IBM WebSphere 5.0 Skills Transfer - LAB EXERCISE

# Building a J2EE 1.3 Application with Container-Managed Persistence and Local Interfaces

## What This Exercise is About

The EJB 2.0 specification, included in the J2EE 1.3 specification, contains a considerable number of new features and enhancements over the prior EJB specification. These enhancements, Container-Managed Persistence (CMP) and Local Interfaces, increase application portability and offer an efficient means of access to enterprise beans. WebSphere Studio Application Developer v5.0.1 fully supports the creation of J2EE 1.3 applications and these EJB 2.0 enhancements. J2EE 1.3 applications can be easily moved from the development environment into production on WebSphere Application Server v5.0.1 which also fully supports J2EE 1.3 applications.

## User Requirements

Windows 2000 Professional Service Pack 3 is required for this lab exercise. IBM WebSphere Studio Application Developer v5.0.1 is also required. The lab source files (LabFiles50.zip) must be extracted to the root directory (i.e., C:\). Experience with previous versions of WebSphere Studio Application Developer and the J2EE programming model are also required.

## What You Should Be Able to Do

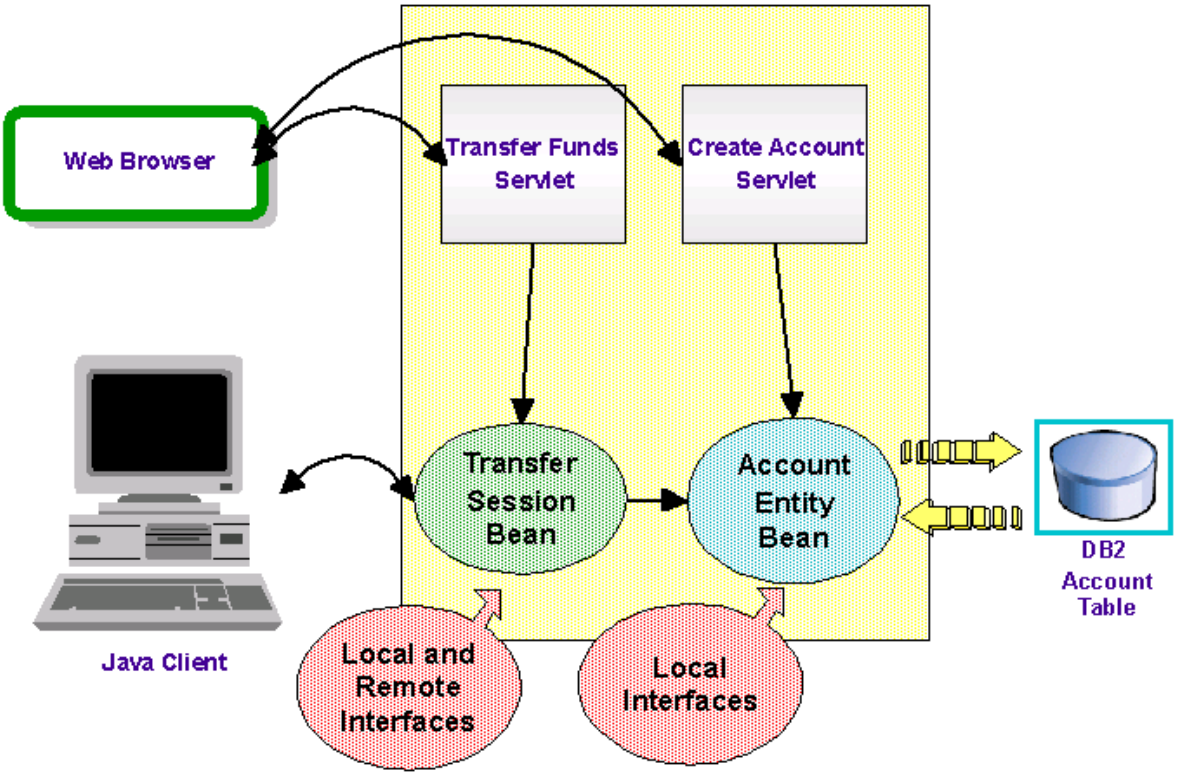
At completion of this lab exercise you should be able to create, with WebSphere Studio Application Developer v5.0, enterprise beans compliant to the EJB 2.0 specification. You should also be able to create local and remote interfaces for session and entity beans. Finally, you should be able to add other modules to your application which access a specific interface in order to call the business logic of your application.

## Introduction

You will begin this exercise by building a small banking application which features CMP and Local Interfaces with WebSphere Studio Application Developer v5.0.1. You will start by creating an entity bean with Container-Managed Persistence and Local Interfaces which represents a bank account.

With the Account bean created, you will create a session enterprise bean which will use the local interface of the entity bean to perform a transfer of funds between accounts. The Transfer session bean will be generated with remote and local interfaces.

With the business logic of your application created, you will next add a web module and an application client module to your J2EE 1.3 application. The web module will contain servlets which use the local interface of the session bean to transfer funds, while the application client will use the remote interface to perform the same transfer operation. With your J2EE 1.3 application complete, you will export out the application as an Enterprise Archive (EAR) file.




## Exercise Instructions

**** NOTE ** Solution instructions are at the end. The solution is provided in the case you do not wish to perform the exercise steps, or are unable to complete the lab and want to see the final solution. If you intend to go through the lab, start at Part One -- assuming you have met the requirements in the section "User Requirements" stated above.**

### Part One: Starting up the Development Environment

__1. Start WebSphere Studio Application Developer.

__ a. Select the menu **Start > Programs > IBM WebSphere Studio** and select **Application Developer 5.0**.

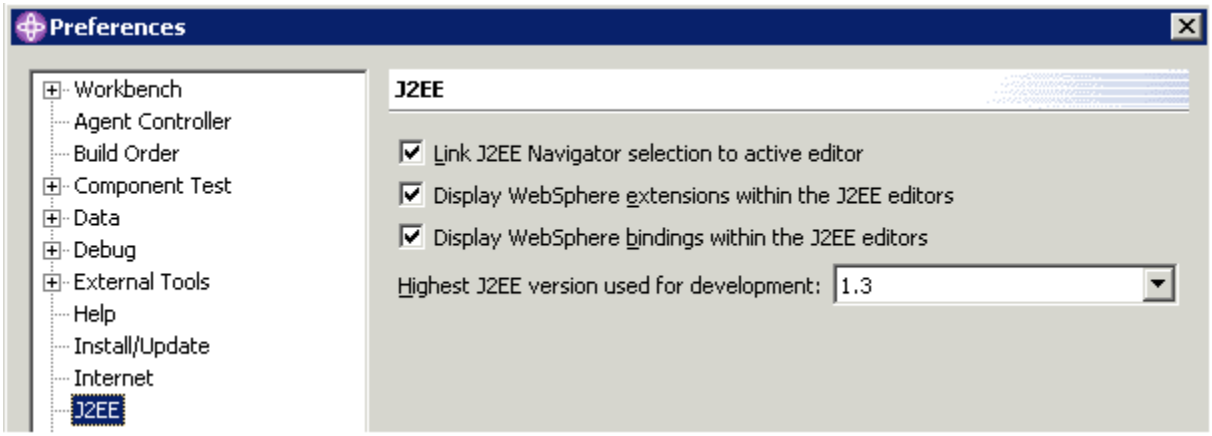
__ b. A dialog box will be displayed allowing you to select the location where you would like the workspace directory to be stored. Enter **c:\labfiles50\cmpbuildlab\workspace** for the location and leave the checkbox next to "Use this workspace as the default workspace" unchecked. Click **OK**. Application Developer will start with an empty workspace. An empty workspace will leave your existing workspace untouched and help avoid name conflicts between what you may already have in your workspace and what you will be creating in this lab.

__2. WebSphere Studio Application Developer supports J2EE 1.2 and J2EE 1.3 applications. You can work with existing applications at these specification levels or create new artifacts at these levels as well. The level at which you would like to work can be set in the J2EE Preferences. Verify the highest level is set to J2EE 1.3.

__ a. Select **Window > Preferences** to open the Preferences window.

__ b. There are numerous settings you can change to customize your development environment and set default values. Verify the highest level of J2EE is selected. Select **J2EE** from the list.

__ c. With the preference set, you will now be prompted for the version of J2EE to use when creating different J2EE components. Verify **1.3** is selected. If it is set to 1.2, change it to 1.3 and click **Apply**.



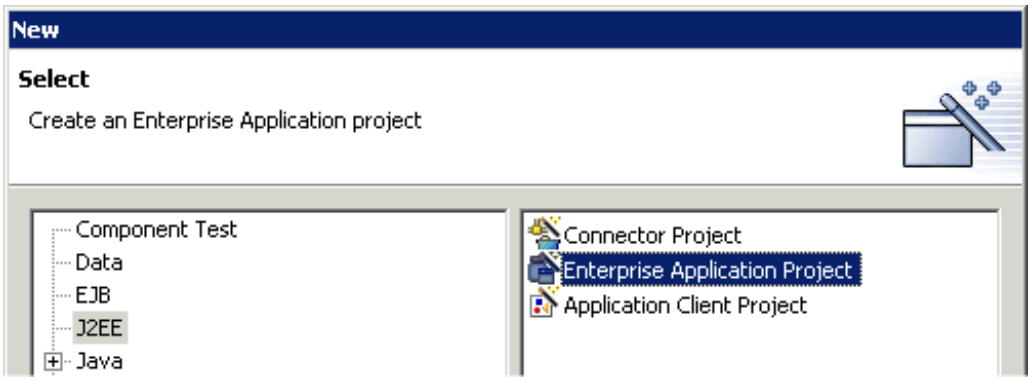

__ d. Click **OK**.

## Part Two: Build the Application

__1. You will first start building the banking application by creating an EAR file and an EJB Module.

__ a. Select **File > New > Other...**.

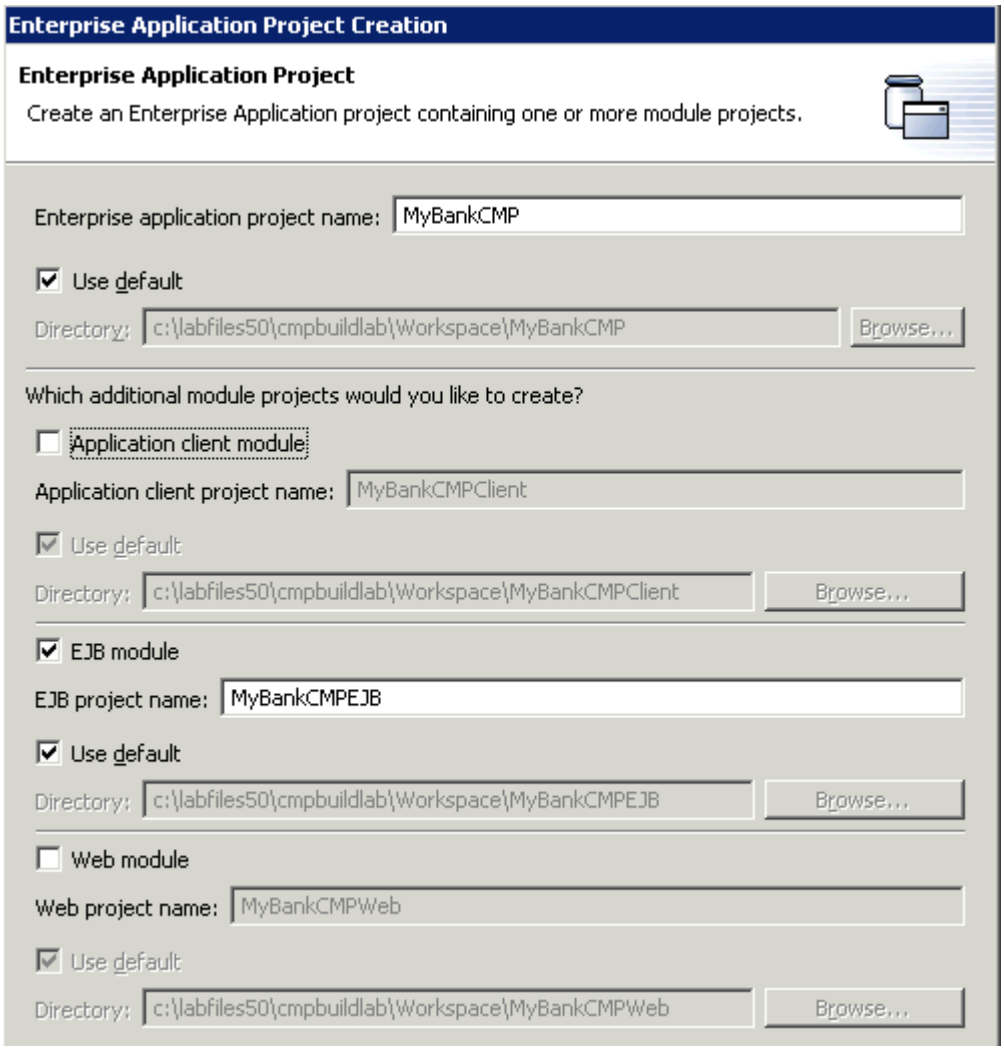
__ b. Select **J2EE** and **Enterprise Application Project**. Select **Next**.




__ c. As indicated in Part One, Application Developer 5.0.1 supports both the J2EE 1.2 and 1.3 specifications. In this window you can select which specification you would like the application to be created at. It is important that you choose the correct level. If you choose to create a J2EE 1.2 Enterprise Application, you will only be able to add J2EE 1.2 modules to the application. Your banking application will be compliant with J2EE 1.3. Accept the default setting and click **Next**.

__ d. For the **Enterprise application project name**, enter **MyBankCMP**.

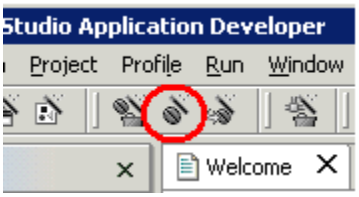
__ e. Uncheck the boxes for **Application Client Module**, and **Web Module**. These modules will be added later. It is best not to create modules until you are ready to add content. Click **Finish**.



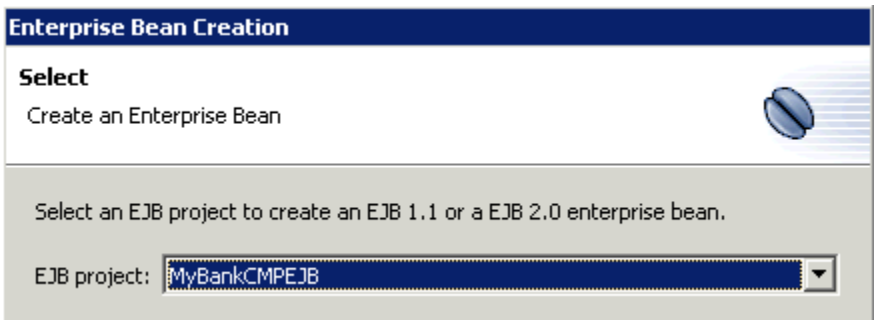

__ f. The J2EE perspective should display the MyBankCMP Enterprise Application within the J2EE Hierarchy view.

__2. With the modules created, start building your application by creating the Account entity bean which will be used within your business logic to represent a particular type of bank account.

__ a. From the toolbar, click the **Create an Enterprise Bean** button. You can also select **File > New > Enterprise Bean** to create an enterprise bean.



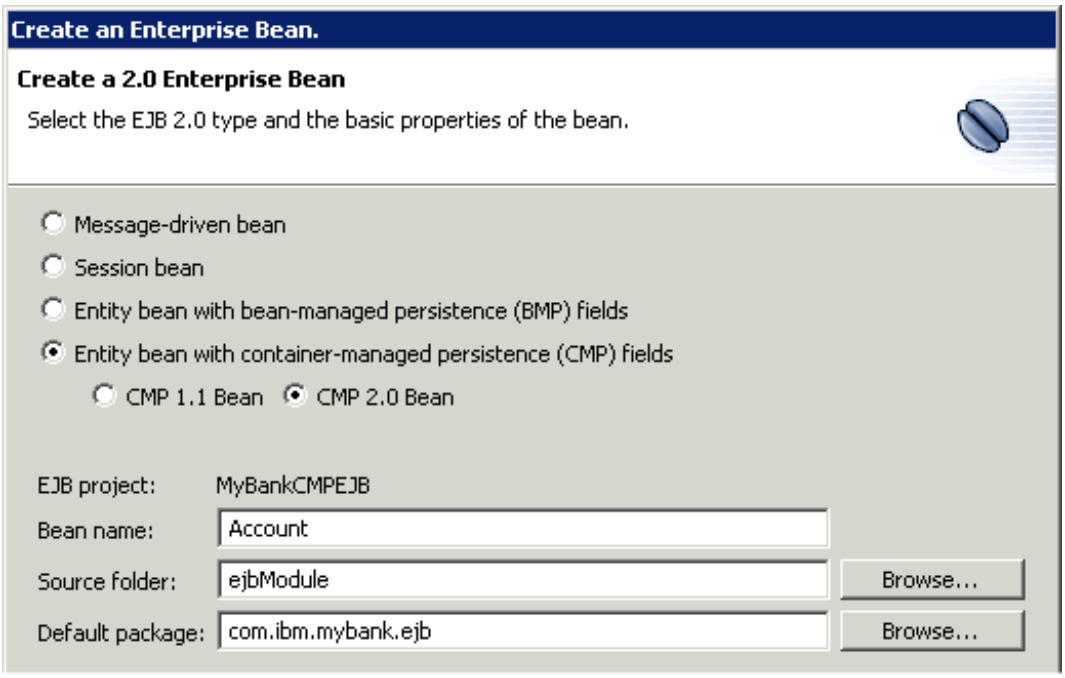

__ b. Select **MyBankCMPEJB** from the drop-down box. Since this EJB Module is a EJB 2.0 module, the entity bean you will create will have attributes declared in compliance with the EJB 2.0 specification. You will also have the option to create Local Interfaces for the entity bean. Click **Next**.



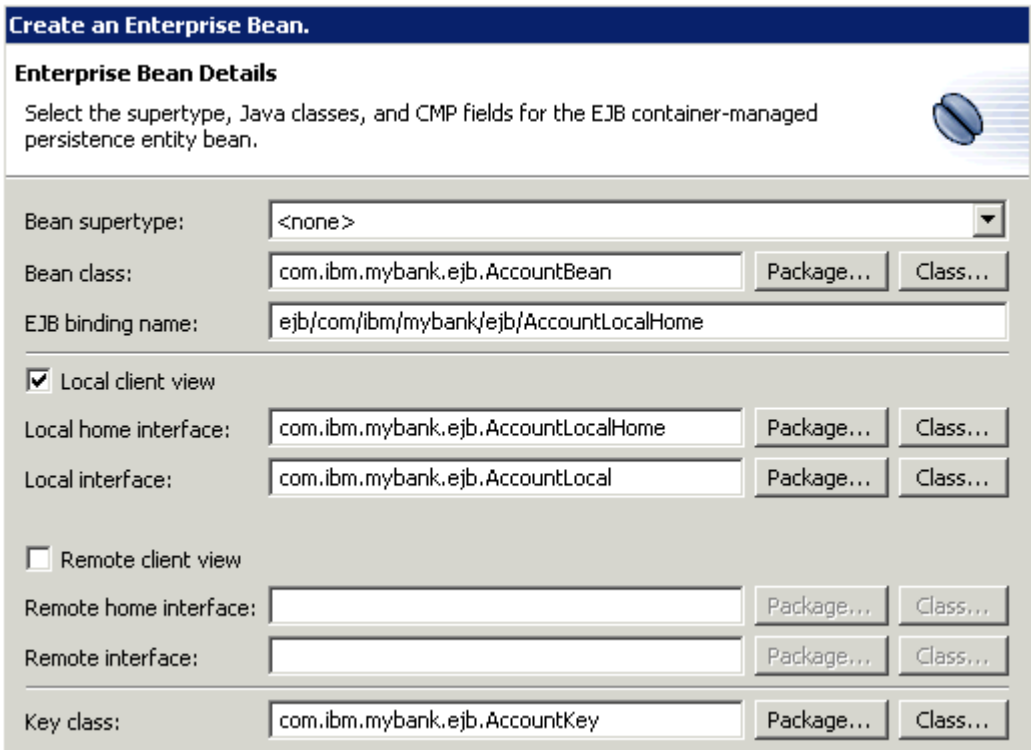

__ c. Click the **Entity bean with container-managed persistence (CMP) fields** radio button.

__ d. For the Bean name, enter **Account**. Be sure Account is capitalized. The bean name will be used in the file names (AccountBean.java, AccountLocal.java, and AccountLocalHome.java) and will need to be capitalized as required by the EJB 2.0 specification.

__ e. In the Default Package field, enter **com.ibm.mybank.ejb** and select **Next**.



__ f. For your application, the account bean will only be accessible through local interfaces, although it can be accessible through both a local and remote interface. This is an important design decision as it will limit access to the Account bean from processes only within the current JVM. The local interface, however, does offer improved performance. Ensure the **Remote client view** checkbox is not selected and the **Local client view** checkbox is selected.




__ g. Entity beans represent business objects and can contain attributes which offer more detailed information. Click **Add...** to the right of the CMP attributes field.

__ h. In the Create CMP Attribute window, enter **accountNumber** for the Name and select **long** for the Type. Select the checkbox **Key field** and click **Apply**.

> **Note:** Please choose the **long** attribute, not java.lang.Long. Choosing java.lang.Long will result in errors.

__ i. Repeat the previous step with a Name of **accountType** and select **int** for the type. Do not select the checkbox for the Key field. Click **Apply**.

__ j. Repeat the previous step again with a Name of **balance** and select **float** for the type. Click **Apply**. Click **Close** to close the window. You should have 3 attributes defined for your Account bean.

__ k. Click **Next** to move to the EJB Java Class Details window.

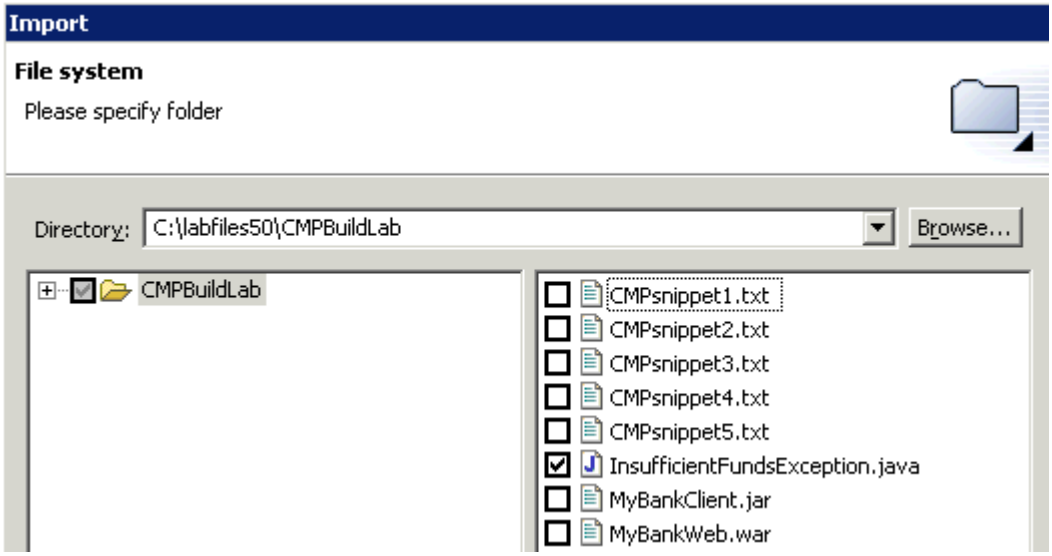
__ l. Click **Finish**. The Account entity bean will be created with the local interfaces. The information describing the entity bean will also be added to the ejb-jar.xml deployment descriptor in the EJB module.

__3. Add a custom exception used within the business logic (the business logic will be added in the next step).

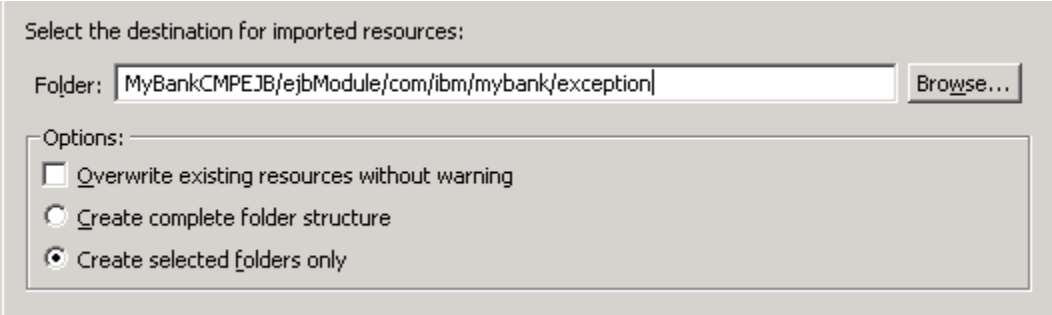
__ a. Import the exception file into the package by selecting **File > Import...** Select **File System** and then **Next**.

__ b. Click **Browse...** and navigate to the directory to **C:\LabFiles50\CMPBuildLab**. With this directory located, click **OK**.

__ c. On the right, select the checkbox for **InsufficientFundsException.java**.



__ d. Use the **Browse...** button to the right of the Folder field. Navigate to **MyBankCMPEJB/ejbModule/com/ibm/mybank.** Append **exception** to the path.




__ e. Click **Finish**. The InsufficientFundsException file will be placed in the EJB Module.

__4. Add support for working with the primary key field of the Account bean.

__ a. Change to the **J2EE Navigator** view if it is not visible and expand **MyBankCMPEJB\ejbModule\com.ibm.mybank.ejb**. Open **AccountBean.java** by double clicking on it.

__ b. Browse through the file or the Outline view. Notice no attributes are explicitly declared. The attributes for the entity bean are actually defined in the deployment descriptor and are accessible with getter and setter methods. This is defined in the EJB 2.0 specification and part of the abstract persistence schema.

__ c. Replace the **ejbCreate** method in the Account Bean (AccountBean.java) with the following code to set the attributes of the Account bean. If you kept the original method and overloaded the ejbCreate method, there would be multiple ways of creating an account with the possibility of not having attributes initialized. If there would be any uncertainty as to whether or not attributes are set, code to handle the null or empty attribute situations would have to be developed. To avoid these situations you will replace the method. There are certain times, however, when you may want to overload the method for performance reasons. (Note: To save time, copy the code from C:\LabFiles50\CMPBuildLab\CMPsnippet1.txt.)

```
public com.ibm.mybank.ejb.AccountKey ejbCreate(
long accountNumber,
int accountType,
float initialBalance)
throws javax.ejb.CreateException {

      setAccountNumber((int)accountNumber);

      setAccountType(accountType);

      setBalance(initialBalance);
```

```
        return null;
    }
```

__ d. Right-click on the editor window containing the AccountBean.java file and select **Format** from the context menu. The code you just added will be formatted based on preference settings.

__ e. Replace the **ejbPostCreate** method in the Account Bean (AccountBean.java) with the following code. Each ejbCreate method must have a corresponding ejbPostCreate method with the same parameters. (Note: To save time, copy the code from C:\LabFiles50\CMPBuildLab\CMPsnippet2.txt.)

```
public void ejbPostCreate(
long accountNumber,
int accountType,
float initialBalance) {
}
```

__5. Add business logic to the Account Bean (AccountBean.java) for adding or subtracting funds from an account.

__ a. Add the following methods to the bottom of the class, right before the last closing bracket. These will allow for adding or subtracting funds from an account. Notice the methods are accessing the Account attributes using the getter and setter methods rather than literal objects. This is part of the CMP model and EJB 2.0 specification. (Note: To save time, copy the code from C:\LabFiles50\CMPBuildLab\CMPsnippet3.txt.)

```
public float subtract(float amount) throws
      InsufficientFundsException {
      if (getBalance() < amount) {
      throw new InsufficientFundsException("The Account has
            Insufficient Funds.");
      }
      setBalance(getBalance() - amount);
      return getBalance();
}

public float add(float amount) {
      setBalance(getBalance() + amount));
      return getBalance();
}
```

**Note:** You will notice some light bulbs appear in your workbench as well as your tasks view. We will resolve this in a subsequent step. Basically, the java editor is stating that it has no reference to methods/exceptions we have added from the previous step. By using the Source > Organize Imports (in step 5b), we will be adding the correct import statements which will correct these errors.

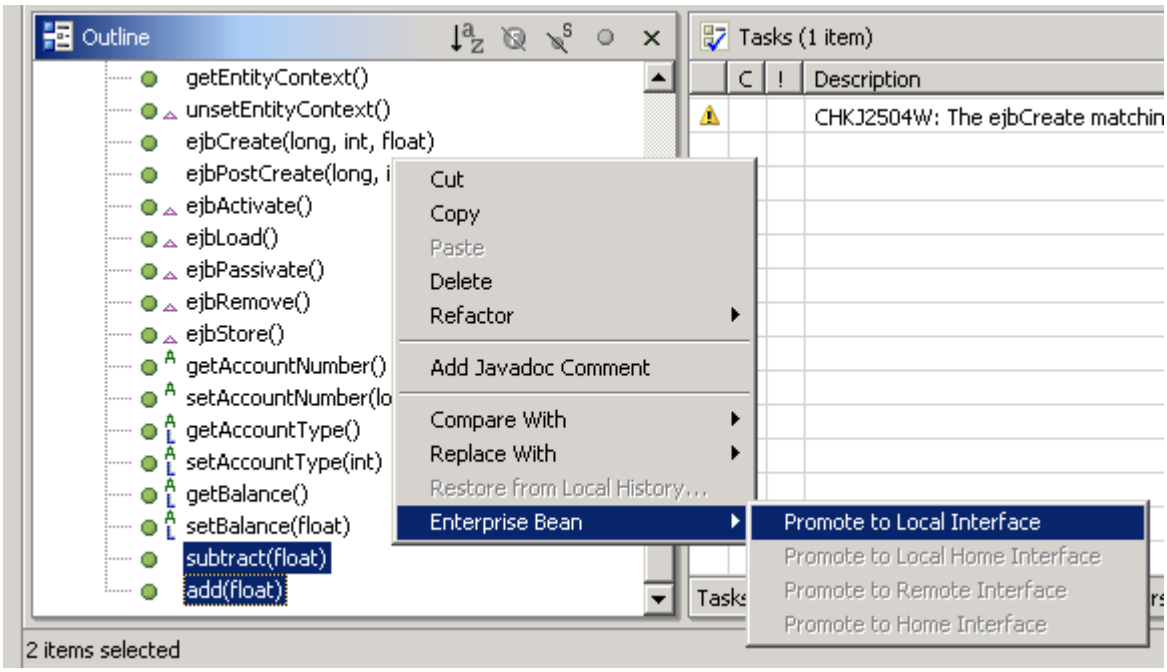
__ b. Before continuing, you will need to add the import statement for the custom exception which is thrown by the subtract method you just added. Right-click within AccountBean.java and select **Source > Organize Imports**. Verify the exception for InsufficientFundsException was added to the top of the file.

```
import com.ibm.mybank.exception.InsufficientFundsException;
```

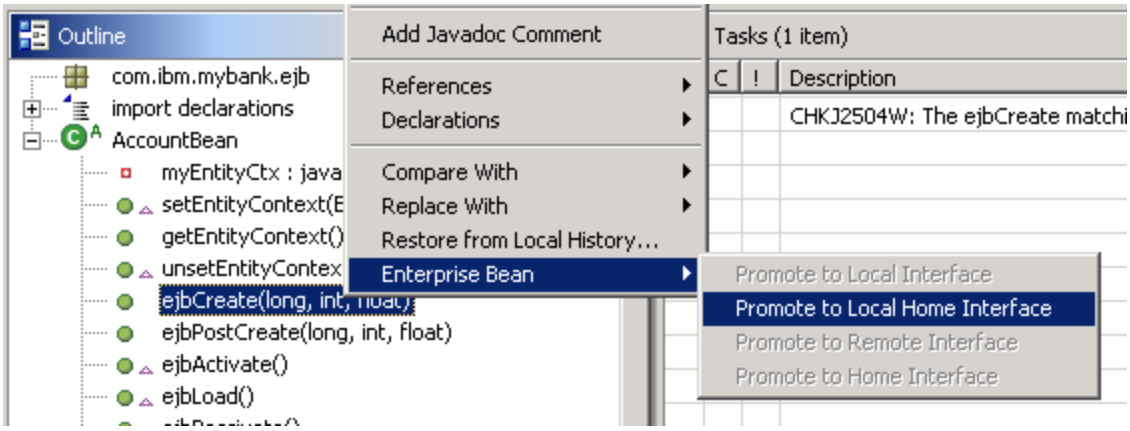
__ c. Save the file.

> **Note:** Ignore any messages listed in the Task View.

__ d. These methods need to be made available in the local interface where they can then be called from outside the bean. In the Outline view, select the methods **subtract** and **add** and from the context menu, select **Enterprise bean > Promote to Local Interface**.



__ e. Since you replaced the ejbCreate method with a method with more parameters, we need to update the create method in the Local Home interface to match this method declaration. In the **Outline** view, select the method **ejbCreate(long, int, float)** and from the context menu, select **Enterprise Bean > Promote to Local Home Interface**.




__ f. Close the **AccountBean.java** file.

__ g. Open the file **AccountLocalHome.java**.

__ h. Verify that the code promoted to **AccountLocalHome.java** is the same as the following:

```
public com.ibm.mybank.ejb.AccountLocal create(
long accountNumber,
int accountType,
float initialBalance) throws javax.ejb.CreateException;
```

__ i. Since you replaced the ejbCreate(long AccountNumber) method in the bean class, you will need to remove the create(long) method. In the Outline view, right click **create(long)** and from the menu select **delete**. Then click **Yes** on the delete confirmation dialog.

__ j. Save and close the file. The errors should go away.

__6. Add the Transfer session enterprise bean.

__ a. Click the **Create an Enterprise Bean** icon from the toolbar.

__ b. The Transfer bean you will create also be a 2.0 enterprise bean. Select MyBankCMPEJB for the EJB project and click **Next**.

__ c. Click the **Session bean** radio button and enter **Transfer** for the Bean name. Be sure Transfer is capitalized. The bean name will be used in the file names (TransferBean.java, TransferLocal.java, TransferLocalHome.java, TransferHome,

and Transfer.java) and will need to be capitalized as required by the EJB 2.0 specification.

__ d. Ensure the Default package is **com.ibm.mybank.ejb** and select **Next**.

__ e. The session bean will be of a stateless type. Ensure the **Stateless** radio button is selected.

__ f. The Transfer session bean should be accessible via remote and local interfaces. This design decision allows for the session bean to be accessed from within the current JVM with optimal performance as well as from outside the current JVM in a distributed implementation. Check the **Local client view** checkbox and ensure that the **Remote client view** checkbox is checked. Click **Next**.

__ g. On the EJB Java Class details page, click **Finish**. The Transfer session bean will be created with local and remote interfaces and the bean information will be added to the ejb-jar.xml deployment descriptor.

__7. Add the business logic to the Transfer bean.

__ a. Switch to the **J2EE Perspective** if it isn't already open and switch to the **J2EE Navigator** view.

__ b. Expand **MyBankCMPEJB\ejbmodule\com.ibm.mybank.ejb**. Open the **TransferBean.java** file.

__ c. Add a private variable to the class. This member will be used for transferring funds between accounts. Place the code after the declaration of the TransferBean class (right after the opening bracket { ).

```
private AccountLocalHome accountHome = null;
```

__ d. The accountHome member will need to be initialized with a handle to the Account bean. Add the following code to the **ejbCreate** method. To position yourself at the method go to the outline view and click on the ejbCreate method. Make sure you copy your code in between the brackets { } of the method. (Note: To save time, copy the code from C:\LabFiles50\CMPBuildLab\CMPsnippet4.txt.)

```
try {
        accountHome = getAccountHome();
}
catch (Exception e) {
   e.printStackTrace();
   throw new EJBException("Error getting accountHome: " +
       e.getMessage());
```

```
}
```

__ e. Add the following methods to the bottom of the class, right before the last closing bracket. These methods will be used to transfer funds between accounts. (Note: To save time, copy the code from C:\LabFiles50\CMPBuildLab\CMPsnippet5.txt.)

```
public float getBalance(int acctId) throws FinderException,
EJBException {
AccountKey key = new AccountKey(acctId);
AccountLocal fromAccount;
      try {
            try {
                  fromAccount=accountHome.findByPrimaryKey(key);
            }
            catch (FinderException ex) {
                  throw new FinderException(
                              "Account " + acctId
                                    + " does not exist.");
            }

            return fromAccount.getBalance();
      } catch (Exception r) {
            throw new EJBException();
      }
}//getBalance

public void transferFunds(int fromAcctId, int toAcctId, float
      amount)
            throws EJBException, InsufficientFundsException,
                  FinderException
      {

      AccountKey fromKey = new AccountKey(fromAcctId);
      AccountKey toKey = new AccountKey(toAcctId);
      AccountLocal fromAccount, toAccount;
      try {
            try {
            fromAccount = accountHome.findByPrimaryKey(fromKey);
            } catch (FinderException ex) {
                  throw new FinderException(
                        "Account " + fromAcctId
                              + " does not exist.");
            }
            try {
                  toAccount=accountHome.findByPrimaryKey(toKey);
            } catch (FinderException ex) {
                  throw new FinderException(
                        "Account " + toAcctId
                              + " does not exist.");
            }
            try {
                  toAccount.add(amount);
                  fromAccount.subtract(amount);
            } catch (InsufficientFundsException ex) {
                  mySessionCtx.setRollbackOnly();
                        throw new InsufficientFundsException(
                  "Insufficient Fund in " + fromAcctId);
            }
      } catch (Exception r) {
            throw new EJBException();
```

```
        }
}//transferFunds

private AccountLocalHome getAccountHome() throws RemoteException
{
        try {
                InitialContext initCtx = new InitialContext();
                Object objref =
                        initCtx.lookup("java:comp/env/bank/Account");
                return (AccountLocalHome) objref;
        } catch (NamingException ne) {
                ne.printStackTrace();
                throw new RemoteException(
                "Error looking up AccountHome object: " +
                ne.getMessage());
        }
}
```

In the methods you added to the TransferBean, notice the use of the local interfaces for calling the Account bean.

> **Note:** You will notice some light bulbs appear in your workbench as well as your tasks view. We will resolve this in a subsequent step. Basically, the java editor is stating that it has no reference to methods/exceptions we have added from the this step. By using the Source > Organize Imports (in step 7h), we will be adding the correct import statements which will correct these errors.

__ f. Press **Ctrl+Shift+F** to format the code.

__ g. There are a number of calls to dependent packages that are referenced within the Transfer Bean. To resolve those references, right-click in TransferBean.java and select **Source > Organize Imports**. Verify the following import statements were added to the file.

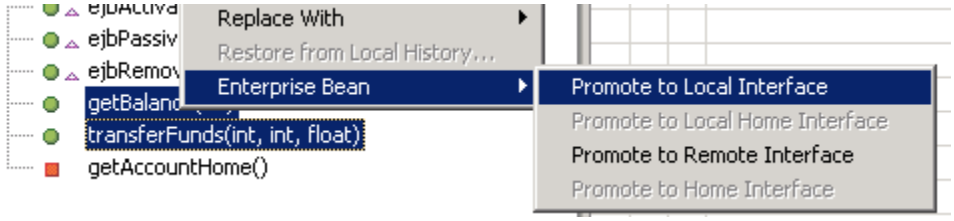
```
import java.rmi.RemoteException;

import javax.ejb.EJBException;
import javax.ejb.FinderException;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import com.ibm.mybank.exception.InsufficientFundsException;
```

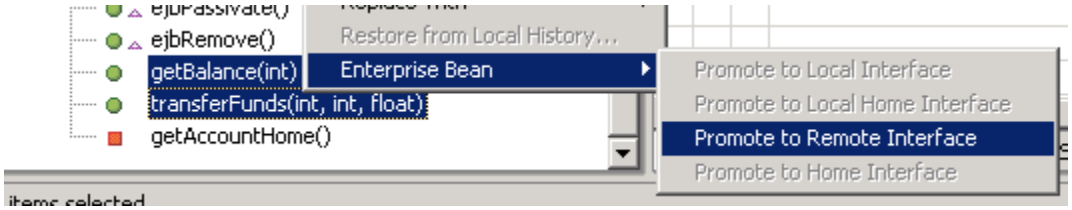
__ h. Save the TransferBean.java file.

__ i. These methods need to be made available in the local interface where they can then be called from outside the bean. In the Outline view, select the methods **getBalance** and **transferFunds** and right click. From the menu select **Enterprise Bean > Promote to Local Interface**. The methods declarations will be added to the TransferLocal.java file and are available to be called on a particular Transfer instance. These methods can only be called from processes

within the JVM the Transfer bean is running In.




__ j. Select the getBalance and transferFunds methods again and right click. From the menu select **Enterprise Bean > Promote to Remote Interface**. The method declarations will be added to the Transfer.java file and are exposed to processes outside the JVM the Transfer bean is running in.

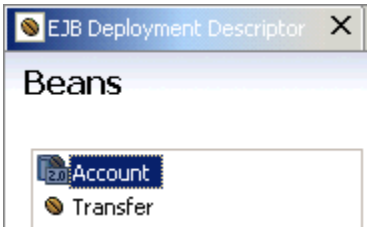



__ k. Close the TransferBean.java file.

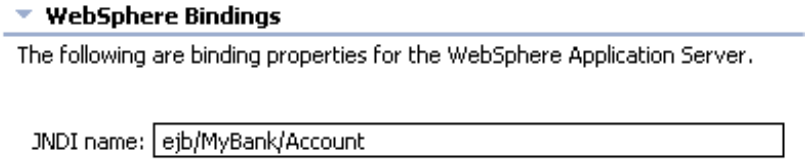
__8. You can specify JNDI names for the Account and Transfer beans. JNDI names offer the ability to separate the implementation class from the name used to call and locate the bean. This separation allows for the underlying implementation to be easily changed as the application changes or per the runtime environment. The JNDI names for the enterprise beans are specified in the EJB Editor. The actual names are stored in the ibm-ejb-jar-bnd.xmi file as they are defined specific to IBM and the WebSphere runtime.

__ a. In the J2EE Navigator view, expand **MyBankCMPEJB** and open **EJB Deployment Descriptor**. The EJB Deployment Descriptor editor will open. By default this editor displays the ejb-jar.xml deployment descriptor and the values stored in the Binding (ibm-ejb-jar-bnd.xmi) and IBM Extension (ibm-ejb-jar-ext.xmi) deployment descriptors specific for WebSphere.

__ b. Select the **Beans** tab and select **Account**.



__ c. On the right side of the editor, under **WebSphere Bindings**, overwrite the existing value and specify **ejb/MyBank/Account** for the JNDI name.

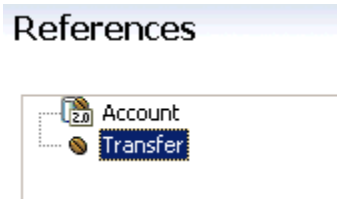



__ d. You can now set the JNDI name for the Transfer bean. Select the Transfer bean and in the JNDI name, enter **ejb/MyBank/Transfer**.

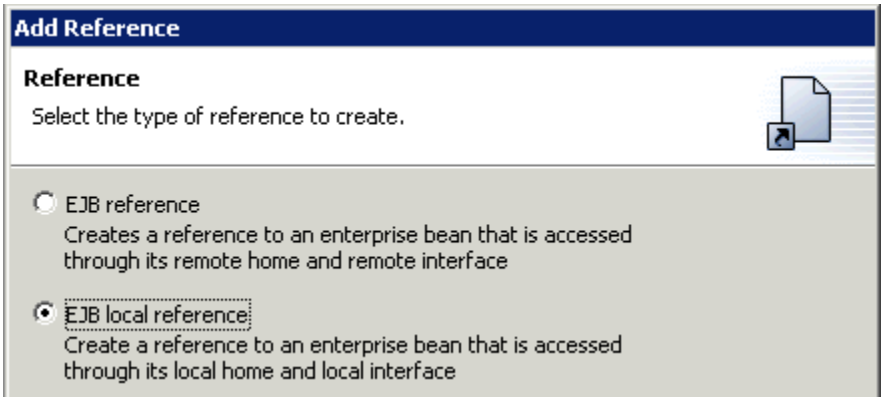
__9. In the ejbCreate method of the Transfer Bean there is a call to getAccount. The method getAccount looks up the account bean object with an EJB Reference. EJB References allow a number of benefits over hard coding in the JNDI name or class name of the enterprise bean. EJB References are specified in the EJB Editor and stored in the ejb-jar.xml deployment descriptor. Having the references defined here, in a way which they can be easily customized per the runtime environment, allows for specific implementation decisions to be postponed until installation time.

__ a. In the EJB Deployment Descriptor editor, click the **References** tab.

__ b. Select **Transfer** from the list of References and click **Add...**.



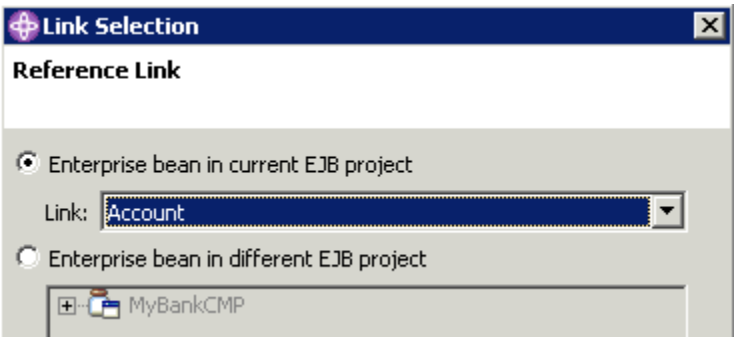

__ c. The Account Bean is only available through Local Interfaces. Click the **EJB local reference** radio button and click **Next**.



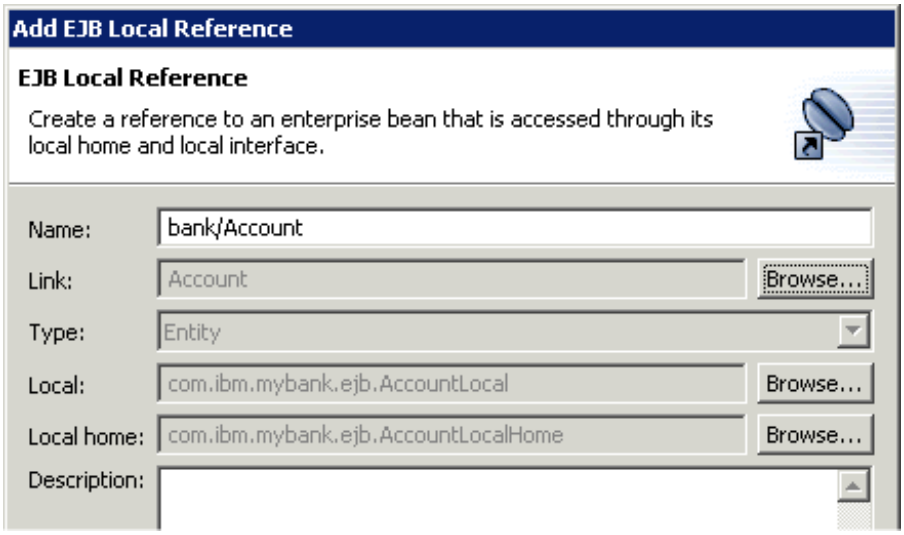
__ d. For the Name, specify **bank/Account**, the name specified in the lookup call to find the Account Bean.

__ e. For the Link, select **Browse...**.

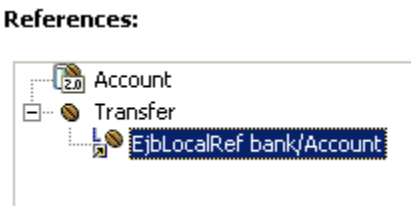
__ f. In the Link pull-down menu, select **Account**.



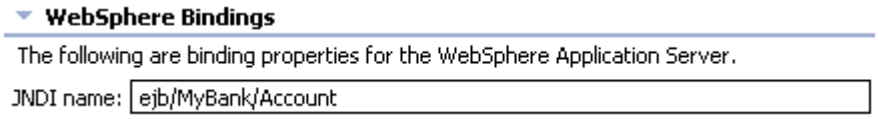

__ g. Click **OK**. The fields for describing the local reference will be populated with the correct values.




__ h. Click **Finish**. The reference will be created. Expand Transfer and select the reference to see the details.



__ i. For the WebSphere runtime environment, you can base this reference off of a JNDI name rather than the name of the implementation class. This offers additional flexibility for configuring your application in the runtime. In the JNDI name field, ensure the JNDI name of the Account bean, **ejb/MyBank/Account**, is specified.




__ j. Close the EJB Deployment Descriptor editor. Click **Yes** to save the changes.

## Part Three: Complete Application Assembly

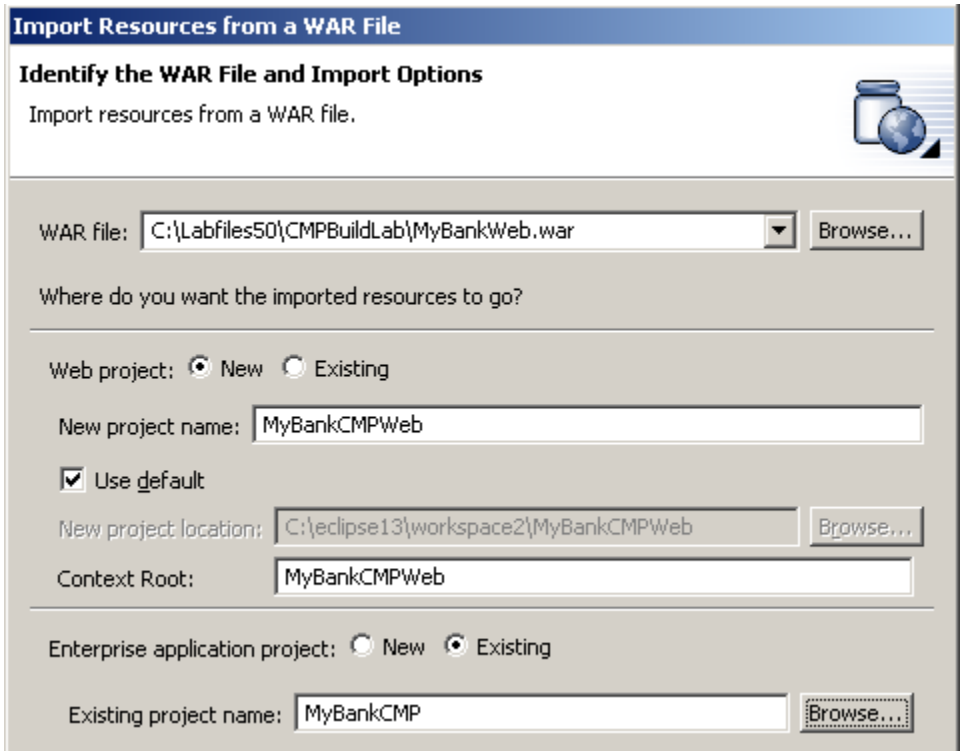
__1. With the business logic creation complete, you can now add a Web interface and a J2EE client application to call the business logic. These modules have been assembled for you and it is just a matter of adding the different modules to your MyBankCMP Application. Start by importing in the Web module

__ a. Select **File > Import...** and select **WAR file** and click **Next**.

__ b. For the WAR file, browse to **C:\LabFiles50\CMPBuildLab\MyBankWeb.war**.

__ c. You will be creating a new Web Project. For the New project name, enter **MyBankCMPWeb**.

__ d. For the EAR Project, select the **Existing** radio button. Use the **Browse...** button and select **MyBankCMP** before selecting **OK**.



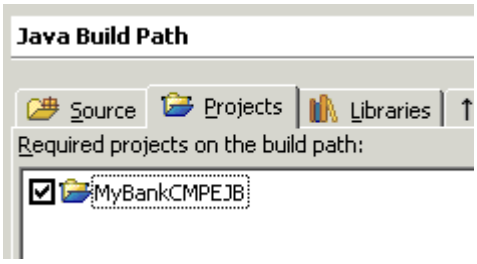

__ e. Click **Finish**.

> **Note**: A number of errors will be displayed in the Task view. You will resolve these errors in the next step.

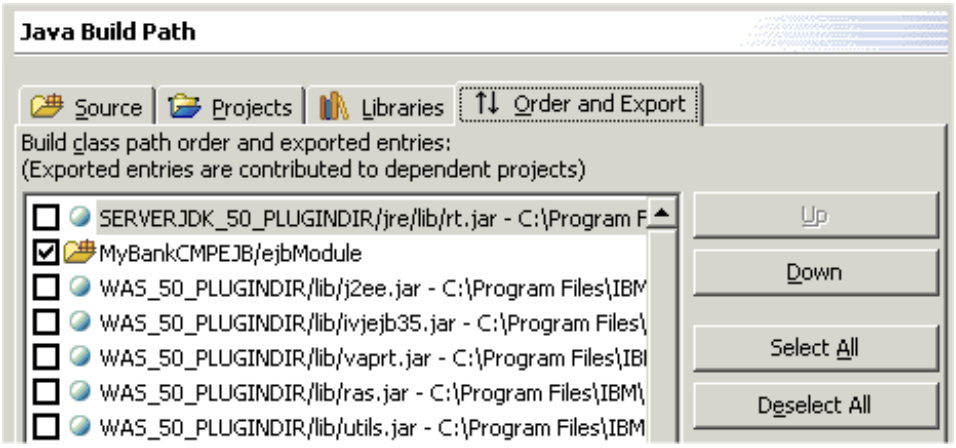
__ f. Within the Web module you imported into Application Developer, there are references (calls) to the enterprise beans in the EJB module. Within the development environment each project is independent of other projects by default. To reference classes in other projects, you must update the Java build path for the project. This is only necessary if you are making changes in Web module and need to compile classes. This not necessary for the application to run in production. Although you will not make any updates to the Web Module, you will update the Java build path for practice. Right-click on MyBankCMPWeb and select **Properties**.

__ g. Select **Java Build Path** and select the **Projects** tab.

__ h. Select the checkbox for **MyBankCMPEJB**.




__ i. Select the **Order and Export** tab. Select the checkbox for **MyBankCMPEJB** again. The second selection is expected. The first selection you made on the projects tab adds the MyBankCMPEJB project to the possible objects in the Java Build Path and the second selection includes MyBankCMPEJB. This allows you to manage all of the build path entries in one list and on one Tab.




__ j. Select **OK** to close the Properties window.

__ k. Optional: Go to **MyBankCMPWeb** project and expand the folders **Java Source\com.ibm.mybank.web**. Open the **TransferFunds.java** file and notice the calls using the **local** interface (transferLocalHome and transferLocal) of the Transfer session bean.

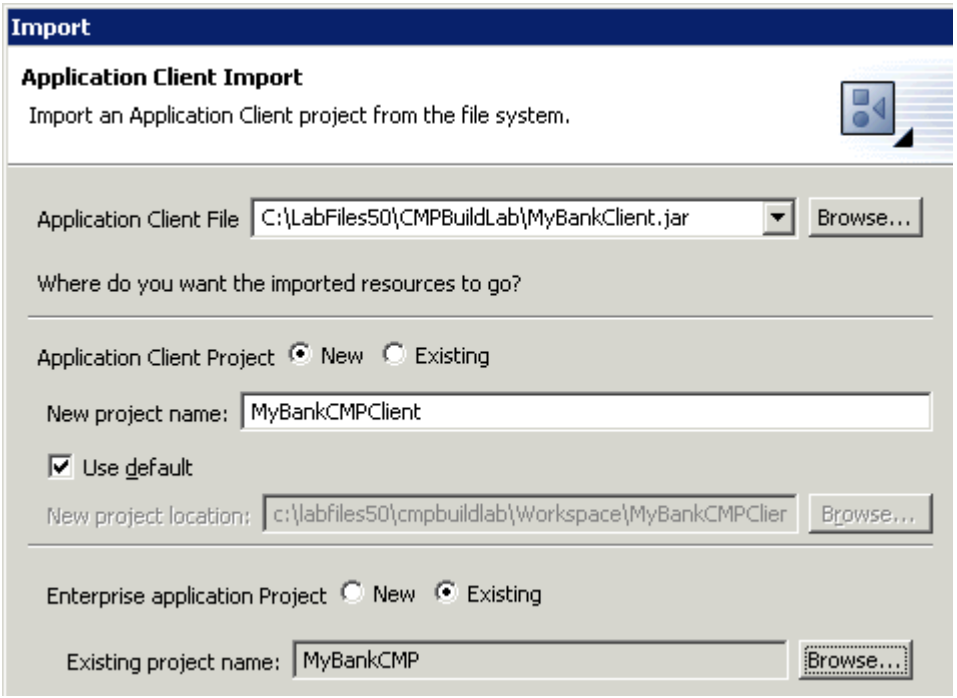
__2. With the Web Module added to your application, now you can add the Client Application Module.

__ a. Select **File > Import...** and select **App Client JAR file** and click **Next**.

__ b. For the Application Client File, browse to **C:\LabFiles50\CMPBuildLab\MyBankClient.jar**.

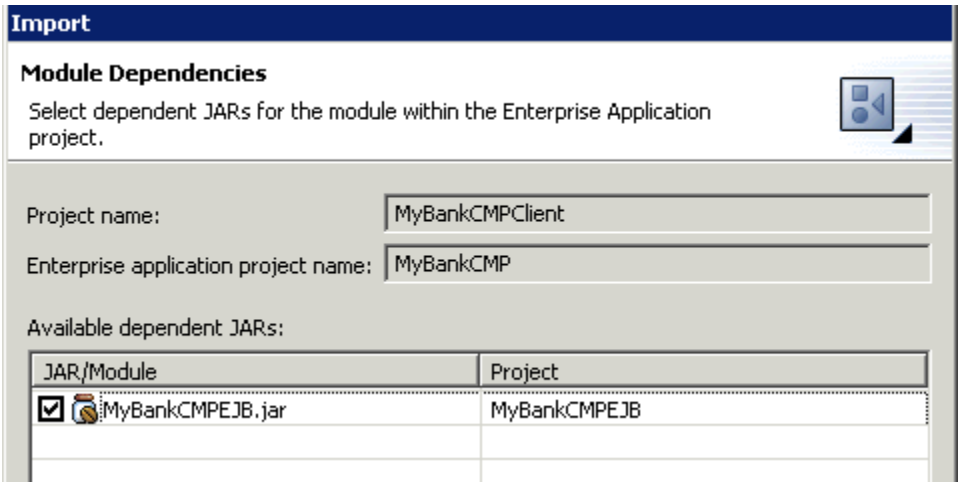
__ c. For the New project name for the Application Client Project, enter **MyBankCMPClient.**

__ d. Next to Enterprise Application Project, select the **Existing** radio button and click the **Browse...** button. Select **MyBankCMP** and click **OK**.




__ e. Click **Next**.

__ f. On the Module Dependencies window, select the checkbox for **MyBankCMPEJB.jar**.




__ g. Click **Finish**.

> **Note:** A number of errors will be displayed in the Task view. You will resolve these errors in the next step.

__ h. Just as you updated the Java Build Path on the Web module, you will also have to update it on the Application Client Module. Right-click on MyBankCMPClient and select **Properties**.

__ i. Select **Java Build Path** and select the **Projects** tab. Check the box for **MyBankCMPEJB**.

__ j. Select the **Order and Export** tab, and check the box for **MyBankCMPEJB** again. Select **OK**.

__ k. Optional: Go to **MyBankCMPClient** project and expand the folders **appClientModule\com.ibm.mybank.client**. Open the **TransferApplication.java** file and notice the calls to the **remote** interface of the Transfer session bean (search for transfer and transferHome).

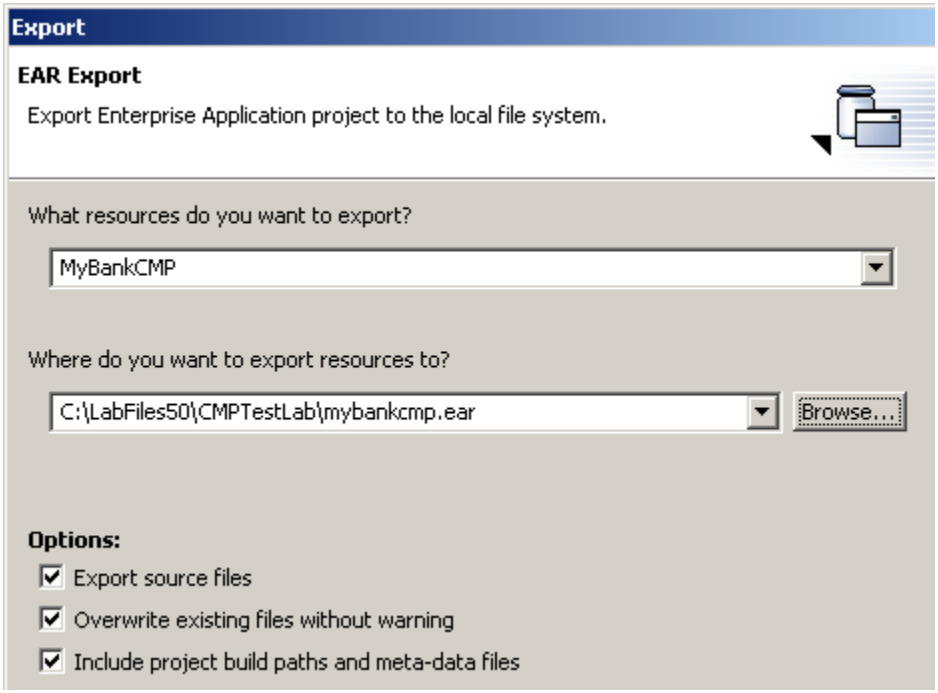
__3. Your application is now complete. It is ready for testing inside Application Developer or installation onto a production server. Installation to a production server can be done right from Application Developer, however, you may want to export out the EAR file and install it manually to your server. Export the application as an Enterprise Archive file.

__ a. Select **File > Export...** and select **EAR file** and **Next**.

__ b. For the resource to export, select **MyBankCMP**.

__ c. For the location where the exported resources should be placed, browse to **C:\LabFiles50\CMPTestLab** and enter **MyBankCMP.ear** for the File name.

__ d. Select the checkboxes **Export source files**, **Overwrite existing resources without warning**, and **Include project build paths and meta-data files**. Application Developer v5.0.1 allows you to export out with your Enterprise Application the project settings and dependencies established in your development environment. Although these settings have no effect on running the application in a production server, they do allow for the EAR file to be imported into a different installation of Application Developer and avoid setting the dependencies and Java Build Paths for projects.




__ e. Click **Finish**.

__ f. Close WebSphere Studio Application Developer by selecting **File > Exit**.

Now your MyBankCMP application is ready for production and can be installed in WebSphere Application Server v5.0.1.

## Part Four: Installing the solution (Optional)

> **Note:** These steps only need to be completed if the preceding steps have not been completed or if they were skipped.

__1. Start WebSphere Studio Application Developer.

__ a. Select **Start > Programs > IBM WebSphere Studio** and select **Application Developer 5.0**. In the dialog box, enter **c:\labfiles50\cmpbuildlab\workspaceSolution**. Click **OK**. It will start Application Developer with an empty workspace. An empty workspace will leave your existing workspace untouched and help avoid name conflicts between what you may already have in your workspace and what you will be creating in this lab. If you already have an empty workspace, you can start Application Developer from the Start menu.

__2. Once Application Developer starts, import in the solution.

__ a. Select **File > Import...**.

__ b. Select **EAR file** and **Next**.

__ c. Select **Browse...** and navigate to **c:\labfiles50\cmpbuildlab\solution\mybankcmp.ear** and select **OK**.

__ d. For the Project name, enter **MyBankCMP**.

__ e. Select **Finish**.

__3. At this point, you can view the different enterprise beans or modules.

## What you did in this exercise

In this exercise you created a J2EE 1.3 application with an entity enterprise bean compliant with the EJB 2.0 specification. The entity bean implements the new persistence model of the EJB 2.0 specification and is available through local interfaces. You also created a session enterprise bean which implemented both local and remote interfaces. The session bean was configured with an EJB Reference to use when looking up the entity bean. Finally, you added Web and Application client modules to the application before exporting the Enterprise Archive file out of Application Developer.