Listing 12-37. *Select Available Proxy from Table*

```
String key = (String)availableProxiesTM.getValueAt(
    availableProxiesTable.getSelectedRow(), 0);
Vector dataVector = appsHashtable.get(key);
appsTM.setDataVector(dataVector, appColumnIdentifiers);
```

Having these pre-built lists of applications for each proxy saves us from having to make a call to Oracle to get the list whenever the application user clicks on a different available proxy user. Getting data from a local `Hashtable` is extremely fast, compared to opening an Oracle connection and querying the database and handling the `ResultSet`. However, we only want to build data structures like this in memory when we know the quantity of items to be stored is small.

Selecting a User from the List

Whenever a user is selected from the drop-down `userComboBox`, we will query Oracle to find what proxy grants the user has already been given. This is done by the `userComboBox actionPerformed()` method. That method shows the `sayWaitDialog` message and executes a query in a delayed `Thread`, as we have seen before. The `run()` method of that thread is shown in part in Listing 12-38.

Listing 12-38 should look familiar. We build a `dataVector` of one-item `itemVectors` that represents our list of proxies granted to the selected user. In processing the `ResultSet`, we skip any returned rows that are not for the selected user. After we have gathered the list of proxies, we set that as the data `Vector` of the user proxies table model, `userProxiesTM`.

Listing 12-38. *Selecting a User from the List*

```
stmt = conn.createStatement();
rs = stmt.executeQuery(
    "SELECT DISTINCT p.proxy, p.client FROM ojsaadm.instance proxy users p " +
    "WHERE p.instance <> 'APVER' " +
    "AND p.proxy NOT IN ('APPVER','AVADMIN','APPSEC','OJSAADM') " +
    "ORDER BY p.proxy");
Vector dataVector = new Vector();
Vector itemVector;
while (rs.next()) {
    if (rs.getString(2).equals(userID)) {
        itemVector = new Vector();
        itemVector.add(rs.getString(1));
        dataVector.add(itemVector);
    }
}
if (rs != null)
    rs.close();
userProxiesTM.setDataVector(dataVector, columnIdentifiers);
```

Adding a Proxy to the User's List

When the Add Selected button is pressed, whatever proxy is selected in the available proxies list will be added to the list of user proxies. This is accomplished by the code shown in Listing 12-39, which comes from the `addButton actionPerformed()` method.

We get the selected value from the available proxies table and add that as an item in a new `Vector`, `itemVector`. Then we test to see whether an identical `itemVector` (proxy) already exists in the users list. If not, we add the new `itemVector` to the list.

There are a couple things to observe here. We get a reference to the current `dataVector` from the `userProxiesTM` table model. We do not replace it with a new `Vector` at any point. Then at the end we set the data `Vector` for `userProxiesTM` back to the `dataVector` we got.

Was that necessary? Couldn't we have just added the new `itemVector` to the existing `dataVector` without getting a local reference? And couldn't we avoid calling `userProxiesTM.setDataVector()` with the local reference?

The answer to the first question is no, it is not necessary. The answer to the second question is yes, we could have referred to the existing `dataVector` in place in the table model without creating a local reference, but that would have made for longer references and more code after all. The answer to the third question is probably, but we are counting on a side effect of the `setDataVector()` method—it also updates the GUI display with the new data. There are other ways to update the displayed table data, but none is so concise.

Listing 12-39. *Add Proxy to User's List*

```
Vector dataVector = userProxiesTM.getDataVector();
String value = (String)availableProxiesTM.getValueAt(
    availableProxiesTable.getSelectedRow(), 0);
Vector itemVector = new Vector();
itemVector.add(value);
if (!dataVector.contains(itemVector)) {
    dataVector.add(itemVector);
    userProxiesTM.setDataVector(dataVector, columnIdentifiers);
}
```

Removing a Proxy from the User's List

When the Remove Selected button is pressed, the operation for removing an item from the user's proxies table is relatively simple. The code is shown in Listing 12-40. Just call the `remove()` method on `dataVector`. Once again, we call the table model `setDataVector()` method to assure the GUI display is updated.

Listing 12-40. *Remove Proxy from User's List*

```
Vector dataVector = userProxiesTM.getDataVector();
String value = (String)userProxiesTM.getValueAt(
    userProxiesTable.getSelectedRow(), 0);
Vector itemVector = new Vector();
itemVector.add(value);
dataVector.remove(itemVector);
userProxiesTM.setDataVector(dataVector, columnIdentifiers);
```

Saving Updates to the User's Proxies

When the **Save Updates** button is pressed, the `saveButton actionPerformed()` method will be run. The significant code for this method is shown in Listing 12-41. This comes from the delayed thread `run()` method that is defined there.

In that thread, the first thing we do is attempt to create the user on both the `orcl` and `apver` instances. We did not need an Oracle user until we decided to grant the user access to the applications.

Then we assign applications. We get the `dataVector` from the user's proxies table model. That is our planned, resultant list of proxies for the user. Some of these may be added proxies. Some may be proxies the user had previously. There may also be proxy grants that the user had that have now been removed from this list. So we have to handle the following three cases:

- Retain existing proxies that are in the list.
- Add new proxies that are in the list.
- Remove existing proxies that are no longer in the list.

In Listing 12-41, we handle these three cases. We select the existing list of proxies from an Oracle query and walk through the `ResultSet`. We are only concerned with proxies for the selected user. For each existing proxy in the `ResultSet`, we create a new `itemVector` with that as an item. We can find if the `dataVector` has an `itemVector` like that by calling the `contains()` method. If it does, then this is a proxy we want to retain. We have dealt with that proxy, so we remove it from the `dataVector`. Later, we will see what proxies are still left in the `dataVector`—those will have to be newly granted to the user.

If the existing proxy we got from our query is not in the `dataVector`, then it was removed from the list, and we need to revoke the proxy grant. We do that by calling the `p drop proxy through` procedure.

Finally, we are left with a `dataVector` with just those proxies that did not exist previously. For each of those, we call the `p set proxy through` procedure to add a new proxy grant for the selected user.

Listing 12-41. *Save Updates to User's Proxies*

```
OracleCallableStatement stmt2 = null;
Statement stmt = null;
ResultSet rs = null;
try {
    // First, try to create user on both instances
    stmt2 =
(OracleCallableStatement)conn.prepareCall(
"CALL sys.usr role adm pkg.p_create_user_once(?)");
    stmt2.setString(1, userID);
    stmt2.executeUpdate();
    if (stmt2 != null)
        stmt2.close();
    stmt2 =
(OracleCallableStatement)conn.prepareCall(
"CALL sys.usr role adm pkg.p_create_user_many(?)");
    stmt2.setString(1, userID);
    stmt2.executeUpdate();
    if (stmt2 != null)
        stmt2.close();
    stmt2 =
(OracleCallableStatement)conn.prepareCall(
"CALL ojsaadm.apver usr adm pkg.p_create_apver_user(?)");
    stmt2.setString(1, userID);
    stmt2.executeUpdate();
    if (stmt2 != null)
        stmt2.close();
```

```
    stmt = conn.createStatement();
    rs = stmt.executeQuery(
        "SELECT DISTINCT p.proxy, p.client FROM ojsaadm.instance proxy users p " +
        "WHERE p.instance <> 'APVER' " +
        "AND p.proxy NOT IN ('APPVER','AVADMIN','APPSEC','OJSAADM') " +
        "ORDER BY p.proxy");
    Vector dataVector = userProxiesTM.getDataVector();
    Vector itemVector;
    String proxyID;
    while (rs.next()) {
        if (rs.getString(2).equals(userID)) {
            proxyID = rs.getString(1);
            itemVector = new Vector();
            itemVector.add(proxyID);
            if (dataVector.contains(itemVector)) {
                //System.out.println("retaining proxy to: " + proxyID);
                dataVector.remove(itemVector);
            } else {
                //System.out.println("removing proxy to: " + proxyID);
                stmt2 = (OracleCallableStatement)conn.prepareCall(
                    "CALL sys.usr role adm pkg.p_drop_proxy_through(?,?)");
                stmt2.setString(1, userID);
                stmt2.setString(2, proxyID);
                stmt2.executeUpdate();
                if (stmt2 != null) stmt2.close();
            }
        }
    }
    if (rs != null)
        rs.close();
    for (Object element : dataVector) {
        itemVector = (Vector)element;
        proxyID = (String)itemVector.get(0);
        //System.out.println("adding proxy to: " + proxyID);
        stmt2 = (OracleCallableStatement)conn.prepareCall(
            "CALL sys.usr role adm pkg.p_set_proxy_through(?,?)");
        stmt2.setString(1, userID);
        stmt2.setString(2, proxyID);
        stmt2.executeUpdate();
        if (stmt2 != null) stmt2.close();
    }
...
```

Application Registration Screen

Next on our Security Administration agenda, and the target of the next button on the `OJSAdmin` menu, is the Register New Application functional screen. We have a couple applications registered so far: `HRVIEW` and this application, `OJSADMIN`, if you have run some of the SQL in this chapter. You can see the list of fields required for registering a new application on Figure 12-6.



Figure 12-6. *Register New Application functional screen*

The Application ID is really just a name for the application—a way to refer to it in the database. The primary use for the application ID is as one of the keys to select the proper secure Application Role. The other key is the name of the Application User (proxy user). These three data elements are what we insert into the `appsec.v application registry` view.

At the bottom of the screen, there are three fields that make up the definition of an application inner class. The first field is the package, then the Class name, and finally the InnerClass name. We will create an instance of that class locally and submit it to Oracle in order to create a placeholder in the `appsec.v app conn registry` view for a list of connection strings for this new application. In order for that entry to be made in `appsec.v app conn registry`, a representative class needs to exist on the Oracle database.

There is an optional button on the Register New Application screen that appears if the user types "Login" into the Class field. In that case the Create App Class on Oracle From Login Template button

appears. Be sure to document this feature for other application security administrators who will be using this screen to register applications. Might I suggest buying them a copy of this book?

If the application uses the `Login` class we developed in this chapter, then creating a representative class on Oracle is easy. The only change in this inner class from one application to another is the package in which the `Login` class is located. We take advantage of that fact in this button. When this button is selected, a new Java class is created on the Oracle database.

When the user selects the Register button, we do several things. For one thing, we insert the data from the top three fields on the form into the `appsec.v application registry` view. Also, we add the current Oracle user as an administrator for this specific application. That data goes into a new table named `appsec.t application admins`. And third, we enter a value in a new table that associates entries in the `appsec.v application registry` view with application inner class entries in the `appsec.v app conn registry` view. This new table is named `appsec.t app class id`. We will discuss these new tables later in this chapter.

There is only one concern when registering a new application. We actually change context from the `OJSAdmin` application to that of the new application in order to communicate with Oracle database as that application and enter the inner class in the `appsec.v app conn registry` view. For that reason, the Close button turns into an Exit button after the Register button has been pressed, and instead of returning to `OJSAdmin`, the **Exit** button exits the JVM.

The Application Verification Administrator Role

In Chapter 11, we created a role named `appver admin`. It is to that role that we delegated some administrative tasks, chief among those being the task of executing the procedures and functions in the `appsec.appsec admin pkg` package. Those procedures and functions include `f set decrypt conns`, `p copy app conns`, and the procedure we will introduce later in this chapter, `p create template class`. These are procedures that handle our stored Oracle connection string lists for various applications, and they are due extra security measures.

In Chapter 11, we assigned the `appver admin` role to a specific user that we have called `osadmin` in our example code (probably you.) In this chapter, we are going to change that role to operate more like one of our standard application roles, being set for persons who are allowed to proxy through an Oracle application account. You might have caught a hint of this from Table 12-1 and Listing 12-5 where we associated a new user, `avadmin` with the `appver admin` role. We also saw how we can assign that proxy user to people in our discussion of the Admin Users functional screen.

Previously, we have defined the `appver admin` role on both `orcl` and `apver` instances, and we had designated it to be `NOT IDENTIFIED`, and we granted the role to `osadmin`. Here we will revoke that grant and alter the role to be identified by our secure application role procedure, `appsec.p check role access`. We do those tasks, shown in Listing 12-42, as `SYS` user on both the `orcl` and `apver` instances. At the same time we will create the new proxy user, `avadmin`, who will set this role. And we grant the right to connect through `avadmin` (proxy) to `osadmin` (you). Be sure to give a very strong password to the `avadmin` user. Also shown in Listing 12-42, on the `apver` instance we will repeat an insert into `appsec.v application registry` that we have already made on the `orcl` instance—setting up the association of the `avadmin` user and the `appver admin` role.

Listing 12-42. *Application Verification Administrator*

```
REVOKE appver admin FROM osadmin;
ALTER ROLE appver admin IDENTIFIED USING appsec.p check role access;

GRANT create session role TO avadmin IDENTIFIED BY password;
ALTER USER osadmin GRANT CONNECT THROUGH avadmin;
```

```
INSERT INTO appsec.v application registry
(application id, app user, app role) VALUES
('OJSADMIN','AVADMIN','APPVER ADMIN');
```

The Create App Class Button

When the application developer has included the `Login` class in his application, modifying the application ID and the package, he has made the registration of his application in Oracle database much easier. If he had not gone this route, then we would have to get his application inner class definition from him, and would have to execute the command `CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED`. However, since he is following the preferred GUI development path, using Login, we know enough from his package alone to create a class on the Oracle database to represent his application.

So, when we get to the Register New Application screen, and we see the word "Login" typed in the Class field, we provide a button to allow easy creation of a representative application inner class on the database. The code in Listing 12-43 is used to display or hide that button. We use a `keyReleased` event to tell when the user has entered a character in the `classTextField`. If after entering the character, the text of the field equals "Login," then we make the button visible. Otherwise, we hide it.

Listing 12-43. *Display or Hide the Create App Class Button*

```
private void classTextField keyReleased(KeyEvent e) {
    if (classTextField.getText().equals("Login"))
        createTemplateButton.setVisible(true);
    else
        createTemplateButton.setVisible(false);
}
```

When the button is visible, it can be selected and it then executes the code in the `createTemplateButton actionPerformed()` method. In that method, we find the code shown in Listing 12-44. To create the template class in the database, we call the `p create template class` procedure. That is a new procedure that we are defining in the `appsec admin pkg` package. We pass in the fully qualified class name; however, we don't read all the user-provided fields to get that. Rather we read the package name that the user provided and append the string ".Login$InnerRevLvlClass" to the package name.

Listing 12-44. *Create a Template Class in Oracle*

```
stmt = (OracleCallableStatement)conn.prepareCall(
            "call appsec.appsec_admin_pkg.p_create_template_class( ?,?,? )");
stmt.registerOutParameter(2, OracleTypes.NUMBER);
stmt.registerOutParameter(3, OracleTypes.VARCHAR);
stmt.setString(1, packageTextField.getText() + ".Login$InnerRevLvlClass");
stmt.setInt(2, 0);
stmt.setNull(3, OracleTypes.VARCHAR);
stmt.executeUpdate();
```

Listing 12-45 shows the code of the new `p create template class` procedure. The procedure needs to live on the apver database instance, so the definition is included in the script file *Chapter12/ApverSys.sql* . The bulk of this procedure is given to the definition of the Java inner class. Notice that there are two places in that inner-class definition that we insert the package name: once in the name of the Java structure and once in the package statement. We do several tests before we decide

to create the class. The first test is done to see whether the class has already been created in the database. We query all the Java classes (`sys.all java classes`) to see if one with the same name already exists. We don't create the class if that is the case. Because we are executing a `CREATE` command instead of a `CREATE OR REPLACE` command, we will get an exception if the class already exists. The second test we make will assure that the word "Login" is in the class name, and the third test assures that something is left, as a package, when we truncate the class name before the word "Login."

In the DDL command that we will `EXECUTE IMMEDIATE`, we are concatenating ASCII character codes 10, 13, and 34. These are used to give some format to our Java class. The pair of 13 and 10 are the carriage return/line feed characters that move to the beginning of the next line. The character 34 is a double quote that we want to have on both sides of our class name to set it off from the schema name, `APPSEC`.

Listing 12-45. *Procedure to Create Template Class*

```
    PROCEDURE p create template class(
        m class name     v app conn registry.class name%TYPE,
        m err no     OUT NUMBER,
        m err txt    OUT VARCHAR2 )
    IS
        v count   INTEGER;
        v package v app conn registry.class name%TYPE;
    BEGIN
        m err no := 0;
        SELECT COUNT(*) INTO v count
        FROM sys.all java classes
        WHERE owner='APPSEC' AND name = m class name;
        IF v count < 1 THEN
            v count := INSTR( m class name, 'Login' );
            IF v count > 0 THEN
                v package := SUBSTR( m class name, 0, v count - 2 );
                IF LENGTH( v package ) > 0 THEN

                    EXECUTE IMMEDIATE

'CREATE AND RESOLVE JAVA SOURCE NAMED APPSEC.' || CHR(34) ||
v_package || '/Login' || CHR(34) || ' AS ' || CHR(13) || CHR(10) ||
'package ' || v_package || '; ' || CHR(13) || CHR(10) ||
'import java.io.Serializable; ' || CHR(13) || CHR(10) ||
'import orajavsec.RevLvlClassIntfc; ' || CHR(13) || CHR(10) ||
'public class Login { ' || CHR(13) || CHR(10) ||
'    public static class InnerRevLvlClass ' || CHR(13) || CHR(10) ||
'       implements Serializable, RevLvlClassIntfc{ ' || CHR(13) || CHR(10) ||
'        private static final long serialVersionUID = 2011010100L; ' || CHR(13) ||CHR(10)||
'        private String innerClassRevLvl = "20110101a"; ' || CHR(13) || CHR(10) ||
'        public String getRevLvl() { ' || CHR(13) || CHR(10) ||
'            return innerClassRevLvl; ' || CHR(13) || CHR(10) ||
'}   }   }';
                END IF;
            END IF;
        END IF;
    EXCEPTION
        WHEN OTHERS THEN
            m err no := SQLCODE;
```

```
            m_err_txt := SQLERRM;
            app_sec_pkg.p_log_error( m_err_no, m_err_txt,
                ' p_create_template_class' );
    END p_create_template_class;
```

To make this button work, there are requirements. The owner of the `appsec_admin_pkg` package, that is the appsec user, must be able to create Java classes. For that, we have to grant the `CREATE PROCEDURE` system privilege to appsec. Also she must also be able to select from the data dictionary view, `sys.all_java_classes`, to see what classes already exist. We do this as SYS user on the apver instance:

```
GRANT CREATE PROCEDURE TO appsec;
GRANT SELECT ON sys.all_java_classes TO APPSEC;
```

Tables of Specific Application Administrators and Application to Class Registry

At some point we are going to want to delegate administration of certain applications to one person, and other applications to another person—and we will not want them to have access to administer each other's applications. That eventual goal, which we will accomplish later in this chapter, is the reason that we are going to build the `appsec.t_application_admins` table. The definition of `t_application_admins` is given in Listing 12-46. It is simply a list of OS user names that are allowed to manage specific application inner class names. It is the lists of Oracle connection strings represented by those application inner classes that we are intending to protect.

■ **Note** These tables are only needed on the apver instance.

Listing 12-46. *Application Administrators Table*

```
CREATE TABLE appsec.t_application_admins
(
    -- match appsec.t_app_conn_registry.class_name
    class_name VARCHAR2(2000) NOT NULL,
    -- match hr.emp_mobile_nos.user_id
    user_id    VARCHAR2(20) NOT NULL
);
/

CREATE UNIQUE INDEX application_admins_pk ON appsec.t_application_admins
    ( class_name, user_id );

ALTER TABLE appsec.t_application_admins ADD (
    CONSTRAINT application_admins_pk
    PRIMARY KEY
    ( class_name, user_id )
    USING INDEX application_admins_pk
```

```
);
/

CREATE OR REPLACE VIEW appsec.v application admins
    AS SELECT * FROM appsec.t application admins;

INSERT INTO appsec.v application admins
    ( class name, user id )
    ( SELECT DISTINCT class name, 'OSADMIN' FROM appsec.t app conn registry );

COMMIT;
```

We are going to get a start on listing individual administrators for various applications by initially assigning administration of all the applications to the user we are calling `osadmin`. We do that with a single insert command shown in Listing 12-46, and we commit the insert.

There is a second table that we are going to create here, the `appsec.t app class id` table. When each application was managed individually, as we have done up through Chapter 11, we automatically knew what application we were working on. However, in this security administration interface application, we are managing all applications. Now we need to symbolically associate an application with its application inner class and list of connection strings. This association is necessary in order to keep from having to type in both the application ID and the inner class name whenever we want to specify an application.

An example entry in the `t app class id` table is for this application: the class name is `orajavsec.Login$InnerRevLvlClass`, and the application id is `OJSADMIN`. Listing 12-47 shows the script we will use to build and populate the table.

Listing 12-47. *Application Class to ID Table*

```
CREATE TABLE appsec.t app class id
(
    class name    VARCHAR2(2000 BYTE) NOT NULL ENABLE,
    application id VARCHAR2(24 BYTE) NOT NULL ENABLE
);
/

CREATE UNIQUE INDEX app class id pk ON appsec.t app class id
    ( class name, application id );

ALTER TABLE appsec.t app class id ADD (
    CONSTRAINT app class id pk
    PRIMARY KEY
    ( class name, application id )
    USING INDEX app class id pk
);
/

CREATE OR REPLACE VIEW appsec.v app class id AS SELECT * FROM appsec.t app class id;

INSERT INTO appsec.v app class id
(CLASS NAME, APPLICATION ID) VALUES
('testojs.TestOracleJavaSecure$AnyNameWeWant','HRVIEW');
```

```
INSERT INTO appsec.v app class id
(CLASS NAME, APPLICATION ID) VALUES
('orajavsec.Login$InnerRevLvlClass','OJSADMIN');

COMMIT;
```

I mentioned in the introduction to this section that we insert data into these tables when we register new applications. We enter the current Oracle user as an administrator for the new application, inserting a record in the `appsec.v application admins` view. Also we enter the application class to application ID relationship by an insert into the `appsec.v app class id` view. The insert statements for our existing applications are shown in Listing 12-47.

Security Table Access Analysis

We are going to take a look at how we get to tables of data in the `appsec` schema and how we use the data. This will help us define who should get access and what access grants we need to make to each Oracle user.

First of all, we need to understand that most of the access to data in the `appsec` schema is accomplished by procedures, functions, and packages in the `appsec` schema, so no additional grants to the data are required. We have several packages that segregate procedures and functions into these groups:

- `APP SEC PKG`: Procedures and functions that applications need to access. We grant execute
  on this package to application schemas, like `HR`. Those application schemas have sensitive data access procedures that call on `app sec pkg`.
- `APPSEC ONLY PKG`: Procedures and functions that handle the inner workings of our Oracle and Java Security activities. No grants are made to this package.
- `APPSEC PUBLIC PKG`: Procedures that need to be accessible by all Oracle users. This package is granted execute to PUBLIC. Generally, this provides a jumping-off point, part of the application verification process, where users are proxying through `appver`, but have no roles except the `create session role`. One particular procedure in this package is the `p get app conns` procedure, by which a user may acquire the list of connection strings for the application they are using after SSO, two-factor authentication and key exchange. Our plan is to secure SELECTs on this particular procedure so that users can only select applications that they have been granted.
- `APPSEC ADMIN PKG`: Procedures and functions that are only available to the application security administrators, those who can proxy through the `avadmin` user and acquire the `appver admin` role. The execute privilege is granted to `appver admin` for this package.

We also have a group of procedures and functions that do not reside in packages. All those structures have one of two aspects about them that keep them from working in a package: they either execute with the current user privileges, `AUTHID CURRENT USER`, or they are obfuscated by the Oracle *wrap* utility.

We also support some direct access to the application security data from outside the structures of the `appsec` schema. In particular, we have queries in this security administration interface application that select data from three of the views: `v app class id`, `v application registry` and

v app conn registry. These queries run in functional screens where the Oracle application user is proxying through avadmin. So, we grant SELECT on those views to the appver admin role. Additionally, in the Register New Application screen, that we are examining now, we insert data into three views: v app class id, application registry and application admin. We have currently set the appver admin role, so we grant INSERT on those views to appver admin.

There will come a time when we need to select, update and delete records in the v application admin view. We need to identify people, in addition to or in place of the person who initially registered the application, to manage connection strings for each application. We delegate that job of setting application administrators to specific users. We will grant SELECT, UPDATE, and DELETE on v application admin to the osadmin user (you, perhaps.)

On the Assign Applications functional screen, we need to select records from the v application registry view. That screen is operating with the ojs adm admin role by user proxying through ojsaadm. Recall that ojsaadm has a whole set of grants and package of procedures all of his own—look up at the description of the Admin Users screen for that discussion. We grant SELECT on v application registry to the ojs adm admin role.

Listing 12-48 shows the script used to make the remaining grants needed on the apver instance. Those grants take care of all the data requirements I mentioned earlier.

Listing 12-48. *Grants to Appsec Data on Apver*

```
GRANT SELECT ON appsec.v app class id TO appver admin ;
GRANT SELECT ON appsec.v application registry TO appver admin;
GRANT SELECT ON appsec.v app conn registry TO appver admin;

GRANT INSERT ON appsec.v app class id TO appver admin ;
GRANT INSERT ON appsec.v application registry TO appver admin ;
GRANT INSERT ON appsec.v application admins TO appver admin;

GRANT UPDATE, SELECT, DELETE ON appsec.v application admins TO osadmin;
```

We still have one remaining security data access hurdle to deal with. Typically (that is up through Chapter 11), we have just embedded our passwords in the application and used the first pass through the application to set the connection strings. We don't want to have to do that! There was a reason for that which we have discussed. We need to be proxied through the appver user, as the application, in order to set the connection strings for the application. Once we call the getAAConnRole() method to get one of the application connections from the list, we can no longer update the connection strings associated with the application. I'm sure that OracleJavaSecure could be re-written to operate differently.

The issue arose when we moved some of our procedures into the appsec admin pkg package and required a secure role to get access. In effect, we have made the procedure to update our connection strings impossible for us to call. When we are connected by proxying through the appver user, we have not set a secure application role by executing the p check role access procedure, or something similar. Individual Oracle users, proxied through appver, cannot execute appsec admin pkg, even if they are granted the privilege to proxy through avadmin to acquire the appver admin role.

The solution we have is to create another role on apver instance that will be a default role for all users who need to update the connection strings for an application. Our application administrators will have both the grant to proxy through avadmin, and the grant to this new role, which we will call appver conns. The code for creating the appver conns role is shown in Listing 12-49.

Listing 12-49. *Application Verification Connections Role*

```
CREATE ROLE appver conns NOT IDENTIFIED;
GRANT EXECUTE ON appsec.appsec admin pkg TO appver conns ;
GRANT appver conns TO osadmin;
```

Notice that we grant `appver conns` to the `osadmin` user. We will make that same grant to all application administrators who will be maintaining the connection strings for their applications through this security administration interface. This gives them access to the `appsec admin pkg`, which they will need to update connection strings.

When we add administrators, we will need to grant both the privilege to connect through `avadmin`, and the grant to `appver conns` role. For this reason, we have additional procedures in a package on `apver` instance named `sys.appver conns role pkg`. The package code is given in Listing 12-50. There are two procedures, `p grant appver conns role` and `p revoke appver conns role` for both granting and revoking `appver conns`.

Listing 12-50. *Grant Appver Conns Role on Apver Instance*

```
CREATE OR REPLACE PACKAGE BODY sys.appver conns role pkg IS

    PROCEDURE p_grant_appver_conns_role (
        username sys.proxy users.client%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'GRANT appver_conns TO ' || username;
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.p grant appver conns role for ' ||
                username );
    END p grant appver conns role;

    PROCEDURE p_revoke_appver_conns_role (
        username sys.proxy users.client%TYPE )
    AS
        PRAGMA AUTONOMOUS TRANSACTION;
    BEGIN
        EXECUTE IMMEDIATE 'REVOKE appver_conns FROM ' || username;
        COMMIT;
    EXCEPTION
        WHEN OTHERS
        THEN
            appsec.app sec pkg.p log error( SQLCODE, SQLERRM,
                'sys.p revoke appver conns role for ' ||
                username );
    END p revoke appver conns role;
```

```
END appver conns role pkg;
/

-- From New Application Registration
GRANT EXECUTE ON sys.appver conns role pkg TO appver_admin ;

-- From Admin Users, grant to user, not to role
GRANT EXECUTE ON sys.appver conns role pkg TO ojsaadm;
```

When we register a new application, we will call `p grant appver conns role` for the current session user, as we will see in the next subsection of this chapter. At that point, we are proxying through the `avadmin` user, with `appver admin` role; so you see in Listing 12-50 that we grant execute on `sys.appver conns role pkg` to `appver admin`.

When we are adding administrators through the **Admin Users** screen, we want to call either `p grant appver conns role` or `p revoke appver conns role`, depending on whether the App Register check box is selected. However, at that point we are proxying through the `ojsaadm` user with `ojs adm admin` role, and we are on the wrong instance (`orcl` as opposed to `apver`.) We already have a package and a database link to solve this issue, so we will create two additional procedures in the `ojsaadm.apver usr adm pkg` package and use them to call the `sys.appver conns role pkg` package across a database link. The code for this is shown in Listing 12-51. Look back up to the end of Listing 12-50 and you will see that we grant execute on the `sys.appver conns role pkg` to the `ojsaadm` user.

Listing 12-51. *Grant Appver Conns Role Across Link*

```
    PROCEDURE p_grant_apver_appver_conns( username VARCHAR2, proxyname VARCHAR2 )
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt :=
'BEGIN sys.appver_conns_role_pkg.p_grant_appver_conns_role@apver_link( :1 ); END;';
        EXECUTE IMMEDIATE m stmt USING username, proxyname;
    END p grant apver appver conns;

    PROCEDURE p_revoke_apver_appver_conns( username VARCHAR2, proxyname VARCHAR2 )
    AS
        m stmt VARCHAR2(100);
    BEGIN
        m stmt :=
'BEGIN sys.appver_conns_role_pkg.p_revoke_appver_conns_role@apver_link( :1 ); END;';
        EXECUTE IMMEDIATE m stmt USING username, proxyname;
    END p revoke apver appver conns;
```

The Register Application Button

With all the data entered on the Register New Application screen, and a representative application inner class created in the Oracle database, we are ready to register the application. That representative inner class may have been created in the database using the Create App Class button that I described previously, or you will need to execute a `CREATE OR REPLACE AND RESOLVE JAVA` command to create it.

When the Register button is selected, that event calls the `registerButton actionPerformed()` method. Again we are using a delayed thread process, and we are displaying the `sayWaitDialog` while we are processing. In brief, this method does four things. It creates a local instance of the class described by

the user entries; it executes three insert statements to register various aspects of the application; it grants the current session user an administrative role; and it sets the context to the new application and calls methods to create a placeholder in the `v app conn registry` view for a future list of connection strings. There is no two-factor authentication, nor is there key exchange for the new application in this process.

Listing 12-52 shows the code used to create the local instance of the described application inner class. We discard any trailing dot (.) on the package name.

Listing 12-52. *Create Local Instance of Defined Class*

```
String innerClassName = packageTextField.getText();

if (innerClassName.endsWith("."))

    innerClassName.substring( 0, innerClassName.length() - 1 );
innerClassName = innerClassName + "." + classTextField.getText() +
    "$" + innerClassTextField.getText();

Class classToRegister = Class.forName(innerClassName);

Object appClass = classToRegister.newInstance();
```

■ **Note** This application inner class must exist in compiled form on the same client where the Register New Application screen is running in order to register the application. The application inner class must also be on the current CLASSPATH.

The first insert we execute will add the current user as an application administrator for this application. The insert command is shown in Listing 12-53. Notice we use the system context value of the client identifier that we have set equal to the SSO user. He is the designated initial administrator of the application associated with the fully qualified inner class name that we set as the first parameter.

Listing 12-53. *Insert Application Administrator*

```
String updateString =
    "insert into appsec.v_application_admins " +
    "( class name, user id ) values ( ?, " +
    "SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' ))";
PreparedStatement pstmt =
    conn.prepareStatement(updateString);
pstmt.setString(1, innerClassName);
pstmt.executeUpdate();
```

This is a dynamic update statement, but it is not susceptible to SQL Injection. It is a parameterized, prepared statement. We discussed and tested the security of this kind of statement in Chapter 8.

The second insert we make is into the `appsec.v app class id` view. We will associate the fully qualified inner class name with a specific application ID. The code for this insert command is provided in Listing 12-54.This is also done using a prepared statement.

Listing 12-54. *Insert Application Class to ID Relationship*

```
    String updateString =
        "insert into appsec.v_app_class_id " +
        "( class name, application id ) values ( ?, ? )";
    PreparedStatement pstmt = conn.prepareStatement(updateString);
    pstmt.setString(1, innerClassName);
    pstmt.setString(2, applicationIDTextField.getText().toUpperCase());
    pstmt.executeUpdate();
```

The last of the three insert statements is our application registration entry into the `appsec.v application registry` view. This data entry correlates the proxy user for this application with a secure application role. The code for the insert command is given in Listing 12-55.

Listing 12-55. *Application Registration Entry*

```
    String updateString =
        "insert into appsec.v_application_registry " +
        "( application id, app user, app role ) values ( ?, ?, ? )";
    PreparedStatement pstmt = conn.prepareStatement(updateString);
    pstmt.setString(1, applicationIDTextField.getText().toUpperCase());
    pstmt.setString(2, applicationUserTextField.getText().toUpperCase());
    pstmt.setString(3, applicationRoleTextField.getText().toUpperCase());
    pstmt.executeUpdate();
```

When we make an entry into the `v application registry` view on the `apver` database instance, we need to have an identical entry in the `v application registry` view on the `orcl` instance. That is pretty easy to arrange with an insert trigger across a database link. (There are also more robust ways to do this with message queues or streams.) Because we already have a private database link from `appsec` on the `apver` instance to `apsec` on the `orcl` instance, all we need is the trigger. (Previously we had used the `orcl link` to access data in the `HR` schema on the `orcl` instance from procedures running on the `apver` instance.) Listing 12-56 shows the trigger we will use to insert an identical record in the `v application registry` view on the `orcl` instance to the one we inserted on the `apver` instance.

Listing 12-56. *Insert Trigger on Application Registry*

```
CREATE OR REPLACE TRIGGER appsec.t application registry iar
    AFTER INSERT ON appsec.t application registry FOR EACH ROW
BEGIN
    INSERT INTO appsec.v application registry@orcl_link
    ( application id, app user, app role ) VALUES
    ( :new.application id, :new.app user, :new.app role );
    -- OJDBC will auto-commit this insert on orcl instance
END;
/

ALTER TRIGGER appsec.t application registry iar ENABLE;
```

We want to give the current user access to edit the connection strings for this application they just registered. To do that task, the user needs to be granted the `appver conns` role on the `apver` instance. This is in addition to being able to proxy through the `avadmin` user. We create a prepared statement to call the stored procedure `sys.appver conns role pkg.p grant appver conns role`. This is shown in Listing 12-57.

Listing 12-57. *Grant Appver_Conns Role to the Current User*

```
String updateString =
    "BEGIN sys.appver conns role pkg.p grant appver conns role( " +
    "SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) ); END;";
PreparedStatement pstmt =
    conn.prepareStatement(updateString);
pstmt.executeUpdate();
```

Finally, we take one last step in the `registerButton actionPerformed()` method. We create a placeholder in the `v app conn registry` view for the connection strings to be used by this new application. To do that, we change contexts to the context of the new application by calling `setAppContext()`. Then we execute the `getAppConnections()` method, and follow that with a call to `putAppConnections()`. That series of commands, shown in Listing 12-58, will create the entry in `v app conn registry`.

Listing 12-58. *Insert Entry into Application Connections Registry*

```
OracleJavaSecure.setAppContext(
    applicationIDTextField.getText().toUpperCase(), appClass, "");
OracleJavaSecure.getAppConnections();
OracleJavaSecure.putAppConnections();
```

When we set the new application context in Listing 12-58, we provide the application ID that the user entered along with the class we instantiated from the description on the screen and an empty string in place of a two-factor authentication code.

■ **Note** We have not accomplished two-factor authentication nor key exchange for this new application. When we call `putAppConnections()` with an empty list of connections, Oracle database does not balk at decryption and encryption, but rather inserts an empty conn string list for the new application.

When these steps are complete, the new application has been registered. It still remains that we need to enter some application connection strings into the list for this application. We can do that on another functional screen of the Security Administration Interface application.

Recall that after the Register button has been pressed, the **Close** button changes to an Exit button that will exit the JVM. Because we have changed application contexts, we cannot return to the `OJSAdmin` menu and continue working with the list of connection strings for the `OJSAdmin` application.

Application Selection Screen

In the functional screens we have discussed so far, we have done all the application security administration we can as the `OJSAdmin` application. We even set our context to be that of another application in order to act on behalf of the application and to register it. But in order to manage the connection strings associated with a different application, we will have to start out with the identity of that application.

In the Pick Application to Manage functional screen shown in Figure 12- 7, we will be selecting a specific application from the list of registered applications that we want to manage. The list is shown in the top drop-down box, and the Manage Selected Application button will start a new application in a new JVM. Because practically all our security structures, especially those in the `OracleJavaSecure` class, are static, we need to start each different application in a separate JVM in order to have application-specific settings of our static members.



***Figure 12-7.** Pick an Application to Manage functional Screen*

Initializing the List of Applications

We will run the Pick Application to Manage screen, `PickAppManage` class, with a connection that is proxied through the `avadmin` user. We will have the `appver admin` role in order to read the list of applications that are registered. In our `dataInit()` method, we have a single query that we use to get the list of applications. This query is shown in Listing 12-59. Notice in the query that we are getting the application ID and the related application inner class name to represent the application. We display this list in the drop-down combo box at the top of the screen.

***Listing 12-59.** Query to get List of Registered Applications*

```
stmt.executeQuery(
    "SELECT DISTINCT application_id || '/' || class_name o FROM appsec.v app class id "
    + "where class name in " +
    "( select distinct class name from appsec. v app conn registry filtered ) " +
    "ORDER BY o");
```

We haven't configured it yet, but our intent is to limit the list of applications shown here to just those that the current user has been assigned to administer. This will be accomplished by restricting what entries in the `v app conn registry filtered` view are accessible to the current user. Once we have restricted that view, then the query in `PickAppManage`, Listing 12-59 will also be restricted by the `where` clause.

Selecting the Manage Selected Application Button

When you have selected an application to manage and you press the Manage Selected Application button, the event will run the `manageAppButton actionPerformed()` method. We start that method by making sure an application has been selected from the list, and then by parsing out the application ID and the application inner class name from the selected item. The code for this is shown in Listing 12-60.

Listing 12-60. *Parse Selected Application*

```
if (0 == appClassComboBox.getSelectedIndex())
    return;
String appId = (String)appClassComboBox.getSelectedItem();
int place = appId.indexOf("/");
String appClass = appId.substring(place + 1);
appId = appId.substring(0, place);
```

With those two data elements, the application ID and the application inner class name, we can start a new JVM as that specific application. We plan to start a new instance of the `OJSAdmin`, security administration interface application with the identity of the application we selected here, instead of the default identity of `OJSAdmin`. The code we use to start the new JVM and new instance of `OJSAdmin` is shown in Listing 12-61.

Notice in the last line of Listing 12-61 that we are going to run the `main()` method of `OJSAdmin`, and we are providing the application ID and application inner class name as arguments. In our discussion of the `Login` constructors in Listing 12-4 and the `OJSAdmin.ojsInit()` method in Listing 12-11, I mentioned how we can take on the identity of a different application. There we saw code that uses reflection to create an instance of the application inner class that is named in argument two, and we set the application ID of `Login` to the value of argument one.

Listing 12-61. *Start a New JVM and new Instance of OJSAdmin*

```
Runtime rt = Runtime.getRuntime();
Process proc =
    rt.exec("C:/Java/jdk1.6.0_24/bin/javaw -classpath " +
            "C:/dev/mywork/OraJavSec/Project1/classes;C:/dev/ojdbc6.jar " +
            "orajavsec.OJSAdmin " + appId + " " + appClass);
BufferedReader stdError =
    new BufferedReader(new InputStreamReader(proc.getErrorStream()));
String inLine;
while ((inLine = stdError.readLine()) != null) {
    System.out.println(inLine);
}
```

We get a new instance of `Runtime` and call its `exec()` method. The `exec()` method returns an instance of `Process`, and we set a local member variable. Typically, you might get the output and error streams that are coming from the `Process`, and you might print them or handle them. In our case, we have intentionally requested not to have the output stream generated by our `Process`. We did that by running the version of the Java runtime that starts *windowless*, *javaw.exe*. Notice the "w" at the end of the executable name. In this way, we don't get an output stream and we don't have to deal with it. We do, however, choose to print the error stream to the current system out of the current JVM that called the new `Runtime.exec()`. We want to see any exception messages that are thrown, and if we don't deal with the error stream, our process may lock-up while waiting for us to deal with it.

If you can, I recommend that, in your corporate environment, you give full paths to your *javaw* executable and to the current application, `OJSAdmin` and Oracle drivers jar file in the `CLASSPATH` setting. If that doesn't work for all the computers that will be running this code in your environment, you will need to set environment variables for the `Runtime`. You might also consider using the `ProcessBuilder` class to get more fine-grained control of the environment settings. You might also consider placing these paths in a properties file that will accompany the code. You can read these specific values from the properties file; however, you will need a different properties file for each different client computer environment.

A different reason for why you might want to place these setting in a properties file and read them from there is to make handling an upgrade to the JVM easier. When the path to java.exe changes from the jdk1.6.0_24 directory to the jdk1.6.0_28 directory, you only need to edit the properties file. This presumes that it is easier to edit a text file on all your client computers than it is to recompile the code and distribute a new jar file to all. That might be a personal preference.

One argument against recompiling is the possibility of introducing errors in the process, but I have never seen that happen just from the compilation process, and perhaps we want to recompile our classes in order to gain from any efficiency provided by the updated compiler. Maybe this is a personal preference based on experience or standard practice.

■ **Note** When the new Runtime Process starts, the Login Dialog will be instantiated again with the new application identity. That will result in a new prompt for entry of a two-factor authentication code—one specific to the selected application.

Connection String Editor

You can see in Figure 12- 8 that all the elements of an Oracle connection string are represented by text fields on the Edit Application Connections screen, `EditAppConns`. We provide two functional buttons there: one to add the newly described connection to the list, and one to save the list to the Oracle database.

When we select the button to add the connection to our list, the Update Connection String button, it only updates the local list; it does not store the new/updated connection string in the database. After editing or adding connections to the local list, select the **Save List** button to update the database.

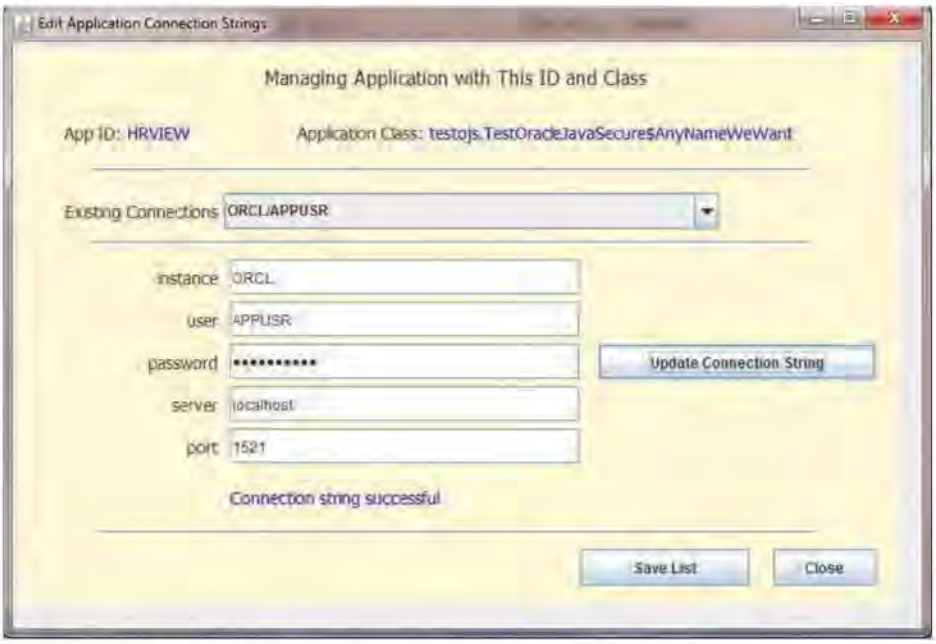


Figure 12-8. Edit Application Connections functional screen

Initializing the List of Connection Strings

In the `dataInit()` method of the Edit Application Connection Strings screen, we call a new method in the `OracleJavaSecure` class, `listConnNames()`. The list itself, `connsHash` is private, so we have to ask `OracleJavaSecure` to provide the names, `keys` of the `Hashtable`. The code for the `listConnNames()` method is given in Listing 12-62.

Listing 12-62. *Get List of Connection Names*

```
static Vector listConnNames() {
    Vector<String> rtrnV = new Vector<String>();
    try {
        for (String key : connsHash.keySet())
            rtrnV.add(key);
    } catch (Exception x) {}
    return rtrnV;
}
```

This method has the default access modifier, because we don't specify `private` or `public`. That default modifier is "package" accessibility. Only classes that are part of the same package, `orajavsec` can execute this method. Fortunately for us, `OJSAdmin` is in the `orajavsec` package.

The call to this new method from the `EditAppConns.dataInit()` method looks like the following. We use the returned Vector to populate the drop-down list of connection string names.

```
Vector<String> connList = OracleJavaSecure.listConnNames();
```

Selecting an Existing Connection String

There is little value in showing the list of existing connection strings for the application. We cannot edit individual parameters of the string, like the password, leaving other parameters as is. That is because we are not going to decrypt and expose the password here despite the convenience it would provide. What we gain by showing the list of connection strings is that we can see our progress as we enter multiple strings, and we can assure that at least the key names of an updated entry are spelled correctly.

When a user selects a connection string name from the list, we populate the first two text fields with the database instance name and user name from the selected item. The other fields remain blank.

Updating a Connection String in the List

When all the text fields on the **Edit Application Connection Strings** screen have been filled appropriately, the user may select the **Update Connection String** button. This action will call the `updateButton actionPerformed()` method, whose primary duty is to call the `OracleJavaSecure.putAppConnString()` method. This call is shown in the code in Listing 12-63.

Listing 12-63. *Update Connection String in the List*

```
connSuccessLabel.setText(OracleJavaSecure.putAppConnString(
    instanceTextField.getText().toUpperCase(),
    userTextField.getText().toUpperCase(),
    new String(passwordField.getPassword()),
    serverTextField.getText(),
    portTextField.getText(),
```

```
        true));
```

We pass the text field values to `putAppConnString()`. For the password field, it is a `JPasswordField` type, which displays an asterisk for each character entered in the text field. We cannot get the text of the password field as a `String`; rather, we call the `getPassword()` method to get a character array, from which we instantiate a `String` to pass to `putAppConnString()`.

When we call `putAppConnString()`, we pass a `boolean` set to `true` as the last argument. That tells `putAppConnString()` to test the resultant connection string to see whether we are able to make a connection to the database with the designated string.

We have modified the `putAppConnString()` method that takes this final `boolean` to return a `String` instead of a `void`. Previously, we had only sent a success or fail message to the `System.out` stream. Now we are additionally returning the success or failure message. We set the value of `connSuccessLabel` on this screen to the value of the message being returned. After we have updated the connection strings in the list, we want to refresh the list in `existingConnComboBox`, so we call `OracleJavaSecure.listConnNames()` like we did in `dataInit()` to repopulate the drop-down combo box.

Saving the List of Connection Strings to the Database

The final action to be taken on the Edit Application Connection Strings screen is a call to the `saveButton actionPerformed()` method, which occurs when the Save List button is selected. In this method, we simply call `OracleJavaSecure.putAppConnections()`. That was easy!

Connection String Copy Screen

If you have modified the `innerClassRevLvl` member of the application inner class in order to designate a new version of the application, with a new/modified set of Oracle connection strings, then you can use the Copy to New Version screen shown in Figure 12-9 to duplicate the previous application version connection strings to the current version.

■ **Note** It only works to copy connection strings from a previous version to the current version if the list of connection strings for the current version is empty, that is the list in the database.



Figure 12-9. Copy to New Version functional screen

The Copy to New Version screen is very simple. It has a text field to enter the previous version number, from which the connection strings will be copied, and a button labeled Copy Existing Conn Strings. When that button is selected, the result is a call to the `OracleJavaSecure.copyAppConnections()` method, passing the user-entered previous version number.

Limiting Certain Administrators to Certain Applications

Before I end this chapter, let me elaborate on something I have alluded to several times: limiting which applications an administrator may manage. You will recall in our discussion of the Register New Application screen, we created a new table to designate an administrator for the new application, `appsec.t application admins`. We have not done anything with that table yet, except maintain the data in it.

You may recall from Listing 12-48 that we limited certain privileges on that table of administrators to just a single user, `osadmin`. This administrator of administrators job could be extended to more individuals, but we didn't create a role to handle this. That might be a good future improvement. We do not limit the insert privilege, because we are inserting the current session user as an initial administrator for the application from the Register New Application screen.

Virtual Private Database

How will we use this table of application administrators? One way is to use a feature of the Oracle database that goes by many names. The feature is implemented in a database package named `DBMS RLS`, which stands for "Row Level Security." I suppose that might be the best name, but this feature is more often referred to as "Virtual Private Database" (VPD), as in the *Oracle Database Security Guide* document. You might also see it called "Dynamic Where Clause" or "Fine-Grained Access Control," depending on how it is being used.

It amounts to a dynamic where clause being applied to queries and execute commands on a table, and it is enforced by a database security policy. The main benefits of this feature are that it is transparently applied to all specified user interactions with the data table, and there is no way around the policy for the user.

As an example, let's say I define a policy that says every time a user deals with the `HR.EMPLOYEES` table, we will apply the dynamic where clause, "`where employee id = this employee id`" assuming we have already calculated `this employee id`. Because this policy is applied to the table, even if the employee with ID `300` is selecting or updating the data through a view or procedure (as `AUTHID CURRENT USER`), they still can only deal with their own record, for their `employee id, 300`.

■ **Note** Procedures that operate with schema privileges, not `AUTHID CURRENT USER` will be restricted to data available to the schema user, not the current user. However, if the VPD policy is restricting access by a different aspect of the user's identity, such as a session context parameter, then a user's access through these procedures is also restricted by the VPD policy.

■ **Note** If you are connected as SYS user or the schema user who owns the table, VPD policies do not apply to your access to the table. You see it all, unless the VPD function depends on a session context parameter.

For all our application connections, we are setting the SYS CONTEXT parameter CLIENT IDENTIFIER to the user ID from our SSO process. We would like to restrict access to the appsec.t app conn registry table to just those users whose SSO user IDs are listed in appsec.v application admins as administrators for the specific application. That is the easy part—our dynamic where clause will be:

```
WHERE class name IN ( SELECT class name FROM appsec.t application admins
    WHERE user id = SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) )
```

But we also want to allow any administrators of administrators to see all rows. We can find out who those individuals are by finding who has been granted extra privileges, like UPDATE on the appsec.v application admins view. We will add this to our dynamic where clause:

```
OR SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) =
    ( SELECT GRANTEE FROM SYS.DBA TAB PRIVS
        WHERE TABLE NAME='V APPLICATION ADMINS'
        AND OWNER='APPSEC'
        AND PRIVILEGE='UPDATE'
        AND GRANTEE=SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) )
```

That addition will also allow the current user to select rows from appsec.v app conn registry where it is true that the current user is found in the SYS.DBA TAB PRIVS data dictionary view with the privilege to UPDATE the appsec.v application admins view. If this is true, our dynamic where clause will return all rows. This requires that we grant select on the data dictionary view, SYS.DBA TAB PRIVS to the appsec user who will execute the VPD code:

```
GRANT SELECT ON SYS.DBA TAB PRIVS TO APPSEC WITH GRANT OPTION;
```

In a couple sections form here, we are going to use a similar dynamic where clause in a view, v app conn registry filtered. Then we will let other users select from that view. This reveals a potential limitation to the grant we issued earlier. In this case we specified WITH GRANT OPTION, as if appsec was interested in granting select on the sys.dba tab privs view to others. Appsec will not be doing that, but she wants to permit other users to view data that is selected from that data dictionary view. If we had not specified WITH GRANT OPTION, then appsec would not be able to let others see what she sees in the dba tab privs view. Grants cannot be transferred to others without specifying WITH GRANT OPTION.

The first step in using VPD is to create a function using a specific template form, that will return the dynamic where clause as a VARCHAR2. Listing 12-64 shows the function that encapsulates the dynamic where clause I just described, appsec.apps for admin.

Listing 12-64. *VPD Function for Administrators*

```
CREATE OR REPLACE FUNCTION appsec.apps for admin(
    m schema nm VARCHAR2,
    m table nm  VARCHAR2 )
RETURN VARCHAR2
IS
    rtrn clause VARCHAR2(400);
BEGIN
    rtrn clause :=
    'class name IN ( SELECT class name FROM appsec.t application admins '
    || 'WHERE user id = SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ) ) '
    || 'OR SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ) = '
    || '( SELECT GRANTEE FROM SYS.DBA TAB PRIVS '
```

```
    || '  WHERE TABLE NAME=''V APPLICATION ADMINS'' '
    || '  AND OWNER=''APPSEC'' '
    || '  AND PRIVILEGE=''UPDATE'' '
    || '  AND GRANTEE=SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ) )';
    RETURN rtrn clause;
END apps for admin;
/
```

We create this procedure on the apver instance, where it is applied to the `t app class registry` table. Listing 12-65 shows the command that creates the policy to use this function. Notice that we only apply this policy to the statement types: `INSERT`, `UPDATE`, and `DELETE`. We consider this a restriction on administrators. Notice also that the policy is declared to be `STATIC`. That lends it great speed since it will reside in and execute from the system global area, SGA memory. See the Oracle Database Security Guide document for more information on VPD.

Listing 12-65. *VPD Policy for Administrators*

```
BEGIN
DBMS RLS.ADD POLICY (
    object schema => 'appsec',
    object name => 't app conn registry',
    policy name => 'apps for admin policy',
    function schema => 'appsec',
    policy function => 'apps for admin',
    statement_types => 'INSERT,UPDATE,DELETE',
    policy_type => DBMS_RLS.STATIC );
END;
/
```

If we can do that, then surely we can restrict what application connection string lists our application users can `SELECT`. We want to apply the same `where` clause as before, with an additional allowance. If the user has been granted the privilege of proxying through the application user associated with an application, then the user should be able to select the list of connection strings. That addition to the `where` clause looks like this:

```
WHERE class name IN ( SELECT class name FROM appsec.t app class id '
    WHERE application id IN ( '
        SELECT application id FROM appsec.t application registry '
        WHERE app user IN ( '
            SELECT proxy FROM ojsaadm.instance proxy users@orcl link '
            WHERE client = SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ))) '
```

Reading from the inside out, that where clause looks at the cross-instance view of proxy users that we configured in Listing 12-29, getting all the proxy users that have been granted to the current user, then selects all applications from the `t application registry` table that are associated with that proxy user, then selects all application inner class names for those applications. The current user will be able to select application connection string lists from `v app conn registry` that are associated with those application inner classes. Listing 12-66 shows both the function that returns the dynamic where clause and the policy that applies it to `SELECT` statements on the `appsec.t app conn registry` table.

Listing 12-66. *VPD for Users*

```
CREATE OR REPLACE FUNCTION appsec.apps for user(
```

```
    m schema nm VARCHAR2,
    m table nm  VARCHAR2 )
RETURN VARCHAR2
IS
    rtrn clause VARCHAR2(400);
BEGIN
--    appsec.app sec pkg.p log error( 122, 'dave',
--    'appsec.apps for user: ' || SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) );

    rtrn clause :=
    'class name IN ( SELECT class name FROM appsec.t app class id '
    || 'WHERE application id IN ( '
    || 'SELECT application id FROM appsec.t application registry '
    || 'WHERE app user IN ( '
    || 'SELECT proxy FROM ojsaadm.instance proxy users@orcl link '
    || 'WHERE client = SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ))) '
    || 'UNION SELECT class name FROM appsec.t application admins '
    || 'WHERE user id = SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ) '
    || 'OR SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' ) = ( '
    || 'SELECT GRANTEE FROM SYS.DBA TAB PRIVS '
    || 'WHERE TABLE NAME=''V APPLICATION ADMINS'' '
    || 'AND OWNER=''APPSEC'' '
    || 'AND PRIVILEGE=''UPDATE'' '
    || 'AND GRANTEE=SYS CONTEXT( ''USERENV'', ''CLIENT IDENTIFIER'' )))';
    RETURN rtrn clause;
END apps for user;
/

BEGIN
DBMS RLS.ADD POLICY (
    object schema => 'appsec',
    object name => 't app conn registry',
    policy name => 'apps for user policy',
    function schema => 'appsec',
    policy function => 'apps for user',
    statement_types => 'SELECT',
    policy type => DBMS RLS.STATIC );
END;
/
```

One of the biggest benefits of VPD is also one of its biggest problems. The problem is that it can be hard to deal with. It is a hidden restriction, not listed in the grants or in the application code. The dynamic where clause gets applied behind the scenes, transparent to the user. That difficulty is a feature, a feature worth paying for in some secure computing environments. If you need VPD on steroids, you can acquire Oracle Label Security.

To disable the VPD policies we configured, you would execute the statements in Listing 12-67. I am going to do that, because for our efforts we can take a different, more manageable approach to achieve the same effect. Also disable VPD because certain procedures in `appsec` packages running as "definer's rights" could not select from `v app conn registry` when `appsec` cannot.

Listing 12-67. *Disable VPD Policies*

```
BEGIN
DBMS RLS.DROP POLICY (
    object schema => 'appsec',
    object name => 't app conn registry',
    policy name => 'apps for admin policy' );
END;
/
BEGIN
DBMS RLS.DROP POLICY (
    object schema => 'appsec',
    object name => 't app conn registry',
    policy name => 'apps for user policy' );
END;
/
```

Adding a Dynamic Where Clause to Procedures

There are only two procedures that access the `t app conn registry` table in a manner that concerns us. Both of these procedures are in the `appsec.appsec only pkg` package: `p get class conns` and `p set class conns`. Outside of those procedures, only `SYS`, the `appsec` schema user account and users who have been granted proxy through `avadmin` (with `appver admin` role) have access to the table or a view of it.

I propose that instead of using VPD to restrict data access, we modify the specific procedures of concern with the same dynamic where clauses that we proposed for VPD. The resulting procedures are shown in Listing 12-68.

Listing 12-68. *Procedures Protected by Dynamic Where Clause*

```
    PROCEDURE p_get_class_conns(
        m class name         v app conn registry.class name%TYPE,
        m class version      v app conn registry.class version%TYPE,
        m class instance OUT v app conn registry.class instance%TYPE,
        m connections    OUT v app conn registry.connections%TYPE )
    IS BEGIN
        SELECT class instance, connections
        INTO m class instance, m connections
        FROM appsec.v app conn registry
        WHERE class name = m class name
        AND class version = m class version

        AND class_name IN ( SELECT class_name FROM appsec.v_app_class_id
        WHERE application_id IN (
        SELECT application_id FROM appsec.v_application_registry
        WHERE app_user IN (
        SELECT proxy FROM ojsaadm.instance_proxy_users@orcl_link
        WHERE client = SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' )))
        UNION SELECT class_name FROM appsec.v_application_admins
        WHERE user_id = SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' )
```

```
    OR SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' ) = (
    SELECT GRANTEE FROM SYS.DBA_TAB_PRIVS
    WHERE TABLE_NAME='V_APPLICATION_ADMINS'
    AND OWNER='APPSEC'
    AND PRIVILEGE='UPDATE'
    AND GRANTEE=SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' )));
END p get class conns;



PROCEDURE p_set_class_conns(
    m class name     v app conn registry.class name%TYPE,
    m class version  v app conn registry.class version%TYPE,
    m class instance v app conn registry.class instance%TYPE,
    m connections    v app conn registry.connections%TYPE )
IS
    v count INTEGER;
    v count able INTEGER;
BEGIN
    SELECT COUNT(*) INTO v count
        FROM appsec.v app conn registry
        WHERE class name = m class name
        AND class version = m class version;

    SELECT COUNT(*) INTO v_count_able
        FROM appsec.v_app_conn_registry
        WHERE class_name = m_class_name
        AND class_version = m_class_version
        AND class_name IN (
        SELECT class_name FROM appsec.v_application_admins
        WHERE user_id = SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' )
        OR SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' ) = (
        SELECT GRANTEE FROM SYS.DBA_TAB_PRIVS
        WHERE TABLE_NAME='V_APPLICATION_ADMINS'
        AND OWNER='APPSEC'
        AND PRIVILEGE='UPDATE'
        AND GRANTEE=SYS_CONTEXT( 'USERENV', 'CLIENT_IDENTIFIER' )));

    IF v count = 0 THEN
        INSERT INTO v app conn registry ( class name, class version,
            class instance, connections ) VALUES
            ( m class name, m class version, m class instance, m connections );
    ELSE

        IF v_count_able > 0 THEN

            UPDATE v app conn registry
                SET class instance = m class instance,
                connections = m connections, update dt = SYSDATE
            WHERE class name = m class name
            AND class version = m class version;
        END IF;
```

```
        END IF;
    END p set class conns;
```

In `p set class conns` we freely allow insert, which will happen during new application registration, but if an entry already exists for this application, we apply a test that uses our administrative dynamic where clause. If the record is found (`v count able > 0`) while using the dynamic where clause, then we allow update to the record.

Adding a Dynamic Where Clause to a View

To complete our emulation of VPD on the `appsec.t app conn registry` table, we need to limit the rows returned by a select on a view of that table, `appsec.v app conn registry filtered`. Only `SYS`, the `appsec` schema user and users who have been granted proxy through `avadmin` will be able to select from the view, but we can limit what those folks see as well. Our plan is really only to limit what administrative users, proxying through `avadmin` will see, since `SYS` and the schema owner can see all the data in the table without using the view.

The dynamic where clause in this view selects from `sys.dba tab privs`, which was granted to `appsec`. `Appsec` grants this view to the `appver admin` role. Fortunately, `appsec` was granted select on `dba tab privs` with `WITH GRANT OPTION` permission, so she can allow others to see what she sees in that data dictionary view; otherwise, other users couldn't select from `v app conn registry filtered`, even though they have the role `appver admin`.

The definition of `appsec.v app conn registry filtered` includes a dynamic where clause that limits access like the administrative limitation we put on `p set class conns` in Listing 12-68. We will not add the allowance to permit users of applications to select for applications they have access to by proxy grants. The code for `v app conn registry filtered` is shown in Listing 12-69. You may recall that we already used this view in one of our functional screens, the Pick App to Manage screen. Look back at Listing 12-59 to see how we limit the applications an `appver admin` user may select from in order to manage connection strings. This view limits what the user sees, but the limit to what the user can do is enforced by the dynamic where clauses we added to `p get class conns` and especially `p set class conns`.

Listing 12-69. *View Protected by Dynamic Where Clause*

```
CREATE OR REPLACE VIEW appsec.v app conn registry filtered
AS
    SELECT * FROM appsec.t app conn registry
    WHERE class name IN (
        SELECT class name FROM appsec.v application admins
        WHERE user id = SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' )
    OR SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' ) = (
        SELECT GRANTEE FROM SYS.DBA TAB PRIVS
        WHERE TABLE NAME='V APPLICATION ADMINS'
        AND OWNER='APPSEC'
        AND PRIVILEGE='UPDATE'
        AND GRANTEE=SYS CONTEXT( 'USERENV', 'CLIENT IDENTIFIER' )));

GRANT SELECT ON appsec.v app conn registry filtered TO appver admin;
```

Scripts Execution and Code Compilation

If you have been running the code in the files in Chapter 12 as we've been discussing it, then you are mostly through this section. You can skip down to where we discuss editing the Java code. Otherwise, you need to execute the following scripts (note that there are some dependencies between them):

Execute *Chapter12/OrclSecAdm.sql* as the `secadm` user on the `orcl` instance (your primary database instance). Change the passwords for both the `ojsaadm` and `avadmin` users before executing the script.

Execute *Chapter12/OrclSys.sql* as `SYS` on the `orcl` instance. Change the placeholder OS user ID, `osadmin` to your OS user ID or whoever is the primary application security administrator.

Execute *Chapter12/OrclHR.sql* as the `HR` schema user on the `orcl` instance.

Execute *Chapter12/OrclAppSec.sql* as the `appsec` user on the `orcl` instance.

Execute *Chapter12/ApverSys.sql* as `SYS` on the `apver` instance. Change the passwords for both the `ojsaadm` and `avadmin` users before executing the script.

Execute *Chapter12/OrclOJSAAdm.sql* as the `ojsaadm` user on the `orcl` instance. Change the password for the `ojsaadm` user on the `apver` instance, in the database link before executing the script.

From the *Chapter12* directory, edit the code as listed here:

- At the top of *orajavsec/OracleJavaSecure.java*, edit the values for `expectedDomain`, `comDomain`, `smtpHost` and `baseURL`. Also, insert the correct value for `prime` (the encoded connection string for `appver` user on the `apver` instance) into place in the `setAppVerConnection()` method.
- In *orajavsec/PickAppManage.java*, edit the paths to *javaw.exe*, your classes, and *ojdbc.jar* file that are used in the `Runtime.exec()` command in the `manageAppButton actionPerformed()` method.
- In *orajavsec/RegNewApp.java*, edit the password for the `avadmin` user in the default constructor. Also, temporarily uncomment the calls to `putAppConnString()` and `putAppConnections()`.

■ **Note** Having the `avadmin` password, and the calls to `putAppConString()` and `putAppConnections()`, in *RegNewApp.java* is only temporary. After bootstrapping `OJSAdmin`, discussed shortly, be sure to remove the password and comment those method calls. Then recompile `OJSAdmin`.

Execute the following command to compile `OJSAdmin` and all related classes:

javac orajavsec/OJSAdmin.java

Finally, uncomment the command to `CREATE OR REPLACE AND RESOLVE JAVA SOURCE` at the top of *OracleJavaSecure.java,* and execute that code as a script on both the `orcl` instance (as `appsec` or `SYS`) and the `apver` instance (as `SYS`).

Final Updates to OracleJavaSecure

There are several legacy methods that we used differently in earlier chapters: `getAppAuthConn()` and three `setConnection()` methods. Initially we declared these methods to have `public` visibility, but now we are using the `getAAConnRole()` public method to handle those calls. So we will change the access modifiers for those legacy methods to be `private`.

There is also one method that we haven't yet put to use: `removeAppConnString()`. It is our intent to someday implement that from the `OJSAdmin` application. `OJSAdmin` resides in the `orajavsec` package, just like the `OracleJavaSecure` class. So to make `removeAppConnString()` visible to `OJSAdmin`, we will change the access modifier to default (package).

Single Oracle Instance Code

If you did not create a second Oracle instance to run the `appver` processes separately, then you need to run the code in *Chapter12single*. Execute all the scripts and commands listed in the last section, except there is no *ApverSys.sql* script. One example of the names for the scripts you will use is *Chapter12single/OrclSecAdm.sql*.

You will edit and compile the Java code similarly as well, except you will compile and run the code in the *Chapter12single* directory.

Bootstrap OJSAdmin

We need a way to get the security administration interface, `OJSAdmin` started. We will be able to get all future applications started by using `OJSAdmin`, but for `OJSAdmin` itself, we are going to take a couple steps to bootstrap the application. Did you realize we have been bootstrapping our applications this whole time, from Chapter 7 through Chapter 11? That's right, when we ran each application with a couple calls to `putAppConnString()` and another call to `putConnections()`, we were priming the pump and getting the application engine started with those password strings. The call to `putConnections()` entered a record in the `appsec.v app conn registry` table that we could use from then on.

After our discussion of the register new application, `RegNewApp` functional screen, you can conclude that getting a new application registered has become more complex in just this chapter. Part of that complexity is for enhanced security – only allowing certain administrators to modify connection strings for certain applications. And part of it is for allowing this Security Administration Interface to manage multiple applications; and even more, to become those applications in order to edit associated connection strings.

It is no longer the case that just inserting a record into `v app conn registry` is sufficient to register an application; however, that is the most significant step. Now we must designate an administrator for the application in `v application admin` and we must associate the application ID with the application inner class name in `v app class id`. We have always needed an entry in the `v application registry` table that associates a secure application role to a specific application ID and application user.

For the `OJSAdmin` class, Security Administration Interface, we have entered a number of those data elements from the scripts. We have an entry in `v application registry` on both `orcl` and `apver` instances. We inserted those records independently into `v application registry` on `orcl` and `apver`, but in the future we will make inserts on `apver`, and they will be automatically inserted in `orcl` by means of the insert trigger. We have also made entries in both `v application admins` and `v app class id` for the `OJSAdmin` application. Now all we need is an entry in `v app conn registry`, which will include our application inner class and a list of connection strings (starting out empty).

We are going to call on the facility of our Register New Application functional screen to get the entry into `v app conn registry`. This is where we experience the chicken or egg quandary (by the way, the chicken came first). In this case, we need to acquire a connection string from `v app conn registry` before we can use Register New Application. At the same time, we want to use Register New Application to insert the initial (blank) list of connection strings into `v app conn registry`.

As we have in the preceding chapters, we are going to bootstrap the application connection list. In Chapter 12, we need to do that in two steps. As before, to bootstrap the application, we place a connection string in memory on the client application by calling `putAppConnString()`. We do that in the

ojsInit() method of RegNewApp as shown in Listing 12-70. We also call putAppConnections() but that has no effect when an there is no existing connection string list in the database—that call, at this point, does not save the list in the database.

Listing 12-70. *Bootstrap Register New Application for OJSAdmin*

```
public RegNewApp() {
    try {
        jbInit();
        // First times through way to build conn string list, or use this tool
        OracleJavaSecure.putAppConnString("apver", "avadmin", "password",
            "localhost", String.valueOf(1521));
        OracleJavaSecure.putAppConnections();
        conn = OracleJavaSecure.getAAConnRole("apver", "avadmin");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

We are going to enter some representative data on the Register New Application screen shown in Figure 12-6. The top three entries for application ID, app user and app role will not be needed – we have already entered records with the necessary data values in our scripts, and we haven't acquired the appver admin role needed to do the associated inserts. However, enter ojsadmin, avadmin, appver admin in the top three text fields. Enter the correct package, orajavsec, and enter the class name, Login. At this point the **Create App Class** button will appear. Click the **Create App Class** button to create the Java structure. If you ever mistype a package name, and create an incorrect class, you can drop the class with a command like this (don't do this, for information only):

```
DROP JAVA SOURCE appsec."badPackageName/Login";
```

Notice when you click the **Create App Class** button that the last text field on the screen is automatically populated with the default inner class name of the Login class, InnerRevLvlClass. With all the fields populated, you are now ready to click on the **Register** button. When you do that, the **Register New Application** screen will instantiate the class described by your package name and the class name Login$InnerRevLvlClass. Some errors will be reported because this event normally inserts data into the v application registry, v application admins and v app conn id views. Those inserts fail, but inserting the application inner class and an empty list of connection strings into v app conn registry succeeds. This was the bootstrap process!

The **Close** button turned to **Exit** because we have changed the application context. You can select the **Exit** button at this time. Your application is registered, but you still need to insert connection strings into the list in order to use the functional screens of OJSAdmin. Start OJSAdmin again, and provide the two-factor authentication code that is sent to you. This may seem like an odd step, but you need to enter the Register New Application screen again. This time as you go in, the putAppConnString() method is called to again add the avadmin connection string to the list in memory, but this time when we call putAppConnections(), that connection string will be saved in the list that resides in the database. Since our connection strings list contains the avadmin connection string, we entered this screen with the appver admin role, and we can UPDATE the v app conn registry view.

■ **Note** You must be the equivalent of `osadmin`, or have been granted proxy through `avadmin` and listed in `v application admins` for this application, `OJSADMIN`, to accomplish this final step.

At this point you can Close the screen and Exit the application, or you can just close the Register New Application screen and go on to our next step. In either case, when you finally do Exit the application, be sure to remove the bootstrap password from the code in *RegNewApp.java*.

There are several Oracle connections required to accomplish the various tasks of the security administration interface application. So far, we have only one connection string in the list, so we need to add the others. Run `OJSAdmin` and click on the Pick App to Manage button. In the drop-down selection box, select the `OJSAdmin` application `OJSADMIN/orajavsec.Login$InnerRevLvlClass`. You will receive another prompt for two-factor authentication; however, this is the same application. If it has been less than 10 minutes since the first two-factor code was sent to you, enter that code; otherwise wait for a new code.

You will see that the menu items at the bottom of the `OJSAdmin` menu screen are available. Select the Update Connection Strings menu item. Enter two additional connection strings as listed in Table 12-2.

Table 12-2. *Additional Connection Strings for OJSAdmin*

instance	user	password	server	port
orcl	appusr	password	localhost	1521
orcl	ojsaadm	password	localhost	1521

After each entry, click the Update Connection String button and assure the connection success message is displayed. After both entries are made, click on the Save List button to store the complete list of connection strings in the database. That completes the configuration of the `OJSAdmin` application, and all of its features are now ready for use. Exit `OJSAdmin` in both JVMs, then reenter to reload the connection strings.

Chapter Review

This chapter has perhaps been more fun to write and read because it provides a visual representation of the concepts we've been studying. The various menus and screens make managing the application security infrastructure much easier and more approachable. Also, by encoding the rules into a GUI application, we don't have to remember all the pieces and parts.

In addition to discussing the operation of the functional screens, we discussed GUI development in general, with specific attention to the single-threaded operation and the use of delayed threads and the `SwingUtilities.invokeLater()` method. We also explored `modal` dialog boxes and making dialogs and functional screens visible and invisible as required.

To make this security administration interface application work, we added a couple tables, several packages, a database link, a couple users and roles, and various other Oracle structures. We also added a couple utility methods to `OracleJavaSecure`, and modified a couple existing methods. A developer's job is never done, and those who implement the security infrastructure presented in this book will find reasons to modify it and expand it.

When you have finished reading and working your way through this book, you will want to continue on to read and work through the Supplemental material included in the source code download. In the supplement, we address several scenarios that you can use as templates and guides for accomplishing computer security. These are the scenarios we will cover:

- Testing with new applications, users and administrators
- Responding to an emergency – securing a new application and data
- Changing all the passwords
- Auditing and logging
- Performing an asset-centric risk assessment
- Tailoring installation and implementation

I encourage you to work on these issues, and I hope you have complete success in Application Security using Oracle and Java! Having finished this book, you are now the expert in Oracle and Java security.

APPENDIX A

List of Methods from OracleJavaSecure Class

Method	Listing	Access	Platform	Called By	How
Constructor	5-7	Private	Neither	none	none
byteArrayToCharArray	6-17	Package	Client	decryptSessionSecretDESPassPhrase	Internal
charArrayToByteArray	6-17	Package	Oracle	getCryptSessionSecretDESPassPhrase	Internal
checkFormat2Factor	10-22	Public	Client	setAppContext and main	Internal and Java
closeConnection	8-26	Public	Client	all applications	Java
copyAppConnections	ref 11-27	Public	Client	some applications	Java
copyPreviousConns	11-29ff	Public	Oracle	f_copy_conns	Stored Procedure
decryptSessionSecretDESPassPhrase	6-16	Private	Client	makeDESKey	Internal
distribToEMail	9-26ff	Private	Oracle	distribute2Factor	Internal
distribToPagerURL	9-27	Private	Oracle	distribute2Factor	Internal
distribToSMS	9-26	Private	Oracle	distribute2Factor	Internal

Method	Listing	Access	Platform	Called By	How
distribute2Factor	9-19	Public	Oracle	f_send_2_factor	Stored Procedure
getAAConnRole	10-56	Public	Client	all applications	Java
getAppAuthConn	10-39	Private	Client	getAAConnRole	Internal
getAppConnections	10-40ff	Public	Client	all applications	Java
getCryptConns	10-47 & 11-24	Public	Oracle	f_get_crypt_conns	Stored Procedure
getCryptData	6-9	Public	Both	f_get_crypt_data and sensitive apps	Stored and Java
getCryptSessionSecretDESAlgorithm	6-6	Public	Oracle	f_get_crypt_secret_algorithm	Stored Procedure
getCryptSessionSecretDESIterationCount	6-8	Public	Oracle	f_get_crypt_secret_count	Stored Procedure
getCryptSessionSecretDESPassPhrase	6-5	Public	Oracle	f_get_crypt_secret_pass	Stored Procedure
getCryptSessionSecretDESSalt	6-7	Public	Oracle	f_get_crypt_secret_salt	Stored Procedure
getDecryptData	6-14	Public	Client	sensitive applications	Java
getDecryptData RAW	6-15	Public	Both	f_get_decrypt_data and sensitive apps	Stored and Java
getLocRSAPubExp	5-6	Public	Client	getAppConnections and sens apps	Internal and Java
getLocRSAPubMod	5-6	Public	Client	getAppConnections and sens apps	Internal and Java
getOSUserID	8-4	Private	Client	setConnection	Internal
getOSUserName	Supplement	Public	Client	TestAppA class	Java

Method	Listing	Access	Platform	Called By	How
getRSACryptData String	6-4	Private	Oracle	getCryptSessionSecretDESAlgorithm	Internal
getRSACryptData bytes	6-4	Private	Oracle	getCryptSessionSecretDESPassPhrase, IC, Salt	Internal
isAppverOK	ref 12-6	Public	Client	Login class	Java
listConnNames	12-62	Package	Client	OJSAdmin app	Java
main	11-8	Public	Client	none	Command Prompt
makeDESKey	7-7	Public	Client	getAppConnections, getDecryptData and sens apps	Internal and Java
makeExtRSAPubKey	5-3	Private	Oracle	getRSACryptData B	Internal
makeLocRSAKeys	5-1	Private	Client	getLocRSAPubMod and getLocRSAPubExp	Internal
makeSessionSecretDESKey	6-3	Private	Both	makeDESKey, getCryptSessionSecretDESPassPhrase…, getCryptData	Internal
makeSessionSecretDESPassPhrase	6-1	Private	Oracle	makeSessionSecretDESKey	Internal
putAppConnections	10-24	Public	Client	OJSAdmin and applications	Java
putAppConnString	10-23	Public	Client	applications	Java
putAppConnString test	10-23	Public	Client	putAppConnString A, OJSAdmin and applications	Internal and Java
removeAppConnString	11-26	Package	Client	[OJSAdmin app]	Java
resetKeys	7-9	Public	Client	OJSAdmin and getAppConnections	Internal and Java
setAppContext	10-21	Public	Client	Login class and applications	Java

Method	Listing	Access	Platform	Called By	How
setAppVerConnection	10-45 & 11-9	Private	Client	getAppConnections and copyAppConnections	Internal
setConnection URL	8-24	Private	Client	setAppVerConnection and getAppAuthConn	Internal
setConnection Connection	8-24	Private	Client	setConnection URL	Internal
setConnection OracleConnection	8-25	Private	Client	setConnection Connection	Internal
setDecryptConns	10-26ff	Public	Oracle	f_set_decrypt_conns	Stored Procedure
test2Factor	12-8	Public	Client	Login class	Java

APPENDIX B

Oracle Procedures, Functions and Triggers for Oracle and Java Security

Procedure/ Function/Trigger	Schema	Package	Listing	Special	Execute By	Called By
p_log_error	appsec	app_sec_pkg	7-6		App Users (HR)	all
f_get_crypt_secret_pass	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	p_get_app_conns
f_get_crypt_secret_algorithm	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	p_get_app_conns
f_get_crypt_secret_salt	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	p_get_app_conns
f_get_crypt_secret_count	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	p_get_app_conns
f_get_crypt_data	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	App Procedures
f_get_decrypt_data	appsec	app_sec_pkg	6-11	Java Stored Proc	App Users (HR)	App Procedures

Procedure/ Function/Trigger	Schema	Package	Listing	Special	Execute By	Called By
p_appsec_errors_janitor	appsec		7-4	Autonomous		t_appsec_errors_iar
t_appsec_errors_iar	appsec		7-5	Trigger		INSERT t_appsec_errors
t_application_key_budr	appsec		11-16	Trigger		UPDATE t_application_key
f_is_sso	appsec		10-3	CURRENT_USER		p_check_role_access, f_set_decrypt_conns, p_get_app_conns
f_copy_conns	appsec	appsec_only_pkg	11-27	Java Stored Proc		p_copy_app_conns
f_is_cur_cached_cd	appsec	appsec_only_pkg	9-13, 10-4			p_copy_app_conns, p_get_app_conns
f_send_2_factor	appsec	appsec_only_pkg	9-14, 10-4	Java Stored Proc		p_copy_app_conns, p_get_app_conns
f_get_app_role	appsec	appsec_only_pkg	10-5			p_check_role_access
p_get_emp_2fact_nos	appsec	appsec_only_pkg	10-8, 11-52			distribute2Factor
p_update_2fact_cache	appsec	appsec_only_pkg	10-9			distribute2Factor
f_is_user	appsec	appsec_only_pkg	10-16			p_appver_logon
p_count_class_conns	appsec	appsec_only_pkg	10-37			setDecryptConns, copyPreviousConns