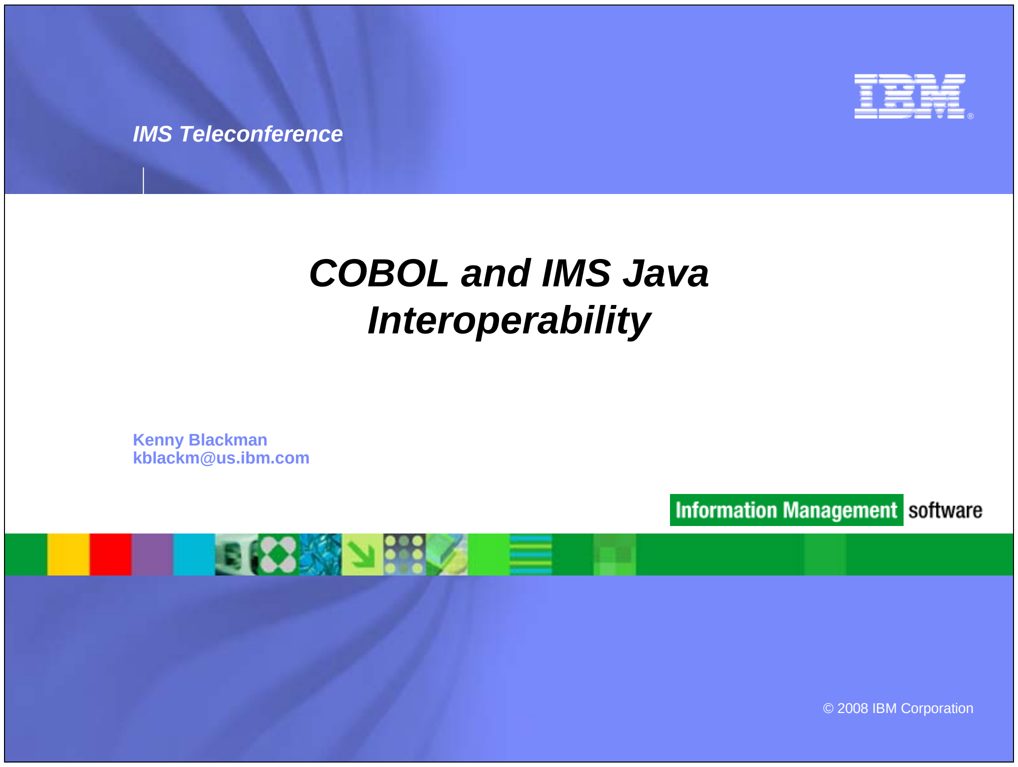

IMS Teleconference

COBOL and IMS Java Interoperability

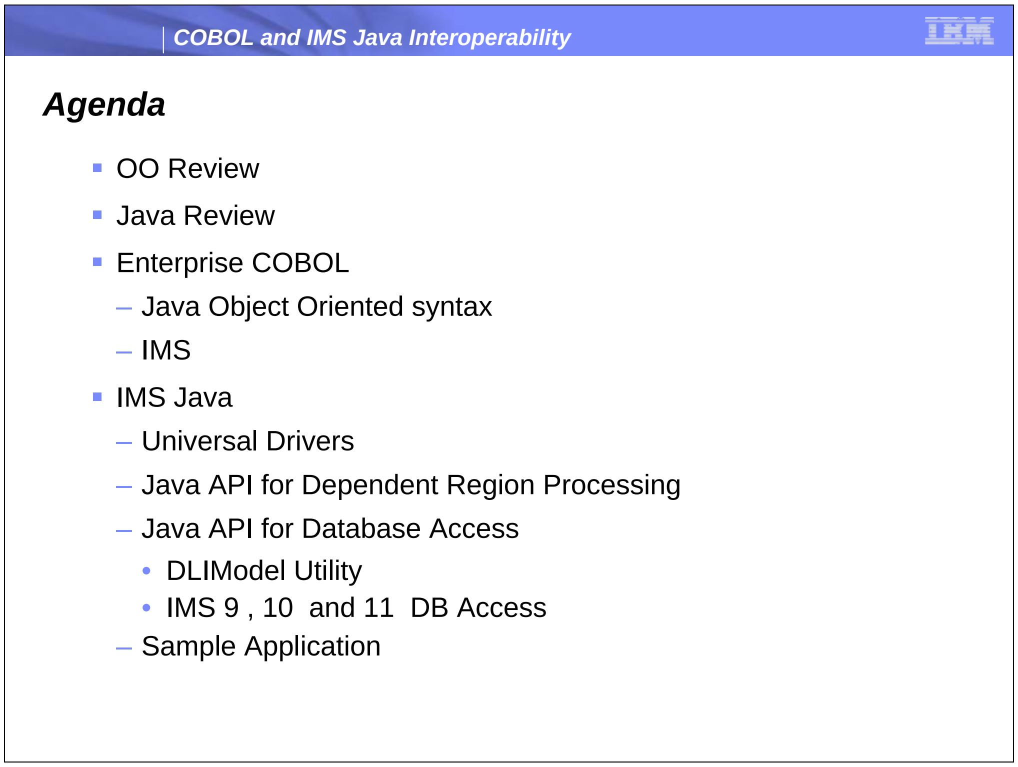
Kenny Blackman
kblackm@us.ibm.com

Information Management software

Abstract:

learn
how you can combine your COBOL business logic
with IMS solutions for Java and gain a more flexible
programming environment. Enterprise COBOL for z/OS®
enables the integration of existing COBOL applications
with IMS Java, while IMS Java provides a complete Java
development solution for both IMS transactions and
IMS databases. In this session, we'll cover the basics
of COBOL and Java interoperability when being used
under IMS, including database access to IMS and
DB2®, calling Java from COBOL, and calling existing
Language Environment (LE) modules from Java using
the Java Native Interface. Possible runtime environments
are the IMS Java Regions persistent Java Virtual
Machine (JVM™) and existing Message Processing
Regions for LE languages

Java has quickly cemented itself as the language of choice for enterprise application development. Due in part to its object-oriented methodology and rapid application development features, many recent and emerging technologies such as SOA, JEE, JCA, and JDBC are centered on Java as the implementation language. This session details how IMS Java applications using the IMS Java class libraries are



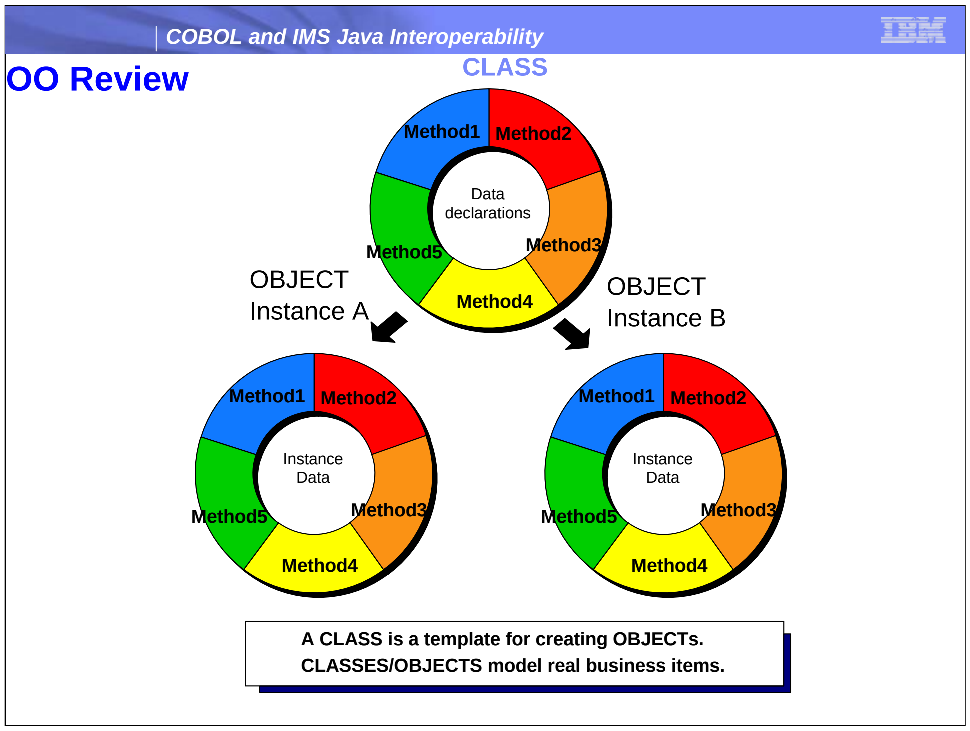
Agenda

- OO Review
- Java Review
- Enterprise COBOL
  - Java Object Oriented syntax
  - IMS
- IMS Java
  - Universal Drivers
  - Java API for Dependent Region Processing
  - Java API for Database Access
    - DLIModel Utility
    - IMS 9 , 10 and 11 DB Access
  - Sample Application

Simplify the development and maintenance of IMS applications by leveraging new Java services as subroutines accessed by existing COBOL business logic

• Use industry standard application APIs, such as JDBC™, J2EE™ and XML, to reduce the programming effort required for building solutions

• Integrate with other products that support the Java API, for example, the Java Message Service support provided by WebSphere® MQ



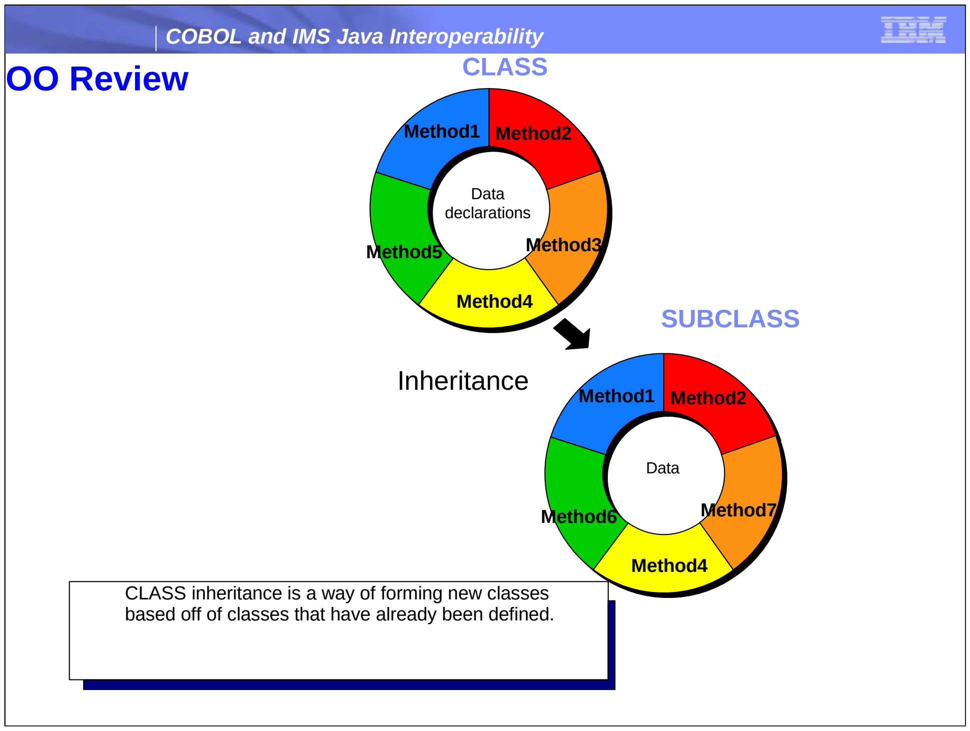
COBOL and IMS Java Interoperability

OO Review

CLASS
Method1
Method2
Data declarations
Method3
Method4
Method5
OBJECT Instance A
OBJECT Instance B
Instance Data

A CLASS is a template for creating OBJECTs.
CLASSES/OBJECTS model real business items.

A *class* is a template that defines the state and the capabilities of an object. Usually a progr
data known as *instance data*, and the capabilities of each instance are called *instance meth*

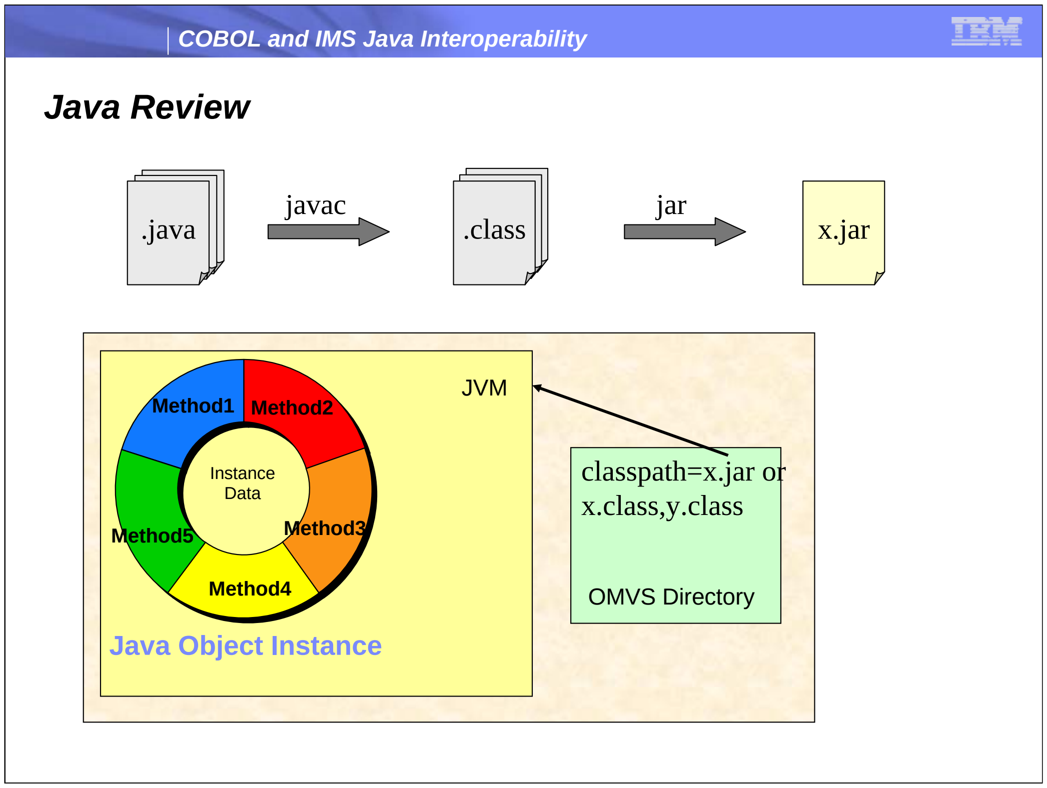


COBOL and IMS Java Interoperability

OO Review

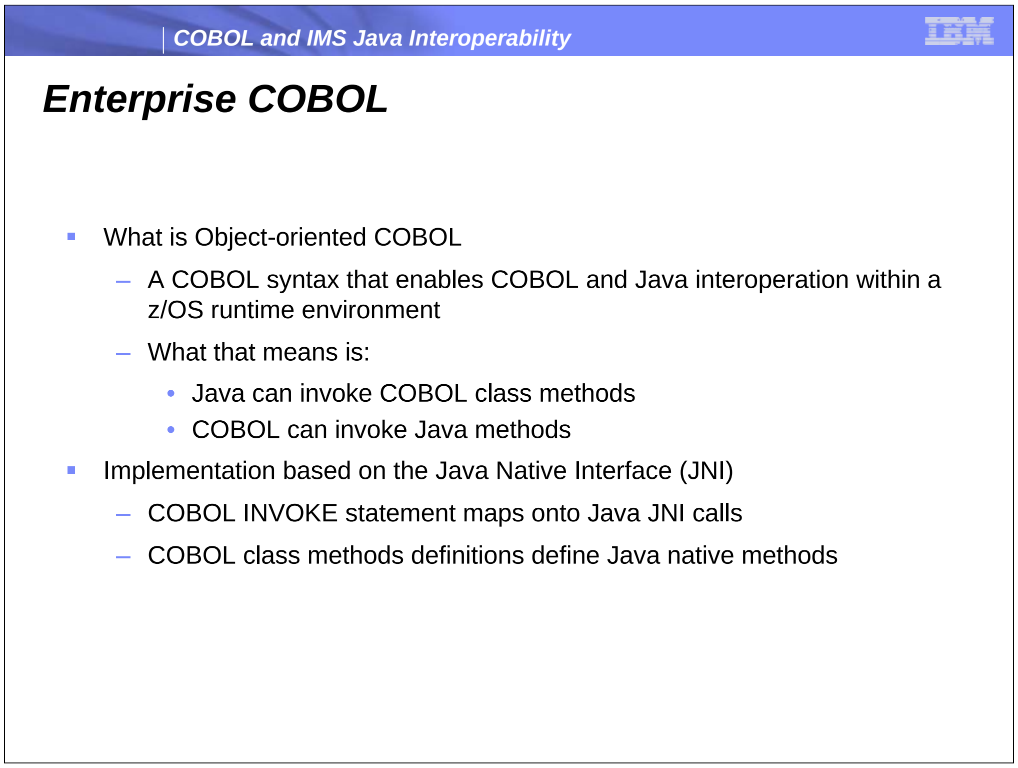
CLASS inheritance is a way of forming new classes based off of classes that have already been defined.

Object-oriented programming allows classes to *inherit* commonly used state and behavior fi


COBOL and IMS Java Interoperability

Java Review

Object instances are automatically freed by the Java runtime system's garbage collection when they are no longer in use. You cannot explicitly free individual objects.



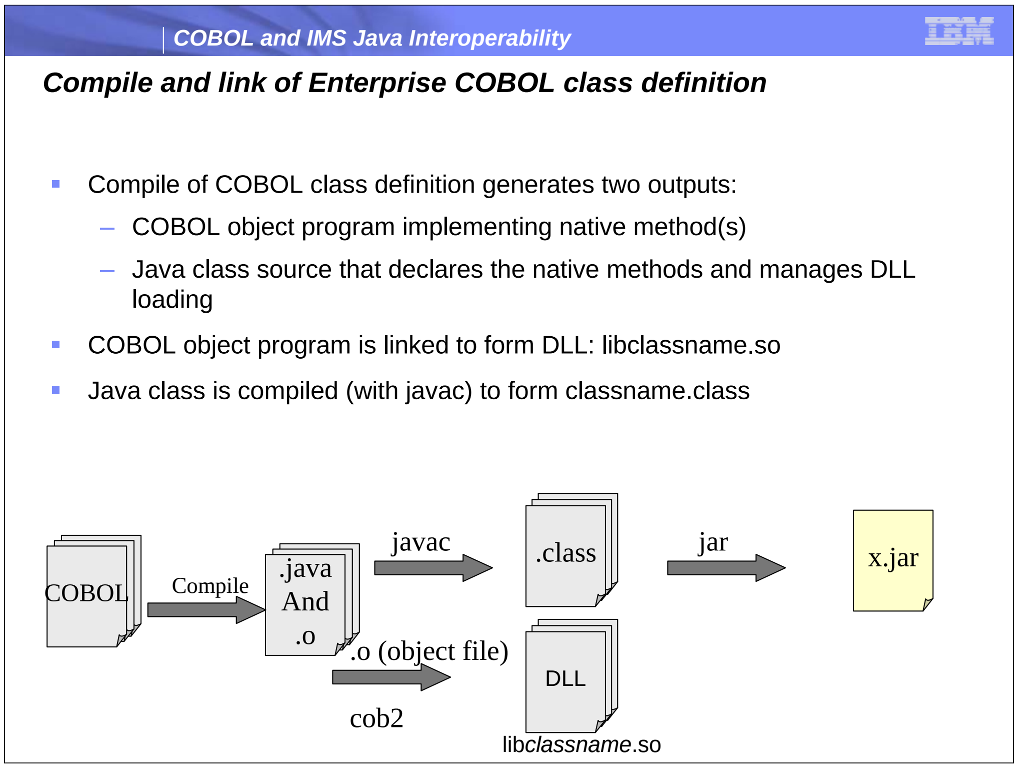
Enterprise COBOL

- What is Object-oriented COBOL
  - A COBOL syntax that enables COBOL and Java interoperation within a z/OS runtime environment
  - What that means is:
    - Java can invoke COBOL class methods
    - COBOL can invoke Java methods
- Implementation based on the Java Native Interface (JNI)
  - COBOL INVOKE statement maps onto Java JNI calls
  - COBOL class methods definitions define Java native methods

Object-oriented COBOL is

A set of syntax rules for defining COBOL that can be used to allow COBOL and JAVA to interact within the same address space. This means that Java can now invoke COBOL and COBOL can in turn invoke Java in a reciprocal relationship.

Java Native Interface is a facility of Java designed for interoperation for non-Java programs. Basically the core of how COBOL and Java can interact with each other. Will look into how the mapping will work in the later slides but it's not terribly important to know the details.


COBOL and IMS Java Interoperability

Compile and link of Enterprise COBOL class definition

- Compile of COBOL class definition generates two outputs:
  - COBOL object program implementing native method(s)
  - Java class source that declares the native methods and manages DLL loading
- COBOL object program is linked to form DLL: libclassname.so
- Java class is compiled (with javac) to form classname.class

It is recommended that you compile, link, and run object-oriented (OO) applications in the z/OS UNIX environment.

When you compile a COBOL class definition, two output files are generated:

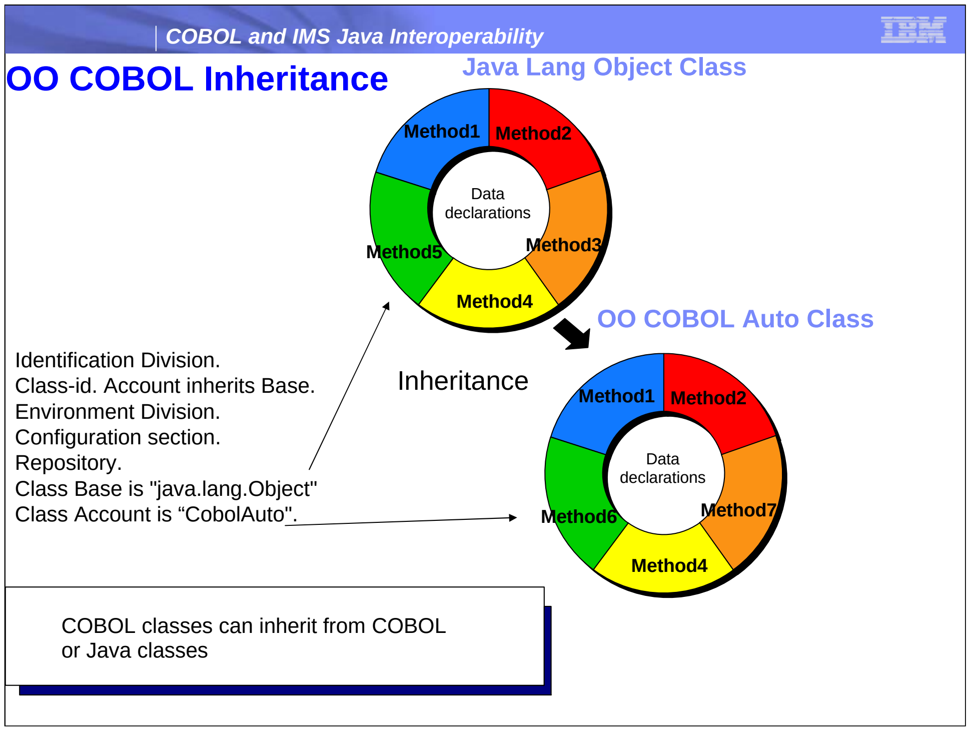
1. The object file (.o) for the class definition.
2. 2 A Java source program (.java) that contains a class definition that corresponds to the COBOL class definition.

Do not edit this generated Java class definition in any way. If you change the COBOL class definition, you must regenerate both the object file and the Java class definition by recompiling the updated COBOL class definition.

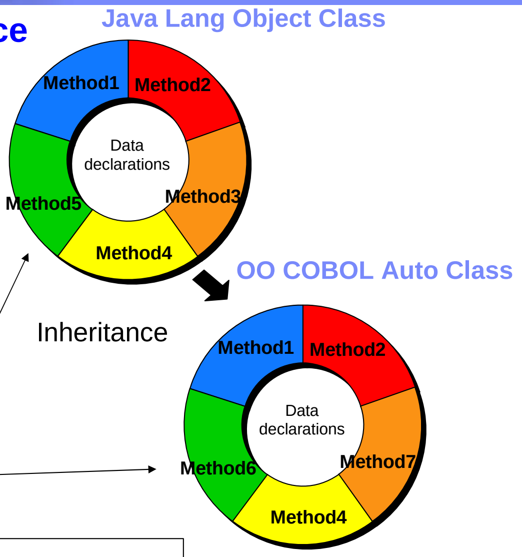
cob2-bdll-olib*classname*.so*classname*.o/usr/lpp/java/IBM/J1.3/bin/classic/libjvm.x/usr/lpp/cobol/lib/igzcjava.x

lib*classname*.so is a DLL

A DLL is a load module or a program object that can be accessed from other separate load modules. A DLL differs from a traditional load module in that it *exports* definitions of programs, functions, or variables to DLLs, DLL applications, or non-DLLs. Therefore, you do not need to link the target routines into the same load module as the referencing routine. When an application references a separate DLL for the first time, the system automatically loads the DLL into memory. In other words, calling a program in a DLL is similar to calling a load module with a dynamic CALL.



OO COBOL Inheritance



Identification Division.
Class-id. Account inherits Base.
Environment Division.
Configuration section.
Repository.
Class Base is "java.lang.Object"
Class Account is “CobolAuto".

COBOL classes can inherit from COBOL or Java classes

Identification Division.

Class-id. Account inherits Base.

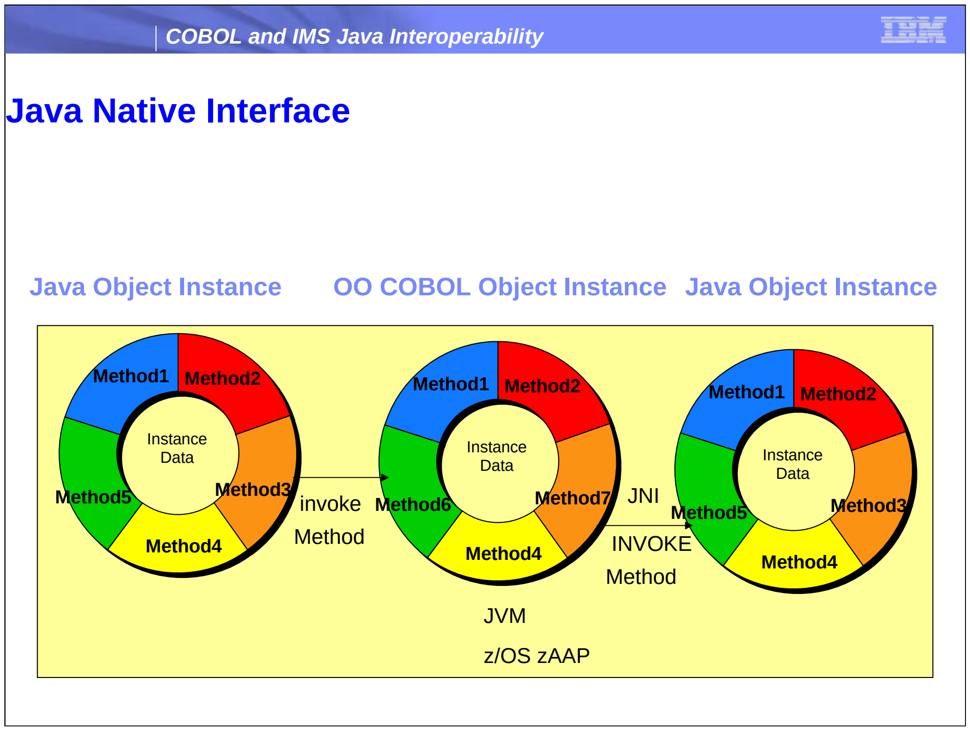
Environment Division.

Configuration section.

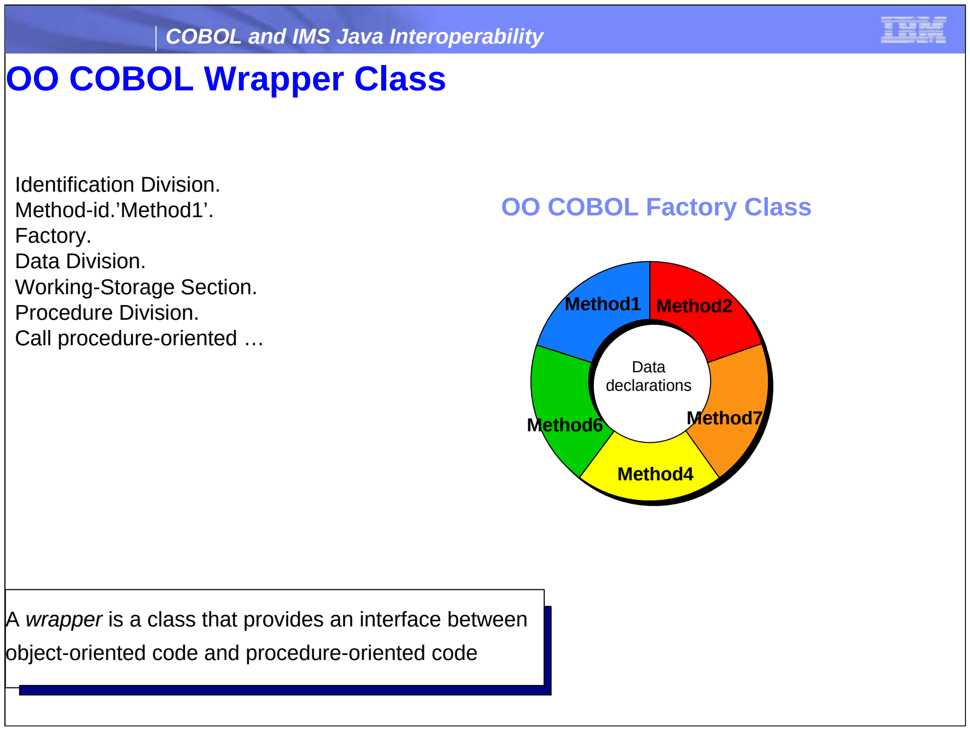
Repository.

Class Base is "java.lang.Object"

Class Account is "Account".

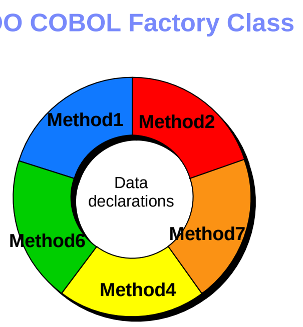


The **Java Native Interface** (**JNI**) is a programming framework that allows Java code runnin
Java applications use java syntax to invoke the COBOL object and COBOL provides INVO



OO COBOL Wrapper Class

Identification Division.
Method-id.'Method1'.
Factory.
Data Division.
Working-Storage Section.
Procedure Division.
Call procedure-oriented …



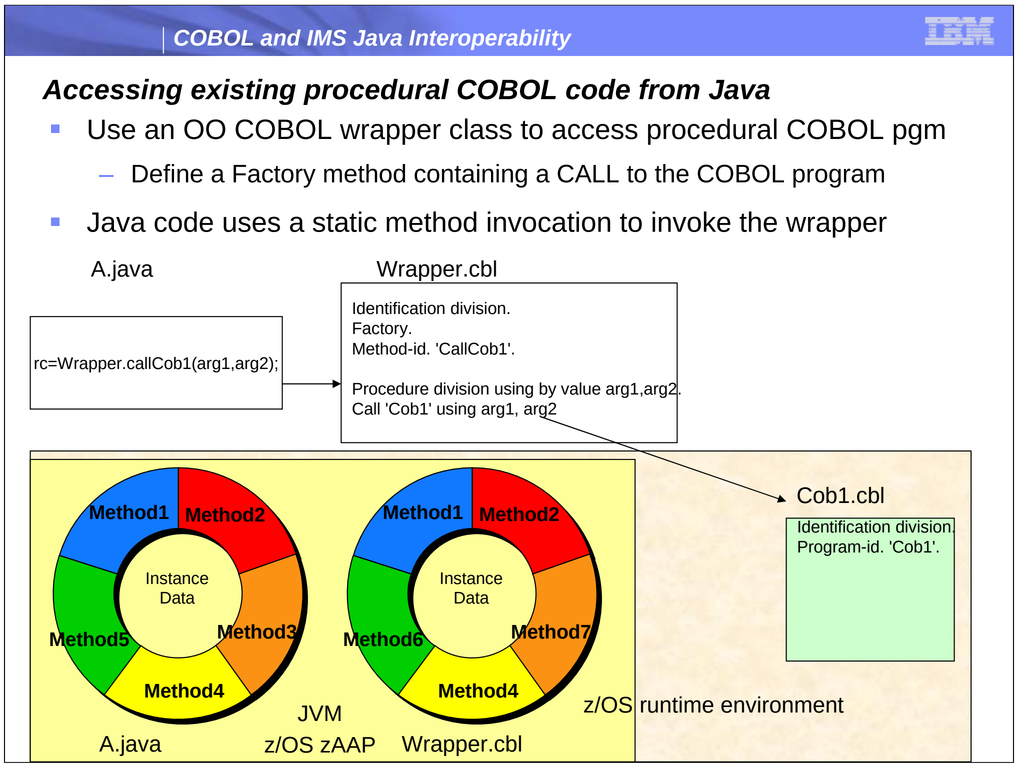
A *wrapper* is a class that provides an interface between object-oriented code and procedure-oriented code

A *wrapper* is a class that provides an interface between object-oriented code and procedure

Define data to be shared by all instances of the class, and methods supported independent

Use the FACTORY paragraph in a class definition to define data and methods that are to b
the class and is shared by all object instances of the class. You most commonly use factory
class that are created. COBOL *factory methods* are equivalent to Java public static method
VALUE clauses alone to initialize instance data

A factory method defines an operation that is supported by a class independently of any ob



Accessing existing procedural COBOL code from Java

- Use an OO COBOL wrapper class to access procedural COBOL pgm
  - Define a Factory method containing a CALL to the COBOL program
- Java code uses a static method invocation to invoke the wrapper

A.java

```
rc=Wrapper.callCob1(arg1,arg2);
```

Wrapper.cbl

```
Identification division.
Factory.
Method-id. 'CallCob1'.

Procedure division using by value arg1,arg2.
Call 'Cob1' using arg1, arg2
```

Cob1.cbl

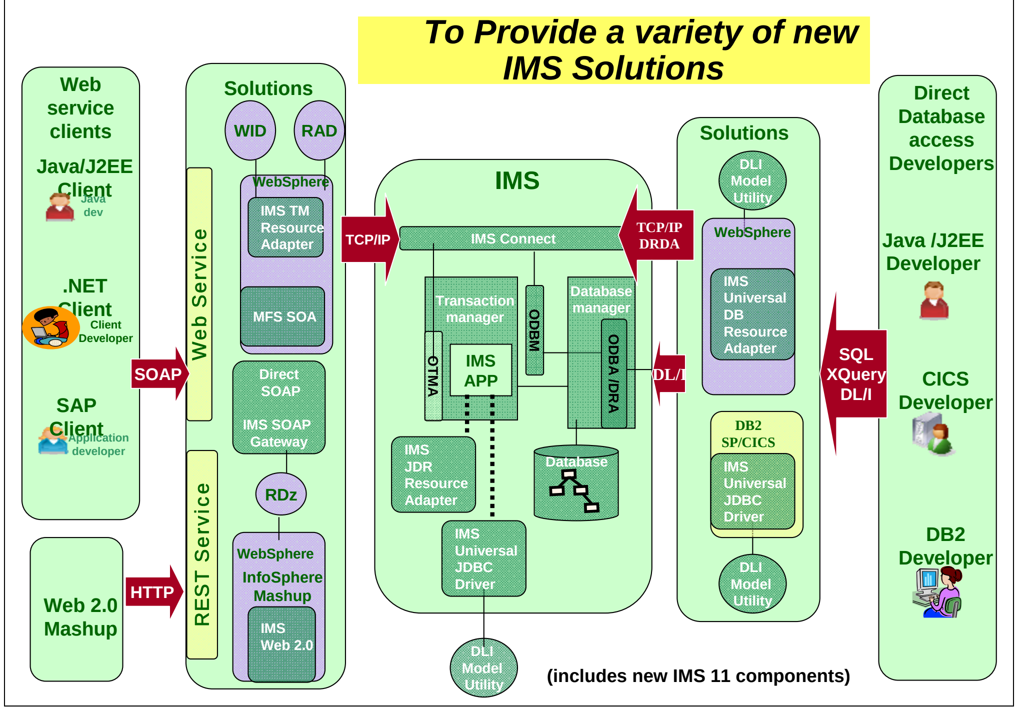
```
Identification division.
Program-id. 'Cob1'.
```

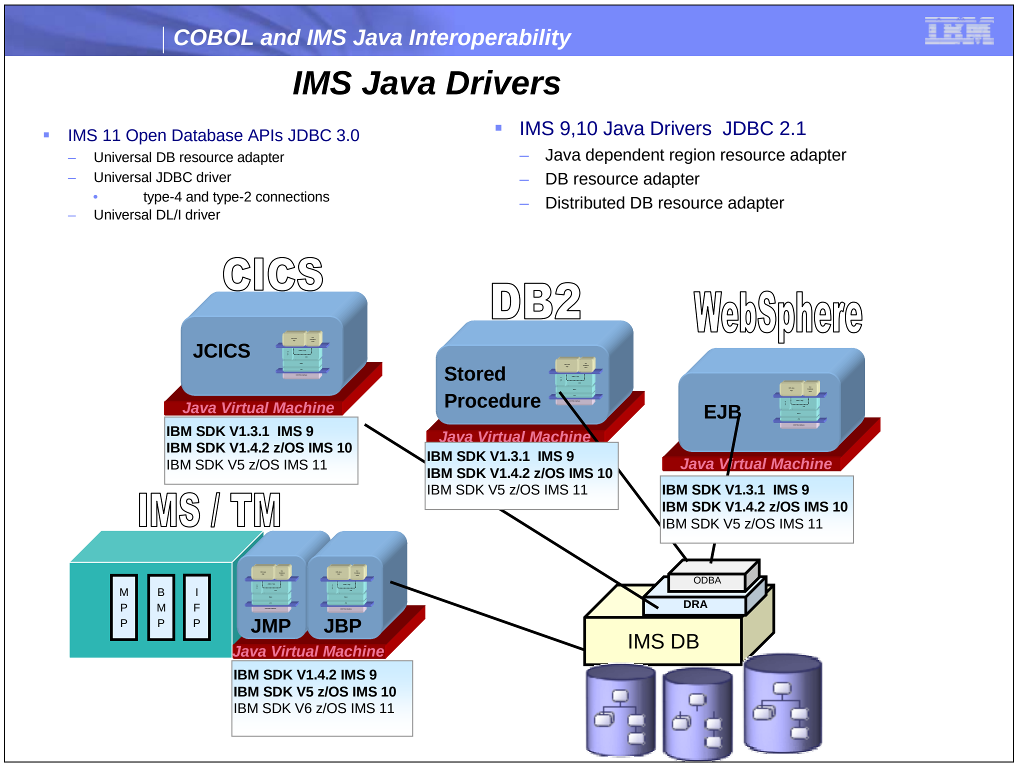
We talked mainly about how object oriented cobol can be used to perform interoperability between java and COBOL but I mentioned earlier that we can do this with all of the currently existing COBOL most of which is procedural.

We don’t modify the procedural code and convert it to Object-Oriented Code. This would be a nightmare. Instead we write a simple wrapper around it In this example the wrapper only handles one procedural cobol application but the same wrapper could actually cover multiple procedural programs by defining different methods for each.

COBOL and IMS Java Interoperability

To Provide a variety of new IMS Solutions





IMS Java Drivers

- IMS 11 Open Database APIs JDBC 3.0
  - Universal DB resource adapter
  - Universal JDBC driver
    - type-4 and type-2 connections
  - Universal DL/I driver
- IMS 9,10 Java Drivers JDBC 2.1
  - Java dependent region resource adapter
  - DB resource adapter
  - Distributed DB resource adapter

The IMS JDBC Driver enables JDBC access to IMS DB from IMS TM JMP/JBP environments, CICS Java application, DB2 Java Stored procedure, and Enterprise Java Beans running on WebSphere distributed and z/OS environments.

IMS V9 requires SDK V1.4.2 for JMP and JBP regions, IMS DB Resource Adapter for CICS, DB2 or WAS requires SDK V1.3.1 or higher.

IMS V10 requires SDK V5 for JMP and JBP regions, IMS DB Resource Adapter for CICS, DB2 or WAS requires SDK V1.4.2 or higher.

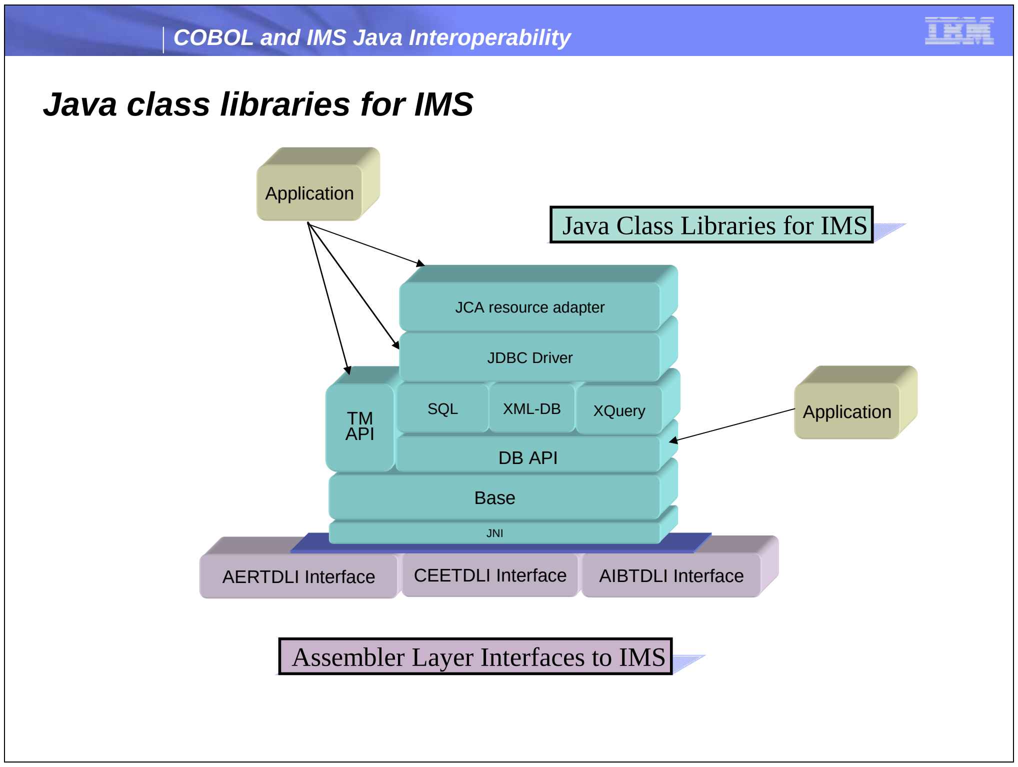
The IMS Universal drivers have the following runtime software requirements:

Java Development Kit (JDK) 5.0 or later for CICS Transaction Server for z/OS Version 3.1

for DB2 for z/OS Version 9.1 or DB2 UDB for z/OS Version 8

for WebSphere Application Server for z/OS or WebSphere Application Server for distributed platforms, Version 6.1

for JMP and JBP regions require Java Development Kit JDK 6.0 or later


COBOL and IMS Java Interoperability

Java class libraries for IMS

Bottom – c i/face (depending on the environment 3 interfaces) – have to drop down to c (thin jni layer)

Base – 1-1 mapping of the way ims works under covers / in java build SSAs & make db calls using dli

Db – really what is turning this in to our jdbc driver/ making sql calls

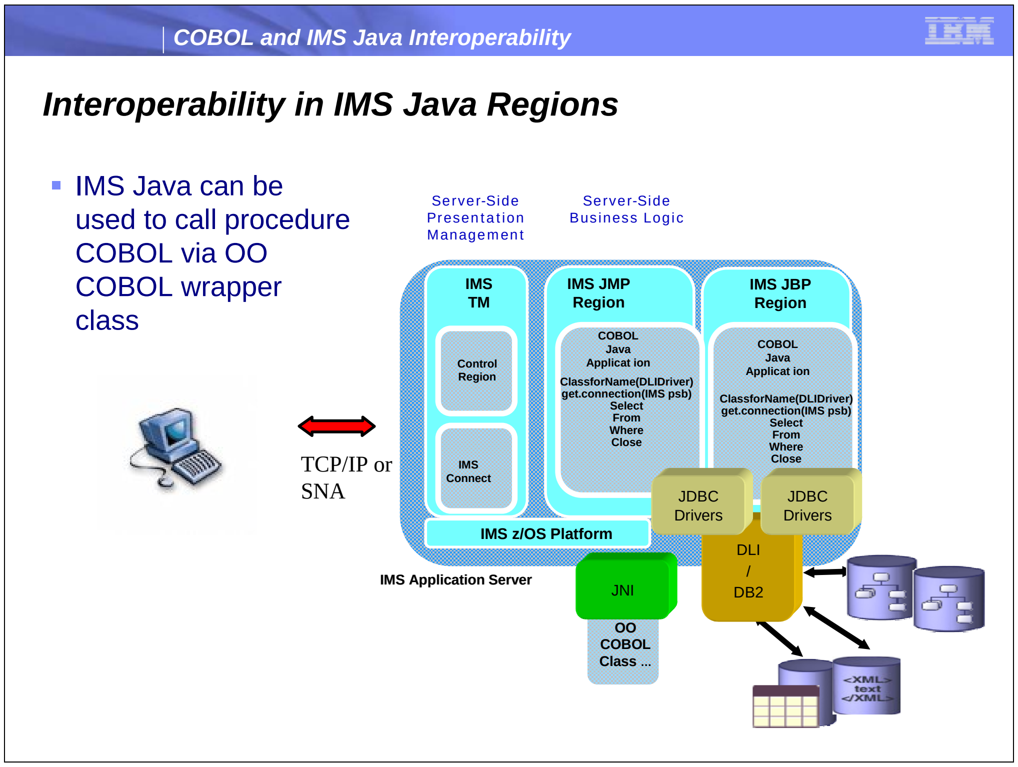
App - running in an ims dependent region and offers reading/writing messages to ims message queue

Customer code – not worry about the dli / only jdbc

Dbview- ims does not have online metadata; good – now we added new stuff (XML

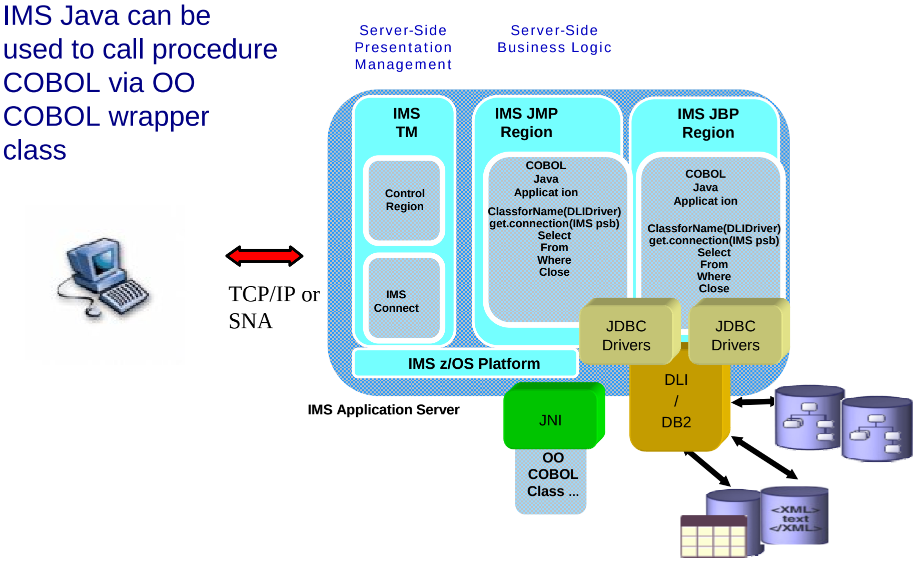
bad –

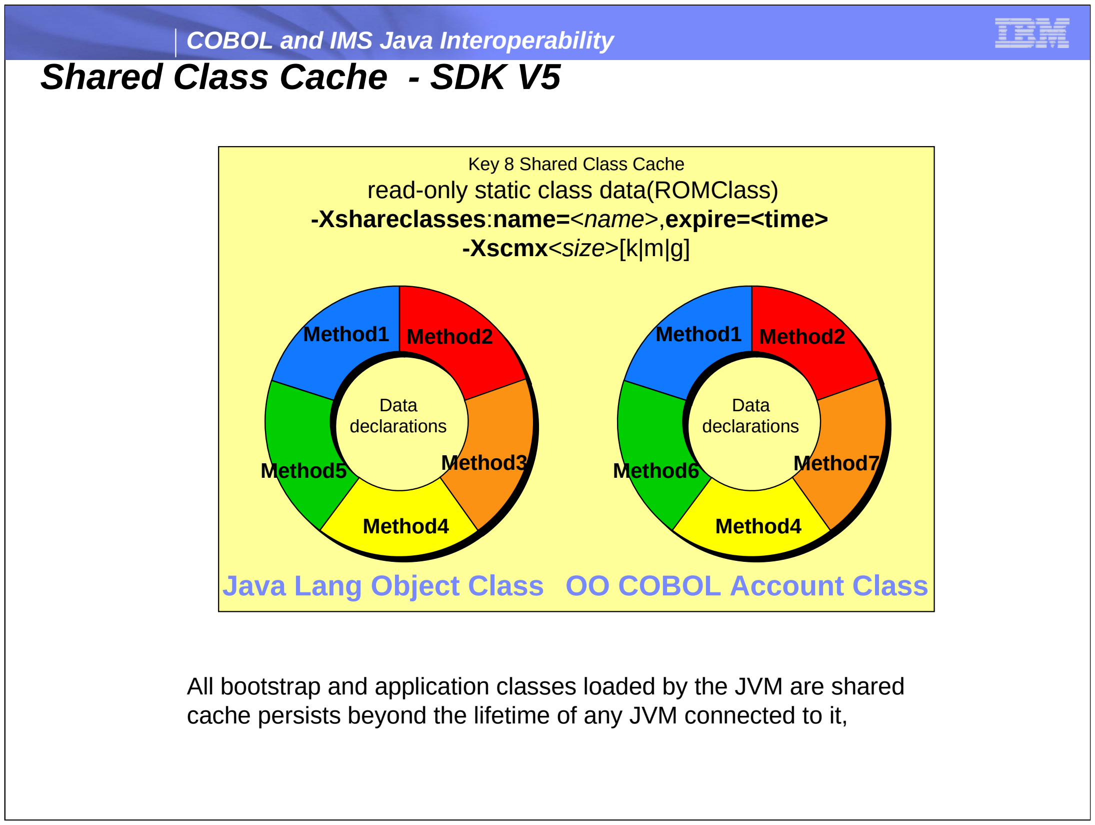
Tooling – generates database view called dli model utility



Interoperability in IMS Java Regions

- IMS Java can be used to call procedure COBOL via OO COBOL wrapper class





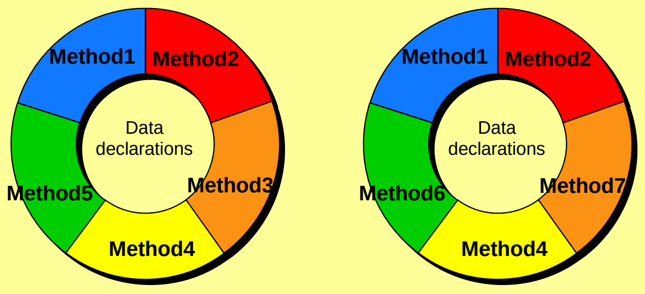
Shared Class Cache - SDK V5

Key 8 Shared Class Cache

read-only static class data(ROMClass)

-Xshareclasses:**name=**<*name*>,**expire=<time>**

-Xscmx<*size*>[k|m|g]



Java Lang Object Class **OO COBOL Account Class**

All bootstrap and application classes loaded by the JVM are shared cache persists beyond the lifetime of any JVM connected to it,

The shared class cache contains read-only static class data and metadata that describes the classes.

To enable class sharing use –Xshareclasses:-name= option when starting a JVM.

The -name**=**<*name*> connects a JVM to the specified name cache or creates the cache if it does not already exist.

expire=<time> suboption is useful for automatically clearing out old caches that have not been used for a period of time. The suboption is added to the **-Xshareclasses** option and takes a parameter **<time>** in minutes. This option causes the JVM to scan automatically for caches that have not been connected to for a period greater than or equal to **<time>** before initializing the shared classes support.

-Xscmx<size>[k|m|g] Specifies cache size. This option applies only if a cache is being created and no cache of the same name exists.

Note shared class is not required.

The cache persists beyond the lifetime of any JVM connected to it, until it is explicitly destroyed or until the operating system is shut down. A cache can be destroyed only if all JVMs using it have shut down

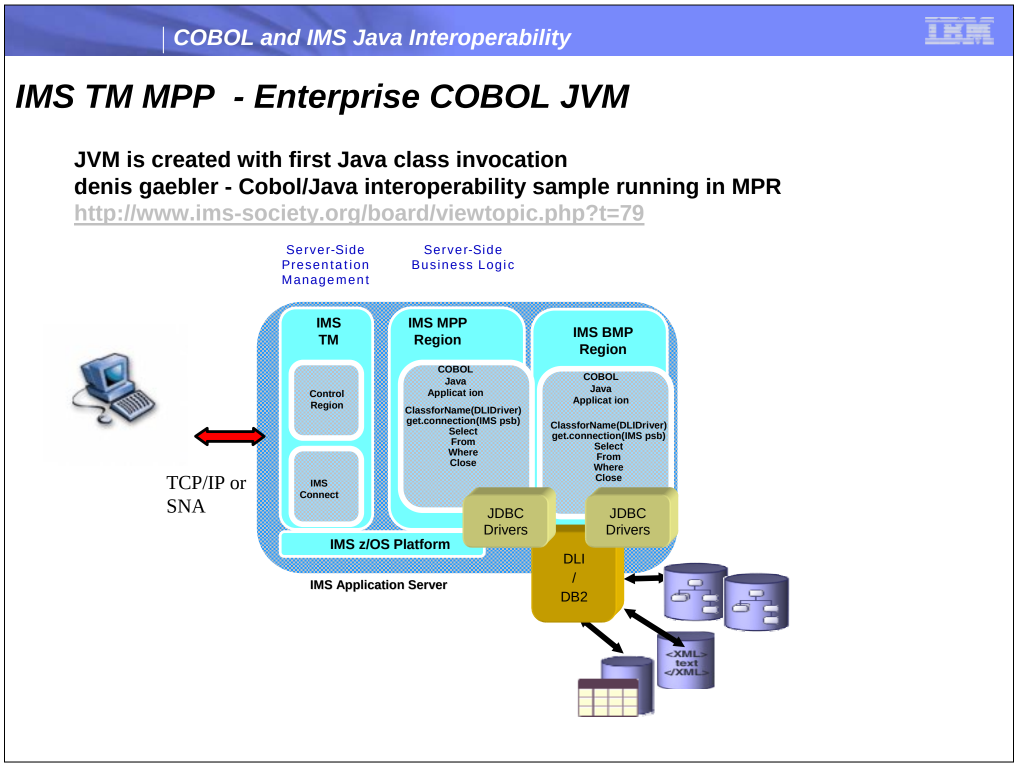
Run

java -Xshareclasses: name=<*name*> **destroy**

java -Xshareclasses:destroyAll

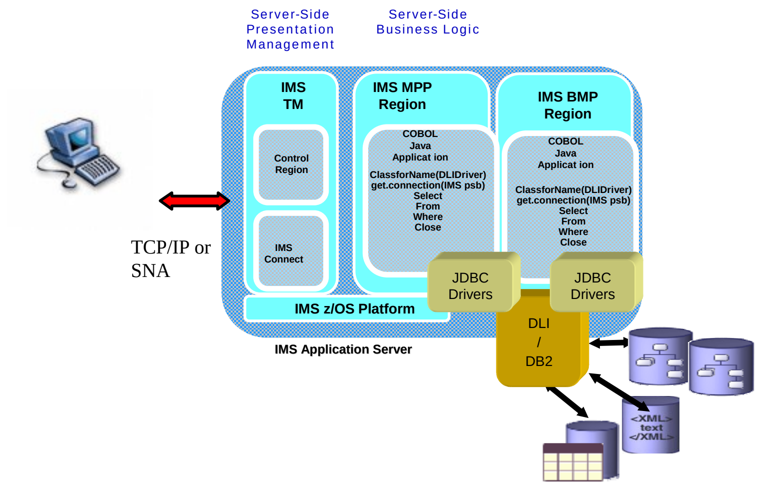
an **immutable object** is an object whose state cannot be modified after it is created. This is in contrast to a **mutable object**, which can be modified after it is created.

Some of the **BPXPRMxx** parmlib settings affect shared classes performance. Using the wrong settings can stop shared classes from working. These settings might also have performance implications. For further information about performance implications and use of these

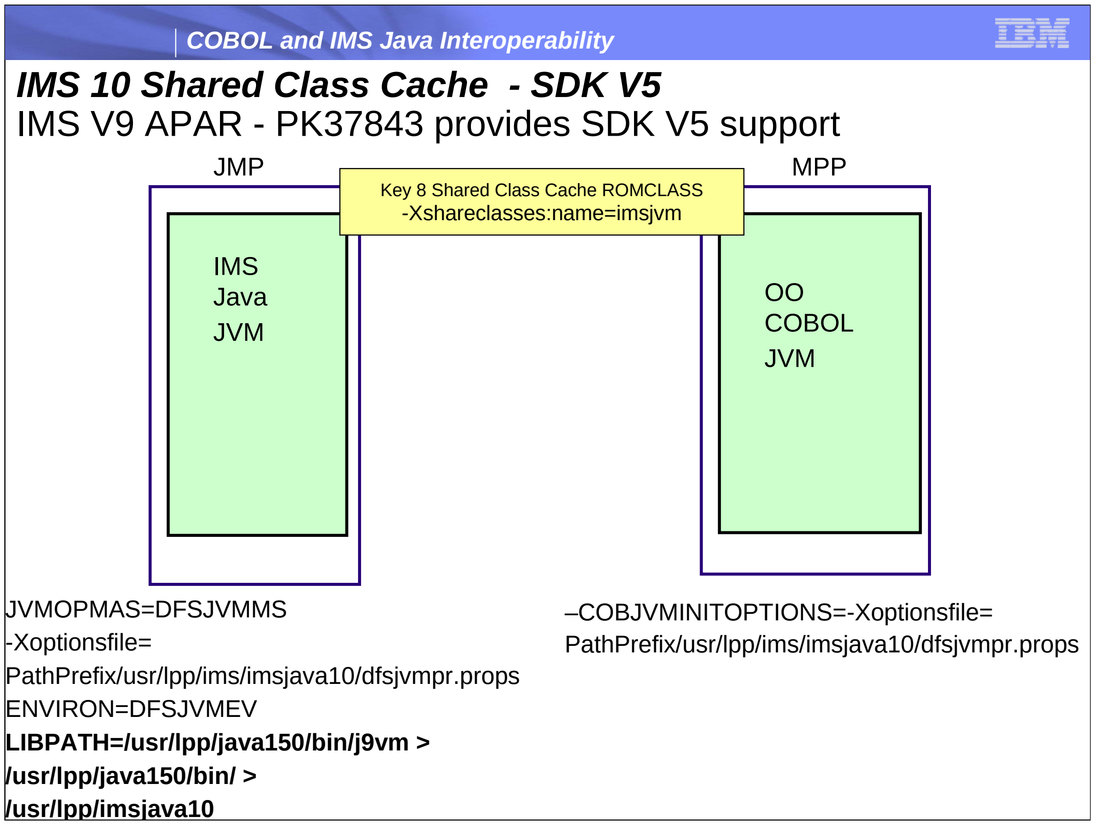


IMS TM MPP - Enterprise COBOL JVM

JVM is created with first Java class invocation
denis gaebler - Cobol/Java interoperability sample running in MPR
http://www.ims-society.org/board/viewtopic.php?t=79



Enterprise COBOL will launch the JVM in the MPP/BMP runtime when a Java method is INVOKED.



IMS 10 Shared Class Cache - SDK V5

IMS V9 APAR - PK37843 provides SDK V5 support

JVMOPMAS=DFSJVMMS
-Xoptionsfile=
PathPrefix/usr/lpp/ims/imsjava10/dfsjvmpr.props
ENVIRON=DFSJVMEV
LIBPATH=/usr/lpp/java150/bin/j9vm >
/usr/lpp/java150/bin/ >
/usr/lpp/imsjava10

–COBJVMINITOPTIONS=-Xoptionsfile=
PathPrefix/usr/lpp/ims/imsjava10/dfsjvmpr.props

JVMOPMAS=DFSJVMMS

Specifies the JVM options

ENVIRON=DFSJVMEV

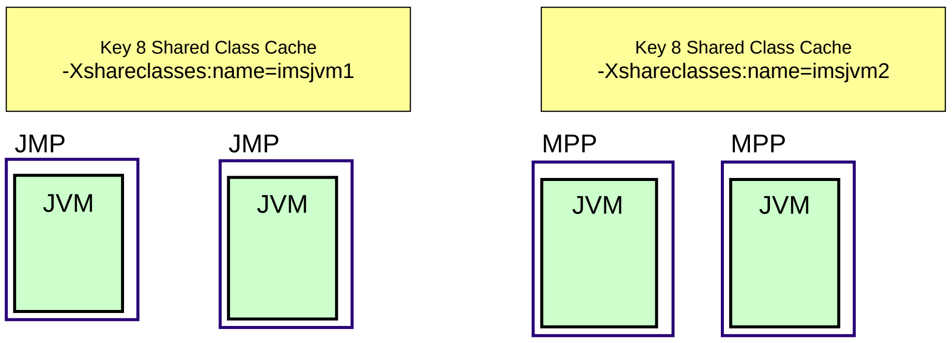
Must contain the pathname to JVM

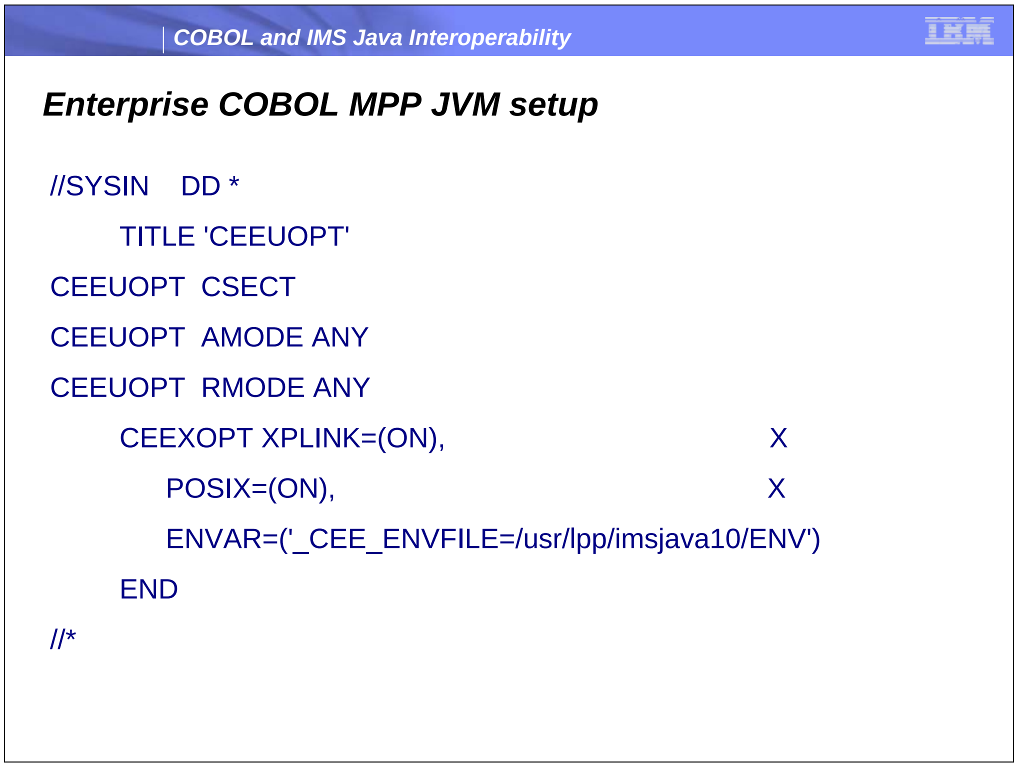
libwrappers.so libjvm.so

Must contain the pathname to the IMS Java native code

libJavTDLI.so

Multiple Shared Class Cache





Enterprise COBOL MPP JVM setup

```
//SYSIN    DD *
         TITLE 'CEEUOPT'
CEEUOPT  CSECT
CEEUOPT  AMODE ANY
CEEUOPT  RMODE ANY
         CEEXOPT XPLINK=(ON),                              X
               POSIX=(ON),                                 X
               ENVAR=('_CEE_ENVFILE=/usr/lpp/imsjava10/ENV')
         END
//*
```

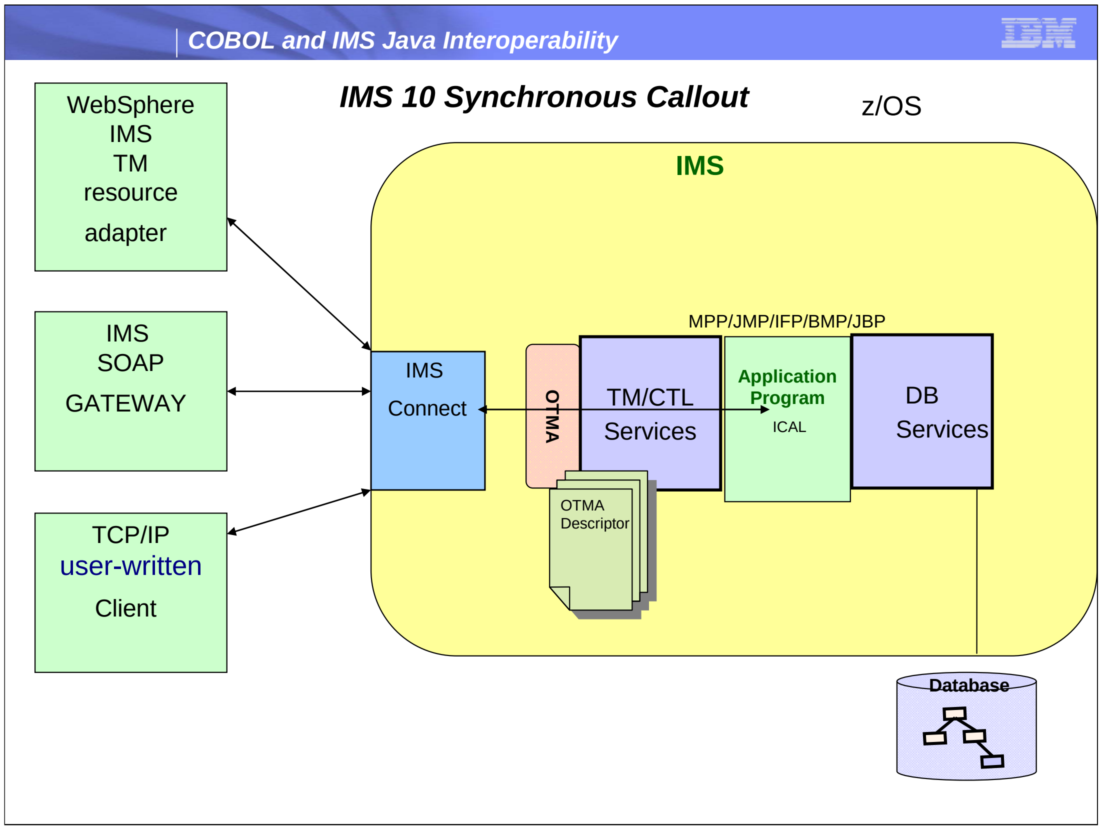
Enterprise COBOL MPP JVM setup

```
JAVA_HOME=/usr/lpp/java/J1.5
PATH=/bin:/usr/lpp/java/J1.5/bin:.
LIBPATH=/lib:/usr/lib:/usr/lpp/java/J1.5/bin/j9vm:/usr/lpp/java/J1.5/bin:/usr/lpp/java
   /J1.5/bin/classic:/u/imsmpp/jcobol
CLASSPATH=/u/imsmpp/jcoboltest.jar:/u/imsmpp/jcobol
```

```
COBJVMINITOPTIONS=-Xoptionsfile=/u/imspp/jcobol/jvm.properties
```

/u/imsmpp/jcobol/jvm.properties e.g. contains:

```
-Xcodecache=32M
-Xquickstart
-Xshareclasses:name=imscobol
-Xscmx32M
-Xmx256M
-Xms128M
```

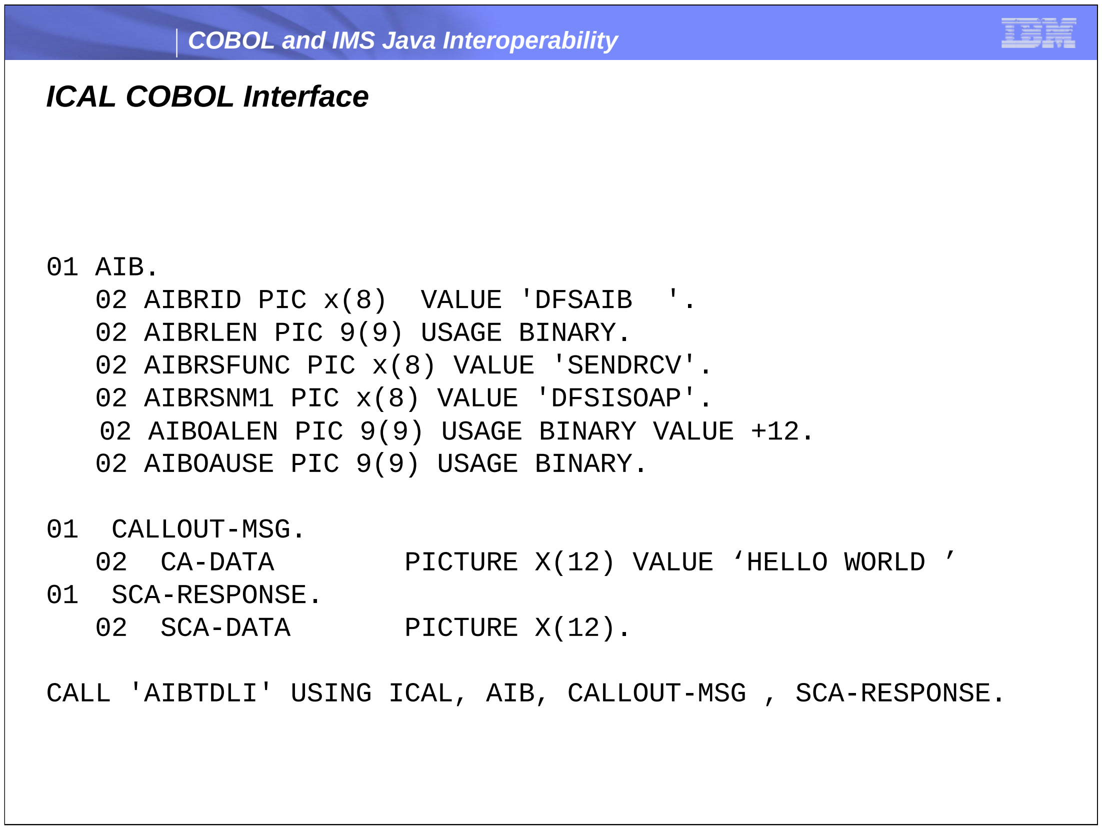


This diagram shows that with the IMS callout support IMS applications can be a client and server. IMS provides bi-directional access between IMS applications and external application and servers. The IMS Application Program can callout to:

user-written IMS Connect client

WebSphere EJB/MDB using IMS TM Resource Adapter

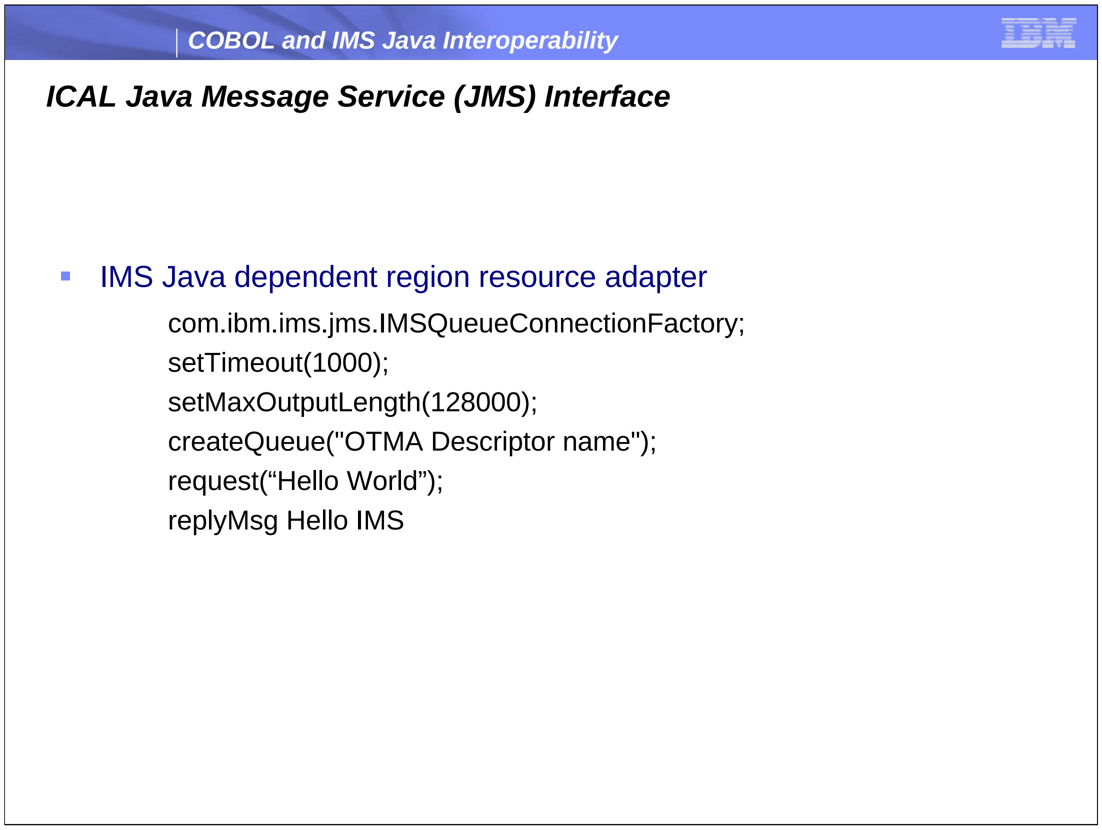
Web Service Provider using IMS SOAP Gateway



ICAL COBOL Interface

```
01 AIB.
   02 AIBRID PIC x(8)  VALUE 'DFSAIB  '.
   02 AIBRLEN PIC 9(9) USAGE BINARY.
   02 AIBRSFUNC PIC x(8) VALUE 'SENDRCV'.
   02 AIBRSNM1 PIC x(8) VALUE 'DFSISOAP'.
   02 AIBOALEN PIC 9(9) USAGE BINARY VALUE +12.
   02 AIBOAUSE PIC 9(9) USAGE BINARY.

01  CALLOUT-MSG.
   02  CA-DATA        PICTURE X(12) VALUE ‘HELLO WORLD ’
01  SCA-RESPONSE.
   02  SCA-DATA       PICTURE X(12).

CALL 'AIBTDLI' USING ICAL, AIB, CALLOUT-MSG , SCA-RESPONSE.
```

COBOL sample



ICAL Java Message Service (JMS) Interface

- IMS Java dependent region resource adapter

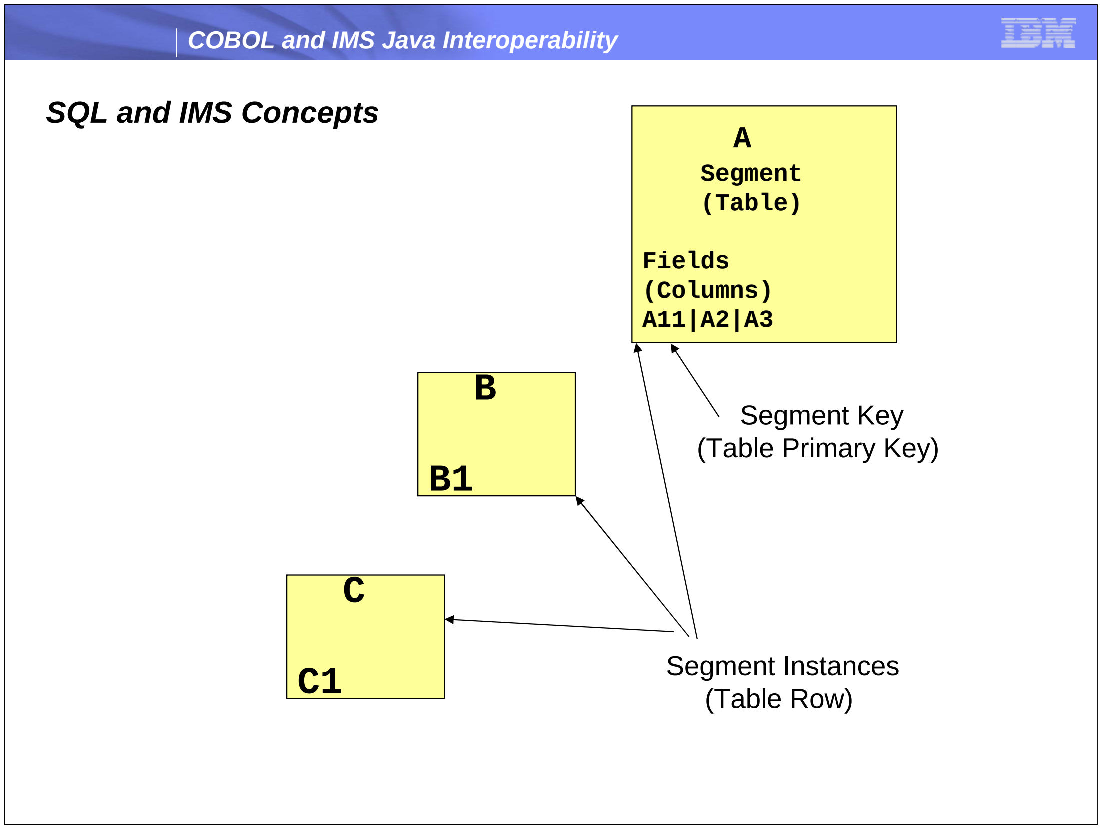
```
com.ibm.ims.jms.IMSQueueConnectionFactory;
setTimeout(1000);
setMaxOutputLength(128000);
createQueue("OTMA Descriptor name");
request(“Hello World”);
replyMsg Hello IMS
```

The Java Message Service (JMS) API is a messaging standard for sending messages between two or more application programs.

IMS supports the point to point model for ICAL.

JDBC Review

- JDBC is an application programming interface (API) that Java applications use to access relational databases or tabular data sources.
- JDBC API provides database-independent connectivity for any database that has implemented the JDBC interface
- Executing JDBC query statements
  - Establish and open connection to database
  - Execute query and obtain results
  - Process results
  - Close connection

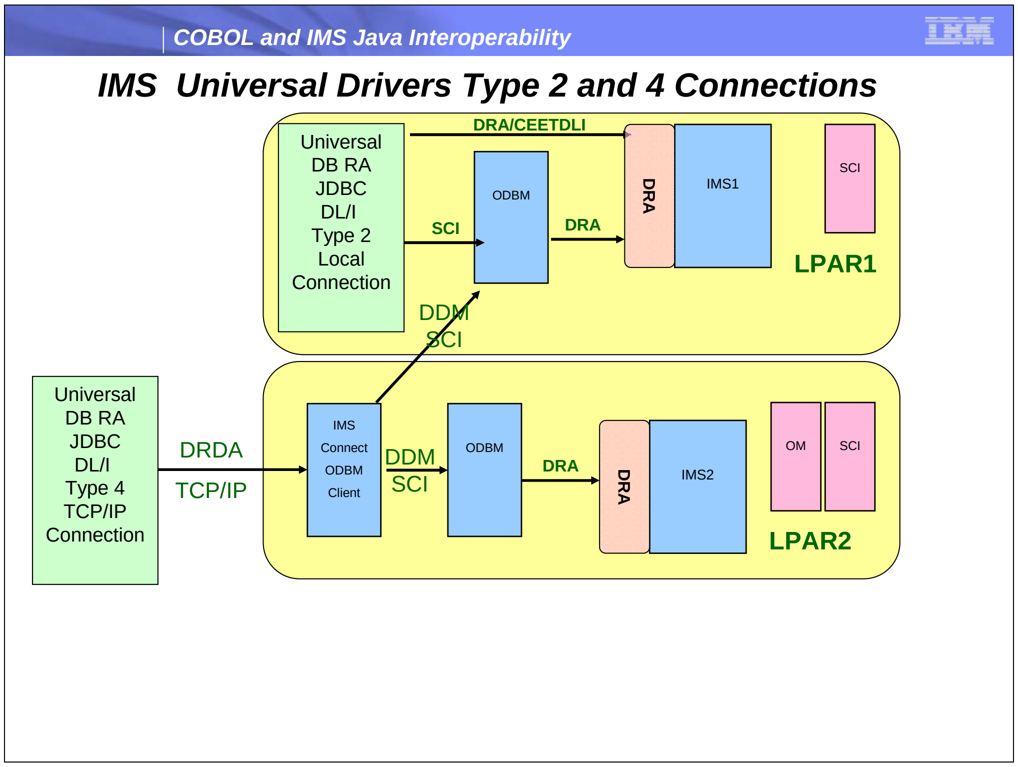


COBOL and IMS Java Interoperability

SQL and IMS Concepts

This slide provides a mapping of IMS hierarchical database concepts and relational database concepts

IMS Java views Segment as a Table, A field as column and segment instance as a row



IMS Universal Drivers Type 2 and 4 Connections

This slide provides an overview of the capabilities provided by Open Database API Universal Drivers.

It shows the Open Database API supporting Type 2 connections for Local(same LPAR as IMS) access to IMS.

When used in IMS JMP/JBP regions the CEETDLI interface is used. For WAS z/OS , DB2 SP and CICS the DRA interface is used.

The TCP/IP Type 4 connection for distributed access to IMS. Note distributed can be across a network or across an LPAR boundary.

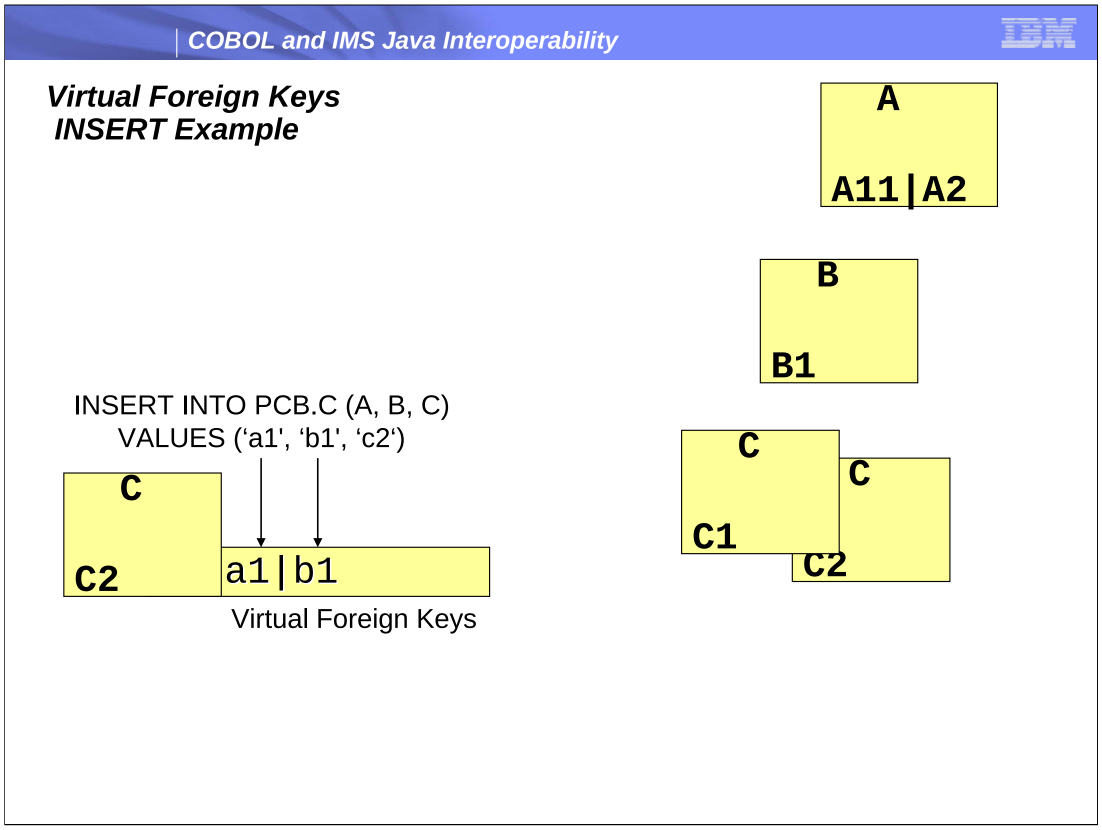
DRDA is the protocol used to communicate to IMS Connect.

DRDA is the industry standard for DB access in a distributed transaction processing environment

The Two Phase Commit and Security flows are imbedded in DRDA

IMS 11 Open Database API JDBC enhancements

- Virtual Foreign Key fields
  - IMS Java maintains the unique keys for segments up to the root
  - SQL SELECT, INSERT, UPDATE, and DELETE queries
    - SQL syntax for IMS appears similar to standard SQL
- Updatable Result Sets
  - Update or delete of current row
- Metadata Discovery
  - Access the IMS Java Metadata classes generated by the DLIModel utility
- autoCommit support
  - updates are committed as they happen
- setFetchSize
  - An application can set the expected or desired number of rows to be returned


COBOL and IMS Java Interoperability

Virtual Foreign Keys INSERT Example

INSERT INTO PCB.C (A, B, C) VALUES (‘a1', ‘b1', ‘c2‘)

In relational databases, hierarchies can be logically built by creating foreign key relationships between tables.

In IMS, the hierarchies are explicit and are part of the database definition itself.

The IMS Open Database API introduces the concept of virtual foreign keys to capture these explicit hierarchies in a relational sense, which makes the SQL syntax for IMS appear similar to standard SQL.

When accessing IMS databases every segment that is not the root segment in a hierarchic path will virtually contain the unique keys of all of its parent segments up to the root of the database. These keys are called virtual foreign key fields.

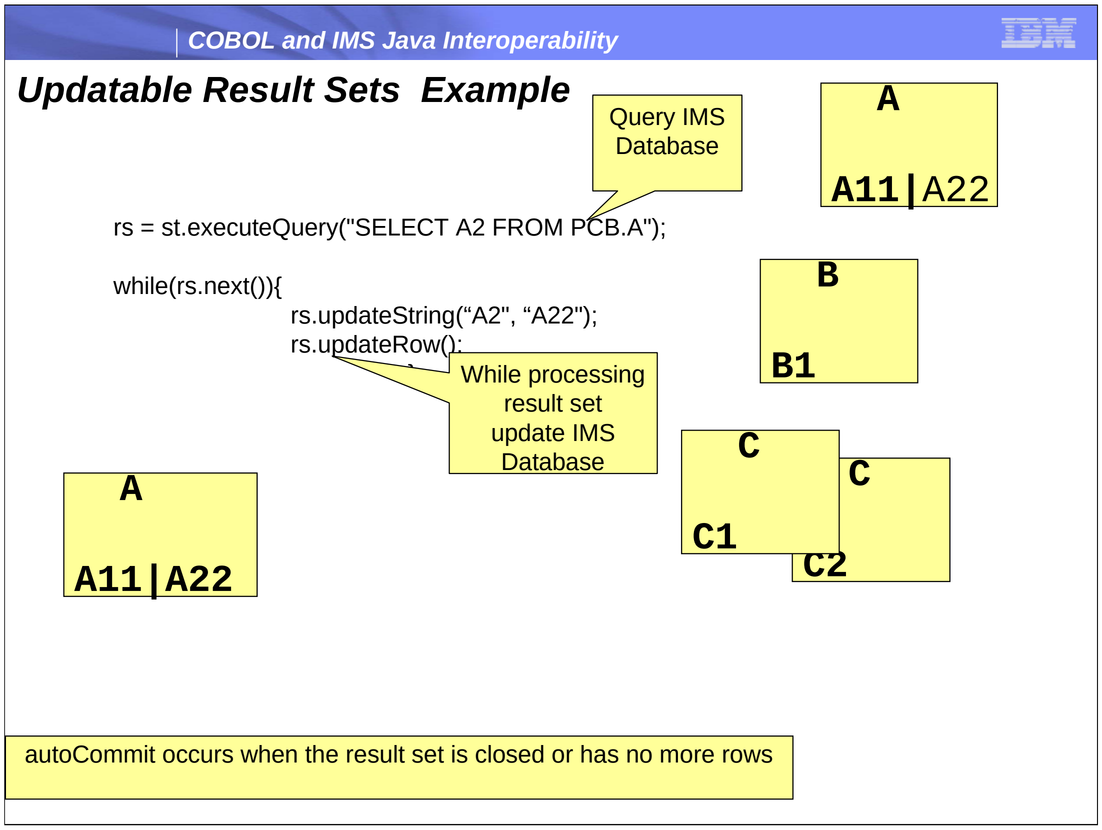
The purpose of the virtual foreign key fields is to maintain referential integrity, similar to foreign keys in relational databases.

This allows SQL SELECT, INSERT, UPDATE, and DELETE queries to be written against specific tables and columns located in the IMS database hierarchic path.

Remember: Virtual foreign keys are maintained internally and are not physically stored in the IMS database.

Virtual Foreign Keys are the same concept as IMS fully concatenated keys.

Virtual foreign keys are maintained internally by the IMS Universal driver; the keys are not physically stored in the IMS database



COBOL and IMS Java Interoperability

Updatable Result Sets Example

```
rs = st.executeQuery("SELECT A2 FROM PCB.A");

while(rs.next()){
                    rs.updateString(“A2", “A22");
                    rs.updateRow();
           }
```

autoCommit occurs when the result set is closed or has no more rows

A ResultSet object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

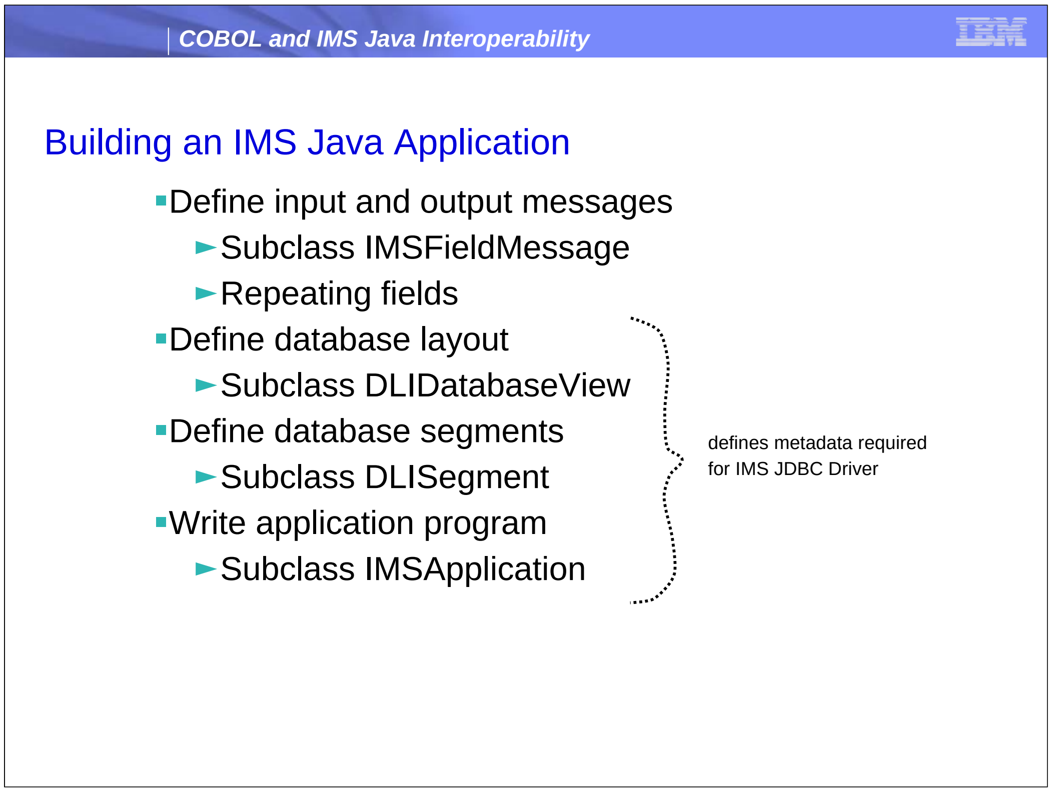
In this example the A2 field in segment A is updated to A22
autocommit occurs when the result set is closed or has no more rows

SQL keywords support

- XML Support
  - Retrieval, Storage

```
SELECT firstName, lastName, retrieveXML(Employees)
FROM DealerTable.Employees
WHERE serialNumber = ‘3A0140’
```

Build an XML document out of the Employee Segment and all dependant Segments in this PCB for the employee with serial number 3A0140.


COBOL and IMS Java Interoperability

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 IMS JDBC Driver

Define metadata - helps to ensure program correctness - allows our product to be "smart" about memory layouts and data conversions
Information about the input and output messages

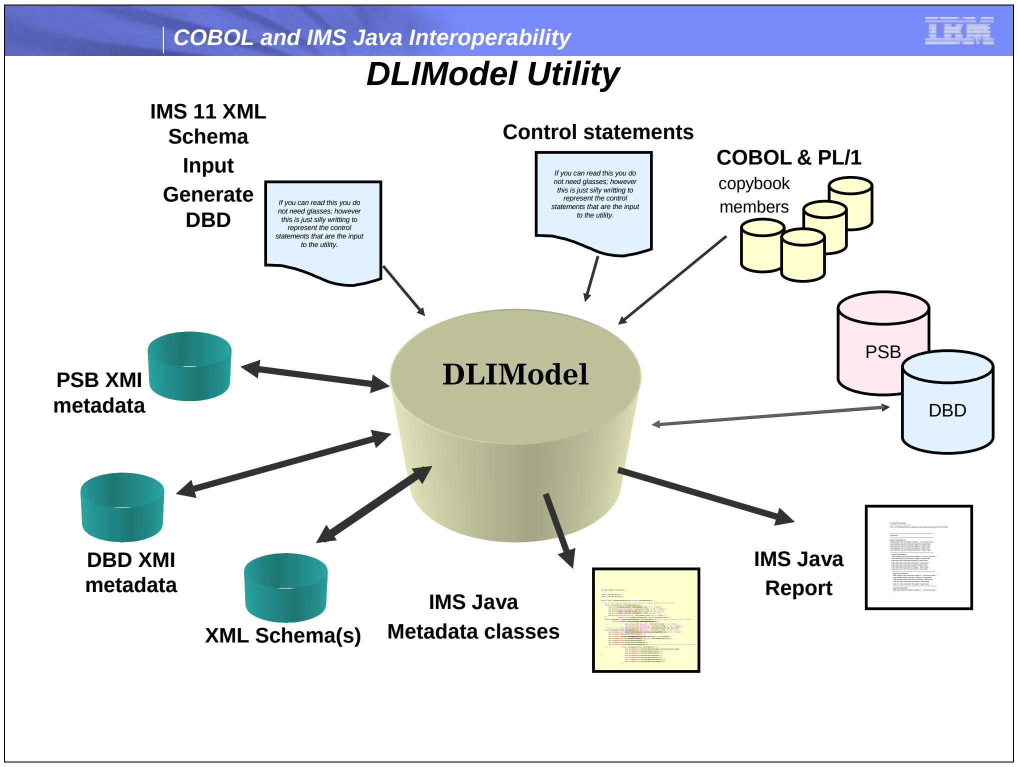
Information about the segments and fields in the database

Type, length

Used for type safety (can't put a short in a String field without converting it first to a String)

Used for data conversion

We need the hierarchy as well (to build SSAs and ensure program correctness

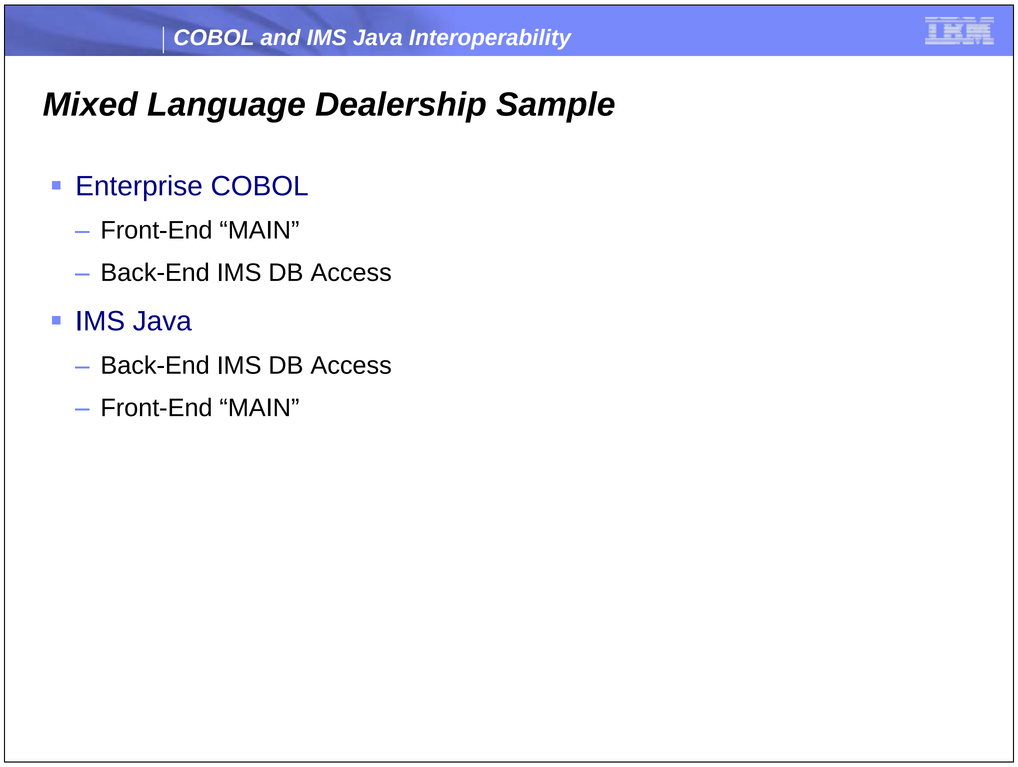

COBOL and IMS Java Interoperability

DLIModel Utility

IMS 11 XML Schema Input Generate DBD
If you can read this you do not need glasses; however this is just silly writting to represent the control statements that are the input to the utility.
Control statements
If you can read this you do not need glasses; however this is just silly writting to represent the control statements that are the input to the utility.
COBOL & PL/1
copybook members
PSB
DBD
DLIModel
PSB XMI metadata
DBD XMI metadata
XML Schema(s)
IMS Java Metadata classes
IMS Java Report

Text report will be removed since application programmers can use the graphical view of PSB/DBD to assist them.

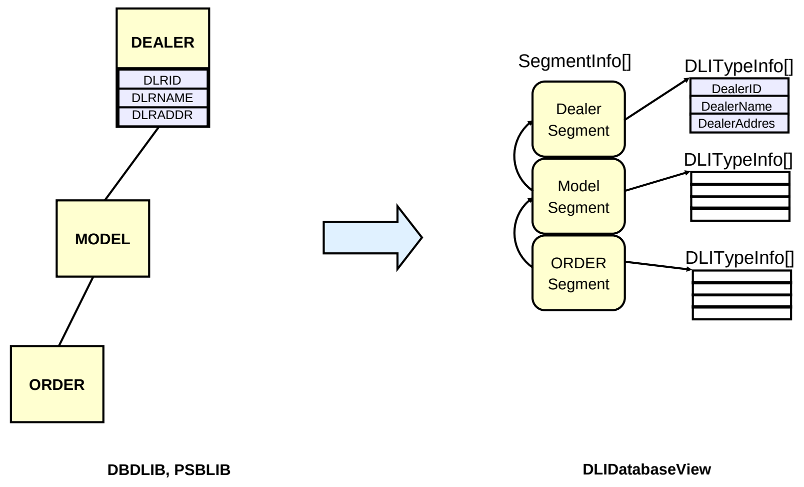
The graphical view of PSB and DBD can now be “Save As…” graphic files (JPG or BMP)

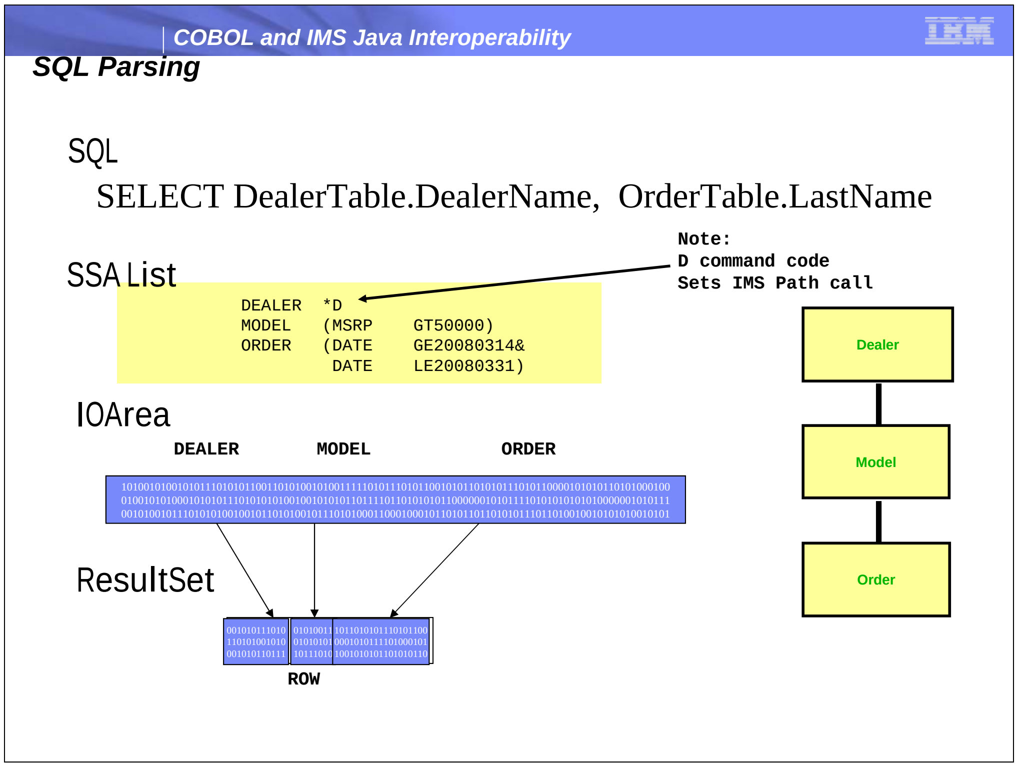


Mixed Language Dealership Sample

- Enterprise COBOL
  - Front-End “MAIN”
  - Back-End IMS DB Access
- IMS Java
  - Back-End IMS DB Access
  - Front-End “MAIN”

IMS Metadata





COBOL and IMS Java Interoperability

SQL Parsing

SQL

SELECT DealerTable.DealerName, OrderTable.LastName

SSA List

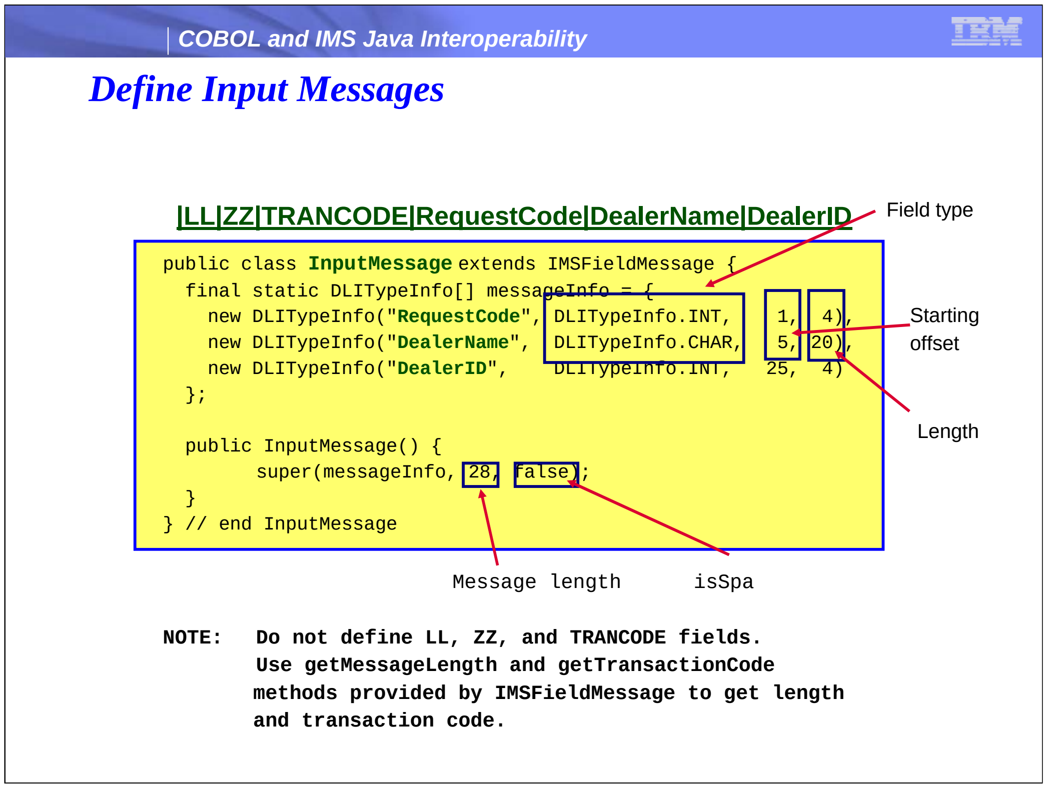
```
DEALER   *D
MODEL    (MSRP     GT50000)
ORDER    (DATE     GE20080314&
          DATE     LE20080331)
```

```
Note:
D command code
Sets IMS Path call
```

Create io area based on size; copy the fields in to resultset and return it as jdbc expects

Command code D Retrieve or insert a sequence of segments in a hierarchic path using only one call, instead of having to use a separate (path) call for each segment.

This provides an implicit SQL Table JOIN for the IMS segments in the path



Define Input Messages

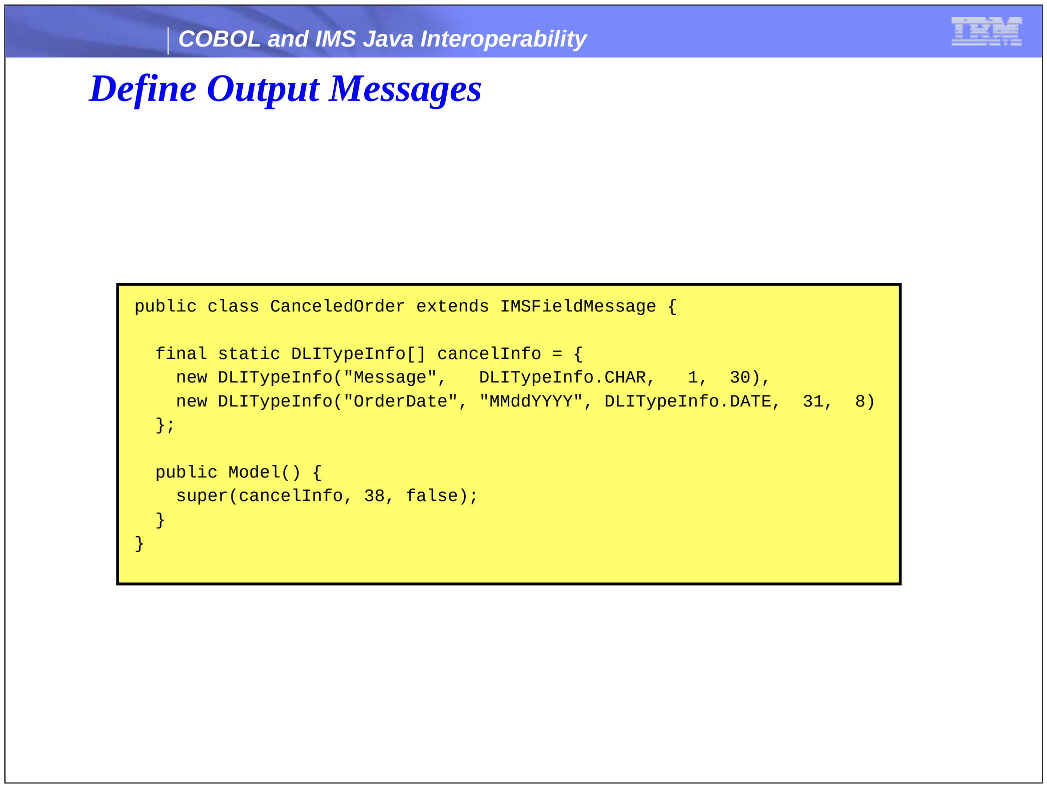
|LL|ZZ|TRANCODE|RequestCode|DealerName|DealerID

```
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.
```

Layout of the message we are going to read from the queue
Field name, type ,starting offset, length
We have taken out LL, ZZ, and TRANCODE from the user space

In the call to super() - pass the array of type info fields, the total length, and false because it is not a SPA message



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);
  }
}
```

Much like input messages

Output messages are those that are going to be written back to the queue

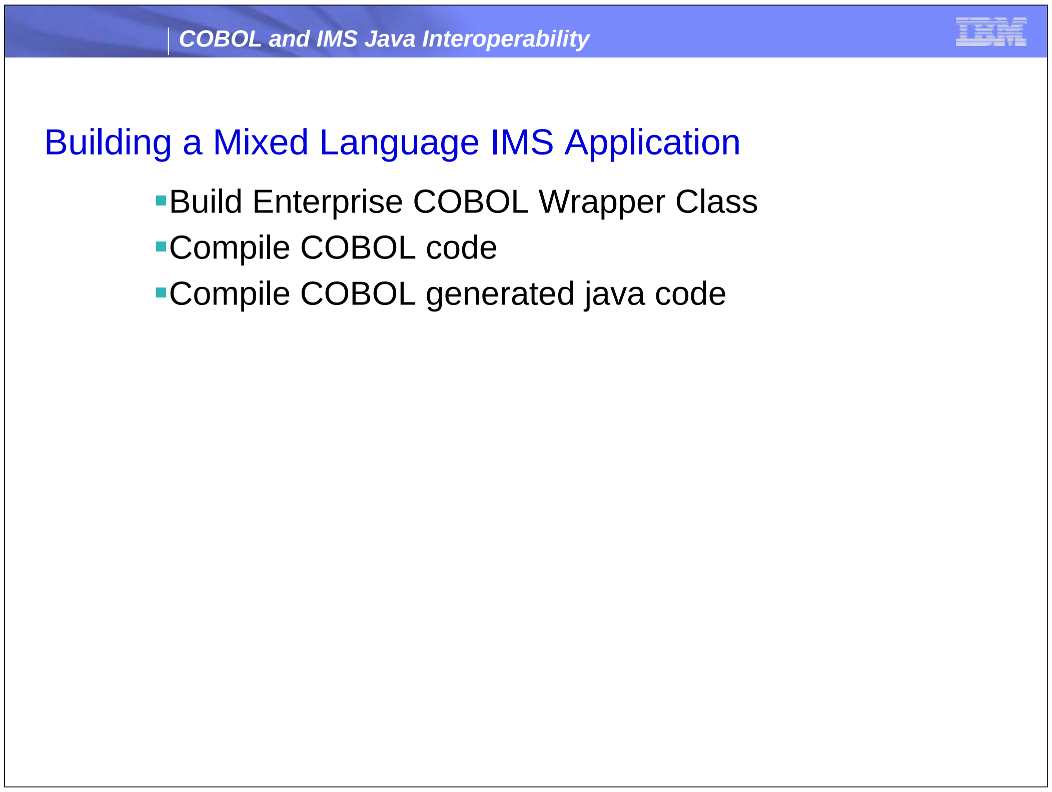
Notice the DATE field - have a typeQualifier giving layout of date in memory
more on that later

IMS Dealer Application Java Front-End

```
package samples.dealership;

public class IMSAuto {

    public static void main(String args []) {
        IMSAuto imsauto = new IMSAuto();

        IMSMessageQueue messageQueue = new IMSMessageQueue();
        FindCarInput  inputMessage  = new FindCarInput();
        FindCarOutput outputMessage = new FindCarOutput();

        try {
            while (messageQueue.getUniqueMessage(inputMessage)) {
                imsauto.proccessMessage(inputMessage, outputMessage);
                messageQueue.insertMessage(outputMessage.format());
         CobolAutoDealership dealer = new CobolAutoDealership();
         dealer.displayModel(input,output);

            }
        } catch (IMSException e) {
            e.printStackTrace();
        }
    }
}
```

IMS Dealer Application COBOL Back-END

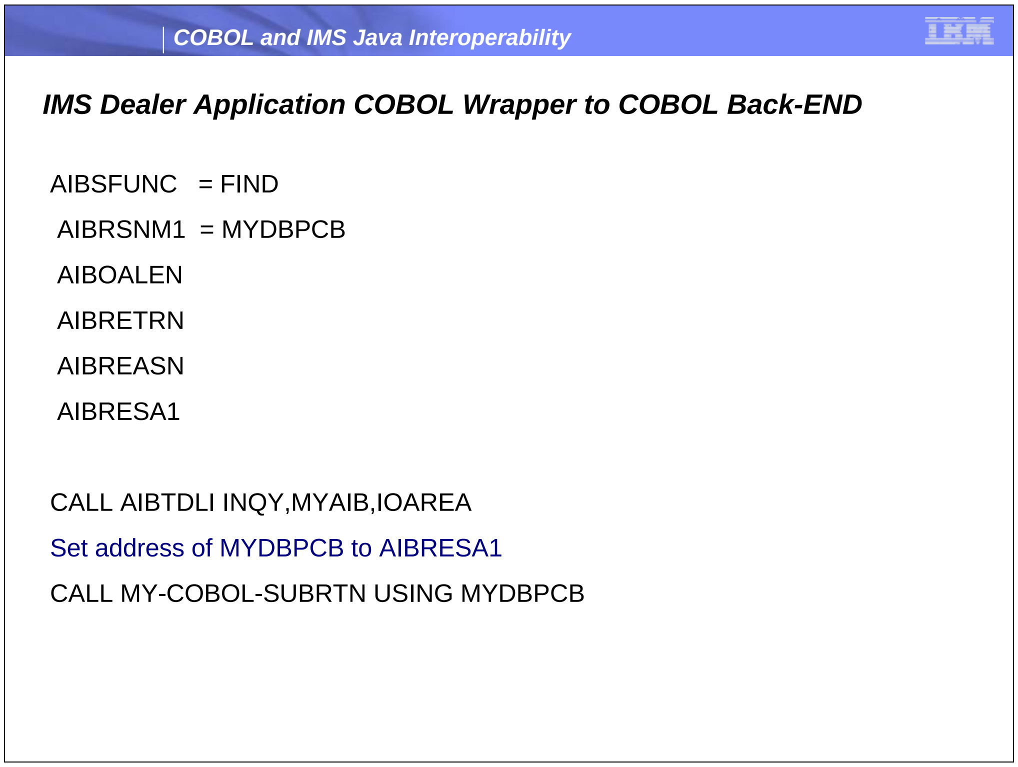
```
Move "AUTOLPCB" to AIBRSNM1
         Move length of MODEL-SEGMENT to AIBOALEN

         Call "CEETDLI" using GN, AIB, MODEL-SEGMENT, model-ssa
         Set address of DBPCB to AIBRESA1

         If dbstat is = spaces
```



Building a Mixed Language IMS Application

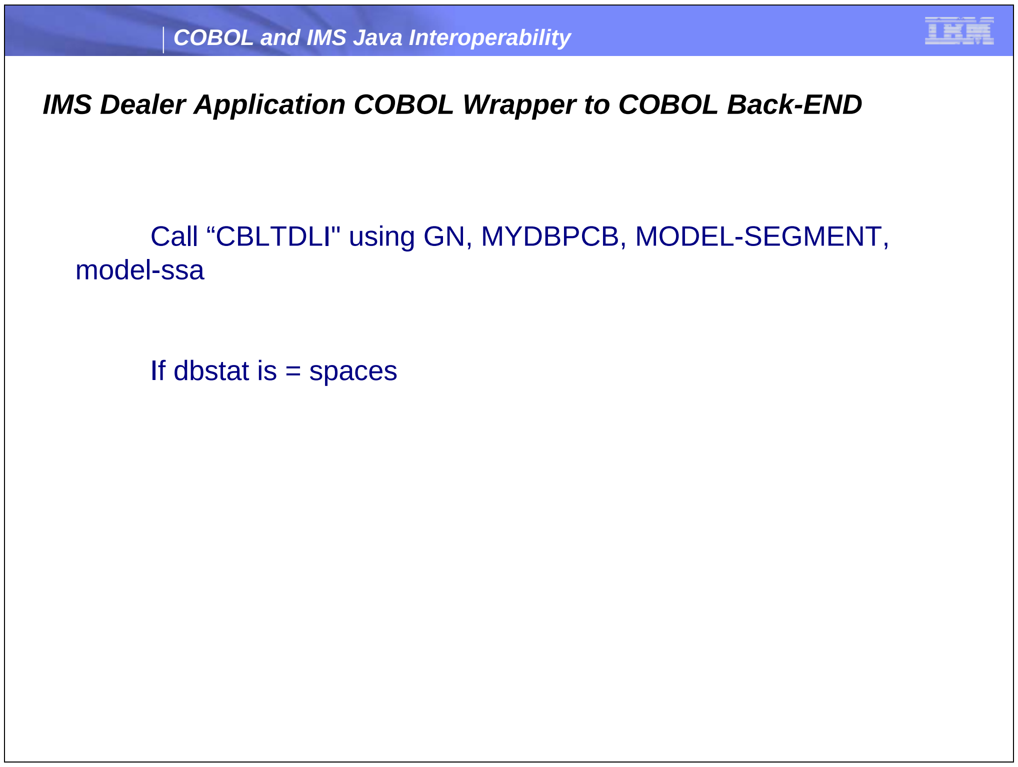
- Build Enterprise COBOL Wrapper Class
- Compile COBOL code
- Compile COBOL generated java code

you can obtain PCB addresses by making an IMS INQY call using the FIND subfunction and the PCB name as the resource name. The INQY call returns the address of the PCB, which you can then pass to a COBOL program. This approach still requires that the PCB name be defined as part of the PSBGEN, but the COBOL application does not have to use the AIB interface.



IMS Dealer Application COBOL Wrapper to COBOL Back-END

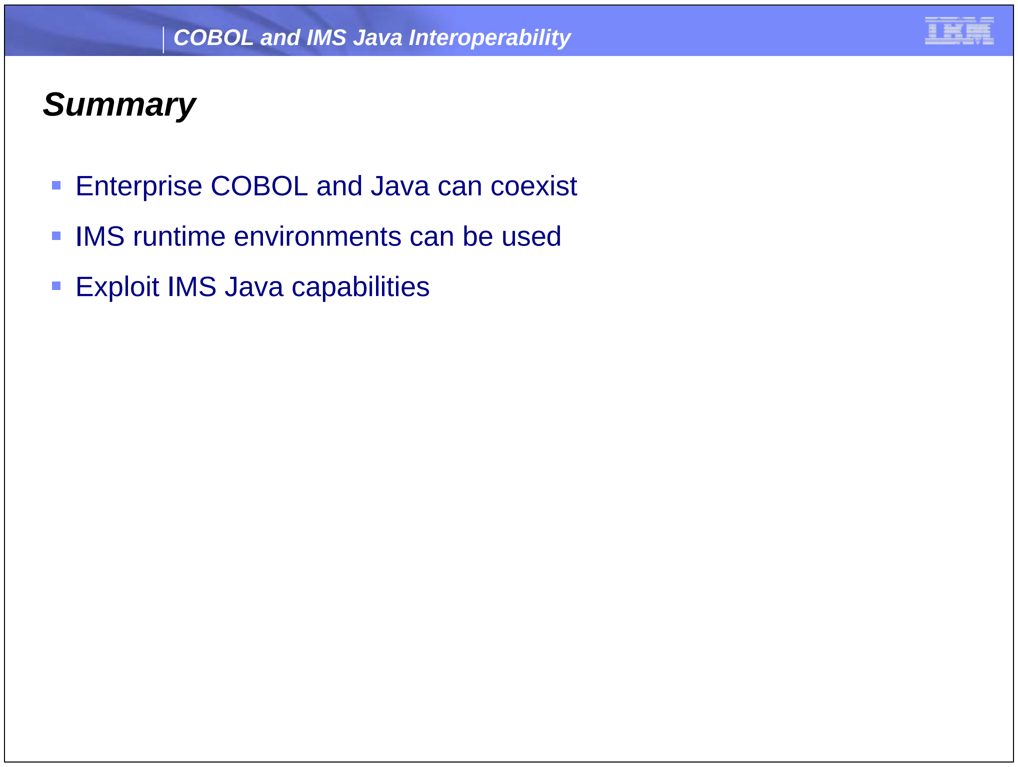
```
AIBSFUNC   = FIND
 AIBRSNM1  = MYDBPCB
 AIBOALEN
 AIBRETRN
 AIBREASN
 AIBRESA1

CALL AIBTDLI INQY,MYAIB,IOAREA
Set address of MYDBPCB to AIBRESA1
CALL MY-COBOL-SUBRTN USING MYDBPCB
```



IMS Dealer Application COBOL Wrapper to COBOL Back-END

Call “CBLTDLI" using GN, MYDBPCB, MODEL-SEGMENT, model-ssa

If dbstat is = spaces



COBOL and IMS Java Interoperability

Summary

- Enterprise COBOL and Java can coexist
- IMS runtime environments can be used
- Exploit IMS Java capabilities

Simplify the development and maintenance of IMS applications by leveraging new Java services as subroutines accessed by existing COBOL business logic

• Use industry standard application APIs, such as JDBC™, J2EE™ and XML, to reduce the programming effort required for building solutions

• Integrate with other products that support the Java API, for example, the Java Message Service support provided by WebSphere® MQ