A15

e-business Tools: A COBOL Programmer's Perspective on the Writing of Java for MPPs

Geoff Nicholls
Senior IT Specialist, Silicon Valley Laboratory
(Australian Branch)

IMS Technical Conference

Miami Beach, FL October 22-25, 2001

RETURN TO INDEX

Background

Writing an MPP or a BMP is second nature to many Cobol and PL/I Programmers. How does this knowledge help when confronted with a request to write an MPP in Java ?

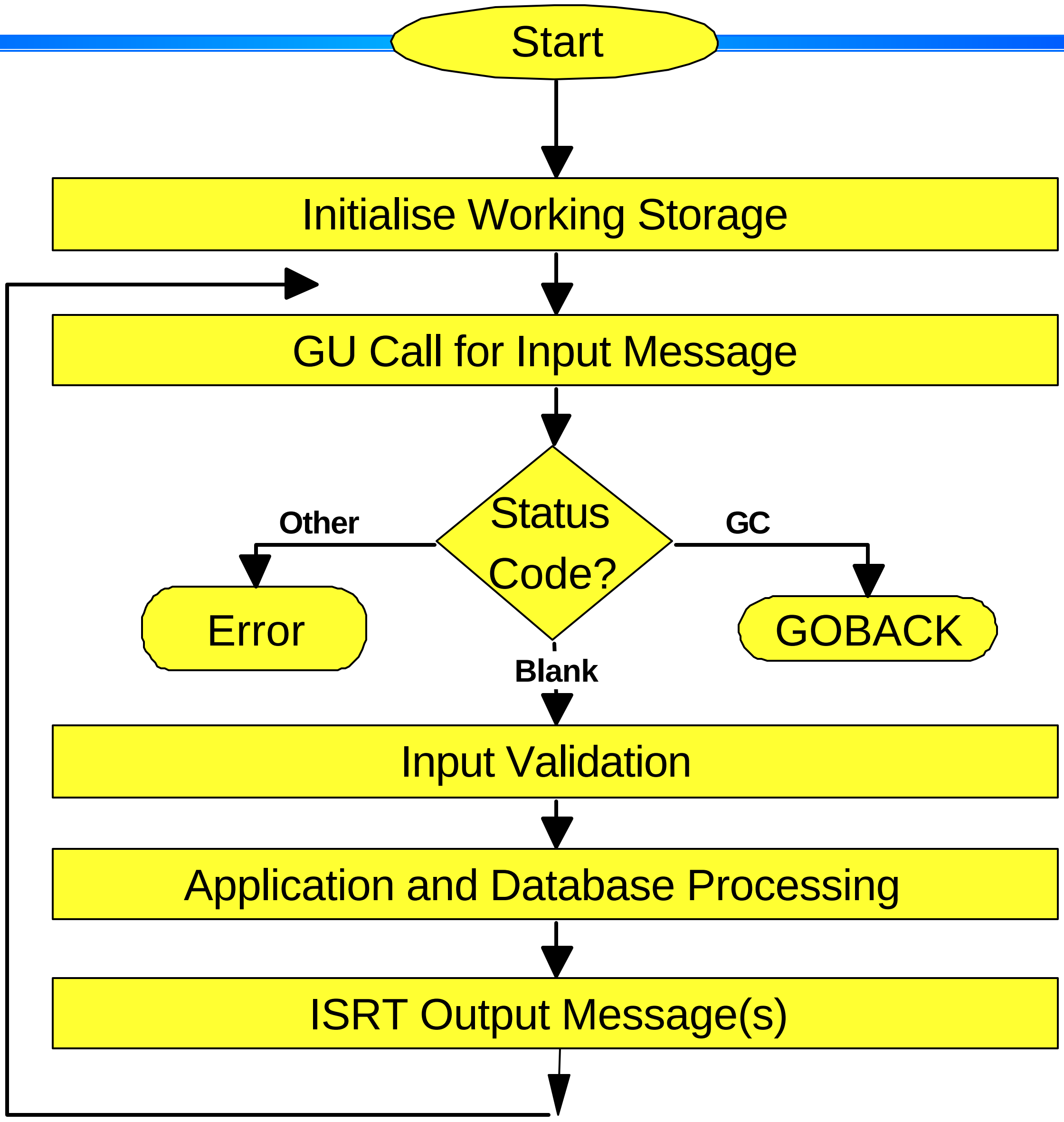
This session discusses the basic structure of an MPP which is written in Java, the differences you want to know about when writing programs in Java, taking into account the differences between the languages, and the environments in which these programs are developed and run.

Bonus Section... Using the power of IBM's VisualAge for Java for program development.

Agenda

- Cobol
  - Program Structure
  - Message Queue Access
  - Database Access
- Java
  - Database Definition
  - Database Access and Update
  - Message Queue Access
  - Commit
- Using IBM VisualAge for Java

Cobol Program Structure


Start
Initialise Working Storage
GU Call for Input Message
Status Code?
Other
Error
GC
GOBACK
Blank
Input Validation
Application and Database Processing
ISRT Output Message(s)

Cobol Access to Message Queues

-
  ```
  77 GET-UNIQUE PICTURE X(4) VALUE ‘GU ‘.
  ```

-
  ```
  01 IOPCB.
     02 LTERM-NAME PICTURE X(8).
     02 FILLER PICTURE X(2).
     02 TPSTATUS PICTURE XX.
     02 FILLER PICTURE X(20).
  ```

-
  ```
  01 INPUT-MSG.
     02 IN-LL PICTURE S9(3) COMP.
     02 IN-ZZ PICTURE S9(3) COMP.
     02 IN-FILL PICTURE X(4).
     02 IN-COMMAND PICTURE X(8).
  ```

-
  ```
  CALL 'CBLTDLI'
  USING   GET-UNIQUE,
          IOPCB,
          INPUT-MSG.
  ```

Cobol Database Access

-
  ```
  CALL 'CBLTDLI'
   USING GET-UNIQUE,
         DB-PCB-DETAIL,
         DETAIL-SEGMENT,
         SSA-FOR-DETAIL.
  ```

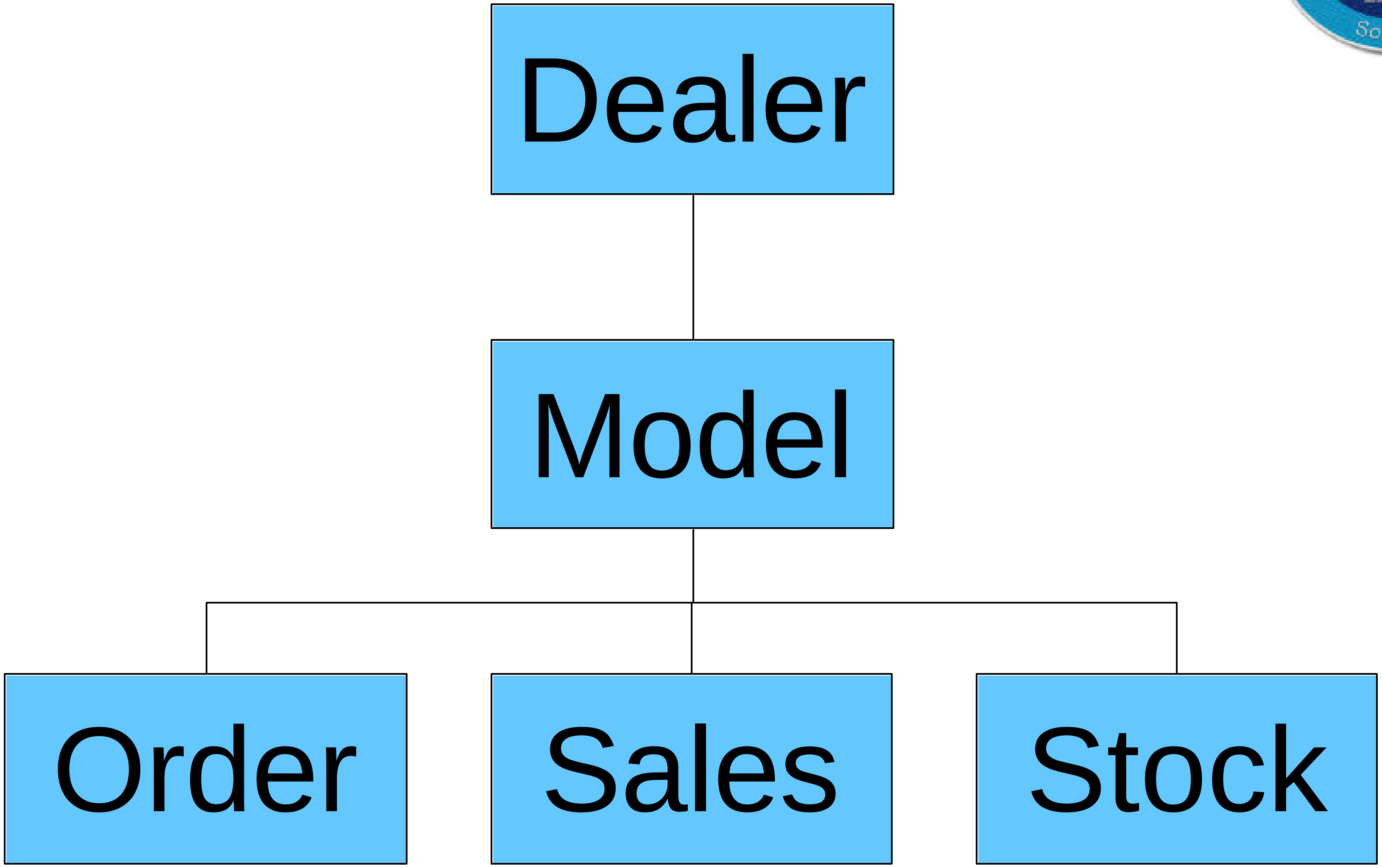
Java Programs

- Database Definition
- Database Access
  - JDBC
  - DLIConnection
- Message Queue Access
- Programming

Mapping an IMS Database in Java Classes

- IMS Database Definition (DBD)
- Mapping the DBD to DLIDatabaseView
  - defines the hierarchical structure to your program
- Mapping the DBD to DLISegment
  - defines the fields in the segment to your prorgam

Sample Database Definition



IMS DBD Definition

```
DBD NAME=DEALERDB,ACCESS=(HDAM,OSAM),RMNAME=(DFSHDC40.1.10)
*
SEGM NAME=DEALER,PARENT=0,BYTES=94,
FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
FIELD NAME=DLRNAME,BYTES=30,START=5,TYPE=C
*
SEGM NAME=MODEL,PARENT=DEALER,BYTES=43
FIELD NAME=(MODTYPE,SEQ,U),BYTES=2,START=1,TYPE=C
FIELD NAME=MAKE,BYTES=10,START=3,TYPE=C
FIELD NAME=MODEL,BYTES=10,START=13,TYPE=C
FIELD NAME=YEAR,BYTES=4,START=23,TYPE=C
FIELD NAME=MSRP,BYTES=5,START=27,TYPE=P
*
SEGM NAME=ORDER,PARENT=MODEL,BYTES=127
FIELD NAME=(ORDNBR,SEQ,U),BYTES=6,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=50,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=75,TYPE=C
*
SEGM NAME=SALES,PARENT=MODEL,BYTES=113
FIELD NAME=(SALDATE,SEQ,U),BYTES=8,START=1,TYPE=C
FIELD NAME=LASTNME,BYTES=25,START=9,TYPE=C
FIELD NAME=FIRSTNME,BYTES=25,START=34,TYPE=C
FIELD NAME=STKVIN,BYTES=20,START=94,TYPE=C
*
SEGM NAME=STOCK,PARENT=MODEL,BYTES=62
FIELD NAME=(STKVIN,SEQ,U),BYTES=20,START=1,TYPE=C
FIELD NAME=COLOR,BYTES=10,START=37,TYPE=C
FIELD NAME=PRICE,BYTES=5,START=47,TYPE=C
FIELD NAME=LOT,BYTES=10,START=53,TYPE=C
*
DBDGEN
FINISH
END
```

1. Mapping the DBD to DLIDatabaseView

```
DBD NAME=DEALERDB,ACCESS=(HDAM,OSAM),RMNAME=(DFSHDC40.1.10)
SEGM NAME=DEALER,PARENT=0,BYTES=94,
SEGM NAME=MODEL,PARENT=DEALER,BYTES=43
SEGM NAME=ORDER,PARENT=MODEL,BYTES=127
SEGM NAME=SALES,PARENT=MODEL,BYTES=113
SEGM NAME=STOCK,PARENT=MODEL,BYTES=62
```

```
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),
   };

public DealerDatabaseView(){
   super(“DEALERDB”, segments);
}
}
```

public
class
extends
static
new
super

2. Mapping the DBD to DLISegment

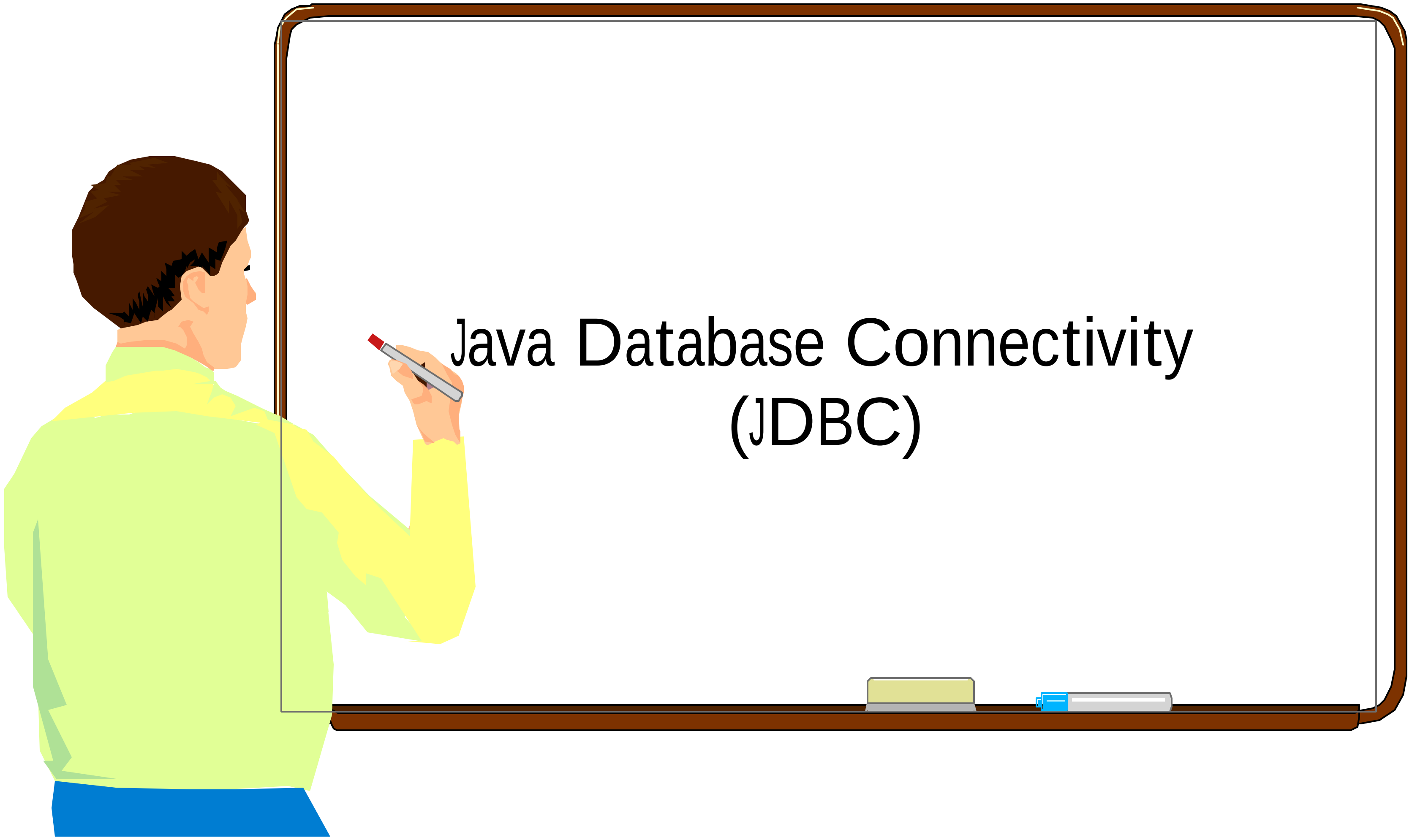
```
SEGM NAME=DEALER,PARENT=0,BYTES=94,
FIELD NAME=(DLRNO,SEQ,U),BYTES=4,START=1,TYPE=C
FIELD NAME=DLRNAME,BYTES=30,START=5,TYPE=C
```

```
public class Dealer extends DLISegment {
   static DLITypeInfo[] typeInfo = {
      new DLITypeInfo("DealerNumber", DLITypeInfo.CHAR,1, 4,"DLRNO"),
      new DLITypeInfo("DealerName", DLITypeInfo.CHAR,5, 30,"DLRNAME"),
      new DLITypeInfo("DealerAddress",DLITypeInfo.CHAR,35, 50),
      new DLITypeInfo("YTDSales", DLITypeInfo.PACKEDDECIMAL,85, 10),
      };
   public Dealer() {
      super("DEALER", typeInfo, 94);
   }
```

IMS Database Access with Java

- Java Database Connectivity (JDBC)
  - limited subset of SQL92 query language
  - JDBC 1.0
    - part of the standard Java APIs
  - familiar to Java programmers
- DLIConnection
  - Lower level interface
  - familiar to IMS applications programmers

Java Database Connectivity (JDBC)



Writing a JDBC Application

- Define the segment hierarchy
  - to identify the segments used from the database
    - Provide a subclass of DLIDatabaseView
- Define the fields and field types
  - for each segment accessed from the database
    - Provide a subclass of DLISegment
- Connect to the Database
  - Load the DLIDriver and retrieve a Connection object from the DriverManager
- Issue the SQL call
  - Retrieve a Statement or PreparedStatement from the connection
- Access the data returned
  - Iterate the ResultSet returned to retrieve specific field results

Writing a JDBC Application ...(review)

1. Provide a subclass of DLIDatabaseView
   - to identify the segments used from the database
   - defines the segment hierarchy
2. Provide a subclass of DLISegment
   - for each segment accessed from the database
   - defines the fields and field types in each segment
3. Load the DLIDriver
   - and retrieve a Connection object from the DriverManager
4. Retrieve a Statement or PreparedStatement
   - from the connection
5. Iterate the ResultSet returned
   - to retrieve specific field results

3. Load the DLIDriver

- Load the DLIDriver
  - using Class.forname
- Retrieve a connection
  - from DriverManager

Class
connection
DriverManager
catch
reply

```
try {
        Class.forName("com.ibm.ims.db.DLIDriver");
        connection = DriverManager.getConnection
                ("jdbc:dli:dealership.application.DealerDatabaseView");
    }
    catch (Exception e){
        reply("Connection not established");
    }
```

4. Retrieve a Statement

```
public boolean getModelDetails(InputMessage inputMessage,
                          ModelOutput modelOutput) throws IMSException {

   // Parse the input message for ModelTypeCode
      String queryString = "SELECT * from Model where ModelTypeCode = "
         + "'" + inputMessage.getString("ModelTypeCode").trim() + "'";
```

public
boolean
throws
SELECT
String

5. Iterate the ResultSet

```
    // Create a statement and execute it to get a ResultSet
    try {
        Statement statement = connection.createStatement();
        ResultSet results = statement.executeQuery(queryString);
        // Send back the result of the query
        // Note: since "ModelTypeCode" is unique - only 1 row
        // is returned
        if (results.next()) {
           modelOutput.setString("ModelTypeCode",
                                 results.getString("Type").trim());
           modelOutput.setString("Make",
                                 results.getString("Make").trim());
           return true;
        }
        else {
           reply("Unknown Type");
           return false;
        }
    }
    catch (SQLException e) {
       reply("Query Failed:"+ e.toString());
       return false;
    }
}
```

try
Statement
statement
connection
createStatement
ResultSet
executequery
setString
getString

Supported SQL Grammar

- ALL
- AND
- DELETE
- DISTINCT
- FROM
- INSERT
- INTO
- OR
- SELECT
- SET
- UPDATE
- VALUE
- VALUES
- WHERE

SELECT Statement

```
public boolean getModelDetails(InputMessage inputMessage,
ModelOutput modelOutput) throws IMSException {
   // Parse the input message for ModelTypeCode
   String queryString = “SELECT * from Model where ModelTypeCode = “
   + “‘” + inputMessage.getString(“ModelTypeCode”).trim() + “‘”;

   // Create a statment and execute it to get a ResultSet
   try {
      Statement statement = connection.createStatement();
      ResultSet results = statement.executeQuery(queryString);
      // Send back the result of the query
      // Note: since "ModelTypeCode" is unique - only 1 row
      // is returned
      if (results.next()) {
         modelOutput.setString(“ModelTypeCode”,
            results.getString(“Type”).trim());
         return true;
      }
      else {
         reply(“Unknown Type”);
         return false;
      }
   }
   catch (SQLException e) {
      reply(“Query Failed:”+ e.toString());
      return false;
   }
}
}
```

FROM Statement

```
SELECT CarModel
FROM Model
WHERE CarYear='2000'
```

UPDATE Statement

```
String updateString = “UPDATE Order “
        + “SET SerialNo = ‘”
        + inputMessage.getString(“StockVINumber”).trim()
        + “‘, “
        + “DeliverDate = ‘”
        + inputMessage.getString(“Date”).trim()
        + “© WHERE OrderNumber = ‘”
        + inputMessage.getString(“OrderNumber”).trim()
        + “‘”;
```

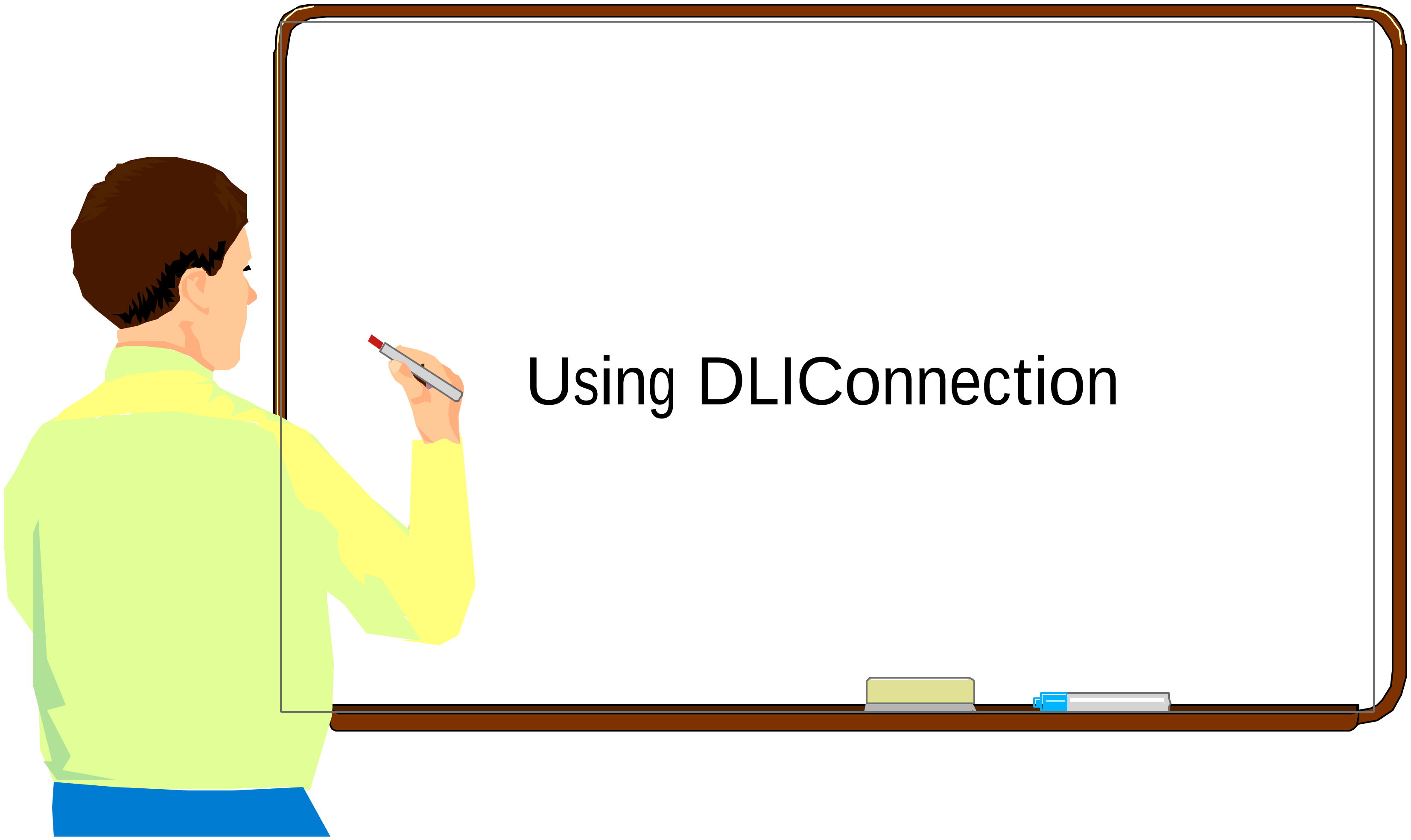
DELETE Statement

```
String updateString = “DELETE from Order where “
        + “Dealer.DealerNumber = ’”
        + dealerDesired+ “‘ AND “
        + “OrderNumber = ‘” + orderDesired + “‘”;
try {
        Statement statement = connection.createStatement();
        int results = statement.executeUpdate(updateString);
        ...
}
catch (SQLException e) {
...
}
```

INSERT Statement

```
String insertString = “INSERT INTO Sales “
        + “(DateSold, PurchaserLastName, PurchaserFirstName, “
        + “PurchaserAddress, SoldBy, StockVINumber)”
        + “ VALUES (‘07032000’, ‘Beier’, ‘Otto’, “
        + “’101 W. 1st Street, Smalltown, CA’, “
        + “’S123’ ’1ABCD23E4F5678901234’) “
        + “WHERE Dealer.DealerNumber = ’A123’”
        + “‘ AND ModelTypeCode = ‘K1’)”;
```

Using DLIConnection



DLIConnection Statements

- DLIConnection
  - Represents a connection to an IMS database
  - DLISegment
    - Segments in an IMS database
    - DLITypeInfo
      - Field Type, Offset and Length
    - SSA
      - Segment Search Argument
    - SSAList
      - Collection of SSAs
    - SSAQualificationStatement
      - logical qualification statements to an SSA. Supports key fields and search fields
    - DLIRecord
      - Set of segment occurrences from a root to a leaf segment
    - DLISegmentInfo
      - links a DLISegment object to it's parent in the hierarchy
    - DLIDatabaseView
      - Applications view of segments in a DL/I Database

Appl. Programming using DLIConnection

1. Provide a subclass of DLIDatabaseView
   - defines the segment hierarchy
2. Provide a subclass of DLISegment
   - for each segment accessed
3. Create a DLIConnection object
   - to access the database
4. Create an SSAList object
5. Use DLIConnection
   - read, write, or delete segments

3. Create a DLIConnection object

- Provide a DLIDatabaseView object

```
DealerDatabaseView dealerView = new DealerDatabaseView();
DLIConnection connection = DLIConnection.createInstance(dealerView);
```

- Provide the fully-qualified name of the DLIDatabaseView

4. Using SSAs to access DL/I Information

```
// Create an SSAList
SSAList modelSSAList = SSAList.createInstance();

// Construct an unqualified SSA for the Dealer segment
SSA dealerSSA = SSA.createInstance(“Dealer”);

// Construct a qualified SSA for the Model segment
SSA modelSSA = SSA.createInstance(“Model”, “CarMake”, SSA.EQUALS, “Alfa”);

// Add an additional qualification statement
modelSSA.addQualification(SSA.AND, "CarYear", SSA.EQUALS, "1989");

// Add the SSAs to the SSAList
modelSSAList.addSSA(dealerSSA);
modelSSAList.addSSA(modelSSA);
```

SSA
SSAList
SSAList.createInstance
modelSSAList.addSSA

5. Use of DLIConnection

```
if !connection.getUniqueSegment(Model,ModelSSA)
   {
   //Data not found in the database
   }
else
   {
   //Model has been retrieved
   }
```

Message Queue

Input Message

- Construct an InputMessage.
  - Allocate the byte array for the input message.
  - Total data length is 49 bytes.
- The input message data has the following layout format as defined in the MID:

Reserved	(4 bytes)
Process Code	(8 bytes)
Input Data : (37 bytes total)	
– Last Name	(10 bytes)
– First Name	(10 bytes)
– Extension Number	(10 bytes)
– Internal Zip Code	(7 bytes)

Define an Input Message

```
package samples.jdbc;
import com.ibm.ims.base.*;

{
   static DLITypeInfo[] fieldInfo = {
      new DLITypeInfo(“Reserved”,DLITypeInfo.CHAR,1,4),
      new DLITypeInfo(“ProcessCode”,DLITypeInfo.CHAR,5,8),
      new DLITypeInfo(“LastName”,DLITypeInfo.CHAR,13,10),
      new DLITypeInfo(“FirstName”,DLITypeInfo.CHAR,23,10),
      new DLITypeInfo(“Extension”,DLITypeInfo.CHAR,33,10),
      new DLITypeInfo(“ZipCode”,DLITypeInfo.CHAR,43,7)
   };

public InputMessage()
{
   super(fieldInfo,49, false);
}
}
```

Output Message

- Construct an OutputMessage.
  - Allocate the byte array for the output message.
  - Total data length is 89 bytes.
- The output message data has the following layout format:

Output Message	(40 bytes)
Process Code	(8 bytes)
Output Data: (37 bytes)	
– Last Name	(10 bytes)
– First Name	(10 bytes)
– Extension Number	(10 bytes)
– Internal Zip Code	(7 bytes)
– Segment Number	(4 bytes)

Define an Output Message

```
package samples.jdbc;
import com.ibm.ims.base.*;

public class OutputMessage extends com.ibm.ims.application.IMSFieldMessage
{
   static DLITypeInfo[] fieldInfo = {
      new DLITypeInfo(“Message”,DLITypeInfo.CHAR,1,40),
      new DLITypeInfo(“ProcessCode”,DLITypeInfo.CHAR,41,8),
      new DLITypeInfo(“LastName”,DLITypeInfo.CHAR,49,10),
      new DLITypeInfo(“FirstName”,DLITypeInfo.CHAR,59,10),
      new DLITypeInfo(“Extension”,DLITypeInfo.CHAR,69,10),
      new DLITypeInfo(“ZipCode”,DLITypeInfo.CHAR,79,7),
      new DLITypeInfo(“SegmentNumber”,DLITypeInfo.CHAR,86,4)
   };
/**
*/
public OutputMessage()
{
   super(fieldInfo,89, false);
}
}
```

Commit the updates

- Explicit COMMIT is required for Java programs
  - IMSTransaction.getTransaction().commit();
  - Else abend U0118

Retrieve a message

```
while(messageQueue.getUniqueMessage(inputMessage))
{
   if (!inputMessage.getString("ModelTypeCode").trim().equals(""))
   {
      if (getModelDetails(inputMessage, modelOutput))
         messageQueue.insertMessage(modelOutput);
   }
   else
   {
      reply("Invalid Input");
   }
   IMSTransaction.getTransaction().commit();
   }
```

Insert a message

```
while(messageQueue.getUniqueMessage(inputMessage))
{
   if (!inputMessage.getString(“ModelTypeCode”).trim().equals(““))
   {
      if (getModelDetails(inputMessage, modelOutput))
         messageQueue.insertMessage(modelOutput);
   }
   else
   {
      reply(“Invalid Input”);
   }
   IMSTransaction.getTransaction().commit();
   }
```

Commit the Updates

```
while(messageQueue.getUniqueMessage(inputMessage))
{
   if (!inputMessage.getString(“ModelTypeCode”).trim().equals(““))
   {
      if (getModelDetails(inputMessage, modelOutput))
         messageQueue.insertMessage(modelOutput);
   }
   else
   {
      reply(“Invalid Input”);
   }
   IMSTransaction.getTransaction().commit();
   }
```

JDBC Application Code

```
package dealership.application;
import com.ibm.ims.base.*;
import com.ibm.ims.application.*;
import com.ibm.ims.db.*;
import java.sql.*;

public class IMSAuto extends IMSApplication
{
    IMSMessageQueue messageQueue = null;
    InputMessage inputMessage = null;ModelOutput modelOutput = null;
    Connection connection = null;
    public IMSAuto()
    {
        super();
    }
    public static void main(String args[])
    {
        IMSAuto imsauto = new IMSAuto();
        imsauto.begin();
    }
    public void doBegin() throws IMSException
    {
        messageQueue = new IMSMessageQueue();
        inputMessage = new InputMessage();
        modelOutput = new ModelOutput();
```

JDBC Application Code (continued)

```
   try
   {
      Class.forName("com.ibm.ims.db.DLIDriver");
      connection = DriverManager.getConnection
      ("jdbc:dli:dealership.application.DealerDatabaseView");
   }
   catch (Exception e)
   {
      reply("Connection not established");
   }
   while(messageQueue.getUniqueMessage(inputMessage))
   {
      if (!inputMessage.getString("ModelTypeCode").trim().equals(""))
      {
         if (getModelDetails(inputMessage, modelOutput))
            messageQueue.insertMessage(modelOutput);
      }
      else
      {
         reply("Invalid Input");
      }
      IMSTransaction.getTransaction().commit();
      }
   }
public void reply(String errmsg) throws IMSException{
   ErrorMessage errorMessage = new ErrorMessage();
   errorMessage.setString("MessageText",errmsg);
   messageQueue.insertMessage(errorMessage); }}
```

VisualAge for Java

```
This space intentionally left blank,
so that I can demonstrate IBM VisualAge for
Java here.
```

Summary

- Java can be used today
  - MPP, IFP, BMP
- Features for Java Programmers
- Features for COBOL Programmers
- Thanks for attending the IMS Technical Conference, Miami Florida