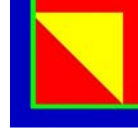


PERCobol™

Enterprise JavaBeans Guide

The contents of this manual may be revised without prior notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical for any purpose without the expressed written permission of LegacyJ Corporation.

©Copyright LegacyJ Corp.. 2001
All Rights Reserved.

First Edition: March, 2001

LegacyJ has made every effort to ensure that this manual is correct and accurate, but reserves the right to make changes without notice at its sole discretion at any time.

© Copyright LegacyJ Corporation 2001. All Rights Reserved

Preface

This Enterprise JavaBeans Guide provides guidelines for the usage of PERCobol programming constructs related to building Enterprise Java Beans. Programming with PERCobol adds features that extend standard COBOL-85 and offers functional capabilities which are not defined as part of the standard.

PERCobol COBOL extensions should be easy to comprehend to the experienced COBOL programmer, and a basic knowledge of COBOL-85 is assumed.

Trademarks

- PERCobol, BlueJ, Synkronix and LegacyJ are trademarks of LegacyJ Corporation.
- AIX, OS/390, OS/2 and OS/400 are trademarks of International Business Machines.
- HP-UX and MPE are registered trademarks of Hewlett-Packard Corporation
- IBM is the registered trademark of International Business Machines.
- Network Computer trademark of Network Computer Inc.
- Java is a trademark of Sun Microsystems
- SCO is a trademark of Santa Cruz Operation Corporation.
- SlickEdit is a trademark of Micro Edge Corporation.
- UNIX is a registered trademark licensed exclusively to X/Open Company Limited.
- Windows NT and Windows 95/98 are trademarks of Microsoft Corporation.
- Other company, product and service names may be trademarks of service marks of others.

Document Overview

Install Guide	The Install Guide contains installation instructions for the PERCobol development platforms.
User's Guide	The User's Guide contains information on how to use PERCobol to develop COBOL programs.
Programmer's Guide	The Programmer's Guide contains information on programming the extensions to PERCobol which add more power and flexibility than found in standard COBOL.
Language Reference	The Language Reference contains the language skeleton of PERCobol, showing the syntax for each part of the COBOL language

Related Documents

COBOL

COBOL programming books for COBOL-85 are generally applicable to PERCobol. All standard features available in HIGH level COBOL-85 are available in PERCobol. Do note that COBOL programming books often speak of proprietary extensions available in particular COBOL versions; unless also documented within PERCobol manuals, these features may not be applicable to PERCobol. System dependent features are not generally applicable to PERCobol or to other systems.

Java

On most platforms, with native COBOL compilers it is generally possible to call operating system functions to perform functions that are not directly available to COBOL, such as putting a process to sleep for a period of time. With PERCobol, those programmers who traditionally called operating system functions will instead call Java API (Application Programming Interface) functions with the INVOKE statement.

Those programmers who wish to learn how to control the Java API may find Java API books useful. The name of the method to call and the order of the parameters is always the same for PERCobol as it is for Java. The simple data types of Java (byte, short, int, long, char, String) will automatically be converted between COBOL and Java when used from the PERCobol INVOKE statement.

Supported Environments

Development

Development is performed through the use of the PERCobol compiler and Java Development Kit and optionally an Integrated Development Environment. The development of PERCobol applications is done on a single system for which a license has been purchased. A PERCobol development environment only works on one platform.

Operating Systems and Hardware Supported by PERCobol:

Windows 95/98 for Intel x86

Windows NT 4.0 for Intel x86

Windows 2000 for Intel x86

SCO UnixWare 7 for Intel x86

Sparc Solaris 2.6

AIX 4.2 for RS/6000

OS/390

HP UX 11.0

HP MPE/iX 6.5 with POSIX and Java patches

Contents

INTRODUCTION 1

EJB TERMS 1
OVERVIEW 1
JAVA NAMING DIRECTORY INTERFACE 2
EJB EXECUTION PROCESS 2

INSTALL JBOSS 3

CREATING THE ENTERPRISE JAVABEAN COMPONENT 4

CODING THE CLASSES AND INTERFACES 6

REMOTE INTERFACE 7
HOME INTERFACE 8
EJB CLASS 9
DEPLOYMENT DESCRIPTOR 12

PACKAGING AND DEPLOYING THE EJB COMPONENT 13

EJB CLIENT 13

Introduction

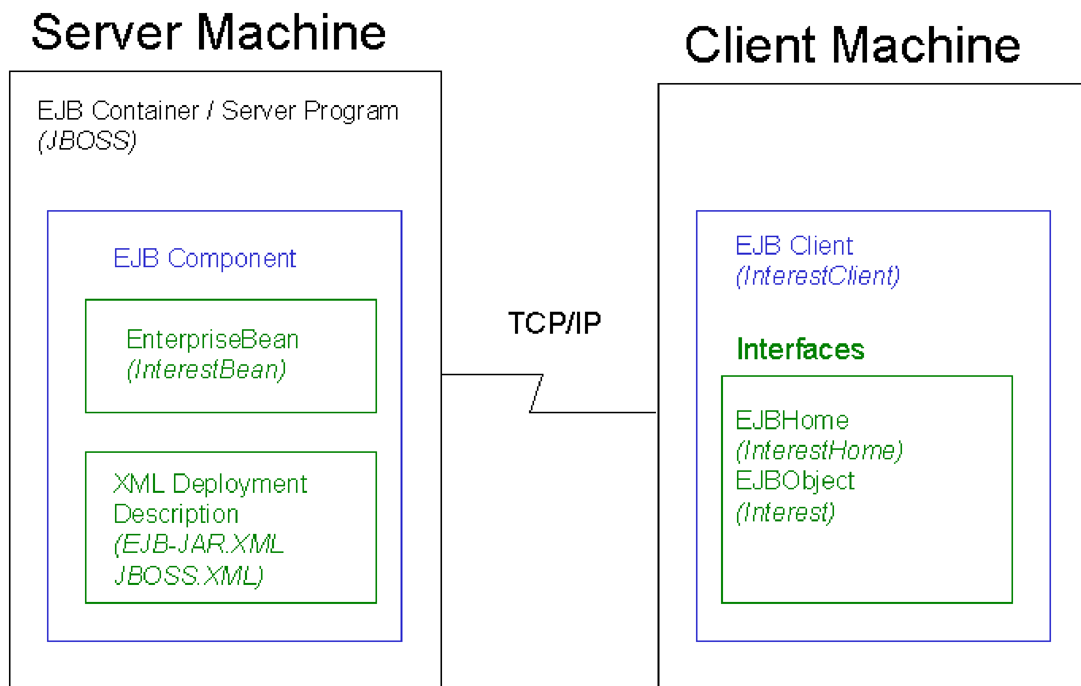
The EJB Container used in this sample is JBoss, a freely available EJB Container available at http://www.jboss.org. This Enterprise JavaBeans (EJB) Guide follows the JBoss tutorial found at http://www.jboss.org/documentation/jboss_win32_1.html.

EJB Terms

Term	Description	Example
EnterpriseBean	Contains the business logic	InterestBean
EJB Client	User of business logic	InterestClient
EJBObject	Client view of the EJB Component	Interest
EJBHome	Used by Client to obtain the EJBObject.	InterestHome
JNDI	Java naming and directory interfaces; used to obtain EJBHome	

Overview

EJB's are server side components in a standard format available for use by clients, which may include applications, applets, servlets and other EJBs.



To execute the EJB sample, an EJB Container, EJB Component and EJB Client are required. The EJB Container used in this example is JBoss, a freely available EJB Container; other EJB Containers will be the same for coding, but may differ in setup and the values for the environment variables.

The EJB Container provides the distributed business object framework. It provides services to the EJB Component, allowing it to concentrate on the business functionality rather than the developing a custom distributed framework. An EJB Container can provide additional services such as transparently allowing a failed object to be replaced by a successful one. Because the Java platform is cross-platform, it can even allow transparent access of a bean to be split among different operating systems. There are many EJB Containers, each with its own set of strengths. Some concentrate on cross-platform issues, some on interoperability, some on wireless clients, etc.

The EJB Component provides the business functionality to remote clients. This EJB component may be written in PERCobol or Java.

The EJB Client uses an EJB Component. The EJB Client may be written in PERCobol or Java. If the EJB container supports it, it may also include additional access methods such as wireless protocols or COM.

Java Naming Directory Interface

Client Settings	**Description**
Java.naming.factory.initial	Class used to lookup EJBs by name
Java.naming.provider.url	Informs lookup class where to located naming service

EJB Execution Process

1. Copy EJB Component to server deploy directory
2. Start Server
3. Start Client

Install JBoss

Installation of JBoss follows the instructions available at the JBoss site for Win32 or Unix.

An additional note is that the PERCobol runtime, percobol.jar, must be found by EJB Container for PERCobol EJB Components to function. This may be done in the deploy tool, which can combine an application such as an EJB Component with the PERCobol runtime into a single deployable .jar file, or it may be done by adding percobol.jar to the EJB Container's CLASSPATH. If added to the CLASSPATH of the EJB Container, the EJB Component .jar file does not need to include the contents of percobol.jar; if not added to the CLASSPATH, then the deploy tool must be used to combine percobol.jar with the applications classes.

The deploy tool is available on all graphical machines as

<percobol>\bin\deployj.bat or <percobol>/bin/deployj.sh.

The EJB Container should execute normally before proceeding.

Creating the Enterprise JavaBean Component

In this step we will write and compile a simple Enterprise JavaBean. The example -- which is called 'Interest' -- is about as simple as an EJB can get: it is a 'stateless session bean'. Its job is to calculate the amount of compound interest payable on a sum of money borrowed over a specified term with a specified interest rate.

This example looks like it may have a number of lines of code, but most of it is purely structural, identical for all EJB Components created for a particular EJB Container. In fact, there is only one functional line of code in the whole package.

Create a new project in the IDE or directory for command line use. The directory '\percobol\samples\ejb\server' will be used in this text.

Generally, EJB components are deployed in packages. A package is separate namespace for an application preventing names of programs from conflicting with one another. A package may be specified in one of three ways; in the program source, in the IDE when compiling in the IDE, or on the command line when compiling from the command line.

In this case package will be 'com.web_tomorrow.interest'. This sample places the package name in the program source. It may be removed and done at compile time if desired. This means that PERCobol will use the -package compiler option or package field in the Project|PERCobol|PERCobol Properties in the IDE.

An Enterprise JavaBean has a minimum of one class and two interfaces.

1. The EJB Component or Bean class. This implements the methods specified by the remote interface. In this example, the Bean class is 'com.web_tomorrow.interest.InterestBean'.
2. The remote interface. This is the class that exposes methods of the Bean to the outside world. In the example, the remote interface is the class 'com.web_tomorrow.interest.Interest'. Only the methods specified in this interface may be called from remote.
3. The home interface. This specifies how a new bean is created, managed and deleted. The methods in the home interface depend on the type of bean. A stateless session bean has exactly one create method. In this example, the home interface is 'com.web_tomorrow.InterestHome'.

Of course, a Bean can include other classes, other programs, or even other packages, but the classes listed above are the minimum. The classes must be packaged into a JAR, a Java Archive, with a directory structure which reflects the hierarchy of packages. The LegacyJ deploy tool may be used to

create the actual JAR file, but the directory layout needs to be done first. In this example, the classes are in the package com.web_tomorrow.interest, so they need to be in the directory

\percobol\samples\ejb\server\com\web_tomorrow_interest

PERCobol will create this directory automatically from the Cobol source files if the package option is used. In PERCobol, packages are created in the directory described by the package below the source file directory. There also needs to be a directory called META-INF to store the deployment descriptor (always called ejb-jar.xml) and, optionally, another XML file to tell the server about name mappings. With JBoss, this file must be called jboss.xml.

So, before writing the classes, we need a directory structure like this:

\percobol\samples\ejb\server

.cob files are compiled here with -package option, yielding .java and .class files below

com

web_tomorrow

interest

.java and .class files are compiled to here

META-INF

ejb-jar.xml

jboss.xml

If the deploy tool has the \percobol\samples\ejb\server directory files added, the directory structure will be complete. The visible files to be deployed should begin with com/ or META-INF/.

Of course, in a custom project rather than this sample project, the directory \percobol\samples\ejb\server would be replaced with a custom directory structure representing the project and company. (The standard for package names is the reverse of the company URL; that is if the web site is www.mycompany.com, then the project xyz at that company would be in package com.mycompany.xyz.)

Coding the classes and interfaces

We need one class, the Bean itself, and two glue interfaces, the remote interface and the home interface. All the '.java' files are compiled into the subdirectory .\com\web_tomorrow\interest.

The interfaces have no method body, but rather only a definition of what methods are guaranteed to be present in any class implementing the interface. Using an interface is like using a class; the object is actually an instance of a class, but when accessing it via an interface, only the interface methods can be used.

In this case, the interface specifies the methods exposed to the EJB client, and the allowed method of creating the bean.

Remote Interface

The remote interface in this example is very simple.

```
* This example is coded for the JBoss EJB Container.  The naming
* provider and setup information may differ for other EJB Containers.
* The logic flow remains the same. (http://www.jboss.org)
*
* This example requires J2EE.  JDK 1.2+ plus JBoss is sufficient.
*
* The CLASSPATH to compile must include \jboss\lib\ext\ejb.jar in
* addition to the percobol.jar required for any Cobol program.
*
* Set the package name; this could be done from the command line
* using the '-package name' directive instead.  Setting a package
* is generally not necessary for an EJB, but it's good practice.

$SET PACKAGE com.web_tomorrow.interest

*
* An INTERFACE-ID is the same as a CLASS-ID, but without any
* method definitions.  No special-names, no data other than
* linkage, and nothing in the procedure division other than
* the procedure division using is allowed.  This defines a
* contract which another class-id program may implement.
*

IDENTIFICATION DIVISION.
INTERFACE-ID. "Interest" INHERITS "javax.ejb.EJBObject".

* Define one method with three parameters which is capable
* of throwing a RemoteException.

METHOD-ID. "calculateCompoundInterest" THROWS "java.rmi.RemoteException".
DATA DIVISION.
LINKAGE SECTION.
  77 principle COMP-2.
  77 rate COMP-2.
  77 periods COMP-2.
  77 result COMP-2.
*
* The BY VALUE clause forces the Java-style usage of a variable;
* this means that COMP-2 will be a Java 'double' floating point
* in the definition.  A BY REFERENCE or BY CONTENT is suitable
* for communication with other PERCobol programs, but for any
* interface or class which will have Java accessors, the BY VALUE
* is simpler.
*

PROCEDURE DIVISION USING BY VALUE principle, rate, periods GIVING result.
END-METHOD.

END-INTERFACE.
```

The remote interface specifies only one 'business method' called calculateCompoundInterest. This is the list of business methods available to any clients.

Home Interface

The home interface is even simpler. The home interface describes the creation method.

```
* This example is coded for the JBoss EJB Container.  The naming
* provider and setup information may differ for other EJB Containers.
* The logic flow remains the same. (http://www.jboss.org)
*
* This example requires J2EE.  JDK 1.2+ plus JBoss is sufficient.
*
* The CLASSPATH to compile must include \jboss\lib\ext\ejb.jar in
* addition to the percobol.jar required for any Cobol program.  This
* is the javax.ejb.* package; its location may vary in other EJB
* vendors.
*
* Set the package name; this could be done from the command line
* using the '-package name' directive instead.  Setting a package
* is generally not necessary for an EJB, but it's good practice.

$SET PACKAGE com.web_tomorrow.interest

*
* An INTERFACE-ID is the same as a CLASS-ID, but without any
* method definitions.  No special-names, no data other than
* linkage, and nothing in the procedure division other than
* the procedure division using is allowed.  This defines a
* contract which another class-id program may implement.
*

IDENTIFICATION DIVISION.
INTERFACE-ID. "InterestHome" INHERITS "javax.ejb.EJBHome".

*
* Only a create method is necessary to define for this interface,
* allowing the client to create the Enterprise JavaBean.
*

METHOD-ID. "create" THROWS "java.rmi.RemoteException",
"javax.ejb.CreateException".
DATA DIVISION.
LINKAGE SECTION.
    77 result OBJECT REFERENCE "com.web_tomorrow.interest.Interest".
PROCEDURE DIVISION GIVING result.
END-METHOD.

END-INTERFACE.
```

EJB Class

Finally there is the Bean class. This is the only one that does any real work in this simple example.

```
* This example is coded for the JBoss EJB Container.  The naming
* provider and setup information may differ for other EJB Containers.
* The logic flow remains the same. (http://www.jboss.org)
*
* This example requires J2EE.  JDK 1.2+ plus JBoss is sufficient.
*
* The CLASSPATH to compile must include \jboss\lib\ext\ejb.jar in
* addition to the percobol.jar required for any Cobol program.  This
* is the javax.ejb.* package; its location may vary in other EJB
* vendors.
*
* Set the package name; this could be done from the command line
* using the '-package name' directive instead.  Setting a package
* is generally not necessary for an EJB, but it's good practice.

$SET PACKAGE com.web_tomorrow.interest

*
* This class-id program is the main Enterprise JavaBean.  It contains
* the business logic which is exposed to the outside world and enough
* hooks to allow the EJB Container to control the bean.
**
* A SessionBean is a particular type of EJB; this is the type most
* corresponding to a CICS transaction and the type most Cobol programs
* will implement.
*
* A SessionBean is a logic bean; an EntityBean is a data bean.

IDENTIFICATION DIVISION.
CLASS-ID. "InterestBean" IMPLEMENTS "javax.ejb.SessionBean".

* IDENTIFICATION DIVISION. is optional for a method-id.

METHOD-ID. "calculateCompoundInterest".

DATA DIVISION.
LINKAGE SECTION.
    01 principle COMP-2.
    01 rate      COMP-2.
    01 periods   COMP-2.
    01 result    COMP-2.

*
* This is the business method exposed to any EJB client.
*
* This example uses COMP-2, which is double precision floating
* point, rather than the more traditional Cobol types of fixed
* point arithmetic which would really be more appropriate.  The
* COMP-2 is used in this case so the signature of the method
* would exactly match the original Java program.  The BY VALUE
* forces normal Java types rather than Cobol types; this is
* the best practice for use in classes which may interact with
* other Java programs.  To keep exact datatypes, it's recommended
* to use PIC X(n) variables and move the values to the desired
* types; this translates to the Java String type which would
* both keep precision and is easy to use from Java as well as
* Cobol.
*
```

```
PROCEDURE DIVISION USING BY VALUE principle, rate, periods GIVING result.
MAIN-PARAGRAPH.

* The DISPLAY UPON SYSOUT goes to the main log file of the EJB Container.
* This may just be printed on the EJB Container sysout.  It will _not_
* be visible to the client.  Only the GIVING result piece will be
* returned and made visible to the client.

    DISPLAY "Someone called 'calculateCompoundInterest' in PERCobol!"
        UPON SYSOUT
    DISPLAY " principle=" principle
        UPON SYSOUT
    DISPLAY " rate     =" rate
        UPON SYSOUT
    DISPLAY " periods  =" periods
        UPON SYSOUT

* There are other ways of computing this, but this example demonstrates
* the similarities between the Java and Cobol implementations of
* EJBs.

    COMPUTE result = principle * ((1+rate) ** periods) - principle

    DISPLAY " result   =" result
        UPON SYSOUT
    DISPLAY " formula is principle*((1+rate)**periods)-principle"
        UPON SYSOUT
    .

END-METHOD.

* Many EJB's can just copy the remaining code into their code.
*
* All remaining methods in this class are structural methods,
* necessary not for the business logic but rather for the EJB
* Container to be able to control this bean.
*
* Add in some DISPLAY UPON SYSOUT's to the procedure division
* of the methods in order to gain some feel over when these
* methods are called.
*
* We are given the session context, but we don't need it in this
* program so we ignore it.  This method is required to fulfill
* the interface of javax.ejb.SessionBean.
*

METHOD-ID. "setSessionContext".
DATA DIVISION.
LINKAGE SECTION.
    01 sc OBJECT REFERENCE "javax.ejb.SessionContext".

PROCEDURE DIVISION USING BY VALUE sc.
END-METHOD.

* The following methods are required to fulfill the interface
* of javax.ejb.SessionBean.  We don't need to do anything special,
* though, so we just create the method without a body.  When
* the method has no parameters and no result, this is the barest
* possible method definition.

METHOD-ID. "ejbCreate". END-METHOD.
METHOD-ID. "ejbRemove". END-METHOD.
METHOD-ID. "ejbActivate". END-METHOD.
METHOD-ID. "ejbPassivate". END-METHOD.

END-CLASS.
```

Notice that most of the methods are empty; they have to exist because they're specified by the SessionBean interface, but they don't need to do anything in this case.

This example uses COMP-2 for data storage. This is really an inappropriate type because floating point is inherently inexact for a currency usage such as this example, but this code is done to parallel the original Java code which used the equivalent data type. This allows the Java client to call the PERCobol server, or the PERCobol client to call the Java server.

All the PROCEDURE DIVISION USING and INVOKING USING for EJB Components are done BY VALUE rather than BY REFERENCE or BY CONTENT. This is important to ensure that interoperability with Java is preserved and that the parameters may be passed correctly across the network. The interfaces generated with INTERFACE-ID are Java source; these Java source files are readable by a Java programmer and are sufficient to allow a Java programmer to access the functionality.

If you haven't already done so, you should create and compile these .cob files in the directory \percobol\samples\ejb\server.

Deployment Descriptor

With the .class files ready, it's time to create the deployment descriptor. The deployment descriptor is identical for Java or Cobol. In current versions of the EJB specification, it is an XML description of the EJB Component, its bean and interfaces. Follow the directions provided by the EJB Container vendor for this information.

The .xml file will be placed in the META-INF subdirectory of the EJB Component. In this case, it's \percobol\samples\ejb\server\META-INF\ejb-jar.xml.

```
<?xml version="1.0" encoding="Cp1252"?>

 <ejb-jar>
   <description>JBoss test application </description>
   <display-name>Test</display-name>
   <enterprise-beans>
     <session>
       <ejb-name>Interest</ejb-name>
       <home>com.web_tomorrow.interest.InterestHome</home>
       <remote>com.web_tomorrow.interest.Interest</remote>
       <ejb-class>com.web_tomorrow.interest.InterestBean</ejb-class>
       <session-type>Stateless</session-type>
       <transaction-type>Bean</transaction-type>
     </session>
   </enterprise-beans>
 </ejb-jar>
```

JBoss can use some additional information in a jboss.xml file. In this case, it's \percobol\samples\ejb\server\META-INF\jboss.xml.

```
<jboss>
   <secure>false</secure>
   <container-configurations />
   <resource-managers />
   <enterprise-beans>
      <session>
        <ejb-name>Interest</ejb-name>
        <jndi-name>interest/Interest</jndi-name>
        <configuration-name></configuration-name>
      </session>
   </enterprise-beans>
 </jboss>
```

Packaging and Deploying the EJB Component

EJB Client

An EJB Client can be any number of types of program. It can be an application, a server, a servlet, an applet, a JSP page, or another EJB Component. EJB Components communicate with each other as if they were clients; by loosely connecting components in this manner, the EJB Components are free to be fully controlled by the EJB Container.

The following is a sample test client.

```
* Enterprise JavaBean Client Sample
*
* for the JBOSS EJB Container.
*
* The original Java version of this EJB client is available at:
* http://www.jboss.org/documentation/jboss_win32_5.html
*
* This simple application tests the `Interest' Enterprise JavaBean which is
* implemented in the package `com.web_tomorrow.interest'. For this to work,
the
* Bean must be deployed on an EJB server.
*
* IMPORTANT If you want to test this in a real client-server
* configuration, this class goes on the client; the URL of the naming provider
* specifed in the class must be changed from `localhost:1099' to the URL of
the
* naming service on the server.
*
* Note: In Cobol, this program may also be used as a servlet if compiling
* with the -servlet flag, or using the com.legacyj.run.servlet as the
* name of the servlet with the servlet parameter 'servlet' pointing
* to interest_client.  The only differences for servlets would be
* to add some additional DISPLAYs of HTML elements surrounding the
* text, such as DISPLAY "<HTML><HEAD><TITLE>EJB Client</TITLE></HEAD>"...
*
* The CLASSPATH must include ejb.jar (found in \jboss\lib\ext\ejb.jar
* and other J2EE implementations), the server-side classes (only
* the pieces actually referenced), and percobol.jar (already setup)
* for compilation.

IDENTIFICATION DIVISION.
PROGRAM-ID. interest-client.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
REPOSITORY.

* All classes are defined in either the CONFIGURATION SECTION/
* ENVIRONMENT DIVISION/REPOSITORY, or in the DATA DIVISION/CLASS-CONTROL.
* The format of either is identical.
*
  CLASS InitialContext IS "javax.naming.InitialContext"
  CLASS InterestHome IS "com.web_tomorrow.interest.InterestHome"
  CLASS InterestClass IS "com.web_tomorrow.interest.Interest"
```

```
  CLASS PortableRemoteObject IS "javax.rmi.PortableRemoteObject"
  .

DATA DIVISION.

* In the Java original, the variable declarations are intermixed with
* the procedural code.
*

WORKING-STORAGE SECTION.

* Declare the variables actually used by the program.  The COMPOUND-INTEREST
* was originally done in a different format for Java, but the INVOKE GIVING
* may reference a traditional Cobol variable, allowing a more appropriate
* display format to be chosen, such as the numeric-edited item below.

       77 compoundInterest PIC $$$$,$$$,$$$.99.

       77 jndiContext OBJECT REFERENCE InitialContext.
       77 home OBJECT REFERENCE InterestHome.
       77 ref OBJECT REFERENCE.
       77 interest OBJECT REFERENCE InterestClass.

PROCEDURE DIVISION.

* This method does all the work. It creates an instance of the Interest EJB
* on the EJB server, and calls its `calculateCompoundInterest' method, then
* prints the result of the calculation.

MAIN.
       SET ENVIRONMENT "java.naming.factory.initial"
           TO          "org.jnp.interfaces.NamingContextFactory"

* Set up the naming provider; this may not always be necessary, depending
* on how your Java system is configured.

       SET ENVIRONMENT "java.naming.provider.url"
           TO          "localhost:1099"

* Java Note:
* Enclosing the whole process in a single `try' block is not an ideal way
* to do exception handling, but I don't want to clutter the program up
* with catch blocks

* Cobol Note:
* No TRY block is necessary for this.  Rather, each INVOKE is automatically
* safe, catching its own exceptions.  If you want notification when an
* invoke fails, code the individual ON EXCEPTION clause for the INVOKE.

* Get a naming context
       INVOKE InitialContext GIVING jndiContext
       ON EXCEPTION
           DISPLAY "Could not create InitialContext."
           GOBACK
       END-INVOKE

       DISPLAY "Got context"

* Get a reference to the Interest Bean
       INVOKE jndiContext "lookup"
           USING BY VALUE "interest/Interest" GIVING ref
       ON EXCEPTION
           DISPLAY "Could not lookup interest/Interest"
           GOBACK
       END-INVOKE

* Note that if you did not use jboss.xml to overwrite JNDI naming
* the object will be available under "Interest" its ejb-name
* INVOKE jndiContext "lookup" USING BY VALUE "Interest" GIVING ref

       DISPLAY "Got reference"
```

```
* Get a reference from this to the Bean's Home interface

       INVOKE PortableRemoteObject "narrow"
           USING BY VALUE ref InterestHome GIVING home
       ON EXCEPTION
           DISPLAY "Could not narrow."
           GOBACK
       END-INVOKE

* Create an Interest object from the Home interface
       INVOKE home "create" GIVING interest
       ON EXCEPTION
           DISPLAY "Could not create home."
           GOBACK
       END-INVOKE

* call the calculateCompoundInterest() method to do the calculation

       INVOKE interest "calculateCompoundInterest"
           USING BY VALUE 1000 0.10 2
           GIVING compoundInterest
       ON EXCEPTION
           DISPLAY "Could not calculateCompoundInterest"
           GOBACK
       END-INVOKE

       DISPLAY "Interest on 1000 units, at 10% per period, "
             & "compounded over 2 periods is:" compoundInterest

       .
```

The SET ENVIRONMENT statements set Java System Properites. Two in particular are used in access from a client; these set the class used as a naming provider and where the EJB Container is located. Combined, these settings give enough information to find the correct EJB Container implementation.

The lookup finds the individual application/bean on the designated EJB Container.

The narrowing and creation create a proxy object on the client which represents the equivalent object on the server. Only methods defined in the external interface are available; but all calls to this returned object are actually performed on the server. All calls to the object are done using the INVOKE corresponding to the external METHOD-ID in the remote interface.

In this example, every INVOKE has an ON EXCEPTION clause which handles errors. Remember, EJB access is done across the network; this means that EJB objects may fail at any call. The EJB Container will often try to transparently correct certain errors on the server side, but if the client's network cord is pulled in the middle of the call, there is no way it can completely successfully. It's best to always do error handling when the object is remote. (Certain EJB Containers can be setup to control multiple machines such that even machine failure may be circumvented.)

The INVOKE of the business method in this case gives its result to a numeric-edited item. The parameters to an INVOKE must be well-matched, but the GIVING acts as a MOVE of the actual data to the GIVING item. The

double precision floating point may be moved to a numeric-edited item, so this operation is allowed.

Client Output

```
Got context
Got reference
Interest on 1000 units, at 10% per period, compounded over 2 periods is:
$210.00
```

Server Output:

```
[Interest] Someone called 'calculateCompoundInterest' in PERCobol!
[Interest]  principle=1000.0
[Interest]  rate     =0.1
[Interest]  periods  =2.0
[Interest]  result   =210.0000000000002
[Interest]  formula is principle*((1+rate)**periods)-principle
```