CHAPTER 6

Business Services in ADF

"What is the use of repeating all that stuff,"
the Mock Turtle interrupted,
"if you don't explain it as you go on?"
"It's by far the most confusing thing I ever heard!"
—Lewis Carroll (1832–1898), *Alice's Adventures in Wonderland*

s we explained in Chapter 3, ADF Business Components provides the best fit for relational-minded developers who need to wire the database to Java programs. Conventional descriptions of and discussions about ADF Business Components often focus on its major objects: view objects, entity objects, and so on. This approach doesn't really deliver the information that you need and the questions that you probably want answered from the start. This chapter takes a different approach; it presents ADF Business Components essentials and key components in a task-oriented manner.

This chapter answers the questions you will ask during the development of a typical database application:

- **How do I issue a query to the database?**
- **How can I update data?**
- **How do I generate a primary key value?**
- **How do I handle transactions?**
- **How does record locking work?**
- **Where is the login dialog?**
- **How do I define business rules?**
- **How can I dynamically change a query?**
- **How can I interface ADF BC with PL/SQL?**

In answering these questions, we'll necessarily have to cover some related topics, such as validation and key generation.

How Do I Issue a Query to the Database?

Coming from a PL/SQL background, it seems like executing a query on the database should be the simplest thing in the world. In PL/SQL, you just embed the query inside the code and provide variables for the return values. Unfortunately, in Java, this is not a trivial task at all. Unlike PL/SQL, Java has no native understanding of SQL and you have to do a lot of work to just get to the point where you can start submitting a statement to the database. ADF Business Components, of course, is designed to carry out a lot of the low-level work and makes the whole process more straightforward. So, knowing this, if you want to create a query using ADF Business Components, what do you do?

To illustrate the process, we'll provide some high-level steps you would use in JDeveloper to create a query that populates a list of values from the Departments table in the Oracle HR schema.

1. Set up an Application Workspace and Project for ADF Business Components

Chapter 3 highlights the use of application templates through the Create Application dialog. You reach this dialog using the **File** | **New** menu option to display the New Gallery, and then selecting the Application option from the General category. The template to use in this case is called "Web Application [JSF, ADF BC]." Creating such an application produces a workspace containing two projects: Model and ViewController. The Model project is set up for ADF Business Components objects; the ViewController project is used to define the application user interface, so you can ignore it for now.

2. Create the Query Object

With the Model project selected, we open the New Gallery again. Under the Business Tier category, select ADF Business Components, as shown in Figure 6-1.

The gallery is sensitive to the technology scope of the project and filters the display of categories accordingly. If you change the *Filter By* value to "All Technologies," several other options, such as Web Services and EJBs, will appear in the Business Tier category.

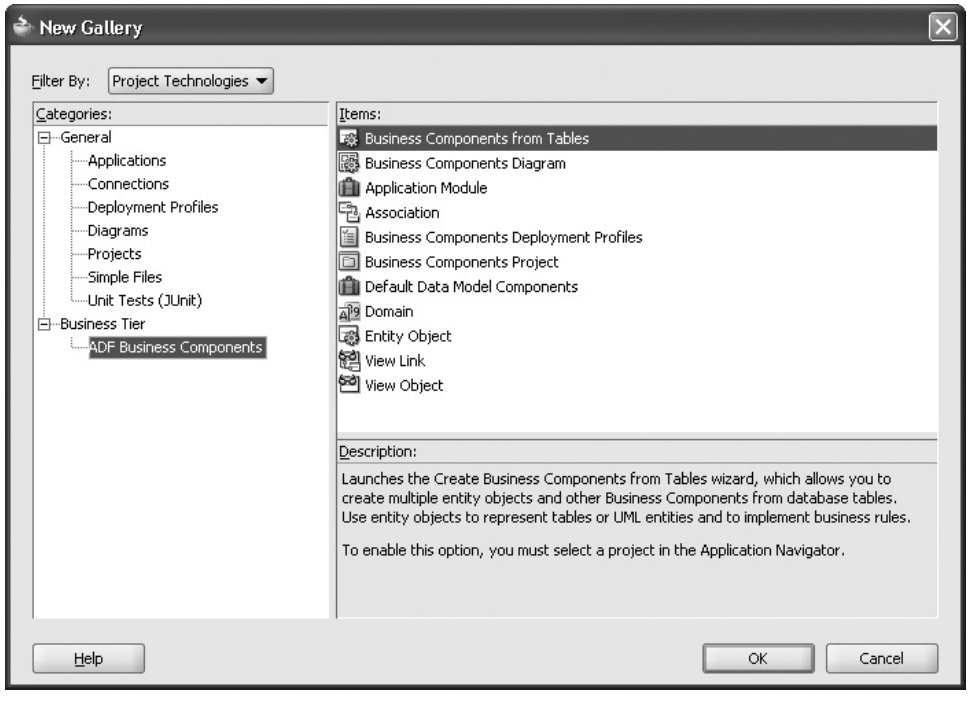


FIGURE 6-1. *The New Gallery for an ADF Business Components project*

With the ADF Business Components category selected, the list of possible items shown in Figure 6-1 is displayed. The first item we need is "View Object." The best way to think of a *view object* is as a query definition, rather like a conventional database view (a full SELECT statement) or the *Query Data Source Name* property of an Oracle Forms data block definition (that defines the database source for the query data). As you go through the process of defining a view object, you'll see the similarity in the information that is being collected to the kind of information that would be used to define an Oracle Forms data block. For example, the information in a view object includes:

- The SELECT clause, used to define the data elements
- The data source (FROM clause)
- The WHERE clause, used to filter the results
- The ORDER BY clause for the query
- The number of rows to fetch when populating the query
- Optimizer hints

The similarity does not end there. As in Oracle Forms, the WHERE clause and the ORDER BY clause can be manipulated at runtime using code.

3. Define the Query

When you create a view object from the New Gallery, after specifying a connection, a wizard (shown in Figure 6-2) walks you through the basic definition process. This wizard does not expose all of the attributes of the view object; the attributes properties and some advanced features, such as the tuning options, can only be accessed after you create the view object.

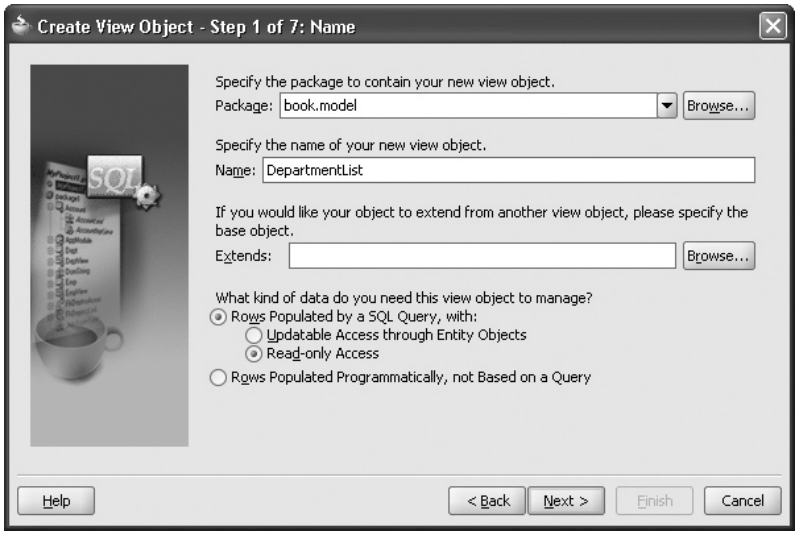


FIGURE 6-2. *Create View Object dialog; Step 1 of 7: Name*

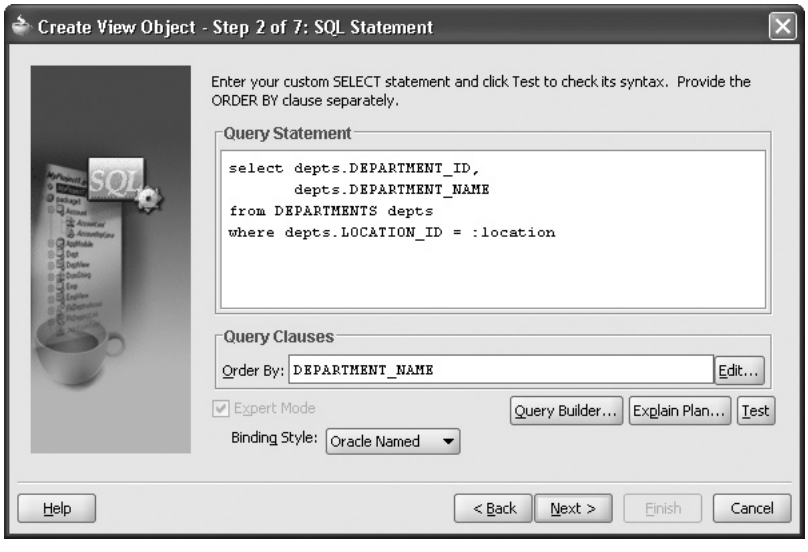
In the first screen of the wizard, you provide a descriptive name, for example, "DepartmentList," and specify a package for the view object. The package is the Java package name, which is used to help organize the code, both logically in code references and also physically on disk. The package defines the directory structure used to store all of the code and metadata files that make up the view object that you're creating.

TIP

Although it is fine to accept the default package names suggested by JDeveloper, it is a good idea to fully define package names early on to help maintain your code as it evolves. For example, you may choose to place all of the read-only view objects (queries) used for list-of-values lookups in a separate package from those used for updateable access.

Also on this screen, we'll select the *Read-only Access* radio button within the *Rows Populated by a SQL Query, with* option. This radio button declares that we will be supplying a SQL statement to populate the view object at runtime.

The next screen in the wizard gives us the opportunity to define the SQL statement, as shown here:



This screen should represent familiar territory. A *Query Statement* and *Order By* clause can be entered directly in property fields, or you can invoke the Query Builder (just as in Oracle Reports). The query defined here can use joins, subqueries, aggregations, set operators, and so forth. There is no restriction confining the query to a single table or database view.

If you are likely to reuse a query statement with variations in the query criteria, you should use bind variables to represent the changeable conditions in the query. This is an alternative to using several similar statements with hard-coded values. The bind variables can then be dynamically set at runtime to customize the query results. This approach of using bind variables bestows several benefits:

- **Database scalability** The statement is cached as a single reusable cursor in the database shared SQL area rather than a separate cursor being created for each combination of query and criteria.
- **Project size** For a database application of any complexity, many queries will be needed. Reusing the same statements with the aid of bind variables will help to keep view object proliferation under control. This makes it both easier to manage the project at design time and reduces the memory requirements at runtime, as fewer object definitions will have to be managed by the framework.

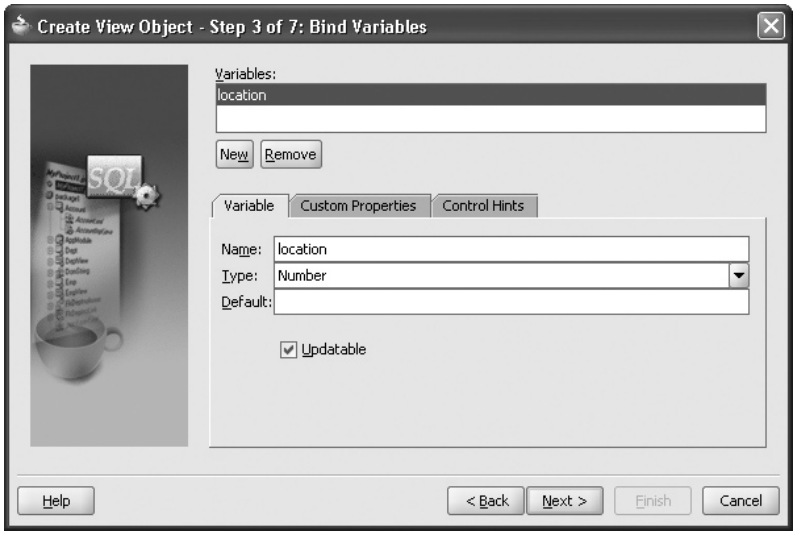
You define embedded bind variables using one of three notations, selected by the *Binding Style* drop-down list:

- **Oracle Named** This bind variable reference uses a meaningful name, for example, location, prefixed with colon.
- **Oracle Positional** Bind variable placeholders are indicated by a colon, followed by a zero indexed number, for example :0, :1, :2, and so on. These numbers are expected to be sequential.
- **JDBC Positional** Bind variables are represented as question marks in the statement.

Coming from a PL/SQL background, using named bind variables will no doubt be the most familiar approach. This feature is specific to the Oracle JDBC drivers and so can only be used when Oracle is your target database. Named bind variables also allow you to reuse variables within the same statement, rather than having to resupply the same value again, as JDBC-style question mark placeholders require.

CAUTION
The Oracle Positional style of numbered bind variables seems to promise bind variable reuse in the same way that the Oracle Named style does. However, this turns out not to be the case, and reuse of the bind references will not work. We recommend that you use Oracle Named as a binding style unless database portability is required, in which case, the JDBC Positional style should be used.

If you do use bind variables in a particular query, be sure to define some default values for them. This is defined in Step 3 of the Create View Object Wizard, as shown here:



This page is used to define the name, datatype, and default value of any bind variables used by the query. The Custom Properties and Control Hints tabs allow additional metadata to be added to the bind variable definition. At runtime, ADF uses this metadata to create default labels and other user interface features, such as date formatting for the bind variable if it appears in a screen as a field. This extra metadata is optional.

At this point, we have everything that is needed in the query definition. Clicking the Next button again in the wizard displays a list of the attributes defined by the query. Click Finish to complete the wizard. To change the definition of the view object after it has been created, double click the view object node in the navigator or select **Edit Department** from its right-click menu. The View Object Editor will present the same pages found in the wizard, plus a few extra pages for advanced features.

4. Expose the Query

There is no obvious way of running or testing the query that we've just defined, other than through the syntactical check offered by the Test button on Step 2 of the wizard. In order to test the query and see the resulting data, a second type of object is required to represent a runable service interface.

To run this object, we need to create another object called an "Application Module." You can return to the New Gallery and choose "Application Module" from the ADF Business Components category, or you can display the right-click menu on the package node of the Model project and select **New Application Module**.

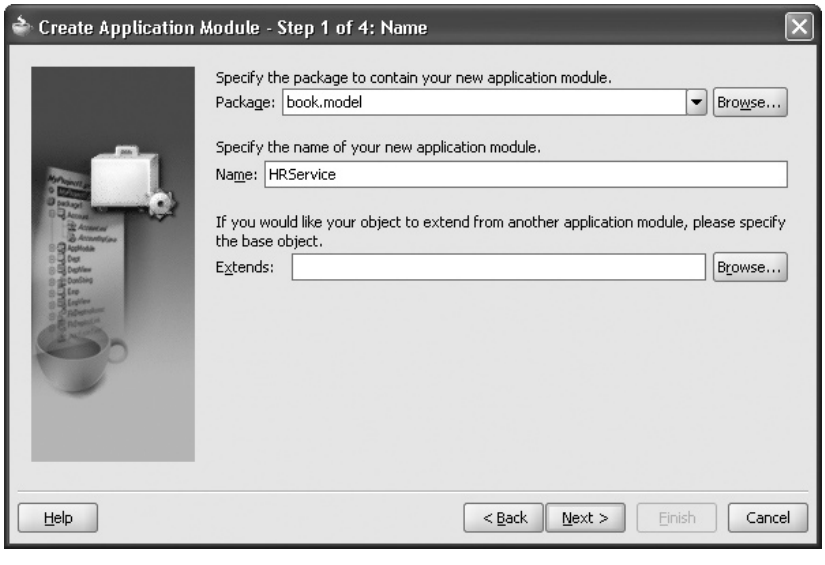
The *application module* is the service façade (see the sidebar "What is a Service Façade?" if that term is not familiar) for an ADF Business Components project. Everything that needs to be visible to the consumers of the service has to be exposed through this object. In principle, the

What Is a Service Façade?

The term *service façade*, or *service interface*, is used to describe the public functions that a module, typically one that interacts with a database, exposes. You program consumers of this service to use this external API, but these consumer programs have no knowledge of the service's internal implementation. A parallel in the PL/SQL world is the specification portion of a PL/SQL package, which acts as the façade for the body. The details of the code in the package body are hidden from the consumers (calling program units).

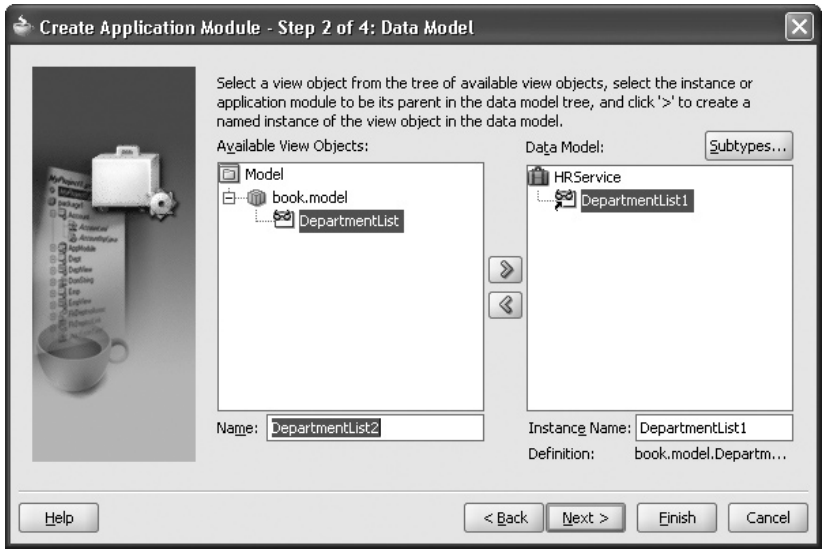
application module acts a little like the Oracle Forms runtime session, as both the entry point to the application and the thing that maintains the database connection and the transactional state (something we'll cover a little later on).

Creating an application module displays the Create Application Module Wizard shown here (after dismissing the Welcome page):



The next page of this wizard (Step 2 of 4: Data Model, shown in the next illustration) is the important one. It allows us to specify which view objects should be exposed to the outside world through the application module. The view objects defined in an application module are called *view object instances*. In this case, we only have a single query to run, so that can be shuttled to the right side of the selection control. You do not need to use the application

module to expose view objects for queries that are only used within your business logic. You can still access these view objects within the business services layer, but they are essentially private to that scope.



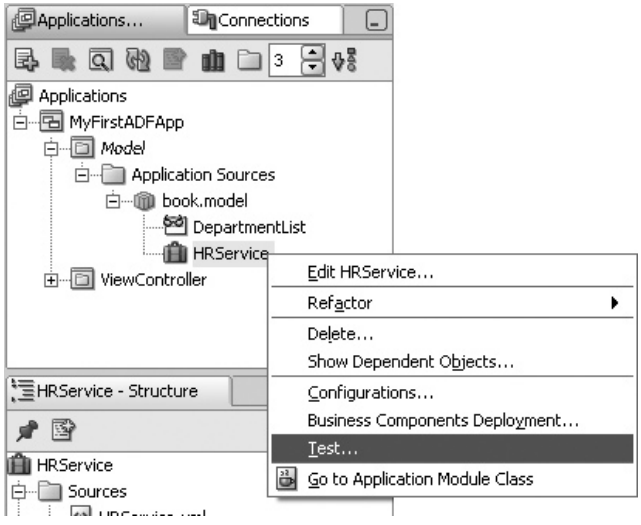
Notice that when the DepartmentList view object is selected and exposed through the data model, it is given the name "DepartmentList1." The numerical suffix is automatically added to the name of the view object instance to ensure unique naming if the view object is reused in several different contexts. You can alter this name by changing the value in the *Instance Name* field beneath the *Data Model* list.

Once you define the view object instance, you can click Finish to complete the wizard.

5. Test the Query

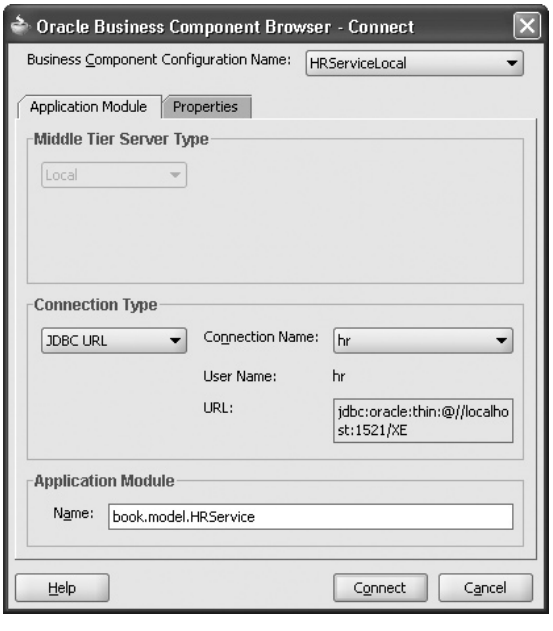
So far, you've created an object that defines a query—the view object—and a way of exposing it to the outside world through the application module. Despite all of this, we still don't appear to be much closer to seeing a list of departments. Consider an analogy with Oracle Forms here. When you define a block with the wizards, the Data Block Wizard runs first and defines the query part of the block definition. Then, the Layout Wizard (optionally) runs to define the layout of the user interface.

In the process described so far, we have effectively run that first wizard, but not the second. It would be inconvenient if we now had to go and build a user interface just to test this query. Fortunately, however, ADF Business Components provides a built-in testing utility (the *Oracle Business Component Browser*), which you can invoke by selecting **Test** from the right-click menu on the application module in the Application Navigator, as shown next.



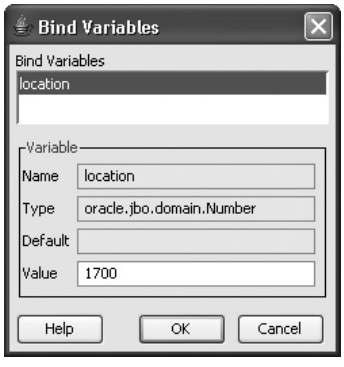
This testing utility will examine the application module for all of the publicly exposed view objects and generate a simple screen to allow you to run the query and test any embedded business logic.

The first screen of the tester (shown next) defines the connection and the configuration that the tester should use.

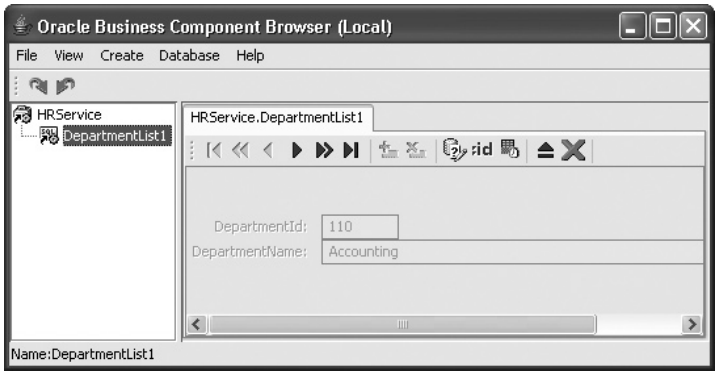


In most cases, you just accept the default, which will use the same database connection used for creating the objects in the first place.

Once connected, the tester displays a list of available view objects. To view the results of the query, double click the name of the view object. The right pane will then show the records for the query. If one or more bind variables are defined for the view object, as in this case, an intermediate dialog (shown here) will pop up, allowing a suitable test value to be entered.



After you enter a query value and click OK, the query will be run and the records displayed, as shown next:



The fields for the view object are disabled in this case because we are testing a read-only view object, but you can scroll through the records to view the query results.

How Can I Update Data?

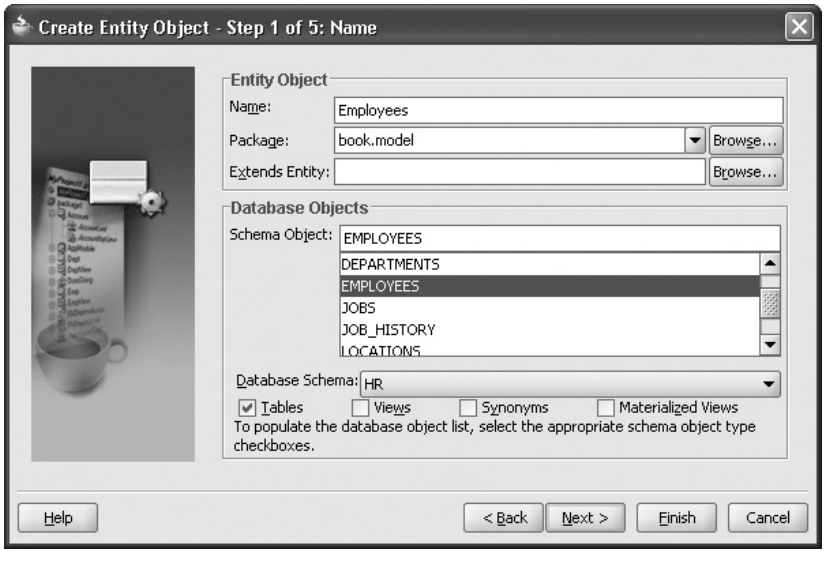
The process of defining a simple read-only query has introduced two of the five major ADF Business Components artifacts: view objects and application modules.

Now we need to explain how data can be queried from the database, changed, and then put back—essentially the basic functionality of an Oracle Forms data block. If we think of the view object as defining the block's *Query Data Source*, how does the corresponding block property *DML Data Source* get defined?

1. Create an Entity Object

In order to implement update functionality, a new object type needs to be created: the entity object. *Entity objects* have a one-to-one mapping relationship with database objects, for example, a table or a view. They represent a sort of business-tier local cache of the rows from that database artifact. In Chapter 3, we looked at the logical structure of an Oracle Forms block and discussed the Record Manager. In Oracle Forms, the Record Manager keeps a local cache of rows and enables the engine to track updates and manage scrolling. Entity objects fulfill basically the same role and offer some additional functionality, such as validation.

Like the view object, an entity object can be created from the New Gallery or the package right-click menu. This launches the Create Entity Object Wizard, shown here (after dismissing the Welcome page):

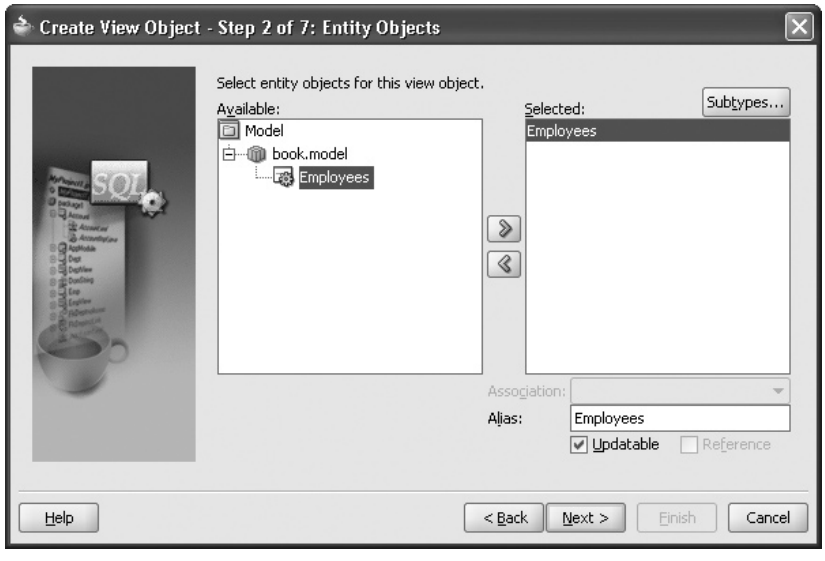


The wizard page for selecting a source database object (EMPLOYEES in this example) is about what you'd expect. The additional pages in the wizard allow you to select the columns required from the source table or view, and then to further refine these with properties, such as a primary key flag. Columns are mapped to entity object *attributes* that combine information about the underlying database column with additional metadata about validation and user interface properties.

When creating an entity object based on an object in an Oracle database (such as the EMPLOYEES table), the wizard is smart enough to query the data dictionary for the key properties of each attribute, such as size and data type. Therefore, you are finished once the source object has been selected, since everything else will be set to intelligent defaults based on definitions in the data dictionary.

2. Create an Updateable View Object

How does the existence of an entity object help for updating data? To retrieve the records to be updated, you need to create a view object (called EmployeesUpdateable, for example) that is associated with that entity object. In Figure 6-2, you'll notice that the other option within the *Rows Populated by a SQL Query with* radio group was *Updatable Access through Entity Objects*. If you select this option when creating a view object, the following wizard screen (Step 2 of 7: Entity Objects) will be shown:



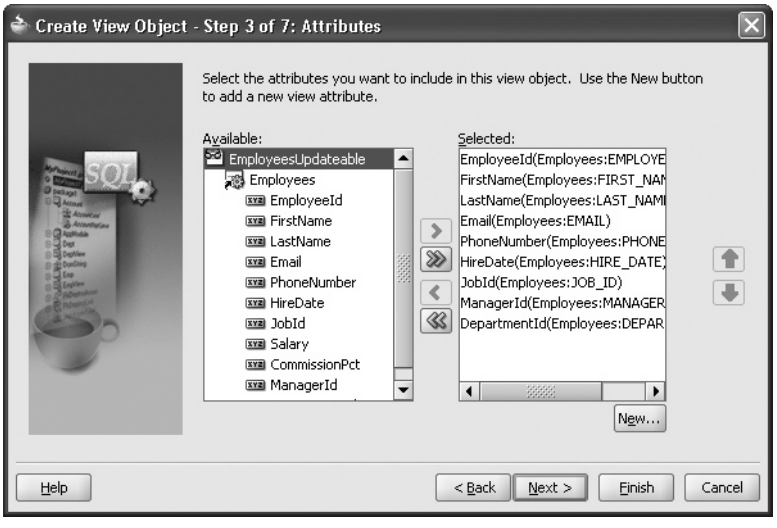
We are not confined to selecting just a single source entity object here. View objects can literally be a "view" over several tables as represented by several entity objects. At this point, things begin to diverge from the viewpoint that we might have as Oracle Forms programmers. One of the conventions of Oracle Forms programming is that a block can, for the purposes of update anyway, only be based upon a single database object. (The *DML Data Target Name* block property can only hold one name.) Creative use of transactional triggers or updateable join views or procedure-based targets can circumvent this restriction, but it is not an out-of-the-box experience. On the other hand, a view object may be based upon several entity objects and all of them can be made updateable if required.

NOTE

Unlike the situation with Oracle Forms where the Record Manager maintains a separate cache for each data block, the entity object row cache is shared among all view objects that use that particular entity object. This means that all of the view objects see any local uncommitted changes in the cache, and you are spared some of the self-locking issues that can occur in a form with multiple data blocks containing the same records.

3. Define the View Object Attributes

The next screen of the Create View Object Wizard (Step 3 of 7: Attributes), shown next, allows you to select attributes from the entity object(s) that are to be exposed through this view object.



Any or all of the attributes can be selected. The only restriction is that the attributes representing primary key columns of the underlying tables must be included.

TIP

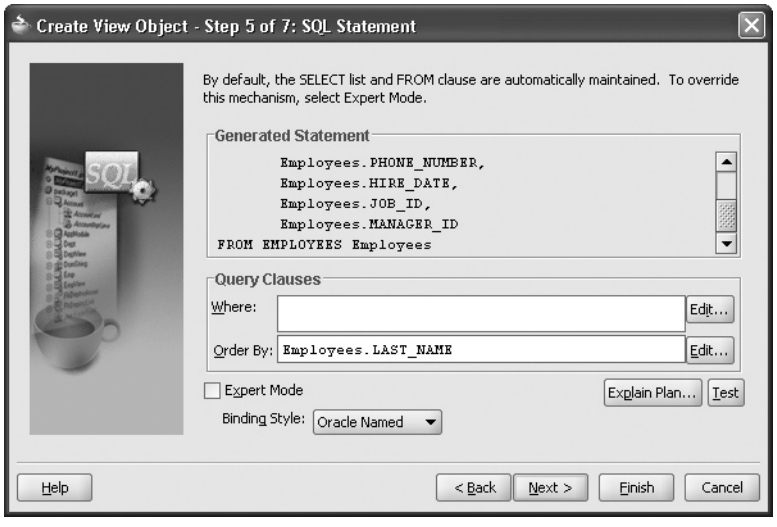
A view object can also contain "nondatabase" attributes that are not directly based on fields from the underlying entity objects. These attributes function in the same way as "non-base table items" in an Oracle Forms block. These view object attributes may be indirectly based on entity attributes, for example, in the form of a calculation, in which case they are referred to as "entity-derived attributes." Alternately, they may be unrelated to the entity cache in any way, in which case, they are referred to as transient attributes. *The New button in the Attributes page is used to create both of these.*

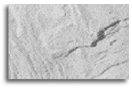
Once the columns are chosen, the next screen (Step 4 of 7: Attribute Settings) allows us to further define the field attributes. The properties for the initial field settings are inherited from the base entity object, so the default selections are usually sufficient.

4. Refine the View Object Query

The information gathered so far, in the selection of entities and columns, is enough for a view object's query to be generated. Again, this is similar to a base table query in Oracle Forms where the text of the query statement is generated (by the Oracle Forms runtime) from the selected table

and columns. The next page of the wizard (Step 5 of 7: SQL Statement) allows the definition of WHERE and ORDER BY clauses for the query, as shown here:





NOTE
The query statement generated from properties you set in the wizard's pages appears in the "Generated Statement" pane. As you type in the WHERE clause, this statement is updated. In addition, adding or deleting attributes or changing the entity object assignments in preceding pages will update this query statement.

That is all that you need for the updateable view object to function, so you can complete the wizard at this point.

5. Test the View Object

The view object can be exposed through the application module (by including the EmployeesUpdateable view object in the data model) and tested. This time, the fields in the tester are enabled, and we are able to make updates to the Employees object. Just like Oracle Forms, any data changes are held in the cache (the entity object cache, in this case) and will be applied to the database when a commit is issued. The tester provides a toolbar containing functions such as "Commit" to help you to test view object functionality comprehensively, as shown in Figure 6-3.

How Do I Generate a Primary Key Value?

If an entity object is updateable, chances are that new records will be inserted through it, and that being the case, it may well need to have a primary key value allocated to it. In Oracle Forms or

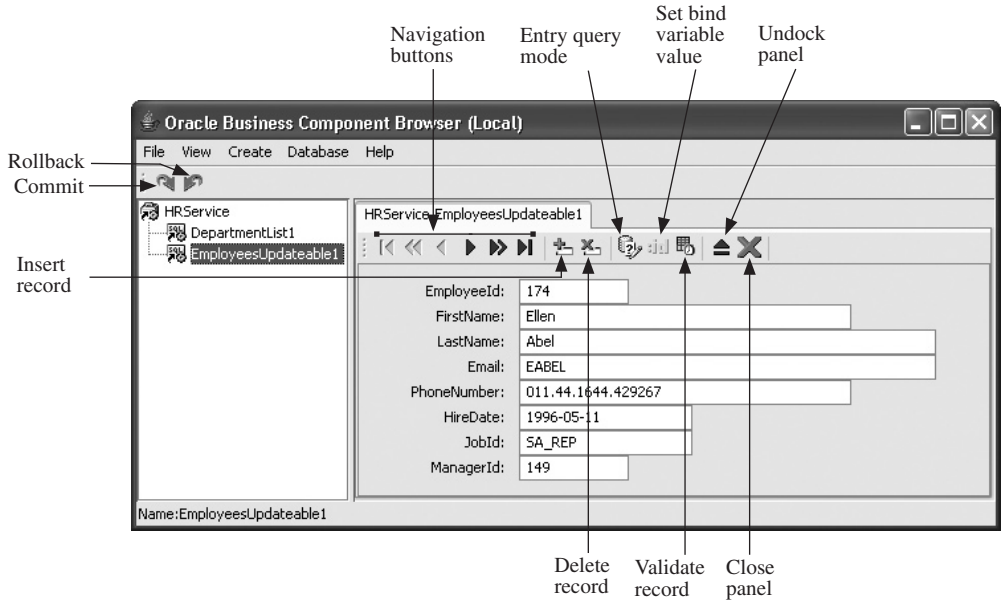


FIGURE 6-3. *The ADF Business Component Browser*

PL/SQL, this is generally achieved using a database sequence to provide the unique key value, which can then be used in one of three ways:

- **In the insert statement** Including a reference to <sequence name>.NEXTVAL directly as the value of the key column
- **In a trigger** Either an Oracle Forms PRE-INSERT trigger or a database BEFORE INSERT row-level trigger on the table
- **By reference** Oracle Forms can reference a sequence number in the initial value for a key field, and the framework will generate the correct insert SQL to load the field from the database sequence.

In ADF Business Components, there are two ways of carrying out this function:

- **Using a database trigger** and some declarative properties on the entity object
- **Writing code** in the entity object create method

We will look at the second of these two techniques later in the chapter when we cover adding code to the entity object. For now, let's look at the more common declarative technique.

Using a Database Trigger to Allocate Sequences

Using a database table trigger is a common approach to allocating a unique sequence number to a record as it is created in the database. The problem with this method is that the actual primary

key value is not allocated until the record is committed, and that can lead to difficulties if multiple master and detail records are committed to the database as part of the same transaction. The lack of a known key for each master means that the associated detail records have no parent identity with which to be associated.

ADF Business Components solves this problem by using a specialized data type or *domain* (see the sidebar "About ADF BC Domains" for more detail) called a *DBSequence*. When an entity object attribute is defined as being of this type, it will gain two useful behaviors:

- When a new entity object instance is created, it is allocated a temporary (negative) value. Any detail rows that hang off this record will refer to that temporary value, maintaining their relationship to the master record.
- The local temporary key value in the master (and the master's detail records) is automatically refreshed with the correct value from the database after the commit process is completed.

Setting unique key generation for the Employees entity object in our example, therefore, consists of three steps:

1. Create a database sequence (if one does not exist already) to be the source of the unique key (using "`CREATE SEQUENCE employees_seq;`").
2. Define a BEFORE INSERT row-level trigger on the EMPLOYEES table to allocate a value to the EMPLOYEE_ID column from the sequence when a record is created.
3. Set EmployeeId, the primary key attribute (based on the EMPLOYEE_ID column) in the Employees entity object, to be of type `DBSequence`.

NOTE
Chapter 13 provides steps for using DBSequence to implement a primary key value based on a database sequence in the context of developing a sample application. This will give you more practice in implementing this frequent requirement and will allow you to test the code within a J2EE web application page.

Create the BEFORE INSERT Trigger The following code provides an example of the database trigger that would be used to load the EMPLOYEE_ID of the EMPLOYEES table. The sequence object in this case is called EMPLOYEES_SEQ. Triggers can be defined in SQL*Plus or from within

About ADF BC Domains

Domains within ADF Business Components are specialized data types that can be allocated to entity and view object attributes. The domain you will encounter most often is DBSequence. A domain can define both a standard set of properties for the datatype (for example, the length, whether required, and so on) and behaviors, such as validation rules. Using domains for commonly occurring datatypes that always have the same validation logic—for example, a postal code or telephone number—promotes consistency and reduces the amount of code in the application.

JDeveloper by opening the connection to the database in the Connection Navigator and selecting **New Trigger** from the right-click menu on the Triggers node. The resulting screen will allow you to create a skeleton BEFORE INSERT trigger on the Employees table, which should then be filled out with the following code:

```
TRIGGER employees_bi BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
   SELECT EMPLOYEES_SEQ.nextval
   INTO :new.employee_id
   FROM dual;
END;
```

This trigger loads a generated number into the EMPLOYEE_ID column from the sequence regardless of whether the INSERT statement contains an EMPLOYEE_ID value. This ensures that the value is always loaded from the sequence and that it will be unique.

Set the Data Type of the Entity Attribute The attribute definition panel on the entity is used to set the attribute type to DBSequence. This screen is displayed as Step 3 of the Create Entity Object Wizard, and you can also access it after an entity object has been created by double clicking the entity object node in the Application Navigator, expanding the Attributes page, and selecting the EmployeeId node in the Property Navigator, as shown in Figure 6-4. Here, the default `Number` data type for EmployeeId has been replaced with the `DBSequence` type.

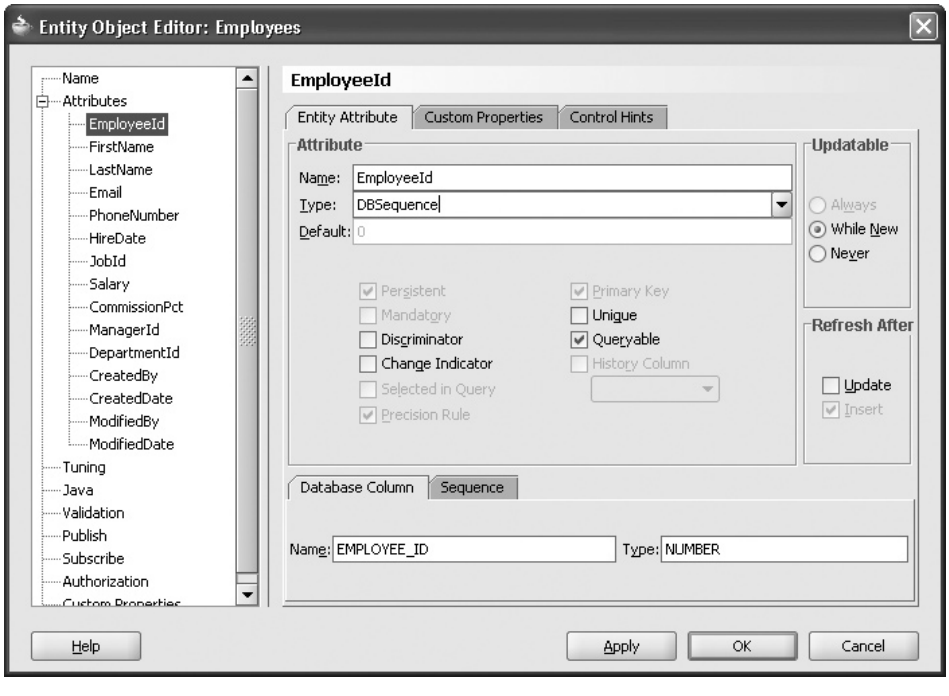


FIGURE 6-4. *The Entity Object Editor attributes page*

CAUTION
Do not be fooled by the Sequence tab at the bottom of this page. This allows you to define a sequence name to associate with the attribute. However, the sequence name, as defined, is only used if a schema is forward-engineered (to create database objects) from the ADF Business Components definition. It is not used to generate sequence values for new entities.

Mutating Data

Primary key generation is a situation where the data that is posted to the database is changed within the database transaction before being committed. Such mutating data has always posed a problem to frameworks such as Oracle Forms that maintain a local cache of data. The changes made in the database transaction are out of view of the framework. Both ADF Business Components and Oracle Forms solve this in the same way by allowing attributes to be flagged for "refresh after commit." When this is set, the framework is alerted that the data may be changed and that any change needs to be synchronized with the local cache.

If you refer to Figure 6-4, the area on the right side of the page labeled "Refresh After" contains two checkboxes offering the options *Update* and *Insert*. In the case of the Employee attribute shown in Figure 6-4, the value will only be mutated by an insert, so that checkbox is selected and disabled automatically when you enter the type as DBSequence.

If your application makes use of database triggers that could mutate data in the same way, you should set these options according to the operations that cause the mutation.

History Columns

While on the subject of data changes made by database triggers, the entity object *history column* feature is worth a mention. In Figure 6-4, this can be seen but it is disabled because the column has already been marked as a primary key.

The history column feature provides a declarative way to carry out auditing operations that you would often achieve through database or Oracle Forms trigger code. When an entry object attribute is marked as a particular history column, ADF BC automatically populates it for you, removing the need to write code to do the same thing. Table 6-1 lists the available history columns and their functions.

History Column	Purpose
created on	This value causes the entity object to store the date and time of the record's creation. It can only be used for an entity attribute that is associated with a date or timestamp database column.
modified on	This value stores the date and time of the last update to the record. Again, this must be mapped to a date or timestamp column.
created by	This value stores the identity of the user that created the row. It must be mapped to a character column of some type (CHAR or VARCHAR2).
modified by	This value stores the identity of the user that last updated the record. It also must be mapped to a character column.
version number	This value specifies a number that will be automatically incremented when the record is updated. This is not a sequence number in the sense of generating a primary key, but rather is a value that indicates the version of the record itself. Applications can compare the value of this column held in cache with the value in the database to see if another session has updated the record since it was queried. It must be mapped to a number column in the database.

TABLE 6-1. *History Column Meanings for Entity Attributes*

NOTE
History columns require that you use J2EE application security. The identity mentioned in reference to "created by" and "modified by" is the J2EE login, not the database account being used. We discuss J2EE application security in Chapters 10 and 14.

How Do I Handle Transactions?

Now that we've explained the basics of querying and updating, it is time to see how ADF BC handles some of the key transactional concepts, such as committing and locking.

The ADF Business Components tester provides Commit and Rollback buttons, but what are they doing and what is their scope? In order to explore this, we'll need to look at a little code.

The Transaction Object

In ADF Business Components, the application module is the unit of transactional scope. When a commit is issued, that event will affect all entity objects encapsulated within view objects instantiated in that application module. Additionally, if you have nested application modules (See the sidebar "Nested Application Modules"), then these will commit as well.

Commit (or rollback) events are issued in one of two ways. Most simply, the ADF Model layer can issue a commit in response to a user interface event. This is normal and involves no coding at the ADF Business Components level. Another option, however, is to interact with the transaction directly in Java code. This approach is useful if you want to produce an application with discrete transactions carrying out defined units of work exposed through functional interfaces, such as web services.

Programmatic access to operations such as Commit takes place through a `Transaction` object. The following fragment of code shows a simple function in an application module file (for example, HRServiceImpl.java) that issues a COMMIT:

```
public void myCustomMethod() {
  //Do some work
  ...
  //Commit any changes
  getTransaction().commit();
}
```

Nested Application Modules

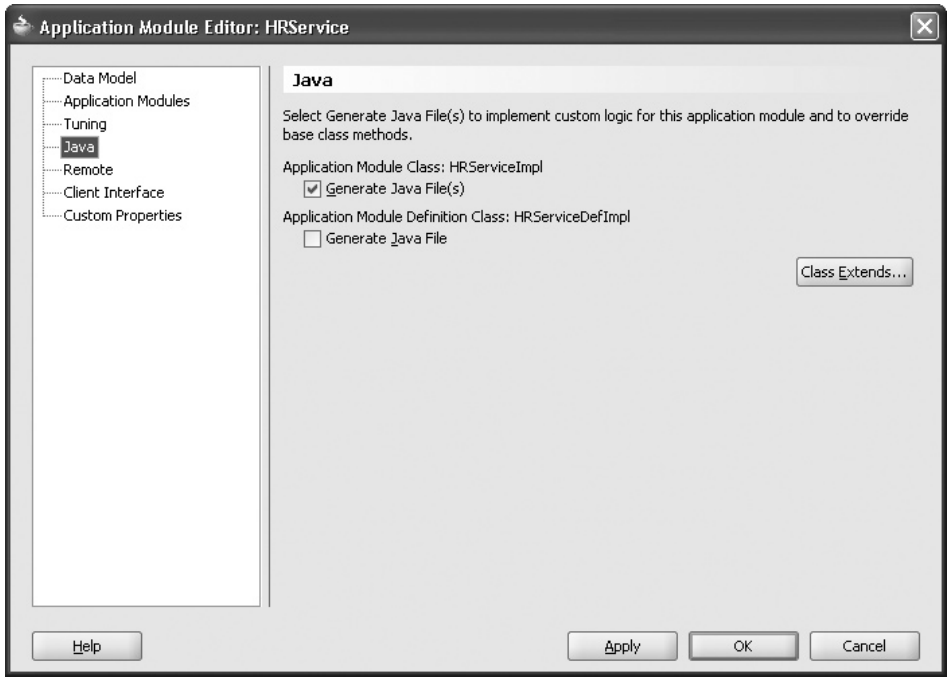
When we introduced application modules as a service façade for an application, the implication was that there would only ever be one application module to represent the interface of the entire business service. This is partially correct in that an application will usually only expose one top-level module. However, an ADF BC project may contain multiple application modules, which can be nested within the master application module. Any view objects or methods exposed by these child application modules are exposed through the Application Modules page of the application module properties.

Nested application modules provide a convenient way of organizing large projects into functional areas. The child modules can be developed and tested in separate projects and then combined under a master application module for deployment.

The transaction object is available using a getter method, `getTransaction()`. But where did this code come from? It is provided by the `ApplicationModuleImpl` class that this application module .java file has subclassed. The `Transaction` object exposes methods to commit and rollback, as well as many other transaction-related functions.

Extending Application Modules with Custom Code

Now we need to examine some Java source code. The application module .java file was created automatically in the Create Application Module Wizard (on Step 4, a page which was skipped over by clicking Finish earlier). The same options are included on the Java page of the Application Module Editor, as shown in the following illustration:



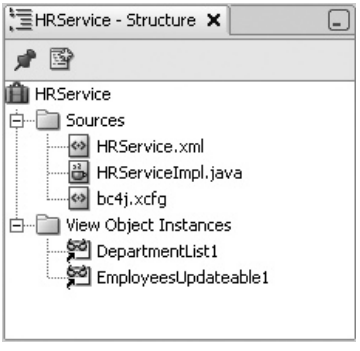
If both of these checkboxes are unselected, the application module will only exist as an XML definition file with no place to add code. However, the "Application Module Class: <name>Impl" option is selected by default, so that file will be created unless you have changed the base preferences, as explained in the sidebar "Altering the Defaults."

Altering the Defaults

ADF Business Components is designed to provide a set of default behaviors, such as generating a .java file for the application module, which makes sense in the majority of circumstances. These default behaviors can be configured if they are not suitable. Like other preferences in the Integrated Development Environment (IDE), these settings, and many others for ADF BC, are defined in the Preferences dialog (**Tools** | **Preferences**).

This application module *Impl* (short for Implementation) class is where we can add extra methods to the public service interface exposed by the application module.

The second class mentioned on this screen, the `HRServiceDefImpl`, is used to customize the XML metadata that the application module uses. This class is an advanced usage, and you should not worry about creating it at this stage. To see if the application module Impl file has been created, select the application module in the Application Navigator, and then view the Structure window. The result should look something like this:



Here we see the XML definition for the application module in the HRService.xml file, the Java Impl class file (HRServiceImpl.java), and a third file—bc4j.xcfg, which contains runtime configuration information used by the application module. We will examine this configuration file later in this chapter.

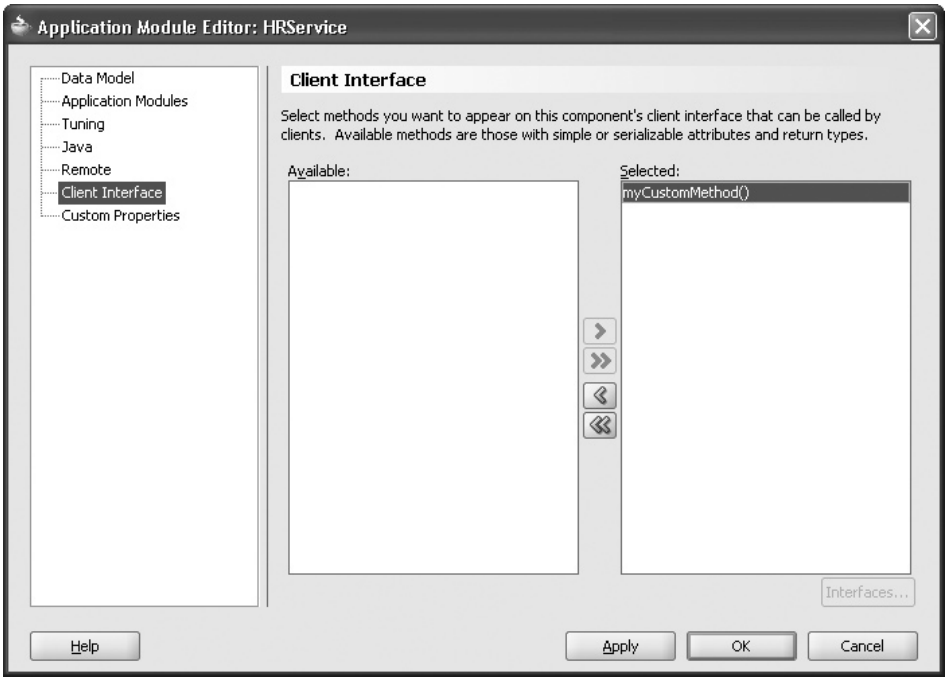
You can access the Java Impl file in two ways:

- Double click the Impl Java node in the Structure window, or select **Open** from the right-click menu.
- On the application module object in the Application Navigator, select **Go to Application Module Class** from the right-click menu.

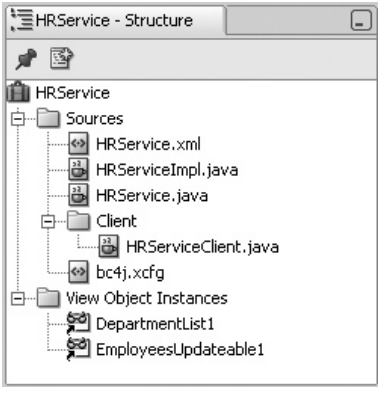
NOTE
If you use the System Navigator to view the project rather than the Application Navigator, you will see the .xml and .java files listed separately. You do not need to look at the Structure window.

However, creating the Impl file and adding some code is only half the story. You also need to specially flag methods that will be exposed as part of the application module service interface so that they can be used by a UI client. In typical Java development, you would just mark the methods as "public." In ADF Business Components, however, you need to complete an additional step to ensure that methods are exposed correctly. This step exposes the method through the ADF Model layer for use in user interfaces, and also makes the methods available as remote method calls. This is important, because the business service provided by the application module may not be co-located in the same JVM (Java Virtual Machine) as the web interface or Swing UI that is calling it.

Publishing these methods is handled in the Client Interface page of the Application Module Editor, as shown here:



Once one or more custom methods are published in this way, a number of extra files are generated into the project. These include an interface that represents the service's contract (that is, the signatures of the methods that have been published) and a client-side proxy Java file. This is used to handle the remote invocation of the method across the network. This simple publishing stage actually generates extra code to allow you to access the service effectively in an *n*-tier environment. Again, the extra Java files that have been created will appear in the Structure window for the application module, as shown here:



Once a method has been published through the Client Interface page in the application module, it will also be exposed through the Data Control Palette as a custom operation that can be bound into the user interface using drag and drop. Chapter 7 discusses how such methods can be used.

How Does Record Locking Work?

Record locking is a mechanism that the database provides to prevent simultaneous update access to the same resource by multiple users. Locking strategies and granularity differs from database vendor to database vendor, but with the Oracle database, locks are taken out on a row-level basis.

Just like Oracle Forms, ADF BC automatically handles the day-to-day process of issuing locks to protect updates in progress. You do not have to write code to issue locks. However, you can configure the way that these locks are managed.

Within ADF Business Components, the `Transaction` object, which we encountered earlier as a home for COMMIT and ROLLBACK operations, also provides a way to configure the locking mode of the application. This is achieved through the `setLockingMode()` method of the `Transaction` object. You need to pass this method one of the following constants:

- **Transaction.LOCK_NONE** ADF Business Components should not issue any explicit lock statements, leaving it to the underlying database mechanisms to manage locking.
- **Transaction.LOCK_OPTIMISTIC** ADF Business Components should issue explicit lock statements for updated and deleted data immediately prior to issuing those Data Manipulation Language (DML) statements to the database.
- **Transaction.LOCK_OPTUPDATE** ADF Business Components should issue explicit lock statements for updates only (partial optimistic locking). Deletions are the responsibility of the database.
- **Transaction.LOCK_PESSIMISTIC** ADF Business Components should issue explicit lock statements for updates and deletions as soon as the row is modified in the cache.

Ignoring the case of no locking, there are two basic modes: pessimistic and optimistic.

Pessimistic Locking

Pessimistic, as the name suggests, assumes that the row that has just been modified may also be vulnerable to change by someone else in the lifetime of the transaction. So, the pessimistic view is to immediately issue an explicit row lock to make sure that no other session can change the same row. In the case of the Oracle database, this will not prevent other users from reading the locked row.

Pessimistic locking occurs when you make normal updates through server-side PL/SQL. In this case, however, it is the database that carries out the locking rather than any mid-tier framework. Pessimistic locking is the default locking mode in ADF BC. It is also the default locking mode for Oracle Forms applications, although it is disguised as the "Immediate" value for the data-block property *Locking Mode*.

Optimistic Locking

Optimistic locking, on the other hand, makes the assumption that the changed rows will probably not be changed by anyone else in the immediate future, and the lock is deferred until the last

possible moment. This is equivalent to the Oracle Forms setting "Delayed" for the *Locking Mode* property. Optimistic locking differs from no locking, because all locks will be issued before the actual rows are touched in the database. This means that no DML will be issued if one or more of the locks fail.

In the case of no locking at all, the DML will be issued against the database until an implicit lock fails, and then all the work done so far will have to be rolled back.

Which Mode to Use?

The main reason that Oracle Forms applications primarily use pessimistic locking is that it provides much earlier feedback to the user about a locking problem. The alternative (optimistic) allows the user to enter the whole transaction's worth of data and commit before issuing a locking error.

Web-deployed J2EE applications generally exhibit block-mode characteristics for Online Transaction Processing (OLTP) data entry. In block mode, data is entered or updated a screen at a time and then submitted to the middle tier to process and pass to the database, if required. As such, the immediate feedback of pessimistic locking is not really applicable, because the user has already filled in a whole page of data. In fact, pessimistic mode is seen as a universally bad thing in web-deployed J2EE applications because of the asynchronous nature of browser-based interfaces. If the application secures locks in this way, there is no real guarantee that the user will issue a commit or rollback to complete the transaction and release the locks. It is not unusual for users to simply exit the browser, leaving hanging transactions and consuming resources, both on the application server and the database. Although both of these tiers will eventually clean up after the relevant timeout, in the meantime, other users may be blocked from changing the data. Therefore, the best practice for locking is to use optimistic mode.

This topic was introduced by showing the programmatic calls required to configure the locking mode, but surely there is a simpler way. In Oracle Forms, there is a declarative setting on the block. Does ADF Business Components have something similar? It turns out that it does, but the setting is effectively buried. To find it, we need to learn about configurations.

ADF Business Component Configurations

In logical terms, ADF Business Components uses two distinct sets of metadata:

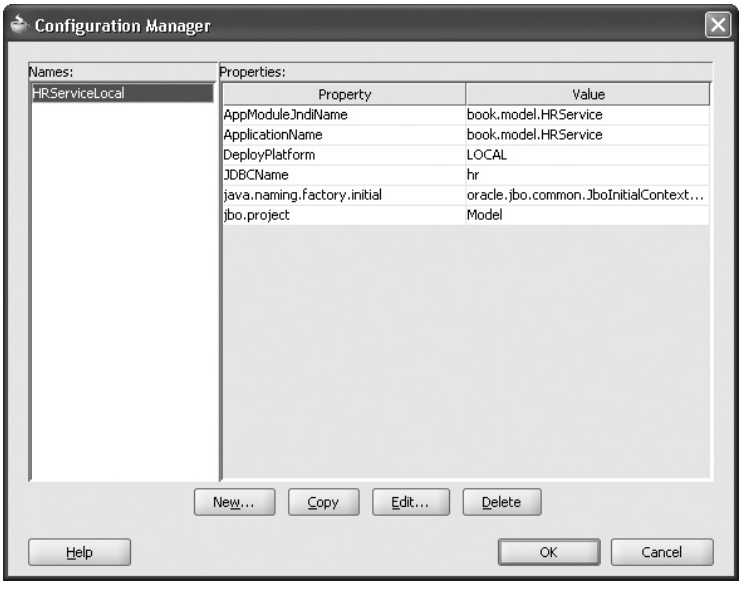
- **XML object definitions** are created in the design time environment. These files define the essential properties of artifacts, such as entity objects and view objects.
- **Configurations** are runtime settings for the ADF Business Components framework as a whole, defining session-level information, such as the database connection.

Configurations control the runtime behavior of the framework. An application may have several configurations defined for it, but at runtime, only one configuration can be active. When an application is installed into an application server, the configuration file will usually be customized as a post-deployment step. For example, the administrator may configure the database connection and various application-tuning parameters, such as application module pooling and failover settings. The configuration file differs in this way from the basic XML object definition metadata, which would not be customized after deployment.

Accessing the Configurations

The configuration file is an XML file named "bc4j.xcfg," which is visible in the Structure window for the application module. In physical terms, it is located in a subdirectory called "common" underneath the directory that stores the application module XML definition. You use the Configuration Manager to view and edit configurations. This is accessed by selecting **Configurations** from the application module right-click menu.

The first screen of the Configuration Manager shows the defined configurations and a summary of their properties—mainly the database connection information, as shown next:



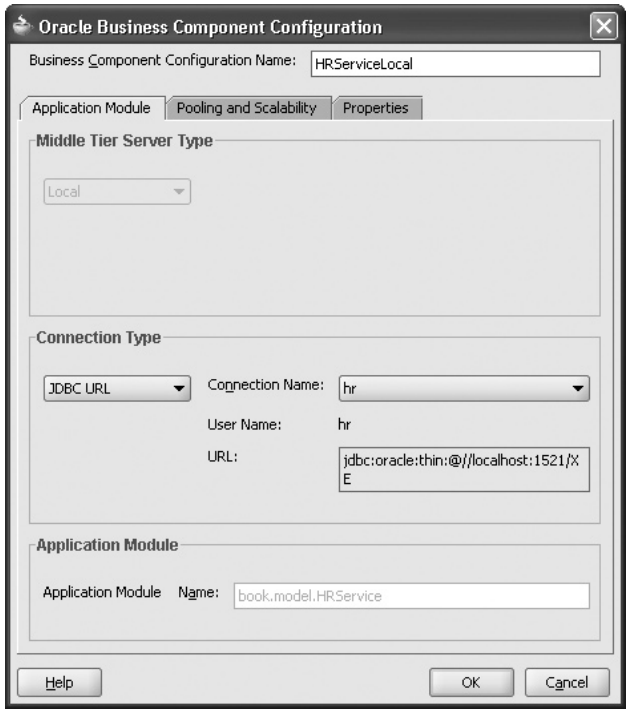
The default configuration shown here will be named with the application module name plus "Local" The word "Local" in this context indicates that this default configuration operates in local deployment mode, where the ADF Business Components module and its client are co-located in the same JVM.

Although only one configuration is created initially, you can add others with different settings. It is not unusual to have one configuration for development use, another for acceptance testing and QA, and yet another for live deployment.

TIP

Although the most obvious difference between development and deployment configurations will be the database connection, other settings may also differ. For example, it is unlikely that enabling application module pooling and failover will be useful during development. However, these features may well be needed for production deployment.

Clicking the Edit button displays the following tabbed dialog:



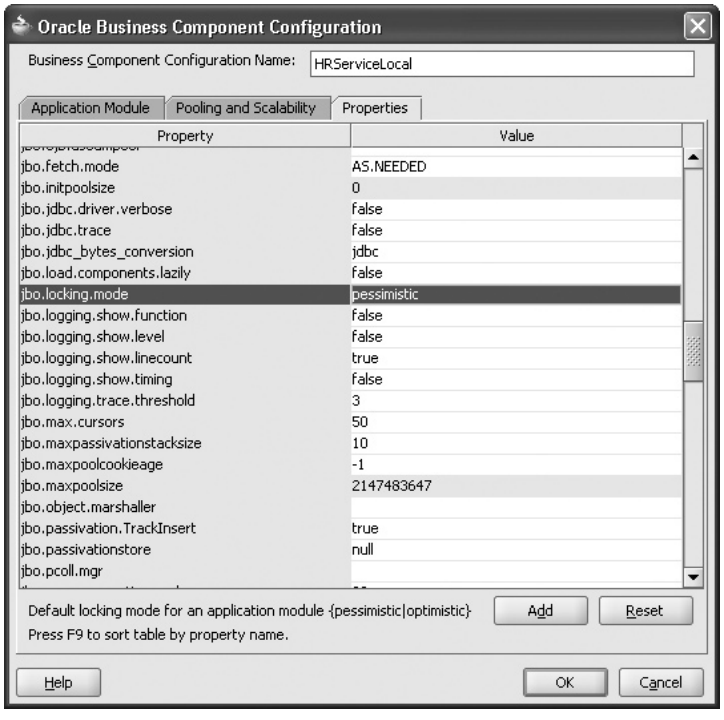
Here you can change the name of the configuration and, more importantly, define the connection information. There are two approaches to defining the connection:

- **JDBC URL** This uses a hard-coded reference to a connection as defined in the Connection Navigator.
- **JDBC DataSource** This looks the connection up from the application server at runtime. In this case, the connection information will be resolved at runtime using a Java Naming and Directory Interface (JNDI) look-up on the generic data sources defined by the application server administrator. Each servlet container uses a slightly different method of defining these data sources. In the Oracle Application Server OC4J case, data sources are defined in a file called "data-sources.xml" in the OC4J configuration directory. The administrator can set up data sources in the Oracle Application Server Control web application (also called "EM").

For design-time purposes, you will use the JDBC URL connection type. For production deployments, you will probably want the flexibility provided by the named data source. An application server administrator can configure such a data source without having to touch any of the application files.

The second tab of the Configuration Manager (Pooling and Scalability) contains information concerned with runtime tuning. However, the third tab (Properties) is of interest right now, as it

gives the ability to change the locking mode (the reason we are looking at configurations). This page provides the expert mode view of ADF Business Components configuration, including several options, most of which will never need to be changed in normal operation. The property we're looking for is *jbo.locking.mode*, shown here:



You can set locking mode to "none," "optimistic," or "pessimistic." You can only set it to OPTUPDATE programmatically. Note that "none" is valid even though it is not mentioned as a valid value in the tooltip. Also note that you need to type in this value, so be careful of spelling. Once you make that change and click OK, the Configuration Manager summary screen displays the changed property, indicating that it has been assigned a nondefault value.

Looking at the underlying XML file for the configuration—bc4j.xcfg—you will see that a new `jbo.locking.mode` entry has been added, as shown in this code listing:

```
<BC4JConfig>
  <AppModuleConfigBag>
    <AppModuleConfig name="HRServiceLocal">
      <DeployPlatform>LOCAL</DeployPlatform>
      <JDBCName>hr</JDBCName>
      <jbo.project>Model</jbo.project>
      <jbo.locking.mode>optimistic</jbo.locking.mode>
      <AppModuleJndiName>book.model.HRService</AppModuleJndiName>
      <ApplicationName>book.model.HRService</ApplicationName>
```

```
    </AppModuleConfig>
  </AppModuleConfigBag>
  <ConnectionDefinition name="hr"/>
</BC4JConfig>
```

TIP

As mentioned, the default locking mode in ADF Business Components is pessimistic. This can lead to problems in a development environment, where a lot of testing and incomplete transactions can take place on a limited set of records. To prevent frustrating locking problems, make the switch to optimistic mode in the configuration early on in the project life cycle.

Where Is the Login Dialog?

In discussing configurations, we mentioned the database connection definition, so at this point, it's probably a good idea to think about how the database login works. Many PL/SQL and Oracle Forms developers find this area rather confusing when dealing with an ADF Business Components application because they never see a login dialog.

Authenticating a user login for Oracle Forms and PL/SQL developers will nearly always mean the use of one of two methodologies. Either the user logs in with an individual database account or the user logs in as some shared database user account (usually automatically), but then has to authenticate through an application security layer, which maintains its own concept of users and roles.

Connection Pooling

J2EE applications usually take the latter approach—a shared database account. This approach helps in maximizing scalability. Most J2EE applications are designed for high transaction rates and are not necessarily stateful. That is, database transactions may be autonomous and short-lived, and the end user does not have a continuous connection to the database. Most frameworks provide maximum scalability for this scenario by using a single database account for the connection. A single account means that a pool of identical connections can be created and allocated by the application server to each session that needs to carry out a database transaction. Connections to the database are expensive to create because they require network and database server resources. This way, if multiple sessions share the same credentials, they can just borrow a connection from the pool for a short time and return it when they are done. If every session used its own user name and password, this would not be possible, and the cost of connecting to the database server would have to be incurred for every session.

As we have already seen, in ADF Business Components, the information about the database account to use is maintained in the runtime configuration file, bc4j.xcfg.

Security the J2EE Way

We have seen how best practice dictates the use of a shared database account to increase scalability. However, applications still need security. How is that achieved?

The best solution is to leverage the built-in security that is defined for J2EE—J2EE container security. This is a standard security mechanism that web-deployed applications can use to protect sets of pages within an application. We will spend more time on container-managed security in Chapter 14 as part of developing a sample application.

How Do I Define Business Rules?

So far, this chapter has covered the basics of ADF Business Components. It has tackled the core issues of both read-only and updateable queries, but it has not discussed the definition of any rules or logic associated with that basic activity. For most applications, we can consider three levels of business logic:

- **Database referential integrity constraints** These are referential integrity rules defined through relationships in the database that enforce structural aspects of the relational data itself, for example, "Each employee must be assigned to a department that actually exists."
- **Basic data content validations** This is logic, which, like the referential constraints, may be defined by hard rules defined or coded within the database (in check constraints, unique constraints, and triggers), but which checks the content of the data rather than its relationships, for example, "Salary cannot be less than zero."
- **Complex validations and rules** These are higher-level functions that perform validation across a row or between rows of data, or that implement other rules, such as setting default attribute values.

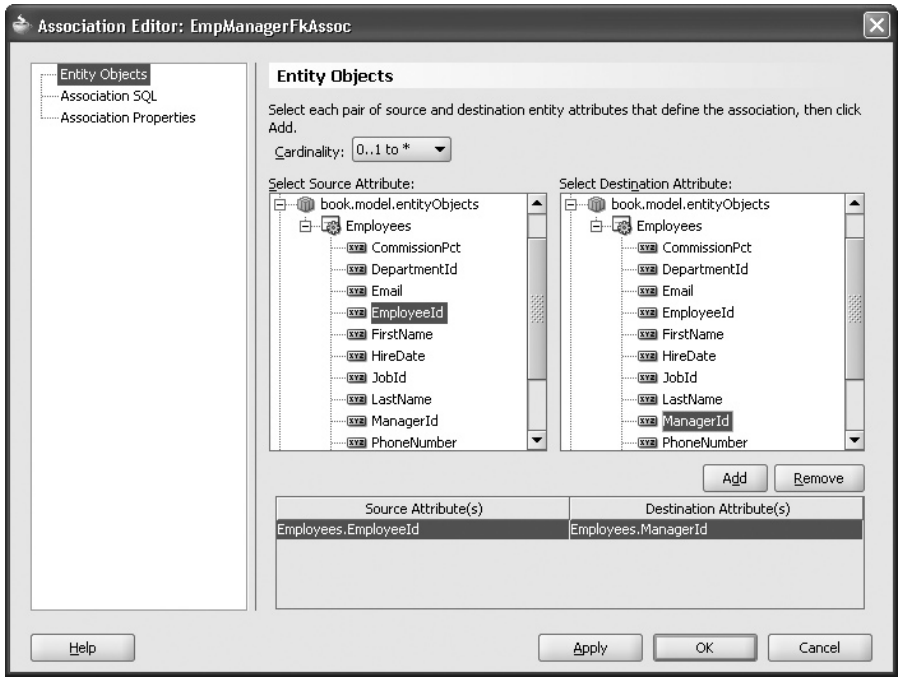
Chapter 10 examines application logic design. For now, we will look at how each of these rule categories is addressed within the ADF Business Components framework.

Database Referential Integrity Constraints

We assume that the database has been built using good relational design and that referential integrity constraints have been created in the database. At the very least, these will define the primary and foreign keys of the related tables. The database will validate these constraints automatically when rows are inserted, but it is also possible to apply the same rules in the ADF Business Components layer using associations.

Associations

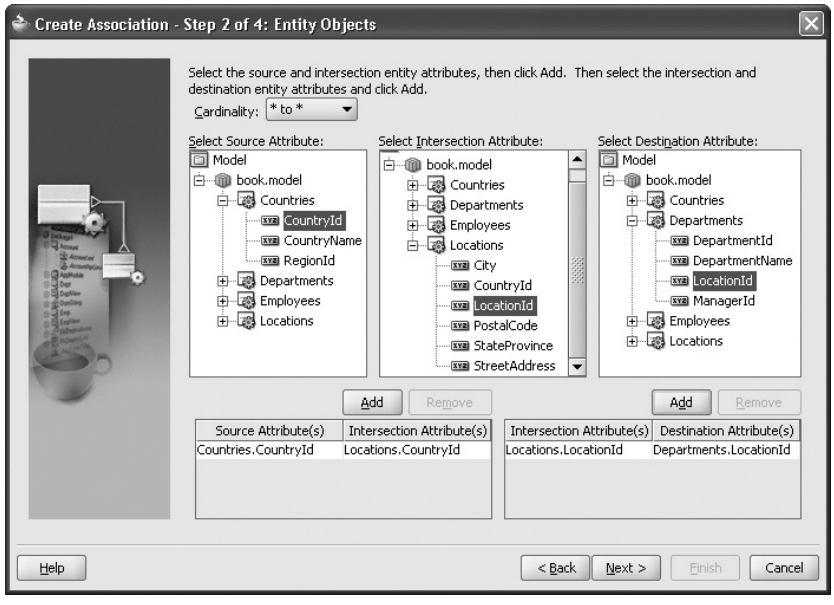
Associations define the relationship between two entity objects. If the entity objects are created using the Create Business Components from Tables Wizard, the IDE will automatically create associations based on the foreign key constraints defined in the data dictionary. For example, when creating the Employees entity object earlier in this chapter, an association called "EmpManagerFKAssoc" is created to implement the constraint enforcing that an employee's manager should be a valid employee. You can examine the association in the Association Editor (shown next) by selecting **Edit** from the right-click menu on the association node:



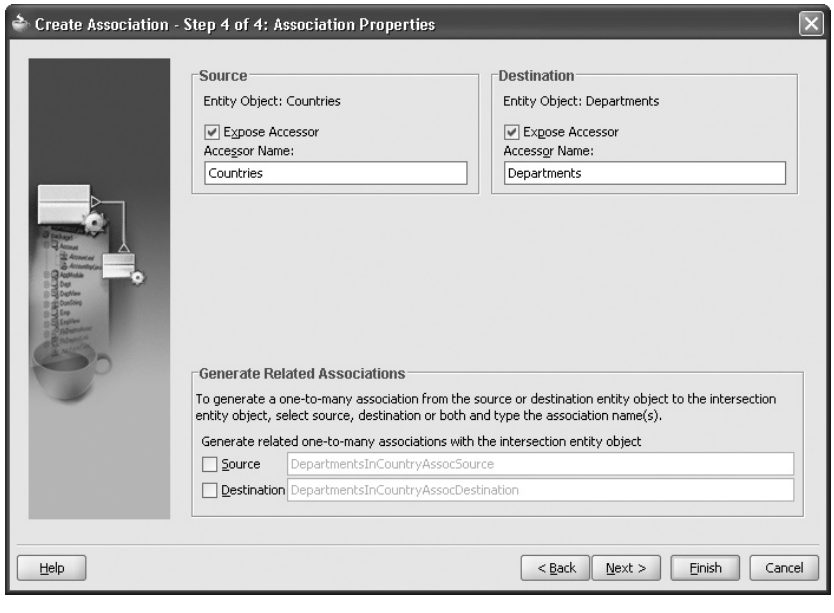
Why Use Associations? Given the information so far, it seems that an association is not giving you much more than a foreign key relationship on the database. Remember, however, that ADF BC maintains relationships between entity objects, and entity objects are detached from the database. Some of the entity object instances will have been created and cached by ADF BC and not yet committed to the database. Other entity object instances may not be derived from a table at all. For example, an entity object may be mapped to a PL/SQL collection (as we explain later). In this case, associations can enforce relationship constraints that cannot exist in the database.

In addition to being a basis for referential integrity, associations manage the process of cascading deletions and updated keys through the object tree; they also provide programmatic access in Java from the master records to the details and from the details to the master. We've already seen an implicit example of this mechanism in action with the DBSequence mechanism, where the primary key generated for the master record is propagated to all of the detail records. Programmatic access might also be useful in a scenario where a value on the master record needed to be calculated based on information from all of the children—for example, an order total attribute.

Associations are also somewhat richer than foreign key relationships on the database, because you can declare the precise cardinality of the relationship, for example, 1 to * (one-to-many), 0..1 to *(optional one-to-many), 1 to 0..1 (one-to-optional-one), and so on. An association can even define a direct * to * (many-to-many) association by mapping through a third, intersection entity, as in this example, which maps the association between Countries and Departments:



Notice the two relationships declared in the bottom panes. These many-to-many relationships are useful from a programmatic point of view. When an association is created, it will (by default) expose an accessor method in the Impl file for the either or both entity objects. This accessor method will allow you to follow the relationship to access the detail (or conversely the master) entity object instance in code should you need to. You define these accessors on the Association Properties page of the Association Editor, shown here:

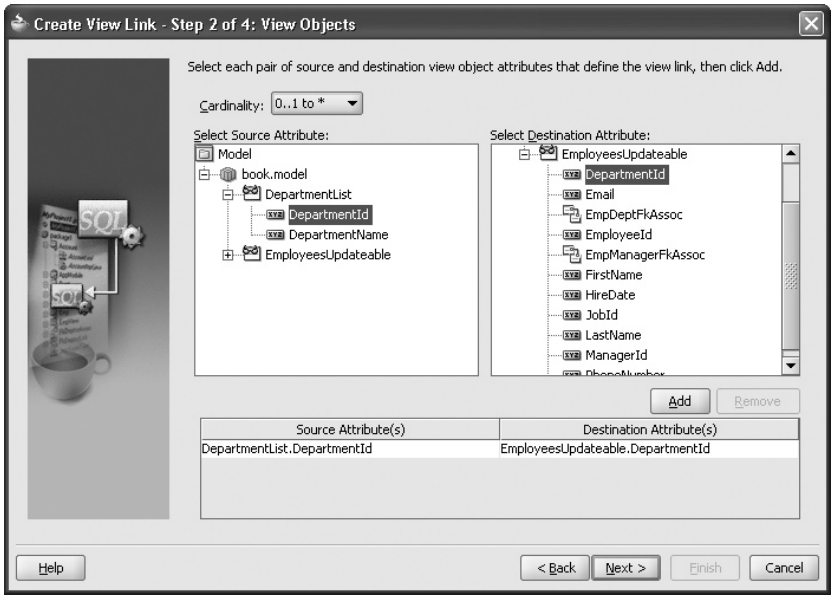


View Links

Associations define the relationship between two entity objects and its cardinality. However, they don't address the issue of querying related data to answer a question like, "Can you give me a list of all employees within a certain department?"

Writing code to traverse a list of departments and view the employees in each one is not a scalable approach. Oracle Forms already has a solution to this—the relation object. A *relation* is a declarative join between two data blocks so that the contents of the detail block are automatically synchronized with the current row in the master block (using triggers created when you define the relation). In fact, as well as providing this function, the relation also handles cascading deletions much like an association.

Not surprisingly, ADF Business Components also has a relation-like construct: the view link. Just like the association, the *view link* defines a relationship between objects, in this case, between view objects rather than entity objects. When creating a view link, you select the master and detail view objects and the joining columns in the Create View Link Wizard, shown here:



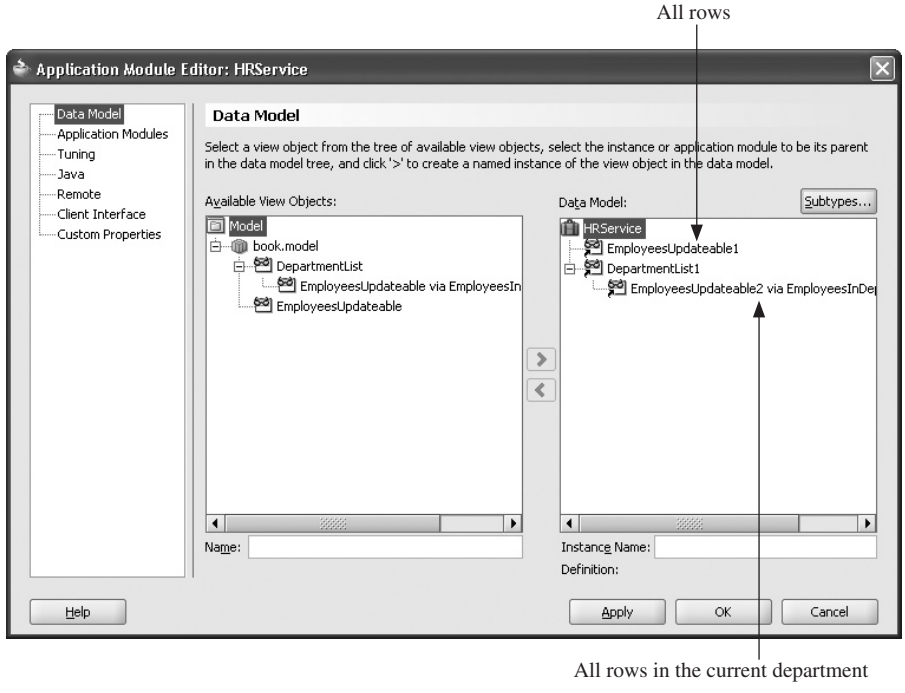
View links provide a way to access the contents of a view object in a filtered way. For example, in the case of Departments and Employees, the Employees view object may be accessed as is, that is, to retrieve the entire list of employees based on the view object's WHERE clause. Alternatively, it may be accessed through a view link from the Departments view object. In this case, the WHERE clause of the EmployeesUpdateable view object will be refined to restrict the results to employees within the currently selected row in the DepartmentList view object. Therefore, we have two possible usages of the EmployeesUpdeatable view object.

At runtime, if a view object is accessed through a view link, the relationship defined between the two view objects is applied in the form of an additional WHERE clause predicate for the detail. By the same token, if the detail view object is updateable, then any rows inserted through it will automatically contain the correct foreign key value to reference the selected master record.

This is a powerful feature that you should leverage rather than trying to maintain key relationships manually in your own code.

If you are following along with creating the Business Components project discussed in this chapter, create a view link between EmployeesUpdateable and DepartmentList, as shown previously, and call it "EmpDeptFkLink." We'll be using this in Chapters 7 and 8 when we discuss binding.

View Links in the Application Module Once view links are in the picture, we have to revise our idea of what is going on when we expose view objects through the "Application Module Data Model" screen. This screen is all about defining instances of view objects to expose. Where there are view links, you can see the master-detail relationships through the nesting of the view objects in the "Available View Objects" panel of the dialog:



At the bottom of the Data Model page is a field you can use to rename the view object instances. It is a good idea to use this field to provide a name that describes the query. The default names are somewhat meaningless, as you can see in this illustration. The EmployeesUpdateable view object instances could be renamed to *AllEmployees* (for the unbound query) and *EmployeesByDepartment* (for the employees in the current department) to make their function clearer. These names will identify queries exposed through the Data Control Palette, which you will use to construct a UI for this data.

Data Content Validations

Now we need to discuss validation at the attribute or field level—this covers the kinds of validation that you might define within the database using check constraints and possibly database triggers as well.

Constraints

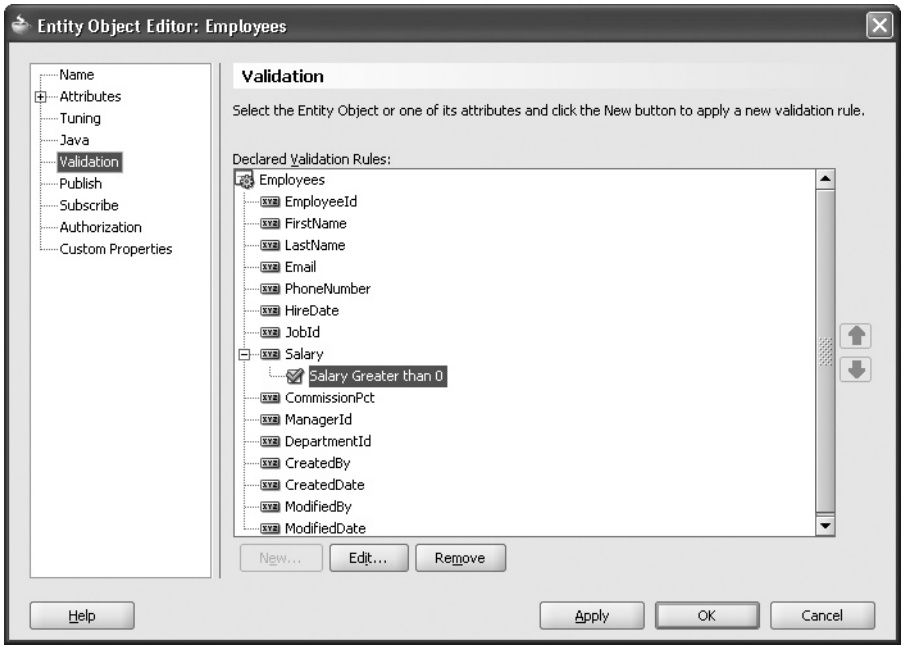
We have seen that associations and view objects provide a way of defining relational constraints within ADF BC, but in database terms, constraints can also act on the content of data. The most familiar manifestation of this is the NOT NULL constraint applied to a column when it is created. Check constraints are common, too, as a way of performing basic validation against constant values. For example, the SALARY column in the HR schema EMPLOYEES table has a check constraint to ensure that the salary value is greater than zero.

When discussing the creation of entity objects, we mentioned how ADF BC uses the data dictionary to intelligently select default properties, such as the primary key for entity object attributes. Once you have created an entity object in this way, if you look in the Structure window, you will see the following top-level nodes: Sources, Attributes, Association Accessors, and Constraints. The Constraints node shows a list of all of the rules that have been extracted from the data dictionary for this entity object. You might be forgiven for thinking that these would therefore implement your validation for these simple conditions (such as the aforementioned check on SALARY). But it's another trap—just like the sequence name mentioned in relation to the DBSequence domain that we looked at when generating primary keys. The constraints defined here in the entity object are only used if you generate a table from the entity object.

To implement basic validation rules and constraint equivalents in the entity object, you actually use declarative validation rules.

Declarative Validation Rules

You can define one or more declarative validation rules in ADF BC for each attribute in an entity object. *Declarative validation rules* are simple validations declared in the Entity Object Editor (accessed by double clicking the entity object in the Navigator). The Validation page of this editor will display a list of attributes with any declarative validation rules you declare, as shown here:



To add a declarative validation rule for a particular attribute (such as Salary) in the Entity Object Editor, select it in the list and click New.

The Add Validation Rule For dialog (shown in Figure 6-5) appears. You can use this to select from the following range of predefined validations:

- **Compare Validator** Allows you to compare (using operations such as equals, greater than, and so on) the new value for an attribute with a single value of various types: a literal value, the result of a SQL query, or the value of a view object attribute.
- **List Validator** Like the compare validator, this allows you to use literals, the results of queries, or other view objects, but this validator compares with a set of values rather than with a single value. The operator can be either In or Not In as required. This validator can act much like the *Use LOV for Validation* flag on an Oracle Forms item.
- **Range Validator** Provides a Between or Not Between comparison with high and low literal values.
- **Length Validator** Checks the length of the value. You can base the check on the number of characters or the number of bytes.
- **Regular Expression Validator** Gives an immense amount of power within a declarative context. It allows you to use regular expressions to validate formatted strings, such as email addresses, URLs, or postal codes.
- **Method Validator** If all else fails, you can write your own code as a custom method. This method has access to the other attributes within this row (and potentially across the entire row set). Unlike the other validators, a method validator is not fully declarative, because you write your own validation code. This means that you have the freedom to create a complex validator, which can be reused throughout the application. You create the method in the entity object Impl class with a `boolean` return and an argument of the attribute for which this rule is defined.

A particular attribute can have multiple declarative validation rules defined for it. The rules are evaluated in the order of definition. The actual rules, with the exception of the method validators, are stored inside the XML definition file for the entity object. No code is generated for them.

As shown in Figure 6-5, each rule can also have an associated error message, defined through the rules dialog. These messages are stored in a message bundle .java file (described and demonstrated in Chapter 9) that is listed as a child of the entity object in the Structure window. The advantage of storing the messages in a separate class rather than putting the message into the XML file is that you can create files for multiple translations of the message bundle class.

Complex Validations and Rules

We have seen how the method validator allows the creation of more complex rules in Java code. In most cases, this will be sufficient for single attribute validations, but as we discussed at the start of this section, applications generally have more complex requirements than this, such as validation across sets of attributes or records. To handle such requirements, custom code needs to be added to the entity object Impl file.

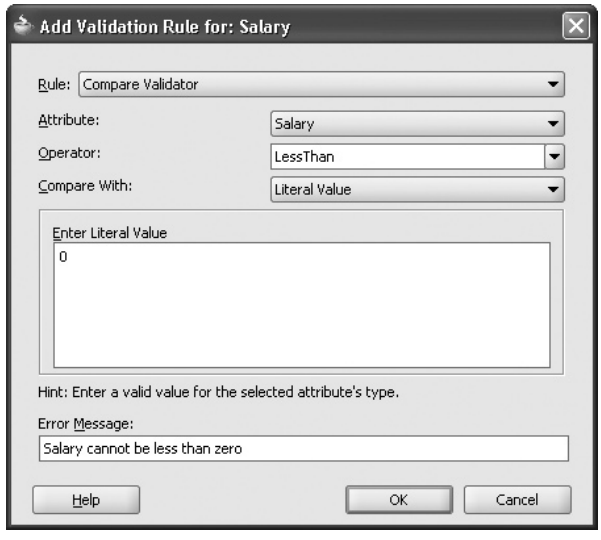
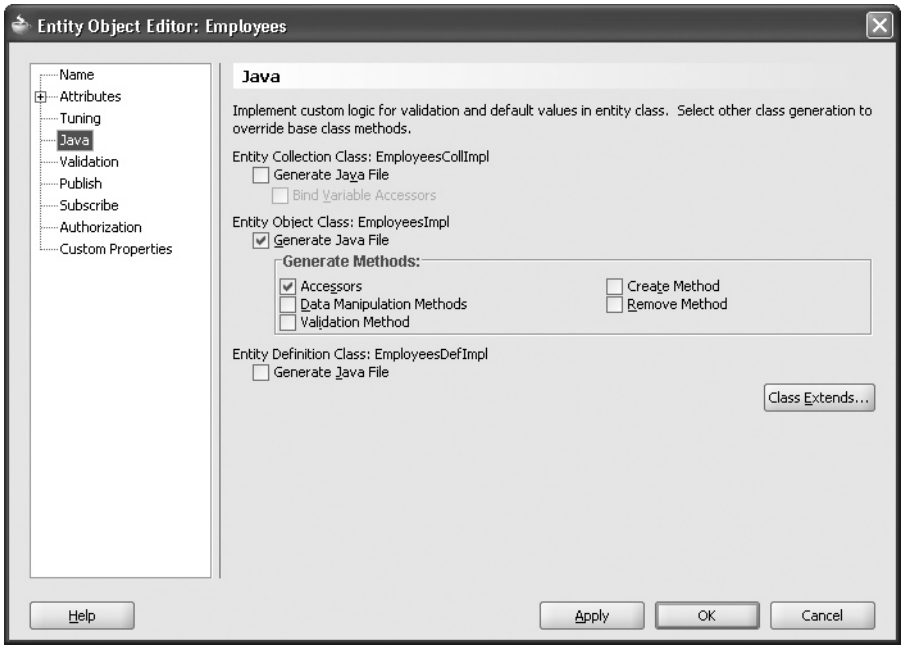


FIGURE 6-5. *Creating a validation rule in the Add Validation Rule For dialog*

Just as in the case of the application module, the Impl file for an entity object is usually created automatically. If not, you can force its creation from the Entity Object Editor's Java page, shown next.



Notice that the "Generate Java File" selection for the entity object class offers several options. Selecting each of these will generate extra code stubs (or trigger points) into the Impl .java file:

- **Accessors** This generates get and set methods for each attribute. For example, it will generate a `getFirstName()` and a `setFirstName()` method for the *FirstName* attribute. You can write code to each of these to supplement, or occasionally replace, the default functionality.
- **Validation Method** In Oracle Forms terms, this is a WHEN-VALIDATE-RECORD trigger. If this option is selected, a `validateEntity()` method will be created in the Impl file. This method executes once per entity object that is updated or created, allowing you to define validations that operate across all the attributes.
- **Create Method** The `create()` method is not a place for validation, but it is a place for initialization code for a new entity object instance (row). This is ideal for setting the default values of dynamic data elements, such as timestamps and user information, that cannot be set as static values in the attribute definitions. This is similar to the Oracle Forms WHEN-NEW-RECORD-INSTANCE trigger. See the sidebar "Using the create() Method for Key Generation" for an example of this.
- **Remove Method** The `remove()` method is triggered as the entity is marked as deleted from the entity object cache. This is more or less equivalent to the WHEN-REMOVE-RECORD trigger in an Oracle Forms application.
- **Data Manipulation Methods** This checkbox will generate a `lock()` method and a `doDML()` method. These act just like the Oracle Forms "ON-" transactional triggers (for example, ON-LOCK, ON-INSERT, ON-DELETE, and so on), and you can use them to override default entity object behavior. Although these methods are not normally needed, they do provide a way of basing an entity object on a PL/SQL package rather than on a table, as we will see later.

These options provide a rich set of extension points for the basic entity object, allowing you to customize the default functionality.

CAUTION
Just as checking one of the Generate Methods options in the Entity Object Editor creates method stubs for you, unchecking them will delete the methods, along with any code that you've added inside of them. As a precaution, it is wise to put any custom code into your own user-named methods and call those from the generated stubs. In this way, should you inadvertently uncheck an option, the damage will be minimal.

Evaluation Order of Validations

It is important to know in what order these validations will fire. Consider an entity object, which has a `validateEntity()` method, a declarative validation on an attribute, and some custom validation code in the setter for that same attribute. In what order will these validations occur?

Using the create( ) Method for Key Generation

Earlier in the chapter, we saw how you can use a database trigger and the DBSequence domain to manage sequence number allocation into entity primary keys. At that point, it was also mentioned that you could use code to achieve the same thing. The `create()` method is the place to do this. It is such a common operation that ADF BC provides a helper class to simplify the code. Here is the typical code to carry out the operation:

```
01: protected void create(AttributeList attributeList)
02: {
03:   super.create(attributeList);
04:   DBTransaction trans = getDBTransaction();
05:   SequenceImpl seq = new SequenceImpl("EMPLOYEES_SEQ",trans);
06:   setEmployeeId(seq.getSequenceNumber());
07: }
```

- **Line 03** calls the entity object superclass to create the new row.
- **Line 05** creates a new `oracle.jbo.server.SequenceImpl` object, passing the name of the database sequence to use and a reference to the `DBTransaction` obtained in **Line 04**.
- **Line 06** calls the setter for the key column on the new row (`setEmployeeId()`), passing the next sequence number for the key obtained by calling the `getSequenceNumber()` method on the `SequenceImpl`.

Unlike the DBSequence technique discussed earlier, allocating a sequence number in this way does not require a database trigger to manage the allocation of the new key. The key is also allocated as soon as the new entity object instance (row) is created, so any user interface item bound to the key column will reflect the new value before the record is committed. However, the side effect of this is that key wastage increases as users create and then abandon new records.

The entity object `validateEntity()` method will always execute at the end. But the execution order of the other two depends on how the setter method has been defined. Here is a sample setter method:

```
public void setFirstName(String value)
{
  setAttributeInternal(FIRSTNAME, value);
  checkCustomRule(value);
}
```

In this case, the `checkCustomRule()` method will perform additional validation and may fail by raising an exception. The declarative validation rule will fire before `checkCustomRule()` because the `setAttributeInternal()` call within the setter is the application point for the declarative rules. Therefore, if `checkCustomRule()` was placed before

the `setAttributeInternal()` call, it would fire first. The implication here is that you can define custom validation in the setter, both before and after the declarative rules are applied.

Raising Errors in Code

We've looked at how code can be added to define validation rules in various places. The question then arises about how to generate error conditions if something goes wrong. In the case of a method validator, the method just needs to return "false" to indicate that all is not well. The `validateEntity()` and `lock()` method signatures define no return type, so something else is needed. In these cases, the solution is to raise an exception—specifically, an `oracle.jbo.JboException`. This will signal to the framework that something has gone wrong, and the framework can pass a suitable error message back. For example:

```
public void lock()
{
  if (!auditLockingChange(key, user))
  {
    throw new JboException("Unable to audit record lock");
  }
  super.lock();
}
```

In this example, if the audit activity fails, the locking process is aborted by throwing a JboException. From the Oracle Forms perspective, this is much like coding RAISE FORM_TRIGGER_FAILURE to abort a trigger.

How Can I Dynamically Change a Query?

We've seen how view objects can be created to define multiple queries of records cached in the entity objects, but we know that fixed queries will not suffice. Most applications require functions, which need to be customized at runtime, for example, a search function where the results are based on the criteria specified through the UI.

TIP

The rest of this chapter requires writing Java code to interface with the ADF Business Components APIs. You can find more information about these APIs in the JDeveloper help system. Search the Contents tab for the topic "Reference\ Oracle ADF Business Components Java API Reference."

Fortunately, you can customize the view object to accommodate this requirement. As you saw earlier, you can define bind variables for substitution into the statement. Like entity objects and application modules, you can also define a Java Impl file for the view object as a place to put your own code for functions such as rewriting the WHERE and ORDER BY clauses.

Here is an example method, which sets the value of the `:location` bind variable in the DepartmentList view object (DepartmentListImpl.java) we created earlier:

```
public void searchByLocationId(Number locationId)
{
  setNamedWhereClauseParam("location",locationId);
}
```

In this example, the code just sets the value of the `:location` named bind variable to whatever value is passed into the method. Here is another example for the same file that will toggle the sort order for DEPARTMENT_NAME in the view object between ASC and DESC:

```
public void toggleOrdering()
{
  String orderBy = getOrderByClause();
  StringBuffer newOrderBy = new StringBuffer("Depts.DEPARTMENT_NAME ");

  //Append the correct new order indicator
  if (orderBy.endsWith(" ASC"))
  {
    newOrderBy.append("DESC");
  }
  else
  {
    newOrderBy.append("ASC");
  }

  //Set the new order by
  setOrderByClause(newOrderBy.toString());
}
```

TIP

If you're not sure what methods are available within a particular context, press CTRL-SPACEBAR to display a list. Press F1 after selecting a variable or method to display the relevant Javadoc. Finally, use CTRL- - (the CTRL and minus keys) to display the "Go To Java Class" dialog. This dialog provides a search facility to locate a class, even if you only know part of its name. You can then view the code or the Javadoc for that class.

In both of these examples, the code alters the view object definition but does not re-execute the query to apply those changes. The alterations will be used the next time ADF BC internally executes the view object. You can explicitly force a view object to requery using the `executeQuery()` method. For example, you might add a method, `toggleDepartmentSorting()`, to the application module Impl file to call the `toggleOrdering()` method defined previously and then re-execute the query to obtain the new order. Here is a sample call (the line numbers are to aid explanation and do not appear in the code):

```
01: public void toggleDepartmentSorting()
02: {
03:   DepartmentListImpl deptVO = getDepartmentList1();
04:   deptVO.toggleOrdering();
05:   deptVO.executeQuery();
06: }
```

- **Line 03** calls a method `getDepartmentList1()`. This method is one that ADF BC automatically generates when a view object (in this case, DepartmentList) is exposed in the data model of the application module. Recall the discussion earlier about the default names generated for view object instances on the Data Model page of the Application Module Editor. The `findViewObject()` function is provided by the application module to look up a view object instance by name, which is then cast to the appropriate implementation type of DepartmentListImpl.

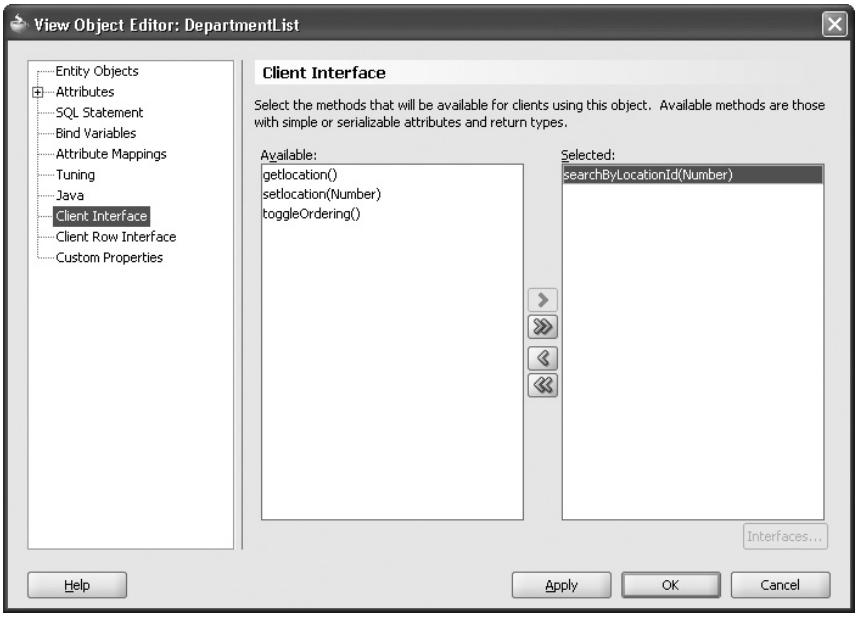
  The generated function to access the view object instance is as follows:

```
public DepartmentListImpl getDepartmentList1() {
  return (DepartmentListImpl)findViewObject("DepartmentList1");
}
```

- **Line 04** calls the `toggleOrdering()` custom method on the view object Impl file.
- **Line 05** calls `executeQuery()` to refresh the data in the view object, and in this case re-sort the rows according to the new ORDER BY.

Exposing View Object Methods on the Façade

Earlier in this chapter, we discussed how methods defined in the application module can be made available to clients using the Client Interface page of the Application Module Editor. We've also just looked at an example method (`toggleDepartmentSorting()`) that is exposed in this way. View object methods can be directly exposed to UI clients as well. This is done using the Client Interface page of the View Object Editor, as shown here:



In this case, the `searchByLocationId()` function that we defined at the beginning of this section is published as a view object method.

Such methods are shown as custom operations in the Data Control Palette for that view object instance, provided that the view object itself is exposed for client use. In Chapter 8, we'll look at how such methods can be used in your application code. For example, after exposing the `searchByLocationId()` method, you will be able to drop it into the user interface as a button and pass a new location to customize the query.

NOTE
Configuring bind variables for a view object at runtime is such a frequent operation that the ADF framework provides a shortcut called ExecuteWithParams, saving you from frequently having to write code to manage the task. We also discuss ExecuteWithParams in Chapter 8.

How Can I Interface ADF BC with PL/SQL?

As the final topic in this chapter, we will look at a key requirement: how do you call PL/SQL from ADF BC? We'll break this question down into three different usages of PL/SQL:

- **Calling a stored PL/SQL procedure** as a standalone operation, for example, from validation code.
- **Returning data from PL/SQL to ADF BC** as a variation on the preceding technique. You can use this technique for PL/SQL functions that return a value or for PL/SQL functions or procedures that contain OUT (or IN OUT) parameters.
- **Basing an entity object on PL/SQL** for DML operations—similar to basing an Oracle Forms data block on a stored package rather than directly on a table.

Calling a Stored PL/SQL Procedure

The `Transaction` object introduced earlier in the chapter provides the hooks we need to execute a PL/SQL procedure or function or, for that matter, any SQL statement. This technique will use a JDBC construct called a *prepared statement* (`java.sql.PreparedStatement`), an object into which you define a PL/SQL block or SQL statement; you then execute the statement within the context of the `Transaction` object. The following sections step through the code required to call a PL/SQL procedure that has the following signature:

```
PROCEDURE update_department_name(
            p_department_id IN PLS_INTEGER,
            p_new_dept_name  IN VARCHAR2)
```

And here is the Java code that will be needed to call it (line numbers for reference purposes only):

```
01: import java.sql.PreparedStatement;
02: import java.sql.SQLException;
03: import oracle.jbo.CSMessageBundle;
04: import oracle.jbo.SQLStmtException;
05:
```

```
06: // other application module Impl code omitted
07:
08: public void callUpdateDepartmentName (int deptNo,
09:                                       String newName)
10: {
11:   PreparedStatement plsqlBlock = null;
12:   String statement = "BEGIN update_department_name(:1,:2); END;";
13:   plsqlBlock = getDBTransaction().createPreparedStatement(statement,0);
14:   try
15:   {
16:     plsqlBlock.setInt(1,deptNo);
17:     plsqlBlock.setString(2,newName);
18:     plsqlBlock.execute();
19:   }
20:   catch (SQLException sqlException)
21:   {
22:     throw new SQLStmtException(CSMessageBundle.class,
23:                                CSMessageBundle.EXC_SQL_EXECUTE_COMMAND,
24:                                statement,
25:                                sqlException);
26:   }
27:   finally
28:   {
29:     try
30:     {
31:       plsqlBlock.close();
32:     }
33:     catch (SQLException e)
34:     {
35:       // We don't really care if this fails, so just print to the console
36:       e.printStackTrace();
37:     }
38:   }
39: }
```

1. Define the Java Method (Lines 08-09) The previous sample code represents an extract from an application module Impl file. The entire code for the class is not included, just the salient points for the task of calling PL/SQL. The interesting section begins with the declaration of a method that will encapsulate the PL/SQL call. Such methods are generally referred to as *wrapper methods* since they wrap a convenient interface around something more complex—in this case, a call out to PL/SQL:

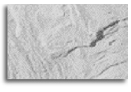
- **Line 08** defines the name of the Java method that contains the PL/SQL call as `callUpdateDepartmentName()`.
- **Lines 08-09** specify the two arguments, `deptNo` and `newName`, that will be passed through to the PL/SQL function.

In addition to this method declaration, the application module Impl will have to contain various import statements (as shown by **Lines 01-04**) to make all of the required classes available in code.

JDeveloper will pop up hints as to when import statements are needed. You use the ALT-ENTER keyboard shortcut to add these imports as you go along.

2. Create the Prepared Statement (Lines 11-13) The first part of the PL/SQL wrapper method creates a prepared statement:

- **Line 11** declares a variable (`plsqlBlock`) to hold the `PreparedStatement` object that will be created.
- **Line 12** defines the text of the actual PL/SQL call. The statement is written as an anonymous PL/SQL block with BEGIN and END keywords. Parameters that will be passed to the PL/SQL procedure are represented by the bind variable references :1 and :2.



CAUTION
The PreparedStatement syntax expects bind placeholder variables to be in this numbered style and based on a one-indexed scheme (that starts at :1 not :0). Do not confuse this with the Oracle Number style of bind variables used for view object queries, which are zero-indexed. The style of bind variable placeholders in prepared statements cannot be changed.

NOTE
If you were calling a packaged procedure, you would use the package_name.procedure_name syntax within the prepared statement just as in normal PL/SQL.

- **Line 13** creates the prepared statement and assigns it to the `plsqlBlock` variable defined in line 11. The transaction object that we discussed earlier in relation to locking and commits is responsible for creating the `PreparedStatement` object. It uses the method `createPreparedStatement()`, passing a String representation of the PL/SQL block (defined in line 14) as the first argument.

  The second argument to `createPreparedStatement()` is only used when the statement being executed is a query that returns values. It can be left as zero (indicating no return values) in this case. The technique for calling PL/SQL functions with return values is discussed later on in this section.

3. Set the Arguments (Lines 16-17) Now that the statement is prepared, the parameter values to pass to the PL/SQL procedure are defined:

- **Line 16** calls a method `setInt()` on the prepared statement. The first argument to `setInt()` indicates the index number of the placeholder bind variable that is being set with this call. In this case, the code is setting the first parameter to the PL/SQL procedure— `p_department_id`. The second argument to `setInt()` is the actual value to substitute into the `p_department_id` parameter. The underlying PL/SQL type for the parameter is PLS_INTEGER, so using `setInt()`, which expects a Java `int` type, ensures that only a valid value can be passed through to PL/SQL.
- **Line 17** calls `setString()` to set the value of the second PL/SQL procedure argument, which is of type VARCHAR2.

The `PreparedStatement` interface defines a whole set of these type-safe methods for converting from Java types to the equivalent SQL types. Use CTRL- - (Ctrl and minus) to display the "Go To Java Class" dialog, and enter "java.sql.PreparedStatement" to view them

4. Execute the Statement (Line 18) Now that the arguments are set, you can execute the statement using the call to `execute()` shown on **Line 18**.

5. Handle Errors (Lines 14, 20-26) In the code listing, the call to set the `PreparedStatement` arguments and execution are enclosed in a Java try-catch block. If the setting of the parameters or the execution of the statement fails, the catch statement in **Line 20** passes control to **Lines 22-25**. This section of code throws a new `SQLStmtException` exception, passing the text of the offending PL/SQL block and the actual error that was raised. The references to `CSMessageBundle` in this new exception refer to the framework class `oracle.jbo.CSMessageBundle`. This class is a resource bundle class provided by ADF BC that contains strings and error constants used by the `SQLStmtException` exception.

6. Clean Up (Line 31) The prepared statement must be cleaned up when it is complete. If the statement will only be called once, you can clean up after the `execute()` call. However, if the statement will be reused several times within the lifetime of the application module, the `PreparedStatement` object can be stored as an instance attribute in the Impl class; you can then clean up in the application module's `remove()` method. The clean-up code just calls `close()` on the `PreparedStatement`, as shown on **Line 31**.

In this case, the statement is cleaned up as soon as it is finished with, so this `close()` statement is called within the `finally` clause of the `try-catch` block that you used to enclose the `execute()`. Putting the `close()` call in the `finally` block ensures that the statement will be cleaned up, even if the `execute()` statement raised an exception.

7. Run the Code In Step 1, the wrapper function for the PL/SQL method was declared. It can be run from Java passing the required arguments:

```
callUpdateDepartmentName(99,"Offshore Telesales");
```

This Java code can now be called internally from any code in the application module or through the service façade, if the method is exposed through the Client Interface panel in the application module properties editor.

Returning Data from PL/SQL to ADF BC

The previous example called a procedure with no return value. What if you want to call a PL/SQL function or a procedure with IN OUT or OUT parameters? The procedure is essentially the same, except you use a `CallableStatement` rather than a `PreparedStatement`; also, you need to define variables for the return values.

For example, let's rework the same PL/SQL to be a function with a BOOLEAN return value to indicate success or failure, as in the following signature:

```
FUNCTION update_department_name(
           p_department_id IN NUMBER,
           p_new_dept_name  IN VARCHAR2)
           RETURN BOOLEAN
```

Here are the changed and extra steps to call this function from Java:

1. Create a CallableStatement Like the prepared statement, the `CallableStatement` object is also created by a call to the `DBTransaction` object. This time, an extra bind variable is added to the PL/SQL block to hold the return value from the function call, as shown here:

```
CallableStatement plsqlBlock = null;
String statement = "BEGIN :1 = update_department_name(:2,:3); END;";
plsqlBlock = getDBTransaction().createCallableStatement(statement,0);
```

2. Register the OUT Parameter With a `CallableStatement`, the `registerOutParameter()` method is used to indicate which of the bind variables will return a value and the variable's datatype. This applies to both the return value of a function and the OUT or IN OUT parameters in the PL/SQL signature:

```
plsqlBlock.registerOutParameter(1,OracleTypes.BOOLEAN);
```

In this case, the first bind variable in the statement (`:1`) will hold the result of the function call, and its type is BOOLEAN. The types of the variables are defined in the class `oracle.jdbc.driver.OracleTypes` when using the Oracle JDBC drivers. (As before, you can review all possible datatypes using the Javadoc for that class.)

3. Set the Arguments and Executing The arguments for a `CallableStatement` are set in the same way as for a `PreparedStatement`; the only difference is the numbering of the bind variables to account for the first return parameter, as shown here:

```
plsqlBlock.setInt(2, deptNo);
plsqlBlock.setString(3,newName);
```

Likewise, execution is identical, as in this call:

```
plsqlBlock.execute();
```

4. Get the Result Back The `CallableStatement` offers type-safe getters to retrieve the values of those return bind variables registered earlier. Therefore, we make this call:

```
boolean result = plsqlBlock.getBoolean(1);
```

5. Put It All Together The following is a version of the function excluding the error handling:

```
public boolean callUpdateDepartmentName (int deptNo,
                                         String newName)
{
  boolean result = false;
  CallableStatement plsqlBlock = null;
  String statement = "BEGIN :1 = update_department_name(:2,:3); END;";
  plsqlBlock = getDBTransaction().createCallableStatement(statement,0);
  try
  {
    plsqlBlock.registerOutParameter(1,OracleTypes.BOOLEAN);
    plsqlBlock.setInt(2,deptNo);
    plsqlBlock.setString(3,newName);
```

```
    plsqlBlock.execute();
    result = plsqlBlock.getBoolean;
  }
  //catch and finally blocks omitted

  return result;
}
```

Basing a Entity Object on PL/SQL

Oracle Forms programmers are familiar with the technique of combining a database view for retrieving data and a PL/SQL package to handle inserts, updates, deletions, and locks. This method simplifies the coding required in the form because the data block is based on a single data source (the view); the use of PL/SQL as the DML interface affords a lot of flexibility to manipulate and validate the data changes. You have the following two choices when using PL/SQL as a data source:

- **Base the business components on a database view with INSTEAD OF triggers.** Using this technique, you need to code the INSTEAD OF trigger to perform all logic that you want to occur for INSERT, UPDATE, and DELETE operations. The view serves as the query object. The work in ADF BC is nothing more than basing an entity object and view object on the database view. The database view can be arbitrarily complex.
- **Base the business components on a database view, and call PL/SQL procedures from ADF BC for INSERT, UPDATE, and DELETE operations.** This strategy uses the database view for the query but replaces the normal ADF BC operations for INSERT, UPDATE, and DELETE.

Since the first of these techniques requires no additional code in ADF BC, we will not explain it further. However, the second technique demonstrates how to fulfill a common requirement. In addition, if you have already defined these PL/SQL APIs, this technique will show how to leverage them from ADF Business Components. Therefore, we will explain the second one here.

We have already discussed how to call PL/SQL from within Java, and that is exactly the technique that you would use to perform the DML calls to your PL/SQL APIs. The key question is, of course, where to put the prepared statement calls that will interface with PL/SQL.

In Oracle Forms, you can use the transactional triggers mechanism to replace the default DML calls. Likewise, in ADF Business Components, you can replace the default functionality in the `doDML()` method in the entity object Impl class. Recall how this method can be generated by selecting the *Data Manipulation Methods* option in the Java panel of the Entity Object Editor.

The `doDML()` method handles INSERT, UPDATE, and DELETE. Locking is handled by the separate `lock()` method, which is also created if the Data Manipulation Methods option is selected.

Overriding Insert, Update, and Delete

Each instance of the entity object Impl file represents a single row, and the framework passes an operation argument to `doDML()` that indicates if this row operation is to be an insert, update, or delete. The `doDML()` method uses a switch/case statement to carry out code appropriate to the operation argument. This argument is passed as an integer value that corresponds to a constant, such as `DML_INSERT`. A typical `doDML()` method will look like this:

```
public void doDML(int operation,
                  TransactionEvent e)
{
  switch (operation)
  {
    case DML_INSERT:
    {
      plsqlProcInsert();
      break;
    }
    case DML_UPDATE:
    {
      plsqlProcUpdate();
      break;
    }
    case DML_DELETE:
    {
      plsqlProcDelete();
      break;
    }
  }
}
```

In this code, each DML operation is handled by a call to a method elsewhere in the Impl method, for example, `plsqlProcUpdate()`. These methods are coded to call PL/SQL in the same way as the techniques shown before. Because they are coded within the Impl file, they have direct access to the contents of the entity object through the appropriate getter methods. As an illustration, here is the `plsqlProcInsert()` method for a view based on the Departments table:

```
private void plsqlProcInsert()
{
  CallableStatement plsqlBlock = null;
  Integer generatedDepartmentId = null;
  String statement = "BEGIN deptv_api.do_insert(:1,:2,:3,:4); END;";
  plsqlBlock = getDBTransaction().createCallableStatement(statement, 0);
  try
  {
    plsqlBlock.registerOutParameter(1, Types.INTEGER);
    plsqlBlock.setInt(1,
                      getDepartmentId().getSequenceNumber().intValue());
    plsqlBlock.setString(2, getDepartmentName());
    plsqlBlock.setInt(3, getManagerId().intValue());
    plsqlBlock.setInt(4, getLocationId().intValue());
    plsqlBlock.execute();

    // The PL/SQL will generate a new Dept Id so get it
    generatedDepartmentId = new Integer(plsqlBlock.getInt(1));

    // Reset the local value
    populateAttribute(DEPARTMENTID,
```

```
                        new DBSequence(generatedDepartmentId),
                        true, false, false);
  }

// catch and finally blocks ommitted
}
```

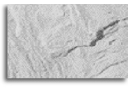
The insert method is the most interesting to look at because the corresponding PL/SQL procedure (DO_INSERT in the DEPTV_API package) declares the `p_department_id` parameter as an IN OUT variable.

```
PROCEDURE do_insert(p_department_id   IN OUT PLS_INTEGER,
                    p_department_name IN     VARCHAR2,
                    p_manager_id      IN     PLS_INTEGER,
                    p_location_id     IN     PLS_INTEGER);
```

In this example, the PL/SQL procedure allocates a new sequence number for inserted records. The new department ID then is passed back to ADF Business Components. Therefore, the first argument (bind variable) is registered as an OUT parameter. Once the statement has been executed, the key value generated by the procedure is retrieved so that the mid-tier version of the DepartmentId can be reset in the entity object using the `populateAttribute()` method.

The code required for the other methods—`plsqlProcUpdate()` and `plsqlProcDelete()`—follows the same pattern as the prepared statement example before.

A complete example of both the PL/SQL and Java code required for overriding the DML is available in JDeveloper's online help for ADF Business Components.



TIP

To find the Java and PL/SQL examples we refer to, navigate to the Help system Search tab and look for "calling stored procedures." Open the topic "Calling Stored Procedures."

Overriding the lock( ) Method

In a similar way to Oracle Forms, when the DML is being overridden, the programmer needs to supply a lock procedure in the PL/SQL. This can be called from the entity object `lock()` method. The generated `lock()` method will look like this:

```
public void lock() {
  super.lock();
}
```

Your implementation would replace the call to `super.lock()`, with a suitable PL/SQL call to lock the required objects for this transaction.

This chapter has covered a huge amount of ground in relation to ADF Business Components. There is, of course, much more that we could have covered, because ADF BC is a rich and mature framework. However, the material that we presented here will hopefully address many of your initial needs when using the framework, and will help you understand enough to explore its more complex aspects as your knowledge grows more sophisticated. We'll revisit ADF Business Components and show some more complex code examples in Chapter 15.