E82

Building IMS Applications in Java

Kyle Charlet
charletk@us.ibm.com

IMS TECHNICAL CONFERENCE

Anaheim, California October 23 - 27, 2000

RETURN TO INDEX

© IBM CORPORATION 2000

Overview

- IMS Java Classes: What, When, and Why use them?
- Class Library Architecture?
- Java Application Basics
- Message Definition
- Database Definition
- JDBC Access to IMS DB Data

© IBM CORPORATION 2000

Why IMS Java?

- 95% of colleges teach Java, very few teach COBOL
- JDBC is simple to implement
  - ►Standard APIs make learning easier
  - ►Minimizes required knowledge of IMS and DL/I
- Customer requests for Java

© IBM CORPORATION 2000

What is IMS Java?

- A new feature in IMS V7
- A set of classes supporting IMS Applications in Java
  - Database access to DB2 and IMS DB data using JDBC
    - Industry standard API for database connectivity from Java programs
  - Message services including multi-segment and conversational, alternate I/O PCB, and MFS
  - Transaction services
- Compiled program using High Performance Java

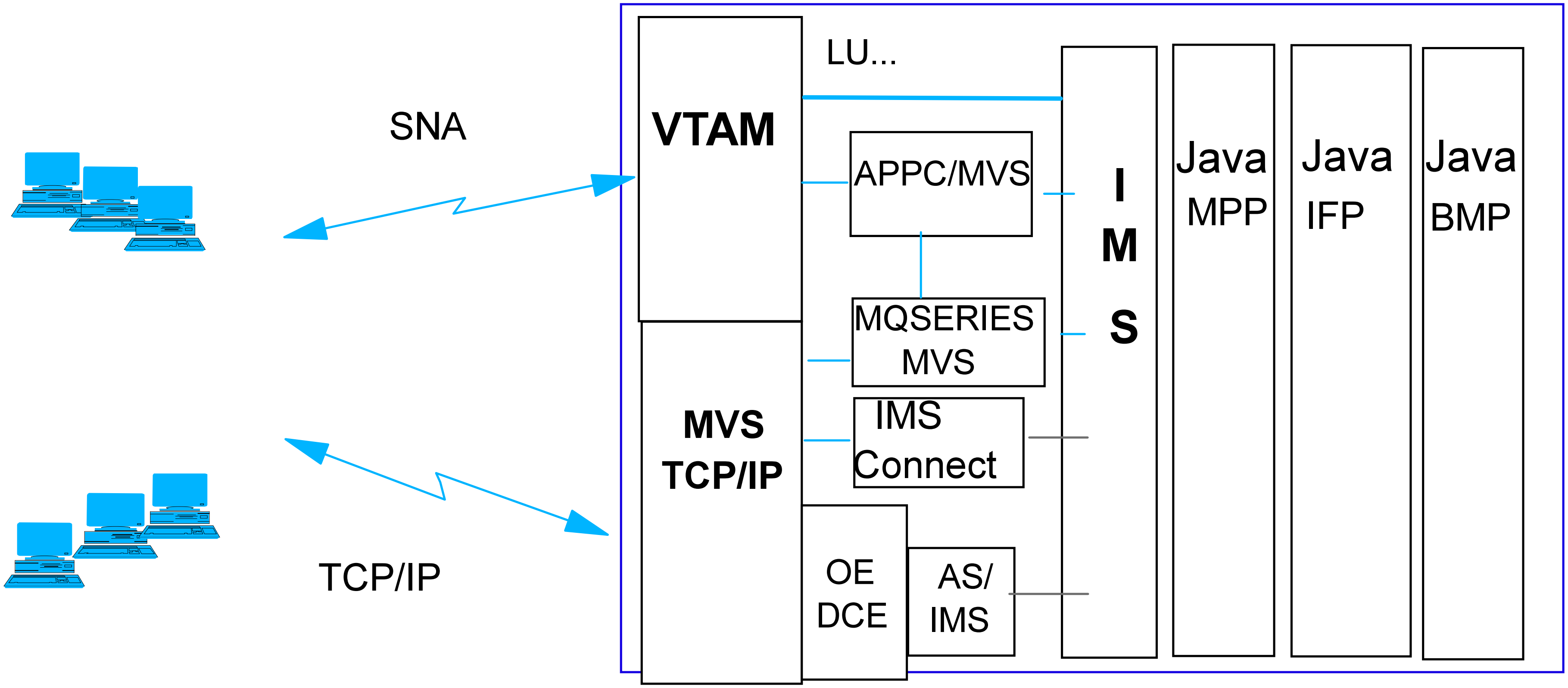
© IBM CORPORATION 2000

Sample Application

```
IMSMessageQueue msgQueue = new IMSMessageQueue();
InputMessage inputMessage = new InputMessage();
OutputMessage outputMessage = new OutputMessage();
Connection con = Connection.createConnection("jdbc:dli:DealerDatabaseView");

while (msgQueue.getUniqueMessage(inputMessage)) {

    String command = inputMessage.getString("Command");

    if (command.equals("ListDealerships")) {
        Statement stmt = con.createStatement("SELECT DealerName FROM Dealer");
        ResultSet results = stmt.executeQuery();

        int dealerIndex = 0;
        while (results.next()) {
            dealerIndex++;
            outputMessage.setString("Dealers."+dealerIndex+".DealerName",
                                                      results.getString("DealerName"));

        }
        msgQueue.insertMessage(outputMessage);
        IMSTransaction.getTransaction().commit();
    }

}
```

© IBM CORPORATION 2000

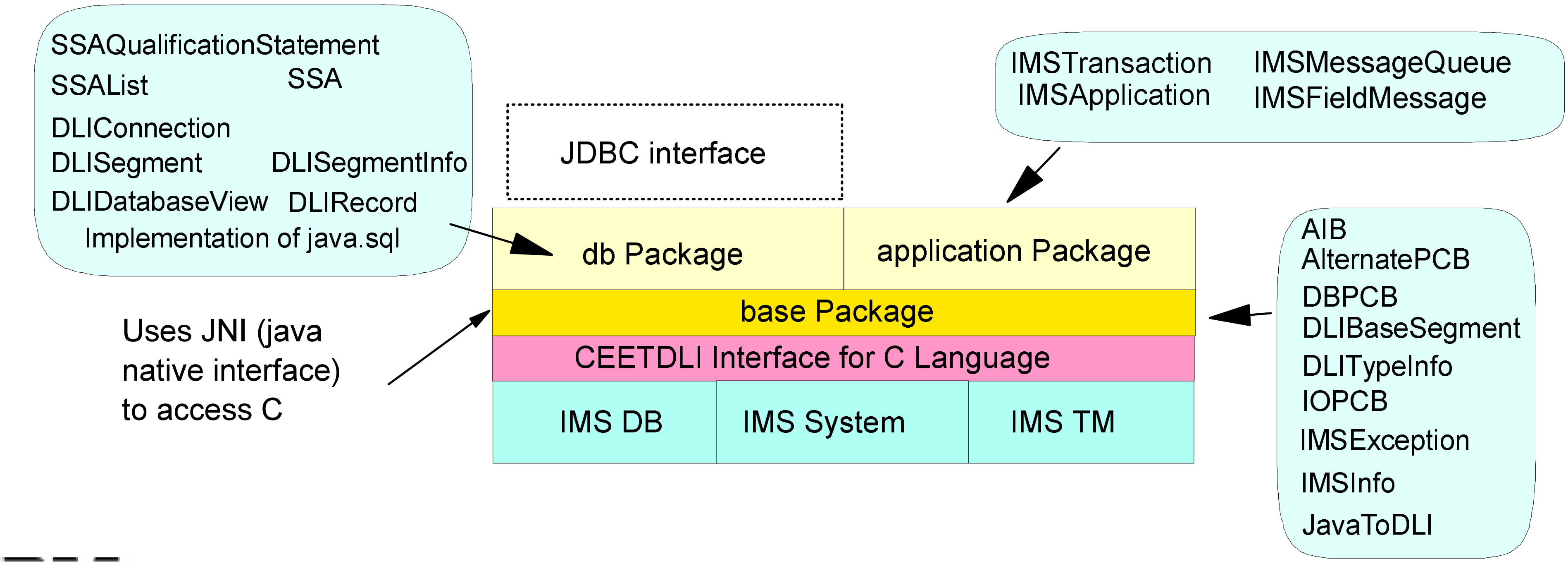
Access to IMS Java Application Programs



Java Class Library

Java program uses the APIs that are provided

- application Package classes to
  - initialize and begin the program
  - get the input message from the message queue
  - put the output message on the message queue
  - commit
- JDBC interface or db Package classes to
  - access the IMS databases



SSAQualificationStatement
SSAList SSA
DLIConnection
DLISegment DLISegmentInfo
DLIDatabaseView DLIRecord
Implementation of java.sql

JDBC interface

IMSTransaction IMSMessageQueue
IMSApplication IMSFieldMessage

db Package

application Package

base Package

CEETDLI Interface for C Language

IMS DB

IMS System

IMS TM

Uses JNI (java native interface) to access C

AIB
AlternatePCB
DBPCB
DLIBaseSegment
DLITypeInfo
IOPCB
IMSException
IMSInfo
JavaToDLI

Building an IMS Java Application

- Define input and output messages
  - ► Subclass IMSFieldMessage
  - ► Repeating fields
- Define database layout
  - ► Subclass DLIDatabaseView
- Define database segments
  - ► Subclass DLISegment
- Write application program
  - ► Subclass IMSApplication

defines metadata required for JDBC

Define Input Messages

|LL|ZZ|TRANCODE|RequestCode|DealerName|DealerID

```
public class InputMessage extends IMSFieldMessage {
  final static DLITypeInfo[] messageInfo = {
    new DLITypeInfo("RequestCode", DLITypeInfo.INT,     1,   4),
    new DLITypeInfo("DealerName",  DLITypeInfo.CHAR,    5,   20),
    new DLITypeInfo("DealerID",    DLITypeInfo.INT,    25,   4)
  };

  public InputMessage() {
    super(messageInfo, 28, false);
  }
} // end InputMessage
```

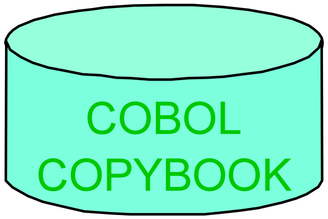
NOTE: Do not define LL, ZZ, and TRANCODE fields. Use getMessageLength and getTransactionCode methods provided by IMSFieldMessage to get length and transaction code.

Define Output Messages

```
public class CanceledOrder extends IMSFieldMessage {

  final static DLITypeInfo[] cancelInfo = {
    new DLITypeInfo("Message",   DLITypeInfo.CHAR,   1,  30),
    new DLITypeInfo("OrderDate", "MMddYYYY", DLITypeInfo.DATE,  31,  8)
  };

  public Model() {
    super(cancelInfo, 38, false);
  }
}
```

© IBM CORPORATION 2000

Repeating Fields



COBOL COPYBOOK

```
01  MODEL-OUT.
    05  MODEL-COUNT    PIC 9(6).
    05  MODEL-INFO     OCCURS 100 TIMES.
        10  MAKE          PIC X(20).
        10  MODEL         PIC X(20).
        10  COLOR         PIC X(20).
```

```
public class ModelOutput extends IMSFieldMessage {

  static DLITypeInfo[] modelTypeInfo = {
     new DLITypeInfo("Make",   DLITypeInfo.CHAR,       1,  20),
     new DLITypeInfo("Model",  DLITypeInfo.CHAR,      21,  20),
     new DLITypeInfo("Color",  DLITypeInfo.CHAR,      41,  20)
   };

  static DLITypeInfo[] modelOutputTypeInfo = {
     new DLITypeInfo("ModelCount",  DLITypeInfo.INTEGER, 1,    4),
     new DLITypeInfoList("Models",  modelTypeInfo, 5, 60, 100)
  };

  public ModelOutput() {
     super(modelOutputTypeInfo, 6004, false);
  }
}
```

© IBM CORPORATION 2000

Nested Field Access

- Support a dotted notation for specifying the fields and the index of the field within a repeating structure
  - ►Can use either field names or field indexes

Example: access the fourth "Color" in the ModelOutputMessage

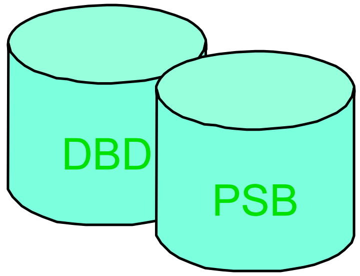
using field names: getString("Models.4.Color")
using field indexes: getString("2.4.3")

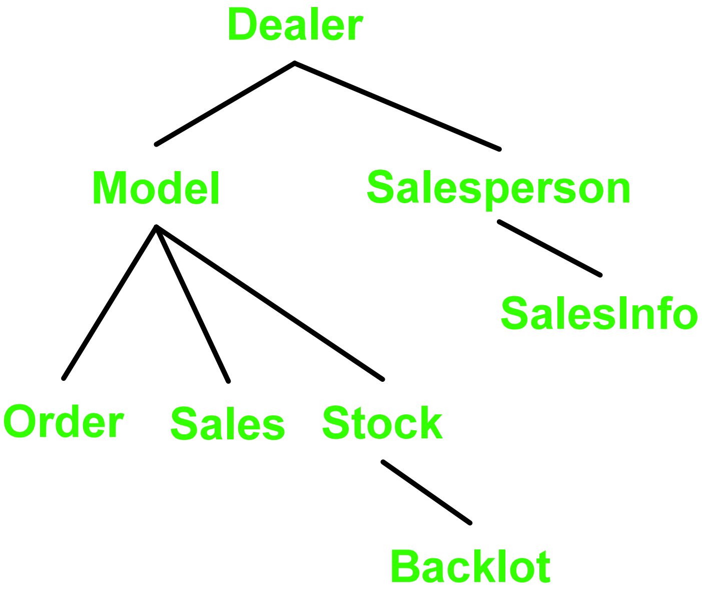
```
static DLITypeInfo[] modelTypeInfo = {
  /*1*/     new DLITypeInfo("Make",   DLITypeInfo.CHAR,      1,  20),
  /*2*/     new DLITypeInfo("Model",  DLITypeInfo.CHAR,     21,  20),
  /*3*/     new DLITypeInfo("Color",   DLITypeInfo.CHAR,     41,  20)
};

static DLITypeInfo[] modelOutputTypeInfo = {
  /*1*/     new DLITypeInfo("ModelCount",  DLITypeInfo.INTEGER, 1,    4),
  /*2*/     new DLITypeInfoList("Models",  modelTypeInfo, 5, 60, 100)
};
```

© IBM CORPORATION 2000

Define Database Layout

```
DBD NAME=AUTODBD,ACCESS=DEDB
SEGM NAME=DEALER,PARENT=0
SEGM NAME=MODEL,PARENT=DEALER
SEGM NAME=ORDER,PARENT=MODEL
SEGM NAME=SALES,PARENT=MODEL
SEGM NAME=STOCK,PARENT=MODEL
SEGM NAME=BACKLOT,PARENT=STOCK
SEGM NAME=SALESPERSON,PARENT=DEALER
SEGM NAME=SALESINFO,PARENT=SALESPERSON
```



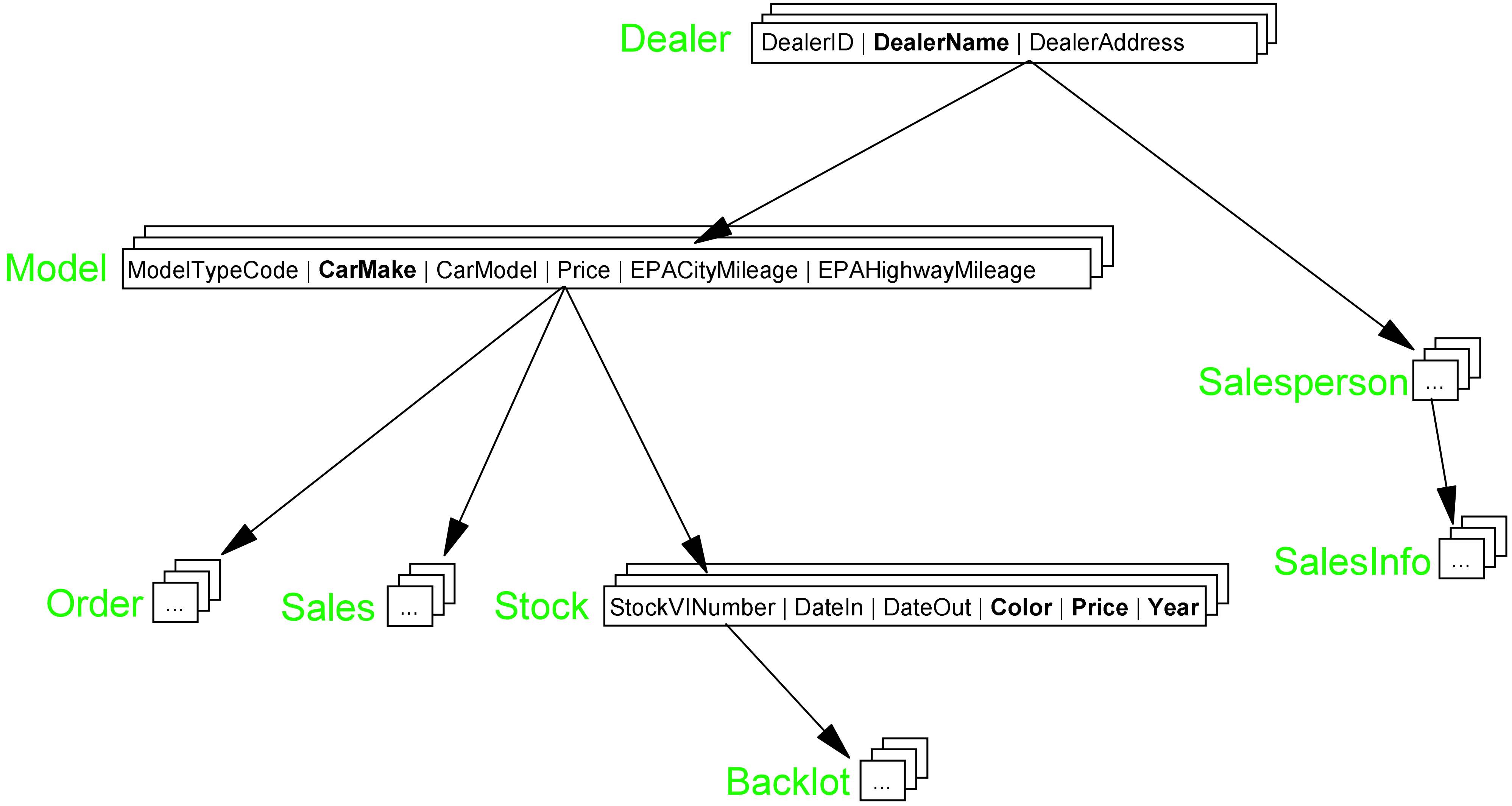


DLIDatabaseView subclass: Defines the hierarchy of DLISegments

```
public class DealerDatabaseView extends DLIDatabaseView {
static DLISegmentInfo[] segments =  {
   new DLISegmentInfo(new Dealer(),DLIDatabaseView.ROOT),
   new DLISegmentInfo(new Model(),0),
   new DLISegmentInfo(new Order(),1),
   new DLISegmentInfo(new Sales(),1),
   new DLISegmentInfo(new Stock(),1),
   new DLISegmentInfo(new BackLot(),4)
   new DLISegmentInfo(new Salesperson(),0),
   new DLISegmentInfo(new Salesinfo(),6)
  };
public DealerDatabaseView() {
     super("AUTOPCB", segments);
}
} // end DealerDatabaseView
```

PCB   TYPE=DB,DBDNAME=AUTODBD,PROCOPT=A,KEYLEN=4,PCBNAME=AUTOPCB

Database layout with properties



Dealer

DealerID | **DealerName** | DealerAddress

Model

ModelTypeCode | **CarMake** | CarModel | Price | EPACityMileage | EPAHighwayMileage

Salesperson ...

SalesInfo ...

Order ...

Sales ...

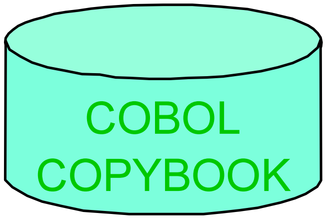
Stock

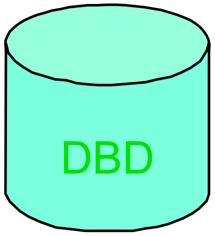
StockVINumber | DateIn | DateOut | **Color** | **Price** | **Year**

Backlot ...

© IBM CORPORATION 2000

Define Database Segments


COBOL
COPYBOOK


DBD

```
01 Dealer_Segment
    02 Dealer_ID  PIC 9(6).
    02 Dealer_Name PIC X(20).
    02 Dealer_Address PIC X(30).
```

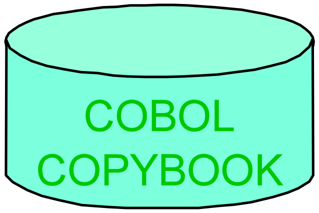
```
DBD NAME=AUTOPCB,ACCESS=DEDB
SEGM NAME=DEALER,PARENT=0,BYTES=54,
FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
FIELD NAME=DLRNAME,BYTES=20,START=5,TYPE=C
```

DLISegment subclass: Defines the type and layout of fields in a segment.

```
public class Dealer extends DLISegment {
static DLITypeInfo[] segmentInfo = {
 new DLITypeInfo("DealerID", DLITypeInfo.INT,    1, 4,   "DLRNO"),
 new DLITypeInfo("DealerName",   DLITypeInfo.CHAR,  5, 20, "DLRNAME"),
 new DLITypeInfo("DealerAddress",DLITypeInfo.CHAR, 25, 30)};

public Dealer() {
     super("DEALER", segmentInfo, 54);
}

} // end Dealer
```

© IBM CORPORATION 2000

Redefining Fields



COBOL COPYBOOK

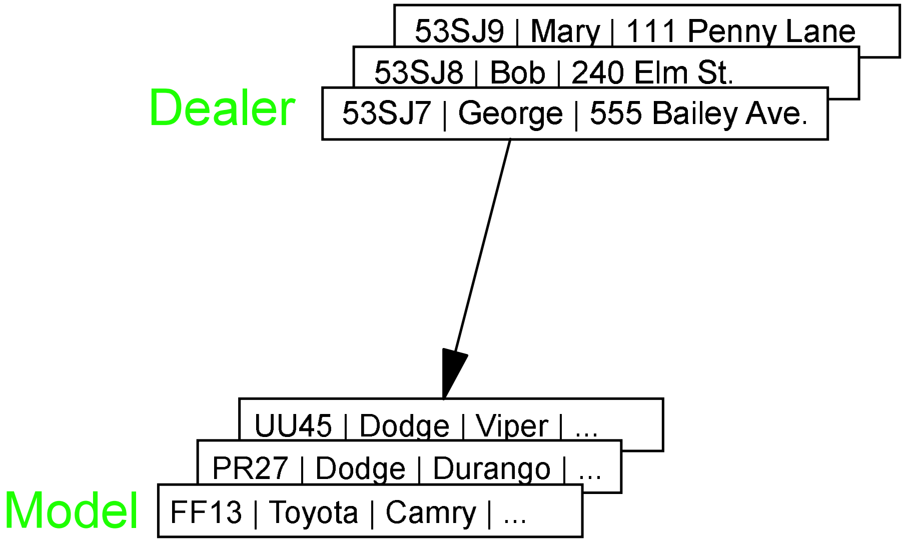
```
01 Dealer_Segment
    02 Dealer_ID  PIC X(4).
    02 Dealer_Name PIC X(20).
    02 Dealer_Address PIC X(30)
        05 Dealer_Street PIC X(14).
        05 Dealer_City PIC X(14).
        05 Dealer_State PIC X(2).
```

```
public class Dealer extends DLISegment {
 static DLITypeInfo[] segmentInfo = {
  new DLITypeInfo("DealerID",       DLITypeInfo.INT,    1,  4, "DLRNO"),
  new DLITypeInfo("DealerName",     DLITypeInfo.CHAR,   5, 20, "DLRNAME"),
  new DLITypeInfo("DealerAddress", DLITypeInfo.CHAR, 25, 30),
      new DLITypeInfo("Street",     DLITypeInfo.CHAR, 25, 14),
      new DLITypeInfo("City",       DLITypeInfo.CHAR, 39, 14),
      new DLITypeInfo("State",      DLITypeInfo.CHAR, 53,  2)
 };

public Dealer() {
     super("DEALER", segmentInfo, 54);
}
} // end Dealer
```

© IBM CORPORATION 2000

Hierarchical vs Relational

Hierarchical DB design


53SJ9 | Mary | 111 Penny Lane
53SJ8 | Bob | 240 Elm St.
Dealer
53SJ7 | George | 555 Bailey Ave.
UU45 | Dodge | Viper | ...
PR27 | Dodge | Durango | ...
Model
FF13 | Toyota | Camry | ...

Equivalent relational design

Dealer Table

DealerID	DealerName	DealerAddress
53SJ7	George	555 Bailey Ave.
53SJ8	Bob	240 Elm St.
53SJ9	Mary	111 Penny Lane

Model Table

ID	Make	Model	
FF13	Toyota	Camry	...
PR27	Dodge	Durango	...
UU45	Dodge	Viper	...

foreign key captures relationship

Note: Segment names ~ Table names
Segment instances ~ Table rows

JDBC Explained

- Set of classes and interfaces written in Java
- Java API for executing SQL statements
  - Step 1: Establish and open connection to database
  - Step 2: Execute request to obtain results
  - Step 3: Process results
- Not an acronym
  - Often thought of as standing for "Java Database Connectivity"

© IBM CORPORATION 2000

Step 1: Connection

- Load the IMS Java JDBC driver
- Create the connection
  - URL must begin with '**jdbc:dli:**' followed by fully qualified class name

```
//load driver
Class.forName(com.ibm.ims.DLIDriver);

//create connection
Connection con = Connection.createConnection("jdbc:dli:DealerDatabaseView");
```

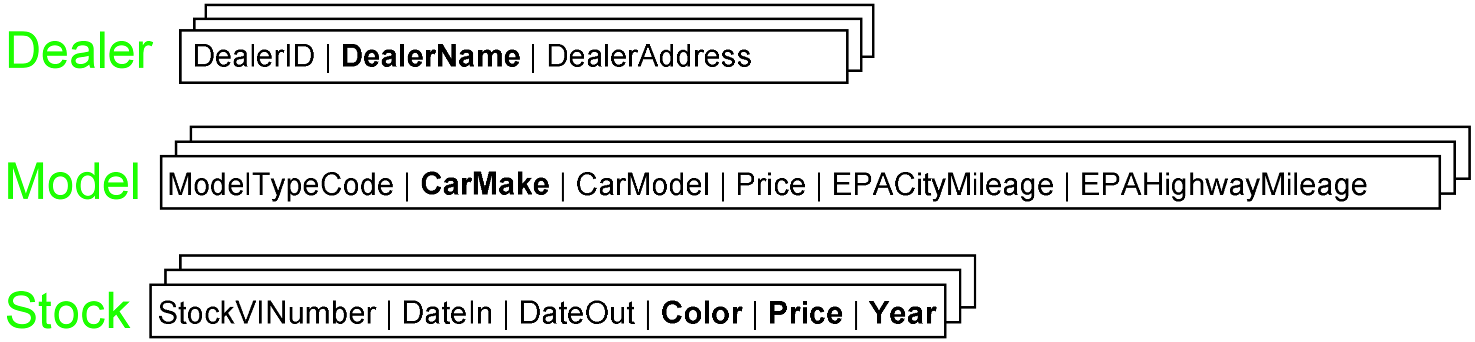
Step 2: Executing Request

```
Statement stmt = con.createStatement("SELECT Model.CarMake, Stock.Year, Stock.Price " +
                                     "FROM Stock " +
                                     "WHERE Dealer.DealerName = 'Fjord' " +
                                     "AND Stock.Price < 10000 " +
                                     "AND Stock.Color = 'Blue'");
ResultSet results = stmt.executeQuery();
```

*** make sure that the segment name in your SQL FROM clause matches the class name defined for the segment**

(i.e.; public class Stock extends DLISegment {...})

recall...



Dealer

DealerID | **DealerName** | DealerAddress

Model

ModelTypeCode | **CarMake** | CarModel | Price | EPACityMileage | EPAHighwayMileage

Stock

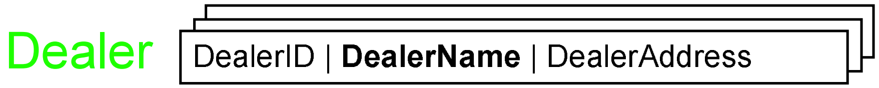
StockVINumber | DateIn | DateOut | **Color** | **Price** | **Year**

Step 2: Execute Request

- Using a PreparedStatement
  - ► Advantage: parse query once and execute multiple times
- Call PreparedStatement.setXXX methods to set the prepared values before statement is executed

```
PreparedStatement pstmt = con.prepareStatement(
        "UPDATE Dealer SET DealerName = 'Fiord' WHERE DealerName = ?");

pstmt.setString(1, "Fjord");

int updateCount = pstmt.executeUpdate();
```

recall...

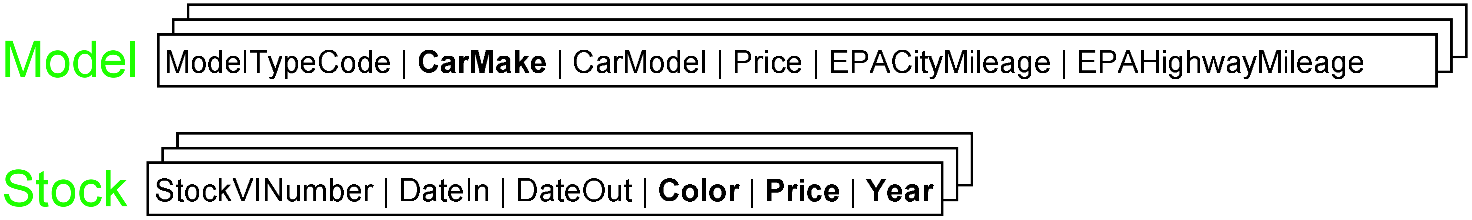


Step 3: Process Results

- Iterate through ResultSet by calling next() method
  - Returns false when no more results
- Call ResultSet.getXXX methods to access individual fields in results

```
while (results.next()) {
    String make = results.getString("CarMake");                    //or results.getString(1);
    Date year =  results.getDate("Year");                          //or results.getDate(2);
    BigDecimal price = results.getBigDecimal("Price"); //or results.getBigDecimal(3);
}
```

recall...



Model ModelTypeCode | **CarMake** | CarModel | Price | EPACityMileage | EPAHighwayMileage

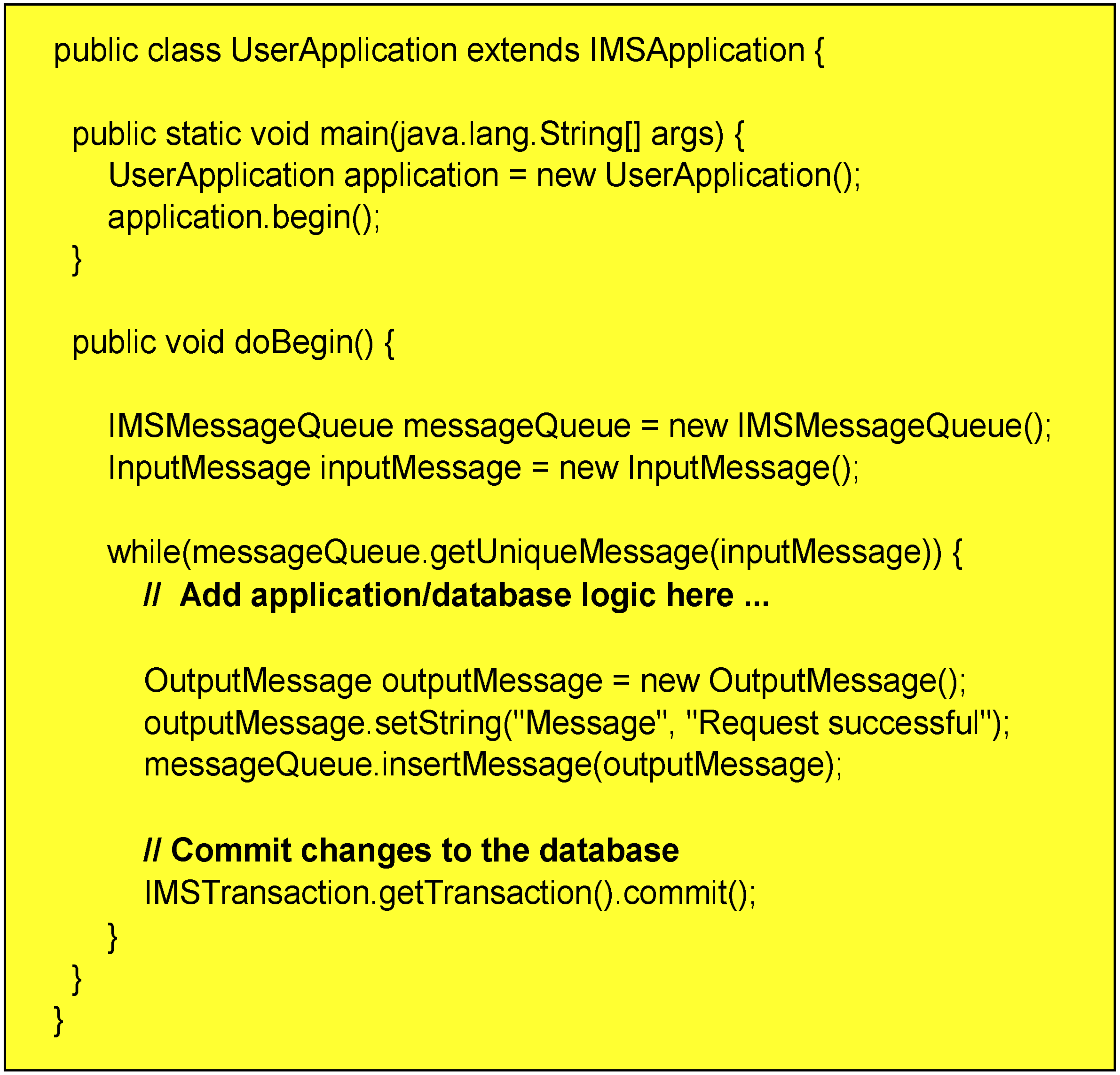
Stock StockVINumber | DateIn | DateOut | **Color** | **Price** | **Year**

Put it all together...

```
Class.forName(com.ibm.ims.DLIDriver);

Connection con = Connection.createConnection("jdbc:dli:DealerDatabaseView");

Statement stmt = con.createStatement("SELECT Model.CarMake, Stock.Year, Stock.Price " +
                                     "FROM Stock " +
                                     "WHERE Dealer.DealerName = 'Fjord' " +
                                     "AND Stock.Price < 10000 " +
                                     "AND Stock.Color = 'Blue'");
ResultSet results = stmt.executeQuery();

while (results.next()) {
  String make = results.getString("CarMake");             //or results.getString(1);
  Date year =  results.getDate("Year");                   //or results.getDate(2);
  BigDecimal price = results.getBigDecimal("Price"); //or results.getBigDecimal(3);
}

PreparedStatement pstmt = con.prepareStatement(
     "UPDATE Dealer SET DealerName = 'Fiord' WHERE DealerName = ?");

pstmt.setString(1, "Fjord");

int updateCount = pstmt.executeUpdate();

con.close();
```

Writing an Application Program

- Subclass IMSApplication and implement main() and doBegin()



```
public class UserApplication extends IMSApplication {

  public static void main(java.lang.String[] args) {
      UserApplication application = new UserApplication();
      application.begin();
  }

  public void doBegin() {

      IMSMessageQueue messageQueue = new IMSMessageQueue();
      InputMessage inputMessage = new InputMessage();

      while(messageQueue.getUniqueMessage(inputMessage)) {
         //  Add application/database logic here ...

         OutputMessage outputMessage = new OutputMessage();
         outputMessage.setString("Message", "Request successful");
         messageQueue.insertMessage(outputMessage);

         // Commit changes to the database
         IMSTransaction.getTransaction().commit();
      }
  }
}
```

© IBM CORPORATION 2000

IMSMessageQueue Class

- Option 1: Application services request -- Read from and write messages back to a single IMS message queue

To read a message from the message queue:
IMSMessageQueue.getUniqueMessage(IMSFieldMessage)

To read subsequent segments from a multi-segmented message:
IMSMessageQueue.getNextMessage(IMSFieldMessage)

To place a message on the message queue:
IMSMessageQueue.insertMessage(IMSFieldMessage)

```
IMSMessageQueue messageQueue = new IMSMessageQueue();
while (messageQueue.getUniqueMessage(inputMessage)) {
    // process message
}
```

© IBM CORPORATION 2000

IMSMessageQueue Class

- Option 2: Application is a broker -- read from one queue and write to a different queue

To read a message from the message queue (same as before):
    IMSMessageQueue.getUniqueMessage(IMSFieldMessage)

To read subsequent segments from a multi-segmented message (same as before):
    IMSMessageQueue.getNextMessage(IMSFieldMessage)

To place a message on an alternate message queue:
    // show how to construct alternatePCB object and insert to it

```
IMSMessageQueue messageQueue = new IMSMessageQueue();

//create queue with alternate PCB name to send message to another application program
IMSMessageQueue altQueue = new IMSMessageQueue("altPCB");

while (messageQueue.getUniqueMessage(inputMessage)) {
   altQueue.insertMessage(inputMessage);        //send message to another program
   messageQueue.insertMessage(new OutputMessage("Message delivered"));
   IMSTransaction.getTransaction().commit();
}
```

Database Activity With DL/I Layer

Create a connection

DLIConnection.createInstance(DLIDatabaseView);

(creates an AIB with the DBPCB name supplied in the DLIDatabaseView subclass)

```
public DealerDatabaseView() {
      super("AUTOPCB", segments);
}
```

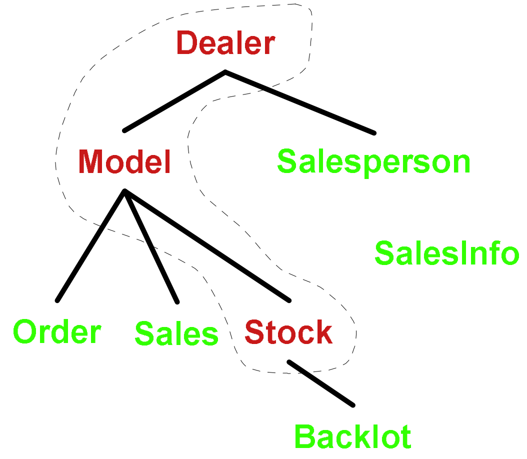
Use a connection to access segments and records in a DL/I database

To access a segment:

```
    DLIConnection.getUniqueSegment(DLISegment, SSAList);  // for first segment
    DLIConnection.getNextSegment(DLISegment, SSAList);    // for subsequent segments
```

To access a record:

```
    DLIConnection.getUniqueRecord(DLIRecord, SSAList);    // for first record
    DLIConnection.getNextRecord(DLISegment, SSAList);     // for subsequent records
```



Note: All calls accessing segments are HOLD calls.

Creating an SSAList

- An SSA defines the search criteria to be used to locate a segment
- SSALists are used to bundle together SSAs

Example: Find all blue cars sold by the 'Fjord' dealership less than $10000

```
//Create empty SSAList
SSAList ssaList = new SSAList();

//Create the individual SSAs
SSA dealerSSA = SSA.createInstance("Dealer", "DealerName", SSA.EQUALS, "Fjord");
SSA modelSSA = SSA.createInstance("Stock", "Price", SSA.LESS_THAN, "10000");
modelSSA.addQualificationStatement(SSA.AND, "Color", SSA.EQUALS, "Blue");

ssaList.addSSA(dealerSSA);
ssaList.addSSA(modelSSA);

// at this point, use the SSAList to retrieve the list of cars from the database
```

© IBM CORPORATION 2000

Retrieving Data from Segments

Use *get* methods in DLISegment to access data in individual fields

Note: The ssaList used in the call below is the list created in the previous slide

```
//Create an object to hold each of the stock segments that match our search criteria
Stock stockInfo = new Stock();

while (connection.getNextSegment(stockInfo, ssaList)) {
   System.out.println("Year: " + stockInfo.getDate("CarYear"));
   System.out.println("Price: " + stockInfo.getBigDecimal("Price"));
}
```

Supported Data Types

JDBC/DLI Type	Java Type
CHAR	String
VARCHAR	String
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
FLOAT	float
DOUBLE	double
BINARY	byte[]
PACKEDDECIMAL	java.math.BigDecimal
ZONEDDECIMAL	java.math.BigDecimal
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Defining Types - Basic Types

INTEGER
LONG
FLOAT
DOUBLE
CHAR
VARCHAR
BINARY
TINYINT
SMALLINT
BIT

```
DLITypeInfo(String fieldName,
            int type,
            int startingOffset,
            int length)
```

FIELD-A PIC X(25)	new DLITypeInfo("FieldA", DLITypeInfo.CHAR, 1, 25)
FIELD-B PIC 9(4)	new DLITypeInfo("FieldB", DLITypeInfo.SMALLINT, 26, 2)
FIELD-C PIC 9(6)	new DLITypeInfo("FieldC", DLITypeInfo.INTEGER, 28, 4)
FIELD-D PIC 9(12)	new DLITypeInfo("FieldD", DLITypeInfo.LONG, 32, 8)
FIELD-E COMP-2	new DLITypeInfo("FieldE", DLITypeInfo.DOUBLE, 40, 8)

Defining Types - Complex Types

PACKEDDECIMAL
ZONEDDECIMAL
DATE
TIME
TIMESTAMP

```
DLITypeInfo(String fieldName,
            String typeQualifier,
            int type,
            int startingOffset,
            int length)
```

```
FIELD-A  PIC S999 COMP-3              new DLITypeInfo("FieldA", "S999", DLITypeInfo.PACKEDDECIMAL, 1, 2)
FIELD-B  PIC 9(4)V99 COMP-3 DISPLAY   new DLITypeInfo("FieldB", "9(4)V99", DLITypeInfo.ZONEDDECIMAL, 3, 6)
DATE.                                 new DLITypeInfo("Date", "ddMMyyyy", DLITypeInfo.DATE, 9, 8)
    DD      PIC X(2)
    MM      PIC X(2)
    YYYY    PIC X(4)
```

More on *typeQualifier*

- Indicates layout of packed or zoned decimal fields
  - Any valid combination of the characters S, 9, V, P, and '.' is supported
- Indicates the formatting and layout of date, time and timestamp fields
  - Any valid date, time, or timestamp format is supported (see javadoc for class java.text.SimpleDateFormat)

Examples:

```
new DLITypeInfo("SalePrice", "S9(5).99",     DLITypeInfo.ZONEDDECIMAL, 1, 8)
new DLITypeInfo("SaleDate", "yyyyMMdd",      DLITypeInfo.DATE,             9, 8)
```

Length for packed fields:	ceiling[(numberDigits + 1)/2]
Length for zoned fields:	numberDigits
Length for date, time, and timestamp fields:	numberCharacters

Digits are the following characters: 9 and '.'

Data Conversions

	TINYINT	SMALLINT	INTEGER	BIGINT	FLOAT	DOUBLE	BIT	CHAR	VARCHAR	PACKEDDECIMAL	ZONEDDECIMAL	BINARY	DATE	TIME	TIMESTAMP
getByte	x	o	o	o	o	o	o	o	o	o	o				
getShort	o	x	o	o	o	o	o	o	o	o	o				
getInt	o	o	x	o	o	o	o	o	o	o	o				
getLong	o	o	o	x	o	o	o	o	o	o	o				
getFloat	o	o	o	o	x	o	o	o	o	o	o				
getDouble	o	o	o	o	o	x	o	o	o	o	o				
getBoolean	o	o	o	o	o	o	x	o	o	o	o				
getString	o	o	o	o	o	o	o	x	x	o	o	o	o	o	o
getBigDecimal	o	o	o	o	o	o	o	o	o	x	x				
getBytes												x			
getDate								o	o				x		o
getTime								o	o					x	o
getTimestamp								o	o				o	o	x

An 'X' indicates the getXXX method is recommended to access the given data type
An 'O' indicates the getXXX method may be legally used to access the given data type

© IBM CORPORATION 2000

RETURN TO INDEX

Conclusions

- IMS Java allows Java developers to create new applications quickly, easily, and without in-depth IMS knowledge
- We discussed two approaches to application programming
  - DLI
  - JDBC

© IBM CORPORATION 2000