Integrating COBOL with Java

Session S09
GSE Oslo

Tom Ross
May 30, 2006

Overview

- COBOL:Java interoperation
  - OO COBOL syntax
- Uses other features of Enterprise COBOL
  - Unicode
  - Multi-threading
- New: J2C connector tool
  - Java data mapping for COBOL group

COBOL:Java Interoperation

- Object-oriented COBOL syntax
  - enable COBOL and Java interoperation within an address space
    - COBOL invocation of Java
    - COBOL class defines methods that can be invoked from Java
- Implementation based on the Java Native Interface (JNI)
  - COBOL INVOKE statement maps onto Java JNI calls
  - COBOL class methods definitions define Java native methods
- Documentation and assistance in mapping Java data types to/from COBOL
- Support for JNI programming in COBOL
  - COBOL COPY file analogous to jni.h,
    enables access to JNI callable services
- Prerequisite: IBM Java 2 Technology Edition SDK 1.3 or 1.4

COBOL:Java Interoperation

- You can now use the Java interoperability extensions to access Enterprise Java Beans (EJB) that run on a J2EE-compliant EJB server
  - WebSphere Application Server (WAS) is J2EE-compliant
- Client COBOL would access the following programming interfaces using INVOKE:
  - Java Naming and Directory Interface (JNDI) to locate
  - EJB services and components
  - Java ORB to invoke methods on enterprise beans
- WAS requires several of the COBOL V3 features:
  - Java-based OO *and therefore*
  - Unicode *plus*
  - Multithreading

COBOL:Java interoperation - environments

- z/OS Unix
  - Including WAS
- z/OS Batch
- IMS Java dependent regions
  - JMP - Java Message Processing region
  - JBP - Java Batch Processing region
- Windows (Windows COBOL component of WebSphere Developer for z/Series)
- AIX (IBM COBOL for AIX)

Note:

- COBOL object-oriented syntax for Java interoperation is not supported under CICS
- Under CICS, Java and COBOL can interoperate (at the LINK level) using JCICS commands

Client-side syntax

Declare referenced class and full external class name

```
Configuration section.
Repository paragraph.
Class Employee is 'com.acme.Employee'.
```

Declare object reference

```
01 anEmployee usage object reference Employee.
```

Create instance object

```
Invoke Employee New using by value id
  returning anEmployee
```

Invoke instance method

```
Invoke anEmployee 'payRaise'
  using by value amount
```

Invoke static method

```
Invoke Employee 'getNbrEmployees'
  returning totalEmployees
```

COBOL native method - syntax

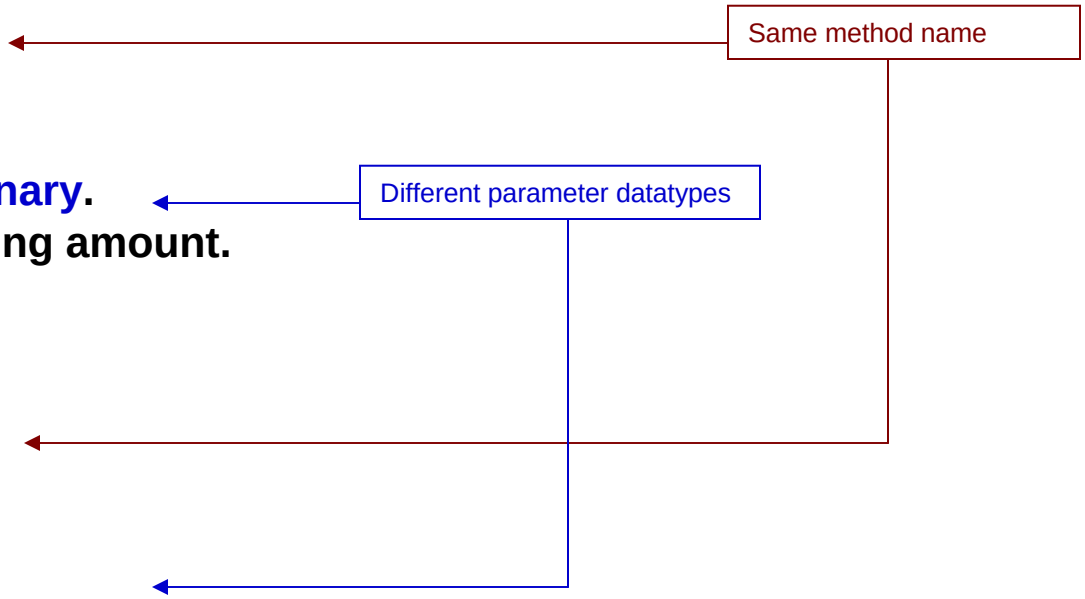
```
Identification Division.
Class-id. Manager inherits Employee.
Environment Division.
Configuration section.
Repository.
Class Manager is 'com.acme.Manager'
Class Employee is 'com.acme.Employee'.
Identification division.
Object.
Procedure Division.
  Identification Division.
  Method-id. 'Hire'.
  Data Division.
  Linkage section.
  01 anEmployee usage object reference Employee.
  Procedure Division using anEmployee.
     …
  End method 'Hire'.
End Object.
End class Manager.
```

COBOL classes

- OBJECT paragraph defines object instance methods
- FACTORY paragraph defines static methods
- COBOL classes can inherit from COBOL or Java classes
- Java classes can inherit from COBOL classes
- Methods can override inherited methods
- Methods can be overloaded
- Method names can be formed using Unicode characters
- Method parameters must be COBOL data types that map to Java data types
- Method parameters must be passed BY VALUE
- Methods can CALL procedural COBOL programs or INVOKE other methods (COBOL or Java)

COBOL methods can be overloaded

```
Identification Division.
Class-id. Account inherits Base.
  …
 Identification Division.
 Method-id. 'credit'.
 Data Division.
 Linkage section.
 01 amount pic S9(9) binary.
 Procedure Division using amount.
  …
 End method 'credit'
 Identification Division.
 Method-id. 'credit'.
 Data Division.
 Linkage section.
 01 amount comp-2.
 Procedure Division using amount.
  …
 End method 'credit'.
End Object.
End class Account.
```



Access to JNI services from COBOL

- Function pointers for JNI services are in the JNI Environment Structure

Access JNI Environment pointer

- New special register JNIEnvPtr

Access JNI Environment Structure and JNI callable services

```
Linkage section.
COPY 'JNI.cpy'
Procedure division.
Set address of JNIEnv to JNIEnvPtr
Set address of JNINativeInterface to JNIEnv
```

Check if an exception has been thrown by a Java routine

```
Invoke aJavaObject 'someJavaMethod'
Call ExceptionOccurred              ←this is a JNI function pointer
  using by value JNIEnvPtr
  returning exceptionObject
If exceptionObject not = null
  Display 'Caught an unexpected exception'
  Call ExceptionClear using by value JNIEnvPtr
  Invoke exceptionObject 'PrintStackTrace'
  Goback
End-if
```

JNI services for string data

Unicode-oriented JNI services for Strings, part of the standard SDK:

NewString GetStringChars
GetStringLength ReleaseStringChars

- Convert between Java String objects and COBOL Unicode data (PIC N(*n*) USAGE NATIONAL)
- Access these services with CALL *function-pointer* statements
  - function pointers in the JNI Environment Structure

EBCDIC-oriented services, provided by IBM Java 2 SDK for z/OS:

NewStringPlatform GetStringPlatformLength GetStringPlatform

- Convert between Java String and COBOL alphanumeric data (PIC X(*n*) USAGE DISPLAY)
- Access CALL '*literal*' statements
  - these services are DLLs

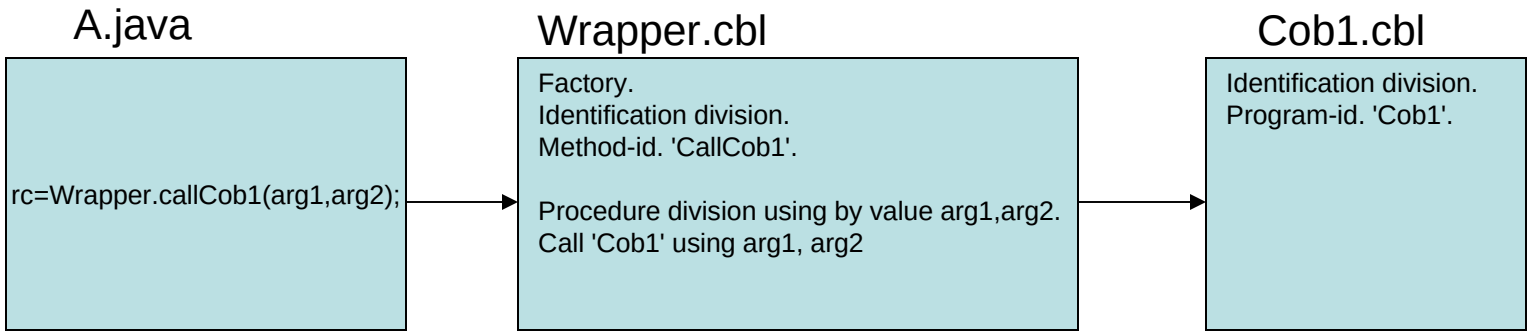
Interoperable data types for method parameters

Java	**COBOL**
boolean	01 B pic X. 88 B-false value X'00'. 88 B-true value X'01' through X'FF'.
byte	Pic X or Pic A
short	Pic S9(4) usage binary or comp-5
int	Pic S9(9) usage binary or comp-5
long	Pic S9(18) usage binary or comp-5
float	Usage comp-1
double	Usage comp-2
char	Pic N usage national
class types (object references) including strings and arrays	Usage object reference *class-name*

Accessing existing procedural COBOL code from Java

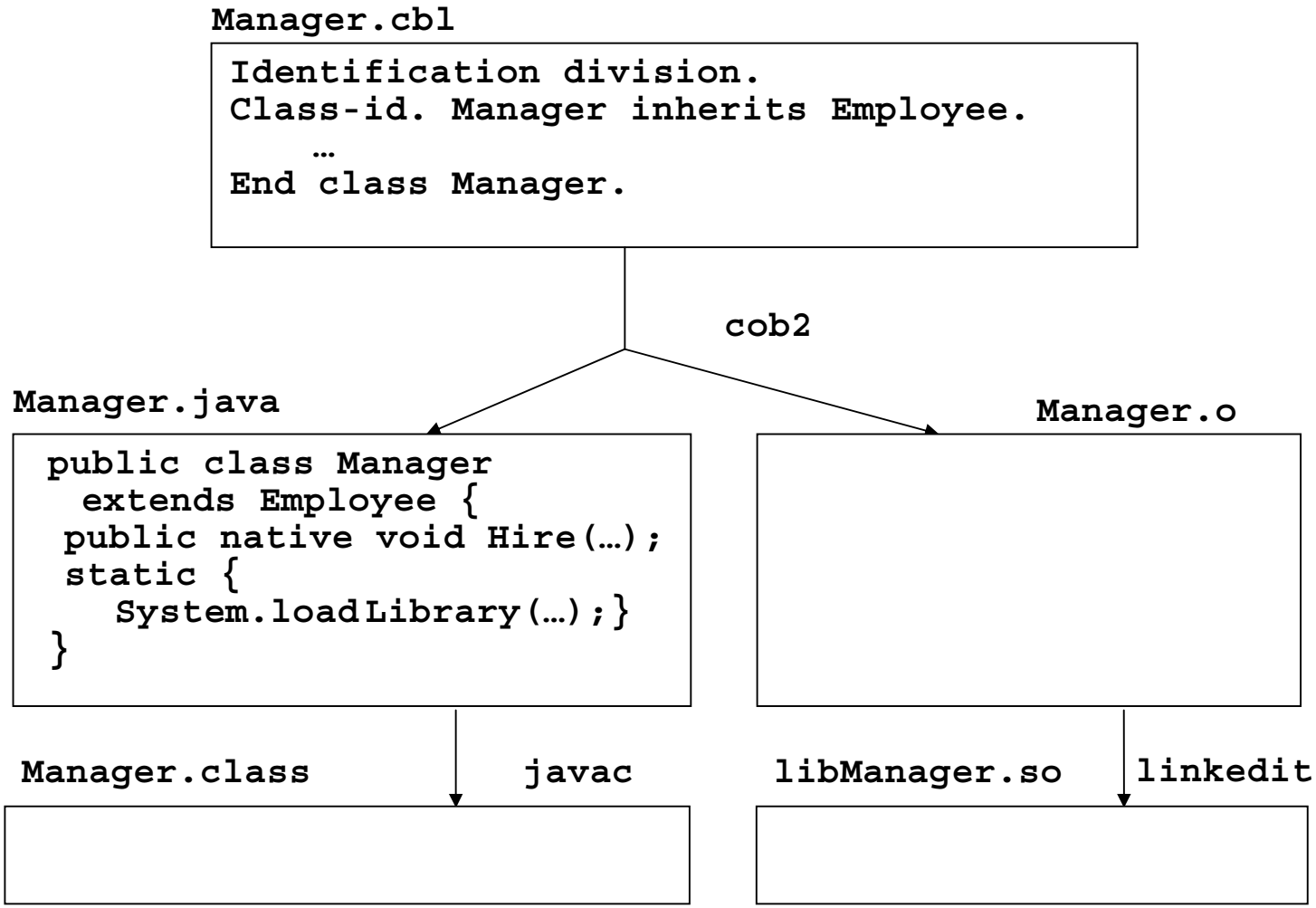
- Write an OO COBOL wrapper class for the existing procedural COBOL program
- Define a Factory method containing a CALL to the COBOL program
- Java client uses a static method invocation to invoke the wrapper, e.g.

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



Compile and link of 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: lib*classsname*.so
- Java class is compiled (with javac) to form *classname*.class


Manager.cbl
Identification division.
Class-id. Manager inherits Employee.
…
End class Manager.
cob2
Manager.java
public class Manager
extends Employee {
public native void Hire(…);
static {
System.loadLibrary(…);}
}
Manager.o
Manager.class
javac
libManager.so
linkedit

Getting started with COBOL and Java interoperability

- Ensure you have the Java 2 Technology Edition SDK installed
  - SDK 1.4,
  - SDK 1.3.1 (minimum level for under IMS), or
  - SDK 1.3.0
- Ensure that the optional HFS components of Enterprise COBOL V3 have been installed
- Ensure that z/OS Unicode Conversion Services are configured for COBOL use.
- See the sample OO application and makefile shipped with COBOL in /usr/lpp/cobol/demo/oosample. Try compiling and running this application.

New: using z/OS Java 2 Technology Edition V1.4

- z/OS Java 2 SDK 1.4 is based on the Language Environment XPLINK linkage convention
  - must run in an XPLINK environment
- COBOL uses standard LE linkage, but will run in an XPLINK environment
- LE supports XPLINK:non-XPLINK transition at DLL boundaries
- Applications that will use XPLINK at any point must be initialized in XPLINK mode

Using z/OS Java 2 Technology Edition V1.4

- Building COBOL:Java applications with V1.4 is unchanged
- Running COBOL:Java applications is unchanged if
  - application starts with Java or the **`main`** method of a COBOL class, and
  - the application is run using the **java** command

  *In these cases, the XPLINK environment is automatically started by the **java** command.*
- If COBOL:Java application starts with COBOL, the LE runtime option XPLINK(ON) must be used
  - *this explicitly initializes the XPLINK environment*
- For z/OS UNIX shell
  - Set _CEE_RUNOPTS="XPLINK(ON)"
  - Set selectively, do not use as default for entire shell session

Java data binding tool

- New tool in Rational Application Developer V6
  - also in products containing RAD V6
    - WebSphere Developer for z/Series,
    - Rational Software Architect, WebSphere Integration Developer, …
- Creates a Java class wrapper for a COBOL group
- COBOL group can contain general COBOL data types:
  - packed or zoned decimal, currency values
  - alphanumeric types, edited types, etc.
- Use Java wrapper objects to pass general COBOL data between Java and COBOL
  - apps no longer limited to the elementary interoperable data types

  Note elementary types must be passed BY VALUE, can not be updated by the receiving code
  - wrapper object reference data items are passed BY VALUE
  - but the objects themselves can be updated by the receiving code

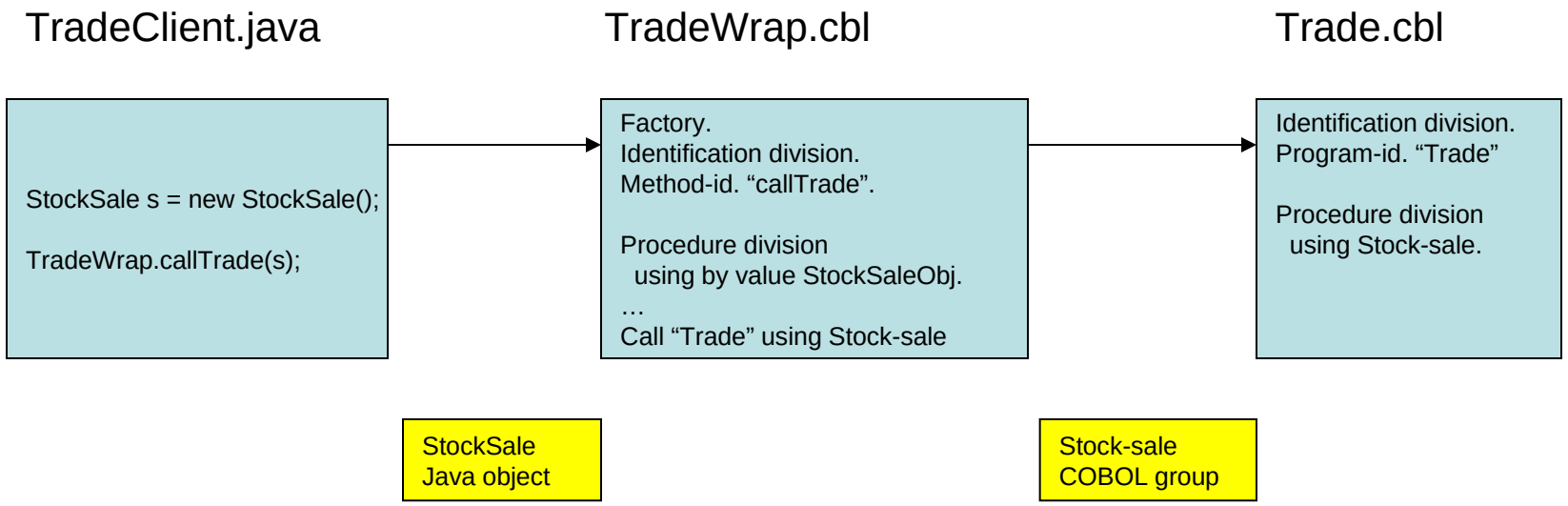
COBOL Importer

- A component of the Java data binding tool
  - provided by the IBM COBOL development group
- Based on the IBM COBOL compiler front-end
- Accurately generates metadata information about COBOL data structures to the Java data binding tool
  - Understands and supports all COBOL syntax
  - Stays in sync with the COBOL language as new releases of IBM COBOL are delivered
  - Currently based on Enterprise COBOL V3R3 syntax level
  - Enterprise COBOL V3R4 syntax will be supported as soon as V3R4 syntax is shipped in IBM Windows COBOL

Enabling the Java data binding tool

- A component of the connector building (J2C) tools
- For information, search help for “J2C”
  - see the entry “Creating J2C Applications”
- To install and enable the tool:
  - Install the J2EE Connector Tools
    (with the Rational Product Updater)
  - Switch to J2EE perspective
  - If you don’t see the J2C wizard under File > New > Other then you need to Enable J2C Capabilities:
    - **Window > Preferences**
    - Expand “Workbench”, Click **Capabilities**
    - Expand “Enterprise Java”
    - Select **Enterprise Java** check box.
    - Click **Apply**, **OK**

Sample application: Java to COBOL
Stock trade demo application



Sample application : Java to COBOL
Stock trade demo application

- StockSale.cpy - COBOL input group data structure
- StockSale.java - wrapper class generated with J2C Connector Tools
- TradeClient.java - standard Java
  - Creates and initializes parameter: a StockSale object instance
  - Drives COBOL trade application via wrapper class
  - Receives back StockSale as updated by COBOL Trade app
- TradeWrap.cbl - OO COBOL wrapper class for Trade.cbl
  - Makes Trade.cbl accessible to Java
  - Input parameter: object instance of StockSale.java
  - Outbound parameter: Stock-sale group data structure
  - Bridges from Java data types to COBOL data types
  - Bridges from Java by-value parameters to COBOL by reference parameters
- Trade.cbl – standard procedural COBOL
  - Processes stock trade
  - Input/Output parameter: StockSale group data structure

Trade.cbl

```
cbl lib,thread,dll,pgmname(longmixed)
Identification division.
 Program-id. "Trade" recursive.
 Environment division.
 Data division.
 Working-storage section.
 1 Commission pic 9(3)V99 value 39.95.
 1 printNumberShares pic z(8)9.
 1 printSharePrice  pic $,$$$,$$$.99.
 Linkage section.
 Copy "StockSale.cpy".
 Procedure division using Stock-sale.
    Move inNumberShares to printNumberShares
    Move inSharePrice to printSharePrice
    Display ">>> COBOL Trade entered. Sell " printNumberShares
          "shares of "  inStockSymbol  " at " printSharePrice
    Move Commission to outCommission
    Compute outTotal = inNumberShares * inSharePrice - Commission
    Goback.
 End program "Trade".
```

StockSale.cpy

```
1 Stock-sale.
      2 inStockSymbol pic X(4).
      2 inNumberShares pic 9(9) binary.
      2 inSharePrice  pic 9(6)V99 packed-decimal.
      2 inSaleType pic X.
        88 LimitOrder  value "L".
        88 MarketOrder value "M".
      2 outCommission pic $$$9.99.
      2 outTotal  pic $,$$$,$$9.99.
```

StockSale.java wrapper class

Generated from Stock-sale.cpy with Java data binding tool.

Contains:

- Java byte array object containing actual COBOL data structure contents
- get*XXX()*, set*XXX()* methods for each COBOL data item *XXX, e.g.*

  public void setInStockSymbol(String inStockSymbol)
  public BigDecimal getInSharePrice()

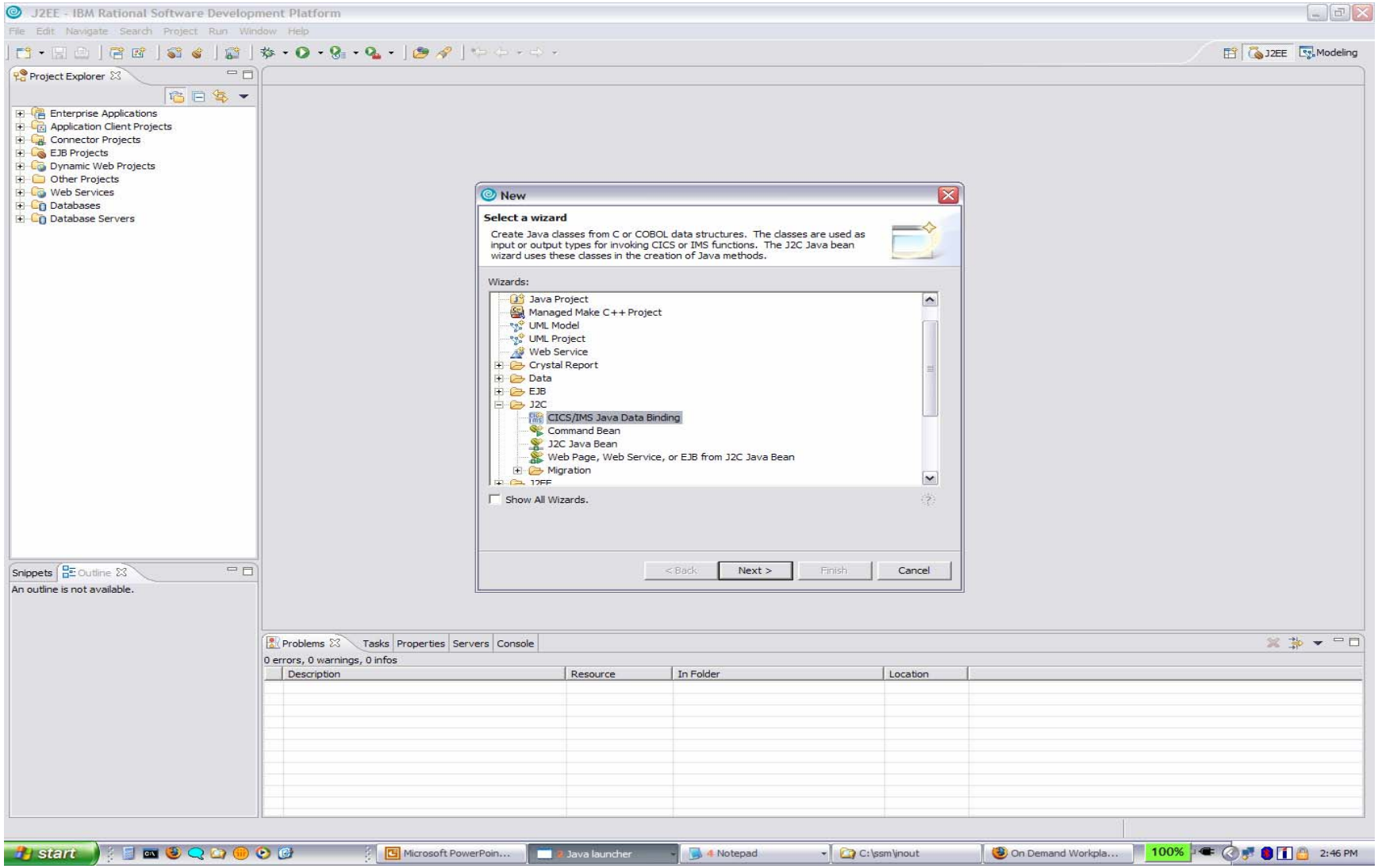
  – used by Java client to access the COBOL data
- getBytes() method
  – retrieve Java byte array containing COBOL data structure contents
- setBytes() method
  – set the Java byte array to COBOL data structure contents

Create StockSale java wrapper class from Stock-sale.cpy

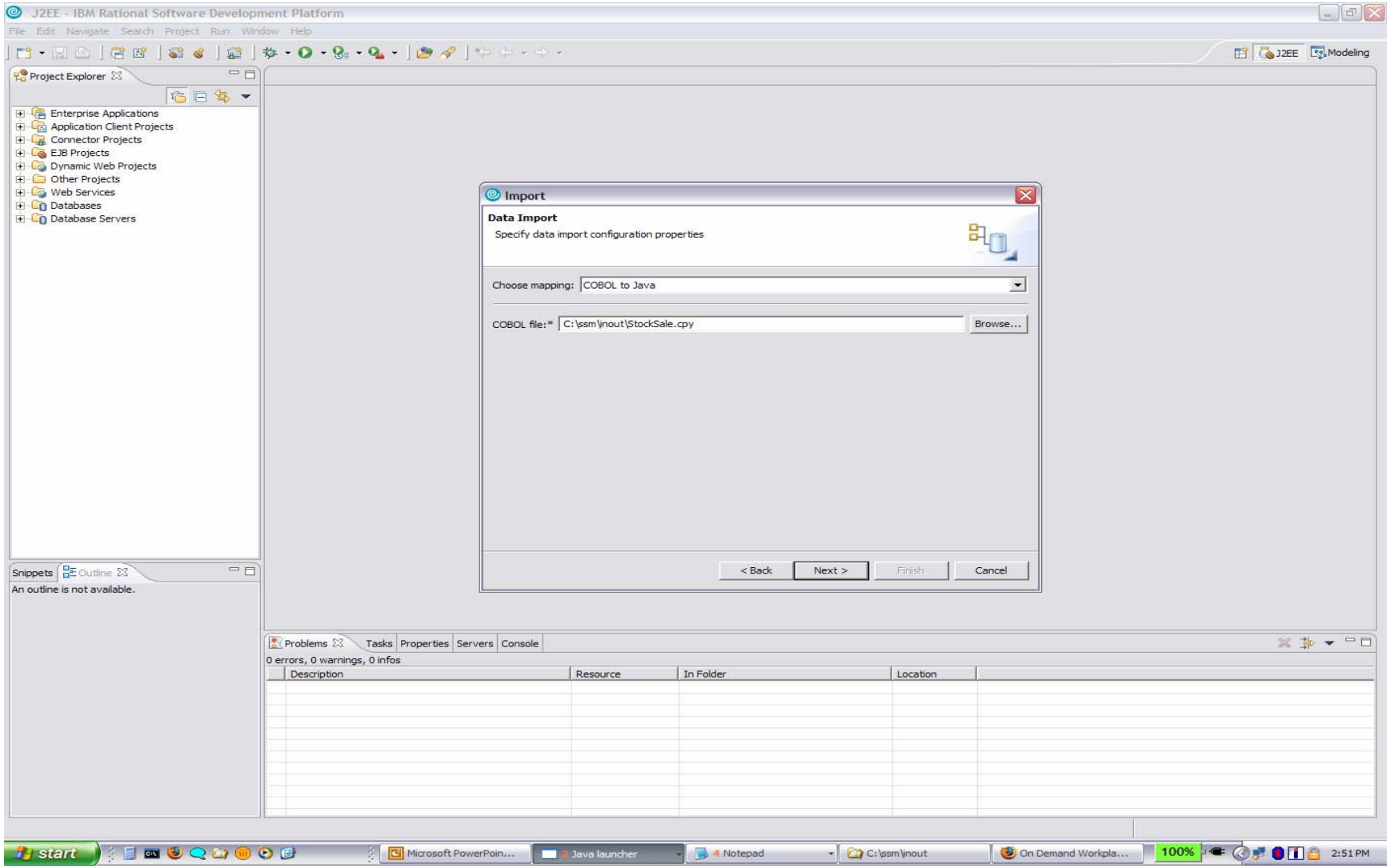
In Rational Application Developer

- Start the Java data binding wizard: File > New > Other > J2C
- Select "CICS/IMS Java Data Binding"
- Choose mapping: "COBOL to Java"
- Browse for COBOL copy book in file system, select: StockSale.cpy
- In the **Importer** panel,
  - Set Platform: z/OS
  - Set Code page: as appropriate, e.g. IBM-1142 for Denmark/Norway
  - Click "Query" and select Data Structure: Stock-sale
  - Next
- In the **Saving Properties** panel,
  - Set Generation Style: Default
  - Project Name: Trade, use "New" to create new Java project
  - Package Name: trade
- Click Finish

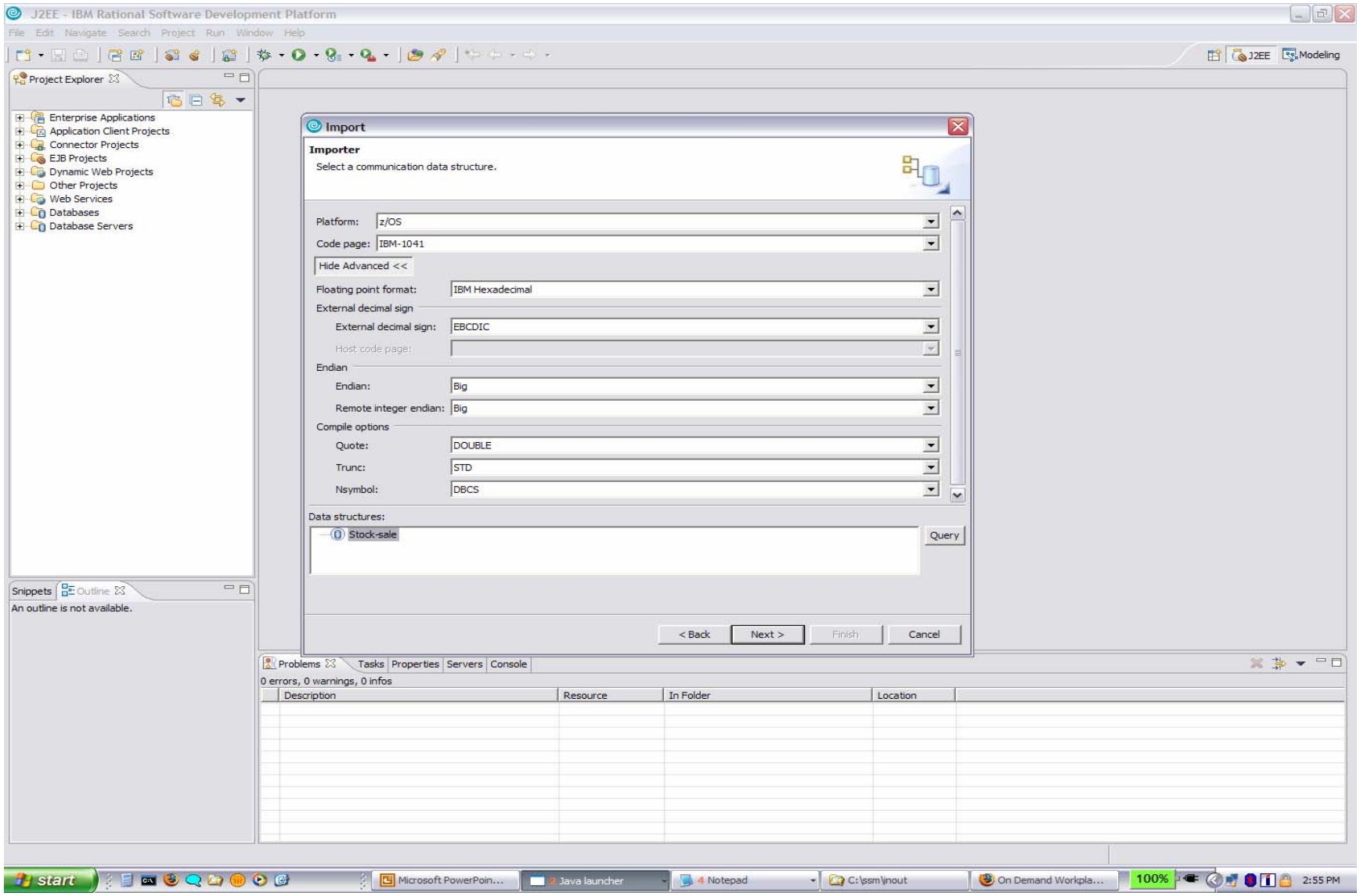
Start Java data binding wizard …



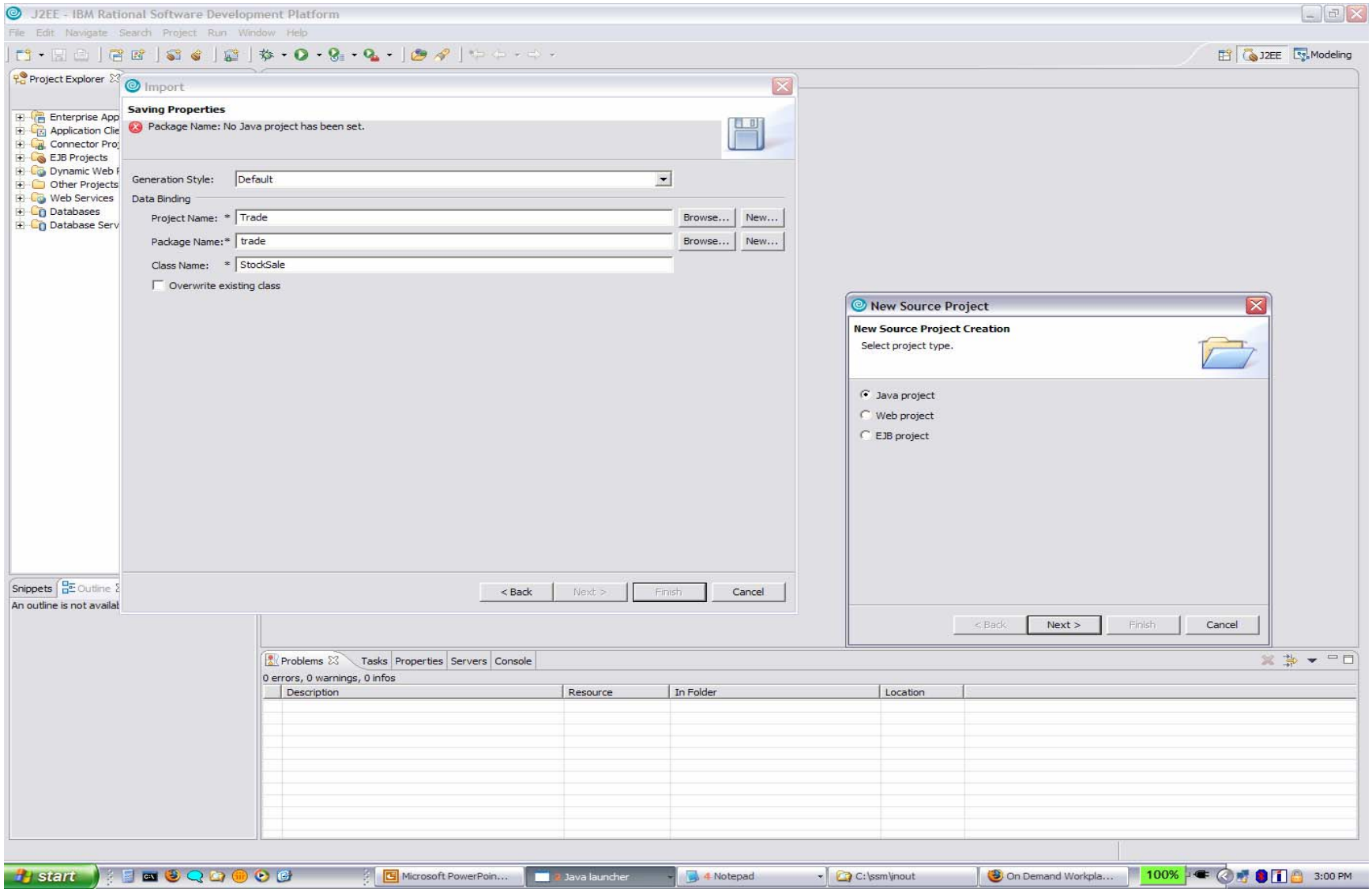
Start COBOL importer …



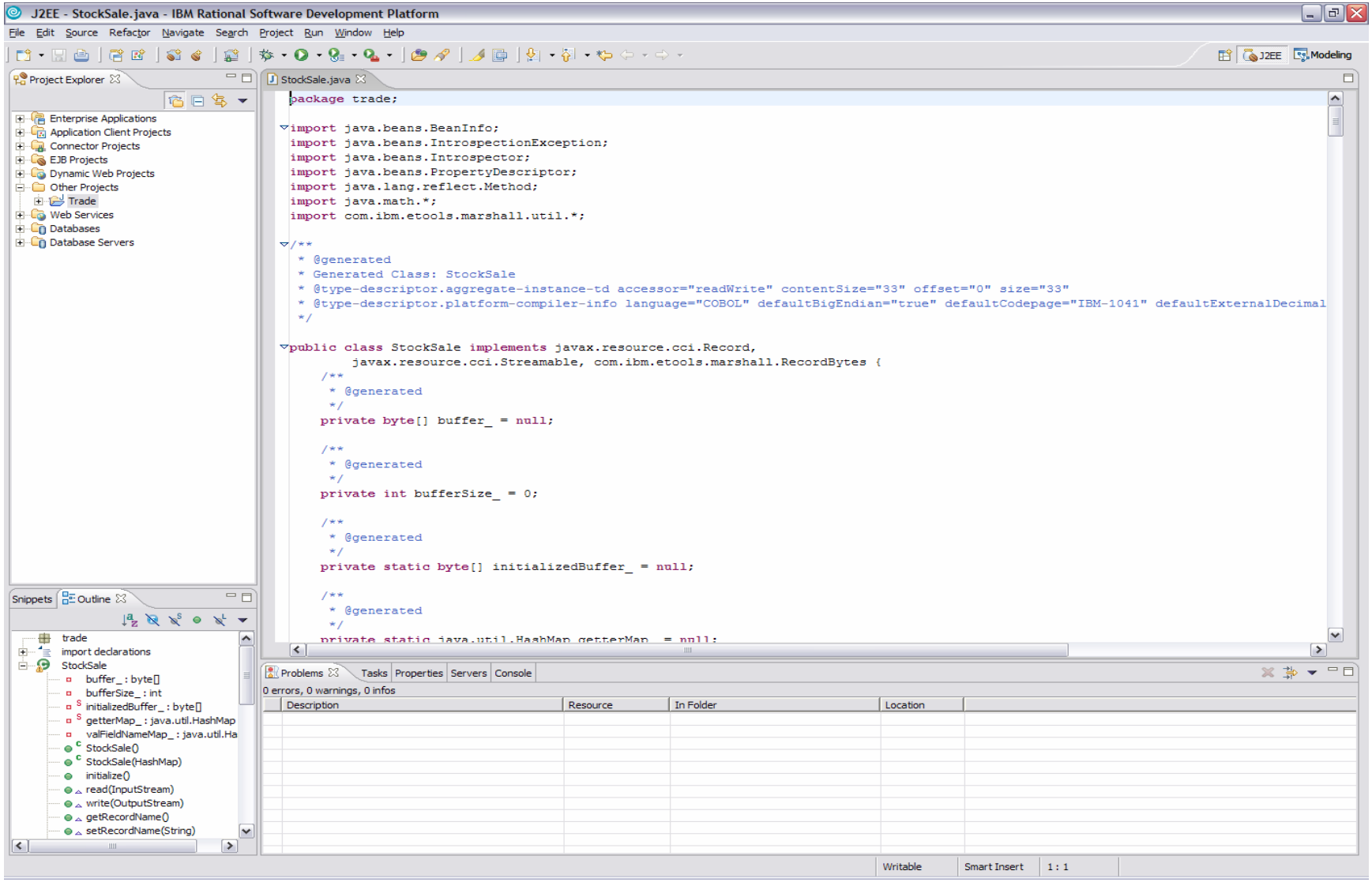
COBOL Importer …



Saving properties …



Generated StockSale.java wrapper class



TradeWrap.cbl

```
cbl lib,thread,dll,pgmname(longmixed)
 Identification Division.
 Class-id. TradeWrap inherits Base.
 Environment Division.
 Configuration section.
 Repository.
     Class Base is "java.lang.Object"
     Class jbyteArray is "jbyteArray"
     Class StockSale is "trade.StockSale"
     Class TradeWrap is "trade.TradeWrap".
  Identification Division.
  Factory.
   Procedure division.
    Identification Division.
    Method-id. "callTrade".
    Data division.
    Local-storage section.
    01 StockSalePtr pointer.
    01 StockSaleByteArray object reference jbyteArray.
    Linkage section.
    01 StockSaleObj object reference StockSale.
       Copy "StockSale.cpy".
       Copy "JNI.cpy".
    Procedure division using by value StockSaleObj.
      Display ">>> OO COBOL TradeWrap.callTrade entered"
      set address of JNIEnv to JNIEnvPtr
      set address of JNINativeInterface to JNIEnv
      Invoke StockSaleObj "getBytes"  returning StockSaleByteArray
      Call GetByteArrayElements using by value JNIEnvPtr StockSaleByteArray 0
        returning StockSalePtr
      Set address of Stock-sale to StockSalePtr
      Call "Trade" using Stock-sale
      Call ReleaseByteArrayElements  using by value JNIEnvPtr StockSaleByteArray StockSalePtr 0.
    End method "callTrade".
  End Factory.
 End class TradeWrap.
```

TradeWrap.cbl - callTrade method

```
Method-id. "callTrade".
  Data division.
  Local-storage section.
  01 StockSalePtr pointer.
  01 StockSaleByteArray object reference jbyteArray.
  Linkage section.
  01 StockSaleObj object reference StockSale.
    Copy "StockSale.cpy".
    Copy "JNI.cpy".
  Procedure division using by value StockSaleObj.
   Display ">>> OO COBOL TradeWrap.callTrade entered"
   set address of JNIEnv to JNIEnvPtr
   set address of JNINativeInterface to JNIEnv
   Invoke StockSaleObj "getBytes"  returning StockSaleByteArray
   Call GetByteArrayElements
     using by value JNIEnvPtr StockSaleByteArray 0
     returning StockSalePtr
   Set address of Stock-sale to StockSalePtr
   Call "Trade" using Stock-sale
   Call ReleaseByteArrayElements
     using by value JNIEnvPtr StockSaleByteArray StockSalePtr 0.
  End method "callTrade".
```

TradeClient.java

```
import trade.*;
import java.math.*;
class TradeClient {
   public static void main(String[] args) {
      System.out.println(">>> Java TradeClient entered.");
      StockSale s = new StockSale();
      s.setInStockSymbol("IBM");
      s.setInNumberShares(200);
      s.setInSharePrice(new BigDecimal("83.09"));
      s.setInSaleType("L");

      TradeWrap.callTrade(s);

      System.out.println(">>> Trade completed, Commission: "
                        + s.getOutCommission()
                        + " Total proceeds: "
                        + s.getOutTotal());
   }
}
```

Makefile to build the application on z/OS Unix

```
CLASSP=marshall.jar:$(JAVA_HOME)/standard/jca/connector.jar:trade.jar:.

all: trade/libTrade.so trade/libtrade_TradeWrap.so trade/TradeWrap.class TradeClient.class
Trade.o: trade/Trade.cbl
    cob2 -c -bdll -I./trade ./trade/Trade.cbl
trade/libTrade.so: Trade.o
    cob2 -o ./trade/libTrade.so -bdll,map,list,xref,compat=pm3 Trade.o  >Trade.blst
TradeWrap.o: trade/TradeWrap.cbl
    cob2 -c -bdll -I$(COBOLCOPYPATH):./trade ./trade/TradeWrap.cbl
    mv TradeWrap.java trade
trade/libtrade_TradeWrap.so: TradeWrap.o libTrade.x
    cob2 -o ./trade/libtrade_TradeWrap.so -bdll,map,list,xref,compat=pm3 TradeWrap.o \
    libTrade.x $(JAVASIDEDECKPATH)/libjvm.x $(COBOLSIDEDECKPATH)/igzcjava.x \
    >TradeWrap.blst
trade/TradeWrap.class: trade/TradeWrap.java
    javac -classpath $(CLASSP) trade/TradeWrap.java
TradeClient.class: TradeClient.java
    javac -classpath $(CLASSP) TradeClient.java
clean:
    rm trade/TradeWrap.java trade/*.so *.o *.class trade/*.class *.lst *.blst *.log *.x \
    2>/dev/null
```

Application build on z/OS Unix

- Export the Trade project to a jar file in the Windows file system
- Upload the jar file to an HFS directory in z/OS Unix, together with the Java client
- Also upload marshall.jar from the WebSphere V6.0 runtime in RAD, if not already available on z/OS from z/OS WebSphere
- z/OS Java V1 R4 contains the required JCA connector frameworks
- Source files for Trade.cbl and TradeWrap.cbl must be in a subdirectory trade, since the application is in a package trade
- Use make command to run the makefile

Application execution on z/OS Unix

Sample command file **run**

```
export LIBPATH=$LIBPATH:./trade
CMD="java   \
 -cp trade.jar:marshall.jar:$JAVA_HOME/standard/jca/connector.jar:./trade:.  \
 TradeClient"
echo $CMD
$CMD
```

Application output:

```
run
java -cp trade.jar:marshall.jar:/usr/lpp/java/J1.4/standard/jca/connector.jar:./trade:.
     TradeClient
>>> Java TradeClient entered.
>>> OO COBOL TradeWrap.callTrade entered
>>> COBOL Trade entered. Sell       200 shares of IBM  at       $83.09
>>> Trade completed, Commission:  $39.95 Total proceeds:   $16,578.05
```

Enterprise COBOL prerequisites

- z/OS 1.4 or later
- z/Architecture processor with extended-translation facility 2 *(for Unicode support)*
- Java 2 Technology Edition SDK 1.3.x or 1.4 *(for COBOL:Java interoperability support*)
- Language Environment:
  PTFs for APAR PQ95214
- Enterprise COBOL V3R4 compiler:
  PTF for APAR PK07977
- DB2 coprocessor:
  PTF for APAR PQ93857