は、EGL デバッガーを実行しているマシンの TCP/IP ホスト名をリンケージのロケーション・フィールドで指定する必要があります。呼び出しを J2EE サーバーで行う場合は、J2EE サーバーをデバッグ・モードで開始していること、および EGL デバッガーの jar ファイルを追加していることを確認してください。

EGL Java ランタイム・エラー・コード CSO8181E

CSO8181E: ホスト名 %1、ポート %2 の EGL デバッガーに連絡を取ることができません。例外は %3 です

説明

EGL デバッガーに連絡を取ることができなかったため、DEBUG 呼び出しを完了できません。

ユーザー応答

指定されたホスト名およびポートの EGL デバッガーで EGL リスナーが実行されていることを確認してください。

EGL Java ランタイム・エラー・コード CSO8182E

CSO8182E: ホスト名 %1、ポート %2 の EGL デバッガーとの通信中にエラーが発生しました。例外は %3 です

説明

EGL デバッガーと呼び出し側プログラムの通信が失敗しました。

ユーザー応答

例外メッセージの情報をを使用して問題を訂正してください。

EGL Java ランタイム・エラー・コード CSO8200E

CSO8200E: 最大サイズを超えて配列ラッパー %1 を展開することはできません。
メソッド %2 でエラーが発生しました。

説明

配列の最大サイズを超えました。

ユーザー応答

配列への追加を試みる前に、配列のサイズおよび最大サイズを検査してください。

EGL Java ランタイム・エラー・コード CSO8201E

CSO8201E: %1 は配列ラッパー %2 に対して無効なインデックスです。最大サイズ: %3。現行サイズ: %4

説明

インデックスが配列の境界を超えています。

ユーザー応答

有効なインデックスを使用してください。

EGL Java ランタイム・エラー・コード CSO8202E

CSO8202E: %1 は配列ラッパー %2 の有効な最大サイズではありません。

説明

プロパティ maxSize は 0 以上でなければなりません。

ユーザー応答

プロパティ maxSize を負の数値に設定しないでください。

EGL Java ランタイム・エラー・コード CSO8203E

CSO8203E: %1 は、タイプ %2 の配列ラッパーへの追加が無効なオブジェクト型です。

説明

配列の内容は定義に一致する必要があります。

ユーザー応答

配列が保管するオブジェクトのタイプを変更するか、このタイプのオブジェクトを配列に保管するよう試みないでください。

EGL Java ランタイム・エラー・コード CSO8204E

CSO8204E: パラメーターとして Any、Dictionary、ArrayDictionary、Blob、Clob、または Ref 変数を渡すことはできません。

説明

リストされたタイプは、call 文でパラメーターとして使用することはできません。さらに、リストされたタイプを含むタイプもパラメーターとして使用できません。

ユーザー応答

この種類のパラメーターを呼び出し先プログラムに渡さないでください。

EGL Java ランタイム・エラー・コード EGL0650E

EGL0650E: %1RequestAttr 関数がキー %2 で失敗しました。エラー: %3

説明

示されたキーで呼び出されたときに EGL GetRequestAttr 関数または SetRequestAttr 関数が失敗しました。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。この関数は、必ずページ・ハンドラー関数内で使用してください。

EGL Java ランタイム・エラー・コード EGL0651E

EGL0651E: %1SessionAttr 関数がキー %2 で失敗しました。エラー: %3

説明

示されたキーで呼び出されたときに EGL GetSessionAttr 関数または SetSessionAttr 関数が失敗しました。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。この関数は、必ずページ・ハンドラー関数内で呼び出してください。

EGL Java ランタイム・エラー・コード EGL0652E

EGL0652E: ラベル %1 の forward ステートメントが失敗しました。エラー: %2

説明

示されたラベルに制御を渡せませんでした。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。
このラベルに関連した EGL オブジェクトが正しく生成されていること、およびラベルがアプリケーション構成ファイルで定義されていることを確認してください。

EGL Java ランタイム・エラー・コード EGL0653E

EGL0653E: EGL オブジェクト %1 からの Bean の作成が失敗しました。エラー: %2

説明

EGL レコードまたはページ・ハンドラー定義からアクセス Bean を作成できませんでした。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。

EGL Java ランタイム・エラー・コード EGL0654E

EGL0654E: SetError 関数が項目 %1、キー %2 で失敗しました。エラー: %3

説明

示されたメッセージ・キーで呼び出されたときに SetError 関数が失敗しました。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。
項目のエラー・エントリーが JSP にあること、およびキーがメッセージ・リソース・ファイルで定義されていることを確認してください。

EGL Java ランタイム・エラー・コード EGL0655E

EGL0655E: Bean から EGL レコード %1 へのデータのコピーが失敗しました。
エラー: %2

説明

書式 Bean からレコードへのデータの移動の試行が失敗しました。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。
Bean 定義がレコード定義と一致していることを確認してください。

EGL Java ランタイム・エラー・コード EGL0656E

EGL0656E: サイズ %1 の配列をサイズ %2 の静的配列に割り当てられません。

説明

配列のサイズが一致している必要があります。

ユーザー応答

EGL 配列定義をチェックし、配列サイズが同じであることを確認してください。

EGL Java ランタイム・エラー・コード EGL0657E

EGL0657E: onPageLoad パラメーターの処理が失敗しました。 エラー: %1。

説明

EGL が onPageLoad 関数のパラメーターの値を受け取ろうとしたときにエラーが発生しました。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。渡す値の型定義が onPageLoad 関数のパラメーターの型定義に一致していることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0001E

VGJ0001E: %1 からの最大値オーバーフローが発生しました。

説明

算術計算時に、値を 0 で割ったか中間結果の有効数字が 18 桁を超えました。システム変数 `VGVar.handleOverflow` が 2 に設定されている場合を除いて、プログラムが終了します。

ユーザー応答

以下のアクションを 1 つ以上実行してください。

- エラーを回避するようにプログラムのロジックを訂正する。
- オーバーフロー条件を処理するためのプログラム・ロジックを定義する。システム変数 `VGVar.handleOverflow` と `overflowIndicator` を使用します。

EGL Java ランタイム・エラー・コード VGJ0002E

VGJ0002E: エラー %1 が発生しました。 このエラーのメッセージ・テキストはメッセージ・ファイル %2 にありません。

説明

メッセージ・ファイルが壊れているか、 EGL の旧リリースのメッセージ・ファイルからのものである可能性があります。

ユーザー応答

以下のいずれかの指示を実行します。

- ファイル `fda6.jar` からクラス・ファイルを抽出した場合は、使用しているクラスのリリース・レベルと保守レベルが、そのファイルのクラスと同じであることを確認する。一致しない場合は、古いクラスを正しいバージョンで置き換えてください。
- EGL から `fda6.jar` を再インストールする。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。

3. IBM サポートに問題を報告する方法の詳細な指示について、製品のインストール資料を参照する。

EGL Java ランタイム・エラー・コード VGJ0003E

VGJ0003E: ロケーション %1 で内部エラーが発生しました。

説明

このエラーは、システムの制約または要件が満たされていない場合、または EGL プログラム・パーツが適切に使用されていない場合のみに発生します。エラー内で指定されるロケーションは、IBM が診断目的でのみ使用します。

ユーザー応答

プログラムのセットアップを検査し、システムを再始動してください。問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。

3. IBM サポートに問題を報告する方法の詳細な指示について、製品のインストール資料を参照する。

EGL Java ランタイム・エラー・コード VGJ0004I

VGJ0004I: %1 の関数 %2 でエラーが発生しました。

説明

このエラーは、エラー発生時に他のメッセージとともに出力されます。エラーが発生したプログラムまたはレコード、およびそのときに実行中であった関数が識別されます。

ユーザー応答

なし。

EGL Java ランタイム・エラー・コード VGJ0005I

VGJ0005I: %1 でエラーが発生しました。

説明

このメッセージは他のメッセージとともに出力され、エラーが発生したプログラムまたはレコードを識別します。

ユーザー応答

なし。

EGL Java ランタイム・エラー・コード VGJ0006E

VGJ0006E: 入出力操作時にエラーが発生しました。 %1

説明

入出力操作が失敗しました。 EGL ステートメントにはこのエラーを処理する try ステートメントがありません。

ユーザー応答

このエラーをプログラムに処理させたい場合は、handleHardIOErrors を 1 に設定し、I/O ステートメントを try ステートメント内に配置します。次に例を示します。

```
VGVar.handleHardIOErrors = 1;

if (userRequest == "A")
  try
    add record1
  onException
    myErrorHandler(12);
  end
end
```

EGL Java ランタイム・エラー・コード VGJ0007E

VGJ0007E: %1 からの最小値オーバーフローが発生しました。

説明

算術演算により、データ・タイプに許容される最小値を超える結果が生成されました。

ユーザー応答

演算式を適宜調整してください。

EGL Java ランタイム・エラー・コード VGJ0008E

VGJ0008E: 回復可能リソース・エラーが発生しました。 %1

説明

回復可能リソースを閉じている最中、コミット中、またはロールバック中にエラーが発生しました。

ユーザー応答

エラー・メッセージの情報をを使用して、問題を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0009E

VGJ0009E: ID %1 を持つフィールドが %2 に見つかりませんでした。

説明

示されたフィールドが存在しないため、動的アクセスが失敗しました。

ユーザー応答

存在しないフィールドにアクセスしないでください。

EGL Java ランタイム・エラー・コード VGJ0010E

VGJ0010E: %1 への割り当てが失敗しました: 割り当てソース %2 の非互換です。

説明

ソースの型が、ターゲットへ割り当て可能な型ではありません。

ユーザー応答

値を割り当てるときは、ソース型とターゲット型に互換性があることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0011E

VGJ0011E: %1 の値をプリミティブ型へ解決できません。

説明

変数がデータ項目として使用されましたが、データ項目ではありませんでした。

ユーザー応答

変数をデータ項目として使用しないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0012E

VGJ0012E: 演算式を評価できませんでした: %1 での型が非互換です。

説明

式にある値の型の非互換です。

ユーザー応答

式で互換性のある型を使用するようにプログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0013E

VGJ0013E: set ステートメントが失敗しました: %1 を %2 状態に設定できません。

説明

指定された状態は、変数ではサポートされていません。

ユーザー応答

この操作を行わないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0014E

VGJ0014E: %1 に添え字を付けることはできません。 配列ではありません。

説明

変数が配列として使用されましたが、それは配列ではありません。

ユーザー応答

変数を配列として使用しないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0015E

VGJ0015E: %1、%2

説明

エラーがありました。例外とそのメッセージがこのメッセージへの挿入として使用されています。

ユーザー応答

メッセージ挿入からの情報を使用して、問題を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0016E

VGJ0016E: 変数 %1 に値が指定されていません。

説明

この変数は、値が指定される前に使用されました。

ユーザー応答

使用する前に変数に値を割り当てるように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0017E

VGJ0017E: Ref 変数 %1 が Nil です。

説明

この変数は、使用できるようにするには、値を参照しなければなりません。

ユーザー応答

変数を使用する前に、変数に値を指定してください。

EGL Java ランタイム・エラー・コード VGJ0018E

VGJ0018E: 構造化レコード %1 では動的アクセスを実行することはできません。

説明

構造化レコードでは動的アクセスは許可されません。

ユーザー応答

構造化レコードで動的アクセスを使用しないでください。

EGL Java ランタイム・エラー・コード VGJ0019E

VGJ0019E: %1 はコピーできません。

説明

操作でコピーできないものをコピーしようとしたか、コピーの試行が失敗しました。

EGL Java ランタイム・エラー・コード VGJ0020E

VGJ0020E: %1 という名前の変数を %2 として使用することはできません。

説明

この変数の型では、それが指定された型であるかのように使用することはできません。

ユーザー応答

この変数を別の型であるかのように使用しないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0021E

VGJ0021E: %1 は %2 状態についてはテストできません。

説明

IS または NOT 式でエラーが発生しました。式の左側の変数は、式の右側として指定された状態をサポートしていません。

ユーザー応答

式を除去または変更してください。

EGL Java ランタイム・エラー・コード VGJ0050E

VGJ0050E: プログラム %1 のロード中に例外が発生しました。例外: %2。メッセージ: %3。

説明

プログラムのクラスをロードできませんでした。

ユーザー応答

例外メッセージを使用し、問題を診断して修正してください。このエラーの最も一般的な原因は、プログラムのクラス・ファイルが含まれている jar ファイルまたはディレクトリーが CLASSPATH 環境変数にリストされていないということです。

EGL Java ランタイム・エラー・コード VGJ0055E

VGJ0055E: プログラム %1 の呼び出し時にエラーが発生しました。エラー・コードは %2 (%3) でした。

説明

ローカル Java プログラムの呼び出し中にエラーが発生しました。

ユーザー応答

例外メッセージを使用し、問題を診断して修正してください。

EGL Java ランタイム・エラー・コード VGJ0056E

VGJ0056E: 呼び出し先プログラム %1 は、%2 個のパラメーターを予期していましたが %3 個のパラメーターが渡されました。

説明

誤った数のパラメーターが呼び出し先プログラムに渡されました。

ユーザー応答

呼び出し側プログラムまたは呼び出し先プログラムを作成し直し、両方に同じ数のパラメーターが渡されるようにします。

EGL Java ランタイム・エラー・コード VGJ0057E

VGJ0057E: 呼び出し先プログラム %1 にパラメーターを渡すときに例外が発生しました。例外: %2。メッセージ: %3。

説明

Java プログラムの呼び出し中にエラーが発生しました。エラーは、プログラムの開始前または開始後に発生しています。

ユーザー応答

例外およびメッセージを使用し、問題を診断して修正してください。

EGL Java ランタイム・エラー・コード VGJ0058E

VGJ0058E: プロパティ・ファイル %1 をロードできませんでした。

説明

プログラムのプロパティ・ファイルをロードできませんでした。プロパティ・ファイルの名前は、システム・プロパティ `vgj.properties.file` から取得されます。

ユーザー応答

`vgj.properties.file` のファイル名が正しいこと、およびプロパティ・ファイルが `CLASSPATH` 環境変数内にリストされた Jar ファイルまたはディレクトリーにあることを確認します。

EGL Java ランタイム・エラー・コード VGJ0060E

VGJ0060E: クラス %1 への `StartTransaction` が失敗しました。例外は %2 です。

説明

指定したサーバー・クラスを新規のトランザクションとして実行するためにプログラムが新規の JVM の起動を試行しているときに、例外がスローされました。

`vgj.java.command` プロパティは、新規の JVM の起動に使用されるコマンドを指定します。デフォルト・コマンドは `java` です。

ユーザー応答

`vgj.java.command` プロパティの値が正しく、プログラムが新規プロセスを作成する許可を持っていることを確認します。

`startTransaction` ステートメントを `try` ステートメントの内部に配置して、これが致命的エラーにならないようにします。 `try` ステートメントの内部で `startTransaction` が失敗すると、`errorCode` システム変数内にエラー・コードが保管されます。

EGL Java ランタイム・エラー・コード VGJ0062E

VGJ0062E: MQ プログラム %1 に渡された 1 つ以上のパラメーターの型が正しくないものでした。 %2

説明

MQ プログラムを呼び出すときに例外がスローされました。パラメーターが誤っています。

ユーザー応答

MQ プログラムの資料および例外のメッセージを参照し、エラーを訂正してください。

EGL Java ランタイム・エラー・コード VGJ0064E

VGJ0064E: プログラム %1 はテキスト書式 %2 を予期していましたが、`show` ステートメント上でテキスト書式 %3 を渡されました。

説明

両方のプログラムが同じテキスト書式を使用する必要があります。

ユーザー応答

同じテキスト書式を使用するようにプログラムを変更し、再生成します。

EGL Java ランタイム・エラー・コード VGJ0100E

VGJ0100E: %1 のデータが %2 形式ではありません。

説明

項目のデータが予期しない形式になっています。指定の項目が別のデータ項目によって上書きされた可能性があります。

ユーザー応答

エラーを回避するようにプログラム・ロジックを訂正してください。

EGL Java ランタイム・エラー・コード VGJ0104E

VGJ0104E: %1 は、%3 の添え字 %2 の有効なインデックスではありません。

説明

多次元配列で使用された添え字の 1 つが無効です。添え字の値は、1 と、添え字付き項目に定義された出現数の間の値でなければなりません。

ユーザー応答

インデックス値が添え字付き項目の有効な添え字であることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0105E

VGJ0105E: %1 は %2 の有効なインデックスではありません。

説明

添え字の値は、1 と、添え字付き項目に定義された出現数の間の値でなければなりません。

ユーザー応答

インデックス値が添え字付き項目の有効な添え字であることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0106E

VGJ0106E: %1 を %2 に割り当てるときにユーザー・オーバーフローが発生しました。

説明

有効数字を切り捨てずに結果を保持できる十分な大きさが割り当て先にありません。システム変数 `VGVar.handleOverflow` の値が 1 であるため、プログラムは終了します。

ユーザー応答

以下のようにします。

- ターゲットの有効桁数を増加させる。または
- オーバーフロー条件を処理するためのプログラム・ロジックを定義する。システム変数 `VGVar.handleOverflow` と `overflowIndicator` を使用します。

EGL Java ランタイム・エラー・コード VGJ0108E

VGJ0108E: HEX 項目 %1 に 16 進数以外の値 %2 が割り当てられました。

説明

HEX 項目は 16 進数字のみを受け取ることができます。

ユーザー応答

ソース値が 16 進数字しか持たないことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0109E

VGJ0109E: HEX 項目 %1 に %2 からの 16 進数以外の値 %3 が割り当てられました。

説明

HEX 項目は 16 進数字のみを受け取ることができます。

ユーザー応答

割り当て元が 16 進数字しかもっていないことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0110E

VGJ0110E: HEX 項目 %1 を 16 進数以外の値 %2 と比較しました。

説明

HEX 項目は 16 進数字のみと比較できます。

ユーザー応答

比較値には 16 進数字しか含まれていないことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0111E

VGJ0111E: HEX 項目 %1 を %2 からの 16 進数以外の値 %3 と比較しました。

説明

HEX 項目は 16 進数字のみと比較できます。

ユーザー応答

比較値には 16 進数字しか含まれていないことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0112E

VGJ0112E: NUM 項目 %1 に非数値 %2 が割り当てられました。

説明

NUM 項目には数値のみを割り当てることができます。この値には数字が含まれ、先行スペースおよび後続スペース、小数点、および先行する符号を持つことができ

ます。小数点は、2 つの数字の間、最初の桁の直前、または最後の桁の直後に置くことができます。

ユーザー応答

ソース値が数値であることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0113E

VGJ0113E: NUM 項目 %1 に %2 からの数値以外の値 %3 が割り当てられました。

説明

NUM 項目には数値のみを割り当てることができます。この値には数字が含まれ、先行スペースおよび後続スペース、小数点、および先行する符号を持つことができます。小数点は、2 つの数字の間、最初の桁の直前、または最後の桁の直後に置くことができます。

ユーザー応答

ソース値が数値であることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0114E

VGJ0114E: 項目 %1 (%2) の値は添え字として無効です。

説明

値の桁数が配列の任意の要素の添え字として多すぎます。添え字の値は、1 と、構造体項目に宣言された出現数の間の値でなければなりません。

ユーザー応答

インデックス値が配列の有効な添え字であることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0115E

VGJ0115E: %1 にはストリングを割り当てることはできません。 ストリングは %2 でした。

説明

この項目にはストリングを割り当てることはできません。

ユーザー応答

この項目にストリングを割り当てないでください。

EGL Java ランタイム・エラー・コード VGJ0116E

VGJ0116E: %1 には数値を割り当てることはできません。 数値は %2 でした。

説明

この項目には数値を割り当ててはできません。

ユーザー応答

この項目に数値を割り当てないでください。

EGL Java ランタイム・エラー・コード VGJ0117E

VGJ0117E: %1 は long に変換できません。

説明

この項目は long に変換できません。

ユーザー応答

以下の手順を実行してください。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。
3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0118E

VGJ0118E: %1 を数値に変換することはできません。

説明

この項目を数値に変換することはできません。

ユーザー応答

数値が必要な場所にこの項目を使用しないでください。

EGL Java ランタイム・エラー・コード VGJ0119E

VGJ0119E: %1 は無効な数値です。

説明

デバッガーの使用中にユーザーが数値項目の値を設定しようとしたましたが、新規の値は数値ではありません。

ユーザー応答

数値を使用してください。

EGL Java ランタイム・エラー・コード VGJ0120E

VGJ0120E: %1 は、項目 %2 のサブストリング演算子の開始インデックスとして有効な値ではありません。

説明

開始インデックスは、1 より小さいかまたは項目の長さより大きくすることはできません。

ユーザー応答

サブストリング演算子の開始インデックスとして有効なインデックスを使用してください。

EGL Java ランタイム・エラー・コード VGJ0121E

VGJ0121E: %1 は、項目 %2 のサブストリング演算子の終了インデックスとして有効な値ではありません。

説明

終了インデックスは、1 より小さいかまたは項目の長さより大きくすることはできません。

ユーザー応答

サブストリング演算子の終了インデックスとして有効なインデックスを使用してください。

EGL Java ランタイム・エラー・コード VGJ0122E

VGJ0122E: 項目 %1 のサブストリング演算子の終了インデックスは %2 であり、これは開始インデックス %3 より小さくすることはできません。

説明

サブストリング演算子の終了インデックスは開始インデックスより小さくすることはできません。

ユーザー応答

開始インデックスが終了インデックスより小または等しいことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0123E

VGJ0123E: サブストリング演算子が失敗しました。 %1 をストリング値として使用することはできません。

説明

この変数はサブストリング演算子をサポートしません。

ユーザー応答

この変数でサブストリング演算子を使用しないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0124E

VGJ0124E: %1 にはレコードを割り当てることはできません。レコードは %2 でした。

説明

このデータ項目の型では、レコードからの割り当ては許可されません。

ユーザー応答

レコードをこのデータ項目に割り当てないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0125E

VGJ0125E: %1 はフィールドとして使用できません。

説明

この変数はフィールドではありません。

ユーザー応答

この変数をフィールドとして使用しないように、プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0126E

VGJ0126E: %2 に対する %1 の比較で型に互換性がありません。

説明

値の型が比較において非互換です。

ユーザー応答

比較で、互換性のある型が使用されていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0127E

VGJ0127E: %1 には date または time 値を割り当てることはできません。値は %2 でした。

説明

この項目には date または time 値を割り当てることができません。

ユーザー応答

この項目に date または time 値を割り当てないでください。

EGL Java ランタイム・エラー・コード VGJ0140E

VGJ0140E: 配列関数 %1 が失敗しました。 配列 %2 を最大サイズを超えて拡張しようとしたためです。

説明

この配列はこれ以上の値を保持できません。

ユーザー応答

配列への追加を試行する前に配列のサイズを検査するようにプログラムを変更します。

EGL Java ランタイム・エラー・コード VGJ0141E

VGJ0141E: %1 は配列 %2 の無効なインデックスです。 現行サイズ: %3。最大サイズ: %4

説明

このインデックスはこの配列の範囲外です。

ユーザー応答

有効な配列インデックスを使用するようにプログラムを変更します。

EGL Java ランタイム・エラー・コード VGJ0142E

VGJ0142E: 配列 %1 の `maximumSize` は変更できません。 %2 を予期していましたが、%3 が渡されました。

説明

`call` ステートメントでこの配列が渡されました。呼び出し先プログラム内の対応する配列には、異なる `maximumSize` があります

ユーザー応答

両方のプログラムが同じ `maximumSize` の配列を使用するように、いずれかのプログラムを変更します。

EGL Java ランタイム・エラー・コード VGJ0143E

VGJ0143E: %1 は配列 %2 の有効なサイズではありません。

説明

call ステートメントでこの配列が渡されました。呼び出し先プログラムが、配列のサイズをゼロより小さい値か、またはプロパティ `maxSize` の値より大きい値に変更しました。

ユーザー応答

プロパティ `maxSize` の同じ値を使用するようプログラムを変更します。

EGL Java ランタイム・エラー・コード VGJ0144E

VGJ0144E: %1 が配列 %2 について失敗しました。 指定されたサイズが多すぎます。

説明

指定された関数が失敗しました。その引数はサイズの配列ですが、含まれているエレメントが多すぎます。

ユーザー応答

引数を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0145E

VGJ0145E: %1 が配列 %2 について失敗しました。 サイズは数値データ項目でなければなりません。

説明

指定された関数が失敗しました。その引数は数値データ項目でなければなりません、数値データ項目ではありませんでした。

ユーザー応答

引数を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0146E

VGJ0146E: %1 が配列 %2 について失敗しました。 指定されたサイズがゼロを下回っていました。

説明

指定された関数が失敗しました。その引数はサイズの配列です。 サイズの 1 つがゼロを下回っていましたが、それは許可されていません。

ユーザー応答

引数を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0147E

VGJ0145E: %1 が配列 %2 について失敗しました。指定された `maxSize` が現行サイズを下回っています。

説明

配列の `maxSize` は、現行サイズより小さい値に変更することはできません。

ユーザー応答

引数を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0160E

VGJ0160E: 数学関数 %1 がエラー・コード 8 (領域エラー) で失敗しました。

説明

関数の引数が無効です。

ユーザー応答

以下のようにします。

- 関数の資料に従い、プログラム・ロジックを変更して関数の引数が有効な値であることを確認する。または
- `try` ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0161E

VGJ0161E: 数学関数 %1 がエラー・コード 8 (領域エラー) で失敗しました。

説明

引数は -1 と 1 の間でなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す引数が -1 と 1 の間であることを確認する。または
- `try` ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0162E

VGJ0162E: 数学関数 `atan2` がエラー・コード 8 (領域エラー) で失敗しました。

説明

両方の引数を 0 にすることはできません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す引数のうち少なくとも一方が 0 でないことを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0163E

VGJ0163E: 数学関数 %1 がエラー・コード 8 (領域エラー) で失敗しました。

説明

第 2 引数が 0 であってはなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、第 2 引数が 0 でないことを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0164E

VGJ0164E: 数学関数 %1 がエラー・コード 8 (領域エラー) で失敗しました。

説明

引数は 0 より大きくなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す引数が 0 より大きいことを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0165E

VGJ0165E: 数学関数 `pow` がエラー・コード 8 (領域エラー) で失敗しました。

説明

第 1 引数が 0 の場合は、第 2 引数は 0 より大きくなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す第 1 引数が 0 の場合は第 2 引数が 0 より大きいことを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0166E

VGJ0166E: 数学関数 `pow` がエラー・コード 8 (領域エラー) で失敗しました。

説明

第 1 引数が負の場合は、第 2 引数は整数でなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す第 1 引数が負の場合は第 2 引数が整数であることを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0167E

VGJ0167E: 数学関数 `sqrt` がエラー・コード 8 (領域エラー) で失敗しました。

説明

引数は 0 以上でなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す引数が 0 以上であることを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0168E

VGJ0168E: 数学関数 `%1` がエラー・コード 12 (範囲エラー) で失敗しました。

説明

中間結果を倍精度浮動小数点数として表現できないか、最終結果を結果項目の精度で表現できません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、ターゲット項目に結果値を保持できる大きさがあることを確認する。または
- プログラム・ロジックを変更し、関数の引数がこの問題を起こさない値であることを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0200E

VGJ0200E: スtring関数 %1 がエラー・コード 8 で失敗しました。

説明

インデックスは 1 とStringの長さの間でなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数のインデックス関係の引数が 1 とStringの長さの間の範囲にあることを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0201E

VGJ0201E: スtring関数 %1 がエラー・コード 12 で失敗しました。

説明

長さは 0 より大きくなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す長さ引数の値が 0 より大きいことを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0202E

VGJ0202E: ストリング関数 `setNullTerminator` がエラー・コード 16 で失敗しました。

説明

ターゲット・ストリングの最後のバイトはブランクまたは NULL 文字でなければなりません。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、ターゲット・ストリングの最後のバイトがブランクまたは NULL 文字であることを確認する。または
- `try` ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0203E

VGJ0203E: ストリング関数 %1 がエラー・コード 20 で失敗しました。

説明

DBCHAR サブストリングまたは UNICODE サブストリングのインデックスによって文字の先頭のバイトを識別するには、インデックスを奇数にする必要があります。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡すインデックス引数が有効であることを確認する。または
- `try` ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0204E

VGJ0204E: ストリング関数 %1 がエラー・コード 24 で失敗しました。

説明

文字数全体を参照するには、DBCHAR サブストリングまたは UNICODE サブストリングの長さを偶数にする必要があります。

ユーザー応答

以下のようにします。

- プログラム・ロジックを変更し、関数に渡す長さ引数の値が有効であることを確認する。または
- try ステートメントでこの関数を呼び出すか、この関数を呼び出す前に `VGVar.handleSysLibraryErrors` を 1 に設定して、プログラムがエラーを処理できるようにする。

EGL Java ランタイム・エラー・コード VGJ0215E

VGJ0215E: %1 に非数値ストリング %2 が渡されました。

説明

長さ引数によって定義されるストリングの部分のすべての文字は、数値である必要があります。

ユーザー応答

プログラム・ロジックを変更して、長さ引数によって定義されるストリングの部分の文字が数値になるようにします。

EGL Java ランタイム・エラー・コード VGJ0216E

VGJ0216E: %1 は %2 の有効な日付マスクではありません。

説明

関数で使用するためにプロパティ・ファイルで定義された日付マスクが無効です。

日付マスクに有効な文字は以下のとおりです。

D、M、Y

D は日、M は月、Y は年

セパレーター文字

数字以外かつ D、M、Y 以外の 1 バイト文字

有効な日付マスクは以下の形式のいずれかです。

- 長いグレゴリオ暦

長いバージョンのグレゴリオ・マスクは、以下のパーツを以下の順序で含んでいなければなりません。

YYYY 4 桁の年

MM 2 桁の数値による月

DD 2 桁の数値によるその月における日

マスク・パーツは、数字以外かつ D、M、Y 以外の 1 バイト文字で区切る必要があります。

たとえば、YYYY/MM/DD のマスクを使用すると、1997 年 8 月 25 日は 1997/08/25 と表示されます。

- 長いユリウス暦

長いバージョンのユリウス・マスクは、以下のパーツを以下の順序で含んでいなければなりません。

YYYY 4 桁の年

DDD 3 桁の数値による年間通算日

マスク・パーツは、D、M、Y 以外の 1 バイトの非数字で区切る必要があります。

たとえば、DDD-YYYY のマスクを使用すると、1997 年 8 月 25 日は 237-1997 と表示されます。

ユーザー応答

日付マスク・プロパティを有効な値に変更し、プログラムを再開してください。日付マスク・プロパティを定義していない場合は、デフォルトの日付マスクが使用されます。

日付マスクを設定するには、プロパティ `vgj.datemask.gregorian.long.NNN` および `vgj.datemask.julian.long.NNN` を使用します。ここで、`NNN` は現在の NLS コードです。

EGL Java ランタイム・エラー・コード VGJ0217E

VGJ0217E: convert 関数の引数 %1: %2 でエラーが発生しました

説明

引数のデータを変換しようとして失敗しました。失敗した原因はメッセージに含まれています。

ユーザー応答

エラー・メッセージを使用して、問題を診断および修正します。

EGL Java ランタイム・エラー・コード VGJ0218E

VGJ0218E: GetMessage が失敗しました。キー %1 のメッセージが見つかりませんでした。

説明

`getMessage` システム関数に渡されたキーのメッセージが見つかりません。

ユーザー応答

メッセージを追加するか、または別のキーを使用してください。

EGL Java ランタイム・エラー・コード VGJ0250E

VGJ0250E: 収容側のパーツ %2 から項目 %1 を検索できませんでした。

説明

内部エラーが発生しました。レコードまたはテーブルの指定のインデックスを持つ項目にアクセスしようとした。

ユーザー応答

以下のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。
3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0300E

VGJ0300E: テーブル %1 のテーブル・ファイルをロードできませんでした。 %2 または %3 という名前のファイルは見つかりませんでした。

説明

指定したいずれのファイルもリソース・ロケーションにありません。すべてのリソース・ロケーションは最初のファイルについて検索されます。そのようなファイルが存在しない場合は、2 番目のファイルについてすべてのリソース・ロケーションが検索されます。

リソース・ロケーションは、テーブル・ファイルの検索に使用するメカニズムによって異なります。

アプレット内でエラーが発生した場合、リソース・ロケーションはサーバー・マシン上のロケーションを指すため、Java 仮想マシンの実装に応じて異なることがあります。しかし、すべての実装は CODEBASE 値によって指定されたサーバーでディレクトリを検索する必要があります。この値は、アプレットを含む HTML ファイルの APPLET タグで設定します。CODEBASE 値が指定されていない場合は、デフォルトでその HTML ファイルが含まれている Web サーバーのディレクトリになります。

アプリケーションでエラーが発生した場合は、有効なリソース・ロケーションは以下のとおりです。

- Java 仮想マシンが起動されたディレクトリ (実行可能ファイルの作業ディレクトリ)。
- 実行しているアプリケーションの CLASSPATH で指定されたすべてのディレクトリ。この値の指定はシステムに依存します。システムによっては、環境変数として指定する場合があります。-classpath オプションを使用して Java 仮想マ

シンを起動する場合は、すべてのシステムでこの値を指定できます。
CLASSPATH の値の詳細については、ご使用の Java 仮想マシンのコピーに付属している資料を参照してください。

ユーザー応答

最初にテーブル・ファイルを探し、テーブル・ファイルにアクセスするために必要なアクセス権を設定してください。

アプレット内でエラーが発生した場合、またはアプリケーション内でエラーが発生したが既存のリソース・ロケーションのセットを変更したくない場合は、有効なリソース・ロケーションにテーブル・ファイルをコピーします。

それ以外の場合は、以下の指示のいずれかを完了してください。

- Java インタープリターが CLASSPATH 環境変数の値を使用する場合は、テーブル・ファイルが入っているディレクトリーを CLASSPATH の現行値に追加します。
- Java インタープリターを起動するときに -classpath オプションを使用して、テーブル・ファイルが入っているディレクトリーを指定します。-classpath オプションを指定すると CLASSPATH 環境変数の値がオーバーライドされる場合は、リソース・ロケーションとして追加するすべてのディレクトリー以外に、Java ランタイム・クラス (classes.zip または rt.jar など) へのパスも指定する必要があります。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0301E

VGJ0301E: テーブル・ヘッダーの読み取り操作時に誤ったバイト数が戻されたため、テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルが壊れている。
- テーブル・ファイルが EGL または VisualAge Generator で生成されていない。

ユーザー応答

テーブルを再生成します。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所

- 内部エラーの種類
- 2. このメッセージが出力された状況を記録する。
- 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0302E

VGJ0302E: テーブル・ヘッダーの検査時に予期しないマジック・ナンバーが発生したため、 テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルが壊れている。
- テーブル・ファイルが EGL または VisualAge Generator で生成されていない。

ユーザー応答

テーブルを再生成します。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した箇所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0303E

VGJ0303E: 読み取り操作時またはクローズ操作時に内部入出力エラーが発生したため、 テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルが壊れている。
- テーブル・ファイルが EGL または VisualAge Generator で生成されていない。

ユーザー応答

テーブルを再生成します。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。
3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0304E

VGJ0304E: テーブル・データの読み取り操作時に誤ったバイト数が戻されたため、テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルの列が変更された後、テーブル・ファイルが再生成されましたが、テーブルをロードするプログラムが再生成されていない。列定義を変更した後にテーブルしか生成しないと、テーブル・ファイルの定義とテーブル・クラス・ファイルの定義 (ランタイム・コード生成時にのみ生成される) の間に不整合が起こります。
- テーブル・ファイルが壊れている。
- テーブル・ファイルが EGL または VisualAge Generator で生成されていない。

ユーザー応答

以下のようになります。

- 列定義を変更していない場合は、テーブルを再生成する。
- 列定義を変更した場合は、変更を除去してテーブルを再生成するか、テーブルを使用するプログラムのランタイム・コードを再生成する。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。

3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0305E

VGJ0305E: テーブル %2 のテーブル・ファイル %1 はロードできませんでした。
項目 %3 のテーブル・ファイルで発生したデータが正しい形式ではありません。
対応するデータ形式エラーは %4 です。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルの列が変更された後、テーブル・ファイルが再生成されたが、テーブルをロードするアプレットまたはアプリケーションが再生成されていない。列定義を変更した後にテーブルしか生成しないと、テーブル・ファイルの定義とテーブル・クラス・ファイルの定義 (ランタイム・コード生成時にのみ生成される) の間に不整合が起こります。
- テーブル・ファイルが壊れている。
- テーブル・ファイルが Rational Application Developer for z/OS または VisualAge Generator で生成されていない。

ユーザー応答

以下のようになります。

- 列定義を変更していない場合は、テーブルを再生成する。
- 列定義を変更した場合は、変更を除去してテーブルを再生成するか、テーブルを使用するプログラムのランタイム・コードを再生成する。

問題が解決しない場合は、次のようになります。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した箇所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0306E

VGJ0306E: テーブル・ファイルのデータは、 テーブル %2 と型が異なるため、
テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

以下のいずれかの条件が存在します。

- テーブル・ファイルの列が変更された後、テーブル・ファイルが再生成されたが、テーブルをロードするアプレットまたはアプリケーションが再生成されてい

ない。列定義を変更した後にテーブルしか生成しないと、テーブル・ファイルの定義とテーブル・クラス・ファイルの定義 (ランタイム・コード生成時にのみ生成される) の間に不整合が起こります。

- テーブル・ファイルが壊れている。
- テーブル・ファイルが EGL または VisualAge Generator で生成されていない。

ユーザー応答

以下のようにします。

- テーブル・タイプを変更していない場合は、テーブルを再生成する。
- テーブル・タイプを変更した場合は、テーブル定義を編集して正しいタイプにしてからテーブルを再生成するか、テーブルを使用するプログラムのランタイム・コードを再生成する。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0307E

VGJ0307E: テーブル・ファイル %1 は VisualAge Generator C++ テーブル・ファイルであり、ビッグ・エンディアン形式ではないため、テーブル %2 のテーブル・ファイル %1 はロードできませんでした。

説明

VisualAge Generator C++ 生成プログラムで生成したテーブル・ファイルは、テーブル内部の数値データをエンコードするときに使用したバイト順序がビッグ・エンディアンの場合、Java プログラムでのみ使用できます。

ユーザー応答

ビッグ・エンディアン形式または Java プラットフォームとは独立のテーブルとしてテーブルを再生成します。

ビッグ・エンディアン形式でテーブルを再生成するには、VisualAge Generator を使用し、ビッグ・エンディアン (AIX など) の C++ ターゲット・システムのためのテーブルを生成します。Java プラットフォームとは独立のテーブルとしてテーブルを再生成するには、1 VisualAge Generator または EGL で Java ターゲット・システムのテーブルを生成します。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0308E

VGJ0308E: テーブル %2 のテーブル・ファイル %1 はロードできませんでした。テーブル・ファイル %1 は VisualAge Generator C++ テーブル・ファイルであり、テーブル (%3) で使用している文字エンコードがランタイム・システムでサポートされていません。

説明

VisualAge Generator C++ 生成プログラムで生成したテーブル・ファイルは、テーブル内部のデータに使用した文字エンコードが、ランタイム・システムで使用するエンコードと同じ場合にのみ、Java プログラムで使用できます。

ユーザー応答

以下のようにします。

1. システムで使用する文字エンコードを判別する。Java プログラムは ASCII または EBCDIC の文字エンコードを使用します。ほとんどのワークステーションは ASCII エンコードを使用します。ほとんどのホスト・プラットフォームは EBCDIC エンコードを使用します。システムで使用するエンコードが不明な場合は、システム管理者に問い合わせてください。
2. 適切な文字エンコードを使用するか Java プラットフォームとは独立のテーブルとしてテーブルを再生成します。

正しい文字エンコードを使用してテーブルを再生成するには、VisualAge Generator を使用し、使用しているターゲット・システムまたは同じ文字エンコードを使用する別の C++ ターゲット・システムのためのテーブルを生成します。Java プラットフォームとは独立のテーブルとしてテーブルを再生成するには、1 VisualAge Generator または EGL で Java ターゲット・システムのテーブルを生成します。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0315E

VGJ0315E: テーブルのアンロード・プロセス時にテーブル %1 の共用テーブル・エントリが見つかりませんでした。

説明

内部エラーが発生しました。

ユーザー応答

以下のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0320E

VGJ0320E: テーブル列 %2 とフィールド %3 を比較するときに、テーブル %1 の編集ルーチンが失敗しました。

説明

テーブル列とフィールドの型が比較には無効な型になっています。

ユーザー応答

以下のいずれか 1 つを行います。

- 以下のことを実行して、列とフィールドの型が比較に有効であることを確認します。
 1. 列の型またはフィールドの型を訂正し、比較が有効であるようにします。
 2. プログラムを再生成してください。
 3. プログラムを実行します。
- 列とフィールドの比較が有効になるように別のテーブルを編集ルーチンに使用するようにプログラムを変更します。

詳細については、トレース出力を参照してください。

EGL Java ランタイム・エラー・コード VGJ0330E

VGJ0330E: ID %1 のメッセージがメッセージ・テーブル %2 に見つかりませんでした。

説明

このエラーは以下の操作のときに発生します。

- 書式の msgField の値のルックアップ。
- 編集メッセージとして指定された ID を持つ値のルックアップ。

以下のいずれかの条件が存在します。

- この ID を持つメッセージがメッセージ・テーブルに存在しません。
- テーブル・ファイルまたはテーブルのメッセージ・リソース・バンドルが壊れています。

ユーザー応答

以下のいずれか 1 つを行います。

- 以下のことを行い、このメッセージ ID を持つメッセージが存在することを確認します。
 1. このメッセージ ID を持つメッセージがまだテーブルに存在しない場合は、メッセージを追加します。
 2. テーブルを再生成してください。
 3. プログラムを実行します。
- テーブルですでに定義されている別のメッセージを使用するようにプログラムを変更します。
- このメッセージ ID を持つメッセージが含まれている別のメッセージ・テーブルを使用するようにプログラムを変更します。

EGL Java ランタイム・エラー・コード VGJ0331E

VGJ0331E: メッセージ・テーブル・ファイル %1 をロードできませんでした。

説明

プログラムのメッセージ・テーブルのクラスをロードできなかったか、クラスのインスタンスを作成できませんでした。

ユーザー応答

メッセージ・テーブルが生成されていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0350E

VGJ0350E: プログラム %1 の呼び出し時にエラーが発生しました。エラー・コードは %2 でした。

説明

指定されたプログラムのリモートまたは EJB 呼び出しが失敗しました。

ユーザー応答

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0351E

VGJ0351E: コミットが失敗しました: %1

説明

リソースをコミットできませんでした。

ユーザー応答

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0352E

VGJ0352E: rollBack が失敗しました: %1

説明

リソースをロールバックできませんでした。

ユーザー応答

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0400E

VGJ0400E: 無効なパラメーター・インデックス %1 が関数 %2 に使用されました。

説明

これは内部エラーです。

ユーザー応答

IBM サポートに連絡してください。

EGL Java ランタイム・エラー・コード VGJ0401E

VGJ0401E: 関数 %1、パラメーター %2 で無効なパラメーター記述子を検出しました。

説明

これは内部エラーです。

ユーザー応答

IBM サポートに連絡してください。

EGL Java ランタイム・エラー・コード VGJ0402E

VGJ0402E: 関数またはプログラム %2 のパラメーター %1 に使用された値の型が無効です。

説明

値の型とパラメーターの型に互換性がないため、この値をパラメーターとして渡すことはできません。

ユーザー応答

以下のいずれか 1 つを行います。

- 値の型に一致するようにパラメーターの定義を変更する。

- パラメーターの定義に一致するように値の型を変更する。

EGL Java ランタイム・エラー・コード VGJ0403E

VGJ0403E: スクリプト %1 の実行中にエラーが発生しました。例外テキストは %2 です。

説明

スクリプトによって例外がスローされました。

ユーザー応答

エラーを回避するようにプログラム・ロジックを訂正してください。

EGL Java ランタイム・エラー・コード VGJ0416E

VGJ0416E: プログラム %1 の呼び出し時にエラーが発生しました。エラー・コードは %2 (%3) でした。

説明

呼び出し先プログラムを実行しようとしたときに例外がスローされました。問題の原因としては以下のいずれかが考えられます。

- プログラムが新規プロセスの作成を許可されていない。
- 呼び出し先プログラムが存在しない。
- 呼び出し先プログラムがシステム・パスに存在しない。

ユーザー応答

以下のようにします。

1. プログラムによる新規プロセスの作成を許可をもっていることを検証する。
2. 呼び出し先プログラムが存在することを検証する。
3. 呼び出し先プログラムがシステム・パスで検出できることを検証する。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ0450E

VGJ0450E: 入出力オブジェクト %2 による入出力操作 %1 が失敗しました。原因は %3 です。

説明

try ステートメントの外側で、またはシステム変数 `sysVar.handleHardIoErrors` の値がゼロのときに、EGL I/O 文が失敗しました。

ユーザー応答

エラー・メッセージを検討し、適切に対処してください。

EGL Java ランタイム・エラー・コード VGJ0500E

VGJ0500E: 必要フィールドへの入力を受け取っていません。 もう一度入力してください。

説明

フィールドにデータが入力されませんでした。フィールドは必要フィールドとして定義されています。

ユーザー応答

データをフィールドに入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれのタイプのフィールドにおいても、ブランクはデータ入力要件を満たしません。また、ゼロは数値フィールドのデータ入力要件を満たしません。プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0502E

VGJ0502E: 入力でデータ・タイプ・エラーが発生しました。 もう一度入力してください。

説明

フィールドのデータは無効な数値データです。フィールドは数値として定義されていました。

ユーザー応答

このフィールドに数値データのみを入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの状況でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0503E

VGJ0503E: 許容される有効数字の桁数を超えました。 もう一度入力してください。

説明

小数点以下の桁、符号、通貨記号、または数字分離記号編集で定義された数値フィールドにデータが入力されました。入力データが、編集基準内に表示できる有効数字の桁数を超えています。 入力された数値が大きすぎます。有効数字の桁数は、フィールド長から小数点以下の桁数を引き、さらに編集文字に必要な桁数を引いた長さを超えることはできません。

ユーザー応答

有効数字の桁数が少ない数値を入力してください。

EGL Java ランタイム・エラー・コード VGJ0504E

VGJ0504E: 入力データが定義された範囲内ではありません。もう一度入力してください。

説明

フィールドのデータが、この項目に定義された有効なデータの範囲に入っていない。

ユーザー応答

定義された範囲内のデータを入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの場合でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0505E

VGJ0505E: 入力で最小長エラーが発生しました。もう一度入力してください。

説明

フィールドのデータに、必要な最小長を満たす十分な文字がありません。

ユーザー応答

最小長を満たす必要な文字数を入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの場合でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0506E

VGJ0506E: テーブル編集妥当性エラーが発生しました。もう一度入力してください。

説明

フィールドのデータが、可変フィールドに定義されたテーブル編集要件を満たしていません。

ユーザー応答

テーブル編集要件に適合するデータを入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの場合でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0507E

VGJ0507E: 入力のモジュラス・チェックでエラーが発生しました。もう一度入力してください。

説明

フィールドのデータが、可変フィールドに定義されたモジュラス・チェック要件を満たしていません。

ユーザー応答

可変フィールドに定義されたモジュラス・チェックに適合するデータを入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの場合でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0508E

VGJ0508E: 定義された日付または時刻の形式 %1 に対する入力が無効です。

説明

日付編集で定義されたフィールドのデータが形式指定の要件を満たしていません。

ユーザー応答

メッセージに示された正しい日付形式で日付を入力してください。

EGL Java ランタイム・エラー・コード VGJ0510E

VGJ0510E: ブール・フィールドに対する入力が無効です。

説明

フィールドに入力された値がブール・チェックに適合しません。ブール・フィールドへの入力は、文字フィールドの場合は「Y」または「N」のいずれかでなければならず、数値フィールドの場合は 1 または 0 のいずれかでなければなりません。

ユーザー応答

文字フィールドの場合は「Y」または「N」を、数値フィールドの場合は 1 または 0 を入力するか、編集バイパス・キーを押して編集チェックをバイパスしてください。いずれの場合でも、プログラムは続行します。

EGL Java ランタイム・エラー・コード VGJ0511E

VGJ0511E: 編集テーブル %1 は %2 に定義されていません。

説明

ユーザー・メッセージが要求されましたが、ユーザー・メッセージ・テーブル接頭部がプログラムに定義されていませんでした。

ユーザー応答

プログラム開発者に以下のいずれかを行うように依頼してください。

- メッセージ・テーブル接頭部をプログラム仕様に追加し、プログラムを再生成する。
- フィールド編集からユーザー・メッセージ番号を除去して再生成する。

EGL Java ランタイム・エラー・コード VGJ0512E

VGJ0512E: 16 進データが無効です。

説明

可変フィールドのデータは 16 進形式でなければなりません。入力した文字のうち、1 つ以上が a b c d e f A B C D E F 0 1 2 3 4 5 6 7 8 9 のセットに含まれていません。

ユーザー応答

可変フィールドには 16 進文字のみを入力してください。文字は左寄せされ、文字 0 が埋め込まれます。埋め込まれたブランクの使用は許されません。

EGL Java ランタイム・エラー・コード VGJ1234E

VGJ1234E: 入力された値は、設定されたパターンと一致しないため無効です。

説明

パターンと一致しない値が入力されました。

ユーザー応答

パターンで指定されているとおりに値を入力してください。

EGL Java ランタイム・エラー・コード VGJ1234E

VGJ0514E: 最大入力文字数エラー - 再度入力してください。

説明

このフィールドに指定された最大長を超過しています。

ユーザー応答

指定された最大長を超えないようにしてください。

EGL Java ランタイム・エラー・コード VGJ0516E

VGJ0516E: 入力が定義されたリスト内にありません - 再入力してください。

説明

入力が定義されたリスト内にありません - 再入力してください。

ユーザー応答

入力が定義されたリスト内にありません - 再入力してください。

EGL Java ランタイム・エラー・コード VGJ0517E

VGJ0517E: 指定された日付/時刻形式 %1 は無効です。

説明

指定された日付/時刻形式が無効です。

ユーザー応答

指定された日付/時刻形式が無効です。

EGL Java ランタイム・エラー・コード VGJ0600E

VGJ0600E: プログラム %1 のリンケージを取得できません。

説明

以下のいずれかの理由のため、指定したプログラムのエントリーが CSO プロパティー・ファイルにありません。

- 誤ったプロパティー・ファイルが GatewayServlet 構成で指定された。
- プログラムのエントリーが CSO プロパティー・ファイルで指定されなかった。
- GatewayServlet 構成で指定されたディレクトリーに CSO プロパティー・ファイルがない。

ユーザー応答

Web サーバー管理者に連絡し、以下のことが実行されていることを確認してください。

- GatewayServlet 構成が linkageTable 初期化パラメーターを使用して正しい CSO プロパティー・ファイルを指定していることを確認する。
- プログラムが CSO プロパティー・ファイルで定義されていることを確認する。

EGL Java ランタイム・エラー・コード VGJ0601E

VGJ0601E: エントリー・ポイント・プログラム %1 の呼び出しの試行時に例外が発生しました。例外: %2。メッセージ: %3。

説明

エントリー・ポイント・プログラムの呼び出し時の試行時に説明されないエラーが発生しました。例外およびメッセージがエラーを詳細に定義します。エントリー・ポイント・ページまたはエントリー・ポイント・プログラムは、GatewayServlet を使用して開始できるプログラムのメニューをユーザーに提供します。

ユーザー応答

Web サーバー管理者に連絡し、エントリー・ポイント・ページまたはエントリー・プログラムが GatewayServlet 構成で正しく指定されていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0603E

VGJ0603E: bean %1 は無効です。

説明

ページ Bean または Bean 名が無効です。

ユーザー応答

Web サーバー管理者に連絡し、Bean 名が正しいこと、ページ Bean および Java Server Page がデプロイされ、GatewayServlet で使用可能なことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0604E

VGJ0604E: Bean %1 のロード試行時に例外が発生しました。 例外: %2。メッセージ: %3。

説明

ページ Bean のロード試行時に説明されないエラーが発生しました。例外およびメッセージがエラーを詳細に定義します。

ユーザー応答

Web サーバー管理者に連絡し、Bean 名が正しいこと、ページ Bean および Java Server Page がデプロイされ、GatewayServlet で使用可能なことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0607E

VGJ0607E: サーバー %1 と Bean %2 のバージョンが一致していません。

説明

ユーザー・インターフェース・レコード Bean のバージョンが、サーバー・プログラムが使用するユーザー・インターフェース・レコードのバージョンと一致していません。正常に操作させるためには、バージョンに互換性がなければなりません。

ユーザー応答

プログラム開発者に連絡し、プログラムとユーザー・インターフェース・レコード Bean の両方の生成を行ってください。 Web サーバー管理者に連絡し、ユーザー・インターフェース・レコード Bean が正しいロケーションにデプロイされていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0608E

VGJ0608E: Bean %1 でのデータ設定試行時にエラーが発生しました。例外: %2。
メッセージ: %3。

説明

サーバー・アプリケーションからユーザー・インターフェース・レコード Bean にレコード・データを設定しようとするときに例外が発生しました。問題の判別に役立つ例外およびメッセージが含まれています。

ユーザー応答

メッセージに含まれている例外およびメッセージを使用して問題を判別してください。

EGL Java ランタイム・エラー・コード VGJ0609I

VGJ0609I: ユーザー %1 のためにゲートウェイ・セッションをバインドしています。

説明

この通知メッセージは、アプリケーション・サーバーの標準出力または標準エラー出力に出力されます。このメッセージは、ユーザーのために Web セッションを作成するたびに表示されます。

ユーザー応答

応答は不要です。

EGL Java ランタイム・エラー・コード VGJ0610I

VGJ0610I: ユーザー %1 のためにゲートウェイ・セッションをアンバインドしています。

説明

この通知メッセージは、アプリケーション・サーバーの標準出力または標準エラー出力に出力されます。このメッセージは、ユーザーのために Web セッションが終了するたびに表示されます。セッションは、一定期間の非活動状態の後、またはセッションが終了するような重大エラーが発生した場合に終了します。

ユーザー応答

応答は不要です。

EGL Java ランタイム・エラー・コード VGJ0611E

VGJ0611E: SessionIDManager との接続を確立できません。

説明

GatewayServlet が SessionIDManager に接続できませんでした。 SessionIDManager は、ゲートウェイ・ユーザーにセッション ID を与えるコンポーネントです。セッション ID は、アクティブ・セッションごとに取得され、サーバー・プログラムによってアプリケーション・データの保管および復元を行うために使用されます。

SessionIDManager は、ID の接続および要求を listen する別個のアプリケーションです。セッションが終了すると、SessionIDManager はセッション ID を他のセッションで使用できるようにします。 GatewayServlet を実行するには、SessionIDManager がアクティブである必要があります。

ユーザー応答

Web サーバー管理者に連絡し、SessionIDManager を開始してください。すでに開始している場合は、SessionIDManager のロケーションを GatewayServlet の構成で設定する必要があります。

EGL Java ランタイム・エラー・コード VGJ0612I

VGJ0612I: ゲートウェイ・セッションをユーザー %1 の SessionIDManager に接続します。

説明

この通知メッセージは、Web サーバーの標準出力または標準エラー出力に表示されます。セッション ID を取得するために、セッションが正常に SessionIDManager に接続しました。セッション ID は、サーバー・プログラムがプログラム・データの保管および復元を行うために使用します。

ユーザー応答

応答は不要です。

EGL Java ランタイム・エラー・コード VGJ0614E

VGJ0614E: 必要パラメーター %1 が GatewayServlet 構成にありません。

説明

必要パラメーターがサーブレット構成で指定されていませんでした。 GatewayServlet は、これらのパラメーターがないと実行しません。

ユーザー応答

Web サーバー管理者に連絡し、GatewayServlet が適切に構成されていることを確認してください。アプリケーション・サーバーの資料を参照し、サーブレット・パラメーターの構成方法を判別してください。

EGL Java ランタイム・エラー・コード VGJ0615E

VGJ0615E: EGL Action Invoker のこのインスタンスでは Web トランザクション %1 は実行を許可されていません。

説明

プログラムの GatewayRequestHandler を作成または取得するときに問題が発生しました。

ユーザー応答

指定されたアプリケーションが生成され、サーバーにデプロイされていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0616E

VGJ0616E: ゲートウェイ・パラメーター %1 が有効なクラス %2 を指定していません

説明

指定されたゲートウェイ・プロパティーで示されるクラスをロードまたはインスタンス化できませんでした。

ユーザー応答

指定されたクラスがサーバーにデプロイされ、ゲートウェイ・プロパティー・ファイルで正しく指定されていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0617E

VGJ0617E: ゲートウェイ・プロパティー・ファイルに有効な公開ユーザー情報を指定してください。

説明

ゲートウェイ・プロパティー・ファイルで指定された公開ユーザー名またはパスワードが無効です。

ユーザー応答

ゲートウェイ・プロパティー・ファイルの公開ユーザー名およびパスワードの値が正しいことを確認してください。

EGL Java ランタイム・エラー・コード VGJ0700E

VGJ0700E: データベース接続時にエラーが発生しました: %1。

説明

データベースへの接続の試行時にエラーが発生しました。エラー・メッセージの最後の部分はデータベース管理システムからのテキストです。

ユーザー応答

エラー・メッセージを検討し、適切に対処してください。プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0701E

VGJ0701E: SQL 入出力操作の前に、データベース接続を確立する必要があります。

説明

データベース接続を確立する前に SQL 入出力操作が試行されました。

ユーザー応答

SQL 入出力操作は、プログラムがデータベース接続を作成した後にのみ有効になります。プログラムは、プログラム・プロパティに基づいてデフォルト接続を作成し、システム接続関数を実行してデフォルトをオーバーライドできます。プログラム・プロパティおよびデータベース・アクセスの設定については、EGL ヘルプ・ページを検討してください。

EGL Java ランタイム・エラー・コード VGJ0702E

VGJ0702E: SQL 入出力操作 %1 時にエラーが発生しました。%2。

説明

指定された SQL 入出力操作時にエラーが発生しました。メッセージの最後の部分はデータベース管理システムからのテキストです。

ユーザー応答

メッセージを検討し、適切に対処してください。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0703E

VGJ0703E: SQL 入出力操作 %1 のセットアップ時にエラーが発生しました。
%2。

説明

指定された SQL 入出力操作のセットアップ時にエラーが発生しました。

ユーザー応答

メッセージを検討し、適切に対処してください。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0705E

VGJ0705E: データベース %1 の切断中にエラーが発生しました。%2。

説明

指定されたデータベースからの切断の試行中にエラーが発生しました。エラー・メッセージの最後の部分はデータベース管理システムからのテキストです。

ユーザー応答

メッセージを検討し、適切に対処してください。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0706E

VGJ0706E: データベース %1 への接続を設定できません。この接続は存在しません。

説明

指定されたデータベースへの接続の設定の試行時にエラーが発生しました。接続は、トランザクション内のアクティブ・データベース接続にのみ設定できます。

ユーザー応答

データベース名が、トランザクションに設定されたいずれかのアクティブ・データベース接続に一致することを確認してください。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0707E

VGJ0707E: %1 で SQL 入出力シーケンス・エラーが発生しました。

説明

シーケンス・エラーは以下の場合に発生します。

- EGL replace または delete が発生しているが、その前に同じ SQL レコードに対する setupd または update ステートメントがない。
- EGL scan が発生しているが、その前に同じ SQL レコードに対する setupd または setinq ステートメントがない。

メッセージには、プログラムで最後に試行した入出力操作が replace、delete、scan のいずれであるかが示されています。

ユーザー応答

EGL 文の順序が正しいことを確認してください。

プログラム・トレースを使用可能にすると、追加の診断情報が得られる場合があります。

EGL Java ランタイム・エラー・コード VGJ0708E

VGJ0708E: JDBC ドライバー・クラスのロード中にエラーが発生しました: %1

説明

SQL I/O に必要な JDBC ドライバー・クラスのロード中にエラーが発生しました。

ユーザー応答

プロパティ `vgj.jdbc.drivers` で JDBC ドライバー・クラスを正しく指定していることを確認してください。複数必要な場合は、名前をセミコロンで分離してください。また、クラスがクラスパスで見つかることを確認してください。

EGL Java ランタイム・エラー・コード VGJ0709E

VGJ0709E: 文 (%1) が、準備されていない準備済み文を使用しました。

説明

エラー・メッセージで示された準備済み文は存在しません。準備済み文は、EGL の `prepare` 文を呼び出すことによって作成します。

ユーザー応答

準備済み文を使用する前に `prepare` を追加してプログラム・ロジックを訂正してください。

EGL Java ランタイム・エラー・コード VGJ0710E

VGJ0710E: %1 ステートメントが、閉じているか存在しない結果セットを使用しました。

説明

ステートメントが使用する結果セットがオープンされていないか存在しないため、その結果セットは使用できません。

ユーザー応答

無効な結果セットの使用を回避するようにプログラム・ロジックを訂正してください。

EGL Java ランタイム・エラー・コード VGJ0711E

VGJ0711E: データベース %1 への接続時にエラーが発生しました。%2

説明

メッセージで示されたデータベースへの接続を確立できませんでした。

ユーザー応答

このメッセージの「エラー」部分を使用して問題を診断および訂正してください。

EGL Java ランタイム・エラー・コード VGJ0712E

VGJ0712E: デフォルト・データベースに接続できません。 デフォルト・データベースの名前が指定されていませんでした。

説明

デフォルト・データベースの名前が指定されていないため、プログラムから接続できません。

ユーザー応答

デフォルト・データベースの名前はいくつかの方法で指定できます。プロパティー `vgj.jdbc.default.database.programName` (ここで `programName` はプログラムの名前) および `vgj.jdbc.default.database` のいずれかを設定する必要があります。プロパティーの値は、デフォルト・データベースの実際の名前にすることも、デフォルト・データベースの論理名にすることもできます。論理名を使用する場合は、別のプロパティー `vgj.jdbc.database.logicalName` も設定する必要があります。このプロパティーの値は、デフォルト・データベースの実際の名前でなければなりません。

EGL Java ランタイム・エラー・コード VGJ0713E

VGJ0713E: 結果セット %1 がスクロール付きでオープンされていないため、GET が失敗しました。

説明

OPEN ステートメントで SCROLL が指定されていない場合は、GET NEXT のみが許可されます。

ユーザー応答

結果セットが作成される OPEN ステートメントに SCROLL オプションを追加してください。

EGL Java ランタイム・エラー・コード VGJ0750E

VGJ0750E: ファイル %1 の入出力ドライバは作成できませんでした。%2

説明

指定されたファイルに対する入出力ドライバーの作成時に失敗が発生しました。このエラーは以下のときに発生することがあります。

- 指定されたファイルに関連したレコードに対する最初の入出力操作の場合。または
- 指定されたファイルに関連するレコードのシステム変数 `resourceAssociation` に対する最初のアクセス。

メッセージの最後には失敗の理由が示されます。

ユーザー応答

エラー・メッセージを検討し、適切に対処してください。

EGL Java ランタイム・エラー・コード VGJ0751E

VGJ0751E: ファイル %1 の `fileType` プロパティは、Java ランタイム・プロパティ `vgj.ra.fileName.fileType` にはありませんでした。

説明

次のランタイム・プロパティを有効なファイル・タイプに設定する必要があります。

`vgj.ra.fileName.fileType`

fileName

メッセージで指定されたファイルの名前。このファイル名は、EGL レコードに関連付けられた論理ファイル名です。

MQ レコードの場合の値は `mq` です。シリアル・レコードの場合の値は `seqws` です。値のソースは生成時リソース関連パーツ (詳しくはファイルの関連要素、プロパティ `fileType`) です。

ユーザー応答

以下のようにします。

- ランタイム `fileType` プロパティをランタイムのプロパティ・ファイルまたはデプロイメント記述子に追加する。または
- 生成時に `fileType` 値を設定してプログラムを再生成する。
 - リソース関連パーツのファイル名固有関連パーツでプロパティ `fileType` を設定する。
 - 生成時に使用したビルド記述子で、オプション `genProperties` を GLOBAL に設定する。

その他の詳細については、関連要素、Java ランタイム・プロパティ、および環境のセットアップに関する EGL ヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード VGJ0752E

VGJ0752E: リソース関連パーツのファイル %2 に対して無効な **fileType** %1 が指定されました。

説明

次のランタイム・プロパティーを有効なファイル・タイプに設定する必要があります。

```
vgj.ra.fileName.fileType
```

fileName

メッセージで指定されたファイルの名前。このファイル名は、EGL レコードに関連付けられた論理ファイル名です。

MQ レコードの場合の値は `mq` です。シリアル・レコードの場合の値は `seqws` です。値のソースは生成時リソース関連パーツ (詳しくはファイルの関連要素、プロパティー **fileType**) です。

ユーザー応答

以下のようにします。

- ランタイムのプロパティー・ファイルまたはデプロイメント記述子のランタイム **fileType** プロパティーを変更する。または
- 生成時に **fileType** 値をリセットしてプログラムを再生成する。
 - リソース関連パーツのファイル名固有関連要素で、プロパティー **fileType** を変更する。
 - 生成時に使用したビルド記述子で、オプション **genProperties** を GLOBAL に設定する。

その他の詳細については、関連要素、Java ランタイム・プロパティー、および環境のセットアップに関する EGL ヘルプ・ページを参照してください。

EGL Java ランタイム・エラー・コード VGJ0754E

VGJ0754E: レコード長項目には、項目境界で非文字データを分割する値が入っている必要があります。

説明

レコードは可変長です。データを書き出す場合は、レコード長項目は書き込むバイト数を示します。項目が `char` でない限り、データの最終バイトは項目の最終バイトでなければなりません。

ユーザー応答

レコード長項目の値が、項目の最終バイトを指すか `char` 項目内に収まるようにプログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0755E

VGJ0755E: occursItem または lengthItem の値が大きすぎます。

説明

レコードは可変長です。現在レコードに含まれているバイト数より多くのバイト数を書き出す試行が行われました。

ユーザー応答

lengthItem または occursItem の値がレコードのサイズに収まるようにプログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ0770E

VGJ0770E: InitialContext の作成中または java:comp/env 環境の検索中にエラーが発生しました。 エラーは %1 でした。

説明

javax.naming.InitialContext のコンストラクターまたは値 "java:comp/env" による lookup メソッドの呼び出しによって例外がスローされました。プログラムが J2EE 環境設定値にアクセスするには、InitialContext オブジェクトを作成し、"java:comp/env" を検索する必要があります。

ユーザー応答

例外のテキストおよび J2EE 環境の資料を使用して、問題を訂正してください。

EGL Java ランタイム・エラー・コード VGJ0800E

VGJ0800E: %2 への %1 の割り当ては無効です。

説明

デバッガーの使用中にシステム変数を無効な値に設定しようと試みました。

ユーザー応答

システム変数のヘルプ・ページに記述されている、有効な値を選択してください。

EGL Java ランタイム・エラー・コード VGJ0801E

VGJ0801E: %1 は変更できないか、または存在しません。

説明

デバッガーの使用中に、設定できないまたは存在しないシステム変数の値を設定しようと試みました。

ユーザー応答

システム変数のリストおよび各ワードの説明について、ヘルプ・ページを検討してください。

EGL Java ランタイム・エラー・コード VGJ0802E

VGJ0802E: %1 のデバッグ時にエラーが発生しました。%2

説明

ページ・ハンドラーのデバッグ試行時にエラーが発生しました。

ユーザー応答

メッセージの「エラー」部分を使用して、問題を診断および訂正してください。

EGL Java ランタイム・エラー・コード VGJ0901E

VGJ0901E: 日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) が無効です。

説明

日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) が無効です。

ユーザー応答

日付/時刻幅パターンを適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0902E

VGJ0902E: 日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の精度が無効です。

説明

日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の精度が無効です。

ユーザー応答

日付/時刻幅パターンの精度を適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0903E

VGJ0903E: 日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の開始コードが無効です。

説明

日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の開始コードが無効です。

ユーザー応答

日付/時刻幅パターンの開始コードを適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0904E

VGJ0904E: 日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の終了コードが無効です。

説明

日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の終了コードが無効です。

ユーザー応答

日付/時刻幅パターンの終了コードを適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0905E

VGJ0905E: 日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の開始コードまたは終了コードのいずれかが無効です。

説明

日付/時刻幅パターン (timeStamp/interval 項目の長さおよび日付/時刻コンポーネントを宣言する文字ストリング) の開始コードまたは終了コードのいずれかが無効です。

ユーザー応答

日付/時刻幅パターンの開始コードまたは終了コードを適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0906E

VGJ0906E: INTERVAL 値が無効です。

説明

INTERVAL 値が無効です。

ユーザー応答

INTERVAL 値を適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0907E

VGJ0907E: TIMESTAMP 値が無効です。

説明

TIMESTAMP 値が無効です。

ユーザー応答

TIMESTAMP 値を適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0908E

VGJ0908E: TIME 値が無効です。

説明

TIME 値が無効です。

ユーザー応答

TIME 値を適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0909E

VGJ0909E: DATE 値が無効です。

説明

DATE 値が無効です。

ユーザー応答

DATE 値を適切に調整してください。

EGL Java ランタイム・エラー・コード VGJ0910E

VGJ0910E: BLOB または CLOB はメモリー不足です。

説明

BLOB または CLOB はメモリー不足です。

ユーザー応答

BLOB または CLOB のサイズを適切に調整するか、またはこれをファイルに関連付けてください。

EGL Java ランタイム・エラー・コード VGJ0911E

VGJ0911E: loadTable の実行中に内部エラーが発生しました。%1

説明

loadTable の実行中に内部エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0912E

VGJ0912E: loadTable の実行中に SQL エラーが発生しました。 %1

説明

loadTable の実行中に SQL エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0913E

VGJ0913E: loadTable の実行中に入出力エラーが発生しました。 %1

説明

loadTable の実行中に入出力エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0914E

VGJ0914E: VGJSystemCommandProcessing システム・ライブラリーのロード中にエラーが発生しました。 %1

説明

VGJSystemCommandProcessing システム・ライブラリーのロード中にエラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0915E

VGJ0915E: システム・コマンド %1 の実行中にシステム・エラーが発生しました。 コマンドが存在するかどうか、またはコマンドが実行可能かどうかなどについて、システムのパスを検査してください。

説明

システム・コマンドの実行中にエラーが発生しました。

ユーザー応答

コマンドが存在するかどうか、またはコマンドが実行可能かどうかなどについて、システムのパスを検査してください。

EGL Java ランタイム・エラー・コード VGJ0916E

VGJ0916E: loadTable の実行中に内部エラーが発生しました。%1

説明

unloadTable の実行中に内部エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0917E

VGJ0917E: unloadTable の実行中に SQL エラーが発生しました。%1

説明

unloadTable の実行中に SQL エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0918E

VGJ0918E: unloadTable の実行中に入出力エラーが発生しました。%1

説明

unloadTable の実行中に入出力エラーが発生しました。

ユーザー応答

エラーの原因については、拡張エラー・メッセージを参照してください。

EGL Java ランタイム・エラー・コード VGJ0920E

VGJ0920E: ネイティブ C 関数から %1 を戻しているときにエラーが発生しました。

説明

C から EGL に値を戻しているときにエラーが発生しました。

ユーザー応答

これは内部エラーです。

EGL Java ランタイム・エラー・コード VGJ0921E

VGJ0921E: %1 をネイティブ C 関数に渡しているときにエラーが発生しました。

説明

EGL から C に値を渡しているときにエラーが発生しました。

ユーザー応答

これは内部エラーです。

EGL Java ランタイム・エラー・コード VGJ0922E

VGJ0922E: ネイティブ C 関数から戻された値を %1 に割り当てているときにエラーが発生しました。

説明

C から EGL に値を戻しているときにエラーが発生しました。

ユーザー応答

これは内部エラーです。

EGL Java ランタイム・エラー・コード VGJ0923E

VGJ0923E: 値が大きすぎて %1 に収まりません。

説明

数値が smallint または int 受取変数の制限を超えています。

ユーザー応答

smallint または int の範囲外の数値を保管するには、int または decimal 型を使用する変数を再定義する必要があります。

EGL Java ランタイム・エラー・コード VGJ0924E

VGJ0924E: 指定された型同士の間で変換を行うことができません。

説明

ネイティブ関数呼び出し中に、EGL は意味を成すデータ変換を行おうと試みましたが、interval から date、または timestamp から money などの一部の变換はサポートされていません。

ユーザー応答

意図したデータ・タイプが指定されていることを検査してください。

EGL Java ランタイム・エラー・コード VGJ0925E

VGJ0925E: 引数スタックが空です。

説明

ネイティブ C 関数に値を渡すとき、またはネイティブ C 関数から値を戻すときに、空のスタック例外が発生しました。

ユーザー応答

受変数の数が、渡される値の個数または戻される値の個数を超えないことを検査してください。

EGL Java ランタイム・エラー・コード VGJ0926E

VGJ0926E: メモリー割り振りが失敗しました。

説明

現行ネイティブ関数呼び出し内の何かがメモリーの割り振りを要求しましたが、メモリーは使用可能ではありませんでした。

ユーザー応答

原因として考えられることがいくつかあります。たとえば、ご使用のアプリケーションが、構成されたシステムが許可するリソースよりも多くのリソースを要求している、または、オペレーティング・システムの問題でシステムのリブートが必要である、などです。

EGL Java ランタイム・エラー・コード VGJ0927E

VGJ0927E: `datetime` または `interval` 修飾子が無効です。

説明

ネイティブ C 関数で `timestamp` または `interval` 値を受け取るときに、無効な修飾子が使用されました。

ユーザー応答

意図した修飾子が指定されていることを検査してください。

EGL Java ランタイム・エラー・コード VGJ0928E

VGJ0928E: データに対して文字ホスト変数が短すぎます。

説明

ネイティブ C 関数で文字ストリングを受け取るときに、大きさが十分でない文字ホスト変数が使用されました。

ユーザー応答

変数のサイズを検査してください。

EGL Java ランタイム・エラー・コード VGJ0929E

VGJ0929E: ネイティブ C 関数 %1 が見つかりませんでした。

説明

示された C 関数は関数テーブルにありませんでした。

ユーザー応答

この関数のエントリーを関数テーブルに追加して、共用ライブラリーを再作成してください。

EGL Java ランタイム・エラー・コード VGJ0930E

VGJ0930E: ネイティブ C コードで loc_t 構造が不適切に変更されました。

説明

Clob または Blob データ・タイプがネイティブ C 関数に渡されましたが、そのデータ・タイプが受け取られた loc_t C 構造は不適切に変更されました。

ユーザー応答

loc_t 構造の loc_loctype、loc_type、または loc_fname のいずれが、ネイティブ C コードで変更されたのかを検査してください。

EGL Java ランタイム・エラー・コード VGJ0931E

VGJ0931E: ラージ・オブジェクトの処理中にエラーが発生しました。

説明

Clob または Blob データ・タイプに対する内部操作の実行中にエラーが発生しました。

ユーザー応答

これは内部エラーです。

EGL Java ランタイム・エラー・コード VGJ0932E

VGJ0932E: ネイティブ C 関数 %1 が、呼び出し側関数によって予期される正しい値の数を戻しませんでした。

説明

関数は式の一部として呼び出された場合、複数の値を返します。そうではない場合は、戻される変数の数は、受取変数の数と異なります。

ユーザー応答

正しい関数が呼び出されたことを検査します。ネイティブ C 関数のロジックを、特にその関数によって戻される値について検討し、常に、予期される個数の値を関数が戻すようにしてください。

EGL Java ランタイム・エラー・コード VGJ0933E

VGJ0933E: この操作の `interval` に互換性がありません。

説明

`interval` 値の組み合わせの中に意味のないものがあり、これらは許可されません。

ユーザー応答

渡されるまたは戻される `interval` データ・タイプについて互換性を検討してください。

EGL Java ランタイム・エラー・コード VGJ1000E

VGJ1000E: %1 が失敗しました。 メソッドの起動または %2 と呼ばれるフィールドのアクセスで未処理エラーが発生しました。 エラー・メッセージは %3 です。

説明

Java アクセス関数でエラーが発生しました。 `Exception` がスローされて `try` 文内で関数が呼び出されていなかったか、または `VGVar.handleSysLibraryErrors` が 0 であるか、`Exception` 以外の結果 (`Error` など) がスローされました。

ユーザー応答

エラー・メッセージの情報を使用して問題を訂正してください。なんらかの `Exception` が スローされた場合は、`try` 文内で Java アクセス関数を呼び出すか、`VGVar.handleSysLibraryErrors` を 1 に設定してから Java アクセス関数を呼び出すことによってエラーを処理するように、プログラム・ロジックを変更してください。

EGL Java ランタイム・エラー・コード VGJ1001E

VGJ1001E: %1 が失敗しました。 %2 は ID でないか NULL オブジェクトの ID です。

説明

Java アクセス関数でエラーが発生しました。 この ID は NULL 以外のオブジェクトを参照していないため、使用できません。

ユーザー応答

NULL 以外のオブジェクトの ID を使用してください。

EGL Java ランタイム・エラー・コード VGJ1002E

VGJ1002E: %1 が失敗しました。名前が %2 の **public** メソッド、フィールド、またはクラスが存在しないかロードできない、またはパラメーターの数か型が誤っています。エラー・メッセージは %3 です。

説明

Java アクセス関数が使用するメソッド、フィールド、またはクラスが見つかりませんでした。

ユーザー応答

以下のようにします。

- ターゲットが **public** メソッド、フィールド、またはクラスであることを確認する。
- メソッド、フィールド、またはクラスの名前が正しいことを確認する。クラス名は、それぞれのパッケージ名で修飾する必要があります。
- 問題がクラスの欠落であり、名前が正しい場合は、クラスが含まれているディレクトリーまたはアーカイブが Java クラスパスにあることを確認してください。
- メソッドが欠落していることが問題であり、名前が正しい場合は、パラメーターの型および数が正しいことを確認する。Java アクセス関数に渡す値を、メソッドによって予期される値と比較してください。

EGL Java ランタイム・エラー・コード VGJ1003E

VGJ1003E: %1 が失敗しました。EGL の値の型が、%2 の Java で予期される型と一致しません。エラー・メッセージは %3 です。

説明

Java アクセス関数に渡した値の型が正しくありません。

ユーザー応答

フィールドに割り当てる値、およびメソッドとコンストラクターに渡すパラメーターは適切な型でなければなりません。型相互間の変換が Java で有効な場合は、完全一致は必要ではありません。たとえば、スーパークラスの代わりにサブクラスを使用したり、小さいプリミティブ型 (**short** など) の代わりに大きいプリミティブ型 (**int** など) を使用したりすることができます。

EGL Java ランタイム・エラー・コード VGJ1004E

VGJ1004E: %1 が失敗しました。ターゲットは、NULL を戻したメソッド、値を戻さないメソッド、または値が NULL のフィールドです。

説明

Java アクセス関数が操作の結果として NULL 以外のオブジェクトを予期していましたが、そのようなオブジェクトを受け取りませんでした。

ユーザー応答

NULL を戻すか値を戻さない場合があるメソッドを呼び出すには、javaStore を使用するか、java システム関数を使用して結果を項目に割り当てないようにしてください。 NULL の場合があるフィールドの値を取得するには、javaStoreField を使用します。

EGL Java ランタイム・エラー・コード VGJ1005E

VGJ1005E: %1 が失敗しました。 戻り値が戻り項目の型に一致しません。

説明

型が一致しないため、Java アクセス関数によって戻された値を戻り項目に割り当てることができません。

ユーザー応答

適切な型の戻り項目を使用するようにプログラム・ロジックを変更してください。

EGL Java ランタイム・エラー・コード VGJ1006E

VGJ1006E: %1 が失敗しました。 NULL にキャストする引数のクラス %2 をロードできませんでした。 エラー・メッセージは %3 です。

説明

Java アクセス関数に渡した引数のクラスが見つかりませんでした。

ユーザー応答

以下のようにします。

- ・ クラスの名前が正しいことを確認する。クラス名はパッケージ名で修飾する必要があります。
- ・ 名前が正しい場合は、クラスが含まれているディレクトリーまたはアーカイブが Java クラスパスにあることを確認してください。

EGL Java ランタイム・エラー・コード VGJ1007E

VGJ1007E: %1 が失敗しました。 メソッドまたは指定のフィールド %2 に関する情報を取得できなかったか、final 宣言されたフィールドに値を設定しようとする試みがなされました。 エラー・メッセージは %3 です。

説明

メソッドまたはフィールドに関する情報を取得するときに SecurityException または IllegalAccessException がスローされたか、final 宣言されたフィールドに値を設定し

ようとする試行がなされました。 `final` 宣言されたフィールドは変更できません。

ユーザー応答

以下のようにします。

- 値の設定時に問題が発生した場合は、コードが `final` 宣言されたフィールドに値を設定しようと試みないようにプログラム・ロジックを変更するか、代わりに、フィールドの宣言を変更する。
- 問題が、情報へのアクセスに関するものである場合は、システム管理者に連絡して Java 仮想マシンのセキュリティ・ポリシー・ファイルを更新し、プログラムに必要なアクセス権を指定する。通常、管理者は `ReflectPermission "suppressAccessChecks"` を認可する必要があります。

EGL Java ランタイム・エラー・コード VGJ1008E

VGJ1008E: %1 が失敗しました。 %2 はインターフェースまたは抽象クラスであるため、コンストラクターを呼び出すことはできません。

説明

インターフェースまたは抽象クラスのコンストラクターを呼び出すことはできません。

ユーザー応答

抽象クラスでないクラスのコンストラクターを呼び出すようにプログラム・ロジックを変更してください。

EGL Java ランタイム・エラー・コード VGJ1009E

VGJ1009E: %1 が失敗しました。 メソッドまたはフィールド %2 が `static` ではありません。 クラス名の代わりに ID を使用してください。

説明

`static` 宣言されていない場合は、メソッドまたはフィールドはクラス自体ではなく、クラスの特定のインスタンスのみに存在します。この場合は、オブジェクトの ID を使用する必要があります。

ユーザー応答

クラス名の代わりに ID を使用するようにプログラム・ロジックを変更してください。

EGL Java ランタイム・エラー・コード VGJ1101W

VGJ1101W: 進んでいる方向にはこれ以上行はありません。

説明

エンド・ユーザーが最終行を超えてナビゲートしようとした。

EGL Java ランタイム・エラー・コード VGJ1148E

VGJ1148E: アクション・フィールド ”%1” が存在しません。

説明

現在の OnEvent アクションが、見つけられないフィールドを参照しています。

ユーザー応答

フィールドが現在の書式に存在することを検証してください。

EGL Java ランタイム・エラー・コード VGJ1149E

VGJ1149E: 別の行を挿入できません。 入力配列がフルです。

説明

配列データを保持するために使用される変数に、別の行を保持するためのスペースがありません。

ユーザー応答

EGL 変数のストレージ・サイズを増やしてください。

EGL Java ランタイム・エラー・コード VGJ1150E

VGJ1150E: 配列 ”%1” が見つかりません。

説明

指定された配列が ConsoleForm で見つかりませんでした。

ユーザー応答

配列が ConsoleForm と EGL プログラムで正しく定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1151E

VGJ1151E: プロンプト結果変数への割り当てが失敗しました。

説明

プロンプト結果変数への割り当てが失敗しました。

ユーザー応答

結果変数がプロンプト・アクションからの結果を保持できることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1152E

VGJ1152E: 画面配列フィールド '%1' のサイズが正しくありません。

説明

示された画面配列フィールドのサイズが正しくありません。

ユーザー応答

画面配列の定義と、EGL プログラムでのその画面配列の使用法を検証してください。

EGL Java ランタイム・エラー・コード VGJ1153E

VGJ1153E: DrawBox パラメーターが範囲外です。

説明

DrawBox パラメーターが、現行の画面/ウィンドウの寸法内に収まりません。

ユーザー応答

drawbox 関数のパラメーターと現行ウィンドウの寸法を検証してください。

EGL Java ランタイム・エラー・コード VGJ1154E

VGJ1154E: 表示座標がウィンドウ境界の外側にあります。

説明

表示座標がウィンドウ境界の外側にあります。

ユーザー応答

使用される座標がウィンドウのサイズ内に収まることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1155E

VGJ1155E: キー名 '%1' の形式に誤りがあります。

説明

指定されたキー名は、キー命名規則に従っていません。

ユーザー応答

EGL キー命名規則に従うようにキー名を再度書き込んでください。

EGL Java ランタイム・エラー・コード VGJ1156E

VGJ1156E: ピクチャーが存在するため、この編集機能を使用できません。

説明

ピクチャー属性によって、このフィールドについて編集機能が制限されます。

ユーザー応答

望ましい結果が得られるように代替の編集キーおよびアクションを使用してください。

EGL Java ランタイム・エラー・コード VGJ1157E

VGJ1157E: ウィンドウ "%1" を見つけれられません。

説明

ウィンドウを見つけることができませんでした。

ユーザー応答

ウィンドウが正しく定義および使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1158E

VGJ1158E: 新規ウィンドウ位置/サイズの値が無効です。

説明

指定された位置/サイズ値が、現在の表示環境で無効です。

ユーザー応答

ウィンドウ位置/サイズが現在の表示環境で有効であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1159E

VGJ1159E: コマンド・スタックの同期が取れていません。

説明

OnEvent 文節で実行されている文が原因で、EGL の同期が取れていません。

ユーザー応答

OnEvent ブロック文内の文/関数呼び出しの使用法を検証してください。

EGL Java ランタイム・エラー・コード VGJ1160E

VGJ1160E: コンソール UI ライブラリーが初期化されていません。

説明

コンソール UI ライブラリーを初期化する前に、これを使用しようとする試みがなされました。

ユーザー応答

コンソール UI ステートメントシーケンスが有効であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1161E

VGJ1161E: 構成に対して正しくないフィールド・タイプ。

説明

コンソール・フィールドに指定されたフィールド・タイプが、構成クエリー操作で無効です。

ユーザー応答

コンソール・フィールドのフィールド・タイプが構成クエリー操作で有効であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1162E

VGJ1162E: 変数リストを使用して `ConstructQuery` を呼び出すことができません。

説明

構成クエリー操作が、変数リストを使用して呼び出されました。

ユーザー応答

構成クエリー操作が正しく呼び出されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1163E

VGJ1163E: 不可視のメニュー項目を使用不可にできません。

説明

不可視のメニュー項目を非表示にしようとする試行は、無効な操作です。

ユーザー応答

使用不可にする正しいメニュー項目が不可視のメニュー項目でないことを検証してください。

EGL Java ランタイム・エラー・コード VGJ1164E

VGJ1164E: 編集アクションが失敗しました。

説明

指定された編集アクションの実行が失敗しました。

ユーザー応答

コンソール・フィールドが正しく定義されていること、および実行される編集アクションが有効な操作であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1165E

VGJ1165E: ホット・キー・アクションの実行中にエラーが発生しました。

説明

ホット・キー操作の実行が失敗しました。

ユーザー応答

指定されたホット・キーが有効であること、および文ブロックも有効であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1166E

VGJ1166E: 終了するアクティブ・コマンドがありません。

説明

存在しない現在のコマンドを終了しようとしてしました。

ユーザー応答

exit ステートメントが正しいコンテキストで使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1167E

VGJ1167E: 続行するアクティブ・コマンドがありません。

説明

現在のコマンドを続行しようとしてしました。

ユーザー応答

continue 文が正しいコンテキストで使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1168E

VGJ1168E: 致命的エラー: %1

説明

致命的なランタイム・エラーが発生しました。

ユーザー応答

Console UI 文が正しいコンテキストおよびシーケンスで使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1169E

VGJ1169E: フィールド ”%1” が存在しません。

説明

指定されたコンソール・フィールドが存在しません。

ユーザー応答

コンソール・フィールドがコンソール書式に正しく定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1170E

VGJ1170E: 画面配列フィールド ”%1” は配列ではありません。

説明

コンソール書式にある参照されるコンソール・フィールドが配列ではありません。

ユーザー応答

コンソール・フィールドが配列として定義されていることを検証してください。正しいコンソール・フィールドが参照されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1171E

VGJ1171E: フィールド ”%1” が見つかりません。

説明

示されたコンソール・フィールドが見つかりませんでした。

ユーザー応答

コンソール・フィールドがコンソール書式に正しく定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1172E

VGJ1172E: ウィンドウなしで `ConsoleField` を作成することはできません。

説明

コンソール書式/ウィンドウ・コンテキスト外でコンソール・フィールドを作成しようとしていました。

ユーザー応答

コンソール書式およびコンソール・フィールドの定義が正しいことを検証してください。

EGL Java ランタイム・エラー・コード VGJ1173E

VGJ1173E: 配列フィールド・カウントの不一致。

説明

指定されたコンソール UI 配列フィールドは、参照される EGL 配列と一致しません。

ユーザー応答

ConsoleField と配列定義を検証してください。openui ステートメントで正しい EGL 配列変数が使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1174E

VGJ1174E: 書式 "%1" が存在しません。

説明

指定されたコンソール書式が存在しません。

ユーザー応答

指定されたコンソール書式が定義されていて、正しいコンテキストで使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1175E

VGJ1175E: 書式 "%1" がウィンドウ "%2" に収まりません。

説明

書式の寸法が、現行ウィンドウの寸法に収まりません。

ユーザー応答

書式定義またはウィンドウ定義のいずれかの寸法を変更してください。

EGL Java ランタイム・エラー・コード VGJ1176E

VGJ1176E: フィールド・リストが一致しません。

説明

指定されたフィールド・リストには、提供された変数リストと同じ数の項目が含まれていません。

ユーザー応答

同数のフィールドと変数が指定されるように openUI を変更してください。

EGL Java ランタイム・エラー・コード VGJ1177E

VGJ1177E: 書式 "%1" が使用中です。

説明

現在、書式参照が別のコンテキストですでに使用されています。

ユーザー応答

EGL プログラム・ロジックが書式を一度に 1 回のみ使用することを検証してください。

EGL Java ランタイム・エラー・コード VGJ1178E

VGJ1178E: 書式名 "%1" はすでに使用されています。

説明

書式の定義が理由で、書式名が競合しています。

ユーザー応答

固有の書式名を使用するように書式定義を変更してください。

EGL Java ランタイム・エラー・コード VGJ1179E

VGJ1179E: 書式 "%1" がオープンしていません。

説明

定義されていない書式オブジェクトを参照しようとしてしました。

ユーザー応答

指定された書式が正しく定義され、有効な ConsoleUI 文内で使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1180E

VGJ1180E: ウィンドウなしで ConsoleForm を作成することはできません。

説明

有効なウィンドウ参照のない ConsoleForm を作成しようとしてしました。

ユーザー応答

ConsoleForm が正しく定義され、有効な ConsoleUI 文内で使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1181E

VGJ1181E: `KeyObject.getChar()` は仮想キーに使用できません。

説明

consoleUI は、仮想キーに `KeyObject.getChar()` を使用できません。

ユーザー応答

仮想キー定義用にストリングを構成するように EGL プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ1182E

VGJ1182E: `KeyObject.getCookedChar()` は仮想キーに使用できません。

説明

ConsoleUI は、仮想キーに `KeyObject.getCookedChar()` を使用できません。

ユーザー応答

ストリングを使用して仮想キーを定義するように EGL プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ1183E

VGJ1183E: プロンプト結果ストリングの検索が失敗しました。

説明

プロンプト結果ストリングの検索が失敗しました。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1184E

VGJ1184E: ヘルプ・メッセージ・キー "%1"が、リソース・バンドル "%2"にありません。

説明

指定されたメッセージ・ヘルプ・ファイル内でヘルプ・メッセージ・キーが見つかりませんでした。

ユーザー応答

正しいヘルプ・メッセージ・キーとヘルプ・メッセージ・ファイルが使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1185E

VGJ1185E: 正しくない配列添え字。

説明

無効な配列エレメントを参照しようとしてしました。

ユーザー応答

プログラム・ロジックが、定義された配列のサイズ内の配列エレメントを参照していることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1186E

VGJ1186E: コンソール UI ライブラリーを初期化できません。

説明

プログラムの開始時に、コンソール UI ライブラリーを初期化できませんでした。

ユーザー応答

サポートされる表示環境およびプラットフォームでプログラムが使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1187E

VGJ1187E: 内部エラー

説明

ConsoleUI 内部エラーが発生しました。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1188E

VGJ1188E: 割り込み信号を受け取りました。

説明

割り込み信号を受け取りました。

EGL Java ランタイム・エラー・コード VGJ1189E

VGJ1189E: アクセラレーターなしの不可視のメニュー項目は許可されません。

説明

アクセラレーター・キーのない不可視のメニュー項目を作成しようとしてしました。

ユーザー応答

不可視のメニュー項目用にアクセラレーターを定義するようにメニュー項目定義を変更してください。

EGL Java ランタイム・エラー・コード VGJ1190E

VGJ1190E: ウィンドウなしで `ConsoleLabel` を作成することはできません。

説明

`ConsoleLabel` の作成中に、有効なウィンドウ参照を見つけることができませんでした。

ユーザー応答

コンソール書式でコンソール・ラベルが正しく定義されていること、および EGL プログラムでコンソール書式が正しく使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1191E

VGJ1191E: メニュー項目 `%1` がウィンドウに適合しません。

説明

指定されたメニュー項目は、大きすぎて現行のアクティブ・ウィンドウに適合しません。

ユーザー応答

現行のアクティブ・ウィンドウの幅の寸法よりもメニュー項目の名前がより小さくなるように、メニュー項目を変更してください。

EGL Java ランタイム・エラー・コード VGJ1192E

VGJ1192E: メニュー項目 `”%1”` が存在しません。

説明

指定されたメニュー項目が見つからなかったか、または存在しません。

ユーザー応答

参照されるメニュー項目が定義されており、現在のメニュー・インスタンスに追加されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1193E

VGJ1193E: メニューのニーモニックが競合しています (キー=%1)。

説明

現在のメニュー項目定義では、結果としてニーモニックの競合が起こります。

ユーザー応答

accelerator/OnEvent キーが競合しないようにメニュー項目を変更してください。

EGL Java ランタイム・エラー・コード VGJ1194E

VGJ1194E: アクティブな書式がありません。

説明

コンソール UI にはアクティブな書式参照がありません。

ユーザー応答

書式が定義および表示されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1195E

VGJ1195E: アクティブな書式が DISPLAY ARRAY が必要です。

説明

存在しない現行のアクティブな書式から配列を表示しようとしてしました。

ユーザー応答

配列を表示しようとする前に、配列を持つ書式が定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1196E

VGJ1196E: アクティブな書式が READ ARRAY が必要です。

説明

存在しないアクティブな書式から配列を読み取ろうとしてしました。

ユーザー応答

アクティブな書式から配列を読み取ろうとする前に、書式が定義され、アクティブになっていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1197E

VGJ1197E: 現行コマンドなしではイベント・ループを開始できません。

説明

現行コマンドなしではイベント・ループを開始できません。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1198E

VGJ1198E: blob エディターが指定されていません。

説明

blob を編集しようとしたますが、blob エディターが指定されていませんでした。

ユーザー応答

適切なエディターを blob コンソール・フィールドに定義してください。

EGL Java ランタイム・エラー・コード VGJ1199E

VGJ1199E: 内部エラー: 形式オブジェクトがありません。

説明

内部エラー: 形式オブジェクトがありません。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1200E

VGJ1200E: ヘルプ・ファイルが指定されていません。

説明

ヘルプ要求を受け取りましたが、ヘルプ・ファイルが指定されていませんでした。

ユーザー応答

EGL プログラムで有効なヘルプ・ファイルを定義してください。

EGL Java ランタイム・エラー・コード VGJ1201E

VGJ1201E: ヘルプ・メッセージが指定されていません。

説明

ヘルプ要求を受け取りましたが、ヘルプ・メッセージが指定されていませんでした。

ユーザー応答

ヘルプ・メッセージを提供するように EGL プログラムを変更してください。

EGL Java ランタイム・エラー・コード VGJ1202E

VGJ1202E: メニューがレイアウトされていません。

説明

表示されていないメニュー関数を使用しようとしてしました。

ユーザー応答

メニューが表示された後にメニュー関数を使用することを検証してください。

EGL Java ランタイム・エラー・コード VGJ1203E

VGJ1203E: 現行画面配列がありません。

説明

現行画面配列を使用する参照を行おうとしてしましたが、現行画面配列は存在しません。

ユーザー応答

現在のアクティブな書式に画面配列が含まれていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1204E

VGJ1204E: 可視のメニュー項目がありません。

説明

メニューの構成中に、どのメニュー項目も可視ではないことが検出されました。

ユーザー応答

少なくとも 1 つのメニュー項目が可視で、表示可能になるように、メニュー作成を変更してください。

EGL Java ランタイム・エラー・コード VGJ1205E

VGJ1205E: 新規ウィンドウの名前が NULL でした。

説明

ウィンドウの宣言が NULL でした。

ユーザー応答

ウィンドウを宣言するときに、ウィンドウ名を指定してください。

EGL Java ランタイム・エラー・コード VGJ1206E

VGJ1206E: NULL ウィンドウを開こうとしました。

説明

存在しないウィンドウか、または空のウィンドウを開こうとしました。

ユーザー応答

open window ステートメントが有効なウィンドウ参照を使用していることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1207E

VGJ1207E: プロンプトでの例外の発生。

説明

プロンプトの実行中に例外が発生しました。

ユーザー応答

プロンプト OnEvent ステートメントブロックが正しいことを検証してください。

EGL Java ランタイム・エラー・コード VGJ1208E

VGJ1208E: 終了信号を受信しました。

説明

終了信号を受信しました。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1209E

VGJ1209E: アクティブ画面配列がありません。

説明

現在のアクティブな書式には画面配列が含まれていません。

ユーザー応答

プログラム・ロジックが画面配列定義を含む書式を使用していることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1210E

VGJ1210E: アクティブな書式がありません。

説明

現在のコンソール UI セッションにはアクティブな書式インスタンスが含まれていません。

ユーザー応答

書式を参照する前に、その書式が定義および表示されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1211E

VGJ1211E: メニューを現在の項目にスクロールできません。

説明

メニュー・カーソルをメニュー項目に移動する試みは失敗しました。

ユーザー応答

メニュー・ロジックによってメニュー・カーソルが正しいメニュー項目に移動されていることを検証してください。メニュー項目が使用不可になっていないことを検証してください。

EGL Java ランタイム・エラー・コード VGJ1212E

VGJ1212E: 不明な属性 '%1'。

説明

指定された属性は認識されませんでした。

ユーザー応答

現在のコンソール UI コンテキストについて属性が正しいことを検証してください。

EGL Java ランタイム・エラー・コード VGJ1213E

VGJ1213E: フィールド '%1' にエラーがあります。

説明

フィールドへの入力が正しくありません。

ユーザー応答

入力されたデータが、フィールドのデータ・タイプまたは形式プロパティと一致することを検証してください。

EGL Java ランタイム・エラー・コード VGJ1214E

VGJ1214E: 十分な数の変数が提供されませんでした。

説明

コンソール書式にバインドするために十分な変数が openUI ステートメントに提供されませんでした。

ユーザー応答

openUI ステートメント用の変数をより多くリストするように EGL プログラムを変更するか、コンソール・フィールドの数を制限するように openUI 文を変更してください。

EGL Java ランタイム・エラー・コード VGJ1215E

VGJ1215E: ウィンドウ名 ”%1” はすでに使用されています。

説明

新たに定義されたウィンドウ名は、別のウィンドウによってすでに使用されています。

ユーザー応答

他のウィンドウ名と競合しないようにウィンドウの名前を変更してください。

EGL Java ランタイム・エラー・コード VGJ1216E

VGJ1216E: ヘルプ画面用には、ウィンドウ・サイズが小さすぎます。

説明

小さすぎる表示環境への、ヘルプ画面の表示の試み。

ユーザー応答

表示環境のサイズを調整してください。

EGL Java ランタイム・エラー・コード VGJ1217W

VGJ1217W: 進んでいる方向にはこれ以上フィールドはありません。

説明

フィールド・リストの終わりを越えるカーソルの移動の試み。

EGL Java ランタイム・エラー・コード VGJ1218E

VGJ1218E: 画面配列 '%1' の内容が無効です。

説明

画面上の arrayDictionary には、各列に 1 つずつエントリーが含まれています。すべてのエントリーは、オブジェクト・タイプが同じでなければならず、ConsoleField、ConsoleFields の配列、およびすべての配列 (存在する場合) は、いずれも含まれるエレメントの数が同じでなければなりません。

画面上の arrayDictionary の宣言が正しいか調べて、訂正してください。

EGL Java ランタイム・エラー・コード VGJ1219E

VGJ1219E: 画面配列 '%1' にはセグメント化されたフィールド '%2' を含めることはできません。

説明

consoleField セグメント・プロパティを画面上の arrayDictionary で使用される任意の consoleField に設定しないでください。

ユーザー応答

画面上の arrayDictionary に含まれているすべての consoleField のセグメント・プロパティを除去してください。

EGL Java ランタイム・エラー・コード VGJ1220E

VGJ1220E: 画面配列 '%1' はデータ配列と互換性がありません。

説明

画面上の arrayDictionary は各列に 1 つずつエントリーを含み、動的配列はその arrayDictionary にバインドします。arrayDictionary にある各レコードには、画面上の arrayDictionary の列と同数のフィールドがなければならず、各フィールドは対応する consoleField と割り当て整合性がなければなりません。

ユーザー応答

画面上の arrayDictionary またはその arrayDictionary に結合する動的配列のレコードが正確かどうかを検証し、訂正してください。

EGL Java ランタイム・エラー・コード VGJ1221E

VGJ1221E: 同一名 '%1' を持つ複数のフィールドが検出されました。

説明

`consoleForm` は同一名のフィールドを複数持つことはできません。

ユーザー応答

`consoleForm` 宣言を訂正してください。

EGL Java ランタイム・エラー・コード VGJ1222E

VGJ1222E: コンソール・フィールド '`%1`' の長さが無効です。

説明

`consoleField` の長さは 0 より大きくなければなりません。

ユーザー応答

`consoleField` 宣言を訂正してください。

EGL Java ランタイム・エラー・コード VGJ1223E

VGJ1223E: [`%1`, `%2`] のラベルが使用可能なスペースと一致しません。

説明

ラベルはウィンドウの境界内に完全に収まる大きさでなければなりません。

ユーザー応答

ラベルまたはウィンドウの位置またはサイズを訂正してください。

EGL Java ランタイム・エラー・コード VGJ1224E

VGJ1224E: (`%2`, `%3`) のフィールド '`%1`' セグメントが使用可能なスペースに適合しません。

説明

`consoleField` は、ウィンドウの境界内に完全に収まる大きさでなければなりません。

ユーザー応答

`consoleField` またはウィンドウの位置またはサイズを訂正してください。

EGL Java ランタイム・エラー・コード VGJ1225E

VGJ1225E: Prompt スtringがアクティブなウィンドウに対して長すぎます。

説明

ユーザー入力を促すメッセージはアクティブなウィンドウに収まる長さでなければなりません。プロンプトの `isChar` プロパティが `no` に設定されている場合、ユ

ユーザー応答のために必要なスペースも考慮しなければなりません。

ユーザー応答

プロンプトまたはウィンドウの宣言を訂正してください。

EGL Java ランタイム・エラー・コード VGJ1226E

VGJ1226E: OpenUI 配列の引数が無効です。

説明

画面上の `arrayDictionary` には、表示の各列に 1 つずつエントリーが含まれています。すべてのエントリーは、オブジェクト・タイプが同じでなければならず、`ConsoleField`、または `ConsoleFields` の配列、およびすべての配列 (存在する場合) は、いずれも含まれるエレメントの数が同じでなければなりません。

ユーザー応答

画面上の `arrayDictionary` の宣言が正しいかどうか調べて、訂正してください。

EGL Java ランタイム・エラー・コード VGJ1227E

VGJ1227E: OpenUI フィールドの引数が無効です。

説明

openUI ステートメントでは、`consoleField` を指定するか、`consoleField` 名前フィールドの値を含むストリングを指定することにより、`consoleField` のリストを指定することができます。この場合、リストに無効な値がインクルードされています。

ユーザー応答

openUI ステートメントの値を調べて、訂正してください。

EGL Java ランタイム・エラー・コード VGJ1228E

VGJ1228E: プロンプト・ステートメントにバインドできる変数は 1 つのみです。

説明

プロンプトは、ユーザー応答を受信する 1 つの変数にのみバインドしなければなりません。他の種類のバインディングが試行されています。

ユーザー応答

openUI ステートメントを調べて、訂正してください。

EGL Java ランタイム・エラー・コード VGJ1229E

VGJ1229E: コンソール・フィールド '`%1`' のデータ・バインディングを決定できませんでした。

説明

openUI ステートメントが consoleFields を他の変数にバインドしていない場合、各 consoleField の **binding** プロパティの値が使用されます。しかし、この場合は、**binding** プロパティがありませんでした。

ユーザー応答

consoleField または **openUI** ステートメントでバインディングを追加してください。

EGL Java ランタイム・エラー・コード VGJ1290E

VGJ1290E: %1 は、Blob/Clob 関数の有効なパラメーターではありません。

説明

Blob/Clob 関数の処理中にエラーが発生しました。エラーの原因は、メッセージ挿入で説明されています。

ユーザー応答

メッセージ挿入の内容に基づいて適切なアクションを行ってください。

EGL Java ランタイム・エラー・コード VGJ1301E

VGJ1301E: レポートの記入エラー %1。

説明

レポートの記入エラー。レポートに提供されたデータが正しくありません。動的配列レコードのフィールド名がレポートのフィールド名に一致しないこと、接続が存在しないこと、または SQL ステートメントが無効であることが原因である可能性があります。

ユーザー応答

レコードの動的配列を使用する場合、レポート設計で定義されたフィールド名が、レコード内のエレメントの名前と一致することを確認してください。SQL ステートメントを使用する場合、SQL が有効であることを確認してください。接続を使用する場合、接続が確立されていて、接続名が正しいことを確認してください。さらに、reportDesignFile に指定されたパス名が有効であること、およびファイルが存在することも確認してください。

EGL Java ランタイム・エラー・コード VGJ1302E

VGJ1302E: レポートのエクスポート・エラー %1。

説明

レポートのエクスポート・エラー。レポートを、指定された形式にエクスポートできませんでした。

ユーザー応答

パス名が正しいこと、記入されたレポート・オブジェクトが指定されたロケーションに存在し、reportDestinationFile フィールドに正しく割り当てられていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ1303E

VGJ1303E: レポートの動的アクセス・エラー。コンテンツが見つかりません。%1

説明

フィールド名がレコードの動的配列に存在しません。

ユーザー応答

レポート設計のフィールド名と、EGL プログラムで使用するレコードのフィールド名が一致することを確認してください。

EGL Java ランタイム・エラー・コード VGJ1304E

VGJ1304E: 正しくない接続名

説明

接続名が無効です。

ユーザー応答

接続が有効な EGL 接続であること、および接続に名前を割り当てるために defineDatabaseAlias 関数が使用されたことを確認してください。

EGL Java ランタイム・エラー・コード VGJ1305E

VGJ1305E: 指定された名前 %1 の接続は存在しません。

説明

示された接続名の接続は存在しません。

ユーザー応答

次の文が EGL プログラムに存在することを確認してください。有効なパラメータをもつ接続関数と defineDatabaseAlias によって接続に名前が指定されます。

EGL Java ランタイム・エラー・コード VGJ1306E

VGJ1306E: EGL とレポートの型マッピングが正しくありません。マッピング・テーブルを検査してください。

説明

レポート設計フィールドと EGL プログラムのデータ・タイプの間には型のミスマッチがあります。

ユーザー応答

資料に述べられているとおりに型に互換性があることを確認してください。たとえば、EGL char 型の場合、設計ファイルに、java.lang.String として定義されたクラスが必要で、EGL int 型の場合、設計ファイルに、java.lang.Integer として定義されたフィールド・クラスが必要です。

EGL Java ランタイム・エラー・コード VGJ1401E

VGJ1401E: 位置 (%2,%3) のフィールド ”%1” は書式内にありません。

説明

所定の位置にある、指定されたフィールドは書式内にありません。

ユーザー応答

書式とフィールドが正しく定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1402E

VGJ1402E: フィールド ”%1” が ”%2”に重なり合っています。

説明

2 つのフィールドが、そのサイズと位置が理由で重なり合っています。

ユーザー応答

フィールドのサイズおよび位置座標を調整してください。

EGL Java ランタイム・エラー・コード VGJ1403E

VGJ1403E: 内部エラー: 書式グループを判別できません。

説明

内部エラー: 書式グループを判別できません。

ユーザー応答

書式と書式グループが正しく定義されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1404E

VGJ1404E: 書式 ”%1” はどの浮動域にも適合しません。

説明

書式がどの浮動域にも適合しません。

ユーザー応答

書式を浮動域で正しく表示できることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1405E

VGJ1405E: フィールド ”%1” の座標が無効です。

説明

フィールド座標が無効です。

ユーザー応答

指定されたフィールド座標が書式で有効であることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1406E

VGJ1406E: 印刷関連を取得できません。

説明

印刷関連のセットアップの試みが失敗しました。

ユーザー応答

印刷装置関連が正しくセットアップされていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1407E

VGJ1407E: 適切な印刷装置サイズが存在しません。

説明

適切な印刷装置サイズが存在しません。

ユーザー応答

EGL Java ランタイム・エラー・コード VGJ1408E

VGJ1408E: プリンター ’%1’ が検出できませんでした。¥n以下のプリンターが使用可能です。¥n%2

説明

ユーザーは、特定のプリンター装置に印刷しようとしたますが、そのプリンターをシステムで検出できませんでした。

ユーザー応答

環境のプリンター構成を調べてください。プリンターが存在することを確認するか、別のプリンターに印刷してください。

EGL Java ランタイム・エラー・コード VGJ1409E

VGJ1409E: 書式のためのディスプレイ装置がありません。

説明

書式のためのディスプレイ装置がありません。

ユーザー応答

EGL プログラムが、サポートされるプラットフォームおよび表示環境で実行されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1410E

VGJ1410E: 表示された書式には互換性がある装置サイズがありません。

説明

表示された書式には互換性がある装置サイズがありません。

ユーザー応答

EGL プログラムが、サポートされるプラットフォームおよび表示環境で実行されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1411E

VGJ1411E: ヘルプ書式クラス "%1" が存在しません。

説明

ヘルプ書式クラスを参照しようとしたますが、これは存在しません。

ユーザー応答

ヘルプ書式クラスが定義され、正しく参照されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1412E

VGJ1412E: 不明な属性 '%1'。

説明

指定された属性は認識されませんでした。

ユーザー応答

正しい属性名が使用されていることを検証してください。

EGL Java ランタイム・エラー・コード VGJ1414E

VGJ1414E: '%1' からヘルプ書式を作成できません。

説明

表示のためのヘルプ書式が作成できませんでした。

ユーザー応答

適切な書式グループがアプリケーションに存在し、適切に生成されていることを確認してください。

EGL Java ランタイム・エラー・コード VGJ1415E

VGJ1415E: 内部エラー: %1

説明

内部エラーが発生しました。

ユーザー応答

IBM 技術サポートに連絡してください。

EGL Java ランタイム・エラー・コード VGJ1416E

VGJ1416E: 使用可能なプリンターがありません。

説明

ユーザーが印刷しようとしたが、システムにプリンター装置がありません。

ユーザー応答

代わりにファイルに出力するか、環境にプリンター装置を構成するかしてください。

EGL Java ランタイム・エラー・コード VGJ1417E

VGJ1417E: デフォルトのプリンターがありません。

説明

ユーザーがデフォルトのプリンターへの印刷を試行しましたが、システムにデフォルトのプリンターが指定されていません。

ユーザー応答

構成済みの特定のプリンターを使用して印刷するか、環境にデフォルトのプリンターを定義するかしてください。

EGL Java ランタイム・エラー・コード VGJ1419E

VGJ1419E: 書式設定された '%2' の値 '%1' の長さは、許容最大長の (%3) を超えています。

説明

書式設定された値 (date, time, currency) がフィールドに適合する長さを超えています。

ユーザー応答

フィールドのサイズを増やすか、より短い長さの値に書式設定するかしてください。

EGL Java ランタイム・エラー・コード VGJ9900E

VGJ9900E: エラーが発生しました。エラーは %1 でした。エラーの説明をロードできません。

説明

プログラムが、デフォルト・メッセージ・クラス・ファイルとロケールのメッセージ・クラス・ファイルのいずれも検出またはロードできませんでした。これらのメッセージ・クラス・ファイルの一方または両方が欠落しているか破壊されています。

注: メッセージ・ファイルのロード中に問題が発生しているため、実行時にはこのメッセージは米国英語でのみ表示されます。

ユーザー応答

ファイル fda6.jar からクラス・ファイルを抽出した場合は、使用しているクラスが、最新ファイルのクラスと同じリリース・レベルと保守レベルにあることを検証してください。古いクラスを使用している場合は、正しいバージョンで置き換えてください。また、EGL から fda6.jar を再インストールすることもできます。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した箇所
- 内部エラーの種類

2. このメッセージが出力された状況を記録する。

3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

EGL Java ランタイム・エラー・コード VGJ9901E

VGJ9901E: エラーが発生しました。エラーは %1 でした。 %1 のメッセージ・テキストはメッセージ・クラス・ファイル %2 にありません。 VGJ0002E のメッセージ・テキストも見つかりませんでした。

説明

メッセージ・クラス・ファイルに、このメッセージ ID またはメッセージ ID VGJ0002E のランタイム・メッセージがありません。メッセージ・クラス・ファイルが壊れているか、前のリリースの EGL のメッセージ・クラス・ファイルである可能性があります。

注: メッセージ・ファイルのロード中に問題が発生しているため、実行時にはこのメッセージは米国英語でのみ表示されます。

ユーザー応答

ファイル fda6.jar からクラス・ファイルを抽出した場合は、所有しているクラスが、そのファイルのクラスと同じリリース・レベルまたは保守レベルであることを検証してください。古いクラスを使用している場合は、正しいバージョンで置き換えてください。また、EGL から fda6.jar を再インストールすることもできます。

問題が解決しない場合は、次のようにします。

1. メッセージ番号およびメッセージ・テキストを記録する。

注: エラー・メッセージには以下の重要な情報が含まれています。

- エラーが発生した個所
 - 内部エラーの種類
2. このメッセージが出力された状況を記録する。
 3. IBM サポートに問題を報告する方法の詳細な指示について、「*EGL Installation Guide*」を参照する。

付録. 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態を提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Lab Director
IBM Canada Ltd. Laboratory
8200 Warden Avenue
Markham, Ontario, Canada L6G 1C7

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性があります。その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者をお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があります、単に目標を示しているものです。

表示されている IBM の価格は IBM が小売り価格として提示しているもので、現行価格であり、通知なしに変更されるものです。卸価格は、異なる場合があります。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを

経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。お客様は、IBM のアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp. 2000, 2004. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

プログラミング・インターフェース情報

プログラミング・インターフェース情報は、プログラムを使用してアプリケーション・ソフトウェアを作成する際に役立ちます。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

警告: 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

商標

以下は、IBM Corporation の商標です。

- AIX
- CICS
- CICS/ESA®
- ClearCase
- DB2
- IBM
- IMS
- Informix
- iSeries
- MQSeries
- MVS™
- OS/400
- RACF®
- Rational
- VisualAge
- WebSphere

- z/OS

Intel は、Intel Corporation の米国およびその他の国における商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標または登録商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft[®]、Windows、および Windows NT は、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

値の設定ブロック 72
暗黙的な SQL 文 278, 279, 280, 281, 282, 283
印刷書式 166
インストール, Java 用 EGL ランタイム・コード 381
インポート 36
エディター
 コンテンツ・アシスト 135, 524
 設定, EGL 123
 ソース・ファイルの位置決め 300
 パーツのオープン 299
 EGL 523
演算子
 優先順位 726
 in 575
 isa 583

[カ行]

カーソル, SQL 247
開発過程 9
型定義 30
環境ファイル, J2EE
 更新 387
 説明 388
関数, Java アクセス 865
関数パーツ 149, 570
 作成 149
 パラメーター 564
 変数 562
関数呼び出し 560
関連要素 407
キーボード・ショートカット 136
キーワード
 新規 195
キーワード文
 アルファベット順リスト 96
 英字リスト 96
 add 605
 call 608
 case 611
 close 613

キーワード文 (続き)
 continue 615
 converse 616
 delete 617
 display 619
 execute 619
 exit 624
 for 626
 forEach 627
 forward 629
 freeSQL 631
 get 631
 get absolute 638
 get current 640
 get first 641
 get last 643
 get next 644
 get previous 650
 get relative 654
 goTo 656
 if, else 656
 move 657
MQSeries 関連 289
open 664
prepare 679
print 681
replace 681
return 684
set 685
show 696
transfer 697
try 698
while 699
起動構成
 暗黙 310
 明示的 310
 リスナー 311
機能, 使用可能化 129
基本アプリケーション
 開始 369
基本プログラム・パーツ 784
基本レコード・パーツ 413
結果セットの処理, SQL 247, 799
結果ビュー, 生成 575
結果ファイル 357
検索機能, SQL 247, 272
検索設定, SQL 128
検証プロパティ 72
コード生成, タイプ 10
コードの断片
 挿入 157

コードの断片 (続き)
 autoRedirect 158
 databaseUpdate 160
 getClickedRowValue 159
 setCursorFocus 158
構造体 28
構造体フィールド配列 82
構文図 812
固定レコード・パーツ
 説明 142
コマンド・ファイル 521
コメント 476
コメント, ソース 297
コンソール UI 変数
 errorLine 832
コンソール・ユーザー・インターフェース
 の概要 189
コンテンツ・アシスト
 使用 135
 説明 524

[サ行]

作業論理単位 334
索引付きレコード・パーツ 578
作成 190
サブストラング 810
参照
 定数 62
 パーツ 24
 変数 62
参照タイプ 195
参照の互換性 795
式
 数値 93, 546
 説明 536
 日時 536
 論理 93, 538
 string 93
 text 547
辞書
 関数
 containsKey() 90
 getKeys() 90
 getValues() 90
 insertAll() 90
 removeAll() 91
 removeElement() 91
 size() 91
 説明 87
 プロパティ 88

システム制限 535
システム・ライブラリー
 DateTimeLib 849
システム・ワード
 Web アプリケーション 861
実行単位 798
出力
 構築 355
 生成されるタイプ 572, 573
 「build project (プロジェクトのビルド)」メニュー・オプション 354
 Java 生成 357, 727
 Java ラッパー生成 728
 「rebuild all (すべての再ビルド)」メニュー・オプション 354
 「rebuild project (プロジェクトの再ビルド)」メニュー・オプション 354
使用宣言 1020
商標 1153
初期化、データ 511
書式パーツ
 印刷書式の作成 165
 検証プロパティ 72
 説明 164
 テキスト書式の作成 168
 テンプレート 181
 フィールド表示プロパティ 70
 フィルター 176, 187
 フォーマット設定プロパティ 71
編集 175
EGL 書式エディターを使用した書式の作成 177
EGL ソース形式 553
print 166
text 168
シリアル・レコード・パーツ 799
推奨事項、開発
 パーツ割り当て 15
 パッケージ 15
 ビルド記述子 15
数式 546
生成
 ウィザード 358
 概要 351
 結果ビュー 575
 出力の型 572, 573
 ディレクトリー・ターゲット 364
 バッチ・インターフェース 360, 361, 362
 ライブラリー・パーツ 700
 ワークベンチ 359
 EGL SDK 362
 EGL コマンド・ファイル 360, 361
 eglpath 517
 EGLSDK 362, 529

生成 (続き)
 EGL_GENERATORS_PLUGINDIR の設定 368
 EJB プロジェクト、配置コード 368
 Java オプション 326
 Java 出力 357, 727
 Java ラッパー 327
 Java ラッパー出力 728
生成のためのオプション
 Java 326
生成用バッチ・インターフェース 360, 361, 362
静的配列 78
セグメンテーション
 テキスト・アプリケーション 170
設定
 ソース・スタイル 124
 テンプレート 124
 EGL 119
 EGL エディター 123
 EGL から EGL へのマイグレーション 117
 EGL 書式エディター 186
 EGL 書式エディターのパレット・エントリー 181
 EGL デバッガー 120
 SQL 検索 128
 SQL データベース接続 126
text 119
宣言
 定数 57
 変数 57
ソース・スタイル、設定 124
ソース・ファイル
 エディター
 ソース・コードのコメント 297
 コメント 297
 コンテンツ・アシスト 135, 524
 作成 135
 説明 15
 プロジェクト・エクスプローラーでの位置決め 300
format 532
相対レコード・パーツ 796
双方向言語テキスト変換 510

[タ行]

代入 407
代入の互換性 401
多次元配列 78
断片
 挿入 157
 autoRedirect 158
 databaseUpdate 160
 getClickedRowValue 159

断片 (続き)
 setCursorFocus 158
データの初期化 511
データベース許可 506
データベース接続設定 126
データ変換 507
データ・コード、SQL 801
データ・パーツ
 基本レコード 413
 索引付きレコード 578
 シリアル・レコード 799
 相対レコード 796
 dataItem 139, 512
 dataTable 155, 513
 MQ レコード 714
 SQL レコード 804
定数、宣言 57
定数、-の参照 62
ディレクトリー、生成先 364
テキスト、設定 119
テキスト式 547
テキスト書式 168
テキスト・アプリケーション
 開始 370
 セグメンテーション 170
 変更データ・タグ 171
formGroup パーツ 163
デバッガー、EGL
 概要 303
 起動構成の作成 310
 コマンド 303
 サーバーの開始 313
 サーバーの準備 312
 システム・タイプ設定 303
 推奨 303
 生成されたコードからの呼び出し 303
 設定 120
 ビルド記述子 303
 ブレイクポイントの使用 314
 プログラムの開始 310
 プログラムのステップスルー 315
 変数の表示 316
 リスナー起動構成の作成 311
call 文 303
SQL データベース・アクセス 303
Web セッションの開始 313
デフォルト・データベース、SQL 270
展開、J2EE の外部の Java アプリケーション 381
テンプレート、設定 124
動的 SQL 文 259
動的配列 78

[ナ行]

名前

規則 725

別名 720, 721, 722

日時式 536

入出力エラー値 579

入力書式 792

入力レコード 792

[ハ行]

パーツ

オープン 299

検索 297

説明 19

プロパティ 68

-の参照 24

パーツのプロパティ

ConsoleForm 492

配置記述子

値の設定 386

更新 388

配置セットアップ、J2EE

記述子値 386, 388

ランタイム環境 384

ConnectionFactory、CICSJ2C 389

JDBC 接続 394

TCP/IP listener 383, 390

配列 78

関数 80

appendAll() 80

appendElement() 80

getMaxSize() 80

getSize() 81

insertElement() 81

removeAll() 81

removeElement() 81

reSizeAll() 82

resize() 81

setMaxSizes() 82

setMaxSize() 82

構造化フィールド 82

動的配列 78

バインディング 205

パッケージ

作成 134

説明 15

-の推奨事項 15

パラメーター、プログラム 781

パラメーター、関数 564

ハンドラー・パーツ

作成 237

引数スタック

C 関数 471, 474

日付、時刻、およびタイム・スタンプのフ

ォーマット指定子 49

標準 JDBC 接続 283

ビルド記述子パーツ

オプション、英字リスト 415

除去 331

説明 319

追加 324

デフォルトの設定 122

汎用オプションの編集 325

マスター・ビルド記述子 323

Java オプション 326

Java ランタイム・プロパティの編集

328

ビルド計画

生成後の呼び出し 363

説明 356

ビルド・サーバー

説明 373

AIX、Linux、または Windows

2000/NT/XP 上での始動 373

ビルド・スクリプト

説明 372

必須指定のオプション 442

ビルド・パーツ

ビルド記述子 122, 319

リソース関連 331

リンケージ・オプション 338

ビルド・パス、EGL

概要 517

編集 348

ビルド・ファイル

作成 319

説明 15

format 414

import 文の追加 347, 348

import 文の編集 347

ファイル

結果 357

作成 135, 319

ビルド 15, 319

プログラム・プロパティ 380

プロジェクト・エクスプローラーの削

除 300

リンケージ・プロパティ 396, 708

レコード・タイプとの関連 793

EGL コマンド 521

J2EE 環境 388

source 15, 135

Web サービス定義 15

ファイルとデータベースのシステム・ワー
ド

recordName.resourceAssociation 920

フィールド

構造体 808

プロパティ 68

フィールド (続き)

プロパティ、ページ 738

プロパティ、SQL 71

ConsoleField 478

Menu 494

MenuItem 495

PresentationAttributes 497

Prompt 499

Window 500

フィールド表示プロパティ 70

フォーマット設定プロパティ 71

フォルダー、作成 134

プリミティブ型

説明 37

ANY 41

BIN 53

BLOB 52

CHAR 41

CLOB 51

DATE 44

DBCHAR 42

DECIMAL 54

FLOAT 54

HEX 42

INTERVAL 45

MBCHAR 43

MONEY 55

NUM 55

NUMC 55

PACF 56

Page Designer 212

SMALLFLOAT 56

STRING 43

TIME 47

TIMESTAMP 47

UNICODE 44

プリミティブ・フィールド・レベル・プロ
パティ 739

action 743

align 744

byPassValidation 745

color 745

column 746

currency 747

currencySymbol 748

dateFormat 748

displayName 751

displayUse 751

fieldLen 752

fill 753

fillCharacter 753

help 753

highlight 754

inputRequired 754

inputRequiredMsgKey 754

intensity 755

プリミティブ・フィールド・レベル・プロ
パティアー (続き)
isBoolean 755
isDecimalDigit 756
isHexDigit 756
isNullable 756
isReadOnly 757
lineWrap 758
lowerCase 759
masked 759
maxLen 759
minimumInput 760
minimumInputMsgKey 760
modified 761
needsSOSI 761
newWindow 762
numElementsItem 763
numericSeparator 763
outline 763
pattern 764
persistent 764
protect 765
selectFromListItem 766
selectType 767
sign 767
sqlDataCode 768
sqlVariableLen 769
timeFormat 770
timeStampFormat 771
typeChkMsgKey 771
upperCase 772
validationOrder 772
validatorDataTable 773
validatorDataTableMsgKey 774
validatorFunction 774
validatorFunctionMsgKey 775
validValues 776
validValuesMsgKey 777
value 778
zeroFormat 778
ブレイクポイント 314
プログラム間での制御の移動 98
プログラム転送 10
プログラム呼び出し 10
プログラム・パーツ
基本 784
作成 146
使用宣言 1021
説明 147
入力書式 792
入力レコード 792
パラメーター 781
パラメーター以外のデータ 779
プロパティアー 789
EGL ソース形式 783
Java 生成 727

プログラム・パーツ (続き)
Java プログラム生成 357
Java ラッパー生成 728
textUI 786
プログラム・プロパティアー・ファイル
380
プロジェクト
作成 132
説明 15
データベース・オプションの指定 133
EJB、配置コード生成 368
EJB、JNDI 名 389
プロパティアー
可変長レコード 793
検証 72
パーツ 68
フィールド 68
フィールド表示 70
フォーマット 71
プログラム・パーツ 789
ページ・フィールド 738
ConsoleField 478
Java ランタイム 377, 584
MQ レコード 717
SQL フィールド 71
プロパティアー、プリミティブ・フィール
ド・レベル
action 743
align 744
byPassValidation 745
color 745
column 746
currency 747
currencySymbol 748
dateFormat 748
displayName 751
displayUse 751
fieldLen 752
fill 753
fillCharacter 753
help 753
highlight 754
inputRequired 754
inputRequiredMsgKey 754
intensity 755
isBoolean 755
isDecimalDigit 756
isHexDigit 756
isNullable 756
isReadOnly 757
lineWrap 758
lowerCase 759
masked 759
maxLen 759
minimumInput 760
minimumInputMsgKey 760

プロパティアー、プリミティブ・フィール
ド・レベル (続き)
modified 761
needsSOSI 761
newWindow 762
numElementsItem 763
numericSeparator 763
outline 763
pattern 764
persistent 764
protect 765
selectFromListItem 766
selectType 767
sign 767
sqlDataCode 768
sqlVariableLen 769
timeFormat 770
timeStampFormat 771
typeChkMsgKey 771
upperCase 772
validationOrder 772
validatorDataTable 773
validatorDataTableMsgKey 774
validatorFunction 774
validatorFunctionMsgKey 775
validValues 776
validValuesMsgKey 777
value 778
zeroFormat 778
文
関数呼び出し 93, 560
キーワード 93
代入 93, 407
定数宣言 93
変数宣言 93
null 93
SQL 247
ページ・ハンドラー・パーツ
コマンド・コンポーネントのバインデ
ィング 215
作成 204
出力コンポーネントのバインディング
217
使用宣言 1024
説明 207
単一選択コンポーネントのバインデ
ィング 219
チェック・ボックス・コンポーネント
のバインディング 218
入力コンポーネントのバインディング
217
複数選択コンポーネントのバインデ
ィング 221
EGL ソース形式 732
ページ・フィールドのプロパティアー 738

別名

概要 720

Java 721

Java ラッパー 722

別名、transfer 関連要素プロパティ
1019

変換

双方向言語テキスト 510

data 507

変更データ・タグ 171

変数、宣言 57

変数、-の参照 62

ホスト変数、SQL 801

[マ行]

マスター・ビルド記述子

概要 323

eglmaster.properties 531

plugin.xml 548

明示的な SQL 文 280, 281, 282

メッセージ・キュー

リモート 291

MQ オプション・レコード 718

MQ レコードのプロパティ 717

MQSeries 関連の EGL キーワード
289

MQSeries 直接呼び出し 292

MQSeries のサポート 285

[ヤ行]

ユーザー・インターフェース (UI) パーツ

書式 164, 553

ページ・フィールドのプロパティ
738

編集 175

formGroup 163, 550

呼び出し

C 関数 466

予約語

EGL 527

[ラ行]

ライブラリー・パーツ

作成 150

使用宣言 1021

生成される出力 700

EGL ソース形式 700

ライブラリー・パーツ、basicLibrary タイ
プ

説明 151

ライブラリー・パーツ、nativeLibrary タイ
プ

説明 152

ランタイム環境、J2EE セットアップ
384

リソース関連パーツ

関連要素 407

除去 338

説明 331

追加 336

編集 336

リンクージ・オプション・パーツ

除去 346

説明 338

追加 340

asynchLink 要素の編集 343

callLink 要素の編集 341

transfer 関連要素の編集 344

リンクージ・プロパティ・ファイル

詳細 708

説明 396

配置 395

例外

処理 100

入出力エラー値 579

EGL システム 100, 533

try ブロック 100

レコードの内部、SQL 803

レコード・タイプ

説明 143

ファイル・タイプとの関連 793

レコード・パーツ

基本 413

索引 578

作成 140

シリアル 799

説明 141

相対 796

プロパティ、可変長 793

MQ 714

Page Designer 214

SQL 247, 273, 804

レポート

エクスポート 245

エクスポートされたファイルの形式
245

概要 225

概要、作成と生成の 226

作成 226

作成後の生成 244

設計文書の追加 236

データ・ソース 228

データ・ソース・サンプル・コード
234

テンプレート 237

ドライバ関数用のコード例 234

レポート (続き)

ライブラリー 922

レポートを呼び出すためのコード 242

レポート・ドライバ・コードの作成
242

レポート・ハンドラー 230

レポート・ハンドラーのコード例 238

XML 設計文書 225

XML 設計文書内のデータ・タイプ
233

レポートの設計文書

概要 225

データ・タイプ 233

パッケージへの追加 236

レポートのデータ・ソース 228

レポート・ハンドラー

概要 230

関数 231

コード例 238

作成 237, 238

呼び出し可能な関数 232

レポート・ライブラリー

概要 922

Report レコード 229

ReportData レコード 229

論理式 538

論理パーツ

関数 149, 570

基本プログラム 784

ページ・ハンドラー 207, 732

ライブラリー 700

ライブラリー、basicLibrary タイ
プ 151

ライブラリー、nativeLibrary タイ
プ 152

textUI プログラム 786

[ワ行]

ワークベンチ、生成 358, 359

[数字]

1 次元配列 78

A

abs() 900

acos() 901

action

プリミティブ・フィールド・レベル・
プロパティ 743

activateWindowByName() 820

activateWindow() 819

activeForm 819

activeWindow 820
add ステートメント 605
addReportData() 923
addReportParameter() 923
alias callLink エLEMENTのプロパティ
444
align
プリミティブ・フィールド・レベル・
プロパティ 744
ANY 41
appendAll() 80
appendElement() 80
arrayDictionary パーツ
説明 91
arrayIndex 993
asin() 901
asynchLink エLEMENT
説明 410
package 412
recordName 412
atan2() 902
atan() 902
attachBlobToFile() 892
attachBlobToTempFile() 892
attachClobToFile() 892
attachClobToTempFile() 893

B

beginDatabaseTransaction() 952
BIGINT 関数 465
BLOB 52
「build project (プロジェクトのビルド)」
メニュー・オプション 354
buildPlan ビルド記述子オプション 419
byPassValidation
プリミティブ・フィールド・レベル・
プロパティ 745
bytes() 952

C

C 関数
引数スタック 471, 474
呼び出し 461, 466, 470
DATE 467
DATETIME 467
DECIMAL 468
EGL との併用 461
INTERVAL 467
C データ型 466
calculateChkDigitMod10() 953
calculateChkDigitMod11() 954
call ステートメント 608
callCmd() 955
callConversionTable 995
callLink エLEMENT
説明 442
ライブラリー 448
alias 444
conversionTable 445
ctgKeyStore 446
ctgKeyStorePassword 446
ctgLocation 446
ctgPort 447
JavaWrapper 447
linkType 448
location 449
luwControl 450
package 451
parmForm 452
pgmName 453
providerURL 453
refreshScreen 454
remoteBind 455
remoteComType 456
remotePgmType 458
serverID 459
type 460
cancelArrayDelete() 820
cancelArrayInsert() 820
case ステートメント 611
ceiling() 903
CHAR 41
characterAsInt() 931
CICSJ2C 呼び出しセットアップ 389
cicsj2cTimeout ビルド記述子オプション
420
clearActiveForm() 821
clearActiveWindow() 821
clearFieldsByName() 822
clearFields() 821
clearForm() 822
clearRequestAttr() 861
clearScreen() 847
clearSessionAttr() 861
clearWindowByName() 823
clearWindow() 822
clip() 931
CLOB プリミティブ型 51
close ステートメント 613
closeActiveWindow() 823
closeWindowByName() 824
closeWindow() 823
color
プリミティブ・フィールド・レベル・
プロパティ 745
column
プリミティブ・フィールド・レベル・
プロパティ 746

commentLevel ビルド記述子オプション
420
commentLine 824
commitOnConverse 986
commit() 956
compareNum() 903
compareStr() 932
concatenateWithSeparator() 934
concatenate() 933
conditionAsInt() 957
ConnectionFactory、CICSJ2C 389
connectionService() 980
connect() 957
ConsoleField
フィールド 478
プロパティ 478
ConsoleForm
パーツのプロパティ 492
ConsoleLib
activateWindowByName() 820
activateWindow() 819
activeForm 819
activeWindow 820
cancelArrayDelete() 820
cancelArrayInsert() 820
clearActiveForm() 821
clearActiveWindow 821
clearFieldsByName() 822
clearFields() 821
clearForm() 822
clearWindowByName() 823
clearWindow() 822
closeActiveWindow() 823
closeWindowByName() 824
closeWindow() 823
commentLinet 824
currentArrayCount() 824
currentArrayDataLine() 825
currentArrayScreenLine() 825
currentDisplayAttrs 825
currentRowAttrs 825
cursorWrap 826
defaultDisplayAttributes 826
defaultInputAttributes 826
deferInterrupt 827
deferQuit 827
definedFieldOrder 827
displayAtLine() 827
displayAtPosition() 828
displayError() 828
displayFieldsByName() 829
displayFields() 828
displayFormByName() 830
displayForm() 829
displayLineMode() 830
displayMessage() 830

ConsoleLib (続き)

- drawBoxWithColor() 831
- drawBox() 830
- errorLine 832
- errorWindow 832
- errorWindowVisible 832
- formLine 832
- getKeyCode() 833
- getKeyName() 833
- getKey() 832
- gotoFieldByName() 834
- gotoField() 833
- gotoMenuItemByName() 834
- gotoMenuItem() 834
- hideAllMenuItems() 835
- hideErrorWindow() 835
- hideMenuItemByName() 835
- hideMenuItem() 835
- interruptRequested 836
- isCurrentFieldByName() 836
- isCurrentField() 836
- isFieldModifiedByName() 837
- isFieldModified() 837
- key_accept 838
- key_deleteLine 838
- key_help 838
- key_insertLine 838
- key_interrupt 839
- key_pageDown 839
- key_pageUp 839
- key_quit 839
- lastKeyTyped() 839
- menuLine 840
- messageLine 840
- messageResource 840
- nextField() 840
- openWindowByName() 841
- openWindowWithFormByName() 842
- openWindowWithForm() 841
- openWindow() 841
- previousField() 842
- promptLine 842
- promptLineMode() 842
- quitRequested 843
- screen 843
- scrollDownLines() 843
- scrollDownPage() 843
- scrollUpLines() 844
- scrollUpPage() 844
- setArrayLine() 844
- setCurrentArrayCount() 844
- showAllMenuItems() 845
- showHelp() 845
- showMenuItemByName() 846
- showMenuItem() 845
- sqlInterrupt 846

ConsoleUI 190

- 概要 192
- OpenUI ステートメント 669
- ConsoleUI 画面オプション
 - UNIX ユーザー 196
- ConsoleUI の概要 189
- containsKey() 90
- continue ステートメント 615
- converse ステートメント 616
- ConverseLib
 - clearScreen() 847
 - displayMsgNum() 847
 - fieldInputLength() 848
 - pageEject() 848
 - validationFailed() 849
- ConverseVar
 - commitOnConverse 986
 - eventKey 987
 - printerAssociation 988
 - segmentedMode 990
 - validationMsgNum 991
- conversionTable callLink エlement・プロパティー 445
- convert() 960
- copyStr() 935
- cosh() 905
- cos() 904
- csouidpwd.properties 411
- ctgKeyStore callLink エlementのプロパティー 446
- ctgKeyStorePassword callLink エlementのプロパティー 446
- ctgLocation callLink エlementのプロパティー 446
- ctgPort callLink エlementのプロパティー 447
- currency
 - プリミティブ・フィールド・レベル・プロパティー 747
- currencySymbol
 - プリミティブ・フィールド・レベル・プロパティー 748
- currencySymbol ビルド記述子オプション 420
- currentArrayCount() 824
- currentArrayDataLine() 825
- currentArrayScreenLine() 825
- currentDate() 851
- currentDisplayAttrs 825
- currentFormattedGregorianDate 1007
- currentFormattedJulianDate 1008
- currentFormattedTime 1009
- currentGregorianDate 1010
- currentJulianDate 1010
- currentRowAttrs 825
- currentShortGregorianDate 1011

- currentShortJulianDate 1011
- currentTimeStamp() 852
- currentTime() 851
- curses ライブラリー、UNIX 383
- cursorWrap 826

D

- dataItem パーツ
 - 作成 139
 - 説明 139
 - EGL ソース形式 512
- dataTable パーツ
 - 作成 154
 - 説明 155
 - EGL ソース形式 513
- DATE 44
- DATE 関数 467
- dateFormat
 - プリミティブ・フィールド・レベル・プロパティー 748
- dateOf() 853
- DATETIME 関数 467
- DateTimeLib 849
 - currentDate() 851
 - currentTimeStamp() 852
 - currentTime() 851
 - dateOf() 853
 - dateValueFromGregorian() 853
 - dateValueFromJulian() 854
 - dateValue() 853
 - dayOf() 854
 - extend() 854
 - intervalValueWithPattern() 856
 - intervalValue() 855
 - mdy() 856
 - monthOf() 857
 - timeOf() 857
 - timestampFrom() 858
 - timestampValueWithPattern() 859
 - timestampValue() 858
 - timeValue() 859
 - weekdayOf() 860
 - yearOf() 860
- dateValueFromGregorian() 853
- dateValueFromJulian() 854
- dateValue() 853
- dayOf() 854
- DBCHAR 42
- dbms ビルド記述子オプション 421
- DECIMAL 54
- DECIMAL 関数 468
- decimalSymbol ビルド記述子オプション 421
- defaultDateFormat 936
- defaultDisplayAttributes 826

defaultInputAttributes 826
defaultMoneyFormat 937
defaultNumericFormat 937
defaultTimeFormat 937
defaultTimestampFormat 938
deferInterrupt 827
deferQuit 827
defineDatabaseAlias() 962
definedFieldOrder 827
delete ステートメント 617
destDirectory ビルド記述子オプション 421
destHost ビルド記述子オプション 422
destPassword ビルド記述子オプション 423
destPort ビルド記述子オプション 423
destUserID ビルド記述子オプション 423
disconnectAll() 964
disconnect() 963
display 文 619
displayAtLine() 827
displayAtPosition() 828
displayError() 828
displayFieldsByName() 829
displayFields() 828
displayFormByName() 830
displayForm() 829
displayLineMode() 830
displayMessage() 830
displayMsgNum() 847
displayName
 プリミティブ・フィールド・レベル・
 プロパティ 751
displayUse
 プリミティブ・フィールド・レベル・
 プロパティ 751
drawBoxWithColor() 831
drawBox() 830

E

ear ファイル、重複する jar ファイルの除去 385
EGL 6.0 iFix の新機能 3
EGL 6.0 の新機能 5
EGL Java ランタイム用のメッセージのカスタマイズ 713
EGL Java ランタイム・エラー・コード 1027
EGL SDK (EGL Software Development Kit) 362
EGL エディター
 概要 523
 コンテンツ・アシスト 524
 設定 123
EGL からの値の受け取り 471

EGL コマンド・ファイル 521
EGL 書式エディター
 概要 174
 設定 186
 表示オプション 186
EGL ソース形式 532
EGL での JDBC ドライバーの要件 604
EGL デバッガー
 概要 303
 起動構成の作成 310
 コマンド 303
 サーバーの開始 313
 サーバーの準備 312
 システム・タイプ設定 303
 推奨 303
 生成されたコードからの呼び出し 303
 設定 120
 ビルド記述子 303
 ブレークポイント 314
 プログラムの開始 310
 プログラムのステップスルー 315
 変数の表示 316
 文字エンコード・オプション 121
 リスナー起動構成の作成 311
 call 文 303
 SQL データベース・アクセス 303
 Web セッションの開始 313
EGL の概要 1
EGL の新機能 2
EGL ビルド・パス
 概要 517
 編集 348
EGL ビルド・ファイル形式 414
EGL プリミティブ型 466
EGL プロパティ
 概要 68
EGL への値の戻り 474
EGL 予約語 527
EGLCMD 360, 361, 518
eglmaster.properties 531
eglpath 517
EGLSDK 529
EGL_GENERATORS_PLUGINDIR 変数 368
EJB セッション
 コンポーネント 343
 説明 343
EJB プロジェクト
 配置コード生成 368
 JNDI 名の設定 389
eliminateSystemDependentCode ビルド記述子オプション 424
enableJavaWrapperGen ビルド記述子オプション 424
errorCode 996
errorLine 832

errorLog() 964
errorWindow 832
errorWindowVisible 832
eventKey 987
execute ステートメント 619
exit ステートメント 624
exportReport() 924
exp() 905
extend() 854
externallyDefined transferToTransaction 素子のプロパティ 1020

F

fieldInputLength() 848
fieldLen
 プリミティブ・フィールド・レベル・
 プロパティ 752
fill
 プリミティブ・フィールド・レベル・
 プロパティ 753
fillCharacter
 プリミティブ・フィールド・レベル・
 プロパティ 753
fillReport() 925
findStr() 938
floatingAssign() 906
floatingDifference() 906
floatingMod() 907
floatingProduct() 907
floatingQuotient() 908
floatingSum() 908
floor() 909
for ステートメント 626
forEach ステートメント 627
formatDate() 939
formatNumber() 940
formatTimeStamp() 942
formatTime() 941
formConversionTable 997
formGroup パーツ
 作成 163
 使用宣言 1024
 説明 163
 編集 175
 EGL ソース形式 550
pfKeyEquate プロパティ 739
formLine 832
forward ステートメント 629
freeBlob() 893
freeClob() 893
freeSQL ステートメント 631
frexp() 909

G

genDataTables ビルド記述子オプション 425
genDirectory ビルド記述子オプション 425
genFormGroup ビルド記述子オプション 426
genHelpFormGroup ビルド記述子オプション 426
genProject ビルド記述子オプション 427
genProperties ビルド記述子オプション 428
get absolute ステートメント 638
get current ステートメント 640
get first ステートメント 641
get last ステートメント 643
get previous ステートメント 650
get relative ステートメント 654
get ステートメント 631, 644
getBlobLen() 894
getClobLen() 894
getCmdLineArgCount() 965
getCmdLineArg() 964
getFieldValue() 926
getField() 872
getKeyCode() 833
getKeyName() 833
getKeys() 90
getKey() 832
getMaxSize() 80
getMessage() 966
getNextToken() 943
getProperty() 967
getReportData() 926
getReportParameter() 927
getReportVariableValue() 927
getRequestAttr() 862
getSessionAttr() 862
getSize() 81
getStrFromClob() 894
getSubStrFromClob() 895
getVAGSysType() 983
getValues() 90
goTo 文 656
gotoFieldByName() 834
gotoField() 833
gotoMenuItemByName() 834
gotoMenuItem() 834

H

handleHardIOErrors 1012
handleOverflow 1012
handleSysLibraryErrors 1013

help

プリミティブ・フィールド・レベル・
プロパティ 753

HEX 42

hideAllMenuItems() 835

hideErrorWindow() 835

hideMenuItemByName() 835

hideMenuItem() 835

highlight

プリミティブ・フィールド・レベル・
プロパティ 754

I

I4GL データ型 466

if, else ステートメント 656

in 演算子 575

Informix

特殊な考慮事項 271

inputRequired

プリミティブ・フィールド・レベル・
プロパティ 754

inputRequiredMsgKey

プリミティブ・フィールド・レベル・
プロパティ 754

insertAll() 90

insertElement() 81

integerAsChar() 945

intensity

プリミティブ・フィールド・レベル・
プロパティ 755

interruptRequested 836

INTERVAL 45

INTERVAL 関数 467

intervalValueWithPattern() 856

intervalValue() 855

invoke() 874

isa 演算子 583

isBoolean

プリミティブ・フィールド・レベル・
プロパティ 755

isCurrentFieldByName() 836

isCurrentField() 836

isDecimalDigit

プリミティブ・フィールド・レベル・
プロパティ 756

isFieldModifiedByName() 837

isFieldModified() 837

isHexDigit

プリミティブ・フィールド・レベル・
プロパティ 756

isNullable

プリミティブ・フィールド・レベル・
プロパティ 756

isNull() 877

isObjId() 878

isReadOnly

プリミティブ・フィールド・レベル・
プロパティ 757

J

J2EE JDBC 接続 394

J2EE 環境ファイル

更新 387

説明 388

J2EE 配置セットアップ

記述子値 386, 388

ランタイム環境 384

ConnectionFactory、CICSJ2C 389

JDBC 接続 394

TCP/IP listener 383, 390

J2EE ビルド記述子オプション 431

J2EELib

clearRequestAttr() 861

clearSessionAttr() 861

getRequestAttr() 862

getSessionAttr() 862

setRequestAttr() 863

setSessionAttr() 864

jar ファイル、ランタイム

アクセスの提供先 397

ear ファイルからの重複の除去 385

Java アクセス関数 865

Java 別名 721

Java 用 EGL ランタイム・コード、イン
ストール 381

Java ラッパー

クラス 595

使用 10

生成 327

生成の出力 728

説明 327

別名 722

Java ランタイム・プロパティ 377,
584

JavaLib

getField() 872

invoke() 874

isNull() 877

isObjId() 878

qualifiedTypeName() 879

removeAll() 881

remove() 880

setField() 882

storeCopy() 886

storeField() 887

storeNew() 889

store() 884

JavaServer Faces 211

JavaWrapper callLink エレメントのプロパ
ティ 447

JDBC 接続
標準 283
J2EE 394
JNDI 名、EJB プロジェクトのための設定
389
JSP 205

K

key_accept 838
key_deleteLine 838
key_help 838
key_insertLine 838
key_interrupt 839
key_pageDown 839
key_pageUp 839
key_quit 839

L

lastKeyTyped() 839
Ldexp() 910
library callLink エレメントのプロパティ
448
like 707
lineWrap
プリミティブ・フィールド・レベル・
プロパティ 758
linkage ビルド記述子オプション 431
linkType callLink エレメントのプロパティ
448
loadBlobFromFile() 895
loadClobFromFile() 896
loadTable() 967
LobLib 891
attachBlobToFile() 892
attachBlobToTempFile() 892
attachClobToFile() 892
attachClobToTempFile() 893
freeBlob() 893
freeClob() 893
getBlobLen() 894
getClobLen() 894
getStrFromClob() 894
getSubStrFromClob() 895
loadBlobFromFile() 895
loadClobFromFile() 896
setClobFromStringAtPosition() 897
setClobFromString() 896
truncateBlob() 897
truncateClob() 897
updateBlobToFile() 898
updateClobToFile() 898
location callLink エレメントのプロパティ
449

log10() 911
log() 911
lowerCase
プリミティブ・フィールド・レベル・
プロパティ 759
lowerCase() 945
luwControl callLink エレメントのプロパティ
450

M

masked
プリミティブ・フィールド・レベル・
プロパティ 759
matches 712
MathLib
abs() 900
acos() 901
asin() 901
atan2() 902
atan() 902
ceiling() 903
compareNum() 903
cosh() 905
cos() 904
exp() 905
floatingAssign() 906
floatingDifference() 906
floatingMod() 907
floatingProduct() 907
floatingQuotient() 908
floatingSum() 908
floor() 909
frexp() 909
Ldexp() 910
log10() 911
log() 911
maximum() 912
minimum() 912
modf() 913
pow() 913
precision() 914
round() 914
sinh() 916
sin() 916
sqrt() 916
stringAsDecimal() 917
stringAsFloat() 918
stringAsInt() 918
tanh() 919
tan() 919
maximumSize() 968
maximum() 912
maxLen
プリミティブ・フィールド・レベル・
プロパティ 759

MBCHAR 43
mdy() 856
Menu
フィールド 494
MenuItem
フィールド 495
menuLine 840
messageLine 840
messageResource 840
minimumInput
プリミティブ・フィールド・レベル・
プロパティ 760
minimumInputMsgKey
プリミティブ・フィールド・レベル・
プロパティ 760
minimum() 912
modf() 913
modified
プリミティブ・フィールド・レベル・
プロパティ 761
monthOf() 857
move 文 657
MQ レコード・パーツ
オプション・レコード 718
プロパティ 717
EGL ソース形式 714
mqConditionCode 1014
MQSeries
関連の EGL キーワード 289
サポート 285
直接呼び出し 292
MQ オプション・レコード 718
MQ レコードのプロパティ 717

N

needsSOSI
プリミティブ・フィールド・レベル・
プロパティ 761
newWindow
プリミティブ・フィールド・レベル・
プロパティ 762
nextBuildDescriptor ビルド記述子オプション
431
nextField() 840
null 247
NUM 55
NUMC 55
numElementsItem
プリミティブ・フィールド・レベル・
プロパティ 763
numericSeparator
プリミティブ・フィールド・レベル・
プロパティ 763

O

open ステートメント 664
OpenUI ステートメント 669
openWindowByName() 841
openWindowWithFormByName() 842
openWindowWithForm() 841
openWindow() 841
outline
 ブリミティブ・フィールド・レベル・
 プロパティ 763
overflowIndicator 997

P

PACF 56
package asynchLink エLEMENTのプロパティ 412
package callLink エLEMENTのプロパティ 451
Page Designer
 クイック編集ビュー、ページ・ハンド
 ラー・コード 216
 コマンド・コンポーネント 215
 サポート 205
 出力コンポーネント 217
 単一選択コンポーネント 219
 チェック・ボックス・コンポーネント
 218
 入力コンポーネント 217
 バインディング 205
 複数選択コンポーネント 221
 ブリミティブ型 212
 レコード 214
pageEject() 848
parmForm callLink エLEMENTのプロパティ 452
pattern
 ブリミティブ・フィールド・レベル・
 プロパティ 764
persistent
 ブリミティブ・フィールド・レベル・
 プロパティ 764
pfKeyEquate プロパティ 739
pgmName callLink エLEMENTのプロパティ 453
plugin.xml 548
pow() 913
precision() 914
prep ビルド記述子オプション 432
prepare ステートメント 679
PresentationAttributes
 フィールド 497
previousField() 842
print 文 681
printerAssociation 988

Prompt
 フィールド 499
promptLine 842
promptLineMode() 842
protect
 ブリミティブ・フィールド・レベル・
 プロパティ 765
providerURL callLink エLEMENTのプロパティ 453

Q

qualifiedTypeName() 879
queryCurrentDatabase() 968
「Quick Edit (クイック編集)」ビュー
 ページ・ハンドラー・コード 216
quitRequested 843

R

「rebuild all (すべての再ビルド)」メニュー
 ・オプション 354
「rebuild project (プロジェクトの再ビル
 ド)」メニュー・オプション 354
recordName asynchLink エLEMENTのプロ
 パティ 412
refreshScreen callLink エLEMENTのプロパ
 ティ 454
remoteBind callLink エLEMENTのプロパ
 ティ 455
remoteComType callLink エLEMENTのプ
 ロパティ 456
remotePgmType callLink エLEMENTのプロ
 パティ 458
removeAll() 81, 91, 881
removeElement() 81, 91
remove() 880
replace ステートメント 681
ReportLib
 addReportData() 923
 addReportParameter() 923
 exportReport() 924
 fillReport() 925
 getFieldValue() 926
 getReportData() 926
 getReportParameter() 927
 getReportVariableValue() 927
 resetReportParameters() 927
 setReportVariableValue() 928
resetReportParameters() 927
reSizeAll() 82
resize() 81
resourceAssociations ビルド記述子オプショ
 ン 432
resultSetID 799

return 文 684
returnCode 998
rollback() 969
round() 914

S

screen 843
scrollDownLines() 843
scrollDownPage() 843
scrollUpLines() 844
scrollUpPage() 844
segmentedMode 990
selectFromListItem
 ブリミティブ・フィールド・レベル・
 プロパティ 766
selectType
 ブリミティブ・フィールド・レベル・
 プロパティ 767
serverID callLink エLEMENTのプロパティ
 459
sessionBeanID ビルド記述子オプション
 433
sessionId 999
set ステートメント 685
setArrayLine() 844
setBlankTerminator() 946
setClobFromStringAtPosition() 897
setClobFromString() 896
setCurrentArrayCount() 844
setCurrentDatabase() 970
setError() 970
setField() 882
setLocale() 971
setMaxSizes() 82
setMaxSize() 82
setNullTerminator() 946
setRemoteUser() 972
setReportVariableValue() 928
setRequestAttr() 863
setSessionAttr() 864
setSubStr() 947
show 文 696
showAllMenuItems() 845
showHelp() 845
showMenuItemByName() 846
showMenuItem() 845
sign
 ブリミティブ・フィールド・レベル・
 プロパティ 767
sinh() 916
sin() 916
size() 91, 973
Software Development Kit, EGL (EGL
 SDK) 362
spaces() 948

SQL

暗黙的な文 247, 278, 279, 280, 281, 282, 283
カーソル 247
結果セットの処理 247, 799
検索機能 247, 271, 272
検索設定 128
サポート 247, 271
データ項目パーツを作成 273
データベース許可 506
データベース接続設定 126
データ・コード 801
デフォルト・データベース 270
動的文 259
ホスト変数 801
明示的な文 247, 280, 281, 282
例 260
レコードの内部 803
レコード・パーツ 247
EGL 文 247
null 247
PREPARE 文の構成 279
SQL フィールドのプロパティ 71
SQL レコード・パーツ 804
sqlca 1000
sqlcode 1001
sqlDataCode
 プリミティブ・フィールド・レベル・
 プロパティ 768
sqlDB ビルド記述子オプション 435
sqlerrd 1015
sqlerrmc 1016
sqlIID ビルド記述子オプション 436
sqlInterrupt 846
sqlIsolationLevel 1016
sqlJDBCClass ビルド記述子オプション 436
sqlJNDIName ビルド記述子オプション 437
sqlPassword ビルド記述子オプション 438
sqlState 1002
sqlValidationConnectionURL ビルド記述子オプション 438
sqlVariableLen
 プリミティブ・フィールド・レベル・
 プロパティ 769
sqlWarn 1017
sqrt() 916
startCmd() 974
startLog() 975
startTransaction() 975
storeCopy() 886
storeField() 887
storeNew() 889
store() 884
STRING 43

stringAsDecimal() 917
stringAsFloat() 918
stringAsInt() 918
strlen() 948
StrLib
 characterAsInt() 931
 clip() 931
 compareStr() 932
 concatenateWithSeparator() 934
 concatenate() 933
 copyStr() 935
 defaultDateFormat 936
 defaultMoneyFormat 937
 defaultNumericFormat 937
 defaultTimeFormat 937
 defaultTimestampFormat 938
 findStr() 938
 formatDate() 939
 formatNumber() 940
 formatTimeStamp() 942
 formatTime() 941
 getNextToken() 943
 integerAsChar() 945
 lowerCase() 945
 setBlankTerminator() 946
 setNullTerminator() 946
 setSubStr() 947
 spaces() 948
 strlen() 948
 textLen() 948
 upperCase() 949

SysLib

beginDatabaseTransaction() 952
bytes() 952
calculateChkDigitMod10() 953
calculateChkDigitMod11() 954
callCmd() 955
commit() 956
conditionAsInt() 957
connect() 957
convert() 960
defineDatabaseAlias() 962
disconnectAll() 964
disconnect() 963
errorLog() 964
getCmdLineArgCount() 965
getCmdLineArg() 964
getMessage() 966
getProperty() 967
loadTable() 967
maximumSize() 968
queryCurrentDatabase() 968
rollback() 969
setCurrentDatabase() 970
setError() 970
setLocale() 971

SysLib (続き)

setRemoteUser() 972
size() 973
startCmd() 974
startLog() 975
startTransaction() 975
unloadTable() 976
verifyChkDigitMod10() 977
verifyChkDigitMod11() 978
wait() 979
system ビルド記述子オプション 439
systemType 1002

SysVar

arrayIndex 993
callConversionTable 995
errorCode 996
formConversionTable 997
overflowIndicator 997
returnCode 998
sessionID 999
sqlca 1000
sqlcode 1001
sqlState 1002
systemType 1002
terminalID 1004
transactionID 1004
transferName 1005
userID 1005

T

tanh() 919
tan() 919
targetNLS ビルド記述子オプション 440
TCP/IP listener 383, 390
terminalID 1004
textLen() 948
textUI プログラム・パーツ 786
TIME 47
timeFormat
 プリミティブ・フィールド・レベル・
 プロパティ 770
timeOf() 857
TIMESTAMP 47
timestampFormat
 プリミティブ・フィールド・レベル・
 プロパティ 771
timestampFrom() 858
timestampValueWithPattern() 859
timestampValue() 858
timeValue() 859
transactionID 1004
transfer 文 697
transferName 1005
transferToProgram 要素
 別名 1019

transferToTransaction 要素

説明 1019

別名 1019

externallyDefined 1020

truncateBlob() 897

truncateClob() 897

try 文 698

type callLink エレメントのプロパティ
460

typeChkMsgKey

プリミティブ・フィールド・レベル・
プロパティ 771

typedef 30

U

UNICODE 44

UNIX curses ライブラリー 383

UNIX ユーザー

ConsoleUI 画面オプション 196

unloadTable() 976

updateBlobToFile() 898

updateClobToFile() 898

upperCase

プリミティブ・フィールド・レベル・
プロパティ 772

upperCase() 949

userID 1005

V

VAGCompatibility ビルド記述子オプション 441

validateSQLStatements ビルド記述子オプション 441

validationFailed() 849

validationMsgNum 991

validationOrder

プリミティブ・フィールド・レベル・
プロパティ 772

validatorDataTable

プリミティブ・フィールド・レベル・
プロパティ 773

validatorDataTableMsgKey

プリミティブ・フィールド・レベル・
プロパティ 774

validatorFunction

プリミティブ・フィールド・レベル・
プロパティ 774

validatorFunctionMsgKey

プリミティブ・フィールド・レベル・
プロパティ 775

validValues

プリミティブ・フィールド・レベル・
プロパティ 776

validValuesMsgKey

プリミティブ・フィールド・レベル・
プロパティ 777

value

プリミティブ・フィールド・レベル・
プロパティ 778

verifyChkDigitMod10() 977

verifyChkDigitMod11() 978

VGLib

connectionService() 980

getVAGSysType() 983

VGVar

currentFormattedGregorianCalendar 1007

currentFormattedJulianDate 1008

currentFormattedTime 1009

currentGregorianCalendar 1010

currentJulianDate 1010

currentShortGregorianCalendar 1011

currentShortJulianDate 1011

handleHardIOErrors 1012

handleOverflow 1012

handleSysLibraryErrors 1013

mqConditionCode 1014

sqlerrd 1015

sqlerrmc 1016

sqlIsolationLevel 1016

sqlWarn 1017

VisualAge Generator

移行 13

EGL の互換性 477

VSAM

アクセスの前提条件 285

サポート 285

システム名 285

W

wait() 979

Web アプリケーション

サポート 199

Page Designer 205

Web アプリケーションのシステム・ワー
ド 861

Web サービス定義ファイル 15

weekdayOf() 860

while ステートメント 699

Window

フィールド 500

X

XML レポート設計文書

概要 225

データ・タイプ 233

パッケージへの追加 236

Y

yearOf() 860

Z

zeroFormat

プリミティブ・フィールド・レベル・
プロパティ 778



プログラム番号: 5724-J19

Printed in Japan

SD88-7544-01



日本アイ・ビー・エム株式会社
〒106-8711 東京都港区六本木3-2-12