each connection string used by the application. This declaration of `connsHash` uses generics to require that the keys be `Strings` and the values be `RAW`, with the `<String, RAW>` syntax.

`HashMaps` can be created without generics; that is, without specifying the object type of the key and value, but each time you retrieve a key or value, you would need to cast it to the appropriate type. Additionally, by specifying the object types, we are assured that our application will only place objects of that type in our key and value fields. Otherwise, `HashMaps` can hold a variety of `Object` types all at once.

An Inner Class to Represent the Application

Our applications will identify themselves to this application authentication process by passing an object that implements a specific interface: the `RevLvlClassIntfc` interface. An interface is like a class, but it is hollow—it has no guts. An interface can specify a list of methods, and all classes that implement that interface will need to implement those methods; that is, they will need to have methods with the same signature (same name, arguments and return types). Hopefully, they will also provide the guts or functional code to accomplish whatever's required.

The `RevLvlClassIntfc` interface is in the *orajavsec* package, just like `OracleJavaSecure`. Also like `OracleJavaSecure`, the interface needs to exist in the Oracle database as well as on the client. `RevLvlClassIntfc` needs to be loaded into Oracle, as the first line of Listing 10-19 suggests, and needs to be distributed to developers along with *OracleJavaSecure.class*, most likely in the same *jar* file.

Listing 10-19. *Revision Level Class Interface*

```
//CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/RevLvlClassIntfc" AS
package orajavsec;

public interface RevLvlClassIntfc {
    //private static final long serialVersionUID = 2011010100L;
    //private String innerClassRevLvl = " 20110101a";
    public String getRevLvl();
}
```

■ **Note** You can find this code in the file named *Chapter10/orajavsec/RevLvlClassIntfc.java.*

There is one method in the interface, `getRevLvl()`. This returns a revision level, `innerClassRevLvl` that we can use to support multiple versions of an application. For instance, if you are moving your application data to new Oracle tables or a new Oracle database, you could update `innerClassRevLvl` in your inner class and use our processes to register the updated application and generate a new set of connection strings associated with it. Both the old application/version and the new one would be able to get their respective connections simultaneously. Also, to force everyone to move to a new version of an application and to disable the old version, we can delete the old version of our inner class from the application registry, or just delete the associated connection strings list.

We will ask the implementers of this interface (the developers) to also provide a `static long` named `serialVersionUID` that is used in object serialization (packaging for storage in the database and for transmission across the network.) We will discuss this more shortly.

Implement an Inner Class in OracleJavaSecure

For testing purposes in this chapter, we will define an inner class, `InnerRevLvlClass`, in the `OracleJavaSecure` class that implements `RevLvlClassIntfc`. To generate an inner class, just include the class definition within an existing class definition. In our case, it looks like Listing 10-20. Inside the body of the definition for `OracleJavaSecure`, we declare the inner class, `InnerRevLvlClass`.

Listing 10-20. *OracleJavaSecure Inner Class for Identity and Versioning, InnerRevLvlClass*

```
public class OracleJavaSecure {
…
    public static class InnerRevLvlClass
        implements Serializable, RevLvlClassIntfc
    {
        private static final long serialVersionUID = 2011010100L;
        private String innerClassRevLvl = "20110101a";
        public String getRevLvl() {
            return innerClassRevLvl;
        }
    }
```

■ **Note** For most of the rest of this chapter, we will be referring back and forth to Java code that you will find in the file named *Chapter10/orajavsec/OracleJavaSecure.java,* and to SQL and PL/SQL code that you will find in the file named *Chapter10/AppSec.sql*.

Notice that our inner class is designated as `public static`. The `static` is not necessary, but assures us that the class exists when the parent, `OracleJavaSecure`, is instantiated. On the other hand, the `public` designation is significant from a security standpoint. We would prefer to make this inner class private, and we could for `OracleJavaSecure.InnerRevLvlClass`. But for other independent applications, the inner classes will need to be public so that `OracleJavaSecure` on the Oracle Database can instantiate them.

Our inner class also must implement the `Serializable` interface (another required element of the template we provide to our developers.) By implementing `Serializable` (no methods are required), we are assuring that we can take our inner class, get the bytes of the object and pass them to Oracle database for storage. We can only handle objects in that way if they implement the `Serializable` interface.

The `serialVersionUID` for the class that the application hands to us must be identical to the `serialVersionUID` for the class definition in Oracle database (this discussion is continued in the next section). We set a static member variable for `serialVersionUID`, but if we didn't have that member, the JVM would calculate a value. If that value is not identical, then the class cannot be realized, and the object cannot be instantiated, and an exception will be thrown. Most often, unless we make structural changes to the class, the calculated `serialVersionUID` will be identical, but that statistic of "most often" poses a risk that we are unwilling to take.

Note that the values we provide for `innerClassRevLvl` and `serialVersionUID` are pretty much whatever we want; however, I think a form of the date with flexibility for multiple values for any one date will make a good value. If I release a new application on February 14, 2012, I might give these values:

```
private static final long serialVersionUID = 2012021400L;
private String innerClassRevLvl = "20120214a";
```

If I release a revision later that same day, it might have these changed values (for example only, these are usually modified independently and for different reasons):

```
private static final long serialVersionUID = 2012021401L;
private String innerClassRevLvl = "20120214b";
```

It may be obvious, but the `L` at the end of the value for `serialVersionUID` is part of the number and indicates that it is a `long` value. It is a way of casting what would by default be interpreted as an integer to another type of primitive value. Use an upper case `L` to keep from having the lower case `l` (ell) confused with a `1` (one). There are other *literal* value casts that consist of suffixes (for example `f` for float) and prefixes (for example `0x` for hexadecimal). Take one, as needed.

Deserialization and Version UID

Bringing an object back to life after serialization requires deserialization. But there is one huge requirement for this to happen—a class definition for the object must exist in the JVM. This is a requirement that is often overlooked or assumed in the documentation on serialization. Usually, an application is responsible for serializing an object, and an equivalent application handles the deserialization. The application has a class definition, perhaps in a separate *.class* file, or as an inner class, and so it already knows how to build an object of the type being deserialized. The serialized object does not contain all the details of the class definition; rather, it should be considered to only retain the data or state of the object when serialized. To deserialize the object in a JVM, the existing class definition must provide a skeleton on which to build the body of the object.

That all makes sense to us now, but consider this twist: We are going to deserialize the object in the Oracle database, in a place where the application does not run. To do this, we do have to have the class definition; so for each application, we will be storing the inner class definition (we'll see how we do that later). In the case of the inner class, `InnerRevLvlClass` of `OracleJavaSecure`, the class definition is there on the Oracle database because we create `OracleJavaSecure` on the server. For other applications, we will have to create the inner class on the Oracle database.

In the interim between object serialization and deserialization, some non-structural changes may be made to the class definition, and the Java compiler and runtime versions may continue to advance; but we should be able, even ten years down the road, to deserialize our object. This is where the `serialVersionUID` member variable comes into play. At the outset of use of our serialized objects, we may have no problem depending on the runtime calculation of the *suid* (another name for this value when the `serialVersionUID` member does not exist.) However, as time progresses and things change, we may begin to experience mismatches between what we've serialized and what we are attempting to deserialize into.

We can avoid problems with serial version mismatches (caused by technological progress rather than by changes to our classes) by simply encoding a `serialVersionUID` member, rather than relying on the runtime calculation.

If we do make structural changes to our class definition, we will need to do the following two things:

1. Change the `serialVersionUID` value in our class definition.
2. Store a new version of the class definition in Oracle.

Note that this situation is different from the situation where we are only encoding a new `innerClassRevLvl` in our class definition. We will do that to handle application version changes (and provide an updated list of Oracle connection strings.)

Set Application Context

You will recall from our discussion in the previous section that we are gathering all our application-specific elements in one place. We will store the `applicationID`, `appClass`, and `twoFactorAuth` code in `OracleJavaSecure` as static class members, so we needn't pass them around to all the methods where they will be referenced.

When an application initially comes to `OracleJavaSecure`, the first thing it will likely do is identify itself by calling the `setAppContext()` method, Listing 10-21. `setAppContext()` takes the `applicationID`, inner class, and `twoFactorAuth` code as arguments. The first time we run a client application, the `twoFactorAuth` code is not provided. It will not be until the user has received the two-factor code on their cell phone or other device that they will be able to return and redo this identification and proceed.

In order to request connection strings, we will have to pass muster with the application verification, `appver` security guard. That means we will have to present our inner class to the application authorization procedures. In the `setAppContext()` method, we assure that the inner class we will be presenting implements both the `RevLvlClassIntfc` interface, and the `Serializable` interface by calling the `instanceof` operator.

Listing 10-21. *Set Application Context, `setAppContext()`*

```
    private static String applicationID = null;
    private static Object appClass      = null;
    private static String twoFactorAuth = null;

    public static final void setAppContext( String applicationID,
        Object appClass, String twoFactorAuth )
    {
        twoFactorAuth = checkFormat2Factor( twoFactorAuth );
        if( null == applicationID || null == appClass ) {
            System.out.println( "Must have an application ID and Class" );
            return;
        }
        // Assure the app class has implemented our interface
        if ( !( ( appClass instanceof RevLvlClassIntfc ) &&
            ( appClass instanceof Serializable ) ) )
        {
            System.out.println(
                "Application ID Class must implement RevLvlClassIntfc" );
            return;
        }
        // Set class static member equal to what passed here at outset
        OracleJavaSecure.applicationID = applicationID;
        OracleJavaSecure.appClass      = appClass;
        OracleJavaSecure.twoFactorAuth = twoFactorAuth;
    }
```

As a side note, we have seen methods like this before where we pass in references that are the same as class member names. We typically have set the class members with a statement like this:

```
this.varName = varName;
```

Notice the difference in this case. Because we are setting static class members, there is no `this` to refer to–`this` is a reference to the current instance, but static classes are not instantiated. So we use this syntax to set static class member variables from a method arguments:

```
Class.varName = varName;
```

Format the User-Input Two-Factor Code

We introduced the `checkFormat2Factor()` method in the last chapter, but we didn't discuss its implementation. We want to assure that users who make an effort to enter the two-factor code get some leniency with regard to format. This is especially important in this chapter, and from now on because we are delivering the application ID along with the two-factor code to the user's cell phone and other devices.

We want to assure that we always deliver the two-factor code as the first item in our message. If the user includes the application name after that, we should find it easy to truncate the two-factor code at the appropriate place. If extra data comes as separate arguments on the command line, we ignore the additional arguments.

Occasionally, an old-style pager may drop non-numeric characters from a message. As a general rule, dashes are always included. Perhaps the pager might also drop spaces and include underscore characters. In preparation for any of those exigencies, and without burdening the user with our format requirements, we pass the two-factor code that comes to us in the `setAppContext()` method on to the `checkFormat2Factor()` method in Listing 10-22.

Listing 10-22. *Format User-Provided Two-Factor Authentication Code, checkFormat2Factor()*

```
public static String checkFormat2Factor( String twoFactor ) {
    String rtrnString = "";
    if( null == twoFactor ) return rtrnString;
    // Use only numeric values and insert dash after every 4 chars
    StringBuffer sB = new StringBuffer();
    int used = 0;
    char testChar;
    int twoFactLen = twoFactor.length();
    for( int i = 0; i < twoFactLen; i++ ) {
        testChar = twoFactor.charAt( i );
        if( Character.isDigit( testChar ) ) {
            sB.append( testChar );
            if( sB.length() == twoFactorLength ) {
                rtrnString = sB.toString();
                break;
            }
            // Insert dash if we have accepted a multiple of 4 chars
            used++;
            if( 0 == ( used % 4 ) ) sB.append( "-" );
        }
    }
    return rtrnString;
}
```

This method reads each character of the user-input two-factor code. If a character is not numeric, it is discarded. Numeric characters are appended to a `StringBuffer`, and after each four characters, a dash is appended. This continues until we run out of input or we reach the required two-factor code length,

twoFactorLength. If we have sufficient numeric characters, we return the StringBuffer as a String, else we return an empty string.

There is a perception among programmers, especially those (like me) who have worked extensively with perl, that checking the format of a string is best accomplished by using regular expressions. Regular expressions are a succinct way to express patterns of characters, including optional formatting. For example, this regular expression represents a SSN pattern: "^\\d{3}[-]?\\d{2}[-]?\\d{4}$". I believe pattern matching has limits that need to be recognized and respected. In checkFormat2Factor(), for instance, we are going to throw away extraneous data (words or spaces) that a user might have seen on their cell phone or pager screen and entered. Pattern matching does not work well for free-form user input like this. The second issue I have with complex regular expressions is that to read them often requires a decoder ring and a sheet of scrap paper. (The same can too often be said of perl code in general.)

Save Connection Strings from the Client Perspective

We are going to explore saving connection strings and associating them with an application from both the client side and the server side. In some cases, developers may choose to use all our security except the application authorization, so they will benefit from having an independent method for using connections strings on the client that have not been stored on Oracle database. These can even be combined with a set of connection strings that are maintained by our application authorization.

However, if a developer elects to not store his connection strings under appver's supervision, he will need to find another way to secure the password. Also, he will lose two-factor authentication, which we have delegated to the application authorization process.

Method to Put Connection Strings in the List for an Application

On the client, we have a method, putAppConnString() that we can use to add connection strings to connsHash. It takes 5 arguments (see Listing 10-23): the instance name, user name, password, host name, and port (as a String). By taking these components separately, it leaves the assembly of a connection string to us. We can assure that the connection string format is acceptable. We do take a moment to trim off any whitespace from the head and tail of each argument, calling the String.trim() method.

As an optional parameter, by way of an overloaded method, we accept a boolean value, which can direct us to test the connection string before saving it in connsHash.

We assume the connection string, as we've formatted it, is okay to be added to connsHash; however, if we've been directed to test the connection string, we may change that evaluation. If we are testing the connection string, we simply instantiate a new Connection based on the connection string and use it to query the database. If that fails, we have determined that the connection string is not good.

Listing 10-23. *Put Connection Strings in List,* ***putAppConnString()***

```
public static void putAppConnString( String instance, String user,
    String password, String host, String port )
{
    putAppConnString( instance, user, password, host, port, false );
}
public static void putAppConnString( String instance, String user,
    String password, String host, String port, boolean testFirst )
{
    instance = instance.trim();
    user = user.trim();
```

```
        password = password.trim();
        host = host.trim();
        port = port.trim();
        String key = (instance + "/" + user).toUpperCase();
        String connS = "jdbc:oracle:thin:" + user + "/" + password + "@" +
            host + ":" + port + ":" + instance;
        boolean testSuccess = true;
        if( testFirst ) {
            Connection mConn = null;
            try {
                mConn = DriverManager.getConnection( connS );
                Statement stmt = mConn.createStatement();
                ResultSet rs = stmt.executeQuery(
                    "SELECT SYSDATE FROM DUAL" );
                System.out.println( "Connection string successful" );
            } catch( Exception x ) {
                System.out.println( "Connection string failed!" );
                testSuccess = false;
            } finally {
                try {
                    if( null != mConn ) mConn.close();
                } catch( Exception x ) {}
            }
        }
        if( testSuccess ) {
            try {
                appAuthCipherDES.init( Cipher.ENCRYPT_MODE,
                    appAuthSessionSecretDESKey, appAuthParamSpec );
                byte[] bA = appAuthCipherDES.doFinal( connS.getBytes() );
                connsHash.put(key, new RAW( bA ) );
            } catch( Exception x ) {}
        }
    }
```

Finally, if we've decided we like the connection string (`testSuccess` is `true`), we add it to `connsHash`. And, like the connection strings we (perhaps) received from Oracle, we store it here in encrypted form, based on the shared password key we got from Oracle database.

The `HashMap`, `connsHash`, is keyed on the string "INSTANCE/USER" that we assembled from the instance name and user name that were provided to us. If we are calling this method with the same instance and user as an existing entry in `connsHash`, the new connection string will overwrite the old (assuming it's acceptable).

Client Call to Store List of Connection Strings on Oracle

Once we've filled out the ranks of one or more connection strings that we want to use with our application, we can call the `putAppConnections()` method to submit the `connsHash` to Oracle Database. We see this method in Listing 10-24.

Note again that we need to have already exchanged keys with Oracle database. Assuredly we have, else we wouldn't have been able to get the previously existing `connsHash` from Oracle, nor would we have been able to add new connection strings to `connsHash`—we would have nothing to submit.

We will assure that we have a connection to appver, which we can use. We test whether appVerConn is null, a reference to the connection we are using to converse with appver. If it is null, one of two problems exist: either we haven't connected yet as appver (call getAppConnections()), or we have overwritten our connection to appver, so it is no longer available to execute the update, putAppConnections(). The solution is to always call putAppConnections(), if needed, before getting one of the connection strings from the list for use in the application.

It is a trivial matter to convert a Java object, as long as it implements Serializable, into a byte array. However, it does look at bit daunting at first. Drilling down into this code, we write our object into an ObjectOutputStream, oout. The ObjectOutputStream is tied directly to a ByteArrayOutputStream, baos. After we write our object, we flush oout and close it, assuring that our entire object is delivered to baos. At that point, we call the toByteArray() method of baos to get our object out as a byte array.

Listing 10-24. *Store Connection Strings List in Oracle, putAppConnections()*

```
public static void putAppConnections(){
    OracleCallableStatement stmt = null;
    try {
        if( null == appVerConn ) {
            if( null == conn ) {
                System.out.println( "Call getAppConnections to establish " +
                    "connection to AppVer first, " +
                    "else can not putAppConnections!" );
            } else {
                System.out.println( "Connection to AppVer overwritten - " +
                    "can not putAppConnections!" );
            }
            return;
        }

        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oout = new ObjectOutputStream( baos );
        oout.writeObject( appClass );
        oout.flush();
        oout.close();
        byte[] appClassBytes = baos.toByteArray();
        baos.close();

        baos = new ByteArrayOutputStream();
        oout = new ObjectOutputStream( baos );
        oout.writeObject( connsHash );
        oout.flush();
        oout.close();
        byte[] connsHashBytes = baos.toByteArray();
        baos.close();

        stmt = ( OracleCallableStatement )conn.prepareCall(
            "{? = call appsec.appsec public pkg.f_set_decrypt_conns(?,?)}" );
        stmt.registerOutParameter( 1, OracleTypes.VARCHAR );
        stmt.setBytes( 2, appClassBytes );
        stmt.setBytes( 3, connsHashBytes );
        stmt.executeUpdate();
```

```
            String checkReturn = stmt.getString( 1 );
            if( ! checkReturn.equals( okReturnS ) )
                System.out.println( checkReturn );
        } catch ( Exception x ) {
            x.printStackTrace();
        } finally {
            try {
                if ( null != stmt )
                    stmt.close();
            } catch ( Exception y ) {}
        }
    }
```

We perform this same operation both on our application inner class, `appClass`, and the `connsHash` `HashMap` object, which also implements `Serializable` (as do the `String` keys and `RAW` values held by `connsHash`).

We submit our application class and `connsHash` to Oracle Database as byte arrays. We send them to the `f set decrypt conns` Java stored procedure (function). That function merely calls Java on the Oracle database side, passing these objects to the `setDecryptConns()` method on the Oracle database, discussed in detail in the sections that follow.

Save Connection Strings from the Server Perspective

On the Oracle database side, we go through a rather elaborate process in order to get the `connsHash` table of connection strings into the shape we want before storing them. You see, when we submit them to Oracle, they are encrypted with the secret password key that is unique to this session. If we were to store them as is, we could never decrypt them after this session closes; so we will decrypt them before storage on Oracle database. We will explore and apply encryption to data on disk when we get to Chapter 11. The decryption process, however, is preceded by an equally elaborate process to assure that the application class being submitted is appropriate for either overwriting an existing entry in the registry, or inserting a new one.

Function to Call Java to Decrypt the List of Connection Strings

From the client, we call a Java stored procedure, `f set decrypt conns` on Oracle Database to deliver the connections `HashMap`. `f set decrypt conns` simply passes the objects to our Java code on the Oracle database for processing. Java stored procedures can be seen simply as a doorway to pass data through, and a call to methods in Java on the Oracle database. We add this function to a new package, `appsec public pkg`, see Listing 10-25.

Listing 10-25. *Function Call to Decrypt List of Connection Strings, f_set_decrypt_conns*

```
CREATE OR REPLACE PACKAGE BODY appsec.appsec public pkg IS

    FUNCTION f set decrypt conns(
        class instance RAW, connections RAW )
    RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'orajavsec.OracleJavaSecure.setDecryptConns( oracle.sql.RAW, oracle.sql.RAW ) return
java.lang.String';
```

```
...

GRANT EXECUTE ON appsec.appsec public pkg TO PUBLIC;
```

For now, we will be proxying from a variety of users through `appver` to submit the `connsHash` to `f set decrypt conns`, so we will grant execute to `PUBLIC` on the package `appsec public pkg`.

Obviously, we do not want just anybody submitting objects to this function, so we need to protect it in one or more of the following ways:

1. Revoke the `GRANT EXECUTE` on `f set decrypt conns` from `PUBLIC`, and add grants only to those users (OS user names) who need access.
2. Protect the code used to accomplish this function—separate it out into an administrative application.
3. Implement some additional test in this code, possibly checking for a user's existence (listing) in an Oracle table, built for this purpose.

We will do all of these things in Chapters 11 and 12.

Method to Store List of Connection Strings for Application

Certain actions that we might take are considered risky or uncertain in Java. Java is a strongly typed language, so it doesn't abide uncertainty in object type identity. However, in cases where we are in control of both the origination and the receipt of the object, we can ignore any warnings in this regard and proceed.

In the `setDecryptConns()` method given in Listings 10-26 through 10-35, we engage in such an endeavor. We will be reading an object from an `ObjectInputStream` and then treating it as if we know what kind of object it is. We do this twice. First, we read in the application inner class object and then call its `getRevLvl()` method, assuming it has the wherewithal to respond. The second case is when we read in the `connsHash` object and cast it as a `HashMap`.

When compiling, *javac* will report to us that there are "unchecked or unsafe operations". We can, by way of the `@SuppressWarnings( "unchecked" )` annotation, ask *javac* not to bother us. That annotation applies directly to the method that follows, and only to that method. Notice in Listing 10-26 that there is no punctuation between the annotation and the method declaration. Unfortunately, those annotations are not accepted in the Oracle JVM, so we need to comment them out and just live with the compile-time warnings.

Listing 10-26. *SuppressWarnings() Annotation, setDecryptConns()*

```
@SuppressWarnings( "unchecked" )
public static String setDecryptConns( RAW classInstance, RAW connections ) {
    String rtrnString = "function";
    OracleCallableStatement stmt = null;
    try {
```

Build a Class from a Byte Array

Getting a class back from the `RAW` data type that we received on the Oracle database is the flipside of what we saw earlier when we converted our class objects into byte arrays. The first step here in Listing 10-27,

part of setDecryptConns() is to get a byte array from the RAW that we are calling classInstance. We feed that byte array to a ByteArrayInputStream, bAIS. Then we instantiate an ObjectInputStream, oins, coupled to bAIS. We call the readObject() method of oins to rebuild an Object named classObject. After closing the streams, we get an instance of Class, providedClass for the classObject.

Listing 10-27. *Build a Class from a Byte Array*

```
byte[] appClassBytes = classInstance.getBytes();
ByteArrayInputStream bAIS = new ByteArrayInputStream( appClassBytes );
ObjectInputStream oins =
    new ObjectInputStream( bAIS );
Object classObject = oins.readObject();
oins.close();
Class providedClass = classObject.getClass();
```

Use Java Reflection to Call Method

With the Class object, we can use *reflection* to get the name of the class and even to call its methods. This is also part of setDecryptConns() method. Getting the class name is a single call to the getName() method. To call the getRevLvl() method in the inner class, we get a Method object, classMethod from providedClass by calling the getMethod() method, as shown in Listing 10-28. Then we call the invoke() method of classMethod, passing the classObject we got in Listing 10-27, to retrieve the actual return value from getRevLvl(). Cast the return Object value as a String.

Listing 10-28. *Call Methods Through Reflection*

```
String className = providedClass.getName();
Method classMethod = providedClass.getMethod( "getRevLvl" );
String classVersion = ( String )classMethod.invoke( classObject );

// Do this once we get to Oracle
// Before we store any class, let's assure it has a package (.)
// noted before being an inner class ($) - our planned requirements
if( -1 == className.indexOf( "." ) ||
    className.indexOf( "$" ) < className.indexOf( "." ) )
    return "App class must be in a package and be an inner class!";
```

We are going to place some requirements on developers for their application inner classes. First, we are going to require that their inner classes are declared public. We would also prefer that they are declared static, but don't require that (however, it will be included in the template code we provide them). Next, we require that they are inner classes and that their containing classes are contained in a package. We can assure these requirements are met by assuring that the name of the inner class includes a period, "." character, indicating a package, and that it also includes a dollar sign, "$" character sometime after the period, indicating an inner class of a class in a package. The javac compiler concatenates the class and inner-class names, adding a dollar sign between them, to formulate the fully qualified inner-class name.

Find Out Whether This Class Has Been Registered

If we have never seen this class before, then we will assume this is the initiation, and we will register a new application by inserting the class in the `v app conn registry` view. We find out if we have ever seen this class before by selecting from the view where classes are registered with that same name and version. The `p count class conns` Oracle procedure accomplishes that for us. See Listing 10-29.

Listing 10-29. Determine if the Application Class Is Already Registered

```
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p_count_class_conns(?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.NUMBER );
stmt.setString( 1, className );
stmt.setString( 2, classVersion );
stmt.setInt(    3, 0 );
stmt.executeUpdate();
```

If it turns out that `p count class conns` tells us there are no classes registered by that name / version, then we go on to insert; otherwise we need to check if the class we just received is equal to the class that we have registered by that name. If it's "equal," we will overwrite the existing, stored `connsHash`; but if not equal, we are dealing with an imposter.

SOME DEVELOPER GOTCHA'S AND RESOLUTIONS

Unfortunately, our application developers can be victims of their own actions. If the developers change the code of their inner class without changing the version number, this will cause our equality test to fail. In that case, the developer should change both `serialVersionUID` and the `innerClassRevLvl` in his inner class, register it through our processes, and create a new list of connection strings or copy the connection strings from the previous version.

Another way the application developer might shoot himself in the foot, so to speak, is by moving his inner class around in his code. For instance, if he moves his inner class outside the public class definition, it becomes an outer class, or if he moves it out of the main body of the class into a method (a perfectly acceptable move, technically), the package and hence the class name changes to reflect that move. In these cases, the inner class will be seen as a new entity and registered, but won't have any associated connection strings until the developer rebuilds the list for the new version. In this case, he cannot copy his connection strings list from the previous version, because this is considered to be a new class, being found in a new path (and it may even have the same version number as the previous class).

Get the Application ID Class and HashMap List of Connections

Back to the task at hand in `setDecryptConns()`, we call the `p get class conns` stored procedure to get our registered class and the `connsHash` associated with this class name and version. In Listing 10-30 we handle the `connsHash` as a `BLOB`. You will recall that in the `t app conn registry` table definition, we define it as a `BLOB`; that allows us to store a `connsHash` object that is larger than 2K bytes. When we defined the

procedure `p get class conns`, we also specified the fourth parameter as a `BLOB` by reference to its definition on the table, like this:

```
m connections    OUT v app conn registry.connections%TYPE
```

Elsewhere in our code, we handle the `connsHash` as a `RAW`. Interestingly, in PL/SQL, that is in code, a `RAW` can attain 32K in size, but in storage in an Oracle table, can only be 2K. It is easy to handle the `BLOBs` as `RAWs` in code, but why not just always call a `BLOB` a `BLOB`? `BLOBs` are tightly coupled to either a database table or to a connection. They do not fare well in transmission, so we rely on the `RAW` data type.

Listing 10-30. *Get the List of Connections from Oracle Database As a BLOB*

```
    if( stmt.getInt( 3 ) == 0 ) {
        // Do insert!
    } else {
        // Assure provided instance and cached, if same version, are equal
        // NOTE: handling BLOBs with getBytes and setBytes is new to 11g
        stmt = ( OracleCallableStatement )conn.prepareCall(
            "CALL appsec.appsec only pkg.p_get_class_conns(?,?,?,?)" );
        stmt.registerOutParameter( 3, OracleTypes.RAW );
        stmt.registerOutParameter( 4, OracleTypes.BLOB );
        stmt.setString( 1, className );
        stmt.setString( 2, classVersion );
        stmt.setNull(   3, OracleTypes.RAW );
        stmt.setNull(   4, OracleTypes.BLOB );
        stmt.executeUpdate();

...

    byte[] cachedBytes = stmt.getBytes(3);
    oins = new ObjectInputStream( new ByteArrayInputStream(
        cachedBytes ) );
    classObject = oins.readObject();
    oins.close();
```

Pulling the inner-class object and the `connsHash` out of Oracle database is a two-step process. We get the bytes of the `RAW` and `BLOB`, respectively. Then we move the bytes through streams, as we've seen before, to reconstitute our objects. That process is shown for the inner class in Listing 10-30.

Test for Class Equality

We arrive at a crossroads. We have already instantiated an object (from the byte array that the client provided) from which we got the name and version number. Now we are going to instantiate another class from the bytes stored as a `RAW` in our registration table. The Oracle JVM will complain in a big way if these two objects are not identical: it will throw an `InvalidClassException`.

We will go ahead and assure that the object handed to us by the client and the object stored in our registration table are equal by calling the `equals()` method, as shown in Listing 10-31. In Java, this means more than just that the objects have equivalent values for their variables—it means they are based on the same model in memory. They are, at the core, from one and the same class.

Listing 10-31. *Test for Class Equality*

```
    Class testClass = classObject.getClass();

        if( testClass.equals( providedClass ) ) {
            // further tests are unnecessary
        } else return "Failed to setDecryptConns()";
    }
```

Decrypt Connection Strings for Storage and Reuse

Once we have the identity questions settled (still in the `setDecryptConns()` method), whether we are poised to insert or overwrite a registration entry, we are left to deal with the `connsHash` we received from the client. Currently, the connection strings are encrypted with our shared session DES key—a key that will disappear when the current session closes. That would be an unusable state to store them in. The next user in our application would not be able to read them, nor would we in our own next session.

So, we will decrypt all the connection strings and store them unencrypted. When the next session comes to get the connection strings for this application, we will encrypt them with that session's key before delivery. Notice in Listing 10-32, when we set the `cryptConnsHash` member that we cast it as a `HashMap<String, RAW>`, no questions asked. This kind of blind trust, which is appropriate at this point, is what the Java compiler warns about with the "unchecked" warning. Notice also that we will transition from a `HashMap<String, RAW>` to a new `HashMap<String, String>`.

Listing 10-32. *Cast Encrypted List of Connections and Prepare to Decrypt*

```
    oins = new ObjectInputStream( new ByteArrayInputStream(
        connections.getBytes() ) );
    classObject = oins.readObject();
    oins.close();
    HashMap<String, RAW> cryptConnsHash =
        (HashMap<String, RAW>)classObject;

    HashMap<String, String> clearConnsHash =
        new HashMap<String, String>();
    oins.close();
```

We initialize our shared secret password cipher to do decryption (see Listing 10-33), then walk through the `cryptConnsHash` `HashMap` to decrypt each value. We store each decrypted value, using the same key in our new `clearConnsHash`. With classes like the `HashMap`, from the *Collections* classes, we can traverse their members using the for each syntax. You can read our `for` statement as "for each key in the `cryptConnsHash` set of keys."

Listing 10-33. *Decrypt Each Connection String and Save to New List*

```
    cipherDES.init( Cipher.DECRYPT_MODE, sessionSecretDESKey, paramSpec );
    for( String key : cryptConnsHash.keySet() ) {
        // Decrypt each one
        clearConnsHash.put( key,
            new String(
```

```
                cipherDES.doFinal(
                    (cryptConnsHash.get( key )).getBytes()
                )
            )
        );
    }
```

That syntax for storing our decrypted connection keys looks a bit complex, but we are simply putting a new entry in the `clearConnsHash` using the key we got in this pass through the for each loop and a new `String`. The new `String` is derived from the bytes we got back from decrypting the value associated with this same key in the former `cryptConnsHash`.

Store the connsHash for this Application

We will use a now-familiar process to get a byte array from the `clearConnsHash HashMap`. This is shown in Listing 10-34.

Listing 10-34. *Get Byte Array of List of Connection Strings*

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oout = new ObjectOutputStream( baos );
oout.writeObject( clearConnsHash );
oout.flush();
oout.close();
byte[] connsHashBytes = baos.toByteArray();
baos.close();
```

The final step in `setDecryptConns()` is to store the new `connsHash` in the `v app conn registry` view. We do that by passing the byte arrays for our application ID class and the `clearConnsHash` to the `p set class conns` procedure in Listing 10-35. This is another of our procedures that handles the `connsHash` as a `BLOB`. In Oracle Database 11g, we are able to get and set `BLOBs` from Java using the `Statement.getBytes()` and `.setBytes()` methods.

Listing 10-35. *Store List of Connection Strings in Oracle*

```
// NOTE: handling BLOBs with getBytes and setBytes is new to Oracle Database 11g
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.appsec only pkg.p set class conns(?,?,?,?)" );
stmt.setString( 1, className );
stmt.setString( 2, classVersion );
stmt.setBytes(  3, appClassBytes );
stmt.setBytes(  4, connsHashBytes );
stmt.executeUpdate();
```

Oracle Procedure to Set Values in the Application Registry

The `p set class conns` procedure has three sections. This is shown in Listing 10-36. The first gets a count of records that already exist with the specific class name and version number. If there are 0, we will insert a new record in the second section; and if 1, we will update the existing record in the third. In

the same way as in `p get class conns`, you can see that we handle `m connections` as a `BLOB` by reference to `v app conn registry.connections%TYPE`.

Listing 10-36. *Procedure to Set Values in Application Registry, p_set_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;
BEGIN

    SELECT COUNT(*) INTO v_count

        FROM v app conn registry
        WHERE class name = m class name
        AND class version = m class version;

    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

        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 p set class conns;
```

Oracle Procedures to Get Entries from the Application Registry

We will look briefly at the Oracle stored procedures we use for getting data from `v app conn registry`. There are no surprises here. We have established these procedures so that we do not need to have any dynamic SQL queries in our Java code.

Find if a Registry Entry Exists for this Application

`p count class conns` returns an integer number that indicates how many records exist in `v app conn registry` with this specific class name and version number. Because records in that view are keyed on those two columns, we expected the count to only be 0 or 1. The code is in Listing 10-37.

In review, we use this to determine whether we need to check the identity of a class provided by the client against a registered class, or whether we can simply insert this as a new class. We don't have any prejudice against new applications being registered, if the application class meets our standards.

Listing 10-37. *Find Existing Application Registry Entry*

```
PROCEDURE p count class conns(
    m class name         v app conn registry.class name%TYPE,
    m class version      v app conn registry.class version%TYPE,
    m count          OUT NUMBER )
IS BEGIN
    SELECT COUNT(*)
    INTO m count
    FROM v app conn registry
    WHERE class name = m class name
    AND class version = m class version;
END p count class conns;
```

In hindsight, this was a good candidate for an Oracle stored function. It returns a single value. As a procedure, we return a value through an `OUT` parameter.

Get the List of Connection Strings for a Registered Application

`p get class conns` gets the application ID class and associated `connsHash` from `v app conn registry` for a specific class name and version number. This is shown in Listing 10-38. Not obviously, but again by way of referring to `m connections` as type `v app conn registry.connections% TYPE`, we are handling this column as its native type, a `BLOB`.

Listing 10-38. *Get List of Connection Strings from Registry*

```
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 v app conn registry
    WHERE class name = m class name
    AND class version = m class version;
END p get class conns;
```

Get an Application Connection String: The Java Client Side

There are so many administrative tasks to address with this code that the practical application can be too easily forgotten. We turn now to that practical application. A client application wants to do work using Oracle database. The app uses our code to accomplish security—SSO, two-factor authentication and encryption, along with good Oracle and Java secure programming practices.

To make this easy on the developer for that application, we give the developer just a bit of template to follow to make all this happen. We have him do the following four things:

1. Include the *OracleJavaSecure.class* and *RevLvlClassIntfc.class* files in his `CLASSPATH` (we provide a *jar* file).
2. Write an Application ID inner class that implements `RevLvlClassIntfc` and `Serializable`.
3. Call `OracleJavaSecure.setAppContext()` method, passing his application name, ID class, and two-factor authentication code (yes, he will have to handle the loop from the first request to make a second call with the two-factor code).
4. Ask `OracleJavaSecure` for the Oracle connections he needs by calling the `getAppAuthConn()` method.

In the end, we are able to provide some Oracle connections to the developer's application. Those connections are based on connection strings (including passwords) that are transmitted and maintained securely, in encrypted form.

Get an Oracle Connection from the List for an Application

Our `getAppAuthConn()` method provides connections based on the list of connection strings that already exists in memory. However, this is a great starting place, because if the connection strings do not already exist in memory, `getAppAuthConn()` calls the `getAppConnections()` method to retrieve them form Oracle database, based on the application ID class. This is shown in Listing 10-39.

We also test whether `appAuthSessionSecretDESKey is null. This will be the case when we call this method for the first time—we will not have generated or received our two-factor authentication code. If it is null, we return a null, which lets the application know it, needs to loop and let the user come again with a two-factor code. When we have provided the two-factor code and we call this method again, then we will have exchanged keys and appAuthSessionSecretDESKey will not be null.`

Listing 10-39. *Get an Oracle Connection from the List, getAppAuthConn()*

```
public static OracleConnection getAppAuthConn( String instance, String userName ) {
    OracleConnection mConn = null;
    try {
        if( null == connsHash ) getAppConnections();
        // If we entered without twoFactorAuth, apAuth...DESKey is null
        if( null == appAuthSessionSecretDESKey ) return mConn;
        instance = instance.trim();
        userName = userName.trim();
        String key = ( instance + "/" + userName ).toUpperCase();
        appAuthCipherDES.init( Cipher.DECRYPT_MODE, appAuthSessionSecretDESKey,
            appAuthParamSpec );
        mConn = setConnection( new String( appAuthCipherDES.doFinal(
            connsHash.get( key ).getBytes() ) ) );
```

We need to observe two facts about this method. First, it returns an `OracleConnection`, not a connection string. In fact, we discard the connection string in relatively short order to reduce the possibility of exposing the clear-text password. Second, the connections we return are configured to proxy through an application user in order to pass further SSO testing.

We get the specific connection string that was requested by concatenating the instance name and user name that were requested in the arguments to this method call. As with all data coming from

elsewhere, we need to get it in shape (validate and format it) before use. We trim the white space at the head and tail of each argument and uppercase the concatenated key.

With that key in hand, we are ready to get the required connection string from `connsHash`, but recall that the strings are encrypted. So we set `appAuthCipherDES` to decrypt mode using the shared secret password key.

We then stack several calls together. We get the encrypted connection string from `connsHash` based on our key. Then we get the byte array and pass it to the `Cipher` for decryption. We create a new `String` based on the decrypted bytes, and pass the clear-text connection string to the `setConnection()` method, which returns an `OracleConnection`.

By stacking our method calls in this way, we do not need to identify method member variables for each step along the way. In this case, we could have an additional `RAW`, two `byte` arrays, and a `String` member variable.

Get List of Connection Strings from Oracle Database to Client App

Although we may instruct the developers to call the `getAppAuthConn()` method, the `getAppConnections()` method gets called behind the scene. Behind the scenes, we get the list of connection strings from the Oracle database, then the application calls `getAppAuthConn()` to get individual Oracle connections based on those strings. `OracleJavaSecure` does all the heavy lifting, decrypting the string and making the connection to Oracle, before handing the connection to the application (Listings 10-40 through 10-44).

Start Our Method to Get Connection Strings from Oracle

Here again, we are in control of a procedure that the Java compiler warns us is unchecked and /or unsafe. We cast the object we get from Oracle database as the `connsHash HashMap` without checking that it fits the bill—if we were wrong, a `ClassCastException` would be thrown. Because we control both the source and the receipt of this object, we are justified in ignoring that warning. We could use the `SuppressWarnings()` annotation (see comment in Listing 10-40) to keep the compiler from complaining, but the Oracle JVM compiler does not accept that, so we will live with the compile-time warning.

Listing 10-40. *Get Connection Strings from Oracle, getAppConnections()*

```
//@SuppressWarnings( "unchecked" )
public static void getAppConnections() {
    OracleCallableStatement stmt = null;
    try {
        if( null == appVerConn ) setAppVerConnection();
```

Before we go any further, we check to see if we've got a connection to `appver`, calling `setAppVerConnection()` to create one if needed.

Call Stored Procedure to Get Application List of Connection Strings

Our call from `getAppConnections()` to `p get app conns` is easily one of the most complex Oracle stored procedure calls we make, but only because of the variety and scope of data we are exchanging. There is really nothing new here in Listing 10-41. There are a dozen arguments to the procedure: five `IN` and seven `OUT`. But two of those `OUT` arguments are for error handling.

One of the `IN` parameters is a byte array representing the application ID class. We get this byte array through two Streams classes, as we have seen before in this chapter. The other `IN` parameters are the

artifacts of our local RSA public key, the modulus and exponent, and the two-factor authentication code (if provided).

Our OUT parameters, besides the error message, are the four encrypted artifacts of our shared secret password key and the connsHash object associated with the application ID object that we are submitting.

Listing 10-41. *Procedure Call to Get List of Application Connection Strings, p_get_app_conns*

```
    stmt = ( OracleCallableStatement )conn.prepareCall(
        "CALL appsec.appsec public pkg.p get app conns(?,?,?,?,?,?,?,?,?,?,?,?,?)" );
    stmt.registerOutParameter( 5, OracleTypes.RAW );
    stmt.registerOutParameter( 6, OracleTypes.RAW );
    stmt.registerOutParameter( 7, OracleTypes.RAW );
    stmt.registerOutParameter( 8, OracleTypes.RAW );
    stmt.registerOutParameter( 9, OracleTypes.RAW );
    stmt.registerOutParameter(11, OracleTypes.NUMBER );
    stmt.registerOutParameter(12, OracleTypes.VARCHAR );
    stmt.setString( 1, locModulus );
    stmt.setString( 2, locExponent );
    stmt.setString( 3, twoFactorAuth );
    stmt.setBytes(  4, appClassBytes );
    stmt.setNull(   5, OracleTypes.RAW );
    stmt.setNull(   6, OracleTypes.RAW );
    stmt.setNull(   7, OracleTypes.RAW );
    stmt.setNull(   8, OracleTypes.RAW );
    stmt.setNull(   9, OracleTypes.RAW );
    stmt.setString(10, applicationID );
    stmt.setInt(   11, 0 );
    stmt.setNull(  12, OracleTypes.VARCHAR );
    stmt.executeUpdate();
...
        if( null == stmt.getRAW( 9 ) ) {
            System.out.println( "Please rerun with two-factor Auth Code!" );
            return;
        }
        if( null == sessionSecretDESKey ) {
            makeDESKey( stmt.getRAW( 9 ), stmt.getRAW( 8 ),
                stmt.getRAW( 6 ), stmt.getRAW( 7 ) );
```

We check to see whether any error is being reported. If not, we test one of the values returned as our shared password key artifacts, `stmt.getRAW( 9 )`. If it is null, we assume that the Oracle database has just now sent a two-factor code and must wait until the client application returns with a two-factor code to proceed. We ask the client to rerun this method with a two-factor code and exit (return from) this method.

Based on the artifacts of our shared password key, we build the key by calling the `makeDESKey()` method. The test of whether the `sessionSecretDESKey` is currently `null` is unnecessary during normal operation, but feels more complete. I can't think of an instance where we would arrive here and the `sessionSecretDESKey` not be `null`.

Use Static Class Members to Retain APPVER Connection and Keys

We establish several static class members in Listing 10-42 to retain the appver connection and session-specific keys. These keys, generated for the session for appver use, will continue to be required when the application later attempts to decrypt connection strings in connsHash for application use. Those connection strings were encrypted with the shared password key associated with the appver session.

Listing 10-42. *Static Class Members to Retain Application Verification Decryption Keys*

```
private static OracleConnection appVerConn = null;
private static byte[] appAuthSalt;
private static int appAuthIterationCount;
private static char[] appAuthDESPassPhraseChars;
private static AlgorithmParameterSpec appAuthParamSpec;
private static String appAuthSessionSecretDESAlgorithm;
private static SecretKey appAuthSessionSecretDESKey;
private static Cipher appAuthCipherDES;
```

You may want to refer to Chapter 3 where we discuss objects, static members, pointers, and references. Because our primary encryption keys and all the artifacts and related members are static, we cannot just assign a new member name to refer to and retain them while we point the primary references at a new instance. We need to set our new static members, those retaining the appver session data, to a new value, referencing a different place in memory. Within the getAppConnections() method in Listing 10-43, we set those retainer members to copies or clones of the current key artifacts. We create new instances where that process is supported.

Listing 10-43. *Set Static Class Members for Application Verification Keys*

```
// Cant just set new pointers to existing members
// Since static, updates to one will update both
// Must instantiate, clone or copy values
appAuthSalt = salt.clone();
appAuthIterationCount =
    (new Integer( iterationCount )).intValue();
appAuthDESPassPhraseChars =
    sessionSecretDESPassPhraseChars.clone();
appAuthParamSpec = new PBEParameterSpec( appAuthSalt,
    appAuthIterationCount );
KeySpec keySpec = new PBEKeySpec( appAuthDESPassPhraseChars,
    appAuthSalt, appAuthIterationCount );
appAuthSessionSecretDESAlgorithm =
    new String( sessionSecretDESAlgorithm );
appAuthSessionSecretDESKey = SecretKeyFactory.getInstance(
    appAuthSessionSecretDESAlgorithm ).generateSecret( keySpec );
appAuthCipherDES = Cipher.getInstance(
    appAuthSessionSecretDESKey.getAlgorithm() );
resetKeys();
```

At the end of our effort to retain those members for further use by appver in updating the connsHash for this application, and for decrypting the connection strings in connsHash, we call the resetKeys() method, which points all our primary keys and artifacts at null. We first saw resetKeys() in Chapter 7, where we used it in testing. Here it is identical, with one exception. We do not set the existing

sessionSecretDESKey to null because, by experimentation, we have determined that doing so invalidates the appAuthSessionSecretDESKey. For that reason, we are going to modify several methods to test sessionSecretDESPassPhraseChars for null, instead of testing sessionSecretDESKey for null: getCryptData(), getDecryptData() and the set of methods getCryptSessionSecretDESPassPhrase() / Algorithm / Salt / IterationCount. I know some would prefer I not elaborate on gotcha's and tests and alternate scenarios, but the goal here is not just to develop an application, but also to develop an understanding. If we don't explore these issues here, you will be left to find them out on your own, which is not bad, but can be time-consuming.

Get the List of Connection Strings

From the RAW bytes returned as the application connsHash object, we generate the object by passing the bytes through a ByteArrayInputStream and an ObjectInputStream. If the resultant object is not null, we cast the object as a HashMap<String, RAW>, as shown in Listing 10-44. This is where our compiler reports an "unchecked" warning. If, however, the object we got from Oracle database is null, we assume that no connsHash has been stored yet for this registered application, and we set connsHash to a new, empty HashMap<String, RAW>. After one of those scenarios, we are then free to put new connection strings into connsHash via the putAppConnString() method, and to store them in the v app conn registry view via the putAppConnections() method.

Listing 10-44. *Cast Connections List Object from Oracle Database as a HashMap*

```
if( classObject != null ) {
    connsHash = (HashMap<String, RAW>)classObject;
} else {
    connsHash = new HashMap<String, RAW>();
}
```

Establish a Connection for Application Verification Processes

I have reservations about the form of the setAppVerConnection() method that we have in this chapter—see Listing 10-45. I have already intimated that it is our plan to have this connection available to all, like a bouncer in a night club, treating his user name and password as little more than data, but that is not how I want to leave it.

It's just that this chapter is already quite weighty in scope, and I have a number of rather lengthy considerations in this regard that I feel will be best delayed until the next chapter. Please continue on to the next chapter for that discussion.

Listing 10-45. *Method to Set Application Verification Connection, setAppVerConnection()*

```
private static void setAppVerConnection() {
    setConnection( "jdbc:oracle:thin:appver/password@localhost:1521:orcl" );
    appVerConn = conn;
}
```

For now, ignore the man behind the curtain. This is reminiscent of passwords embedded in our application code; something we hope to get away from.

Notice also the last line of Listing 10-45. We set a static class member (see also Listing 10-42) to retain the application verification connection.

Get a List of Application Connection Strings: The Server Side

Here we are at that foreboding Oracle stored procedure with a dozen IN and OUT parameters, p get app conns. But that is not what is notable about this procedure. Rather, the internal workings are what we need to give attention to. However, even those are familiar.

If we pass SSO, then we deal with two-factor authentication. If the user did not submit a two-factor code, we call the f send 2 factor function, passing our validated user name and the application id in order to create, distribute and cache (store in a table) a two-factor code for this user in this application.

However, if the user did submit a two-factor authentication code, then we call the f is cur cached cd function, passing the validated user, the application id and the two-factor code. If the two-factor code equals the one cached for this user and this application, then we proceed to set return values for the secret password key, and we call f get crypt conns to return the encrypted list of connection strings.

We will add p get app conns to the appsec public pkg package since it gets called by any user using our proxy connection. The core code from this procedure is displayed in Listing 10-46.

Listing 10-46. *Get List of Connection Strings to Return from Oracle, p get app conns*

```
return user := f_is_sso( m app user );
IF( return user IS NOT NULL )
THEN

    IF( m_two_factor_cd IS NULL )

    THEN
        m err txt := appsec only pkg.f_send_2_factor( return user, m application id );
    ELSIF( appsec only pkg.f_is_cur_cached_cd( return user, m application id,
        m two factor cd ) = 'Y' )
    THEN
        secret pass salt :=
            app sec pkg.f get crypt secret salt( ext modulus, ext exponent );
        secret pass count :=
            app sec pkg.f get crypt secret count( ext modulus, ext exponent );
        secret pass :=
            app sec pkg.f get crypt secret pass( ext modulus, ext exponent );
        secret pass algorithm :=
            app sec pkg.f get crypt secret algorithm(ext modulus, ext exponent);
        m_crypt_connections := appsec_only_pkg.f_get_crypt_conns( m_class_instance );
    ELSE
        -- Wrong two-factor code entered
        RAISE NO_DATA_FOUND;
    END IF;
    app sec pkg.p log error( 0, 'Success getting App Conns, ' || return user );
ELSE
    app sec pkg.p log error( 0, 'Problem getting App Conns, ' || return user );
END IF;
```

There are a couple ways for this process to exit with a failure. If the user's connection/session fails to pass our SSO requirements, then we log the error "Problem getting App Conns" and return without sending a two-factor code. Something is seriously wrong with that user, and we don't want to deal with him at all. If however the user is good (passes SSO) but he submits a bad or old two-factor code, then we

raise a NO DATA FOUND exception, which we log here and report back to the application. In both these error cases, we exit without returning a list of connection strings.

The getCryptConns() method is called from the f get crypt conns Java stored procedure. f get crypt conns passes the application ID object through from p get app conns (Listing 10-46) to Java on Oracle database.

In getCryptConns() method, we will get the connsHash object that is associated with the application ID object from the v app conn registry view. Then we will encrypt each clear-text connection string in the connsHash before delivering it to the client, via the p get app conns procedure.

This is the third of three methods we discuss that evoke "unchecked" warnings from the Java compiler. Here we cast the alleged connsHash object coming out of v app conn registry, sight unseen, as a HashMap<String, String>. We also presume the application ID object provided to us is an implementation of RevLvlClassIntfc, on which we call the getRevLvl() method. Both of those actions are questionable in the eyes of the compiler, but we know all the parties involved and are doing what we truly want.

This method, getCryptConns() mirrors the functionality we've already discussed in setDecryptConns(). We are going to skip the description of the inner workings, except that we will point out the code used to encrypt each connection string in Listing 10-47. Recall that the connsHash stored in Oracle Database is stored in clear-text form, as a HashMap<String, String>, clearConnsHash. We use the session secret password key to encrypt the connection strings and place them in a new HashMap<String, RAW>, cryptConnsHash. Oracle database returns the encrypted connection strings to the client application.

Our Cipher is set to encrypt mode. Then we walk through all the keys in clearConnsHash using the for each syntax.

Listing 10-47. *Encrypt Each Connection String in the List*

```
cipherDES.init( Cipher.ENCRYPT_MODE, sessionSecretDESKey, paramSpec );
for( String key : clearConnsHash.keySet() ) {
     // Encrypt each one
    cryptConnsHash.put( key,
        new RAW(
            cipherDES.doFinal(
                (clearConnsHash.get( key )).getBytes()
            )
        )
    );
}
```

The stacked method calls get the connection string from clearConnsHash that is associated with the key we got in the for each loop. We pass that String to cipherDES for encryption using the secret password key. We then create a new RAW from those encrypted bytes, and we put the RAW into cryptConnsHash, keyed with the same key value. At the end of this method, we will return cryptConnsHash from Oracle Database to the client application.

Test Application Authentication, Phase 1

As in the last chapter, here again we need to edit the code to provide our company-specific addresses for two-factor authentication.

■ **Note** Edit the code found in the file named *Chapter10/orajavsec/OracleJavaSecure.java*.

```
    private static String expectedDomain = "ORGDOMAIN" ; // All Caps
    private static String comDomain = "org.com";
    private static String smtpHost = "smtp." + comDomain;
    private static String baseURL =
        "http://www.org.com/servlet/textpage.PageServlet?ACTION=O&NEWPID=";
```

We also have an embedded password in the `setAppVerConnection()` method and several more in the `main()` method. Change each of those to the appropriate password for `appver` and other users. Also change other connection string components in `setAppVerConnection()` and each `putAppConnString()` method call: the server name, port number, and instance name.

Get New Structures into Oracle

Load the new *orajavsec/OracleJavaSecure.java* code into Oracle Database. Again, uncomment the top line that begins "CREATE OR REPLACE AND RESOLVE JAVA…" and execute it in your SQL client (SQL*Plus, SQL Developer, JDeveloper, or TOAD). Remember to set the role and set define off, if using a SQL client that tries to do variable substitution, as documented in the comments:

```
// First
//      SET ROLE APPSEC ROLE;
// Also having ampersands in the code without substitution variables
//      SET DEFINE OFF;
// To run in Oracle, search for and comment @Suppress
```

Next, execute all the commands in *AppSec.sql* and *SecAdm.sql* files. Execute the commands in that order, because of dependencies in *SecAdm.sql*.

■ **Note** You can find these scripts in the files named *Chapter10/AppSec.sql* and *SecAdm.sql*.

Review Steps of Testing

In order to test our application authentication, we are going to take the following steps:

- Set our application context: application ID, inner class, and two-factor code.
- Call `getAppConnections()` to get our list of connection strings for this application—the first time will register our application.
- Call `putAppConnections()` to upload our list of connection strings to Oracle.
- Call `getAppAuthConn()` to get a specific connection for use in this application.
- Use the connection to get data from Oracle.

For this first phase of testing, we will accomplish all these steps in the `main()` method of `OracleJavaSecure`.

We are going to run through this test code at least twice. The first time, we will not have a valid two-factor authentication code, so we will exit the program at a certain point. In short, the point we will exit is just after we know a two-factor code has been sent to us, and before we try to use connection strings associated with this application.

Set the Application Context

If we have a two-factor authentication code, we should pass it to the application on the command line as an argument to the application. Alternatively, if you are running this test from an IDE, it may be easier to embed the two-factor code, once received, into this class. You can do that where shown, then recompile and execute it. See Listing 10-48. Remember that each two-factor code is only good for 10 minutes!

Listing 10-48. *Application Authentication Testing, Phase 1, main()*

```
public static void main( String[] args ) {
    OracleCallableStatement stmt = null;
    Statement mStmt = null;
    ResultSet rSet;

    try {
        // Submit two-factor auth code on command line, once received
        String twoFactorAuth = "";
        if( args.length != 0 && args[0] != null ) twoFactorAuth = args[0];
        // You may place two-factor auth code here for testing from IDE
        // Remember, it's only good for 10 minutes from creation
        //twoFactorAuth = "1234-5678-9012";
```

We call the `setAppContext()` method in Listing 10-49 to set our context for this application. In this case, our application ID is `HRVIEW`, the only application we have configured so far; the one that can get encrypted data from the `HR` schema. Recall that there is a user, `appusr` and a secure application role, `hrview role` associated with this application ID.

Listing 10-49. *Testing Call to setAppContext()*

```
String applicationID = "HRVIEW";
Object appClass = new InnerRevLvlClass();
setAppContext( applicationID, appClass, twoFactorAuth );
```

Multiple Java applications can gain access through that application ID. They will all get the same application user and role. Each Java application will be identified by a specific, representative application inner class that it will hand to Oracle database for validation. If the currently authenticated user (for whom the two-factor code was distributed) has been granted to proxy connect through the application verification, `appver` user, and the application class that he hands Oracle database is valid, then he will receive the connection strings associated with the application class. If the current user has also been granted proxy connect through the Oracle application user embedded in the connection string, then he will be free to use the connection to work with Oracle data.

There is another layer of control that you might consider applying, but which seems superfluous. You might control which users have access to which applications. In my mind, you have already granted the user access to data through whatever application interface they care to use by granting them proxy

connect through the application user associated with the secure application role. If they have access to the data, do we really want to be bothered with controlling how they get there? I don't think so.

If two applications see different data, then they should have different application IDs and different application users and roles.

Call to Get Application Connections

The first time we call `getAppConnections()` from a specific application there will be no list of connection strings stored in Oracle database for us to retrieve. That is almost an ancillary concern, because we will be turned back earlier in the process when a two-factor code is generated and sent to us.

So, even the second time we call `getAppConnections()`, we will get a `null` in return from Oracle, and we will set our list of connection strings to a new, empty `HashMap`. This will be the case until we have called `putAppConnections()` for this application. However, we can use the application in this state by pushing connection strings into the list for our own, local use as shown in Listing 10-50.

Listing 10-50. *Test Call to getAppConnections() and Put Connections in Local List*

```
getAppConnections();
// Go no further until we have a two-factor Auth Code
if( twoFactorAuth == null || twoFactorAuth.equals( "" ) ) return;
System.out.println( "connsHash.size = " + connsHash.size() );

putAppConnString( "Orcl", "hr",
    "password", "localhost", String.valueOf( 1521 ) );
putAppConnString( "Orcl", "appusr",
    "password", "localhost", String.valueOf( 1521 ) );
```

Send List of Connection Strings to Oracle Database for Storage

Storing our list of connection strings in Oracle database is a task we will only need to do periodically, as we change our application passwords. We will do it here in Listing 10-51 to save the initial set of connection strings (from Listing 10-50) with a call to `putAppConnections()`.

Listing 10-51. *Send Connection String List to Oracle*

```
putAppConnections();
```

Get a Unique Connection for Use in This Application

If we did not call `getAppConnections()` previously, it will be called automatically when we call `getAppAuthConn()` (see Listing 10-52). This would be our instruction to application developers who want to use our security structures—just call `getAppAuthConn()`. Note that by getting this application-specific connection, we will no longer be able to use the original `appver` connection to do `putAppConnections()`.

Listing 10-52. *Get and Use a Specific Oracle Connection for This Application*

```
getAppAuthConn( "orcl", "appusr" );
mStmt = conn.createStatement();
rSet = mStmt.executeQuery( "SELECT SYSDATE FROM DUAL" );
if ( rSet.next() )
```

```
    System.out.println( rSet.getString( 1 ) );
```

Notice that the key values we use to select a specific application connection are the instance name and the user name. These are two of the same values we provided when we called `putAppConnString()`. Once we get the connection, we can use it to query Oracle database.

Use or Lose Initial Application Verification Connection

Both `getAppConnections()` and `putAppConnections()` do their work using an `appver` user connection. This is an important point. In establishing the `appver` connection, we have exchanged encryption keys that are specific for that connection. We will retain sufficient vestiges of those keys in order to continue to decrypt the connection strings in `connsHash`; however, once we have connected to a different Oracle user for this specific application and exchanged keys for that connection, we will no longer be able to call `putAppConnections()` using the previous `appver` connection. In other words, we need to run `putAppConnections()` before using any of the connections returned by `getAppConnections()`.

We make another call to `putAppConnections()` in our test code, and it will fail because we already built and used an application connection. The connection to Oracle database as `appver` user is no longer available—we only retained the decryption keys associated with that session.

Get an Application Connection and the Associated Secure Application Role

After we get an application connection from our call to `getAppAuthConn()`, we want to get the secure application role associated with this application. Instead of calling `p check hrview access` to get that specific application role, we call our generic `p check role access` procedure, which grants us the secure application role that is associated with our application ID. In Listing 10-53, notice that we pass the application ID as parameter number 1.

Listing 10-53. *Get and Application Connection and Set the Application Role*

```
int errNo;
String errMsg;
getAppAuthConn( "orcl", "appusr" );
stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL appsec.p_check_role_access(?,?,?)" );
    stmt.registerOutParameter( 2, OracleTypes.NUMBER );
    stmt.registerOutParameter( 3, OracleTypes.VARCHAR );
    stmt.setString( 1, OracleJavaSecure.applicationID );
    stmt.setInt(    2, 0 );
    stmt.setNull(   3, OracleTypes.VARCHAR );
stmt.executeUpdate();
errNo = stmt.getInt( 2 );
errMsg = stmt.getString( 3 );
System.out.println( "DistribCd = " + errMsg );
if( errNo != 0 ) {
    System.out.println( "Oracle error 1) " + errNo + ", " + errMsg );
} else if( twoFactorAuth.equals( "" ) ) {
    System.out.println( "Call again with two-factor code parameter" );
} else {
    if( null != stmt ) stmt.close();
```

```
        System.out.println( "Oracle success 1)" );
```

In our call to p_check_role_access, we do not have to test whether the two-factor authentication code exists, though we do it here out of habit. You will recall that we are only doing two-factor authentication in the initial application verification connection, not in each specific application connection. We removed two-factor authentication from `p check role access` and added it to our application verification procedure to get the list of connection strings for an application, `p get app conns`.

Rehashing the security chain to this point; the user must be granted proxy connect through the application user account, and by extension, was granted access through the secure application role associated with this application connection.

Get Encrypted Data with the Application Connection

The full demonstration of our application authentication is provided when we get encrypted data from our application connection as shown in Listing 10-54.

Listing 10-54. *Test Getting Encrypted Data from Oracle*

```
...
String locModulus = OracleJavaSecure.getLocRSAPubMod();
String locExponent = OracleJavaSecure.getLocRSAPubExp();

stmt = ( OracleCallableStatement )conn.prepareCall(
    "CALL hr.hr sec pkg.p select employees sensitive(?,?,?,?,?,?,?,?,?)" );
...
OracleJavaSecure.closeConnection();
```

We have seen this procedure call before, multiple times. The only difference here is that we are using a connection that we got from our application verification list

Add More Application Connection Strings

A developer may have some call to use connections in her application that are not stored in the `connsHash` list that has been stored in Oracle Database. Those application strings can be added to the local application connections list by calling `putAppConnString()`. This call can even overwrite existing connection strings coming from the Oracle Database store (table), perhaps for testing against a new Oracle instance or a development or acceptance instance:

```
putAppConnString( "Orcl", "appusr",
    "password", "localhost", String.valueOf( 1521 ), true );
```

If a subsequent call to `putAppConnections()` is not made, then the modified connection strings list is not stored in Oracle, and they are only seen by and used by the local client application. The plan is to call `putAppConnections()` once, to store the strings in the Oracle database, and from then on, to remove the connection strings from the application and use only what is stored in the database. We will avoid putting our connection strings into the application at all, and will make that process easy with the administrative interface we build in Chapter 12.

Testing a Second Application

Reflect for a moment on what we have done so far in the previous tests. We have an inner class in the `OracleJavaSecure` class that implements `RevLvlClassIntfc`, and we pass that to Oracle database for application validation. How does the Oracle JVM handle that inner class? Is it on the `CLASSPATH` or otherwise known to the Oracle JVM?

In this specific case, we actually loaded the class to Oracle Database when we loaded the outer class, `OracleJavaSecure`. If you have a SQL client application that lets you browse structures in Oracle, you can see the *orajavsec.OracleJavaSecure.InnerRevLvlClass* listed in the database. You can also execute this query to see it:

```
SELECT * FROM SYS.ALL OBJECTS
WHERE OBJECT TYPE = 'JAVA CLASS'
AND OWNER = 'APPSEC';
```

Unfortunately, occasionally the object name will have been modified by Oracle, and you may see something like "/fddfb98e_OracleJavaSecureInne". In addition to the expected inner class, for `OracleJavaSecure` you may also see an inner class named *OracleJavaSecure$1*—a class that represents our use of reflection to generate a heretofore unknown class.

This is probably the only application inner class for verification whose outer class is loaded into Oracle, so we have to ask, "how will Oracle database instantiate classes from other applications?"

Objects We Have Never Seen

I have repeatedly said that the application will hand us its inner class, which we will validate in order to authorize an application. I have made it sound like the Oracle JVM is able to create the class and object out of thin air using only the class bytes that we are providing from the application, or those stored in Oracle Database for this application. However, that is not really the case.

I am not saying that it is not possible to generate classes and objects based just on a byte array—it certainly should be possible by defining a `BytesClassLoader` class, based on a `URLClassLoader` that takes a byte array rather than a URL directory or jar file. However we will be taking the easy way out.

There are two reasons for taking the easy way out: to avoid the security issues raised by an alternate `ClassLoader`, and because an easy solution exists.

Our approach bears some resemblance to remote method invocation (RMI). In RMI, a local stub class exists that represents the remote object. In RMI, the local stub class and the `RMIClassLoader` handle calls to the actual methods that are running remotely, on the RMI server. That is not what we will be doing.

I say that our approach bears resemblance to RMI because we will have a representation of the application inner class pre-loaded in the Oracle JVM. The class in the Oracle JVM needs to bear strong resemblance to the class that will be passed to us from the client application, but doesn't need to match exactly. In particular, we are going to instantiate client classes with, potentially, different revision labels.

Place Stub Class on Oracle

Here is where and why we can enforce at least a partial code review for every application that intends to use our security structures. This will be a good thing, since you are the expert now, having reviewed all the secure programming concepts discussed in Chapters 2 and 3.

We need to distill the inner class that will be used to validate each application down to its essential structure. This includes the outer, containing class structure. We are going to load just those essentials

into Oracle database to serve as a stub on which to realize classes and objects of this type in Oracle database.

Take a moment to compare the original inner class, *in situ* with the stub for our second application, testojs/TestOracleJavaSecure.

■ **Note** Compare the files named *Chapter10/ testojs/TestOracleJavaSecure.java* (the original) and *TestOracleJavaSecure.sql* (the stub).

Looking at certain aspects of our stub, I'd like to point out what's required. First in Listing 10-55, notice the package name, *testojs*. We are going to require that application inner classes exist in a package, in order to assure that applications may even name their inner classes the same, but have unique names based on the package prefix. Second notice the outer, containing class definition for `TestOracleJavaSecure`. This outer class definition needs to match exactly what we see in the original code. The inner class is named `AnyNameWeWant`.

Listing 10-55. *Stub Class for Second Test Application*

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."testojs/TestOracleJavaSecure" AS
package testojs;
import java.io.Serializable;
import orajavsec.RevLvlClassIntfc;

public class TestOracleJavaSecure {

    public static class AnyNameWeWant
        implements Serializable, RevLvlClassIntfc
    {
        private static final long serialVersionUID = 2011013100L;
        private String innerClassRevLvl = "20110131a";
        public String getRevLvl() {
            return innerClassRevLvl;
        }
    }
}
```

The definition of the inner class itself should be exactly the same as in the original code, with the exception of the `innerClassRevLvl` string (whether you've defined it in the class or in the `getRevLvl()` method). Note that in all cases, the `public` and `private` modifiers should be maintained. We want the inner class to be declared as `static` so that we are dealing with a single object rather than potentially multiple instances.

In addition to being in a package, the inner class needs to be declared `public` so that the `OracleJavaSecure` class can generate the object, even though it's in a different package. The containing, outer class should also be public for this reason.

There is a lot of flexibility in placement of the inner class within the containing class. First of all, the containing class need not be a top level class for the application—it can be an ancillary class. In this way, a diligent programmer may withhold sensitive code from perusal by the application security personnel who will be generating the stub.

There is another reason why we may want our application inner class in an ancillary outer class. There is some concern because inner classes have access to the private members of their outer, containing classes. So having the application inner class in an ancillary outer class would be potentially less sensitive than having the inner class in the core application class.

The inner class can also be placed inside a method, instead of in the main class body. Generally, inner classes that are defined within methods are given a "$1" style marker in the middle of their class name. You can see that when examining the compiled class names.

Get Application Authentication Connection and Role

We should always be on the lookout to make using security easier for developers. With that in mind, we are going to combine into a single step, what had previously been a couple steps that asked our developers to implement separately. Up to this point we have asked our developers to, at a minimum, set their application context, then get an application connection and finally call Oracle Database to set their secure application role. We are still going to have the developers set their application context, but we are going to combine the request for an application connection with the request to get the secure application role. We will do this with a new method, `getAAConnRole()` shown in Listing 10-56.

When we call `getAAConnRole()`, we will have an `OracleConnection` returned to us, and the connection will already have the secure application role set. This new method takes the same arguments that we were providing to the `getAppAuthConn()` method.

Listing 10-56. *Get Application Authentication Connection and Role, getAAConnRole()*

```
public static OracleConnection getAAConnRole( String instance, String userName ) {
    OracleConnection mConn = null;
    OracleCallableStatement stmt = null;
    try {
        mConn = getAppAuthConn( instance, userName );
        // If mConn is null, probably did not send twoFactorAuth
        if( null == mConn ) return mConn;
        int errNo;
        String errMsg;
        stmt = ( OracleCallableStatement )mConn.prepareCall(
            "CALL appsec.p_check_role_access(?,?,?)" );
            stmt.registerOutParameter( 2, OracleTypes.NUMBER );
            stmt.registerOutParameter( 3, OracleTypes.VARCHAR );
            stmt.setString( 1, applicationID );
            stmt.setInt(    2, 0 );
            stmt.setNull(   3, OracleTypes.VARCHAR );
        stmt.executeUpdate();
        errNo = stmt.getInt( 2 );
        errMsg = stmt.getString( 3 );
        //System.out.println( "DistribCd = " + errMsg );
        if( errNo != 0 ) {
            System.out.println( "Oracle error 1) " + errNo + ", " + errMsg );
        } else if( twoFactorAuth.equals( "" ) ) {
            System.out.println( "Call again with two-factor code parameter" );
        }
    } catch ( Exception x ) {
        x.printStackTrace();
    } finally {
```