A13

IMS Java Application Development

Kyle Charlet

IMS Technical Conference

Miami Beach, FL October 22-25, 2001

RETURN TO INDEX

Overview

- **IMS Java Classes**
  - What it is
  - Why use it
- **IMS Java Class Library Architecture**
- **Metadata**
  - Types
  - Segment Definition
  - Database Definition
- **IMS Database Access**
  - SSA Layer
  - JDBC Layer
- **Tracing**

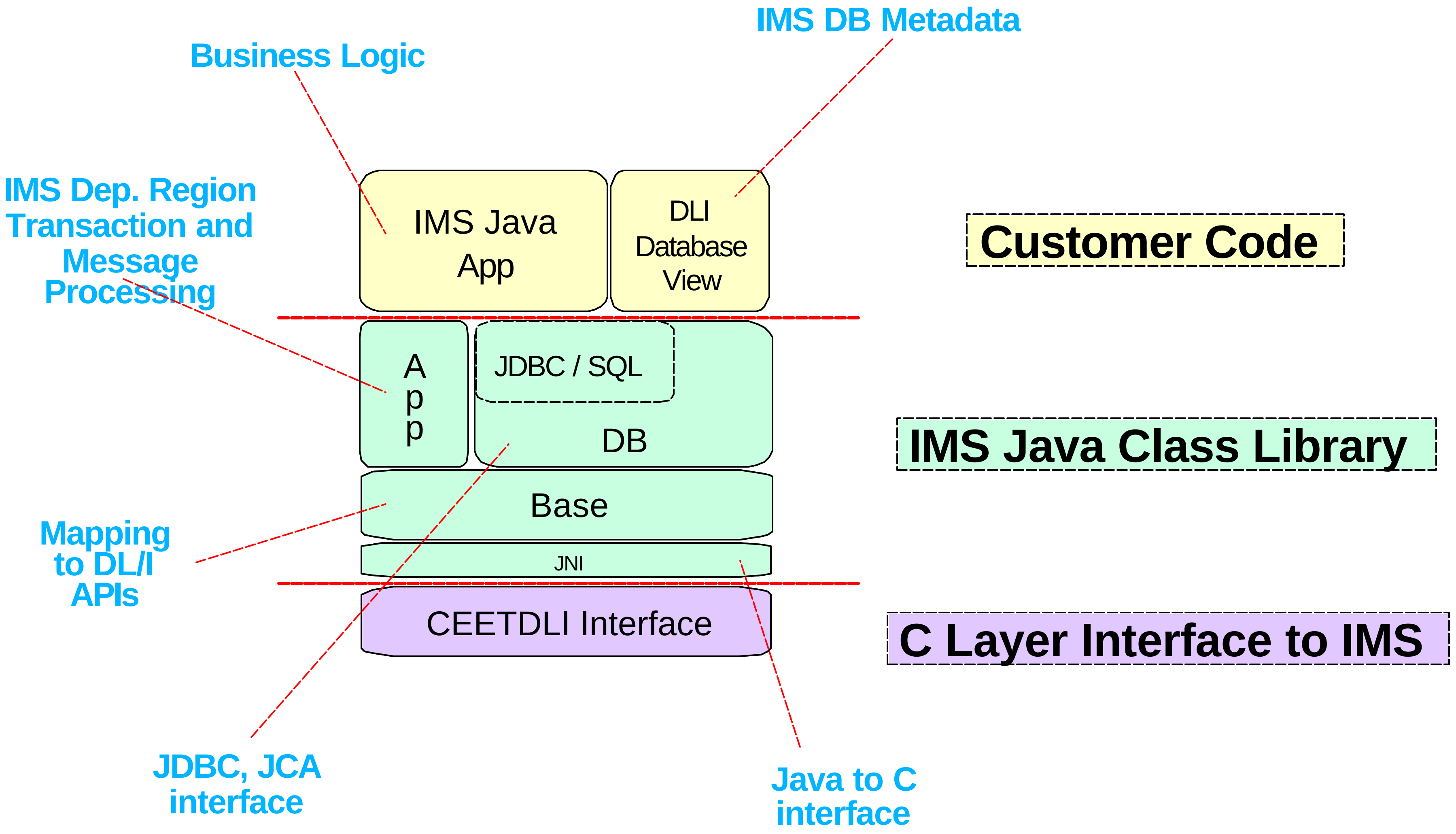
What is IMS Java?

- **A new feature in IMS v7**
- **A set of classes that...**
  - Offers Java support to access IMS Databases
  - Enables SQL access through the JDBC interface
- **Java Virtual Machine (JVM) support in Dependent Regions**
  - JDK 1.3 support
  - JDBC 2.1 support
  - Just-In-Time (JIT) compilation
  - To be made available on IMS v7
- **High Performance Java (HPJ) compiled**
  - Runs as a Language Environment run unit
  - JDK 1.1.8
  - JDBC 1.0

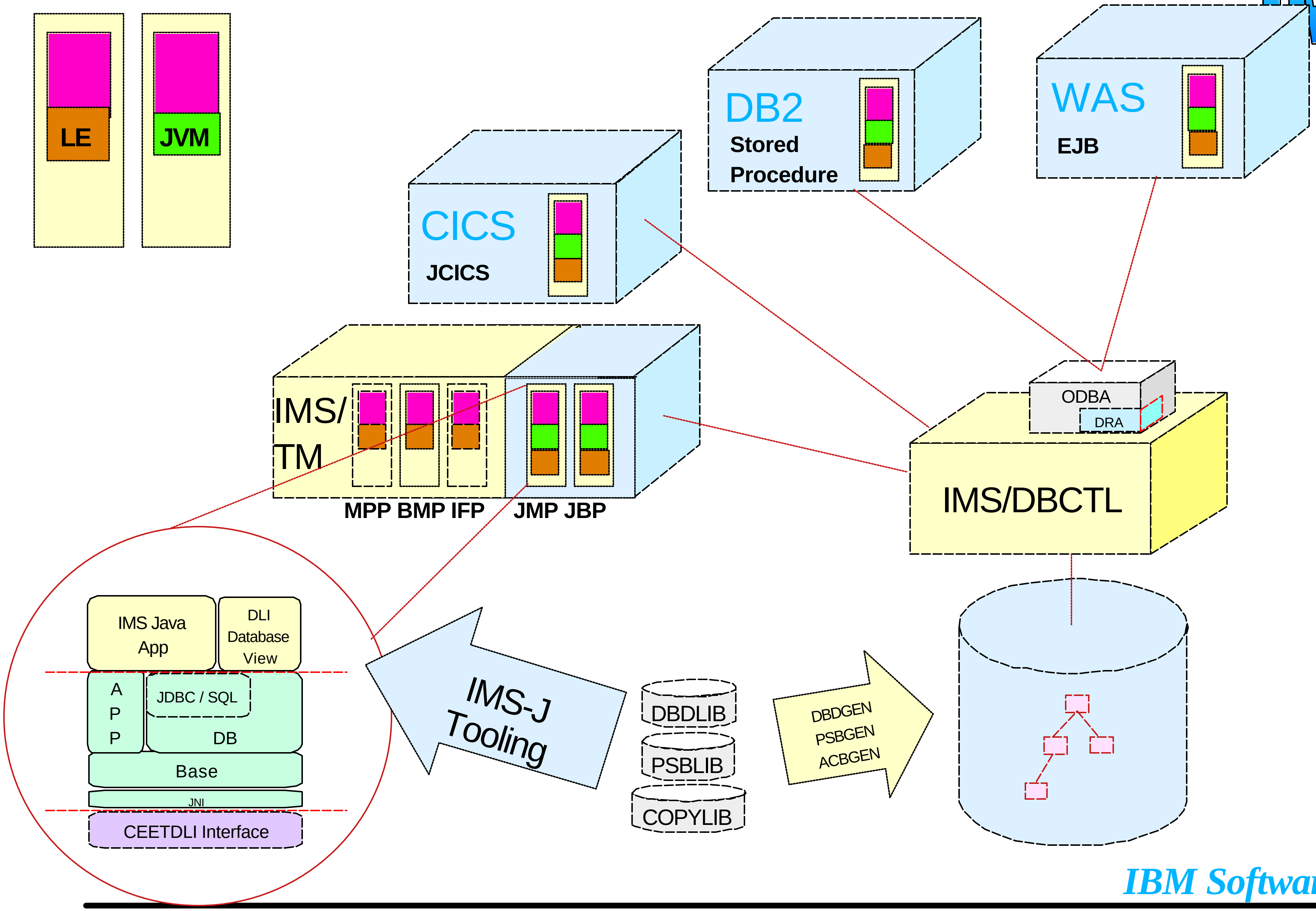
Why IMS Java?

- **Colleges teach Java, very few still teach COBOL**
- **Colleges teach relational DBs with SQL access, very few teach hierarchical with SSA access**
- **JDBC is an industry standard**
  - Minimizes specific backend DB knowledge of IMS
- **Customer requests for Java support**

Java Class Library



IMS Java - The Big Picture



IBM Software

Other Pieces

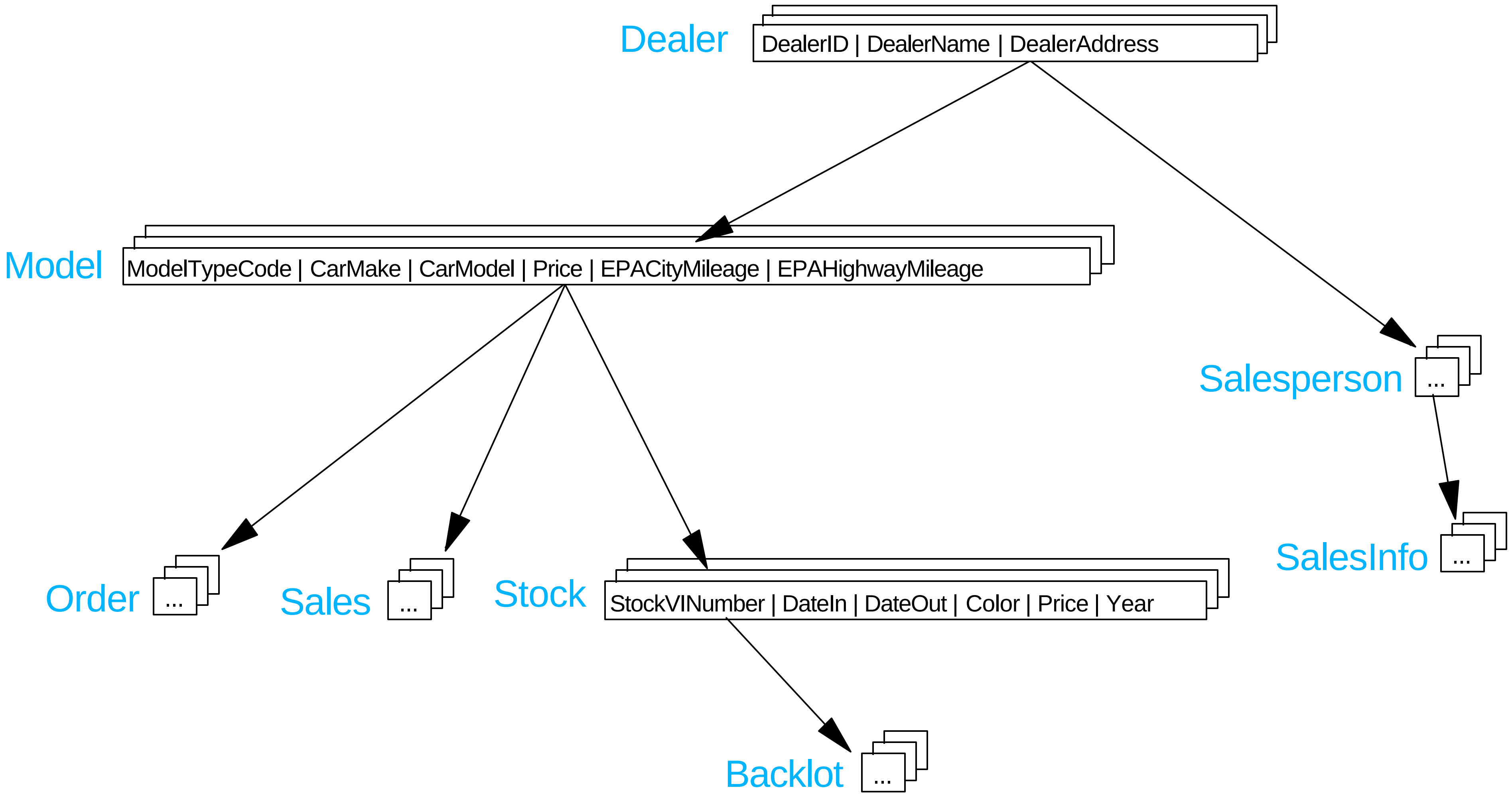
- **Compilers**
  - javac - Sun, VisualAge
  - HPJ - IBM (Toronto)
- **Runtime**
  - Language Environment
  - JVM Region Support (JMP, JBP)
  - Resetable Java Virtual Machine
  - ODBA (WebSphere, DB2)
  - Remote Recovery Services (RRS)

Overview

- **IMS Java Classes**
  - What it is
  - Why use it
- **IMS Java Class Library Architecture**
- **Metadata**
  - Types
  - Segment Definition
  - Database Definition
- **IMS Database Access**
  - SSA Layer
  - JDBC Layer
- **Tracing**

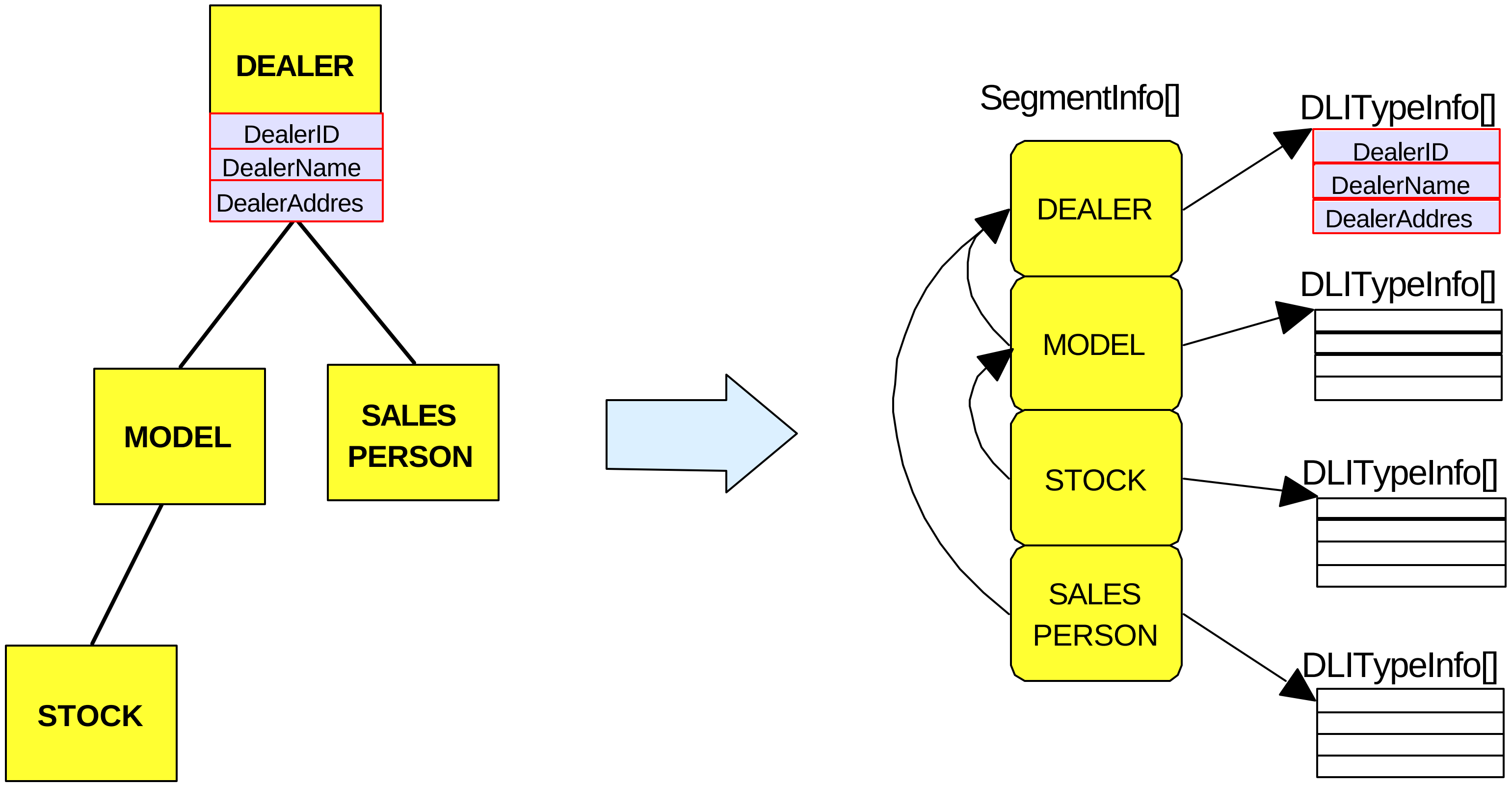
IBM Software

Database layout


Dealer
DealerID | DealerName | DealerAddress
Model
ModelTypeCode | CarMake | CarModel | Price | EPACityMileage | EPAHighwayMileage
Salesperson
...
SalesInfo
...
Order
...
Sales
...
Stock
StockVINumber | DateIn | DateOut | Color | Price | Year
Backlot
...

IBM Software

IMS Metadata



DBDLIB, PSBLIB, COPYLIB

DLIDatabaseView

COBOL to IMS Java Datatypes

Copybook Format	DLITypeInfo Constant	Java Type
PIC X	CHAR	java.lang.String
PIC 9 BINARY	(see next table)	(see next table)
COMP-1	FLOAT	float
COMP-2	DOUBLE	double
PIC 9 COMP-3	PACKEDDECIMAL	java.math.BigDecimal
PIC 9 DISPLAY	ZONEDDECIMAL	java.math.BigDecimal

Digits	Storage Size	DLITypeInfo Constant	Java Type
1 through 4	2 bytes	SMALLINT	short
5 through 9	4 bytes	INTEGER	int
10 through 18	8 bytes	BIGINT	long

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) BINARY	new DLITypeInfo("FieldB", DLITypeInfo.SMALLINT, 26, 2)
FIELD-C PIC 9(6) BINARY	new DLITypeInfo("FieldC", DLITypeInfo.INTEGER, 28, 4)
FIELD-D PIC 9(12) BINARY	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 9(4)V99          new DLITypeInfo("FieldA", "9(4)V99", DLITypeInfo.ZONEDDECIMAL, 1, 6)
FIELD-B  PIC S999 COMP-3      new DLITypeInfo("FieldB", "S999", DLITypeInfo.PACKEDDECIMAL, 7, 2)
DATE.
     DD      PIC X(2)         new DLITypeInfo("Date", "ddMMyyyy", DLITypeInfo.DATE, 9, 8)
     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)

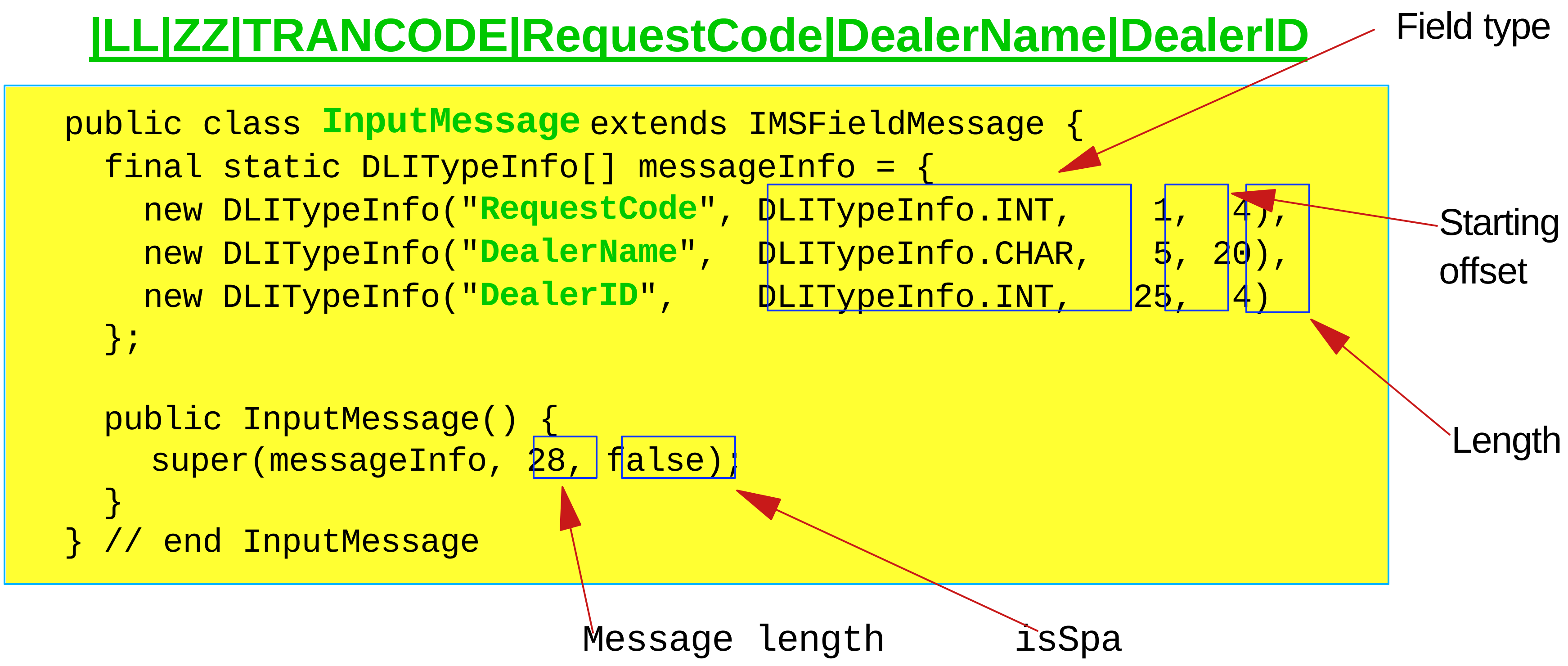
Examples:

```
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 in a zoned or packed field are the following two characters: 9 and '.'

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.

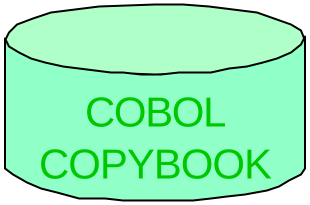
Define Output Messages

```java
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);
  }
}
```

Repeating Fields



```
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);
  }
}
```

Repeat
Count

Total Length = 60*100 + 4

Starting
Offset

Group
Length

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

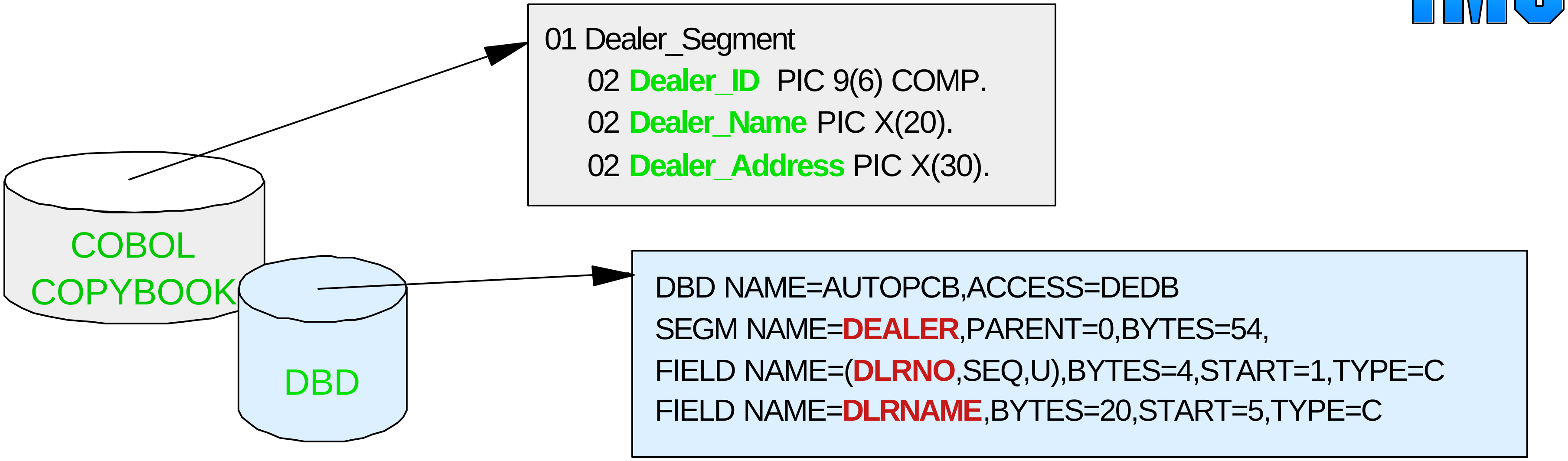
Example: access the fourth "Color" in the ModelOutputMessage

```
        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)
};
```

Define Database Segments



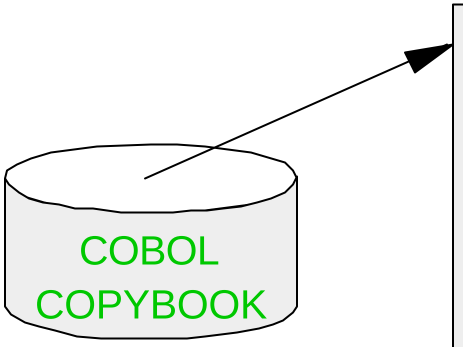
```
01 Dealer_Segment
    02 Dealer_ID  PIC 9(6) COMP.
    02 Dealer_Name PIC X(20).
    02 Dealer_Address PIC X(30).
```

```
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
```

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

static DLISegment dealerSegment =
        new DLISegment("DealerSeg","DEALER", dealerInfo, 54);
                            ⋮
```

Redefining Fields



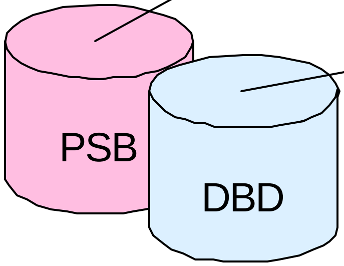
```
01 Dealer_Segment
     02 Dealer_ID  PIC X(6) COMP.
     02 Dealer_Name PIC X(20).
     02 Dealer_Address PIC X(30)
          05 Dealer_Street PIC X(14).
          05 Dealer_City PIC X(14).
          05 Dealer_State PIC X(2).
```

```
    static DLITypeInfo[] dealerInfo = {
      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)
    };

    static DLISegment dealerSegment =
            new DLISegment("DealerSeg","DEALER", dealerInfo, 54);
                               ⋮
```

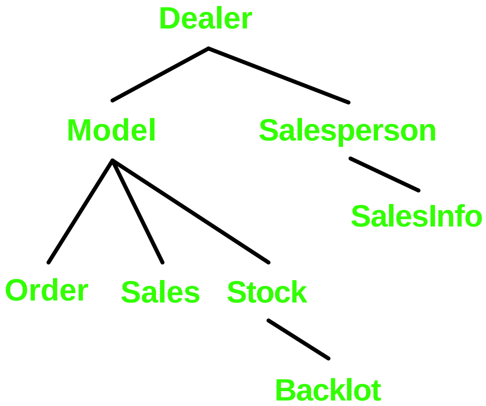
Define Database Layout

```
PCB    TYPE=DB,DBDNAME=AUTODBD,PROCOPT=A,KEYLEN=4,PCBNAME=AUTOPCB
PSBGEN PSBNAME=AUTOPSB
```



```
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
```

```
public class DealerDatabaseView extends DLIDatabaseView {
                 •
                 •
                 •
  static DLISegmentInfo[] autoPCBSegments = {
    new DLISegmentInfo( dealerSegment,      ROOT),
    new DLISegmentInfo( modelSegment,       0),
    new DLISegmentInfo( orderSegment,       1),
    new DLISegmentInfo( salesSegment,       1),
    new DLISegmentInfo( stockSegment,       1),
    new DLISegmentInfo( backLotSegment,     4),
    new DLISegmentInfo( salesPersonSegment, 0),
    new DLISegmentInfo( salesInfoSegment,   6)
  };
  public DealerDatabaseView() {
    super("AUTOPSB", "DealerPCB", "AUTOPCB", autoPCBSegments);
    addDatabase("PCB2Alias", "AUTOPCB2", autoPCB2Segments);
  }
}
```



Overview

- **IMS Java Classes**
  - What it is
  - Why use it
- **IMS Java Class Library Architecture**
- **Metadata**
  - Types
  - Segment Definition
  - Database Definition
- **IMS Database Access**
  - SSA Layer
  - JDBC Layer
- **Tracing**

SSA Layer

- **Java methods to access IMS Databases analogous to COBOL methods**
- **Executing SSA database requests**
  - Create connection to database
  - Build SSAList and make database call
  - Process results
  - Close connection to database

COBOL	Java
GHU	getUniqueSegment(DLISegment,SSAList)
GHU	getUniqueSegment(SSAList)
GHN	getNextSegment(DLISegment,SSAList)
GHN	getNextSegment(SSAList)
GHNP	getNextSegmentInParent(DLISegment,SSAList)
GHNP	getNextSegmentInParent(SSAList)
ISRT	insertSegment(DLISegment,SSAList)
REPL	replaceSegment(DLISegment)
DLET	deleteSegments(DLISegment)

Note: All calls accessing segments are HOLD calls.

Creating a Connection

- Connections are made to a PSB by passing in the PSB Database Metadata (DLIDatabaseView)

```
DLIConnection.createInstance(DLIDatabaseView);
```

Building an SSAList

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

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

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

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

ssaList.addSSA(dealerSSA);
ssaList.addSSA(stockSSA);

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

Recall: `super("AUTOPSB", "DealerPCB", "AUTOPCB", autoPCBSegments);`

IBM Software

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"));
}
```

Datatype Conversion

	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

JDBC Explained

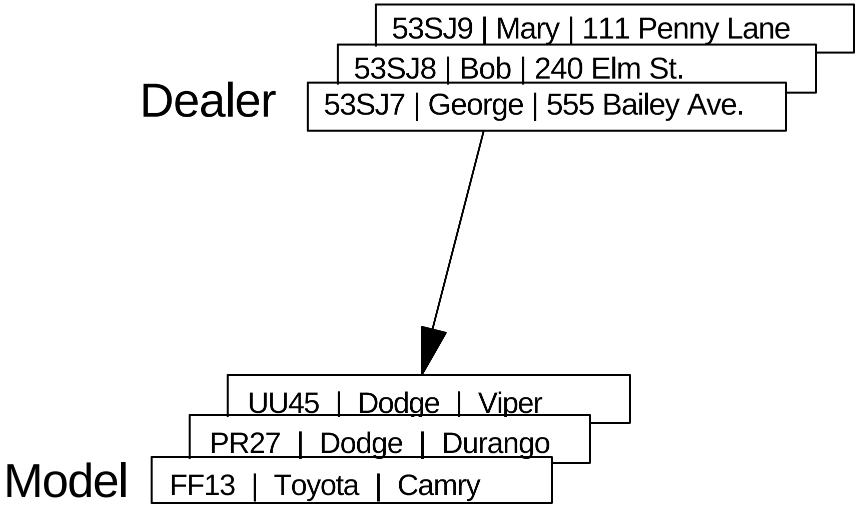
- **Defines a standard Java API for accessing relational databases**

- **Provides an API for sending SQL statements to a database and processing the tabular data returned by the database**

- **Executing JDBC query statements**
  - Establish and open connection to database
  - Execute query to obtain results
  - Process results
  - Close connection

Hierarchical vs. Relational

Hierarchical DB design



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

Equivalent relational design

Dealer Table

DealerID	DealerName	DealerAddress
53SJ7	George	555 Bailey Ave.
53SJ8	Bob	240 Elm St.
53SJ9	Mary111	Penny Ln.
...	...	...

Model Table

ID	Make	Model	Dealer
UU45	Dodge	Viper	53SJ7
PR27	Dodge	Durango	53SJ7
FF13	Toyota	Camry	53SJ7
PR27	Dodge	Durango	53SJ8
...	...	...	...

foreign key captures relationship

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

Establish and Open Connection

- **Load the IMS Java JDBC driver**
- **Get IMS Java Connection from Driver Manager**
  - URL must begin with 'jdbc:dli:' followed by fully qualified class name

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

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

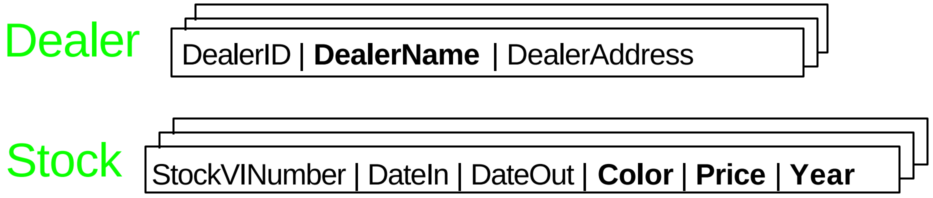
- **DataSource will be recommended way to get a Connection soon**
  - Represents a physical data source
  - DataSource object stored persistently

Executing a Query

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

*** make sure you PCB qualify the segment in the FROM clause**

recall...



Dealer

DealerID | **DealerName** | DealerAddress

Stock

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

```
super("AUTOPSB", "DealerPCB", "AUTOPCB", autoPCBSegments);
addDatabase("PCB2Alias", "AUTOPCB2", autoPCB2Segments);
addDatabase("PCB3Alias", "AUTOPCB3", autoPCB3Segments);
```

```
PCB  TYPE=DB, ..., PCBNAME=AUTOPCB
PCB  TYPE=DB, ..., PCBNAME=AUTOPCB2
PCB  TYPE=DB, ..., PCBNAME=AUTOPCB3
PSBGEN PSBNAME=AUTOPSB
```

Prepared Statements

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

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

pstmt.setString(1, "Fjord");

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

*** make sure you PCB qualify the segment in the UPDATE clause**

recall...

Dealer

DealerID | **DealerName** | DealerAddress

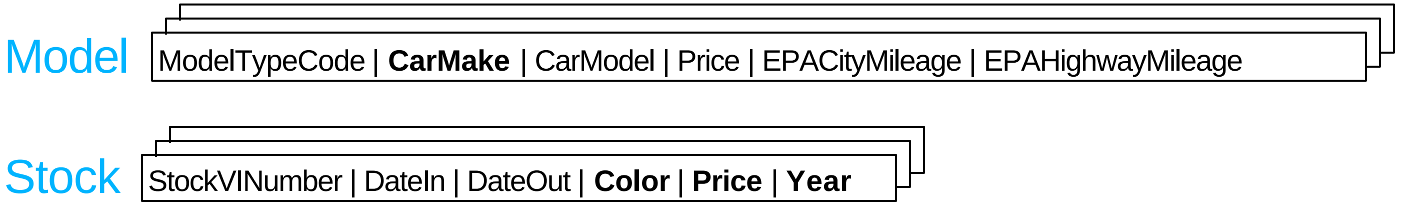
```
super("AUTOPSB", "DealerPCB", "AUTOPCB", autoPCBSegments);
```

Processing 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**

Putting it all together

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

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

Statement stmt = con.createStatement("SELECT Model.CarMake, Stock.Year, Stock.Price " +
                                     "FROM DealerPCB.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 DealerPCB.Dealer SET DealerName = 'Fiord' WHERE DealerName = ?");

pstmt.setString(1, "Fjord");

int updateCount = pstmt.executeUpdate();

con.close();
```

Overview

- **IMS Java Classes**
  - What it is
  - Why use it
- **IMS Java Class Library Architecture**
- **Metadata**
  - Types
  - Segment Definition
  - Database Definition
- **IMS Database Access**
  - SSA Layer
  - JDBC Layer
- **Tracing**

IBM Software

Java Application Tracing

- **Mechanism for writing trace data to a user-supplied output stream**
  - stderr, stdout, file
- **Output is XML (with minor tweak), therefore easily parsed**
  - no XML header or main element
- **Tracing is implemented by most library methods and (most) library-created exceptions**
- **Design allows separation of library tracing from application tracing**

Enable Library Tracing

Establish Output Stream

```
IMSTrace.setOutputStream(System.err);
                    or
FileWriter fileWriter = newFileWriter("/tmp/PrizeDrawing.trace");
IMSTrace.setOutputWriter(fileWriter);
```

Set Trace Level

```
IMSTrace.libTraceLevel = IMSTrace.TRACE_DATA3;
```

Turn tracing on

```
IMSTrace.traceOn = true;
```

IMSTrace.libTraceLevel values

0	none
TRACE_EXCEPTIONS	least
TRACE_CTOR1	
TRACE_METHOD1	
TRACE_DATA1	
TRACE_CTOR2	
TRACE_METHOD2	
TRACE_DATA2	
TRACE_CTOR3	
TRACE_METHOD3	
TRACE_DATA3	most

Note: To ensure maximum tracing, add the trace enabling code to a static block.

Java Application Tracing

- **Supports either Writer or OutputStream**
  - setOutputWriter, setOutputStream
- **Auto-tagged "convenience" methods**
  - logEntry(String methodName)
  - logExit(String methodName)
  - logParm(String parmName, String value)
  - logParm(String parmName, byte[] value)
  - logResult(String result)
  - logResult(byte[] result)

Make sure method name is unique and identical on entry and exit calls

Also 2 and 3 parm versions

- **Non-tagged method**
  - logData(String data)
- **All methods check IMSTrace.traceOn before logging**
- **Stream is flushed after every write**

IBM Software

Java Application Tracing (cont)

- **One IMSTrace object per thread (constructor is private)**
  - IMSTrace.currentTrace() to retrieve
  - IMSTrace.setTIDTracing(true) to log thread ID
- **Binary data has a maximum trace length**
  - default is 50 bytes
  - Call IMSTrace.setMaxBinaryLength to change

Note: non-IMS applications can be multi-threaded

Sample Trace Output

```
<?xml version='1.0'?>
<trace>
<entry>main</entry>
  <entry>OrderStatusJDBC</entry>
  <exit>OrderStatusJDBC</exit>
  <entry>IMSApplication.begin()</entry>
    <entry>IMSApplication.initialize()</entry>
    <exit>IMSApplication.initialize()</exit>
    <entry>doBegin</entry>
      <entry>setup</entry>
        <entry>IMSMessageQueue()</entry>
        <exit>IMSMessageQueue()</exit>
       <entry>IMSFieldMessage(DLITypeInfo, int, boolean)</entry>
          <parm>
            <parmName>length</parmName>
            <parmChar>100</parmChar></parm>
          <parm>
            <parmName>isSPA</parmName>
            <parmChar>false</parmChar></parm>
        <exit>IMSFieldMessage(DLITypeInfo, int, boolean)</exit>
        <entry>IMSFieldMessage(DLITypeInfo, int, boolean)</entry>
          <parm>
            <parmName>length</parmName>
            <parmChar>580</parmChar></parm>
          <parm>
            <parmName>isSPA</parmName>
            <parmChar>false</parmChar></parm>
        <exit>IMSFieldMessage(DLITypeInfo, int, boolean)</exit>
...
```

Added lines

Conclusions

- IMS Java allows Java developers to create new applications quickly, easily, and without in-depth IMS knowledge
- Tooling will alleviate headaches with defining metadata
- Tracing is useful!

RETURN TO INDEX