CICS COBOL CopyBook 用の J2C アプリケーションのビルド: 複数の可能性のある出力

目次

複数の可能性のある出力を含む CICS トランザクション用 J2C アプリケーションの作成

このチュートリアルでは、「J2C Java™ Bean」ウィザードを使用して、複数の種類の出力に対応した、トランザクション処理用のシンプルな Web アプリケーションを作成する方法を説明します。

学習目標

このチュートリアルでは以下のことを説明します。

- 「J2C Java Bean」ウィザードを使用して、外部呼び出しインターフェース (ECI) によって CICS トランザクションとのインターフェースを行う J2C アプリケーションを作成する。このプログラムは、カスタマーの分類 (優良カスタマー、通常のカスタマー、または不良カスタマー) に応じて、カスタマーについて異なる出力情報を戻します。 JSP を作成して、WebSphere Application Server 上にアプリケーションをデプロイする。
- カスタマー番号を受けとる Java メソッドを作成する。
- JSP を作成して、WebSphere Application Server 上にアプリケーションをデプロイする。

所要時間

30 分

関連情報

PDF バージョンを表示する

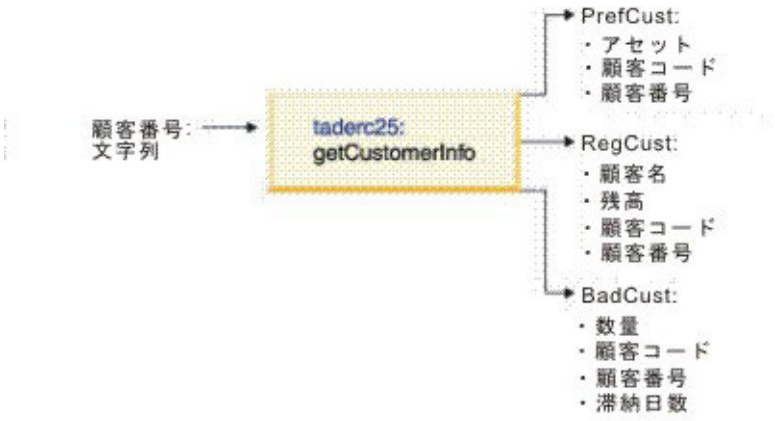
複数の可能性のある出力例

複数の可能性のある出力を含む CICS トランザクション用 J2C アプリケーション作成: はじめに

このチュートリアルでは、「J2C Java Bean」ウィザードを使用して、複数の種類の出力に対応した、CICS® トランザクション処理用のシンプルな Web アプリケーションを作成する方法を説明します。

このチュートリアルには、オプションでインストール可能なコンポーネントが必要な場合があります。システム要件を参照して、適切なオプション・コンポーネントがインストールされていることを確認してください。

このチュートリアルでは、外部呼び出しインターフェース (ECI) を使用して CICS トランザクションとのインターフェースを行う J2C アプリケーションを生成するための、詳細なステップを説明します。このサービスは、CICS COBOL 関数 getCustomerInfo を使用して作成されます。これは、カスタマー番号を受けとる関数です。このプログラムは、カスタマーの分類 (優良カスタマー、通常のカスタマー、または不良カスタマー) に応じて、カスタマーについて異なる出力情報を戻します。



このチュートリアルは幾つかの練習に分かれています。チュートリアルを正しく機能させるためには、練習を順序どおりに実行する必要があります。このチュートリアルでは、「J2C Java Bean」ウィザードを使用して、CICS ECI サーバーに接続する方法を学習します。練習では、次の作業を行います。

- 「J2C Java Bean」ウィザードを使用して、外部呼び出しインターフェース (ECI) によって CICS トランザクションとのインターフェースを行う J2C アプリケーションを作成する。
- カスタマー番号を受けとる Java メソッド *getCustomerInfo* を作成する。このプログラムは、カスタマーの分類 (優良カスタマー、通常のカスタマー、または不良カスタマー) に応じて、カスタマーについて異なる出力情報を戻します。
- アプリケーションをテストするための Java クラス TestECIMPO.java を作成する。

所要時間

このチュートリアルを終了するには、約 30 分かかります。このチュートリアルに関連した他の概念を検討する場合、完了するのにさらに長い時間がかかります。

スキル・レベル

熟練

対象読者

このチュートリアルは、特にエンタープライズ情報システム (EIS) および CICS ECI に詳しいユーザーを対象にしています。

システム要件

このチュートリアルを完了するには、以下のツールとコンポーネントがインストール済みである必要があります。

- IBM® WebSphere® Application Server バージョン 6.1
- J2EE コネクター (J2C) ツール
- **CICS ECI サーバーへの接続**: このチュートリアルでは、サーバー上の CICS プログラムとアプリケーションとの相互作用を行います。具体的には、マシン上の CICS Transaction Gateway をセットアップして、サーバーにアクセスする必要があります。さらに、CICS を実行する CICS サーバー・マシン上でいくつかのセットアップ作業を行う必要もあります。これらのステップについてはここでは触れていません。

- **COBOL ファイル taderc25.cbl のコピー**: このファイルは、製品のインストール・ディレクトリー <installdir>\IBM\SDP70Shared\plugins\com.ibm.j2c.cheatsheet.content_7.0.0\Samples\CICS\taderc25 にあります。このファイルをローカルに保管する場合は、4 ページの『taderc25.cbl』 からコードをコピーすることができます。
- **クリーンなワークスペース**

このチュートリアルを使用するには、アプリケーション・サーバーのインストールと構成が行なわれていることが必要です。サーバー・ランタイム環境が使用可能であるかどうかを検証するには、「**ウィンドウ**」 → 「**設定**」をクリックして、「**サーバー**」を展開し、「**インストール済みランタイム**」をクリックします。このペインを使用して、インストール済みサーバー・ランタイムの定義を追加、除去、または編集することができます。また、新規サーバーのサポートをダウンロードしてインストールすることもできます。

前提条件

このチュートリアルの内容すべてを実行するためには、次の知識が必要です。

- J2EE および Java プログラミング
- COBOL プログラム言語
- CICS ECI サーバー・テクノロジー

演習 1.1: リソース・アダプターの選択

この演習では、リソース・アダプターを選択して、CICS ECI サーバーに接続するように構成するための、詳細なステップについて説明します。

CICS ECI サーバーへの接続

1. J2EE アイコン () がワークスペースの右上部タブに表示されない場合は、J2EE パースペクティブに切り替える必要があります。メニュー・バーから、「**ウィンドウ**」>「**パースペクティブを開く**」>「**その他**」と選択します。「パースペクティブの選択」ウィンドウが開きます。
2. 「**J2EE**」を選択します。
3. 「**OK**」をクリックします。J2EE パースペクティブが開きます。
4. J2EE パースペクティブで、「**ファイル**」>「**新規**」>「**その他**」と選択します。
5. 「新規」ページで、「**J2C**」>「**J2C Java Bean**」と選択します。「**次へ**」をクリックします。
6. 「リソース・アダプター」ページで、「**表示基準**」の下から、「**J2C のバージョン**」を選択します。**1.5** を展開し、「**ECIResourceAdapter (IBM:6.0.2)**」を選択します。「**次へ**」をクリックします。
7. 「接続プロパティー」ページで、「**非管理接続**」を選択します。 (このチュートリアルでは、非管理接続を使用して CICS サーバーに直接アクセスするため、JNDI 名を入力する必要はありません)。デフォルトの**接続クラス名** `com.ibm.connector2.cics.ECIManagedConnectionFactory` を受け入れます。ブランク・フィールドに、接続情報を入力してください。必須フィールドは以下のとおりです。これらはアスタリスク (*) で表されています。
   - **サーバー名**: (必須ではありません) CICS Transaction Gateway サーバーの名前。
   - **接続 URL***: (必須) CICS ECI サーバーのサーバー・アドレス。
   - **ポート番号**: (必須ではありません) CICS Transaction Gateway との通信に使用するポートの番号。デフォルトのポート番号は 2006 です。
   - **ユーザー名**: (必須) 接続用のユーザー名。
   - **パスワード**: (必須) 接続用のパスワード。

これらの接続情報は、CICS サーバーのシステム管理者から入手することができます。

8. 「**次へ**」をクリックします。

taderc25.cbl

taderc25.cbl のコードを次に示します。

taderc25.cbl

```
      identification division.
       program-id. TADERC25.
       environment division.
       data division.
       working-storage section.
       01 tmp pic a(40).
       01  ICOMMAREA.
           02  ICustNo    PIC X(5).
           02  Ifiller    PIC X(11).
       01  GENCUST.
           02  GCUSTCODE PIC X(4).
           02  GFILLER PIC X(40).
       01  PREFCUST.
           02  PCUSTCODE PIC X(4).
           02  PCUSTNO    PIC X(5).
           02  ASSETS  PIC S9(6)V99.
       01  REGCUST.
           02  RCUSTCODE PIC X(4).
           02  RCUSTNO    PIC X(5).
           02  ACCOUNTNAME PIC A(10).
           02  BALANCE PIC S9(6)V99.
       01  BADCUST.
           02  BCUSTCODE PIC X(4).
           02  BCUSTNO    PIC X(5).
           02  DAYSOVERDUE PIC X(4).
           02  AMOUNT PIC S9(6)V99.
       LINKAGE SECTION.
       01 DFHCOMMAREA.
          02 inputfield pic x(50).
       procedure division.
       start-para.
           move DFHCOMMAREA to ICOMMAREA.
           IF ICustNo EQUAL '12345'
              move 'PREC' to PCUSTCODE
              move ICustNo to PCUSTNO
              move 43456.33 to ASSETS
              move PREFCUST TO DFHCOMMAREA
           ELSE IF ICustNo EQUAL '34567'
                   move 'REGC' to RCUSTCODE
                   move ICustNo to RCUSTNO
                   move 'SAVINGS' TO ACCOUNTNAME
                   move 11456.33 to BALANCE
                   move REGCUST TO DFHCOMMAREA
                ELSE
                   move 'BADC' to BCUSTCODE
                   move ICustNo to BCUSTNO
                   move '132' to DAYSOVERDUE
                   move -8965.33 to AMOUNT
                   move BADCUST TO DFHCOMMAREA
      *         END-IF.
           END-IF.
           EXEC CICS RETURN
           END-EXEC.
```

演習 1.2: Web プロジェクトおよび Java インターフェースと実装のセットアップ

この演習では、Web プロジェクトおよび Java インターフェースと実装の作成を学習します。

開始する前に、演習 1.1 を完了しておく必要があります。この演習では、次のことを行います。

- J2C Java Bean を作成する
- 動的 Web プロジェクトを作成する

1. ワークベンチを使用して行うすべての作業は、プロジェクトに関連付ける必要があります。プロジェクトでは、プロジェクトのタイプに基づいた機能により最適化された、作業ファイルとディレクトリーの系統的なビューが提供されます。ワークベンチでは、すべてのファイルがプロジェクト内になければならないため、J2C Java Bean を作成する前に、ファイルを入れるプロジェクトを作成しておく必要があります。
2. 「新規 J2C Java Bean」ページで、「**プロジェクト名**」フィールドに、値 `Taderc25Sample` を入力します。
3. 「**プロジェクト名**」フィールドの横にある「**新規**」をクリックして、新規プロジェクトを作成します。
4. 「新規ソース・プロジェクトの作成」ページで、「**Web プロジェクト**」を選択し、「**次へ**」をクリックします。
5. 「新規動的 Web プロジェクト」ページで、「**詳細を表示**」をクリックします。
6. 次の値が選択されていることを確認します。
   a. **プロジェクト名**: Taderc25Sample
   b. **プロジェクト・コンテンツ**: デフォルトの使用
   c. **ターゲット・ランタイム**: WebSphere Application Server v6.1
   d. **構成**: デフォルトを受け入れる
   e. **EAR にプロジェクトを追加**: チェック
   f. **EAR プロジェクト名**: Taderc25SampleEAR
7. 「**終了**」をクリックします。
8. 動的 Web パースペクティブに切り替えるかどうかを尋ねるダイアログ・ボックスが表示されます。「**はい**」をクリックします。
9. 「J2C Java Bean 出力プロパティー」ページで、以下を実行します。
   a. 「**プロジェクト名**」フィールドで、「**参照**」をクリックして「`Taderc25Sample`」プロジェクトを選択します。「**OK**」をクリックします。
   b. 「**パッケージ名**」フィールドに `sample.ims` と入力します。
   c. 「**インターフェース名**」フィールドに `CustomerInfoMO` と入力します。
   d. 「**実装名**」フィールドに `CustomerInfoMOImpl` と入力します。
   e. 「**Ant スクリプトとしてセッションを保管**」を未チェックのままにします。
10. 「**終了**」をクリックします。

演習 1.3: Java メソッドの作成

演習 1.3 では、Java メソッドの作成を学習します。

この演習を始める前に、「演習 1.2: Web プロジェクトおよび Java インターフェースと Java 実装のセットアップ」を完了しておく必要があります。この演習では、次のことを行います。

- Java メソッドを作成する
- COBOL と Java の間の入出力データ・マッピングを作成する

1. **まず、Java メソッドを 1 つ作成します。**以下の手順で作成する Java メソッドでは、 COBOL インポーターを使用して COBOL ソースと Java メソッドのデータの間でデータ型をマップします。
2. 「**ウィンドウ**」>「**ビューの表示**」 >「**スニペット**」 をクリックし、「スニペット」ビューを開きます。 「スニペット」ビューで、「**J2C**」をクリックします。
3. 「**J2C Java Bean へのメソッドの追加**」を右クリックし、「**挿入**」を選択します。
4. 「新規 Java メソッド」ページで、「**追加**」をクリックします。
5. 「**Java メソッド名**」フィールドに、操作の名前として `getCustomerInfo` と入力します。「**次へ**」をクリックします。
6. **次に、入力パラメーターのデータ・マッピングを作成します。**このステップでは、アプリケーションを作成するために必要な 4 ページの『taderc25.cbl』 (COBOL) ファイルをインポートします。taderc25.cbl ファイルは <installdir>\IBM\SDP70Shared\plugins\com.ibm.j2c.cheatsheet.content_7.0.0\Samples\CICS\taderc25 にあります。ここで <installdir> は、この製品がインストールされているディレクトリーです。この COBOL ファイルには、CICS サーバーで実行されるアプリケーション・プログラムが含まれています。これには、連絡域 (COMMAREA) を介して CICS サーバーに渡される構造の定義があります。この構造は、CICS アプリケーションから戻されるカスタマー・レコードを表します。ファイルで作業を行う前に、ファイル・システムからワークベンチにインポートしておく必要があります。「Java メソッド」ページの「**入力タイプ**」フィールドで、「**新規**」をクリックします。
7. 「データ・インポート」ページで、「**マッピングの選択**」フィールドが **COBOL_TO_JAVA** となっていることを確認します。「COBOL ファイル」の横にある「**参照**」をクリックします。
8. ファイル・システム内で taderc25.cbl ファイルを探して選択し、「**開く**」をクリックします。
9. 「**次へ**」をクリックします。
10. 「COBOL インポーター」ページで、**コミュニケーション・データ構造**を以下の手順で選択します。
    a. 「**プラットフォーム名**」には **Win32** を選択する。
    b. 「**コード・ページ**」には **ISO-8859-1** を選択する。
    c. 「**照会**」をクリックする。
    d. 「**データ構造**」には「**ICOMMAREA**」を選択する。
11. 「**次へ**」をクリックします。
12. 「保管するプロパティー」ページで、次の手順を実行します。
    a. 「**スタイルの生成**」で「**デフォルト**」を選択します。
    b. 「**参照**」をクリックして、Web プロジェクト **Taderc25Sample** を選択する。
    c. 「**パッケージ名**」フィールドに、`sample.cics.data` と入力する。
    d. 「**クラス名**」フィールドで、デフォルト値 **ICOMMAREA** を `InputComm` に置き換える。
13. 「**終了**」をクリックします。
14. **次に、複数の種類の出力を出力パラメーターに作成します。** 「Java メソッド」ページの「**出力タイプの指定**」フィールドで、「出力タイプ」エリアの横にある 「**新規**」をクリックします。
15. 「データ・インポート」ページで、「マッピングの選択」フィールドが **COBOL MPO から JAVA** となっていることを確認します。
16. 複数の可能性のある出力エリアの横の「**新規**」をクリックします。

17. 「インポート・ファイル」フィールドの横にある「**参照**」をクリックし、 **taderc25.cbl** ファイルのロケーションを指定します。「**開く**」をクリックします。
18. 「**次へ**」をクリックします。
19. 「インポーター」ページで、**コミュニケーション・データ構造**を以下の手順で選択します。
    a. 「**プラットフォーム**」には **Win32** を選択する。
    b. 「**コード・ページ**」には **ISO-8859-1** を選択する。
    c. 「**照会**」をクリックする。
    d. 「**データ構造**」に、「**PREFCUST**」、「**REGCUST**」、および「**BADCUST**」を選択する。
20. 「**終了**」をクリックします。「データ・インポート構成プロパティー」を指定するページで、これらの 3 つのデータ型がリストされます。
21. 「**次へ**」をクリックします。
22. **次に保管するプロパティーを指定します。**「保管するプロパティー」ページに、それぞれのカスタマー・タイプ・レコードごとに設定されたデフォルト値が表示されます。「**プロジェクト名**」フィールドに、**Taderc25Sample** と表示されていることを確認します。「**参照**」をクリックして、Web プロジェクト **Taderc25Sample** を選択します。
    a. 「保管するプロパティー」ページで、「**COBOL MPO から Java**」を選択します。
       - 「**パッケージ名**」フィールドに `sample.cics.data` と入力します。
       - 「**クラス名**」フィールドに `OutputComm` と入力します。
       - 「**既存のクラスの上書き**」を選択します。
    b. 「**COBOL MPO から Java 保管プロパティー**」を展開します。 3 つのデータ・バインディング要素が表示されます。
    c. ファイル **taderc25.cbl 内の ″PREFCUST″ の COBOL から Java 保管プロパティー**を選択します。
       - 「**生成のスタイル**」に、「**デフォルト**」を選択します。
       - 「**パッケージ名**」フィールドに `sample.cics.data` と入力します。
       - 「**クラス名**」フィールドに `PrefCust` と入力します。
       - 「**既存のクラスの上書き**」を選択します。
    d. ファイル **taderc25.cbl 内の ″REGCUST″ の COBOL から Java 保管プロパティー**を選択します。
       - 「**パッケージ名**」フィールドに `sample.cics.data` と入力します。
       - 「**クラス名**」フィールドに `RegCust` と入力します。
       - 「**既存のクラスの上書き**」を選択します。
    e. ファイル **taderc25.cbl 内の ″BADCUST″ の COBOL から Java 保管プロパティー**を選択します。
       - 「**パッケージ名**」フィールドに `sample.cics.data` と入力します。
       - 「**クラス名**」フィールドに `BadCust` と入力します。
       - 「**既存のクラスの上書き**」を選択します。
23. 「**終了**」をクリックします。「*OutputComm*」を展開すると、「**出力タイプ**」フィールドに、PrefCust、RegCust および BadCust が確認できます。
24. 「Java メソッド」ページで、「**完了**」をクリックします。
25. 「Java メソッド」ページで、次の手順を行います。
    a. 「**関数名**」フィールドに、「`TADERC25`」(COBOL プログラムの ID) を入力します。

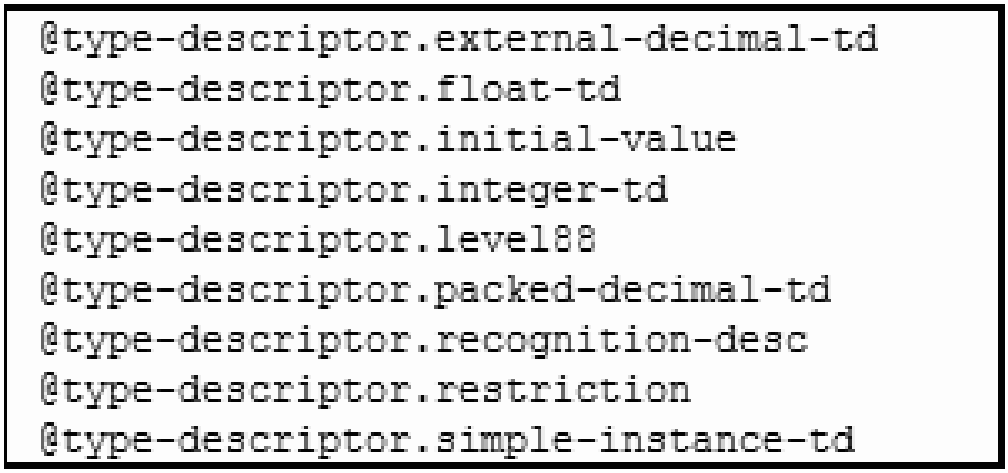
b. 「**詳細を表示**」を選択します。

c. 「**Interaction Verb**」フィールドで、 **SYNC_SEND_RECEIVE(1)** を選択します。

d. 「**応答の長さ**」フィールドに -1 と入力します。

26. 「**終了**」をクリックします。

27. **Java 出力データ・マッピング・ファイル生成に認識パターン・タグを追加します。** これは、どのデータ・タイプで出力が出てくるかが不定のため、事前にデータ・ストリームに何かのパターンを定義しておくことがタイプのマッチングに必要です。match メソッドで、認識パターンをチェックさせます。



a. PrefCust 用の認識パターンを追加するために、次の手順を実行します。

- Java エディターで PrefCust.java ファイルを開く。
- **getPcustcode()** メソッドにナビゲートする。一番簡単な方法は、「アウトライン」ビューを開いて目的のメソッドが見つかるまでスクロールダウンします。
- メソッドのコメント・エリアに、タグ `@type-descriptor.recognition-desc pattern="PREC"` を追加する。または、Ctrl+Space キーを押してコンテンツ・アシストを使用し、リストを下方にナビゲートしてこのタグを見つけ、パターンとして "PREC" を入力することもできます。
- 変更を保管する。コード PrefCust.java が再生成されます。
- match メソッドにナビゲートして、そこに変更が加えられていることを確認する。

```
/**
  * @generated

  */
 public boolean match(Object obj) {
  if (obj == null)
   return (false);
  if (obj.getClass().isArray()) {
   byte[] currBytes = buffer_;
   try {
    byte[] objByteArray = (byte[]) obj;
    buffer_ = objByteArray;
    if (!("PREC".equals(getPcustcode().toString())))
     return (false);
   } catch (ClassCastException exc) {
    return (false);
   } finally {
    buffer_ = currBytes;
   }
```

```
 } else
  return (false);
 return (true);
}
```

b. RegCust 用の認識パターンを追加するために、次の手順を実行します。

- Java エディターで RegCust.java ファイルを開く。
- **getRcustcode()** メソッドにナビゲートする。やはり一番簡単な方法は、「アウトライン」ビューを開いて目的のメソッドが見つかるまでスクロールダウンします。
- メソッドのコメント・エリアに、タグ `@type-descriptor.recognition-desc pattern="REGC"` を追加する。または、Ctrl+Space キーを押してコンテンツ・アシストを使用し、リストを下方にナビゲートしてこのタグを見つけ、パターンとして `"REGC"` を入力することもできます。
- 変更を保管する。コード RegCust.java が再生成されます。
- マッチ・メソッドにナビゲートして、そこに変更が加えられていることを確認する。

```
/**
 * @generated

 */
public boolean match(Object obj) {
 if (obj == null)
  return (false);
 if (obj.getClass().isArray()) {
  byte[] currBytes = buffer_;
  try {
   byte[] objByteArray = (byte[]) obj;
   buffer_ = objByteArray;
   if (!("REGC".equals(getRcustcode().toString())))
    return (false);
  } catch (ClassCastException exc) {
   return (false);
  } finally {
   buffer_ = currBytes;
  }
 } else
  return (false);
 return (true);
}
```

c. BadCust 用の認識パターンを追加するために、次の手順を実行します。

- Java エディターで BadCust.java ファイルを開く。
- **getBcustcode()** メソッドにナビゲートする。やはり一番簡単な方法は、「アウトライン」ビューを開いて目的のメソッドが見つかるまでスクロールダウンします。
- メソッドのコメント・エリアに、タグ `@type-descriptor.recognition-desc pattern="BADC"` を追加する。または、Ctrl+Space キーを押してコンテンツ・アシストを使用し、リストを下方にナビゲートしてこのタグを見つけ、パターンとして `"BADC"` を入力することもできます。
- 変更を保管する。コード BadCust.java が再生成されます。
- マッチ・メソッドにナビゲートして、そこに変更が加えられていることを確認する。

```
/**
 * @generated

 */
public boolean match(Object obj) {
 if (obj == null)
  return (false);
 if (obj.getClass().isArray()) {
  byte[] currBytes = buffer_;
  try {
```

```
          byte[] objByteArray = (byte[]) obj;
          buffer_ = objByteArray;
          if (!("BADC".equals(getBcustcode().toString())))
           return (false);
         } catch (ClassCastException exc) {
          return (false);
         } finally {
          buffer_ = currBytes;
         }
        } else
         return (false);
        return (true);
       }
```

演習 1.4: アプリケーションのデプロイ

演習 1.4 では、アプリケーションのテストを行うための Java クラスの作成を学習します。

開始する前に、演習 1.3 を完了しておく必要があります。この演習では、次のことを行います。

- アプリケーションをテストするための Java クラスを作成する。
- テスト・クラスを実行する。

1. **まず、TestECIMPO ファイルを作成します。** **CustomerProj** プロジェクトを展開し、「**Java リソース**」セクションを展開して、**sample.cics** パッケージを選択します。
2. 右マウス・ボタンをクリックして、「**新規**」を選択します。 クラス・オプションを選択して、新規 Java クラスを作成します。
3. 「**Java クラス名**」フィールドに、TestECIMPO と入力します。
4. Java エディターで **TestECIMPO** を開きます。
5. エディター内のコードをすべて、次のものに置き換えてください。

   注: TestECIMPO.java Java クラスは、英語圏用に作成されているため、その他の地域では、コードを変更する必要がある場合があります。

```
/*****************************************************************
 * Licensed Materials - Property of IBM
 *
 * com.ibm.j2c.cheatsheet.content
 *
 * Copyright IBM Corporation 2004. All Rights Reserved.
 *
 * Note to U.S. Government Users Restricted Rights:
   Use, duplication or disclosure restricted by GSA ADP
   Schedule Contract with IBM Corp.
 *****************************************************************/
package sample.cics;

import sample.cics.data.*;
public class TestECIMPO
{



 public static void process(InputComm input)
 {

  System.out.println("processing....");
     try {
  //CustomerInfoMOImpl proxy = new CustomerInfoMOImpl();
  CustomerInfoMOImpl proxy = new CustomerInfoMOImpl();
      OutputComm output = proxy.getCustomerInfo (input);
```

```
 BadCust badCust = output.getBadCust();
 PrefCust prefCust = output.getPrefCust();
 RegCust regCust = output.getRegCust();

 if (regCust != null)
 {
  System.out.println("Reg Customer");
  System.out.println("account name:  " + regCust.getAccountname());
  System.out.println("balance:  " + regCust.getBalance());
  System.out.println("cust code:  " + regCust.getRcustcode());
  System.out.println("cust no:  " + regCust.getRcustno());
 }
 else if (prefCust != null)
 {
  System.out.println("Pref Customer");
  System.out.println("assets:  " + prefCust.getAssets());
  System.out.println("cust code:  " + prefCust.getPcustcode());
  System.out.println("cust no:  " + prefCust.getPcustno());
 }
 else if (badCust != null)
 {
  System.out.println("Bad Customer");
  System.out.println("amount:  " + badCust.getAmount());
  System.out.println("cust code:  " + badCust.getBcustcode());
  System.out.println("cust no:  " + badCust.getBcustno());
  System.out.println("days overdue:  " + badCust.getDaysoverdue());
 }
 else
  System.out.println("No match");
}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}

}

public static void testPrefCust()
{
   System.out.println("===========testPreCust==============");
try {
 InputComm input = new InputComm();
 String prefC = "12345";
 input.setICustNo (prefC);
 process(input);
}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}


}

public static void testRegCust()
{
   System.out.println("===========testRegCust==============");
try {
 InputComm input = new InputComm();
 String regC = "34567";
 input.setICustNo (regC);
 process(input);
}
catch (Exception exc)
```

```
{
 System.out.println (exc);
 exc.printStackTrace();
}

}

public static void testBadCust()
{

 System.out.println("===========testBadCust==============");
try {

 InputComm input = new InputComm();
 String badC = "123";
 input.setICustNo (badC);
 process(input);

}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}
}


public static void main (String[] args)
{
 testPrefCust();
 testRegCust();
 testBadCust();

}
}
```

6. **次に、アプリケーションをテストします。**
7. **TestECIMPO.java** を右クリックし、「**実行**」>「**Java アプリケーション**」を選択します。
8. コンソールに以下のような出力が表示されれば、テストは成功です。

```
=========testPreCust=======
processing. . . .
Pref Customer
assets:  43456.33
cust code: PREC
cust no: 12345
=========testRegCust=======
processing. . . .
Reg Customer
account name: SAVINGS
assets:  11456.33
cust code: REGC
cust no: 34567
=========testBadCust=======
processing. . . .
Bad Customer
assets:  -8965.33
cust code: BADC
cust no: 123
days overdue: 132
```

おつかれさまでした。これで、CICS Taderc25 チュートリアルは完了です。

複数の可能性のある出力を含む CICS トランザクション用 J2C アプリケーション作成の要約

このチュートリアルでは、「J2C Java Bean」ウィザードを使用して、複数の種類の出力に対応した、CICS トランザクション処理用のシンプルな Web アプリケーションを作成する方法を説明しました。

演習での学習事項

このチュートリアルでは、以下の内容を学習しました。

- 「J2C Java Bean」ウィザードを使用して、外部呼び出しインターフェース (ECI) によって CICS トランザクションとのインターフェースを行う J2C アプリケーションを作成する。
- カスタマー番号を受けとる Java メソッド **getCustomerInfo** を作成する。このプログラムは、カスタマーの分類 (優良カスタマー、通常のカスタマー、または不良カスタマー) に応じて、カスタマーについて異なる出力情報を戻します。
- アプリケーションをテストするための、テスト用Java クラスを作成する。