Also, we will almost always be expecting one or more rows to be returned in a `ResultSet`, and calling `if( rs.next() )` would only ever test whether a single row is returned. When we expect to be getting several rows, we will use a `while` block to process the `ResultSet`, for example:

```
while( rs.next() ) {
…
}
```

Method Syntax in Java Stored Procedures

I described the function of the modifier `static` in the `main()` method in Chapter 3. In Java stored procedures, we will see that Oracle database only calls `static` methods from Java stored procedures. This is for a similar reason to what we saw for the `main()` method: Oracle is calling these methods without instantiating an object of that class beforehand. We do not create an instance of `MyApp4` (listing 4-1) before we call the `getOracleTime()` method. However, just as the `main()` method can instantiate itself or any other object, these static methods (like `getOracleTime()`) called as Java stored procedures can instantiate themselves or other objects as needed (not everything used in a static method is static).

Calling Java from Oracle Database

The second part of our example code for a Java stored procedure (function) in Listing 4-1 is an Oracle statement to create a function, `f_get_oracle_time`. The syntax (repeated in the following) should be familiar—most of the Oracle keywords are identical to what we saw in Chapter 2 when we created procedures. With Java stored procedures (and functions), we modify the procedure definition with an indication that the code performing the processing is in a different language—in our case, Java. We indicate what Java code is going to be called by providing the full Java method specification:

```
CREATE OR REPLACE FUNCTION f_get_oracle_time
    RETURN VARCHAR2
    AS LANGUAGE JAVA
    NAME 'pkg4.MyApp4.getOracleTime() return java.lang.String';
/
```

Notice that our specification of the Java method includes the package name (`pkg4`), the class name (`MyApp4`) and the method name `getOracleTime()`. In this example, we also see that the return type being sent back from Java is specified as `java.lang.String`. If you look at the definition of the return type from the Oracle function, it says:

```
RETURN VARCHAR2
```

The Oracle `VARCHAR2` type corresponds to the Java `String` type. In case you hadn't already guessed, *java.lang* is the package where Sun (Oracle) keeps the `String` class. Specifying a fully qualified package as part of the parameter and return value types is required in Java stored procedure definitions.

We do need to coordinate the consistency of Java return types with the return types available in Oracle database. Oracle includes Java classes that encapsulate their return types in the *oracle.sql* package, and for many Oracle types, there are corresponding generic Java classes in the *java.sql* package. We will see some of these as we proceed. For more information on the Oracle database types and associated Java types, refer to the Oracle Documentation and to Appendix A of the document called *Oracle® Database JDBC Developer's Guide*.

Although this technology is called Java stored procedures, you'll see that for the most part we will be calling Java methods from Oracle functions. Java methods can only return one value. While that value

can be a complex array of objects or an object with many members, it is still a single return value—a model that is analogous to Oracle functions. Oracle defines a stored procedure as code that performs a process, and a stored function as code that returns a value. Of course, an Oracle procedure can return multiple values through OUT parameters, and it would be handy if we could map that to a Java stored procedure that returns multiple values, but we cannot easily do that. And of course, an Oracle function can perform a process before returning a value. Oracle functions can also return multiple values through OUT parameters, but that is discouraged.

VARCHAR OR VARCHAR2

When creating database tables and writing PL/SQL, it is customary to define string columns and variables as type VARCHAR2. For historical reasons, there is no VARCHAR1. Oracle database does currently recognize VARCHAR as a synonym (a subtype in PL/SQL) for VARCHAR2, but Oracle warns against the use of VARCHAR in their official documentation. You can read that warning at the following link:

```
http://download.oracle.com/docs/cd/E11882_01/server.112/e17118/sql_elements001.htm#sthre
f117
```

There is no reason to go against Oracle's advice, and I recommend VARCHAR2 when creating tables and writing PL/SQL. There is an exception to using VARCHAR2 as the string type, and that exception occurs when writing Java code to interface with the database. Later on when we define the parameters that we intend to pass from Java to Oracle in an `Insert` or `Update` statement, we will call them `OracleTypes.VARCHAR`. There is no `OracleTypes.VARCHAR2` defined. `OracleTypes` is a class provided by Oracle, so `VARCHAR2` as a type is conspicuous in its absence.

The important thing to remember here, is that one uses VARCHAR2 when creating a database column, or when defining a PL/SQL variable, and one uses `OracleTypes.VARCHAR` when defining a Java variable or parameter to be compatible with VARCHAR2 values from Oracle database.

Installing and Testing the Example Code

Run the code from Listing 4-1. You can run it using any of *SQL*Plus, SQL Developer, JDeveloper, TOAD* or any other database code editor that enables execution of code. The first portion is one line to set your role to `appsec_role`. The second portion of Listing 4-1 creates the Java Source named `MyApp4`. And the third portion creates the `f_get_oracle_time` function.

After running the code, you can test the Java stored procedure by calling the Oracle function `f_get_oracle_time`. The function will return the current date and time from the Oracle database.

```
SELECT f_get_oracle_time FROM DUAL;
```

A note here about the `FROM DUAL` clause. In order to select data, the `SELECT` query must have a source. The source will typically be an Oracle table or view. In this case, however, we are selecting the return value from a function. By saying `FROM DUAL` we are fulfilling the requirement for a source in the `SELECT` query syntax, but essentially saying "from the database." `DUAL` is actually a dummy table with one column named `DUMMY` and one row with `DUMMY = 'X'`. You can see this from the following query:

```
SELECT * FROM DUAL;
```

If you have by now surmised that we could have selected SYSDATE from any table or view, you would be correct! However, we would get a result for each and every row in the view or table we specified. Try this query and you will see results like those shown:

```
SELECT USERNAME, SYSDATE FROM SYS.ALL_USERS ORDER BY USERNAME;

ANONYMOUS            09-JUN-11
APEX_030200          09-JUN-11
APEX_PUBLIC_USER     09-JUN-11
APPQOSSYS            09-JUN-11
APPSEC               09-JUN-11
APPUSR               09-JUN-11
APPVER               09-JUN-11
```

A row for each user is returned, but the only values we return are username and SYSDATE for each row. This is a feature of the SELECT statement that we can use to our advantage. For example, if we select all the managers from a list of personnel, we might insert the words "Pointy-Haired" between the first and last names, like this:

```
SELECT FIRST_NAME || ' Pointy-Haired ' || LAST_NAME
    FROM HR.EMPLOYEES
    WHERE EMPLOYEE_ID IN (
        SELECT DISTINCT MANAGER_ID FROM EMPLOYEES
);
```

The double pipes "||" is a way to *concatenate* strings in Oracle database. We have a sub-query in the parentheses that selects all the *distinct* (unique) EMPLOYEE_IDs that are listed in the MANAGER_ID column of EMPLOYEES. For all employees that have one of those MANAGER_IDs as their EMPLOYEE_ID, we concatenate their FIRST_NAME, the words "Pointy-Haired" and their LAST_NAME, just for fun mind you.

We can select whatever we want for each row that is returned from a query. We can SELECT SYSDATE for each row, or we can SELECT "First Pointy-Haired Boss" for each row.

Review The Roster of Participants

The following is a quick rundown of the participants in our test of the example code:

1. Your client (Oracle command line) calling.
2. The function on the Oracle database that encapsulates.
3. The Java source named MyApp4 (Listing 4-1) that queries.
4. The Oracle database to select SYSDATE.

When we created the MyApp4 Java source, the Oracle database compiled the code and created the package *pkg4* and the class MyApp4. Our Oracle function calls the static method getOracleTime() in the MyApp4 class. In the getOracleTime() method, we open a connection to the Oracle database and request the system time, SYSDATE which we then return as a String. Our function receives the time string and returns it to our client as a VARCHAR2, and the client displays the date and time string.

You can see how these participants relate to one another in Figure 4-1.

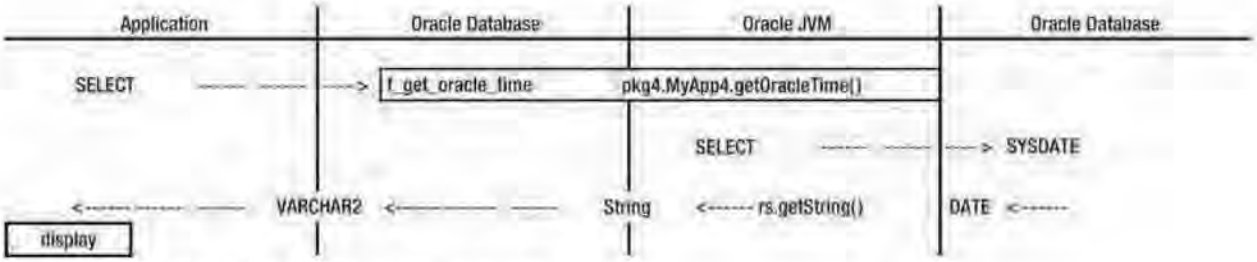


Figure 4-1. Flow of a Java stored procedure

There are two things that stand out in Figure 4-1 that I want to emphasize. First is that the Java stored procedure, function `f_get_oracle_time`, and the associated Java method, `getOracleTime()` are so closely related that I show them in a single box. You might say they are "joined at the hip." All that the function `f_get_oracle_time` does is pass its parameters to the Java method and return the value that is returned from the method.

The second thing I'd like to emphasize is that as the return value is passed along its return path, it is consecutively seen as a `DATE` type, then a `String` type, then a `VARCHAR2` type.

Cleaning Up

What we have just seen is merely an example of how Java stored procedures operate. You wouldn't normally invoke a Java stored procedure just to get the time from a `SELECT` statement. It would be much simpler to just invoke the built-in `SYSDATE` function. For example:

```
SELECT SYSDATE FROM DUAL;
```

The real usefulness of Java stored procedures is to do something hard or impossible to do in PL/SQL.

So let's remove this example from the database with these commands:

```
DROP FUNCTION f_get_oracle_time;
DROP JAVA SOURCE myapp4;
```

The Oracle Java Virtual Machine

When Oracle corporation first included a JVM in its database, it called it Aurora. You will still find some references to Aurora in the directories and files included with Oracle Database 11g.

The Oracle JVM in Oracle Database 11g is heavily customized, and cannot be upgraded or replaced with a standard JVM downloaded from Sun (Oracle corporation). The Oracle JVM has a comprehensive security structure and sandbox that is tied in with the Oracle database. That sandbox is part of what makes the Oracle JVM impossible to replace with a standard JVM. Sandbox security settings can be queried from the database and set with Oracle commands (we will see that in Chapter 9 when we discuss two-factor authentication).

Oracle JVM Based on Java SE 1.5

Probably because of the amount of tailoring that Oracle corporation needed to do to build the Oracle JVM, it is a release behind the current standard JVM. The Oracle JVM is based on Java Standard Edition (SE) version 1.5, while 1.6 is currently supported. 1.7 was released in July 2011. Talking about versions can really make a text like this look dated, and this paragraph may fall behind quickly; but in a few years

when you have Java 1.7 or 1.8 on your workstation and you are using Oracle Database 12, you may still be using an Oracle JVM based on Java 1.5 – the Oracle JVM gets that far behind.

In any case, Java 1.5 is what we needed in Oracle database to support our use of JCE. So we won't criticize or lament.

A Separate JVM for Each Oracle Session

When you connect to Oracle database and use Java, you start a separate JVM for use only by your session. You cannot share any Java objects or data with other running JVMs (Oracle sessions), unless you store them in the database for others to query or hand the objects around through network port services (e-mail, ftp, web, etc.)

When you disconnect from Oracle database, your Oracle session closes and the JVM is terminated. If you immediately make a new connection to Oracle, you get a brand new JVM and nothing in your previous JVM is retained or carried over to the new JVM.

This has repercussions for us. Primarily, since we are going to be generating cryptographic keys (you might have already guessed that), we want to use them for a time to send encrypted data back and forth before we close the connection to Oracle database and the keys are discarded. We could save the keys in the database, or in what Oracle calls a wallet, which is basically a secure file on the hard drive of the server and/or the client. But if they are stored anywhere, then we don't know who might get access to them. No, what we will do is generate the keys on the fly and use them for a time, keeping our connection open during the key exchange and multiple queries, and only close the connection when we are done.

Oracle JVM Sandbox

We briefly discussed the Oracle JVM sandbox at the end of the last chapter. To recap that discussion, this particular sandbox prevents the Java code in Oracle database from reading from/writing to the hard drive; and from communicating on the network at all; however, it provides open access to the Oracle database, limited by the Oracle privileges granted to the Oracle user who is executing the Java code, and his roles. For example, if the `HR` user owns and is executing the Java stored procedure, it will have access to all of the `HR` schema data.

Extended security privileges can be granted to Java code in Oracle database to perform functions normally denied by the sandbox. These privileges are granted by administrative commands and are stored in the database. Privilege grants can be broad or very granular. We will be using extended privileges for Java code running in Oracle database in order to communicate to other machines over the internet. You will see some very granular grants for those privileges when we get to Chapter 9.

Auto-Commit Disabled in the Oracle JVM

In SQL databases, `COMMIT` means to incorporate changes into the existing database. Until `COMMIT` is executed, data updates are held in a cache. The session that made the updates can see the changes, but everyone else sees the data as it existed before the updates. When the session executes `COMMIT`, everyone can see the changes that were made. This is a powerful concept for transactions that have multiple steps. In a single session, each step can be performed, and each successive step can base its actions on updates already made in the session. When all the steps are completed, the session can issue a `COMMIT`, and all the updates will be made to the existing database. However, if there is a problem in any later step, all the completed steps can be *rolled back* (reversed) and not written to the existing database. You can see how this might be handy if a bank is transferring money from you to me—they will take it from your account and put it in my account. But if they have problems giving me your money, you want them to also reverse the step where they took it from you.

Generally, in Oracle database, a `COMMIT` must be issued as needed, which is sometime after every data update (`INSERT`, `UPDATE` or `DELETE`). Oracle *procedures* that update data should not issue a `COMMIT` – they assume that they are just part of a transaction and that whatever code called them will issue a `COMMIT` when all the parts are successful. Tom Kyte, author of *Expert Oracle Database Architecture: Oracle Database 9i, 10g, and 11g Programming Techniques and Solutions* (Apress, 2010) explains on his blog (`asktom.oracle.com`) that procedures do *atomic* units of work, and all transactions must be controlled (and `COMMIT` executed) by the invoker (the one who called the procedure).

Auto-commit is a mode of operation that you can employ where every update is immediately committed to the database. You can turn this feature on by executing:

```
SET AUTOCOMIT ON;
```

You might think that this would defeat the ability to do multiple-step transactional updates, and you would be right. However, this is the default mode for JDBC. So from a Java client, when you update the database, that update is immediately committed, and you cannot roll it back. One benefit is that you don't have to execute another Oracle statement using JDBC to `COMMIT` the update.

Actually you can turn off the auto-commit mode in JDBC by calling the `setAutoCommit()` method of your `Connection`, like this:

```
conn.setAutoCommit(false);
```

Then when appropriate, you can commit or rollback your transaction manually with one of these commands:

```
conn.commit();
conn.rollback();
```

Auto-commit is one area where pure Java is not "run anywhere." For Java code running in the Oracle database (Java stored procedures), the default mode is for auto-commit to be turned off. This adheres to the proposition that procedures are atomic and that transactions are committed by the invoker, but it is the opposite of what you'd expect from JDBC.

This is something to keep in mind as we develop Java stored procedures. If we update data, we need to issue a `COMMIT`.

Chapter Review

This has been a rather brief chapter in which we've been able to focus on running Java in the Oracle database, or Java stored procedures. We saw a couple techniques of loading Java into Oracle.

It may be off that subject, but we also discussed in some detail the reverse process of calling Oracle database from Java using the `Connection` and `Statement` classes. We also diverged into a discussion of `FROM DUAL` and the `SELECT` query.

As we must do as we proceed, we stepped back to talk about general programming issues. In this chapter, we covered some additional aspects of exception handling, and we covered declaring and initializing members.

Most important, we discussed the Oracle JVM and its transient nature and association with a specific Oracle session. Also, we talked about the Oracle JVM sandbox.

CHAPTER 5

Public Key Encryption

I'm sure you've heard of public key encryption (PKE). You probably use it every day—if not in your code or on your servers, then on the Internet. When you use secure socket layers (SSL) in your browser or when you see the lock or key symbol and see `https:\\` (with the s) as the protocol in the address bar, you are using PKE and other encryption. SSL is used to send private data such as your credit card number in encrypted form to online merchants.

The basic concept of PKE is that two entities share a set of two keys (one public and one private) and use them to encrypt data for and decrypt data from each other. The keys are tied to one another so that anything encrypted with one of the pair (e.g., the private key) can only be decrypted with the other (e.g., the public key). This encryption/decryption works in both directions.

The most important aspect of PKE is that one of the keys is private. The private key is never disclosed to anyone; however, the public key can be given to any and all requestors. How does this help privacy?

The one requesting the public key can be sure that if that key successfully decrypts data, then it came from the entity with the private key. Only the private key can encrypt data that can be decrypted with the public key.

The converse is also true: only the public key can encrypt data that can be decrypted by the private key, so the one using the public key can be sure that only the entity with the private key can decrypt data that they encrypted.

Because the public key is, well, public, anyone can decrypt data sent from the originator. Also anyone can encrypt data and send it to the originator for decryption. This is not a flaw, but it is an aspect of PKE that we need to be aware of and account for.

Generate Keys on the Client

We will have the client computer generate a set of keys. That computer will send artifacts (components) of the public key to the Oracle database so that Oracle can build a copy of the public key. Then the Oracle database can encrypt data using the public key that only the originating client can decrypt.

This approach may sound like a complete solution, but there are a couple concerns that we will not address until we get to the next chapter's discussion on secret password encryption. First, anyone can read the public key artifacts as they traverse the network (that is, anyone with software to read all packets going across the network, like a sniffer.) That means that we have to assume that everyone has the public key and that everyone sees and can decrypt data (if any) that the client encrypts with the private key and sends to the server.

The second concern is that PKE, at least the version we are using, does not lend itself to encrypting large amounts data. It is a block cipher with a limited block size. For example, if the block size for our PKE keys is limited to 128 bytes, we would have to break the data into portions of that size and encrypt each portion individually. On the other side of this transaction, the recipient would have to decrypt each portion and reassemble the original data.

To handle larger amounts of data, there are a couple of methods: cipher block chaining (CBC) and stream encryption. With CBC, large data is broken up into appropriate block sizes for encryption and then decrypted and reassembled automatically for the user. (Whew, that takes a lot of burden off our shoulders.) With stream encryption, each bit, byte, or block of bytes would be encrypted/decrypted as it passed through the stream. A stream is simply a channel for data en route. You put bytes of data into a stream, and take bytes of data out in the same order: first in first out (fifo). A stream can exist when reading/writing data to storage, or across the network, or simply from one place (structure) in memory to another.

RSA Public Key Cryptography

We will be using RSA public key cryptography for our PKE encryption algorithm. RSA stands for the last names of the creators of the algorithm: Rivest, Shamir, and Adleman.

Because RSA uses a different key for encryption (e.g., private) from what is required for decryption (e.g., public), it is called an asymmetric algorithm. All PKE is asymmetric encryption. With a long key length, RSA is a very trustworthy encryption algorithm.

Java Code to Generate and Use RSA Keys

All our code for accomplishing Oracle database and Java security will reside in a single Java class (there are some small exceptions; we will have some separate Java classes for testing our processes). As we walk through the remaining chapters of this book, we are going to develop security code in phases, adding layers and concepts as we progress. Our single Java class will grow over time.

Our class will be called `OracleJavaSecure`, and we will define it in a package called `orajavsec`. Because we do not have a single version of this file, we are going to have multiple directories (one per chapter) where different versions of this Java code reside. This will make compiling and running a bit more difficult, but I will provide instructions as needed to reference these files.

■ **Note** You can find the following code in the file *Chapter5/orajavsec/OracleJavaSecure.java*. I recommend that you open that file and refer to it as we proceed through this chapter.

Creating a Set of Keys

Listing 5-1 shows the code that is used for creating a set of PKE keys. This code, along with other Java code in this chapter, comes from the `OracleJavaSecure` class.

Listing 5-1. *Create PKE Keys,* `makeLocRSAKeys()`

```
private static SecureRandom random;
private static int keyLengthRSA = 1024;
private static Key locRSAPrivKey;
private static RSAPublicKey locRSAPubKey;
private static void makeLocRSAKeys() throws Exception {
    random = new SecureRandom();
    KeyPairGenerator generator = KeyPairGenerator.getInstance( "RSA" );
```

```
        generator.initialize( keyLengthRSA, random );
        KeyPair pair = generator.generateKeyPair();
        locRSAPrivKey = pair.getPrivate();
        locRSAPubKey = ( RSAPublicKey )pair.getPublic();
    }
```

We use the JCE class named `KeyPairGenerator` to generate our private and public keys. First, we instantiate a `KeyPairGenerator` named `generator`. We specify that `generator` will create keys conforming to the RSA algorithm. (Notice we call the static method `KeyPairGenerator.getInstance()` and ask it to get an instance of itself.) We initialize `generator` with both the key length (1024 bits) and with an instance of the `SecureRandom` class named `random`. `SecureRandom` is a trustworthy random number generator. We will use `random` to even greater purposes in the next chapter.

■ **Note** Someday, a 1024 bit RSA key may be insufficient, but at this time, it is still considered relatively unbreakable.

`KeyPairGenerator` generates a single object: a `KeyPair` that we call `pair`. We get our separate public and private keys (`locRSAPubKey` and `locRSAPrivKey`) by calling methods in `KeyPair`. Notice also that we cast the public key that we get from `KeyPair` as an `RSAPublicKey` type in this statement:

```
locRSAPubKey = ( RSAPublicKey )pair.getPublic();
```

Casting can add functionality to an object as long as the object you cast to implements (is a superset of) the object in hand. In this case `RSAPublicKey` is a superset of the `PublicKey` class. When `RSAPublicKey` was defined, the definition looked like this:

```
public class RSAPublicKey implements PublicKey …
```

We cast the `PublicKey` to an `RSAPublicKey` so that we can use some methods that only exist for the `RSAPublicKey` class. We will see those in the next section.

When we run Listing 5-1, we end up with the keys. The `locRSAPrivKey` is our private key. We will not share our private key with anyone. We will do our encryption and decryption using the private key—so an entity with our public key can then decrypt data from us and encrypt data to us. The `locRSAPubKey` is the public key of our key pair. We intend to give that key to all requestors.

Hand the Public Key Across the Network

We intend to give `locRSAPubKey` to all requestors (actually we just hand it directly to the Oracle database); however, we don't know (for arguments' sake) which platform they will be running on, so we can't just give them our Java cryptographic extensions (JCE) version of the key. Instead, we will give them two artifacts, which together can be used to build our public key on whatever system.

We can only build another key of the same type with our two artifacts. Our artifacts are `locRSAPubMod` and `locRSAPubExp`. These are the RSA public key modulus and exponent, two numbers that we can use to calculate the public key. Listing 5-2 shows how we generate those artifacts.

Listing 5-2. *Generating Public Key Artifacts*

```
    private static BigInteger locRSAPubMod = null;
```

```
private static BigInteger locRSAPubExp;
…
locRSAPubMod = locRSAPubKey.getModulus();
locRSAPubExp = locRSAPubKey.getPublicExponent();
```

The methods `getModulus()` and `getPublicExponent()` are available in the `RSAPublicKey` class, but not in the `PublicKey` class. We cast the `PublicKey` as an `RSAPublicKey` in order to use these methods (see previous section). In essence, we ask the RSA public key to give us the value of its modulus and exponent.

These numeric values are too large to handle without wrapping them in Java objects. We get them as `BigInteger` objects. They can be very large numbers (especially the modulus) that don't fit well into standard primitive types like `integers` or `longs`, and for that reason, we often handle and transmit these numbers as if they were `Strings`.

You might have noticed that we declared `locRSAPubMod` to be `null`. You will recall our reasons for this from the section called "Declaring and Initializing Members" in Chapter 4. In this case, we plan to test whether `locRSAPubMod` is `null` to see if we need to generate the keys or if they already exist.

Serialize Objects

We are not limited to transmitting only simple type values to and from Oracle database. In Oracle, we can define new types that hold arrays or data structures, and we can transmit objects in `RAW` format (an ordered sequence of bytes). If we want to transmit very large objects, we can transmit and store them as binary large objects (`BLOB`s) in Oracle.

There is one requirement for transmitting a Java object across the network, and for storing a Java object, for that matter; that is, the Java object must implement the `Serializable` interface. Normally, an interface is a set of members and methods that must be implemented (set in code) by any class that implements the interface. (We will create our own interface in Chapter 10.) However, with the `Serializable` interface, there are no members or methods to be implemented; rather, implementing the interface just indicates that you intend to package an instance of your class for transmission or storage. You can send an object that is `Serializable` across an `ObjectStream` or write it to a `File` with an `ObjectWriter` class.

Some objects cannot be serialized (cannot implement `Serializable`). If something about the state of an object exists by association rather than as a value, then the object cannot be serialized. For example, if our object has an Oracle `Connection` member, then the connection cannot be serialized (you can't save a connection for use later), and our object cannot be serialized.

Building the Public Key from Artifacts

On the other side of the encryption/decryption process, we want to get the necessary artifacts (components) of the public key, and we want to rebuild the key. We are going to look at how we transmit the components from the client to the server in just a minute, but for this discussion and for testing, we can simply get them locally and build the key (we will use the same code on the server side). Listing 5-3 shows the code to build the public key from components of the key.

Listing 5-3. *Building Public Key from Artifacts,* `makeExtRSAPubKey()`

```
private static RSAPublicKey extRSAPubKey = null;
private static void makeExtRSAPubKey( BigInteger extRSAPubModulus,
    BigInteger extRSAPubExponent ) throws Exception
{
    RSAPublicKeySpec keySpec =
        new RSAPublicKeySpec( extRSAPubModulus, extRSAPubExponent );
    KeyFactory kFactory = KeyFactory.getInstance( "RSA" );
    extRSAPubKey = ( RSAPublicKey )kFactory.generatePublic( keySpec );
}
```

We hand both the RSA Modulus and the RSA Exponent to this method, `makeExtRSAPubKey()`. Then we instantiate a new instance of the `RSAPublicKeySpec` named `keySpec`, which is a specification object for a key. We also get an instance of a `KeyFactory` named `kFactory`, based on the RSA algorithm. We pass the `keySpec` to the `kFactory` to generate the public RSA key. This key is equivalent to the public key that was originally generated, so we can use it to decode data that is encoded with the original private key.

What does all this mean? How will we use this so-called equivalent key? We generate the key pair on the client. Then we pass the public key modulus and exponent to Oracle database. We assemble an identical public key on the Oracle database. At that point, our plan is to encrypt data with the public key on Oracle and transmit the encrypted data to the client. The client alone will be able to decrypt the data using the private key.

Generating the RSA Cipher

A cryptographic key is really nothing more than a mathematical construct; it does not inherently have the functionality we need to put it to use. Instead, we need to have a `Cipher` class that can use the key to do our work. We generate the `Cipher`, `cipherRSA` with the code shown in Listing 5-4.

Listing 5-4. *Generate RSA Cipher,* `makeCryptUtilities()`

```
private static Cipher cipherRSA;
private static void makeCryptUtilities() throws Exception {
    cipherRSA = Cipher.getInstance( "RSA" );
}
```

Our `makeCryptUtilities()` method will be called from the client when we call `makeLocRSAKeys()`, and from the Oracle database when we call `makeExtRSAPubKey()`.

The `Cipher` class can generate a cipher of various types. We are specifically creating a new instance of `Cipher` that uses the RSA algorithm.

Note that we don't say `new Cipher()`, but we call the static `Cipher.getInstance()` method to get a specific instance of itself. If you review the code we've seen so far in this chapter, you will see that quite frequently we are not instantiating an object in our code with the `new` statement; rather, we are getting new instances from some provider method, some factory or some other provider. It is important to understand that somewhere in that process, some method or other got a `new` instance to deliver to us.

Using the RSA Cipher

We will use `cipherRSA` to do encryption and decryption. To use it, we will have to provide it with the key and tell it what mode to use (encryption or decryption), then we will ask it to perform that task on some data. The `cipherRSA` method calls in Listing 5-5 will be used for that process.

Listing 5-5. Initialize the Cipher to Do Encryption

```
cipherRSA.init( Cipher.ENCRYPT_MODE, extRSAPubKey, random );
byte[] encodedBytes = cipherRSA.doFinal( clearBytes );
```

The first method called here is used for initializing the `Cipher` to a specific mode: encrypt or decrypt. In this case, we have asked it to be ready to encrypt data (`Cipher.ENCRYPT_MODE`). To decrypt, we would initialize the `Cipher` to `Cipher.DECRYPT_MODE`.

The second method of `cipherRSA` that we call is `doFinal()`. This method does the encryption or decryption that we initialized on the data we provide in the parameter (e.g., encrypt `clearBytes`). Note that the `doFinal()` method of `Cipher` takes a byte array as its parameter and returns a byte array. For that reason, we will be converting `Strings` and other objects to and from byte arrays as we perform encryption. Note also that a byte is simply eight bits and a bit is just one memory position being turned Off or On (value of 0 or 1). A byte array is simply a series of bytes assembled in and handled as a single list. If some of these concepts are unfamiliar, you can explore computer architecture at `www.wikipedia.org`.

JAVA CONSTANTS

Notice how we specify `ENCRYPT_MODE` in the `Cipher` initialization call in Listing 5-5. This is a member variable (notice it is not a method call—no parentheses or parameters.) Notice also that it must be static; we do not refer to a specific instance of `Cipher` to get `ENCRYPT_MODE`, but to the static class definition. Lastly, notice that the member name is in all uppercase. The uppercasing of this member variable and underscore characters between words indicates, by Java naming conventions, that this is a constant. In `Cipher`, it would have been declared like this:

```
public static final int ENCRYPT_MODE = <some int>;
```

The modifier `final` helps identify this member as a constant, but `final` may not mean what you might think at first. It does not mean that the value cannot change, but that the address pointed to in memory cannot be changed to a new address. For primitive and simple types, like the `int ENCRYPT_MODE`, this amounts to the same thing. That includes `Strings`, because they are immutable (refer to Chapter 3).

However, if you had a constant `Date` member, defined like the following

```
public static final Date BERLIN_WALL_FELL = new Date( 89, 10, 9 );
```

you could not prevent someone from coming along later and setting:

```
BERLIN_WALL_FELL.setYear( 80 );
```

Even though this `Date` is declared `static final`. Wouldn't it be nice to improve history!

This method for defining a `Date` with three integers representing year, month, and day is *deprecated*, which means it is still available but no longer approved for use in code. Potentially, deprecated features will be removed in future versions of the JDK. I use it here only for illustration purposes. Perhaps the reason that this `Date` constructor is deprecated is the inconsistency in specifying the integers. This constructor adds 1900 to the year specification and adds 1 to the month. The month specification is 0-based (0-11), but the day is 1-based (1-31).

Note that the `BERLIN_WALL_FELL Date` member still points at the same instance of `Date` (we did not instantiate a new `Date`), but we changed the value. In fact, that member variable meets all the requirements for a Java constant, but it turned out to be changeable (rather than constant)! So, my advice is to choose some other, simpler, primitive or immutable type to be a constant, like this `String`:

```
public static final String BERLIN_WALL_FELL = "November 9, 1989";
```

If you must use a mutable class instance as a constant, then I suggest you make it private and have a getter method that is public. Do not provide a setter method. In addition, recall that when you provide an object through a getter method, you are passing the reference to the original object, not sending a copy of the original object to the caller. If you just return your private member, and it is mutable, then the caller can change it. To avoid this problem, create a new instance of your constant class type by cloning the object, and return the clone (actually return a reference to the clone).

Here is how that all might look:

```
private static final Date BERLIN_WALL_FELL = new Date( 89, 10, 9 );
public static final Date getBERLIN_WALL_FELL() {
     return (Date)BERLIN_WALL_FELL.clone();
}
```

Because our constant is private, no one can modify it, unless we expose it. We allow people to get its value through the getter method, `getBERLIN_WALL_FELL()`. However, we don't return a reference to our constant, but rather we return a reference to a clone of our constant. We can do that because `Date` implements the `Cloneable` interface. Note that the `clone()` method returns an `Object`, which we have to cast as a `Date` to return.

The one who calls this getter could always change his personal reference to this `Date` or the value of the `Date`; but even so, each one who calls the original `getBERLIN_WALL_FELL()` method will get a `Date` with the correct value.

While we're on the subject of the `final` modifier, we should observe that the `getBERLIN_WALL_FELL()` method is declared `final` in the last section. This has a couple of effects. The first idea that we need to be prepared to adhere to is that anything we call `final` should never change. The primary reason is that *any* code that refers to something we've called `final` *will* need to be recompiled if we ever modify our code. We should consider these `final` methods to be constant methods. (Perhaps we should agree to indicate this fact by giving our methods all upper case names with underscore characters between words. There is no convention for that, yet.) The exceptions to this idea of recompiling all referencing code are difficult to identify, so this idea is like a rule you should follow.

`final` methods, like `final` members, can be automatically compiled inline, which means the code or value is included in the referrer byte code when compiled. This has some performance benefit, but that is usually outweighed by the requirement to recompile if the `final` code ever changes.

The second effect of declaring our methods to be `final` is that they cannot be overridden by some other class that implements our class. This is our primary reason for declaring methods in `OracleJavaSecure.java` to be `final`. We are protecting our logic from being modified or spoofed. We must abide the rule that our `final` methods may never change, or that the `final` methods are all `private` so that we are the only one that refers to them. Continue with this thought in the following section.

Getting RSA Public Key Artifacts

You may have noticed in Listings 5-1 through 5-4 that all our keys and components thereof are declared not only `static`, but also `private`. We do not give direct access to any of those. If you were very astute, you might have also noticed that our methods so far were declared `private`: that is `makeLocRSAKeys()`, `makeCryptUtilities()` and `makeExtRSAPubKey()` are all `private`. We never expect our client applications, nor our Java-stored procedures in Oracle to call these methods directly. We will define some `public` methods that will call these `private` methods as needed.

The first two `public` methods that we will meet now are for getting the public RSA key artifacts. We will ask client applications to call these methods shown in Listing 5-6.

Listing 5-6. *Methods to Get Public Key Artifacts,* `getLocRSAPubMod()` *and* `getLocRSAPubExp()`

```
private static BigInteger locRSAPubMod = null;
private static BigInteger locRSAPubExp;
public static final String getLocRSAPubMod() {
    String rtrnString = "getLocRSAPubMod() failed";
    try {
        if ( null == locRSAPubMod ) makeLocRSAKeys();
        rtrnString = locRSAPubMod.toString();
    } catch ( Exception x ) {
        x.printStackTrace();
    }
    return rtrnString;
}
public static final String getLocRSAPubExp() {
    String rtrnString = "getLocRSAPubExp() failed";
    try {
        if ( null == locRSAPubMod ) makeLocRSAKeys();
        rtrnString = locRSAPubExp.toString();
    } catch ( Exception x ) {}
    return rtrnString;
}
```

I'd like to point out several things in these `public` methods. First of all, notice that we test whether `locRSAPubMod` is `null`. Recall that when we first declared `locRSAPubMod` (at top of this code) we set it to `null`. (If we had declared it with this statement `private static BigInteger locRSAPubMod`, it would have also been `null`, but because we put it into an `if` test, we would get a "might not have been initialized" error when we try to compile.) If in the `if` test, `locRSAPubMod` is no longer `null`, then we must have already created the RSA key pair. However, if it is still `null`, then we call the `private` method `makeLocRSAKeys()`.

Also, note the test syntax, `if( null == locRSAPubMod )` which is identical to `if( locRSAPubMod == null )`. There is a benefit in using the first syntax. A common error, using one equals sign instead of two, will not be caught by the compiler in the second syntax. But in the first syntax, this would be flagged as an error `if( null = locRSAPubMod )`.

Each of these members, `locRSAPubMod` and `locRSAPubExp` are `BigInteger` classes – a type that doesn't exist in Oracle. I mentioned earlier that we often handle and transmit these large numbers as `Strings`, and that is the case here. We call the `BigInteger.toString()` method. When we get to the Oracle JVM, we will recreate `BigIntegers` based on these `Strings`.

We are returning a `String`, `rtrnString` and if there is a problem (an `Exception` or other problem getting `locRSAPubMod` or `locRSAPubExp`), then we can return an error message in our regular return value. You can see that we define the error message at the top of each method as `rtrnString`, in one case "`getLocRSAPubMod() failed`". Then if we are successful, we overwrite `rtrnString` with the requested key artifact. But in the case that there's a problem, we send back that original message. In a client/server application, we need to send meaningful error messages when there's a problem.

Because we are running on the client at this point, we can easily include a statement to dump the stack trace from the exception, as we do in the catch block of the first method, above. That is generally the first method called in our class, so it is a good place to catch an `Exception`, if there are going to be any. Note that in the code I'm providing, the troubleshooting and debugging has been accomplished and much of the error reporting has been removed.

Using Static Methods and Private Constructor

Recall that we have a new Oracle JVM for each Oracle session that needs Java. As long as we maintain that session, our JVM is retained. Also recall that Java stored procedures can only call `static` methods, because no instance is available to call from the stored procedure. We could always instantiate the `OracleJavaSecure` class and then use non-static methods of our instance; however, it will be much simpler, if we can get away with it, and we can, to make everything `static` that we do and touch inside our class. You can see that approach in the code examples we've seen so far.

Note that any members declared inside a method are not and cannot be declared as `static`. The `static` modifier can only be applied to items outside our methods, at the class level (i.e., class members).On the flip side, any class members referred to within a `static` method *must* be `static`. That is because in a `static` method, you can only refer to members in the class that also exist with or without an instance of the class (i.e., `static` members).

One way to assure that our class is ever and always `static` is to guarantee that it is never instantiated. We can guarantee that by making the default and only constructor for the class to be `private`. We have accomplished that with the code in Listing 5-7.

Listing 5-7. *Private Constructor*

```
private OracleJavaSecure() {
}
```

Instantiating a Connection Member from a Static Initializer

In our code, we will need to have an Oracle `Connection` member. When we run on the client, we will pass a `Connection` to a setter method to store the `Connection` for use in our class, but when running in the Oracle Database, we don't have a good method to use to instantiate the `Connection`. We could instantiate the `Connection` in every `public static` method where it might be used, but a better approach is to have a *static initializer*. That approach looks like what we have in Listing 5-8.

Listing 5-8. *Static Initializer for Connection*

```
private static Connection conn;
static {
    try {
        conn = new OracleDriver().defaultConnection();
    } catch ( Exception x ) {}
}
```

The `static` modifier in front of an independent block of code is a static initializer—this is not a method. Notice that we have to prepare to catch any exceptions that might be generated by instantiating the `Connection`. We do not have to do anything with the possible exceptions (if there is an `Exception`, then our problems are more systemic than our code), but we have to establish the `try /catch` blocks to make the code valid for compiling.

As a `static` block, it executes one time and exists in the class, outside of and without an instance of the class.

Using One Code for Both Client and Server

For me, one of the most enjoyable aspects of coding in Java and Java stored procedures is the ability to write code that operates on both sides of the client/server communication. This is especially helpful when it comes to testing our code—we can, in many cases, test it all on our workstation prior to deploying it to the Oracle database.

Some aspects of our code will only be executed on the client; for example the method to generate the RSA key pair. Likewise, some portions of our code will normally only be executed on the Oracle database, for example rebuilding our RSA public key from the artifacts. However for testing, we can execute the methods destined for Oracle on our workstation.

■ **Note** Please refer to Appendix A for a list of all the methods of `OracleJavaSecure` and the indication of their primary usage(client or Oracle Database or both).

In other instances, we will have to provide specific logic to permit our code to run in both environments. One case in point is the `Connection` member. In the last section, I mentioned the static initializer that will instantiate a `Connection` for use on the Oracle database. That code also runs on the client, but has no effect—there is no local, default Oracle database for the code to connect to. So we provide an additional method called `setConnection()` that will take a `Connection` from some client application code and set the `static Connection` member to that `Connection`, as shown in Listing 5-9.

Listing 5-9. *Setter Method for Client Connection,* `setConnection()`

```
public static final void setConnection( Connection c ) {
    conn = c;
}
```

We will have some additional examples of specific logic to run on either the client or the server, which I will point out as we go along.

In my member naming scheme, I could have chosen about any names with the same effect, but I chose to name the keys and artifacts that I generate locally with a loc prefix, like `locRSAPrivkey`. Artifacts passed to me and keys generated from those artifacts I prefix with an ext prefix (for "external"), like `extRSAPubKey`. This will help us keep things straight, especially as we execute both the client and the server side code on our workstation.

Testing on the Client

Our code in OracleJavaSecure is intended to be run as a client-server application to encrypt data on one end (e.g., server) and decrypt it at the other (e.g., client). However, it is nice to be able to test our code without introducing some of the complexities of client-server communication. So we will test all of our code so far on the client only. On the client we will generate the keys, encrypt data with the private key, build an equivalent public key from the public key artifacts, and decrypt data with the equivalent public key.

Writing the main() Method

In my experience, the best place to put test code is right in the class whose functions you are testing. Obviously you have to run your code in order to test it, and if you can run it from the command prompt, then you needn't build or depend on another class to do your testing. Whenever we are talking about executing from the command prompt, we are talking about using the `main()` method. The client-only testing portion of the `OracleJavaSecure.main()` method is shown in Listing 5-10.

Listing 5-10. *Method for Testing on the Client Only,* ***`OracleJavaSecure.main()`****, Part A*

```
public static void main( String[] args ) {
    try {
        // As a client application
        String clientPubModulus = getLocRSAPubMod();
        String clientPubExponent = getLocRSAPubExp();
        // Send modulus and exponent to Oracle Database, then
        // As if I were the Oracle Database
        makeExtRSAPubKey( clientPubModulus, clientPubExponent );
        cipherRSA.init( Cipher.ENCRYPT_MODE, extRSAPubKey, random );
        Date clientDate = new Date();
        String sampleData = clientDate.toString();
        byte[] clearBytes = sampleData.getBytes();
        byte[] cryptBytes = cipherRSA.doFinal( clearBytes );
        // Send the cryptBytes back to the client application, then
        // As a client application
        cipherRSA.init( Cipher.DECRYPT_MODE, locRSAPrivKey );
        byte[] newClearBytes = cipherRSA.doFinal( cryptBytes );
        String newSampleData = new String( newClearBytes );
        System.out.println( "Client date: " + newSampleData );
    } catch ( Exception x ) {
        x.printStackTrace();
    }
    System.exit( 0 );
}
```

In the `main()` method, we have two method `String` members, `clientPubModulus` and `clientPubExponent`. We set those members by calling our two public methods, `getLocRSAPubMod()` and `getLocRSAPubExp()`. We are acting as the client application, calling those methods. We continue to act as the client application, but only imagine sending those values to the Oracle Database (at this point, we are just going to use those values locally.)

Now, pretending that we are the Oracle Database, we use those values we just "received," and we call the `makeExtRSAPubKey()` method in order to generate a replica of the public key. (Although, on the server that method is `private` and will not be called directly.) Then the server uses (we use) that public key to encrypt some clear sample data. The sample data, `sampleData` we are using is the date and time that we get by instantiating a new `Date` class.

We initialize `cipherRSA` using `extRSAPubKey`, the public key copy that we built, and we set it to `ENCRYPT_MODE`. Then we call the `doFinal()` method, encrypting the `clearBytes into cryptBytes`. Note again that the `cipherRSA.doFinal()` method takes a byte array as its parameter and returns a byte array, so we convert the `sampleData String` to a byte array using `String.getBytes()`, in order to submit the `clearBytes` to `cipherRSA.doFinal()`. We end up with `cryptBytes`, a byte array of encrypted data.

Then back at the client, we "receive" the `cryptBytes` encrypted byte array. (The process of calling the Oracle Database to return encrypted data is a bit more complex than this first test, and we will see the details in the second half of this chapter.) Acting as the client, then, we initialize our own `cipherRSA` using the `locRSAPrivKey` private key, to `DECRYPT_MODE.` Then we call the `doFinal()` method. This should return the original clear byte array, from which we create a new `String` instance named `newSampleData`, and which we print. If all has gone well, what we will see is the `newSampleData`, the client date and time as the output.

Notice at the end of Listing 5-10 that there is a call to `System.exit(0)`. That call actually terminates the running JVM. It is important to include that call at the end of most `main()` methods, because if your application is running other threads of code, the JVM may not terminate when you get to the end of `main()`, even though you are done with it. Be aware that you do not normally call `System.exit(0)` in `main()` for GUI applications where your `main()` method displays the GUI interface and then ends. The `main()` ends, but the GUI exists until some control in the GUI (like an **Exit** button) issues a similar `System.exit(0)` call. By the way, the `0` (zero) means exit with an OK status code. Any other number is considered an error code which can be sensed by the operating system, as needed.

Also take note of the `catch` block and our call to `Exception.printStackTrace()`. With that call, any exception caught in the `main()` method will print out a list of each consecutive method call and the line number from which each consecutive call is made. The line number in `main()` where the exception occurred will be at the bottom of the list and the method it called and line number where the exception occurred will be listed above that, and so on. At the top of the list will be the specific exception that occurred and the method and line number where it happened. That list is called a stack trace.

Running the Code

If you have set your `PATH` and `CLASSPATH` as described in Chapter 3, then your job is easier. First run a command prompt (**Start Menu ↗ Programs ↗ Accessories ↗ Command Prompt**). Then change directories to *Chapter5*. Compile the code with this command:

```
javac orajavsec/OracleJavaSecure.java
```

Then run the code from that same directory with this command:

```
java orajavsec.OracleJavaSecure
```

If you have not set your `PATH` and `CLASSPATH`, then you need to locate your compiler (*javac.exe*). And you need to go to the same directory as mentioned previously. For example, if *javac.exe* is in

C:\java\jdk1.6\bin and *ojdb6.jar* is in *C:\java*, you would execute the following commands (from a command prompt in the *Chapter5* directory):

```
C:\java\jdk1.6\bin\javac.exe -cp .;C:\java\ojdbc6.jar orajavsec/OracleJavaSecure.java
C:\java\jdk1.6\bin\java.exe -cp .;C:\java\ojdbc6.jar orajavsec.OracleJavaSecure
```

As a result, you will see the client date displayed:

```
Client date: Sat Dec 04 11:29:39 EST 2010
```

This date `string` was generated, encrypted with the public key and decrypted with the private key all on the client.

Key Exchange

So far, we have created our RSA private and public keys and demonstrated building a copy of the public key with two components: the exponent and the modulus. We have also demonstrated encrypting a `Date` string with our copy of the public key and decrypting it with the private key. These are all the aspects of our RSA key pair that we are going to utilize; however, we will be building the copy of the private key on the Oracle database, doing our encryption there and decrypting the data with the private key on the client. To do that, we are going to need some Oracle structures.

We will define a function and a procedure in our Application Security, `appsec` schema. To do this, connect to Oracle as the `appsec` user and set your role to the privileged `appsec_role`:

```
CONNECT appsec;
SET ROLE appsec_role;
```

■ **Note** You can find a script of the following commands in the file named *Chapter5/AppSec.sql*.

Creating a Function to Encrypt Data with Public Key

The Oracle script in Listing 5-11 defines a function that calls the `OracleJavaSecure.getRSACryptData()` method. This will encrypt data using the copy of the RSA public key that we will build on the Oracle Database. Execute the command in Listing 5-11 while connected to Oracle Database as the Application Security user, `appsec`.

Listing 5-11. *Oracle Function to RSA Encrypt Data*

```
CREATE OR REPLACE FUNCTION f_get_rsa_crypt(
    ext_rsa_mod VARCHAR2, ext_rsa_exp VARCHAR2, cleartext VARCHAR2 )
    RETURN RAW
    AS LANGUAGE JAVA
    NAME 'orajavsec.OracleJavaSecure.getRSACryptData( java.lang.String, java.lang.String,
java.lang.String ) return oracle.sql.RAW';
/
```

There are three `VARCHAR2` (`String`) parameters that we are going to pass to the function: the public key modulus, the public key exponent, and the clear-text string that we want to encrypt.

Note that the NAME keyword must point to a single line of text. Here we have word wrap, but the definition of the Java method must all appear on a single line. Additionally, both the input and output parameters must be fully qualified—they must have the entire path prepended to the Class name. That includes classes that are naturally part of the Java language like java.lang.String.

Recall that Oracle functions can take parameters, do work and return a single value. In this case, we are going to encrypt a string and return it as a RAW data type. RAW is an immutable byte array that can be transferred between Oracle Database and Java without any interpretation or translation (casting). RAW data retains *fidelity* between Oracle and Java, server and client. The limit of a RAW data element is either 32767 bytes in PL/SQL code (our case) or 2000 bytes when stored in the database.

Creating a Procedure to get SYSDATE in Encrypted Form

We will add a procedure that does not call Java, so it is not a Java stored procedure, but it calls our Java stored function, f_get_rsa_crypt. Execute the command in Listing 5-12 as the appsec user.

Listing 5-12. *Oracle Procedure to Encrypt Date and Time String,* ***p_get_rsa_crypt_sysdate***

```
CREATE OR REPLACE PROCEDURE p_get_rsa_crypt_sysdate(
    ext_rsa_mod    IN VARCHAR2,
    ext_rsa_exp    IN VARCHAR2,
    crypt_sysdate OUT RAW,
    m_err_no      OUT NUMBER,
    m_err_txt     OUT VARCHAR2 )
IS BEGIN
    m_err_no := 0;
    crypt_sysdate := f_get_rsa_crypt( ext_rsa_mod, ext_rsa_exp,
        TO_CHAR( CURRENT_TIMESTAMP, 'DY MON DD HH24:MI:SS TZD YYYY' ) );
EXCEPTION
    WHEN OTHERS THEN
        m_err_no := SQLCODE;
        m_err_txt := SQLERRM;
END p_get_rsa_crypt_sysdate;
/
```

This is fairly simple procedure. It gets the CURRENT_TIMESTAMP from the Oracle Database, converts it to a string using the TO_CHAR SQL function, uses f_get_rsa_crypt (that we defined previously) to encrypt it, and then returns it to the caller.

So why have I complicated it with what appears to be error handling? Well now is as good a time as any to introduce a template for procedures of this type. We don't want to burden the Oracle database with the task of dealing with most of the errors our procedures may generate, and we also want to be able to deal with the errors back at our application. We need Oracle database to tell us what the error is, and we need to be able to deal with it in a logical fashion.

For this reason, we have declared several of our parameters as OUT parameters. The first OUT parameter is a RAW that contains our encrypted timestamp string. The second OUT parameter is the error number that is generated by Oracle Database – every Oracle error type has a distinct error number that is indexed in the Oracle documentation. And the third OUT parameter is the error message.

In our template, when an exception occurs, we don't throw it back to the calling application; rather, we allow the procedure to complete in an orderly way and return the error number (SQLCODE) and message (SQLERRM) to the caller in the OUT parameters. Back in the calling code, before we handle any other of the returned data, we will check to see if the error number is anything besides 0, and if it is, we

report it to the user or handle it in the application. If there is an error, it is usually not beneficial to continue processing the data. There will be examples of this template in action later in our code.

A WORD ABOUT TIME ZONES

You'll notice that Listing 5-12 gets the current time of the Oracle Database with this call:

```
TO_CHAR( CURRENT_TIMESTAMP, 'DY MON DD HH24:MI:SS TZD YYYY' )
```

The goal is to match the time format with what Java on the local client gives us, just for presentation's sake. The time right now (for me), from Java is:

```
Sat Dec 04 11:29:39 EST 2010
```

Notice the format string includes the element "TZD" which presents the time zone. My time zone is EST (eastern standard time). We use CURRENT_TIMESTAMP instead of SYSDATE in order to get the time zone. Oracle timestamp data must be directed to retain time zone if you want that. Generally, Oracle database assumes the time zone of the client for data generated during a session; however, often the time zone provided by the session is some number of hours offset from Greenwich mean time (GMT). Living in the southeast United States with daylight saving time means that our time zone is usually minus 5 hours from GMT.

Note that to get completely identical results, you would have to uppercase the date `String` returned by Java. When we compare the results to what is returned by Oracle database, we'll see that Oracle gives names of the days of the week and names of the months in all uppercase. Looking ahead, we will see these date and time representations:

```
Client date: Sat Dec 04 14:59:49 EST 2010
Server date: SAT DEC 04 14:59:50 EST 2010
```

You'll be interested to know (or just consider) that many cities, states, countries and continents share this offset (minus 5 GMT). Therefore, knowing the offset doesn't guarantee that Oracle database can report the time zone in TZD format; although, the offset can be reported.

With some Oracle clients, like TOAD, the client does not inform the server what time zone name to use. The server knows the offset, but not the name. Observe the output from these commands to observe the phenomenon.

```
SELECT TO_CHAR( CURRENT_TIMESTAMP, 'DY MON DD HH24:MI:SS TZD YYYY' ) FROM DUAL;
SELECT * FROM sys.gv_$timezone_names
    WHERE tzname LIKE 'America%' --AND tzabbrev = 'EST'

;
ALTER SESSION SET TIME_ZONE = 'America/New_York';
SELECT TO_CHAR( CURRENT_TIMESTAMP, 'DY MON DD HH24:MI:SS TZD YYYY' ) FROM DUAL;
```

After altering our session to specify our time zone name, Oracle database can correctly report EST or whatever is your current time zone. Here's what I see as the results:

```
SAT DEC 04 11:57:49  2010
```

```
SAT DEC 04 11:58:31 EST 2010
```

On the other hand, the Java client automatically informs Oracle database of its time zone name, so we do not need to manually set it. There is more complexity to time zones that I am not covering here, especially as it relates to daylight saving time.

Loading OracleJavaSecure Java into Oracle Database

Now that we have our Oracle function and procedure defined, we are ready to put our Java logic (code) into the Oracle Database. There are a couple ways to do this. The first would be to use the *loadjava* utility that comes with the Oracle client and server software. We have made that a bit difficult for ourselves since the role that is needed to accomplish that command as the application security, `appsec` user is not a default role. Instead of using *loadjava*, let's just connect to Oracle Database (or remain connected) as `appsec`.

The next line is at the top of our Java code in the *OracleJavaSecure.java* file. Uncomment it and copy the entire code into your Oracle client.

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/OracleJavaSecure" AS
…
```

For security, scroll down to the class body and *assure you do not have a valid password* in the connection string. If you do, remove the password from the connection string before executing this command in Oracle Database. You do not want to store any passwords in the code on Oracle – they are not needed there, and having them there makes them available to hackers.

```
private static String appsecConnString =
    "jdbc:oracle:thin:AppSec/password@localhost:1521:Orcl";
```

Execute the script in your Oracle client (e.g., SQL*Plus) to load the Java code into the Oracle Database, which compiles it.

Encrypting Data with Public Key

We saw in our definition of the Oracle function `f_get_rsa_crypt`, previously, that we intended to call a method named `getRSACryptData()`. We have just loaded that Java method into Oracle Database in the last section. The code for `getRSACryptData()` is shown in Listing 5-13. We will have the Oracle Database encrypt data with the public key – later we will decrypt it at the client with the private key.

Listing 5-13. *Encrypt Data with Public Key,* `getRSACryptData()`

```
public static final RAW getRSACryptData( String extRSAPubMod,
    String extRSAPubExp, String clearText )
{
    RAW rtrnRaw =
        new RAW( "getRSACryptData() failed".getBytes() );
    try {
        if ( ( null == extRSAPubKey ) ||
            ( !saveExtRSAPubMod.equals( extRSAPubMod ) ) )
            makeExtRSAPubKey( extRSAPubMod, extRSAPubExp );
        cipherRSA.init( Cipher.ENCRYPT_MODE, extRSAPubKey, random );
        rtrnRaw = new RAW( cipherRSA.doFinal( clearText.getBytes() ) );
    } catch ( Exception x ) {}
```

```
        return rtrnRaw;
    }
```

During development, I had occasions to catch exceptions in `getRSACryptData()` and similar methods, but in the debugged code, I'm depending on another mechanism to report errors. You can see that the `catch` block has a close brace } immediately after the open brace {. That is a way of ignoring the exception. This method will successfully return a valid `RAW` value whether an exception is thrown or not. If there is a problem completing the encryption (the last line within the `try` block), then our original "error" message is returned in the `RAW`. That original message informs the client that the method failed. We translate the message into a `RAW` by getting the bytes of the `String` from `getBytes()` and sending them to the `RAW` constructor. If there is no problem when executing the code, then our member `rtrnRaw` is set to the encrypted value, and that is returned; else the error message is returned in `RAW` form.

The first test we do in `getRSACryptData()` is to see whether we have already built a copy of the RSA public key. We pass this method the public artifacts each time we call it, but only want to go through building the key one time. So, we test if `extRSAPubKey` is `null`. We also test to see if the value of the modulus artifact has changed (that should never be the case). If either of these is true, we build our copy of the RSA public key by calling `makeExtRSAPubKey()`.

We initialize the `cipherRSA` to do encryption, `Cipher.ENCRYPT_MODE`. Then we encrypt our clear text by calling the `doFinal()` method.

After this chapter, we will not be calling `getRSACryptData()` directly from our client code, so we will be changing this from a `public` to a `private` method. Eventually, the only thing we will be encrypting with our RSA public key is a secret passphrase. We will be exploring that in the next chapter.

Use Stacked Calls

Let me point out the syntax in one line of code from Listing 5-13:

```
rtrnRaw = new RAW( cipherRSA.doFinal( clearText.getBytes() ) );
```

We call the `getBytes()` method of the `clearText String`, then instead of pointing a byte array member at it, we pass the bytes directly to the `cipherRSA.doFinal()` method to encrypt (it takes a byte array and returns a byte array). Then instead of pointing a local byte array member at the encrypted bytes, we pass the array to a new `RAW` instance constructor.

I call this syntax stacked calls, and I like it. I prefer densely packed code over spacious code. It's just my opinion that it reads and prints easier. The key to reading stacked calls is to look at the innermost pair of parentheses first, and read from the inside out. This is more difficult to read from left to right than from right to left.

Decrypting Data with Private Key

The next method that we need for the demonstrations in this chapter is a method to decrypt the data on the client with our RSA private key. Listing 5-14 defines that method.

Listing 5-14. *Decrypingt Data with Private Key,* ***`getRSADecryptData()`***

```
    public static final String getRSADecryptData( RAW cryptData ) {
        String rtrnString = "getRSADecryptData failed";
        try {
            cipherRSA.init( Cipher.DECRYPT_MODE, locRSAPrivKey );
            rtrnString = new String( cipherRSA.doFinal( cryptData.getBytes() ) );
        } catch ( Exception x ) {
```

```
            x.printStackTrace();
        }
        return rtrnString;
    }
```

This method takes a `RAW` parameter named `cryptData`. The code gets the byte array from `cryptData` and passes it to the `cipherRSA.doFinal()` method to decrypt. We instantiate a new `String` from the decrypted byte array, and return that `String` to the caller.

You will recognize the similarities in error handling between this and the `getRSACryptData()` method described above. One additional measure we are taking here is to print out the stack trace of any `Exception` that's thrown.

This method is not needed in future chapters, because we will not be encrypting just any old data with our RSA public key; rather, with the RSA keys we will only be encrypting / decrypting specific artifacts of our secret password key that we create in the next chapter. So we will repackage this decryption call specifically for that purpose.

Testing on Client and Server

Earlier in this chapter, you compiled *OracleJavaSecure.java* on your workstation. Let's make a couple changes to the code now. First of all, if you saved the version that we installed on the Oracle Database, then you may have uncommented the first line. If so, comment it again:

```
//CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED appsec."orajavsec/OracleJavaSecure" AS
```

Down a little ways from the top of the file, in the body of the class code, set the `boolean` named `testingOnServer` to `true`. That change will cause the second half of the `main()` method to execute.

```
    private static boolean testingOnServer = true;
    private static String appsecConnString =
        "jdbc:oracle:thin:AppSec/password@localhost:1521:Orcl";
```

Additionally, we want to connect from this client code to the Oracle Database, so edit the next line down with the correct host, port, instance and password needed to connect as `appsec` user. Be sure to save your changes.

Inside the `main()` method, after the code we discussed previously, we have another section for testing client/server public key encryption. We check the `testingOnServer boolean` that we set previously, and if `true` we continue our testing. The first thing we want to do is establish our Oracle `Connection`, `conn` based on the `appsecConnString` we edited above. See Listing 5-15.

Listing 5-15. *Testing PKE on Client and Server*

```
    if( testingOnServer ) {
        // Since not on the Server, must load Oracle-specific Driver
        Class.forName( "oracle.jdbc.driver.OracleDriver" );
        // This will set the static member "conn" to a new Connection
        conn = DriverManager.getConnection( appsecConnString );
```

Using IN and OUT Parameters in an OracleCallableStatement

Also in the `main()` method, you will see our code to prepare an `OracleCallableStatement`. We are using a `Statement` class of that type so that we can read the `OUT` parameters. Notice as we prepare the call, in Listing 5-16, that we list the procedure name and have a list of question marks for the parameters we are

going to pass. We also register parameters 3, 4, and 5 as OUT parameters that we will be able to read later. If we fail to register the parameters that way, we may be surprised when we go to read them later—they won't be available.

Listing 5-16. *An OracleCallableStatement for Client/Server Encryption*

```
OracleCallableStatement stmt =
    ( OracleCallableStatement )conn.prepareCall(
    "CALL p_get_rsa_crypt_sysdate(?,?,?,?,?)" );
stmt.registerOutParameter( 3, OracleTypes.RAW );
stmt.registerOutParameter( 4, OracleTypes.NUMBER );
stmt.registerOutParameter( 5, OracleTypes.VARCHAR );
stmt.setString( 1, clientPubModulus );
stmt.setString( 2, clientPubExponent );
stmt.setNull(   3, OracleTypes.RAW );
stmt.setInt(    4, 0 );
stmt.setNull(   5, OracleTypes.VARCHAR );
stmt.executeUpdate();
```

After preparing the Statement, we set the input parameters. The only parameters we have to pass to the procedure are our RSA public key modulus and exponent. We call stmt.setString() for each of those parameters.

Notice, however, that we set the remainder of the parameters to null using the stmt.setNull() method, or to 0 in the case of the int. There are two things to mention about setting these to null: first, it may not be necessary because we don't read or test the values of those parameters in the Oracle procedure, but we do it to satisfy our desire to *address all potential concerns*. Consider it to be just one more part of the secure programming approach—cover all your bases. Second, these are the same parameters that we are expecting to read from—OUT parameters. We want to assure that unless the Oracle procedure puts a value in those parameters, there will be nothing there to read.

Oracle procedures can define their parameters as one of three types: IN (the default), OUT, and INOUT. On occasion you will want to use a single INOUT parameter to both submit data to an Oracle procedure and to read the results coming back. I find it can be more concise to use INOUT parameters, but it can also be a bit confusing in the procedure code—deciding what value is currently being handled. So, for a more security conscientious approach, we will avoid INOUT parameters.

Both when we register OUT parameters and when we set those parameters to null, we have to declare the data type of the parameter. We refer to the types in the OracleTypes class, as in OracleTypes.RAW and OracleTypes.NUMBER. We find OracleTypes in the *oracle.jdbc* package (from *ojdbc6.jar*) and import it from there. Earlier we mentioned this one inconsistency in the use of VARCHAR instead of VARCHAR2. It is in the OracleTypes class, as you can see in Listing 5-16.

The last line of this section of code executes the Statement. That is, it calls the Oracle stored procedure, passing the IN parameters, and retrieving the OUT parameters.

Handle Errors Reported by Oracle Database

We will build our Oracle procedures to send the error number and error message back in two of the OUT parameters. This will be our standard approach for handling errors in processing—it places the responsibility for dealing with errors into the hands of the application developer. This code from the main() method, Listing 5-17, follows the execution of the Statement. First we read the errNo parameter. If it is not equal to zero, some problem has occurred, and we print out the error message; otherwise, we will continue.

Listing 5-17. *Handling Errors from a Stored Procedure*

```
int errNo = stmt.getInt( 4 );
if( errNo != 0 ) {
    String errMsg = stmt.getString( 5 );
    System.out.println( "Oracle error " + errNo + ", " + errMsg );
    System.out.println( (stmt.getRAW( 3 )).toString() );
}
```

Notice how we also print out the data that is returning to us in the `RAW` parameter, number 3. Normally that would hold the encrypted data (`SYSDATE`) from the `p_get_rsa_crypt_sysdate` procedure; but when there's an error, it will hold the name of the Java method, running in the Oracle JVM, where the error occurred. This information is difficult to gather unless you go to these efforts, or log the errors in the Oracle JVM and read through the logs. We will do both.

Decrypting at the Client

Normally, when there is no error, we read our non-error-related `OUT` parameter, number 3, the `RAW` element that is our encrypted data. Here we read it and assign the value to the `cryptData RAW`. Then we call the `getRSADecryptData()` method to decrypt it. And finally, we print out the data, which is the server timestamp. These actions are shown in Listing 5-18.

Listing 5-18. *Decrypting the Data from the Stored Procedure*

```
else {
    RAW cryptData = stmt.getRAW( 3 );
    newSampleData = getRSADecryptData( cryptData );
    System.out.println( "Server date: " + newSampleData );
}
```

Running Our Code Again

If you have any troubles completing this, refer to the previous section in this chapter, "Running Our Code." You will have to have loaded the Java code on the Oracle Database as described in the section "Load OracleJavaSecure Java into Oracle Database."

You will change directories to *Chapter5* and compile the code with the following command. As a reminder, you will need to have *ojdbc.jar* in your `CLASSPATH`.

```
javac orajavsec/OracleJavaSecure.java
```

Then run the code from that same directory with this command:

```
java orajavsec.OracleJavaSecure
```

Observing the Results

With the edits you made in the second half of this chapter to *OracleJavaSecure.java*, the `main()` method will continue to run the client/server tests. The first line shown below is printed from the encryption/decryption methods running on the client only; and the second line comes from the client/server encryption/decryption process:

```
Client date: Sat Dec 04 14:59:49 EST 2010
```

```
Server date: SAT DEC 04 14:59:50 EST 2010
```

Both of these sets of test used our rebuilt copy of the RSA public key to encrypt the date `String`, and used the RSA private key to decrypt the date `String`. In the second set of tests, the public key was rebuilt on the Oracle Database and the date `String` was encrypted there. Then the encrypted `RAW` data was returned to the client, where it was decrypted and printed. Ta da! You have accomplished client-server encrypted data transmission! This process is depicted in Figure 5-1.

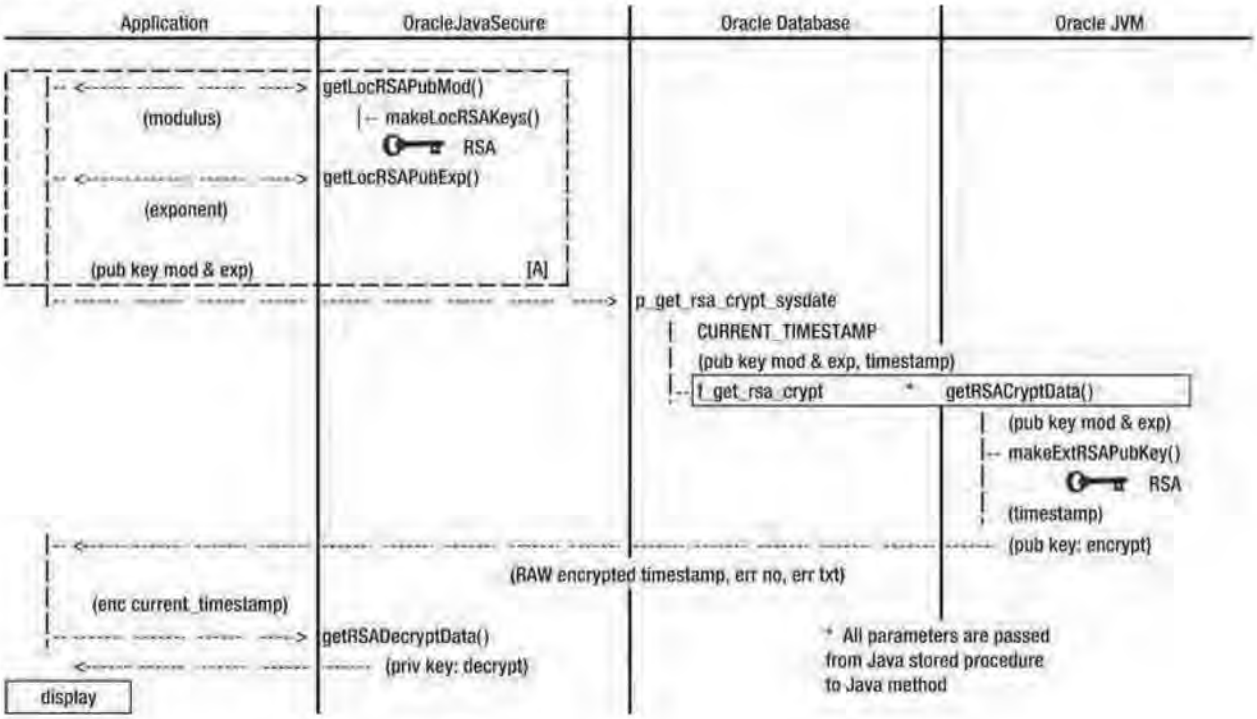


Figure 5-1. *Client/server public key encryption*

There is a section of Figure 5-1 that is outlined and labeled as [A]. That section is a depiction of the generation of the RSA key pair. Within that outline is an image of a key which represents the RSA key pair as it exists on the client.

The client application makes one call to the Oracle Database, to the procedure named `p_get_rsa_crypt_sysdate`. In that call, the RSA public key modulus and exponent are sent to the database. On the database, a java stored procedure, `f_get_rsa_crypt` is called, which in turn calls Java in the Oracle JVM, to a method named `getRSACryptData()`. That is the starting point for building an equivalent RSA public key, depicted by the key image on the right side of Figure 5-1.

The Oracle JVM uses the equivalent public key to encrypt the `CURRENT_TIMESTAMP` value from the Oracle database, and then returns that encrypted value to the client application. From there, the client will decrypt the value using the RSA private key. The private key only exists in this client, so only this client can decrypt the data being returned by the Oracle database.

Removing the Demonstration Oracle Structures

We are not going to need the Oracle function or procedure that we built in this chapter. They were just for our use in demonstrating our client/server RSA public key encryption. We will be building some

more-enduring functions and procedures in the next chapter. But because we are done here, let's clean up after ourselves and remove these structures with the following commands as `appsec` user:

```
DROP PROCEDURE p_get_rsa_crypt_sysdate;
DROP FUNCTION f_get_rsa_crypt;
```

Chapter Review

Perhaps this was a lot of ground to cover in a few short pages, but there wasn't a lot of code required. I think you'll agree that if this is all that's required to do Public Key Encryption, we should all be doing it!

Here is a list of PKE-related steps we took:

- Created the public and private key set.
- Got the modulus and exponent of the public key.
- Re-created the public key from the modulus and exponent.
- Generated an appropriate `Cipher.`
- Initialized the `Cipher` to do encryption or decryption using either the public or private key.
- Called the `Cipher.doFinal()` method to do both encryption and decryption.

In addition to the PKE aspects, we learned about the `Serializable` interface and about how we can serialize objects to pass them around and store them.

We also established a standard form for calling Oracle procedures from an `OracleCallableStatement` that will allow us to exchange data and provide good error reporting back to the client, in a controlled fashion to be handled by the application programmer.

We discussed `final` methods, Java constants, `static` methods, `private` constructors, and `static` initializers, with an aside on time zones.

All of this took place within our introductions to the `OracleJavaSecure` class that we will be building and extending during the course of this book.

CHAPTER 6

Secret Password Encryption

What I am calling secret password encryption (password encryption) is also known as shared secret passphrase encryption and password-based encryption (PBE). Basically, the idea is that a single passphrase (or password—I will be using those words interchangeably) is known by two parties, and they each encrypt messages or data to pass to each other using that password. The same password is used by the recipient to decrypt the messages and data. No one else can decrypt the data because the password is a secret, shared only by the two parties.

Password encryption is beneficial to us for several reasons, chief of which is that it can be used for encryption of larger blocks of data. We will be using the U.S. data encryption standard (DES) with cipher block chaining (CBC) to automatically break any size data into appropriate blocks for encryption; then upon decryption, assemble the results back into the original data.

Another benefit of password encryption is that there is no public key—that is, no one else knows the key we are using to encrypt the data, assuming we have adequately protected the password. An alternative approach is to have public key encryption from both client and the server, each encrypting the data with the other's public key, only to be read by the recipient. And for added assurance, have each encrypt their messages first with their own private key, then with the other's public key. Think that through and you'll see that not only can only the recipient decrypt the message, but also only the expected sender could send it. (Add a trusted Certificate Authority (CA) like VeriSign and you have identity assurance as well—everyone is who they say they are.) However, we are going to get most of those benefits with only our secret password encryption.

We are going to create a passphrase on the Oracle server and pass it back to the client in secret. We will make it secret by encrypting the passphrase with the client's RSA public key. To do this, the client will have already passed the public key artifacts (modulus and exponent) to the Oracle server. Only the client can decrypt the secret passphrase using the private key.

One last benefit of using password encryption in addition to RSA public key encryption is that any assailant will have to attack both protocols to intercept our data.

Approach

As you read through this chapter, you will want to open the referenced files to follow along in the full code listing. First we are going to discuss the Java code that we implement for secret password encryption, because it is in Java that we will be building the encryption key and doing the encryption. However, we will not compile and run the Java code until the end of the chapter. Then we will run it in two phases: the first will be doing both encryption and decryption in Java on the client computer; the second phase will accomplish secret password encryption key exchange between the client computer and Oracle database and will demonstrate client/server encryption/decryption.

Before we get to the testing phases at the end of the chapter, we will also discuss the Oracle SQL code required for this process. Feel free to execute the SQL code to create the structures we need on

Oracle database as we discuss it, or you can execute all the SQL code at the end before we do the phase two tests.

Java Code for Secret Password Encryption

We are returning to add code to and edit the `OracleJavaSecure` class that we introduced in the last chapter. This class will form the core of all our security processes on both the client and the Oracle database. You will benefit by opening *OracleJavaSecure.java* and referring to the full code listing as we progress through this section. We will replace the `OracleJavaSecure` class in the Oracle database and compile and run the updated code on our client computer when we get to the end of the chapter and do some testing.

Sharing the Artifacts of a Secret Password Key

There are several artifacts of DES password-based encryption that must be shared between the client and server in order to have an identical encryption key and `Cipher` at each end. First, there is a passphrase, which is known only to the two parties involved in the encrypted dialog.

There are two other artifacts that must be shared, but are often treated as constants in a specific context. Those are the salt and the iteration count. Fixing those two parameters as constants is a major weakness. Vendors will often obfuscate (make hidden) their code in order to hide those values. Any hacker who can steal the salt and iteration count has a leg up on decrypting your data.

Our plan is to generate all three artifacts to be different for each session. We will use the `SecureRandom` instance to generate a random iteration count and a random salt. We will also generate a maximum-length passphrase from random acceptable characters.

Generating the Password and Artifacts

While we're on the subject, let's go ahead and see how we generate these artifacts. We do it in the `makeSessionSecretDESPassPhrase()` method, the code for which is shown in Listing 6-1.

Listing 6-1. *Generating DES Password Artifacts* `makeSessionSecretDESPassPhrase()`

```
    private static SecureRandom random = new SecureRandom();

    private static final int SALT LENGTH = 8;
    private static int maxPassPhraseBytes;
    private static char[] sessionSecretDESPassPhraseChars = null;
    private static byte[] salt;
    private static int iterationCount;
    private static void makeSessionSecretDESPassPhrase() {
        // Pass Phrase, Buffer size is limited by RSACipher class (on Oracle JVM)
        // Max size of data to encrypt is equal to the key bytes minus padding
        // (key.bitlength/8)-PAD PKCS1 LENGTH (11 Bytes)
        maxPassPhraseBytes = ( keyLengthRSA/8 ) - 11;
        sessionSecretDESPassPhraseChars = new char[maxPassPhraseBytes];
        for( int i = 0; i < maxPassPhraseBytes; i++ ) {
            // I want printable ASCII characters for PassPhrase
            sessionSecretDESPassPhraseChars[i] =
                    ( char )( random.nextInt( 126 - 32 ) + 32 );
```

```
        }
        // Appreciate the power of random
        iterationCount = random.nextInt( 10 ) + 15;
        salt = new byte[SALT LENGTH];
        for( int i = 0; i < SALT LENGTH; i++ ) {
            salt[i] = ( byte )random.nextInt( 256 );
        }
    }
```

■ **Note** You can find this code in the file *Chapter6/orajavsec/OracleJavaSecure.java.*

Calculating the Size of the Password

The first thing you'll notice in Listing 6-1 is a definition of the maximum size of the passphrase. It is calculated based on the size of the RSA key.

```
maxPassPhraseBytes = ( keyLengthRSA/8 ) - 11;
```

The reason we base our password length on the size of the RSA key is that we are going to encrypt our passphrase with the public key, and RSA can only encrypt a byte array smaller than its own key, minus the padding.

We could have made `maxPassPhraseBytes` a constant, but we might have been tempted to try something larger (okay, I tried it) had we not known the derivation. Also, we may at some point increase the RSA key size, which will automatically translate into a larger passphrase length.

Respecting the Power of Random

In our method, shown in Listing 6-1, we instantiate a byte array the size of `maxPassPhraseBytes` and populate it with random characters in the `for` loop. Notice the parameters for the random character. We want our passphrase to be made up of printable characters in the ASCII range of 32 to 126.

We will set the iteration count to be a random number between 15 and 25.

And we will populate the salt byte array with random bytes between 0 and 256. Our salt byte array size is fixed at eight bytes. We declare `SALT LENGTH` as a constant (static final).

All of these artifacts of our password-based encryption will be generated for each Oracle session and passed back to the client. They will be encrypted using the RSA public key prior to transmission. The client will decrypt them using the private key. Then with those artifacts in hand, the client will create an identical secret password key to use in sending and receiving encrypted data.

Initializing Static Class Members

We are going to move the initialization of two static class members from their previous location in a method out into the class body. See Listing 6-2. Instead of two separate calls to the `makeCryptUtilities()` method (one on the client and a different one on the server), we unify the process of creating these components. We will instantiate the `SecureRandom` at the point of definition, and we will instantiate the `cipherRSA` in a static initializer block (catching Exceptions, as required).

Listing 6-2. *Static Class Members*

```
private static SecureRandom random = new SecureRandom();
private static Cipher cipherRSA;
static {
    try {
        cipherRSA = Cipher.getInstance( "RSA" );
    } catch( Exception x ) {}
}

private static Cipher cipherDES;
private static SecretKey sessionSecretDESKey = null;
private static AlgorithmParameterSpec paramSpec;

private static String sessionSecretDESAlgorithm = "PBEWithSHA1AndDESede";
```

In addition to the artifacts of the DES password-based encryption key described in the previous section, we will declare the key itself and the DES `Cipher` we will be using. We will also build and use an `AlgorithmParameterSpec` member for the DES key.

Evaluating the Java 1.5 Password-Based Encryption Bug

We get to select what algorithm we intend to use for our secret password encryption. U.S. (DES) is good, but Triple DES (3DES or DESede) is stronger, and AES even more so; and SHA1 is a better cryptographic hash than MD5. So we would like to use the password-based encryption algorithm fully indicated by `PBEWithSHA1AndDESede`, and we specify it as shown in Listing 6-2.

But wait, there is a problem. In the Java Runtime Edition version 1.5, there is a bug. It is reported at this link: `http://bugs.sun.com/bugdatabase/view bug.do?bug id=6332761`.

The Oracle JVM is based on JRE 1.5, so it manifests that bug. When we generate our secret key specifying our preferred algorithm, the key generator will return a weaker key of type `PBEWithMD5AndDES` instead.

Coding an Automatic Upgrade: Negotiated Algorithm

It is the job of programmers to debug issues like the bug described in the previous section when we run across them, and to build bridges over the obstacles. We will have a method that shows us the algorithm in actual use; that will display the bug. Additionally, we will not assume that what we specify is what we get—we will return the actual algorithm to the client and build the copy of the secret password key using the actual algorithm.

The benefits of this approach are that we will negotiate a common algorithm and continue to specify a stronger encryption algorithm. Continuing to specify the stronger algorithm will predispose our code to use the stronger algorithm whenever it becomes available in the Oracle JVM.

So we will specify `PBEWithSHA1AndDESede`, but at this time we will be using `PBEWithMD5AndDES`. When Oracle next upgrades the Oracle JVM, we are prepared to use the stronger algorithm. Both of these algorithms use CBC as their mode, so they will work equally well for what we are doing.

Generating the Password Key

Now that we have all the artifacts of our secret password key, let's build the key. In Listing 6-3, we have a method to create the key, `makeSessionSecretDESKey()`. The first step in this method checks to see if the