## Audit Trail Logs for the Sensitive View

Let's find the audit log entries for this access. HR user does not have access to read the audit logs (if he tries, he will be audited), but if you will connect as secadm and execute the following commands, you will see the audit logs that were generated by HR querying v employees public:

```
EXEC sys.p check secadm access;

SELECT OBJECT NAME, STATEMENT TYPE, RETURNCODE FROM DBA COMMON AUDIT TRAIL
        WHERE DB USER='HR'
        ORDER BY EXTENDED TIMESTAMP DESC;
```

While we are still here as secadm, we should try to access the sensitive view and see the audit log entries for this failed attempt (secadm cannot read the sensitive view). You will also see logs of the audit trail SELECT statements executed by secadm.

```
SELECT OBJECT NAME, STATEMENT TYPE, RETURNCODE FROM DBA COMMON AUDIT TRAIL
        WHERE DB USER='SECADM'
        ORDER BY EXTENDED TIMESTAMP DESC;
```

A RETURNCODE of 0 is a success, while a non-zero RETURNCODE indicates a failure.

## Regarding Synonyms

Synonyms are like extra names for structures in Oracle, primarily for tables and views. The most persuasive argument for synonyms is the use of public synonyms. If we create a public synonym for a view, then anyone selecting from that view need not prefix the schema name on the view name. HR user can do the following to create a public synonym. (Do not do this—this example is for discussion only.)

```
CREATE PUBLIC SYNONYM v employees public FOR hr.v employees public;
```

This does not change the security or accessibility of the view or data. However, it permits the role hrview role to select the data without the hr. schema name prefix, as in:

```
SELECT * FROM v employees public;
```

It is also possible to give a name to the synonym that is different from the name of the structure it represents, like the following:

```
CREATE PUBLIC SYNONYM employees FOR hr.v employees public;
```

Perhaps this can simplify the name—and if we expected our users to type all their queries at the command prompt, this might be helpful. (Do not do this—this example is for discussion only.) You can imagine there might also be some confusion for the successor database administrator coming along behind the DBA who set this up. An employee might complain that they cannot select from EMPLOYEES, and the successor administrator might reply that no one can, not knowing that the employee was talking about the synonym, not the table or view. Also, if ever the view hr.v employees public is dropped (deleted), the synonym will be broken.

An individual user might create a private synonym for her own use (use by only that specific user/schema). She might do this so she can call HR.EMPLOYEES by her favorite term, "PEERS." However, when she tries to share any database queries or code with her associates, they will be frustrated by her private synonyms. Queries using her private synonyms won't work for others.

I have also observed attempts to incorporate synonyms into a security strategy, hiding the original table with a public or private synonym by the same name. It would be foolish to trust in such a ruse.

Probably the best reason for synonyms is to make it slightly easier for programmers to write code, by not having to remember to prefix the table and view names with those pesky schema names. However, I have observed application failures when a table or view is migrated from a development environment (Oracle *instance*) to production, but the public synonym is left behind. I know this should never happen, but it does. Then the data is there in production, but the application fails, because it is missing the synonym or merely missing the schema name in the code. That can be very difficult to troubleshoot.

I recommend that you rarely if ever use synonyms, and that you just make the developers address the tables and views where they reside, with the schema name prefix. I don't really expect everyone to follow my advice, but please at least consider it.

One further argument for synonyms is to be a reference to a structure on another database instance. In this case, the synonym can also include a database link. I can't really argue against that use of synonyms, because it is certainly easier to specify and update if the database link name changes. I do caution that you need to protect data wherever it appears, even across database links. There should also be restrictions on the visibility and use of production data in non-production environments; and even more so, vice-versa.

## Chapter Review

Now you have a security administrator user who is not a DBA, but who can handle most of the Oracle security tasks required. Also, you have an application security user who will be handling the transactions we require for Oracle application security.

You have locked down our Oracle database and added some basic auditing, and learned about the data dictionary views. We discussed roles: predefined roles, default roles, non-default roles, and secure application roles. In our first foray into application security, we built a couple of secure application roles and the procedures that protect them.

In the application data schema, `HR`, we distinguished between sensitive and non-sensitive data and created a public view of only the non-sensitive data. We also created a view of both the sensitive and non-sensitive data, but we haven't permitted anyone to see that yet. Then we viewed the audit trail logs for entries related to our sensitive view of `EMPLOYEES`.

CHAPTER 3

# Secure Java Development Concepts

This chapter goes beyond what would traditionally be covered in a chapter on Java security, and it does not cover strictly Java security topics in depth. Rather, it addresses fundamental Java development concepts. We should make sure that we are on a solid foundation in Java development. You can develop very secure code in Java, but if you do not realize what you are doing, your code can be unwittingly insecure.

This chapter will also help Java Integrated Development Environment (IDE) programmers better understand the development process outside of the tools (such as JDeveloper, NetBeans, or Eclipse). IDEs do a pretty good job, but do not blindly trust one to understand and enforce secure development practices for you. If you have a strong background in Java, this chapter may be unnecessary—in that case, just skim it to see if there are any areas you need to review.

## Java Development Kit

The Java Standard Edition (SE) Java Development Kit (JDK) is available both stand-alone and bundled with Sun's (Oracle's) IDE, NetBeans. You can download either one from Oracle's web site at `java.sun.com` (`www.oracle.com/technetwork/java`). You will need JDK 1.5 or later, and you will find that earlier versions, such as 1.5, may only be available on the Previous Releases web page.

We need JDK 1.5 or later because that is the version of the JDK that has been included in the Oracle database: the Oracle Java Virtual Machine. We will discuss virtual machines shortly. We want to have at least that same revision level of Java on our workstation as on the Oracle database.

You can also download the Java Runtime Environment (JRE.) Because you are downloading the JDK, you do not need to also download the JRE. A JRE is that portion of the JDK required to run Java applications, but it does not have the tools required to compile Java.

If you have a Java IDE installed, then you also have the JDK included with your IDE. If that is version 1.5 or later, you can simply use the JDK that came with your IDE.

## Oracle Java Database Connectivity

Download a copy of the Oracle Java Database Connectivity (JDBC) code library from Oracle's web site at *www.oracle.com/technetwork/indexes/downloads*. Scroll down to [Drivers]. The file you download can be for the latest release of Oracle database (11g) and must be suitable for JDK 1.5 or later. Note, however, that you should not use JDBC drivers that are later than the JDK you are using (e.g., do not use *ojdbc6.jar* with JDK 1.5). So you will download *ojdbc5.jar*, or *ojdbc6.jar* if you are using JDK 1.6 or later.

## JAR File Directory Separator

A Java Archive (JAR) file (like the `ojdbc6.jar` file that we downloaded previously) is a compressed directory tree with compiled Java files and other files for use as a set. You can, and later we will, create a JAR file with the JAR utility in the JDK. You can look at the contents of a JAR file with any ZIP utility.

In a JAR file, you would find a directory tree. For example, inside the `ojdbc6.jar` file you would see these directories (and many others):

```
/oracle
/oracle/sql
```

In the `/oracle/sql` directory, you would see files like `ARRAY.class`. The `ARRAY.class` file is a compiled Java file.

Do not be disturbed by my use of the / (slash) for a directory separator character. This is the standard separator character for UNIX, and it is the default for Java. Java will understand slash, but the standard Microsoft backslash must be escaped (explained) to Java, in most situations. The backslash serves another purpose in Java as the escape character itself, so an escaped backslash looks like this `\\`. The following two filenames are equivalent in Java on a Windows box, but I will always use the first style.

```
"C:/Program Files/Internet Explorer/SIGNUP/install.ins"
"C:\\Program Files\\Internet Explorer\\SIGNUP\\install.ins”
```

## Java Packages

We don’t have the Java code of the Oracle JDBC drivers, but we can deduce that the code of the `ARRAY` class (mentioned previously) was in a file named `ARRAY.java`. And in that file was a package statement:

```
package oracle.sql;
```

Notice how the package name correlates with the directory tree, using a dot (.) separator character for the package, instead of a slash for the directory. At Oracle, the corporate developers keep the file `ARRAY.java` in a directory named `sql,` which is in a directory named `oracle`. They compile the class in a matching directory tree. And they create the JAR file by collecting all the compiled content, starting with the `oracle` directory.

You need to keep this fundamental concept in mind: packages equal directory paths. Packages also provide security and affect how we reference Java code and how we compile and run it, as we shall see.

## Development at Command Prompt

I have coached many developers, new and experienced, who have primarily accomplished their Java development efforts or education with an IDE. When I ask them to do any troubleshooting or test their application from the command prompt, often they don’t know where to begin. I believe you need to be prepared to execute (run) Java from the command prompt, and we will be doing some of that here.

### Environment

When you get to the command prompt and you want to compile or run your code, you will need a couple of operating system environment settings: your `PATH` to find the JDK executables, and your `CLASSPATH` to find your code and the Oracle JDBC code. From wherever the recent JDK executables exist, they are able to find their own libraries of code (Java can find Java libraries, although it hasn’t always been that way), so you don’t have to specify them in the `CLASSPATH`.

Let's imagine that you have installed the JDK in this directory:

```
C:\Program Files\java\jdk1.6
```

Imagine you also placed the Oracle JDBC file at this location:

```
C:\Program Files\java\ojdbc6.jar
```

You do not need to set the environment variables, but the difference in Java command line syntax is significant. Following are two examples of running the Java Oracle application `MyApp` from the command prompt. In the first command, we have set and rely on our environment settings. The second would be required if we do not have our environment set.

```
java MyApp
C:/Program Files/java/jdk1.6/bin/java -cp ".;C:\Program Files\java\ojdbc6.jar" MyApp
```

You can readily see how much easier it is to run from the command line if your environment is properly configured. If you're running Windows XP or later, then right click on **My Computer** on the desktop and choose Properties, or right-click on Computer in your Start menu and choose Properties. Select the Advanced tab or button and click on Environment Variables. At the top, create two (new) user variables (or modify existing) with the settings listed in Table 3-1.

***Table 3-1.*** *Environment Variables*

| Variable Name | Variable Value |
|---|---|
| PATH | C:\Program Files\java\jdk1.6\bin; |
| CLASSPATH | .;C:\Program Files\java\ojdbc6.jar;%CLASSPATH% |

The PATH user environment variable will be automatically appended to the System PATH, and will tell Windows how to find the Java compiler (javac.exe) and Java runtime (java.exe) executables.

Beware that even after setting your path, there might be other java.exe executables earlier in the PATH. Assure that the versions of javac.exe and java.exe are both 1.5 or later. Open the command prompt from your Start menu and check the results of these commands:

```
javac -version
java -version
```

In the `CLASSPATH` user environment variable, we are saying several things. First of all the dot (.) tells Java to find classes in the current directory. Trust me, you do not want to be caught asking Java, "Why can't you find the file—it's right here?" The second thing we point to is the Oracle code in the ojdbc6.jar file—we want Java to automatically find that code. And the third thing we are saying is that we want whatever system environment `CLASSPATH` already exists (%CLASSPATH%) appended to our `CLASSPATH`.

Just to elaborate a bit, the `CLASSPATH` is a list of places that we want Java to look for compiled classes. These places are really just starting points, and can include directories (like our "dot" current directory) and archive files (JAR files or ZIP files). When we compile or run our own code and need to use existing code, we can find it along the `CLASSPATH`. Java opens archive files and looks at the included directories and files to see if what we need is there. If I refer to `oracle.sql.ARRAY` in my code, Java will eventually open the ojdbc6.jar file from the `CLASSPATH` and find the `oracle` directory. It will then continue to find the `sql` directory, and from there, the `ARRAY.class` file.

## Beginning Java Syntax

Let's put a couple Java code files on display here so that we can talk about some specific programming concerns with concrete references. For discussion only, we will have two .java files exist in a directory named `mypkg`, and `mypkg` is in a directory named `javadev`. The directory structure and filenames look like this:

```
javadev/
        mypkg/
              MyApp.java
              MyRef.java
```

---

■ **Note** You will find the files in the `javadev/mypkg` directory.

---

Consider the following code listings for our two Java files, shown in Listings 3-1 and 3-2. Note that these files don't do anything, but they are valid Java code.

***Listing 3-1.*** *`MyApp.java`*

```
package mypkg;
import oracle.sql.ARRAY;
public class MyApp {
    ARRAY myArray;
    MyRef myRef;
}
```

***Listing 3-2.*** *`MyRef.java`*

```
package mypkg;
public class MyRef {
}
```

We will cover several aspects of the syntax of Java code here, but we will be introducing the bulk of the syntax a bit later. Notice that both of the Java code files begin with the declaration of their package. Looking at `MyRef.java`, we see the simple declaration of the class with the modifier `public`. The code for a class is enclosed in a pair of curly brackets ({}). Each .java file must have only one top-level class declaration, and the top-level class must be named identical to the file (in this case, MyRef).

---

■ **Note** Often these top-level classes are declared `public`, but they don't have to be. They cannot be private or protected, but they can be default, or package accessible with no modifier.

---

Another observation at this point is in the `MyApp.java` file. First, we have an `import` statement, `import oracle.sql.ARRAY`. A statement like this is required every time you use a class from a different package. If

we were using lots of classes from the `oracle.sql package`, we could import them all with the shorthand statement `import oracle.sql.*`. Because `MyRef` is in the same package as `MyApp`, we do not need to import `MyRef`, even though we do refer to it in `MyApp`.

Also notice that, inside the class definition (between the curly brackets) of `MyApp`, we have declared two member classes: one instance (copy) of `ARRAY` that we will refer to as `myArray`, and one instance of `MyRef`, which we will refer to as `myRef`. Notice that Java is case sensitive: compare `myRef` with `MyRef`.

I will be using the terms *member* and *instance* frequently. Briefly, in object-oriented parlance, an object that is created in computer memory is an instance of whatever kind of class it is; we call the creation of objects instantiation. The object that is created by another object and referenced therein is called a member of the object that created it.

# Byte Code Compilation and the Java Virtual Machine

I have already used the word compile, and you will see it a lot in this book. In Java, you will write human-readable code and place your code in files with a .java extension on the filename. In order to run the code, it first needs to be compiled. The compile step creates a file that humans cannot read with a .class extension. This .class file cannot be executed by the computer operating system; it is in byte-code format. Class files are run by the Java Runtime Environment (JRE) executable, java.exe.

The Java runtime creates a Java Virtual Machine (JVM) in computer memory that can interpret and run the byte code. The value of having a JVM read and run the byte code is that, in most cases, the byte code can be written once, run anywhere–a fundamental goal of the Java language. You write it and compile it wherever it is convenient for you, the developer, and then place it on any computer with a compatible JVM to be run: in a workstation, server, browser, cell phone, or web server. The JVM handles all the specifics of talking to the operating system and to the hardware.

We will see the power of this concept in our code, which we will write, compile, and run on our workstation. We will then load it onto an Oracle database (actually, we will store it in the database), and have the Oracle JVM run it also in Oracle.

## Using the Java Compiler

The JDK has a number of command-line utilities in the bin subdirectory. One of these is the primary Java compiler, javac.exe. Another is the primary Java executable for running applications, java.exe.

To compile Java code, you execute javac.exe, passing the name of the Java code file as a parameter, like this (assuming your command prompt is in the directory `javadev/mypkg`):

```
javac MyApp.java
```

You must include the .java extension on the code file name. This command will find the file `MyApp.java` in the current directory and, if successful, it will place a compiled Java file named `MyApp.class` in the current directory. This is true whether or not `MyApp` is in a package. If `MyApp.java` refers to other compiled classes that are not in the current directory, then they must be found along the `CLASSPATH` (for example, the compiler can find `oracle.sql.ARRAY` when `ojdbc6.jar` is listed in the `CLASSPATH`).

If there are other Java code files referred to by `MyApp` that can be found along the `CLASSPATH`, and the code has not been compiled or has been updated since the last compilation, then `javac.exe` will compile those classes also. The referenced code will be compiled and the compiled class file will be placed in the same directory with the referenced Java code file. For example, `MyRef.class` will be placed in the same directory as `MyRef.java`.

Additionally, you can compile multiple Java classes or all in a specific directory by specifying a wildcard, like this:

```
javac *.java
javac mypkg/*.java
```

## Finding Referenced Code/Classes

There is a "gotcha" with compiling Java that you need to understand and anticipate, or at least quickly recognize. When your code refers to other Java code that is not on your `CLASSPATH`, the Java compiler will not find the code, even if the referenced code is in the current directory. Let's imagine your code is in package (directory) named `mypkg` and that your `MyApp.java` refers to `MyRef.java` in the same package. If your `CLASSPATH` does not refer to the parent directory of the `mypkg` subdirectory, then *javac.exe* will not find the `MyRef.java` file. Having dot (.) in your `CLASSPATH` will not fix this gotcha, because the `mypkg` directory is not in the current directory, rather we are in `mypkg`. Remember, `CLASSPATH` (even dot) is merely a list of starting places for packages. Looking in the current directory, we cannot find the package (subdirectory) named `mypkg` nor the code or class file named `mypkg/MyRef.class`.

One way to correct that gotcha is to change directories to the parent of `mypkg`. Once there, you can compile your code, and the dot (.) in your `CLASSPATH` will find the referenced code in the `mypkg` directory. In this case, however, you need to tell `javac.exe` that your `MyApp.java` code is in a subdirectory (both the forward and back slashes are acceptable as directory separator characters in this context), like this:

```
javac mypkg/MyApp.java
```

That is a valid approach, but probably the best way to address this issue is to have a development directory (like `javadev`) and to place all your package directories underneath that directory. Then add your development directory to your `CLASSPATH`.

```
set CLASSPATH = C:\javadev;%CLASSPATH%
```

Often, especially if you manage your development using an IDE, you will not be able to put your packages in a single development directory. Each project will store files in a separate directory.

Perhaps now you are thinking, "Man, I'm just going to let the IDE handle all this for me!" I don't want to disparage IDEs, but being satisfied to take that approach is like voluntarily going to jail because work is hard. Don't quit working hard! Often you don't need an IDE to write, compile, and run your code, so take those opportunities to work at the command prompt. Once you set your environment variables, it is just a matter of knowing about packages and `CLASSPATH` and recognizing problems.

## Running Compiled Code

Once your code is compiled, running (executing) it can be easy:

```
java mypkg.MyApp
```

Please observe that to run a Java class, you need to specify the package it is in. There is no shortcut around this.

The same requirements for finding all classes needed for your application along the `CLASSPATH` apply for execution as they do for compiling. In order to run your Java code as just shown, `java.exe` needs to be able to find the `mypkg` package in one of the starting points listed in `CLASSPATH`. In our example, if you are in the `javadev` directory and you have the "dot" in your `CLASSPATH`, then `java.exe` can find `mypkg`, and can run your code in `mypkg/MyApp.class`. This must be true for all classes that your code refers to, like `oracle.sql.ARRAY`. For that reference, `java.exe` needs to find `ojdbc6.jar` listed in `CLASSPATH`.

There is an alternative to having `CLASSPATH` defined in your Operating System environment. You can pass `CLASSPATH` to `java.exe`, or `javac.exe` for that matter, using the `-cp` parameter.

```
java -cp ".;C:\Program Files\java\ojdbc6.jar" mypkg.MyApp
```

In this example, we alert `java.exe` that the `CLASSPATH` parameter (`-cp`) is provided. The quotation marks are required, because there is a space in the directory name Program Files. Note that forward slashes would also work in this context.

## Java Code and Syntax Concepts

I'm going to give a brief example of Java code. It is 20 lines long and not too complex, but it will introduce you to many Java syntax concepts. You do not need to memorize the details of these concepts—they will present themselves in every program you write in Java.

Throughout the following discussion, we are going to consider some aspects of the code in Listing 3-3. Keep a bookmark at this page for reference as you read the next few sections.

***Listing 3-3.*** *`MyApp2.java`*

```
package pkg2;
import oracle.sql.ARRAY;
import mypkg.MyRef;
public class MyApp2 {
    private ARRAY myArray = null;
    static MyRef myRef;
    public static void main( String[] args ) {
        MyApp2 m = new MyApp2();
        MyRef useRef = new MyRef();
        m.setRef( useRef );
        ARRAY mA = m.getArray();
        myRef = new MyRef();
    }
    public ARRAY getArray() {
        return myArray;
    }
    void setRef( MyRef useRef ) {
        myRef = useRef;
    }
}
```

---

■ **Note** Find this code in the file named *javadev/pkg2/MyApp2.java*

---

Continuing our previous discussions about Java syntax, we see that this new code, *MyApp2.java*, is in a new package (directory), *pkg2*. For that reason, we have to import (refer to) the *MyRef* code in our original package. Recall that both of those packages need to be found from some starting point in your `CLASSPATH` in order to compile and run this code.

# Methods

In the code for `MyApp2` (`MyApp2.java`) you will see three blocks of code named `main()`, `getArray()`, and `setRef()`. Find them by looking for open and close curly brackets within the curly brackets that define the body of the class `MyApp2`. These three blocks of code are called methods (programmers may think of them like subroutines or functions).

Each standard method requires two things: a set of parentheses where parameters (input values or objects) may be passed to the method, and a declaration of the return type (being handed back to whatever code called the method). When I mention methods in this text, I will append the open and close parentheses. Later, I will discuss in more detail the `public` and `private` modifiers. For now, know that they are not return types.

Look at the middle method first. `getArray()` has an empty set of parentheses that indicates it doesn't take any input parameters. Before the name of the method, it has the word `ARRAY`, which shows that it returns an object of type `ARRAY`.

Notice that both the `main()` and `setRef()` methods have a return type of `void`, which is a way to say that those methods do not return anything.

The `main()` method takes a single parameter named `args`, and the type of that parameter is an array of `Strings`. *String* is the term for a series of characters, like the word "tremendous," or the sentence, "This is good!" An array of any type is indicated by open and close square brackets ([ ]). Any single element in the array can be referenced by placing an integer in the square brackets to indicate its place, its index. For example, `args[0]` is the first element of the array of `Strings` passed to the `main()` method. When defining an array, you can put the square brackets on either the type or the name; the following are identical:

```
String[] sAr;
String sAr[];
```

There is another kind of method that is different from the standard methods—it is called a constructor. Constructors are called when an instance of a class is created in computer memory. Constructors do not have a return type, and the name of the constructor method is the same as the name of the class. For example, a constructor for `MyApp2` might look like this:

```
public MyApp2() {
}
```

Notice that there is no return type indicated. If you do not define a constructor in your code, your class will use the default constructor, which would look very much like this example constructor. Also compare this default constructor with the `MyApp2` class definition; they are very different things, but bear some resemblance: `public MyApp2() {` compared with `public class MyApp2 {`.

# Values

In our example code for `MyApp2`, shown in Listing 3-3, we have statements where we set a member variable equal to a value. For example, we said the following:

```
ARRAY mA = m.getArray();
```

In describing this statement, I am as likely to say, "set `mA` equal to `m.getArray()`" as I am to say, "get `m.getArray()` into `mA`." But what I should be saying, and what you need to understand, is that the member variable `mA` is a pointer to an object of type `ARRAY`. What I am doing in this statement is setting the value of the pointer to the address in memory of the object that is returned by `m.getArray()`. Even this is not precise. It is more precise to say that `m.getArray()` is returning a pointer to the location of an `ARRAY`, and I am setting the pointer `mA` to that same location.

This matters, because you need to remember that you are never replicating an object when you pass it around and set member variables equal to it, you are merely setting more pointers to the same location. The only time an object is created is when you instantiate it by a call to `new` or some method that generates a new object and returns it. We will also look at the `clone` method, which can be used to return the address of a new copy of an object instead of returning the address of the original object.

The following examples might help. In the first, we just point another member at the existing object; and in the second, a new object is created and we point the new member variable at it. When we set the name member in the first example, we set the value in the originally existing object; and any other code that points at the original object will see the changed name. In the second example, the change, so far, is only seen locally.

```
ExampleObject newMember = existingObject;
newMember.name = “New Name”;

ExampleObject newMember = new ExampleObject();
newMember.name = “Another Name”;
```

Something different happens with primitive values. They are not Java objects and their value, rather than their address, is passed in all references.

# Members

Objects have both methods and members. Members in object-oriented programming are variables. By variable, I mean a pointer. This variable might point to a primitive—like an `int` (integer)—or a Java object (like a `Date` class). In Java, `Strings` are objects, not primitives. Plain old arrays are also objects (with no methods).

Often, both methods and variables are referred to as “members” of a class, but I prefer to say “methods and member variables. In this discussion, methods are not called *members*. And also, member variables are correctly referred to as *fields*. I can’t help throwing that in once in a while. If I used the terms *field*s and *methods* I wouldn’t need to say *members*; but I like the word *members* and am in the habit of saying it, so I will say *members* and *methods*.

In our sample code, `MyApp2` has several members. The first two we see are named `myArray` and `myRef`. Those are both class member variables, because they exist in the class, outside of any methods, which you can see from the code (repeated in the following). Methods know about the class member variables as you can see in the `getArray()` method, which returns the class member `myArray`. You can also see that the `setRef()` method sets the value of the `myRef` class member to be equal to the parameter that is passed to the method, `useRef`.

```
    private ARRAY myArray = null;
    static MyRef myRef;
    public ARRAY getArray() {
        return myArray;
    }
    void setRef( MyRef useRef ) {
        myRef = useRef;
    }
```

If you look for a minute at the `main()` method (repeated in the following), you will see that there are three members declared: `m`, `useRef` and `mA`. These are method members, because they exist solely in the method. Note that they can be handed around to be used elsewhere. For instance, the `main()` method hands `useRef` to the method `m.setRef()`, which then sets the `myRef` class member (remember, we are

handing and setting memory pointers or references). The `main()` method is unique, and I will discuss it more when I cover the `static` modifier, a little later.

```
public static void main( String[] args ) {
    MyApp2 m = new MyApp2();
    MyRef useRef = new MyRef();
    m.setRef( useRef );
    ARRAY mA = m.getArray();
    myRef = new MyRef();
}
```

## Objects

The words *object*, *class*, and *instance* are practically interchangeable. You have classes (theoretically) before you create any instances. An instance only exists when you instantiate a new one in memory. You have objects that you can move around and store even if you don't know what type they are. Objects can be placed on disk or sent across the network, even when they don't exist as instances in memory.

So, you create an instance in memory of a class type, which is an object that can be moved out of memory (stored on disk), at which point it is not an instance, but can still exist as an object.

## Classes and Null

I have already used the term *instance* in our discussion. In Listing 3-3, there are three instances of classes being created. To pick out the instances, look for the word *new*. Whenever you see the statement `new`, an object is being instantiated.

Sometimes you will call other classes in order to get an instance of some type; however, even though you don't use the word `new` in your code, that word was used somewhere down the line to get the instance. In the example code for `MyApp2`, we see the `main()` method setting the member `mA` equal to an `ARRAY` that is returned from the method `getArray()`. You may notice that the `getArray()` method in `MyApp2` does not use the `new` statement, and when we define `myArray` at the class level (top), we do not say `new`. Instead, we used the word `null` in the definition. When the method `getArray()` returns `myArray`, it is returning a `null` instead of an instance of `ARRAY`. And when the `main()` method set `mA` equal to what's returned from `getArray()`, it is setting `mA` equal to `null`. This is not very productive, but it is valid Java code.

To explain `null`, we will consider what the following statements mean:

```
MyRef useRef = new MyRef();
new MyRef();
ARRAY myArray;
ARRAY myArray = null;
```

The first line instantiates a new instance of `MyRef` class and assigns a member reference (name) to it. The name we will use is `useRef`.

We can instantiate a new instance without assigning it to a name as in the second line. This is often done in graphical user interface (GUI) apps when they get started. The initial `Frame` is instantiated and displays itself on screen—no name is required.

We can also create a new name for a certain type of class and not assign it to any instance, as in the third line. The fourth line is identical in effect to the third line except that in the fourth line we are specifically telling Java that we are assigning the member name `myArray` to a non-existent instance, hence `null`. In effect, `myArray` points at *no* memory location.

We say that member variable names point to instances. At any time, we can point a member variable name at another instance. In hardware, this actually works out to pointing the member at an address in memory. When we point the member at `null`, we are saying that it doesn't point to any address in memory.

## Garbage Collection

A Java application in computer memory can be pictured as a basket full of Java objects, all referencing one another, calling each other's methods and returning data, and getting and setting one-another's members. In Java, we do not have to manually keep track of what objects are in use and release the memory of unused objects, the Java runtime does that through a process called garbage collection. When an object is no longer referenced by any other object in the basket, it falls out of the basket. It doesn't disappear until the periodic garbage collection process walks through memory and sweeps the fallen objects away.

## Primitives

In addition to objects (described previously), we have Java primitives. Some of the primitive types are `int` (integer), `byte`, and `char` (character). Member variables can be of types from both the Java classes (objects) and the primitives (values).

Primitives do not have any methods and do not have any members. You can have arrays of primitives, just like arrays of Java objects.

There are also Java classes that encapsulate the primitives, letting us hold primitives in an object instance and retrieve primitives from an instance. For example, `Integer`, is a class that can encapsulate an `int`. From `Integer`, we can get the `int` value of a `String` by calling `Integer.parseInt( String s )`.

Some Java constructs can only handle objects, not primitives; for example, a `Vector` (a dynamic array object), so we will need to encapsulate our primitive values if we intend to store them in a `Vector`.

## Strings

A `String` is an immutable Java object, meaning you can't change it; however, you can replace it with (point your member variable at) a `new String`. `String` has methods that look like you are changing the object, but that is not the case. For example, you might call the `String.toUpperCase()` method:

```
myString.toUpperCase();
```

This invocation of `toUpperCase()` does not change the value of `myString` at all, but it has a return type of `String`, and what gets returned is the value of `myString` in uppercase. The only way to set `myString` to uppercase is to set it equal to the return value. In effect, you are setting `myString` to point at a new uppercase `String` that is created by the `toUpperCase()` method:

```
myString = myString.toUpperCase();
```

A very common error when handling `Strings` is to try to compare them like you can do with the primitive types, using equal signs for instance:

```
if( stringOne == stringTwo )
```

This is frighteningly valid code, but does not accomplish the test you intend. It is as bad as accidentally using one equals sign in a test like this:

```
if( int1 = int2 )
```

I imagine this intended to compare `int1` with `int2`, but it ended up setting the value of `int1` equal to the value of `int2`. Bad, very bad—too bad.

Because `String` is an object, you need to treat it like an object and call its methods for comparisons. We call `String.equals()` method to compare `Strings`.

```
if( stringOne.equals( stringTwo ) )
```

This requirement for using methods of objects to do comparisons applies to all object types. However, it is most common to run into coding errors in this regard when dealing with `Strings`: they probably appear to programmers to be more like the primitives.

## Static Modifier and the main() Method

You have seen the modifier `static` already. It was in our code example in the definition of the `main()` method, and in the definition of the `myRef` member. Both members and methods can be static. To describe what static means, let me step back for a second.

Imagine you executed `java.exe` and are running some Java code. Picture the JVM holding a bunch of objects (instances). If our code created 100 instances of the `MyApp` class, each one would have certain aspects of our `MyApp.java` code; however, some aspects would only exist one time in the JVM. Those things that only exist one time per JVM are labeled `static`. Items that are `static` exist outside any instance of `MyApp` and are shared by all. An interesting phenomenon that results from existence outside an instance of a class is that those aspects of the class that are `static` are available even when you haven't created an instance.

For example, the first line in my code could be this:

```
if( MyApp2.myRef == null ) {
```

In that line, I do not have an instance of `MyApp2` that I'm referring to by name; rather, I am referring to the class itself and testing the value of the static class member variable, `myRef`.

Now, this is precisely what we need to start running some Java code. By default, if I execute `java.exe` and give it a class name to run, it will run the `main()` method. If I provide any arguments on the command line when I execute `java.exe`, these arguments will be passed to the `main()` method as parameters: an array of `Strings`. For instance, if I give this command:

```
java MyApp2 this is a test
```

Then there are four arguments passed to the `main()` method of `MyApp2`: this, is, a, and test.

Here is the definition of `main()` from our code:

```
public static void main( String[] args ) {
```

After I describe the modifier `public` in the next section, you will know the meaning of that entire definition. This specific syntax for the `main()` method is identical for any class that you want to run from the command prompt (except that you can call the array of `Strings`, the arguments, anything you want, not just `args`).

We need to have a static method to start running Java code, because at that point we have not instantiated any classes. Look again at the `main()` method in Listing 3-3 (repeated in the following). We do something there that is quit typical. In the `main()` method, we instantiate an instance of the class itself—we create a `new MyApp2` instance, named `m`. To use the non-static methods of `MyApp2`, we need to call them within our instance of `MyApp2`, so we call `m.setRef()` and `m.getArray()`. The m. prefix indicates that we are calling the methods of our instance of `MyApp2` named `m`.

```
    public static void main( String[] args ) {
```

```
    MyApp2 m = new MyApp2();

    MyRef useRef = new MyRef();
    m.setRef( useRef );
    ARRAY mA = m.getArray();
    myRef = new MyRef();
}
```

Notice one last thing in the `main()` method definition. It sets the value of the static `myRef` member to an instance of `MyRef`. From a static method, only static members of the class can be referenced. If we tried to set the value of `myArray` from within `main()`, we would get an error at compile time, because until we create an instance of `MyApp2`, there is no `myArray` member—it is not `static`.

# Public and Private Modifiers

`public` and `private` modifiers can exist on both methods and member variables. These help delineate the items in a class that the user of a class has direct access to, from those items that the user of a class cannot see (directly). These are scope modifiers. That is the important thing about `public` and `private`: it is important because it has some implications for security that we will discuss later.

In Listing 3-3, the `myArray` member is private, so someone using this class cannot get or modify `myArray` directly. However, we have provided the method `getArray()` which is public and can return `myArray` to anyone.

```
private ARRAY myArray = null;
static MyRef myRef;
public ARRAY getArray() {
    return myArray;
}
void setRef( MyRef useRef ) {
    myRef = useRef;
}
```

By default, methods and members, like the `myRef` member and the `setRef()` method, which are not declared as `private` or `public`, are generally only accessible to other classes that come from the same package. One additional scope modifier is `protected`. Protected scope is like the default, except it allows you to permit subclasses of your class that are defined outside your package to see your members and methods.

One design pattern (approach) declares all the class member variables are `private`, then sets about establishing getters and setters (public methods that provide a way for other objects to get or set the private members) to read from and write to the members. This is a feature of JavaBeans, and often of Enterprise Java Beans, and lends to their usefulness as components in an IDE. They can be distributed and incorporated into an IDE or application server without prior knowledge or definition by virtue of their getters and setters—by reflection (or XML definition), the IDE can list the public getter and setter methods of a bean and provide access to the private members.

We have provided a getter method in our example code for the `myArray` member: `getArray()`. We have also provided a setter method for the `myRef` member: `setRef()`; although, our method names don't conform to JavaBeans standards, because we are not writing JavaBeans, and we needn't implement all that boilerplate code.

# Exceptions

Good and secure code must have a plan in place for handling errors. Java has included a framework on which we can build error (exception) handling. You have several choices when dealing with exceptions, and Java helps you accomplish any of the following that you choose:

- Do nothing with the exception, but *throw* it to the code that called the code with the exception. Throwing exceptions generates a stack trace, which lists the place (line number and code) where the exception occurred and the place that called this block of code (method) and the place that called that method, and so on. You can print the stack trace and see where you were and what happened. Often, the exception itself will tell you why the exception happened; no further troubleshooting required.
- Catch the exception and deal with it locally, and then either continue running or break out of the code, or even throw the exception after catching it.
- Catch the exception and throw a different exception.

## Exception Handling Syntax

Let's explore some of the syntax and functionality of exception handling. Listing 3-4, `ExceptionDemoA.java`, shows two blocks inside the `main()` method: `try` and `catch`, each surrounded by curly brackets. They go together, so I have concatenated the `catch` block to the close curly bracket of the `try` block. In the second block, the `catch`, we catch the general `Exception`.

---

■ **Note** You can find this Java file and the next file in the folder named *Chapter3*.

---

*Listing 3-4. ExceptionDemoA.java*

```
import java.io.FileInputStream;
public class ExceptionDemoA {
    public static void main( String[] args ) {
        try {
            ExceptionDemoA m = new ExceptionDemoA();
            String mS = m.doWork();
            System.out.println( mS );
        } catch( Exception x ) {
            System.out.println( x.toString() );
        }
        System.exit(0);
    }
    String doWork() throws Exception {
        FileInputStream tempFileIS = new FileInputStream( "C:/Windows/win.ini" );
        tempFileIS.open();
        //…
        if( tempFileIS != null ) tempFileIS.close();
```

```
        return "success";
    }
}
```

Notice that the `doWork()` method is declared with `throws Exception`. The `Exception` class is the mother of all exceptions and represents any of them. Some of her children exceptions are `IOException` when dealing with files or the network, `SQLException` when dealing with databases and `NullPointerException` when your Java code tries to call a method of a nonexistent object. We can have a separate `catch` block for each kind of exception that we are anticipating. A method can throw multiple types of exceptions that may occur.

Catching the general `Exception` would be appropriate whenever we are calling an external block of code (in this case, our call to the `m.doWork()` method) that may throw any number of exceptions. It would also be appropriate if, no matter what exception we catch, we handle it identically, as in this case.

## Print the Exception Message to System Output Stream

Whatever the exception is that we catch, we will print the identity in the catch block using this command:

```
System.out.println( x.toString() );
```

Every object in Java has a `toString()` method, including exception objects and the objects that you will create. The `toString()` method is inherited from the `Object` object – more on that later. The `toString()` method returns a `String` type, which can be used to provide the identity or some details about the object—in the case of exceptions, it provides the exception name.

`System.out.println()` sends text to the command prompt window, ending with a line ending character. Line ending characters differ from one kind of computer to another, for instance, from Windows to UNIX. Lines end with a carriage return character or line feed character or both.

These terms come from the days of typewriters where the paper rolled around a platen in a carriage and the carriage moved from right to left as you typed. At the end of each line, the typist would swing a lever that rolled the platen to feed a line, and push the lever to the left, which would return the carriage to the start.

Appending a line ending character to a string means the next character will appear underneath the first character on the current line, i.e., on the next line.

### STACKED CALLS

Note that we call `x.toString()` in the `catch` block and pass the returned `String` immediately as a parameter to `System.out.println()`. We don't create a `String` member variable to hold the value before printing it. For lack of a better name, I'm going to call this a *stacked call*. See if you can find two very similar examples of such a call in the code in the next section, Listing 3-5, `ExceptionDemoB.java`. We will begin to use stacked calls frequently in Chapter 6.

## Clean Up as part of Exception Handling

There is a problem with the code of `ExceptionDemoA.java` in Listing 3-4. (The code is repeated in the following). There are things that get done in the `doWork()` method that need to get undone there as well.

Primarily, we open a `FileInputStream` (probably to read the contents of a file). From there, anything could happen—specifically, an exception could occur.

```
String doWork() throws Exception {
    FileInputStream tempFileIS = new FileInputStream( "C:/Windows/win.ini" );
    tempFileIS.open();
    //…
    if( tempFileIS != null ) tempFileIS.close();
    return "success";
}
```

If an exception were thrown before we close the `FileInputStream`, then the `doWork()` method would throw the exception to whoever called it (see the `throws` modifier on the line that defines `doWork()`). The line that closes the `FileInputStream` (`close()` method) would never be reached, and in some instances the file would remain open. This could have dire consequences for applications running on the computer. In many cases, a file that is currently open in one application cannot be read by another, so a file that is not closed after an exception might not get backed up, and it might lock up other computer operations.

## Clean Up in a finally Block

With exception handling, there is a third block called `finally` that can be used, as in Listing 3-5. With each `try` block, there must also be a `catch` block, a `finally` block or both. We will rarely have a `try` without a `catch` block (see the sidebar called "`try/finally` Tack-on Debugging and Synchronization" later in this chapter). The idea is that whether you complete the `try` block successfully or generate and catch an exception before finishing the `try`, you can do things to clean up in the `finally`. The `finally` block runs, even if your `catch` block throws an exception. Look at the syntax of the `doWork()` method in Listing 3-5.

*Listing 3-5. `ExceptionDemoB.java`*

```
import java.io.FileInputStream;
public class ExceptionDemoB {
    public static void main( String[] args ) {
        ExceptionDemoB m = new ExceptionDemoB();
        System.out.println( m.doWork() );
        System.exit(0);
    }
    String doWork() {
        String returnString = "attempt";
        FileInputStream tempFileIS = null;

        try {

            tempFileIS = new FileInputStream( "C:/Windows/win.ini" );
            tempFileIS.open();
            //…
            returnString = "success";

        } catch( Exception x ) {

            System.out.println( x.toString() );
```

```
        } finally {
            try {
                if( tempFileIS != null ) tempFileIS.close();
            } catch( Exception y ){}
        }
        return returnString;
    }
}
```

In `doWork()` we have a `try` block around the `open()` of the `FileInputStream`. If anything should happen to cause an exception to be thrown, we will catch it in the next block. Then whether we have caught an exception or not, we will enter the `finally` block and execute the `FileInputStream.close()` method.

You might have a couple questions. The first might be, "Why can't we close the `FiIeInputStream` in both the `try` and in the `catch` blocks?" The answer is you *can* do that. However, repeating code is bad practice, and you may have a problem with the `close()` in the `catch` block that will cause another exception to be handled, thus compounding the troubleshooting effort.

A second question you may have is. "Why don't we just put the `FileInputStream.close()` outside both the `try` and the `catch`, and skip the `finally` block?" That is a bad idea because sometimes you will both catch an exception and throw it or another exception, like this:

```
} catch( IOException x ) {
    throw new AppException( x.toString() );
}
FileInputStream.close();
```

If this were your code, and you caught an `IOException`, you would never reach the `FileInputStream.close()` line. The `throw` statement inside your `catch`, since it is not caught here, would exit the current method, throwing an `AppException` to whatever code called this method.

Looking back up at our example, you will see this set of blocks within the `finally` block:

```
        } finally {
            try {
                if( tempFileIS != null ) tempFileIS.close();
            } catch( Exception y ){}
        }
```

Closing the `FileInputStream` can throw an `IOException`, and we need to deal with it (catch it or throw it). I choose to catch it and do nothing, at this point. I use this as my standard syntax for a `finally` block. Generally, we are just doing cleanup, and whatever work I might have been doing should have been completed, or would have thrown an exception, which I've already handled.

If I can't clean this up (e.g., close the `FileInputStream`), then it is true I may need to fix something, but if I've gotten this far, then I've already used the resource, in which case I should be able to close it. If I failed to use it, then I've already handled a related exception, and another exception here would be superfluous. So I usually do nothing inside the `catch` block that I have inside my `finally` block.

## TRY/FINALLY TACK-ON DEBUGGING AND SYNCHRONIZATION

There is one case where I have found the need for a set of `try` and `finally` blocks, without a `catch` block. If you have written a method and you need to temporarily add debugging to it without over-cluttering your existing code, you can use a `try` and `finally` block pair to your advantage. Look at the following skeleton of a method named `methodName()`:

```
returnType methodName() throws Exception {
    try {
        while() {
            if() {
            }
        }
        for() {
        }
    catch( Exception x ) {
    } finally {
    }
}
```

To do temporary spot debugging on this method, we could declare a file output before the `try` block, then write debug messages to it throughout the method. For completeness, security and good practice, we would then `flush()` the file and `close()` it in the existing `finally` block. The only problem with that is that we have cluttered up our code, and making those changes and then cleaning it up later (removing the debugging) gives an abundance of opportunity to introduce more errors. In brief, that approach would yield code that appears more like permanent debugging code, like this:

```
returnType methodName() throws Exception {
    PrintStream debugOut = null;

    try {
        debugOut = new PrintStream ( new FileOutputStream( "debug.txt" ) );

        while() {
            debugOut.println( "message 1" );
            if() {
            }
        }
        for() {
        }
    catch( Exception x ) {
    } finally {
        if( debugOut != null ) {
            debugOut.flush();

            debugOut.close();

        }
    }
}
```

We can reduce the potential for introducing errors with a simpler `try/finally` block pair. If we are seeing problems that we know are happening in the `while` block in our example method, then we can surround just that block with our troubleshooting. (Be aware that you cannot break existing blocks by putting a new code block around just the open or close of an existing block.) Our solution might look like the following (the code in bold has been added):

```
returnType methodName() throws Exception {
    try {

        // temp code
        PrintStream debugOut = null;

        try {
        debugOut = new PrintStream ( new FileOutputStream( “debug.txt” ) );

        // to here

        while() {
            debugOut.println( “message 1” );
            if() {
                debugOut.println( “message 2” );
            }
            debugOut.println( “message 3” );
        }

        // temp code
        debugOut.println( “message 4” );
        } finally {
            if( debugOut != null ) {
                debugOut.flush();

                debugOut.close();

            }
        }
        // to here

        for() {
        }
    catch( Exception x ) {
    } finally {
    }
}
```

We have successfully added spot debugging, right where needed, and we have used good coding practice by closing our file output in a `finally` block. Note that if our method is not already catching or throwing an `IOException`, we will need a new `catch` block before our new `finally` block, and we will need a set of `try/catch` blocks within our added `finally` block, around the `flush()` and `close()`. Cleaning this up (removing the debugging) only requires removing the opening and closing sections of code (our new `try` and `finally` declarations and the surrounding code we added) and removing each of the `debugOut.println()` statements.

When tacking code on like this, it is good to remember the context and the synchronization demands. If this were a method that is being called by a multithreaded server application, then we would need to assure that only one user at a time will be opening, writing and closing the file. One way to accomplish that would be to declare the method to be *synchronized* (add the `synchronized` keyword to the method signature, before the return type declaration). Another way that doesn't involve changing the existing method signature is to *synchronize* on an object. It can be any type of object, but it should be a static class member so that everyone is synchronizing on the same thing.

Often you will attempt to synchronize the smallest section of code that you can, and in that case you might synchronize each `debugOut.println()` statement. But in that case you might also need to declare your `debugOut PrintStream` as a static class member. However to add synchronization to our spot debugging, we will synchronize a larger section and allow each request of the `methodName()` method to have exclusive access to the debug.txt file from open to close. It will look like this:

```
static String synchOnThis = “”;
returnType methodName() throws Exception {

    try {

        PrintStream debugOut = null;

        synchronized( synchOnThis ) {

        try {
        debugOut = new PrintStream ( new FileOutputStream( “debug.txt” ) );


        while() {
            debugOut.println( “message 1” );
            if() {
                debugOut.println( “message 2” );
            }
            debugOut.println( “message 3” );
        }

        debugOut.println( “message 4” );
        } finally {
            if( debugOut != null ) {
                debugOut.flush();

                debugOut.close();

            }
        } }

        for() {
        }
    catch( Exception x ) {
    } finally {
    }
}
```

Notice that we open a `synchronized` block before our added `try` block, and we close the `synchronized` block (with an extra close curly bracket) after our added `finally` block. Everything within that `synchronized` block will be executed exclusively by one call to the `methodName()` method (by one user) at a time.

We are synchronizing on a blank `String` named `synchOnThis`. It is a `static` class member. Let me point out how we might capitalize on this synchronization arrangement. Let's say we have an additional method in this same class that we need to debug. We can add identical `try`/`finally` blocks and debugging in that second method, again synchronizing on the `synchOnThis` member. Not only will each call to those methods get exclusive access to write to our file, but also a call to either method will have exclusive access—e.g., someone in our first method cannot have access to the file while someone in the second method has `synchronized` on it. If you want to debug a method in a different class to this same file, you can synchronize on the same static member, `synchOnThis` in this class.

There is one assumption in synchronization that you must understand: synchronization is only effective within a single JVM. If you have, for example, two web servers on the same computer, each one will likely be running its own JVM. Those two JVMs cannot coordinate synchronized access to a file (for example). So if one of the JVMs is writing to the file, the other JVM may try to write to the file at the same time. That will likely fail.

# Exception Handling Approaches

There are a variety of philosophies and practices regarding exception handling. I have not seen any proposals that I have entirely agreed with. However, I do have practices that I like, and I do try to be consistent. Allow me to propose a few ideas that help define my approach.

## Don't Code Multiple Exceptions When One Will Do

I find that it is a very rare circumstance that will provoke me to do one thing for one specific exception thrown in a `try` block, and a different thing for another kind of exception in the same block. So I usually only have one `catch` block per `try` block. One "exception" to this rule is when I am throwing generic app exceptions (more on this is a bit).

Additionally, I will frequently have `try` and `catch` blocks inside other `try` and `finally` blocks. You have already seen me do that in the `finally` block listed previously.

## Catch and Handle an Exception Where It Occurs

Deal with the exception immediately. If you're not sure what to do in the current method, it will be even harder to do something from a more remote method. An "exception" to this is when your design is intended to throw exceptions in order to create a complete stack trace.

I'm thankful that the classes provided by Sun and Oracle corporations are designed to throw their exceptions to my code. They are certainly not in a position to handle them like I as the application programmer can. But it's true for you like it's true for me, what my graduate professor once said: "You're the doctor now." It's your job to deal with the exception.

It almost goes without saying that there are many times when the best way to deal with an exception is to ignore it. One way I frequently ignore exceptions is by returning a blank `String`. Let's say, for

example, you are passing a phone number that you selected from Oracle database to a Java method that formats phone numbers with parentheses and dashes (e.g., (800)555-1212). However, the current record has a `null` phone number. You wouldn't want to throw an exception for that trivial concern. You can do one of 2 things (or both):

1. Test if the incoming phone number is `null` and immediately return a blank `String`.

2. Format whatever, and if it throws an exception, return a blank `String`—but don't throw an exception from your phone number format method.

Why wouldn't you just take approach number (1)? Well there might be other problems with formatting a phone number besides a `null` value. What if the phone number is too short or contains alphabetic characters? Therefore, to prevent throwing an exception for this simple formatting problem, we would prefer approach number (2). Note that this example is for data coming from a database, if you were dealing with user input, you would probably want to notify the user of format problems so they could fix them.

## Give Feedback from Your Catch Block

Some exceptions need to be documented and communicated, especially in those cases where a system administrator needs to take immediate action to correct an error, or when the application programmer needs to fix a processing problem. In those cases, sufficient situational details (like the stack trace and the data being processed) need to be transmitted. There are three common ways to transmit the data:

- Log it in an error log file or database.
- Send an e-mail or other message to an on-call support address.
- Print a message to the console or application window.

In client/server applications, the application user is rarely the person who needs to get the error details, so sending that data to the console or application window will not help solve the problem, but you do need to inform the user that there is a problem, with a sensible message like, "An error has occurred. We have informed the administrator. Please try again later. Thank you."

If an error is not likely to be urgent, you might rely on logging the details and hope the administrator reviews the logs at some time; otherwise, you may need to send a message directly to the administrator by e-mail or some other messaging system.

## Govern the Amount of Exception Reporting in Any Major Outage

When catching exceptions and logging or sending e-mails or other messages, beware of the potential to make the problem worse by excessively reporting the problem. When everyone in your organization runs into the same problem at the same time, you don't want your administrator's e-mail to fill up immediately with 1,000 or 10,000 of the exact same error messages. You need to govern the flow.

My approach (not detailed here) is to send the first 10 identical messages, then every tenth in the first hundred, then every hundredth. In this way, the administrator gets less than 30 messages, even if 1,000s are generated.

## Consider Throwing A Generic Exception

For web applications, I have used the approach of throwing generic exceptions. No matter how deep you are in an application's methods, when you catch an exception, you deal with it, and then throw a generic application exception. The exception you throw is nothing special – it is an ordinary exception that extends `Exception` perhaps, but has no other members.

Why would you create such a façade as a generic exception? We need something simple and unique to our application that we can use like a flag. The generic exception is a flag that lets us know that we have already dealt with the exception and are just notifying the chain of command. In all application methods, no matter how deep, you also catch this generic application exception. Let's call this façade exception `GenAppException`. Your standard `try/catch` syntax would look like this:

```
try {
        …
} catch( GenAppException g ) {
        throw g;
} catch( Exception x ) {
        // Deal with x
        throw new GenAppException();
} finally{
        …
}
```

The thing to note here is that no matter what exception we catch here, if we can't just ignore it, we end up throwing a `GenAppException`. We either throw the `GenAppException` we caught from some deeper level, or we throw a new one.

You know two things when you catch one of these generic application exceptions:

- There was an exception that we need to take note of.
- It was handled earlier in the code, so all we need to do is pass it along (throw it).

And here is the clincher: when you get to the top level, ready to present something from the web application to the user's browser, you catch any exception and present a sensible error page to the user. They don't want to read your stack trace or exception details, and displaying that data might be a security concern. All that information should have been logged or e-mailed to the system administrator or application programmer when you initially dealt with the original exception.

## Close Local Resources in a finally Block

We have already discussed this in our example code. Let me conclude by saying that every time you open a database `Connection`, `Statement`, or `ResultSet`, you need to embed the `open()` and all use of that resource in a `try` block, and close the resource in the associated `finally` block. This rule applies not only to Oracle `Connections`, but also to `Files`, `Buffers`, `Streams`, and so on.

# Java Virtual Machine Sandbox

We have already explored the JVM and seen that it is the interface between the compiled Java byte code and the particular machine hardware and operating system where it is running. This gives Java byte code the ability to "run anywhere"—anywhere there is a JVM, that is.

In addition to lending portability, the JVM can enhance security by establishing a runtime security sandbox. The sandbox is a set of rules that apply to all Java code running in the JVM. In most cases where a security policy is in effect, these rules prevent the Java code from reading from or writing to the hard drive, and from opening network connections. When the JVM is running in a browser (like Internet Explorer or FireFox), the Java code (usually a Java applet) is not allowed to communicate with any other machine on the network except the machine from which it was loaded (the web server that provided the web page that included the applet tag).

At the command prompt and in many Java web servers, there is no sandbox in the JVM. In that case, your Java code can open and write files and read from other machines on the network.

In the Oracle JVM, the sandbox is quite rigid, preventing the Java code 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.

With every sandbox, additional privileges can be granted in a number of ways. With applets, the code needs to be signed with a certificate in order to be granted extra privileges. When it runs in a browser, the browser presents the certificate to the computer user, who has to accept the certificate and privileges before the privileges become functional. Once the certificate is accepted, the applet is permitted to perform the privileged actions.

On the Oracle database, 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.

## Chapter Review

Although this has been only a brief introduction to the Java language, it has provided you with enough of the fundamentals to get you started. You cannot build security on sand; rather, you need a foundation. There are many options and styles in Java programming, and we want to intentionally choose our coding practices to assure, enhance and maintain security.

We learned about packages and their relationship to directory paths. We also learned about the `CLASSPATH` environment variable.

There were a boatload of terms and concepts introduced in this chapter, including class, instance, object, method, member, and constructor. Also the modifiers `static`, `public`, and `private` were covered. And we discussed the `main()` method.

Finally, we covered exception handling and the `try/catch/finally` syntax.

We will be using all these Java syntax elements throughout the rest of this book. If they are new to you now, don't worry. They will be like second nature to you before long.

CHAPTER 4

# Java Stored Procedures

When I first discovered Java stored procedures, I made a formal presentation of my findings to the IT department of the company where I worked, giving several examples, including using Java in an Oracle database to send e-mail, to read web pages, and to calculate Unix crypt values for the authentication of application users. That was in 2001, and as we shall see as we go through this text, I am still presenting the same ideas. However, it is only now that these ideas can be realized, on account of the upgrade to the Oracle JVM that happened with Oracle database 11g. With Oracle database 11g, Oracle has upgraded the Oracle JVM to run with Java version 1.5, which includes a significant upgrade to the Java cryptography extension (JCE) package included with the JVM.

So, what does Java encryption have to do with Oracle? Well, the standard Oracle database does not provide encryption of data for transmission to client applications out of the box. You can buy add-ons, such as Oracle Advanced Security, to provide it, but with the Oracle JVM running in the database, we can use Java to do data encryption on the Oracle database. The only way to get to that Java functionality in Oracle database is through Java stored procedures. Having the power of programs written in Java running in the Oracle database will give us encryption and much more.

There was no impediment to running enhanced encryption in previous versions of the Oracle JVM, and I have been doing so ever since my initial presentation in 2001. The only problem was one of effort and standardization, and the vetting (acceptance) of the algorithms. Before enhancements to JCE in Java 1.5, we had to acquire algorithms or write them ourselves and manually install them in Oracle database. However, for encryption, we are now able to call on the standard functions and algorithms included in JCE.

In this chapter, we will learn how to connect from SQL queries on the Oracle database to the Oracle JVM in order to let Java do our processing and return data to us. We will also cover the basics of connecting to Oracle database from Java and running database queries. In fact, we will connect from Oracle through Java, using Java to read Oracle.

## Java Stored Procedure Example

Nothing speaks to a programmer more clearly than code. Let the example that I'm going to present serve as a guide to begin understanding how Java stored procedures work. You will see an Oracle header to the Java code shown in Listing 4-1 in bold font. And you will also observe the definition of an Oracle function, following the Java code, which encapsulates the Java code. You cannot call Java code directly from an Oracle query, but you can call the Oracle function or procedure which will then call and run the Java.

Place a bookmark here so you can refer back Listing 4-1 in the discussion that follows. We will load this code into the Oracle database shortly.

*Listing 4-1. MyApp4 and f_get_oracle_time*

```
SET ROLE appsec_role;

CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED myapp4 AS
package pkg4;
import java.sql.*;
import oracle.jdbc.driver.OracleDriver;
public class MyApp4 {
    public static String getOracleTime() {
        String timeString = null;
        Statement stmt = null;
        try {
            //Class.forName( "oracle.jdbc.driver.OracleDriver" );
            //new oracle.jdbc.OracleDriver();
            //Connection conn = new OracleDriver().defaultConnection();
            Connection conn = DriverManager.getConnection("jdbc:default:connection");
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery( "select sysdate from dual" );
            if( rs.next() ) {
                timeString = rs.getString(1);
            }
        } catch( Exception x ) {
            timeString = x.toString();
        } finally {
            try {
                if( stmt != null ) stmt.close();
            } catch( Exception y ) {}
        }
        return timeString;
    }
}
/

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

■ **Note** The code in Listing 4-1 can be found in the file named *Chapter4/AppSec.sql*.

I want to point out one thing about this example code before we go any further. I know I said that packages, like `pkg4` in our example class `MyApp4`, are coordinated with directories, but there is no *pkg4* directory. I also said that there is always one public class defined in each **.java* file with the class having

the same name as the file. However in our example, there is no *MyApp4.java* file, and the file we have is not even named "MyApp4" dot anything.

The reason for this discrepancy is that we are only defining this class on the Oracle database. As it is right now, we can't compile or run it on the client. And when we load it into the Oracle database, the things I said about packages and classes suddenly becomes true again. The Oracle database creates a virtual package directory that it can search like a CLASSPATH to find the class. Also, it compiles the code just like a regular **.java* file and creates the class files, that is virtual files, stored in the database.

## Acquiring the Privilege to Load a Java Stored Procedure

Application security user, `appsec`, needs the `CREATE PROCEDURE` privilege to load Java into the database. She gets that privilege through the `appsec_role`. Recall that we set the `appsec_role` to be a non-default role for `appsec` user. For that reason, when `appsec` connects to Oracle database, she does not have that role at the outset. Rather, she has to enable it by setting `appsec_role` for her current session. The method shown (`SET ROLE appsec_role`) is sufficient, and as you recall, the new role becomes the only role that `appsec` has in her current session. An alternative approach is to request that all roles granted to `appsec` (both default and non-default) be enabled in the current session with this command:

```
SET ROLE ALL;
```

## Loading Java in the Oracle Database

There are a couple ways to get Java code into the Oracle database. One way is to use the *loadjava.exe* utility. That utility is available in both the Oracle database and in some versions of the Oracle client software. It is in the same bin directory as *sqlplus.exe* and other Oracle applications. Using *loadjava*, we can submit a java file (or *sqlj* file, with SQL embedded in Java code), a class file, or even a jar file to the Oracle database. The server handles whatever we submit in an appropriate manner. Here is an example `loadjava` command that we might have used to load a Java file named *MyApp4.java* (refer to Listing 4-1).

```
loadjava -force -resolve -user appsec/password@orcl pkg4/MyApp4.java
```

That command would attach to Oracle database as `appsec` user, then read the *MyApp4.java* file from the *pkg4* directory and submit it to Oracle database. The new code would overwrite anything that existed previously by the same name, and the Oracle database would compile the code and place a virtual *MyApp4.class* file in a virtual *pkg4* directory.

Another approach we can use to load Java code into Oracle database is what we've used in our example, Listing 4-1. We have the Java code for the MyApp4 class listed after this Oracle statement:

```
CREATE OR REPLACE AND RESOLVE JAVA SOURCE NAMED myapp4 AS
```

We can run this `CREATE` statement while connected to Oracle database, and it will accomplish the same things that we would have accomplished through the `loadjava` command.

## Handling Exceptions in a Java Stored Procedure

As you can see in the `getOracleTime()` method in the `MyApp4` class in Listing 4-1, we start out with a `try/catch/finally` block. Whenever you are dealing with databases, you must be prepared to catch or throw a `SQLException`. There are many cases when a `SQLException` would be thrown; for example, if you misspell a table name or a column name or if you don't have permission to read the data or if the Oracle database has a problem responding. `IOExceptions` can also be generated when doing input/output (IO). Communicating with Oracle database is I/O as is reading and writing files, and as is reading from network resources, like web servers.

Based on the discussions on exception handling in Chapter 3, we select to catch all exceptions and deal with them locally, setting the string that we are about to return to be equal to the value returned from `Exception.toString()`. When we get to the end of the `try` / `catch` / `finally` blocks, the last line returns `timeString`, which may be the Oracle database data and time if successful, else it will be the `Exception.toString()` message. (Please note that the `toString()` method in `Exception` is not static. You will need an instance of `Exception`, like `x`, to call `x.toString()`. I'm just saying `Exception.toString()` for discussion clarity.)

Here is an example `Exception.toString()` message:

```
java.sql.SQLException: Invalid Oracle URL specified.
```

In the `finally` block, we close the `Statement` object, `stmt`. Again, we are dealing with Oracle database, so we must be prepared for and catch any `SQLExceptions`. However, we needn't do anything with exceptions in this case. Notice that we test to see if `stmt` is `null` before we try to close it. That test will keep us from generating a `NullPointerException` in case we never instantiated the `Statement` object. (`NullPointerException` is generated when we attempt to use methods or members of a class that has not been instantiated.) Maybe you already surmised correctly that we don't need to do the test on whether `stmt` is `null`. Since we are in a `try`/`catch` block and doing nothing with exceptions, we could just attempt to close the `stmt` and it will either succeed or we will catch the `NullPointerException`. That is true, but I don't like generating exceptions needlessly, so we test if `stmt` is `null`.

Another notable item at this point in our exploration of Java is the declaration of `timeString` and `stmt`. First, notice that we declare (mention) them before the `try` block. We need to do that so that we can return `timeString` outside the `try` block, and so that we can close the `stmt` in the `finally` block, which is outside the `try` block. Member variables are contextual, and only reside within the scope of their declaration. That is to say, a member variable only exists within the block (curly braces) where it is declared. For example, the `ResultSet` member does not exist outside of the `try` block.

The second thing to notice about the declaration of `timeString` and `stmt` is that they are both declared initially to be `null`. As I mentioned earlier, they would point to no object (`null`) even if we didn't declare them as `null`. This is true of all Java objects, not just Strings and Statements. However, the compiler looks ahead and sees that we are going to return `timeString`, and it doesn't like to return unknowns. If you tried to compile a method with these two lines:

```
String myString;
Return myString;
```

The compiler will report an error and refuse to compile your code, saying that `myString` may not have been *initialized* (set to point at an object or at a value). But if you change the declaration of `myString` to be:

```
String myString = null;
```

then the compiler will accept that (at least) *you* take responsibility for the `null` value, and it will compile your code.

You might ask why the compiler doesn't see that `timeString` is initialized later on (we say `timeString = rs.getString(1)`) when we are reading through the `ResultSet`? *Javac* is a pretty smart compiler. It sees the initialization there but notices that the initialization occurs *conditionally*—meaning, it happens after the `if` statement.

So, why is `stmt` declared to be `null`? It's for a very similar reason. If you declared `stmt` like this, without initializing it even to `null`; then, when the compiler looks ahead and sees that you are going to test `stmt` in an `if` block, it reports an error that `stmt` might not have been initialized.

```
Statement stmt;
…
```

```
if( stmt == null ) …
```

In this case, the complaint is that we are trying to use something in our logic that may not have been initialized. Java wants to make informed decisions and sees this as a potential for confusion, even at compile time.

# Calling Oracle Database from Java

Within the Java method named `getOracleTime()`, in Listing 4-1, are calls to four classes that are used for interfacing with Oracle database: `OracleDriver`, `Connection`, `Statement`, and `ResultSet`. The following sections describe each of those four classes and what they do.

## OracleDriver

The first call is to load the `OracleDriver` in the Java database connectivity (JDBC) drivers registry (list). (You can use multiple drivers; e.g. Oracle database, SQL Server, and MySQL. Each would need to be registered.) I show several options for registering the `OracleDriver`. They are all commented for reasons I'll explain in a second.

```
//Class.forName( "oracle.jdbc.driver.OracleDriver" );
//new oracle.jdbc.OracleDriver();
//Connection conn = new OracleDriver().defaultConnection();
```

The first option uses a call to instantiate an `OracleDriver` class, sight unseen, by using the `Class.forName()` call. The `Class.forName()` method is a way to instantiate an object by simply stating its fully qualified name. By instantiating the `OracleDriver` this way, it is automatically loaded into the drivers registry. This is the standard practice for loading the `OracleDriver`, and you will see it frequently.

The second option is to instantiate a `new OracleDriver()`, which automatically gets loaded in the drivers registry. Lastly, we have an option to create a new instance of `OracleDriver` (again automatically loaded in the drivers registry) and in the same line of code, use the `OracleDriver` to get a `Connection` object.

Now, let me tell you why all of these invocations are commented. There are two reasons. The first has to do with the fact that we are running this code as a Java stored procedure on the Oracle database. As part of the Oracle session, the Oracle JVM automatically loads and registers the `OracleDriver`.

The second reason, but one that is less dependable, is that from Java version 1.6 on, the `OracleDriver` is loaded automatically, as needed (with *ojdbc6.jar* in the `CLASSPATH` and a hint from the connection string, e.g. "*jdbc:**oracle**:thin*"). This is less dependable, because you as the developer have the option of compiling and running this code with whatever version of Java you have available. You are only required to have JDK 1.5 or later to follow this book, so we will continue to load the `OracleDriver` in any code that will run on your client workstation.

## Connection and Statement

Next in Listing 4-1, we establish a connection to Oracle database (repeated in the following). We define the connection based on the default, "*jdbc:default:connection.*" We will place this Java code into the Oracle database, so we do not need to specify an Oracle server or Oracle listener port number, nor do we need to identify an Oracle user or password. By default, the Java stored in the database runs as the user associated with the schema where the Java is loaded. Because we are creating this Java structure using the `appsec` user, it will run with the default roles and privileges of `appsec`.

```
Connection conn = DriverManager.getConnection("jdbc:default:connection");
stmt = conn.createStatement();
```

We use our instance of `Connection` to create a `Statement` by calling `conn.createStatement()`. Note that these are the standard Java classes used in Java to talk to any vendor's SQL database, including SQL Server and DB2. Yet, because we requested (or received by default) the `OracleDriver` to give us the `Connection`, we have a `Connection` that is specific to Oracle database.

Each database vendor must implement these standard interfaces (Connection and Statement) to enable our Java code to talk with their databases using vendor-specific drivers. The Oracle-specific drivers and classes are what we get by including *ojdbc6.jar* in our `CLASSPATH`. There are several additional types of `Statements` and `Connections` that we will use—the variations being specific to Oracle database.

## ResultSet

Once you have your `Statement` object, you can execute Oracle commands, like the one that we execute in the example, Listing 4-1 and repeated in the following. When we execute a query by calling `stmt.executeQuery()`, we expect to get data back. Data from a query is returned in a `ResultSet` object (`rs` in our example). By calling the `next()` method of the `ResultSet`, we bring the values from the next (first) row into the `ResultSet` object so we can request each value (each column in the row specified in the query) with a "getter" method, like the call to `rs.getString()` in our example code.

```
ResultSet rs = stmt.executeQuery( "select sysdate from dual" );
if( rs.next() ) {
        timeString = rs.getString(1);
}
```

An interesting thing about the `ResultSet` "getter" methods is that, for some types that are returned, a translation can be done automatically. For example, in our code, we are requesting a `Date` type (value), because in our query we `SELECT`ed (asked for) the value of `SYSDATE` which is the time and date of the Oracle database. Yet in our call to the `ResultSet` we asked to `getString()`, so the `ResultSet` does a translation from `Date` type to `String` type.

Picture the data being returned in the `ResultSet` as a spreadsheet with rows of data entries, and for each row there are multiple columns of associated values. When you call `ResultSet.next()`, you are getting the equivalent of the next row into the current `ResultSet` values space. Once there, you can get the value from each column using the column number. In our example, we get the value from the first column with the statement:

```
rs.getString(1);
```

Uniquely, the index for a `ResultSet` is 1-based; that is, the first column value is at index 1. This is different from everything else in Java (that I can think of). Java arrays and `Collections` are typically 0-based, and the first element is at index 0. Also, character places in a `String` start with 0.

We will almost always want to see if there is any data being returned in the `ResultSet` before we try to use it. For that reason, we put a `ResultSet.getString()` call in the `if` condition statement that tests whether there is a next row. Alternatively, we could just say:

```
rs.next();
rs.getString(1);
```

However, if we did this and there were no "next," then the `ResultSet` values space would be `null`, and when we call `rs.getString(1)`, we'd generate an `Exception`.