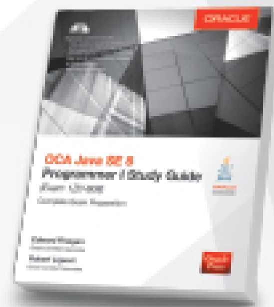
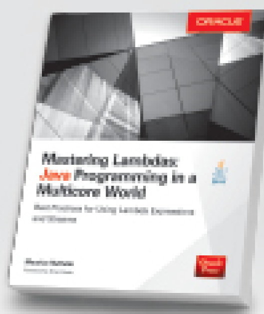
Oracle Press Java Chapter Sampler

Exclusive to JavaOne and Oracle OpenWorld 2015 Attendees!





www.OraclePressBooks.com • @OraclePress

CHAPTER 1

JavaFX Fundamentals

In today's computing environment the user interface is a key factor in determining a program's success or failure. The reasons for this are easy to understand. First, the look and feel of a program defines the initial user experience. Thus, it forms the user's first impression—and first impressions matter because they often become lasting impressions. Second, the user interface is the way in which a user interacts with a program each time it is used. Therefore, the overall quality of a program is judged, in part, by the usability and appeal of its interface. To be successful, a user interface must be convenient, well organized, and consistent. It must also have one thing more: that "visual sparkle" that users have come to expect. For today's Java programmer, JavaFX is the best way to provide such interfaces.

JavaFX is a collection of classes and interfaces that defines Java's modern graphical user interface (GUI). It can be used to create the types of GUIs demanded by rich client applications in the contemporary marketplace. JavaFX supplies a diverse set of controls, such as buttons, scroll panes, text fields, check boxes, trees, and tables, that can be tailored to fit nearly any application. Furthermore, effects, transforms, and animation can be employed to enhance the visual appeal of the controls. JavaFX also streamlines the creation of an application by simplifying the management of its GUI elements and the application's deployment. Thus, JavaFX not only enables you to build more exciting, visually appealing user interfaces, it also makes your job easier in the process. Simply put: JavaFX is a powerful, state-of-the-art GUI framework that is defining the future of GUI programming in Java.

This book provides a compact, fast-paced introduction to JavaFX programming. As you will soon see, JavaFX is a large, feature-rich system, and in many cases, one feature interacts with or supports another. As a result, it can be difficult to discuss one aspect of JavaFX without involving others. The purpose of this chapter is to introduce the fundamentals of JavaFX, including its history, basic concepts, core features, and the general form of a JavaFX program. Subsequent chapters will expand on the foundation presented here, so a careful reading is advised.

One more point: This book assumes that you have a working knowledge of Java. You need not be a Java expert, but you should be comfortable with the fundamentals of the language. However, prior experience with other GUI frameworks is *not* required, although such prior experience may help you advance more quickly.

We will begin by putting JavaFX into its historical context.

A Brief History of Java's GUI Frameworks

Like most things in programming, the GUI frameworks defined by Java have evolved over time, and JavaFX is Java's third such framework. Before you begin programming with JavaFX, it is helpful to understand in a general way why JavaFX was created and how it relates to and improves on Java's previous GUIs.

The AWT: Java's First GUI Framework

Java's original GUI framework was the Abstract Window Toolkit (AWT). The AWT offered only rudimentary support for GUI programming. For example, its set of controls is quite limited by today's standard. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding platform-specific equivalents, or *peers*. Because the AWT components rely on native code resources, they are referred to as *heavyweight*.

The AWT's use of native peers led to several problems. For example, because of variations between operating systems, a component might act differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Also, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Furthermore, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Swing

Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing addressed the limitations associated with the AWT's components through the use of two key features: *lightweight components* and a *pluggable look and feel*. Let's look briefly at each. With very few exceptions, Swing components are *lightweight*. This means the components are written entirely in Java. They do not rely on platform-specific peers. Lightweight components have some important advantages, including efficiency and flexibility. Furthermore, because lightweight components do not translate into platform-specific peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component can work in a consistent manner across all platforms. It is also possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: you can "plug in" a new look and feel. In other words, it becomes possible to change the way that a component is rendered without affecting any of its other aspects or creating side effects in the code that uses the component. In short, Swing solved the problems of the AWT in an effective, elegant manner.

There is one other important aspect of Swing: it uses an architecture based on a modified Model-View-Controller (MVC) concept. In MVC terminology, the *model* corresponds to the state information associated with a component. The *view* determines how the control is displayed on the screen. The *controller* determines

how the component reacts to the user. The MVC approach enables any of its pieces to be changed without affecting the other two. For example, you can change the view without affecting the model. In Swing, the high level of separation between the view and the controller was not beneficial. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity. This is called *separable model architecture*. However, the benefits of the MVC concept are still attained, providing support for Swing's pluggable look-and-feel capabilities.

JavaFX

Swing was so successful that it remained the primary Java GUI framework for over a decade, which is a very long time in the fast-paced world of computing. Of course, computing continued to move forward. Today the trend is toward more dramatic, visually engaging effects—that "visual sparkle" mentioned earlier. Such effects were troublesome to create with Swing. Furthermore, the conceptual basis that underpins the design of GUI frameworks has advanced beyond that used by Swing. To better handle the demands of the modern GUI and utilize advances in GUI design, a new approach was needed. The result is JavaFX: Java's next-generation GUI framework.

JavaFX offers all of the advantages of Swing but provides a substantially updated and improved approach. For example, it defines a set of modern GUI controls and enables you to easily incorporate special effects into those controls. Its improved architecture, based on the *scene graph* feature described later in this chapter, streamlines the management of a program's windows. For example, it automates the once-tedious repaint process. Like Swing, JavaFX uses an MVC-based architecture. Deployment is simplified because JavaFX applications can be run in a variety of environments without recoding. Although not the focus of this book, JavaFX also supports the use of CSS and FXML to style and build a GUI. In short, JavaFX sets a new standard for the contemporary GUI framework.

It is important to mention that the development of JavaFX occurred in two main phases. The original JavaFX was based on a scripting language called *JavaFX Script*. However, JavaFX Script has been discontinued. Beginning with the release of JavaFX 2.0, JavaFX has been programmed in Java itself and provides a comprehensive API. JavaFX has been bundled with Java since JDK 7, update 4. At the time of this writing, the latest version of JavaFX is JavaFX 8, which is included with JDK 8. (The version number is 8 to align with the JDK version. Thus, the numbers 3 through 7 were skipped.) JavaFX 8 is the version of JavaFX described in this book. When the term *JavaFX* is used, it refers to JavaFX 8.

Before we continue, it is useful to answer one question that naturally arises relating to JavaFX: Is JavaFX designed as a replacement for Swing? The answer is, essentially, Yes. However, Swing will continue to be part of Java programming for some time to come. The reason is that there is a large amount of Swing legacy code. Furthermore, there are legions of programmers who know how to program for

Swing, Nevertheless, JavaFX has clearly been positioned as the GUI framework of the future. It is expected that over the next few years, JavaFX will supplant Swing for new projects, and many Swing-based applications will migrate to JavaFX. One other point: it is also possible to use both JavaFX and Swing in an application, thus enabling a smooth transition from Swing to JavaFX.

JavaFX Basic Concepts

Before you can create a JavaFX application, there are several key concepts and features you must understand. Although JavaFX has similarities with Java's other GUIs, it has substantial differences. For example, the overall organization of JavaFX and the relationship of its main components differ significantly from either Swing or the AWT. Therefore, even if you have experience in coding for one of Java's other GUI frameworks, a careful reading of the following sections is advised.

The JavaFX Packages

The JavaFX framework is contained in packages that begin with the **javafx** prefix. At the time of this writing, there are more than 30 JavaFX packages in its API library. Here are four examples: **javafx.application**, **javafx.stage**, **javafx.scene**, and **javafx.scene.layout**. Although we will use only a few JavaFX packages in this chapter, you might want to spend some time browsing the JavaFX packages. Doing so will give you an idea of the wide array of functionality that JavaFX offers.

Setting the Stage with the Stage and Scene Classes

The central metaphor implemented by JavaFX is the *stage*. As in the case of an actual stage play, a stage contains a *scene*. Thus, loosely speaking, a stage defines a space and a scene defines what goes in that space. Or, put another way, a stage is a container for scenes and a scene is a container for the items that comprise the scene. As a result, all JavaFX applications have at least one stage and one scene. These elements are encapsulated in the JavaFX API by the **Stage** and **Scene** classes. To create a JavaFX application, you will, at a minimum, add at least one **Scene** object to a **Stage**. Let's look a bit more closely at these two classes.

Stage is a top-level container. All JavaFX applications automatically have access to one **Stage**, called the *primary stage*. The primary stage is supplied by the run-time system when a JavaFX application is started. Although you can create other stages, for many applications, the primary stage will be the only one required.

As mentioned, **Scene** is a container for the items that comprise the scene. These can consist of various types of GUI elements, such as controls, text, and graphics. To create a scene, you will add elements to an instance of **Scene**. Then, set that **Scene** on a **Stage**.

Nodes and Scene Graphs

The elements of a scene are called *nodes*. For example, a push button control is a node. However, nodes can also consist of groups of nodes. Furthermore, a node can have a child node. In this case, a node with a child is called a *parent node* or *branch node*. Nodes without children are terminal nodes and are called *leaves*. The collection of all nodes in a scene creates what is referred to as a *scene graph,* which comprises a *tree*.

There is one special type of node in the scene graph, called the *root node*. This is the top-level node and is the only node in the scene graph tree that does not have a parent. Thus, with the exception of the root node, all other nodes have parents, and all nodes either directly or indirectly descend from the root node.

The base class for all nodes is **Node**. There are several other classes that are, either directly or indirectly, subclasses of **Node**. These include **Parent**, **Group**, **Region,** and **Control**, to name a few.

Layouts

JavaFX provides several layout panes that manage the process of placing elements in a scene. For example, the **FlowPane** class provides a flow layout and the **GridPane** class supports a row/column grid-based layout. Several other layouts, such as **BorderPane**, which organizes output within four border areas and a center, are available. Each inherits **Node**. The layouts are packaged in **javafx.scene.layout.**

The Application Class and the Life-Cycle Methods

A JavaFX application must be a subclass of the **Application** class, which is packaged in **javafx.application**. Thus, your application class will extend **Application**. The **Application** class defines three life-cycle methods that your application can override. These are called **init( )**, **start( )**, and **stop( )**, and are shown here, in the order in which they are called:

void init( )

abstract void start(Stage *primaryStage*)

void stop( )

The **init( )** method is called when the application begins execution. It is used to perform various initializations. As will be explained, it *cannot,* however, be used to create a stage or build a scene. If no initializations are required, this method need not be overridden because an empty, default version is provided.

The **start( )** method is called after **init( )**. This is where your application begins, and it *can* be used to construct and set the scene. Notice that it is passed a reference to a **Stage** object. This is the stage provided by the run-time system and is the primary stage. Also notice that this method is abstract. Thus, it must be overridden by your application.

When your application is terminated, the **stop( )** method is called. It is here that you can handle any cleanup or shutdown chores. In cases in which no such actions are needed, an empty, default version is provided.

Launching a JavaFX Application

In general, when a JavaFX application begins execution, an instance of the subclass of **Application** defined by the application is created. Then **init( )**, followed by **start( )**, is executed. However, sometimes, such as in the case of a free-standing, self-contained JavaFX application, a call to the **launch( )** method defined by **Application** may be needed. So that the examples in this book can be run in all of the ways supported by JavaFX, **launch( )** is included in all of the programs.

The **launch( )** method has two forms. Here is the one used in this book:

public static void launch(String ... *args*)

Here, *args* is a possibly empty list of strings that typically specifies command-line arguments. When called, **launch( )** causes the application to be constructed, followed by calls to **init( )** and **start( )**. The **launch( )** method will not return until after the application has terminated. This version of **launch( )** starts the subclass of **Application** from which **launch( )** is called. The second form of **launch( )** lets you specify a class other than the enclosing class to start. As a general rule, **launch( )** is called from **main( )**.

It is important to emphasize that neither a **main( )** method nor a call to **launch( )** is necessary in all cases for a JavaFX program. So don't be surprised when you see other JavaFX code that does not include them. However, including **main( )** and **launch( )** ensures that the code can be used in the widest range of circumstances. Also, an explicit call to **launch( )** is needed if your application requires a **main( )** method for a purpose other than starting the JavaFX application. Thus, the programs in this book include both **main( )** and **launch( )** methods.

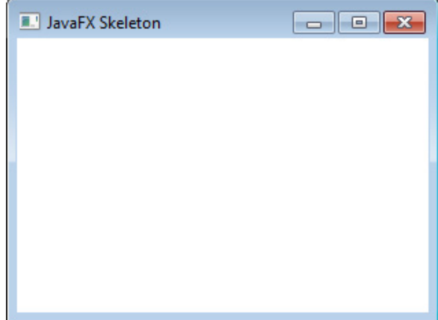
A JavaFX Application Skeleton

All JavaFX applications share the same basic skeleton. Therefore, before looking at any more JavaFX features, it will be useful to see what that skeleton looks like. In addition to showing the general form of a JavaFX application, the skeleton illustrates how to launch the application and demonstrates when the life-cycle methods are called. A message noting when each life-cycle method executes is displayed on the console via **System.out**. The complete skeleton is shown here:

```
// A JavaFX application skeleton.

import javafx.application.*;
import javafx.scene.*;
```

```
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

  public static void main(String[] args) {

    System.out.println("Launching JavaFX application.");

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the init() method.
  public void init() {
    System.out.println("Inside the init() method.");
  }

  // Override the start() method.
  public void start(Stage myStage) {

    System.out.println("Inside the start() method.");

    // Give the stage a title.
    myStage.setTitle("JavaFX Skeleton");

    // Create a root node. In this case, a flow layout
    // is used, but several alternatives exist.
    FlowPane rootNode = new FlowPane();

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Show the stage and its scene.
    myStage.show();
  }

  // Override the stop() method.
  public void stop() {
    System.out.println("Inside the stop() method.");
  }
}
```

Although the skeleton is quite short, it can be compiled and run. It produces the empty window, shown here:



The skeleton also produces the following output on the console:

```
Launching JavaFX application.
Inside the init() method.
Inside the start() method.
```

When you close the window, this message is displayed:

```
Inside the stop() method.
```

Of course, in a real program, the life-cycle methods would not normally output anything to **System.out**. They do so here simply to illustrate when each method is called. Furthermore, as explained earlier, you will need to override the **init( )** and **stop( )** methods only if your application must perform special startup or shutdown actions. Otherwise, you can use the default implementations of these methods provided by the **Application** class.

Let's examine this program in detail. It begins by importing four packages. The first is **javafx.application**, which contains the **Application** class. The **Scene** class is packaged in **javafx.scene**, and **Stage** is packaged in **javafx.stage**. The **javafx.scene.layout** package provides several layout panes. The one used by the program is **FlowPane**.

Next, the application class **JavaFXSkel** is created. Notice that it extends **Application**. As explained, **Application** is the class from which all JavaFX applications are derived. **JavaFXSkel** contains four methods. The first is **main( )**. It is used to launch the application via a call to **launch( )**. Notice that the **args** parameter to **main( )** is passed to the **launch( )** method. Although this is a common approach, you can pass a different set of parameters to **launch( )**, or none at all. One other point: as mentioned earlier, **launch( )** and **main( )** are not required in all cases. However, for reasons already explained, both **main( )** and **launch( )** are included in the programs in this book.

When the application begins, the **init( )** method is called first by the JavaFX run-time system. For the sake of illustration, it simply displays a message on **System.out**, but it would normally be used to initialize some aspect of the application. Of course, if no initialization is required, it is not necessary to override **init( )** because an empty, default implementation is provided. It is important to emphasize that **init( )** cannot be used to create the stage or scene portions of a GUI. Rather, these items should be constructed and displayed by the **start( )** method.

After **init( )** finishes, the **start( )** method executes. It is here that the initial scene is created and set to the primary stage. Let's look at this method line by line. First, notice that **start( )** has a parameter of type **Stage**. When **start( )** is called, this parameter will receive a reference to the primary stage of the application. It is on this stage that you will set a scene for the application.

After displaying a message on the console indicating that **start( )** has begun execution, **start( )** sets the title of the stage using this call to **setTitle( )**:

```
myStage.setTitle("JavaFX Skeleton");
```

Although this step is not necessarily required, it is customary for stand-alone applications. This title becomes the name of the main application window.

Next, a root node for a scene is created. The root node is the only node in a scene graph tree that does not have a parent. In this case, a **FlowPane** is used for the root node, but there are several other classes that can be used for the root.

```
FlowPane rootNode = new FlowPane();
```

As mentioned, a **FlowPane** uses a flow layout. This is a layout in which elements are positioned line by line, with lines wrapping as needed. (Thus, it works much like the **FlowLayout** class used by the AWT and Swing.) By default, a horizontal flow is used, but it is possible to specify a vertical flow. Although not needed by this skeletal application, it is also possible to specify other layout properties, such as a vertical and horizontal gap between elements and an alignment.

The following line uses the root node to construct a **Scene**:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

Scene provides several versions of its constructor. The one used here creates a scene that has the specified root with the specified width and height. It is shown here:

Scene(Parent *rootnode*, double *width*, double *height*)

Notice that the type of *rootnode* is **Parent**. It is a subclass of **Node** and encapsulates nodes that can have children. Also notice that the width and the height are **double** values. This lets you pass fractional values, if needed. In the skeleton, the root is **rootNode**, the width is 300, and the height is 200.

The next line in the program sets **myScene** as the scene for **myStage**:

```
myStage.setScene(myScene);
```

Here, **setScene( )** is a method defined by **Stage** that sets the scene to that specified by its argument.

The last line in **start( )** displays the stage and its scene:

```
myStage.show();
```

In essence, **show( )** shows the window that was created by the stage and scene.

When you close the application, its window is removed from the screen and the **stop( )** method is called by the JavaFX run-time system. In this case, **stop( )** simply displays a message on the console, illustrating when it is called. However, **stop( )** would not normally display anything. Furthermore, if your application does not need to handle any shutdown actions, there is no reason to override **stop( )** because an empty, default implementation is provided.

Compiling and Running a JavaFX Program

One important advantage of JavaFX is that the same program can be run in a variety of different execution environments. For example, you can run a JavaFX program as a stand-alone desktop application, inside a web browser, or as a Web Start application. However, different ancillary files may be needed in some cases, such as a JAR file, an HTML file, or a Java Network Launch Protocol (JNLP) file.

In general, a JavaFX program is compiled like any other Java program. However, depending on the target execution environment, some additional steps may be required. For this reason, often the easiest way to compile a JavaFX application is to use an Integrated Development Environment (IDE) that fully supports JavaFX programming, such as NetBeans. Although the specific instructions for using an IDE differ among IDEs, as a general rule, to compile and run a JavaFX program, first create a JavaFX project and then enter the JavaFX program as the project's source file.

Alternatively, if you are accustomed to using the command line and just want to compile and run the JavaFX applications shown in this book, you can easily do so using Java's command-line tools. First, compile the application in the way you do any other Java program, using **javac**. This creates a **.class** file that can then be run by **java**. For example, to compile and run **JavaFXSkel.java**, you can use this command-line sequence:

```
javac JavaFXSkel.java
java JavaFXSkel
```

If you are comfortable using the command-line tools, they offer the easiest way to try the examples in this book.

NOTE
If you use the command-line tools, you can still convert a JavaFX application contained in ***.class*** *files into a fully deployable form by use of the* ***javapackager*** *command-line tool. (This tool was previously called* ***javafxpackager****, but has been renamed.) Consult Oracle's online documentation for details.*

The JavaFX Application Thread

In the preceding discussion, it was mentioned that you cannot use the **init( )** method to construct a stage or scene. You also cannot create these items inside the application's constructor. The reason is that a stage or scene must be constructed on the JavaFX *application thread*. However, the application's constructor and the **init( )** method are called on the main thread, also called the *launcher thread*. Thus, they can't be used to construct a stage or scene. Instead, you must use the **start( )** method, as the skeleton demonstrates, to create the initial GUI because **start( )** is called on the application thread.

Furthermore, any changes to the GUI currently displayed must be made from the application thread. Fortunately, this is a fairly easy rule to follow because, as a general rule, interactions with your program, such as user input, take place on the application thread. The **stop( )** method is also called on the application thread.

Build a Simple Scene Graph

Although the preceding skeleton is fully functional, its scene graph is empty. Thus, it does not contain any elements and its window is blank. Of course, the point of JavaFX is to build user interfaces. To do this, you must build a scene graph. To introduce the process, we will build a very simple one that contains only one element: a label.

The label is one of the controls provided by JavaFX. As mentioned earlier, JavaFX contains a rich assortment of controls. Controls are the means by which the user interacts with an application. The simplest control is the label because it just displays a message or an image. Because it is quite easy to use, the label is a good way to introduce the techniques needed to begin building a scene graph.

In JavaFX, a label is an instance of the **Label** class, which is packaged in **javafx.scene.control**. **Label** inherits **Labeled** and **Control**, among other classes. The **Labeled** class defines several features that are common to all labeled elements (that is, those that can contain text), and **Control** defines features related to all controls.

The **Label** constructor that we will use is shown here:

Label(String *str*)

The string that is displayed is specified by *str*.

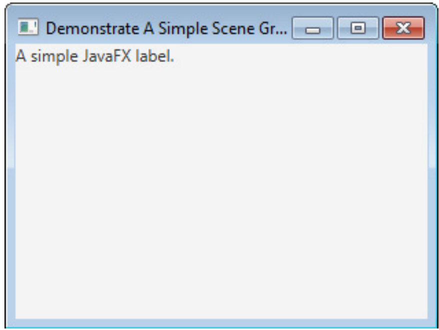
Once you have created a label (or any other control), it must be added to the scene's content, which means adding it to the scene graph. Here is the technique we will use: First call **getChildren( )** on the root node of the scene graph. It returns a list of the child nodes in the form of an **ObservableList<Node>**. **ObservableList** is packaged in **javafx.collections**, and it inherits **java.util.List**, which is part of Java's Collections Framework. **List** defines a collection that represents a list of objects. Although a discussion of **List** and the Collections Framework is outside the scope of this book, it is easy to use **ObservableList** to add child nodes. Simply call **add( )** on the list of child nodes returned by **getChildren( )**, passing in a reference to the node to add, which in this case is a label.

The following program puts the preceding discussion into action by creating a simple JavaFX application that displays a label:

```
// Demonstrate a simple scene graph that contains a label.

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class SimpleSceneGraphDemo extends Application {

  public static void main(String[] args) {

    // Start the JavaFX application by calling launch().
    launch(args);
  }

  // Override the start() method.
  public void start(Stage myStage) {

    // Give the stage a title.
    myStage.setTitle("Demonstrate A Simple Scene Graph");

    // Use a FlowPane for the root node.
    FlowPane rootNode = new FlowPane();

    // Create a scene.
    Scene myScene = new Scene(rootNode, 300, 200);

    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
```

```
    Label myLabel = new Label("A simple JavaFX label.");

    // Add the label to the scene graph.
    rootNode.getChildren().add(myLabel);

    // Show the stage and its scene.
    myStage.show();
  }
}
```

This program produces the following window:



In the program, pay special attention to this line:

```
rootNode.getChildren().add(myLabel);
```

It adds the label to the list of children for which **rootNode** is the parent. Although this line could be separated into its individual pieces if necessary, you will often see it as shown here.

Of course, a scene graph can contain more than one control. Simply add each control to the scene graph, as just shown. For example, this version of **start( )** adds three labels:

```
// Override the start() method. This time, add three labels
// to the scene graph.
public void start(Stage myStage) {

  // Give the stage a title.
  myStage.setTitle("Demonstrate A Simple Scene Graph");

  // Use a FlowPane for the root node.
  FlowPane rootNode = new FlowPane();

  // Create a scene.
  Scene myScene = new Scene(rootNode, 300, 200);
```

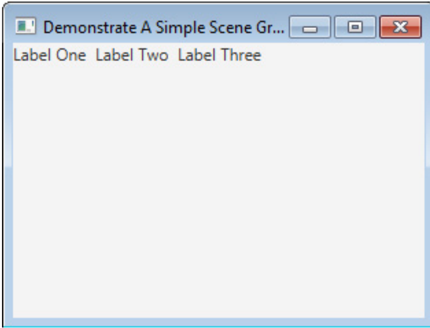
```
    // Set the scene on the stage.
    myStage.setScene(myScene);

    // Create a label.
    Label myLabel = new Label("Label One  ");

    // Create a second label.
    Label myLabel2 = new Label("Label Two  ");

    // Create a third label.
    Label myLabel3 = new Label("Label Three");

    // Add three labels to the scene graph.
    rootNode.getChildren().add(myLabel);
    rootNode.getChildren().add(myLabel2);
    rootNode.getChildren().add(myLabel3);

    // Show the stage and its scene.
    myStage.show();
  }
```

Here, the sequence

```
rootNode.getChildren().add(myLabel);
rootNode.getChildren().add(myLabel2);
rootNode.getChildren().add(myLabel3);
```

adds **myLabel**, **myLabel2**, and **myLabel3** to the root node of the graph. Thus, after this sequence executes, **rootNode** will have three child nodes.

If you substitute this version of **start( )** into the preceding program, it will produce the following window:



Notice that the three labels are positioned left to right in the order in which they were added to the scene graph. This is because a flow layout is used. With a flow

layout, elements in the scene graph are displayed line by line. When the end of a line is reached, the next line is begun. You can see this if you narrow the window produced by the program. At some point, one of the labels will automatically wrap down to the next line. If you want a different layout strategy, simply use a different layout pane. The rest of the program remains the same. Various layout panes are described later in this book, but for now, the flow layout is sufficient for our purposes.

Before moving on, it is useful to point out that **ObservableList** provides a method called **addAll( )** that can be used to add two or more children to the scene graph in a single call. For example, this line adds three labels to the scene graph in a single call:

```
// Add all three labels to the scene graph in one call
// rootNode.getChildren().addAll(myLabel, myLabel2, myLabel3);
```

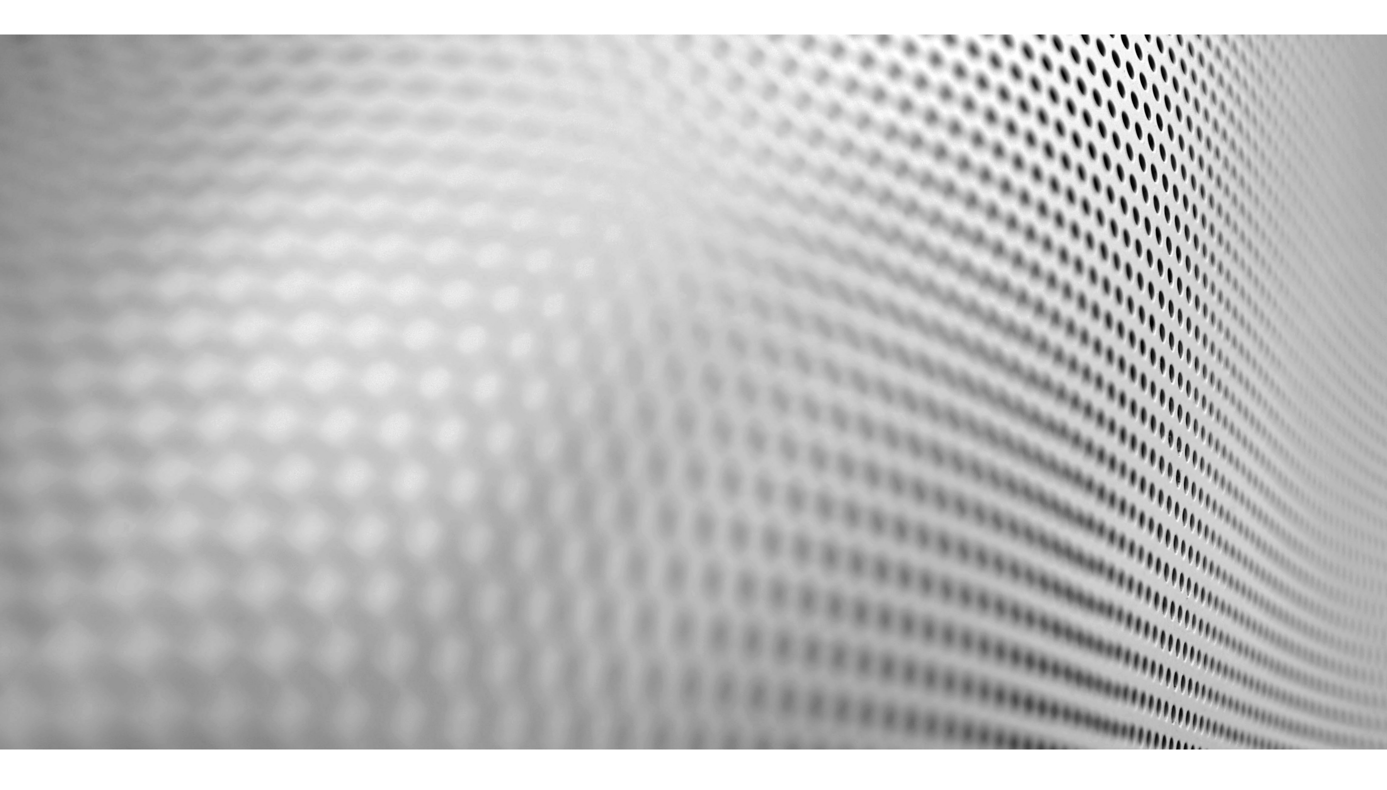
Thus, it produces the same scene graph as the one created by the three separate calls to **add( )** shown earlier. The only difference is that it is accomplished by a single call to **addAll( )**.

In addition to adding a control to a scene graph, you can remove one. This is done by calling **remove( )** on the **ObservableList** returned by **getChildren( )**. For example,

```
rootNode.getChildren().remove(myLabel);
```

removes **myLabel** from the scene.

In general, **ObservableList** supports a wide range of list-management methods. Here are two examples: You can determine if the list is empty by calling **isEmpty( )**. You can obtain the number of nodes in the list by calling **size( )**. You will want to explore **ObservableList** in greater depth as you advance in your study of JavaFX.



CHAPTER 1

Baking Pi

In this chapter I'll walk you through the process of setting up (or baking, if you will) your Raspberry Pi.

This chapter will take you through a first-time installation of Raspbian on a Raspberry Pi with the latest Java version. I will also detail some additional configuration that you may want to change to optimize Java and other visual applications. Finally, I will show you how to create a network between your Raspberry Pi and another machine and run a simple Java application.

Powering Your Raspberry Pi

The Raspberry Pi is a great platform for getting started with embedded computing. It has a great community supporting it with lots of options for hardware. If this is your first time setting up a Raspberry Pi, you will need the following hardware to get started:

- **Raspberry Pi** The same instructions apply to Models B+, A+, B, A, and 2, but in this guide I will be using a B+.
- **SD card** A good quality 8GB or larger SD card is recommended. If you purchase one with the New Out Of Box Software (NOOBS) preinstalled, you can save some time in setup.
- **Power supply** The Raspberry Pi is powered by a micro-USB cable, the recommended specifications for which are 2A at 5V. You can often get away with a smaller power supply (as small as 700mA) depending on what USB devices are connected.
- **Keyboard and mouse** Pretty much any USB keyboard will do. The mouse is optional if you don't mind navigating the GUI via the keyboard.
- **Monitor or TV** The Raspberry Pi supports composite or HDMI displays. HDMI is readily convertible to DVI or VGA if that is what your monitor supports.

The first step is to set up your SD card. If you have a Model 2, B+, or A+ Raspberry Pi, then this will be in the form of a microSD card. If you have the older Model B or A, then this will be a full-size SD card. Both types of cards

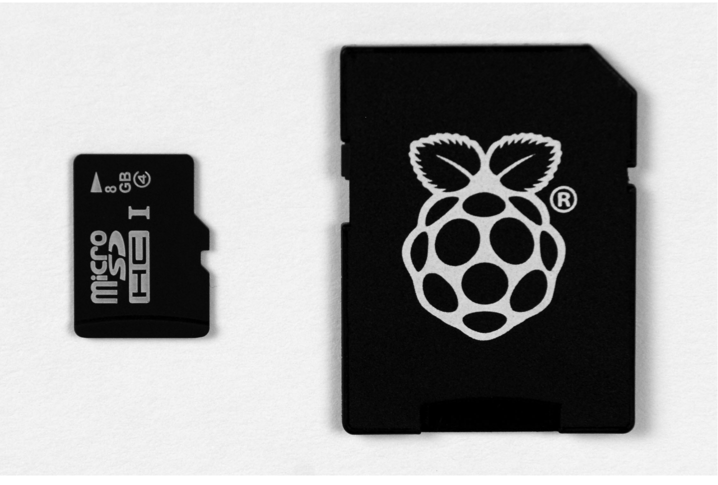


FIGURE 1-1. *NOOBS microSD card with full-size adapter*

operate the same, and microSD cards usually come with adapters to fit in full-size slots, so that is the obvious choice. The difference in size between a microSD card and a full-size SD card can be seen in Figure 1-1.

The Raspberry Pi foundation ships SD cards with NOOBS preinstalled, as photographed in Figure 1-1. This is a good trade-off between cost, convenience, and performance, and is recommended for anyone who is just getting started. Most online retailers offer a Raspberry Pi bundle that includes the NOOBS SD card for a small incremental cost.

If you have purchased an SD card with NOOBS preinstalled, skip to the section entitled “Installing Raspbian.”

Purchasing Compatible SD Cards

If you need a higher-performance or larger-capacity card, you can buy SD cards and format them yourself. This is also typically less expensive, especially if you are purchasing in bulk. The key criteria to use when selecting an SD card are size, write performance, and quality.

The minimum size card you can use with the Raspbian distribution is 4GB, although this is not large enough to support NOOBS and will leave very little room for your software. At least 8GB is recommended, and for a small incremental cost you can get a 16GB card. The largest-capacity card that is

supported in the Raspberry Pi is 64GB, although this will only be helpful if you are doing data-intensive tasks, such as storing sensor data or video over a long period of time.

When shopping for SD cards, you can find a class identifier written on them. A higher number indicates better write performance, with the minimum sustained write speed equal to the number. For example, a class 4 card is tested to support a sustained write speed of 4 megabytes per second (MBps). Similarly, a class 10 card is tested to support a sustained write speed of 10MBps. This matters most if you are developing an application that will write a large amount of sequential data. It also can significantly speed the initial setup time of your card. However, this is an indication of neither read performance nor nonsequential write performance, so the real-world performance of your SD card may vary.

Perhaps the most important factor is the quality of the card. Buying from a well-known manufacturer and reputable vendor greatly increases the chances you will get the size and performance you are paying for. Unknown manufacturers and ill-reputed vendors may sell you low-quality or counterfeit cards that perform well below their advertised specs. A good community resource for researching SD card compatibility and performance is the Raspberry Pi SD cards page on the Embedded Linux (eLinux) wiki: http://elinux.org/RPi_SD_cards.

Formatting SD Cards

The NOOBS installer requires that your SD card be formatted with a File Allocation Table (FAT) filesystem. Both FAT16 (more commonly referred to as FAT) and FAT32 are supported, but not ExFAT. If you have purchased a large SD card, it often comes formatted with ExFAT, so you will need to reformat it with FAT32 in order to proceed with the NOOBS install. The easiest way to make sure your SD card is formatted correctly is to use the SD Association's SDFormatter utility on OS X or Windows: https://www.sdcard.org/downloads/formatter_4/.

Figure 1-2 shows a screenshot of what the SDFormatter utility looks like on OS X. Make sure that the correct SD card is selected so that you don't accidentally delete the wrong drive, and then choose the Overwrite Format option. Specify the name of the card and click the Format button. This process will take a while depending upon the speed and size of your SD card, so this may be a good time to take a coffee break.

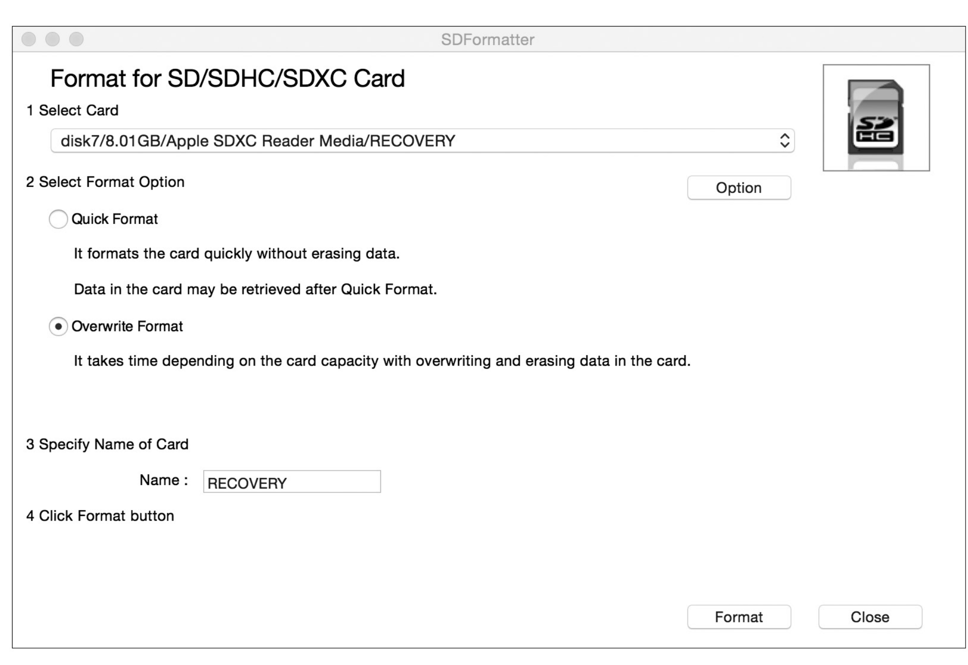


FIGURE 1-2. *SDFormatter utility*

If you are on Linux, you can accomplish the same thing with the GParted tool, which is a visual disk manager. Make sure that you select the correct partition and format as FAT or FAT32.

Once you have a properly formatted card, the rest of the installation is as simple as following these steps:

1. Download the latest version of NOOBS from the Raspberry Pi website: www.raspberrypi.org/downloads/.
2. Unzip the downloaded NOOBS archive. Most operating systems come with built-in unzipping functionality.
3. Copy the contents of the extracted folder to your SD card. Make sure that you do not have an enclosing folder.

Either NOOBS or NOOBS LITE will work, although I recommend the former so that you don't have to worry about networking your Raspberry Pi to get it up and running.

Installing Raspbian

Once you have an SD card with NOOBS on it, you are ready to install the Raspbian operating system and set up your Raspberry Pi. This process has been streamlined with the latest installers, so you should have no trouble getting set up quickly. Along the way I will point out common pitfalls that you may encounter, especially with the older Raspberry Pi models.

Connecting Your Raspberry Pi

Here are the connections you will need to make the first time you turn on your Raspberry Pi:

CAUTION

Never insert or remove the SD card while your Raspberry Pi is plugged in. This can result in corruption of the filesystem and lose important data that you have stored on your Raspberry Pi.

1. Insert the SD card into the slot on the bottom.

   On Models A and B this slot is a full-size friction fit socket, so be careful that you don't use too much force (it goes in upside down). On Models A+, B+, and 2 this slot is a spring-loaded microSD socket that clicks upon insertion (also upside down). When removing the SD card, you can simply pull out the card on Models A and B, but for Models A+, B+, and 2, press it in and allow the spring to eject it.

TIP

The SD slots on Models A and B are easy to damage if you use too much force (for example, if you force the SD card in incorrectly). Fortunately, this can often be remedied by simply bending the pins back in shape.

2. Connect the HDMI or composite cable to your monitor or TV.

   HDMI will give you better resolution and is preferred if you have a supported monitor. If you're using HDMI, plug in your monitor and turn on the power before booting. Using composite on Models A and B is fairly straightforward via the yellow RCA jack. However, on Models A+, B+, and 2 you will need to use an adapter to get the video signal out of the 3.5mm TRRS jack that is shared with audio. For more details on this, see the example project in Chapter 8 that talks about tip ordering and compatible cables.

NOTE
The reason why you should always plug in HDMI and turn on your monitor before booting is because the Raspberry Pi defaults to composite input, which will give you a black screen if you later hook up an HDMI device. However, this is not the case when running NOOBS, so you can get by the first boot without doing this in a specific order.

3. Plug in your keyboard and mouse.

   These devices plug into the full-size USB host ports on the Raspberry Pi. If you are using a Model A or A+, you will be limited to one USB port, so you can either navigate via keyboard shortcuts and skip the mouse altogether, or plug in a powered USB hub to connect more devices.

CAUTION
On Models A and B the USB ports are not hot swappable, so inserting or removing devices can reset the Raspberry Pi, resulting in lost work or filesystem corruption. This was fixed on Models A+, B+, and 2.

4. Connect the micro-USB power.

As mentioned earlier, make sure you have a power supply that can provide 5V and ideally 2A of power. Higher current is fine since the Raspberry Pi

will only consume the power it needs, but with only a keyboard and mouse hooked up, you can get away with a 700mA power supply. Your typical computer USB slot will only provide 500mA and thus is not safe to use with the Pi. On Models A and B, insufficient voltage from a poor USB cable or insufficient current from a bad power supply can result in crashes and filesystem corruption. Fortunately, the Raspberry Pi B+, A+, and 2 come with power circuitry that ensures the voltage and current are sufficient before turning on. They also draw less power than the older models, saving precious battery life for embedded projects.

How to Tell Your Raspberry Pi Is Working

Once powered on, you will notice that the LED status lights on the Raspberry Pi will light up. The red PWR LED indicates power and will stay solid as long as the Raspberry Pi is plugged in. The green ACT LED indicates activity and will start blinking irregularly shortly after you plug in the Pi. If the PWR LED comes on but the ACT LED does not blink irregularly for a few seconds, this is most likely a sign that the SD card is not working. This could be a bad connection or an improperly formatted or installed card. Here are some troubleshooting steps to try out:

- If the red PWR LED is flickering, you likely have a Model A+, B+, or 2 and have tripped the brownout circuitry. Try a different power supply (higher current) or replace your micro-USB cable (which may be too long, thin, or damaged).
- If the green ACT LED doesn't flash irregularly for a few seconds:
  - Try reseating your SD card. Turn off the power, unplug the SD card, and then plug it in again, making sure it is fully inserted. Remember that the SD card goes in upside down and should not require a lot of force to insert or remove.
  - Check your SD card formatting and installation. Your SD card should be formatted as FAT or FAT32 and have the NOOBS files in the root of the filesystem (not in a folder). You can always buy a preinstalled copy of NOOBS if you want to simplify this.

Once you know the Raspberry Pi is working from seeing a few seconds of activity on the ACT LED, the next thing to check is your display. Upon boot the Raspberry Pi shows a rainbow test pattern for a second and then displays a recovery screen for a few more seconds. After this it automatically boots into the NOOBS installer, and you should see the installation screen shown in Figure 1-3.

If your Raspberry Pi is booting according to the LEDs, but you don't see the NOOBS installation screen, try these troubleshooting tips:

- Make sure your monitor power is on and the monitor is set to the correct input (for example, it is easy to forget to switch the input from VGA to HDMI).

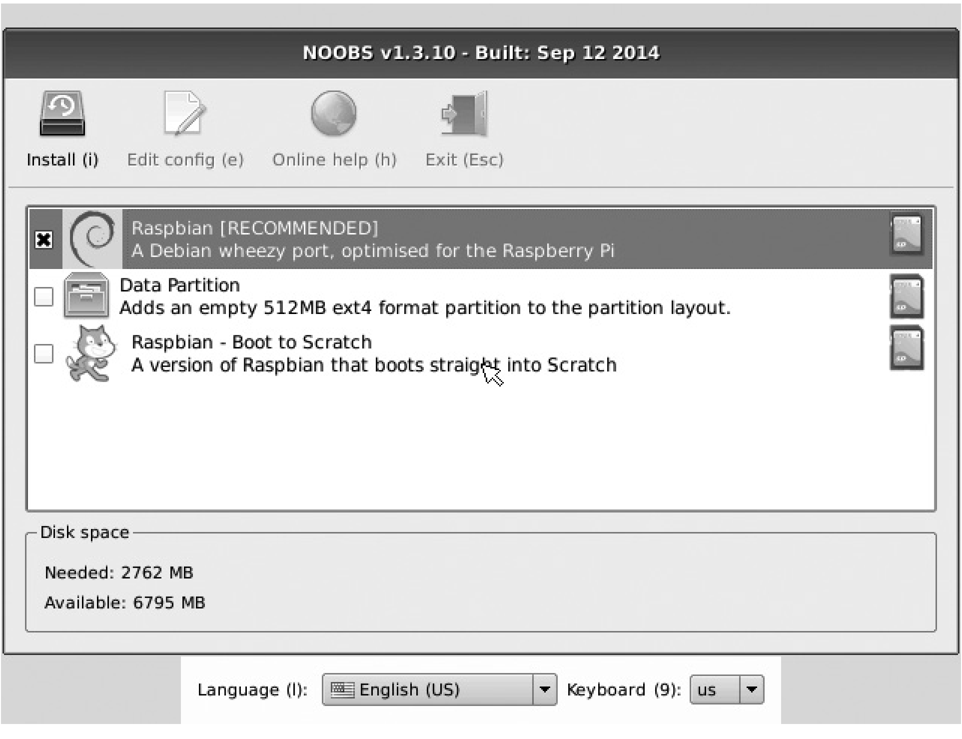


FIGURE 1-3. *NOOBS installer screen*

- If using HDMI, try safe mode. This is accomplished by pressing the number 2 on your keyboard in NOOBS. Safe mode forces a 640 × 480, 60-Hz resolution that most monitors can support. It also boosts the HDMI signal, which may help with long cables or high interference.
- If using composite, switch to PAL or NTSC. This is accomplished by pressing the number 3 (for PAL) or 4 (for NTSC) to switch to composite input and is only required when booting from NOOBS, which defaults to HDMI.

NOTE
When switching NOOBS video modes with your keyboard, make sure it is fully loaded (ACT LED should have stopped blinking). Also, enable NUMLOCK *if you are using the numeric keypad.*

Installing Raspbian with NOOBS

Raspbian is a Linux-based operating system that is a port of Debian and optimized for the Raspberry Pi. It was created by Mark Thompson and Peter Green and has been helped along by enthusiastic members of the Raspberry Pi community. It also comes with optimized Java installed right out of the box thanks to support from Oracle.

Picking up from the NOOBS installation screen shown in Figure 1-3 in the previous section, you will want to select Raspbian as your operating system and also set the correct locale and keyboard for your region. If you don't have a mouse connected, you can access the Language and Keyboard options via the keyboard by using the letter l and number 9 keys, respectively. By default the Raspberry Pi foundation sets the locale to the United Kingdom, which will leave you hopelessly lost on the command line as you attempt to type the pound symbol (#) or at sign (@) on a U.S. layout keyboard.

Figure 1-4 shows the number of Raspberry Pis by country as reported by Rastrack. While the United States has the highest Raspberry Pi sales by country, the UK wins with the most Pis per capita, so there is plenty of room for growth in the rest of the world!

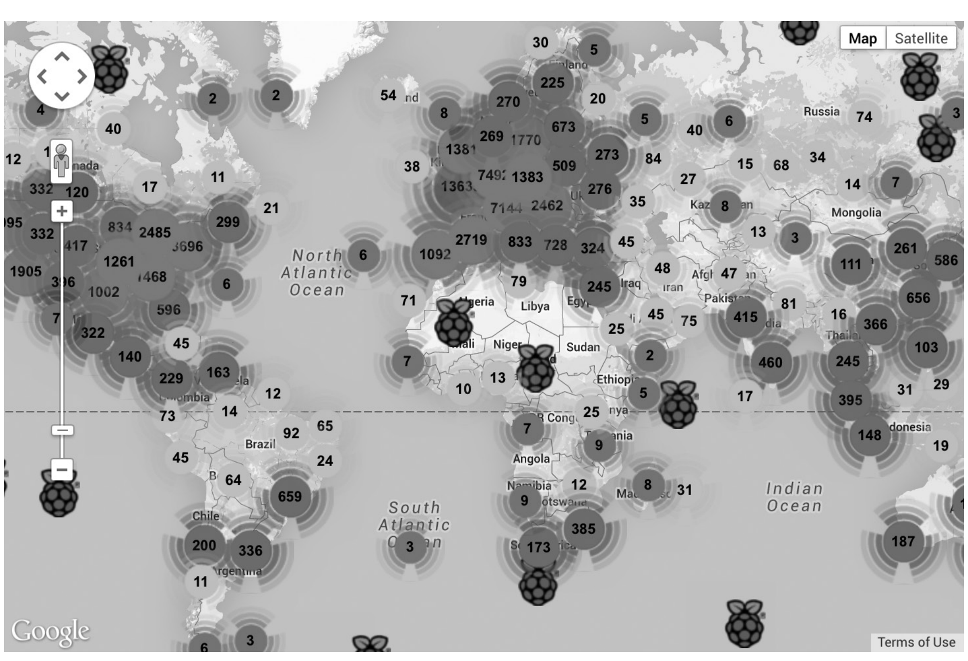


FIGURE 1-4. *Distribution of Raspberry Pis across the world*

After selecting the options, click the Install button or type the letter i to begin the installation process. Installation takes about 20 minutes, give or take a few minutes depending on the speed of your SD card. The installation screen has a pretty accurate progress bar at the bottom, as shown in Figure 1-5, and provides some helpful hints on the top for new Raspberry Pi owners. This is a good time to grab a cup of joe as you wait for the success screen to pop up.

Once you click the OK button or press ENTER, the Raspberry Pi will reboot and start up Raspbian for the first time. Raspbian has a typical Linux boot screen with lots of scrolling text, and a cute Raspberry Pi logo in the top-left corner.

Raspbian is set to automatically log in on your first boot and run the Raspberry Pi Software Configuration Tool (`raspi-config`), as shown in Figure 1-6. It is highly recommended to change the default password (as discussed in the following list), but if you need to log in before you get a chance to set it for some reason, the default username is pi and the default password is raspberry.

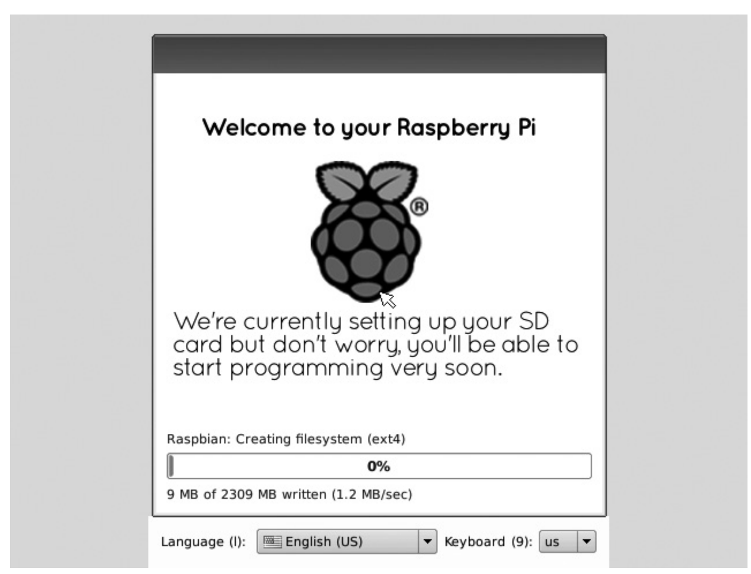


FIGURE 1-5. *Raspberry Pi installation screen*

```
Raspberry Pi Software Configuration Tool (raspi-config)
Setup Options

  1 Expand Filesystem               Ensures that all of the SD card s
  2 Change User Password            Change password for the default u
  3 Enable Boot to Desktop/Scratch  Choose whether to boot into a des
  4 Internationalisation Options    Set up language and regional sett
  5 Enable Camera                   Enable this Pi to work with the R
  6 Add to Rastrack                 Add this Pi to the online Raspber
  7 Overclock                       Configure overclocking for your P
  8 Advanced Options                Configure advanced settings
  9 About raspi-config              Information about this configurat

              <Select>                      <Finish>
```

FIGURE 1-6. *Raspberry Pi Software Configuration Tool*

Here is a quick rundown of what the different options do (recommended changes are noted):

- **Expand Filesystem** This lets you expand the root filesystem to fit the size of the SD card. If you used NOOBS to install, this has already been taken care of during the install.
- **Change User Password** Changing the password is highly recommended for security purposes. Anyone who can access the Raspberry Pi over the network will have root access if you don't change this.
- **Enable Boot to Desktop/Scratch** This lets you boot to a graphical user interface (GUI) with X Window System and optionally open Scratch for visual programming. This book steers you to using the command line, but anytime you want to open X Window System, you can type **startx**.
- **Internationalisation Options** If you forgot to change this during the NOOBS install, you have another chance to rescue your keyboard layout.
- **Enable Camera** This enables support for the Pi Camera and is a recommended setting.
- **Add to Rastrack** This adds your Pi to a worldwide list of Raspberry Pi locations on a map. It is fun to join in and provides valuable statistics on the Raspberry Pi community as you discovered earlier. Since this requires an Internet connection, you may want to revisit this option after completing the upcoming "Networking Your Raspberry Pi" section.
- **Overclock** The default speed of the Raspberry Pi's processor is 700 MHz for the A, B, A+, and B+, and 900 MHz for the Raspberry Pi 2. You can optionally raise this; however, it is recommend to start with the default speed. Overclocking the Pi may result in it running hotter and shortening the life of its components.
- **Advanced Options** Discussed in more detail in the next list.
- **About raspi-config** This displays an information screen about the Raspberry Pi.

Selecting Advanced Options brings up a submenu with the following additional options:

- **Overscan** This allows you to enable or disable overscan. If you have a modern LCD, you can safely disable this and get a little more screen real estate at the edges of your monitor.
- **Hostname** Feel free to change your hostname to be unique.
- **Memory Split** The memory on the Raspberry Pi is shared between the CPU and the GPU. To improve performance of graphics-intensive applications, I recommend setting GPU memory to at least 128MB.
- **SSH** This option is for Secure Shell, which is enabled by default, and is required for deployment of Java apps.
- **SPI** This is general purpose input/output (GPIO) functionality that needs to be enabled for some of the example projects in later chapters.
- **I2C** Another GPIO feature for managing a connected bus of devices, this option also needs to be enabled to support projects in later chapters.
- **Serial** This enables shell access over serial, although you will need to disable it to free up the serial ports for a later project.
- **Audio** This lets you force audio to go out over the HDMI or 3.5mm headphone jacks.
- **Update** This updates the `raspi-config` tool to the latest version.

On the list of options, the recommended changes are to change your password, enable the Pi Camera, set/confirm the memory split to 128MB, enable SPI, enable I2C, and disable serial. If you forget to do any of these steps, don't worry; I will remind you in future sections when the required functionality is needed and instruct you to enable it if you haven't already.

Once you are done making configuration changes, press the TAB key and select Finish. This will reboot your Raspberry Pi and give you your first login prompt as shown in Figure 1-7. To log in, type the username **pi** and the new password you chose.

```
[ ok ] Configuring network interfaces...done.
[ ok ] Cleaning up temporary files....
[ ok ] Setting up ALSA...done.
[info] Setting console screen modes.
[info] Skipping font and keymap setup (handled by console-setup).
[ ok ] Setting up console font and keymap...done.
[ ok ] Setting up X socket directories... /tmp/.X11-unix /tmp/.ICE-unix.
INIT: Entering runlevel: 2
[info] Using makefile-style concurrent boot in runlevel 2.
[ ok ] Network Interface Plugging Daemon...skip eth0...done.
[info] Initializing cgroups.
[....] Kernel lacks cgroups or memory controller not available, not starting cgr
[warn ... (warning).
[ ok ] Starting enhanced syslogd: rsyslogd.
[ ok ] Starting periodic command scheduler: cron.
[ ok ] Starting system message bus: dbus.
Starting dphys-swapfile swapfile setup ...
want /var/swap=100MByte, checking existing: keeping it
done.
[ ok ] Starting NTP server: ntpd.
[ ok ] Starting OpenBSD Secure Shell server: sshd.

Raspbian GNU/Linux 7 raspberrypi tty1

raspberrypi login:
```

FIGURE 1-7. *Raspberry Pi login prompt*

If you need to bring up the `raspi-config` utility again, you can always do this from the command line by typing

```
sudo raspi-config
```

This is probably also a good time to mention the correct way to shut down your Raspberry Pi. If you disconnect power from the Raspberry Pi while it is running, you may damage the filesystem and cause corruption and data loss. To prevent this, make sure that you properly halt the Raspberry Pi before powering it off by using the following command:

```
sudo shutdown -h now
```

This command logs off all users, cleanly closes the filesystem, and terminates before you power off the Pi. You will know your Pi is ready to unplug when you see the green ACT LED flash ten times in sequence.

To reboot you can use a similar command:

```
sudo shutdown -r now
```

As a shortcut you may see some Raspberry Pi users use the `halt` and `reboot` commands. These behave as expected and are perfectly safe on the Raspberry Pi, but they are not best practices when you are administering a variety of Unix operating systems, because the behavior varies.

Networking Your Raspberry Pi

To communicate from your computer to the Raspberry Pi, you will have to put your Raspberry Pi and computer on the same network so that your Pi is accessible via TCP/IP. This is also the easiest networking option for Models A and A+ that lack an Ethernet port. There are several different ways to do this depending on the physical location of your computer, the network topology, and your available hardware.

Connecting via Ethernet

If you have a router that acts as a Dynamic Host Configuration Protocol (DHCP) server, you can simply plug the Raspberry Pi into your network using an Ethernet cable. This only works for Raspberry Pi Models B, B+, and 2, because Models A and A+ lack an Ethernet port.

Once connected, the Raspberry Pi will automatically try to get a network address from the DHCP server. You can check for the IP address that the Raspberry Pi acquired by typing the following command:

```
ip addr show eth0
```

Connecting via a Local Computer Network

You can also connect your Raspberry Pi directly to your PC using an Ethernet cable. Again, this option is only available for Raspberry Pi Models B, B+, and 2, but can be a great alternative if you are traveling or in a setting where the network topology doesn't allow your computer and Raspberry Pi to talk.

TIP

The Raspberry Pi Ethernet adapter includes auto-MDIX to detect and fix cable types, so you can use either a crossover cable or a more common straight Ethernet cable to connect devices.

The easiest way to accomplish this is to assign static IP addresses to both your computer and the Raspberry Pi so they are both in the same subnet. A common local subnet to use is 192.168.x.x, which is one of the reserved subnets for local area networks. The configuration on your desktop computer will look something like Figure 1-8 for OS X or Figure 1-9 for Windows.

On the Raspberry Pi you will need to modify the cmdline.txt file in the boot folder. To do so, log in to the Raspberry Pi with the following command:

```
sudo nano /boot/cmdline.txt
```

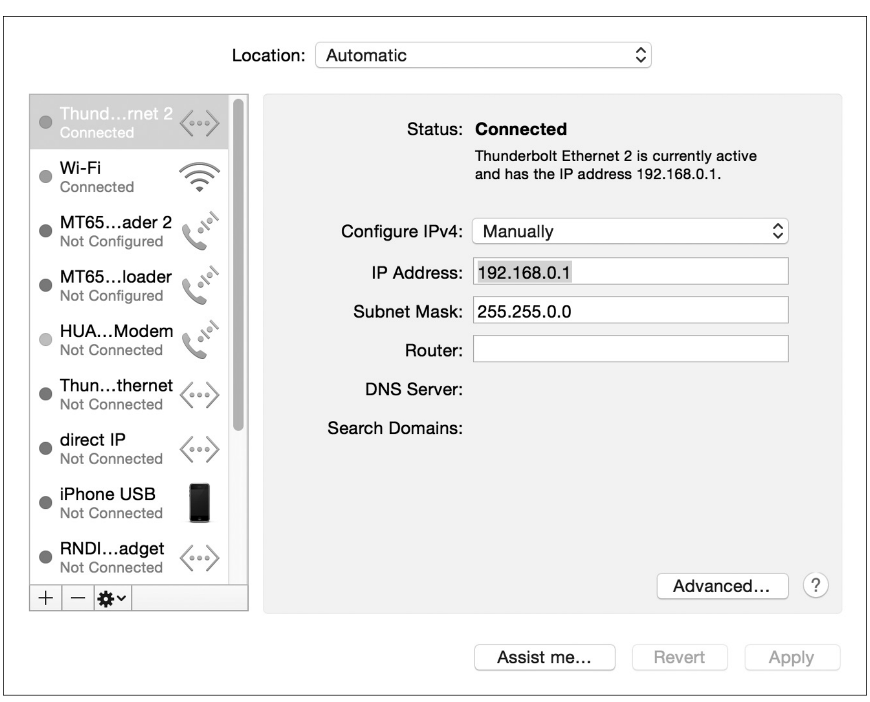


FIGURE 1-8. *Static IP configuration in OS X*

FIGURE 1-9. *Static IP configuration in Windows*

Nano is a simple command-line editor that allows you to edit text files on Unix systems. When you open the cmdline.txt file, you will get an editing screen as shown in Figure 1-10. Scroll to the end of the line using the arrow keys and type **ip=192.168.0.2** (or a similar local IP address). Make sure to leave a space after the last parameter (most likely, rootwait) and do not add any carriage returns.

After rebooting your Pi, it will start up with the new IP address fixed, and will be accessible from your computer with that IP address.

Connecting via a Wireless Network

A great option for networking both Raspberry Pi B and A variants is to use a Wi-Fi adapter. This allows you to connect the Raspberry Pi to a wireless network and access it from your computer remotely.

For this you will need a compatible Raspberry Pi Wi-Fi USB adapter. In general, Wi-Fi devices utilizing the RTL8188CUS chipset are well supported

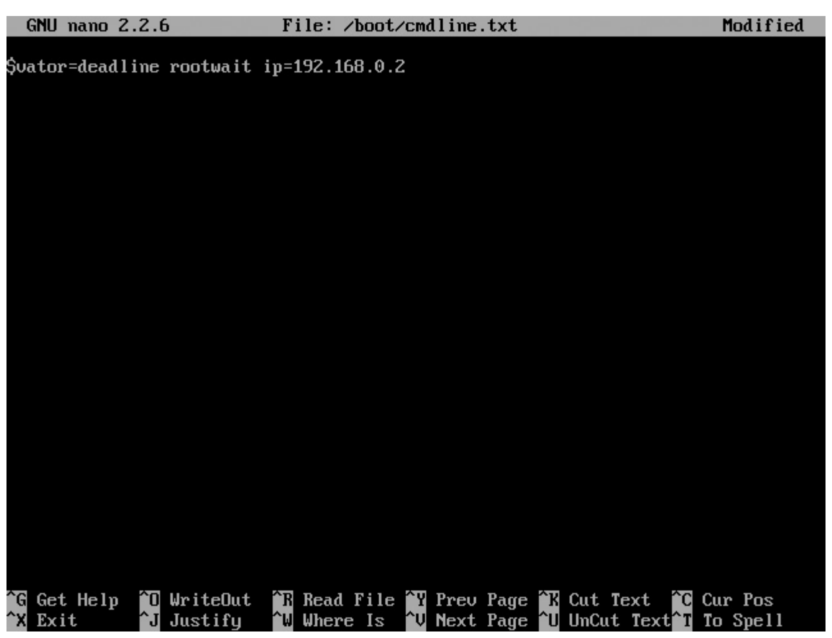


FIGURE 1-10. *Setting a static IP address on the Raspberry Pi*

on the Raspberry Pi. Often you will find certified Wi-Fi devices sold alongside Raspberry Pis at vendor websites, but you may be able to find a cheaper or faster Wi-Fi adapter with a little bit of research. For a full list of devices that are known to work by the community, check the eLinux Wi-Fi adapter listing here: http://elinux.org/RPi_USB_Wi-Fi_Adapters.

Raspbian comes with wpa_supplicant installed and set up for a wireless network, so all you have to do is add your network configuration options from the command line. To do this, I recommend using the WPA command-line tool (wpa_cli), which lets you scan your network and add new wpa_supplicant configurations. The advantage of using the command-line tool over editing the configuration file directly is that you can't make an error in your configuration by missing punctuation or formatting.

Listing 1-1 shows an example of how to use wpa_cli to configure your wireless settings. Obviously, you should replace ssid and psk with the SSID and preshared key of your local network configuration, and this assumes that you are using a network that broadcasts the SSID. Both WPA and WPA2 networks are supported by this configuration.

Listing 1-1 *WPA command-line tool for configuring Wi-Fi*

```
pi@raspberrypi ~ $ wpa_cli
Selected interface 'wlan0'
Interactive mode
> scan
OK
<3>CTRL-EVENT-SCAN-RESULTS
<3>WPS-AP-AVAILABLE
> scan_results
bssid / frequency / signal level / flags / ssid
12:0d:7f:8b:be:9e 2437 92 [WPA2-PSK-CCMP][ESS] NightHacking-Guest
> add_network
0
> set_network 0 ssid "NightHacking-Guest"
OK
> set_network 0 psk "steveonjava"
OK
> enable_network 0
OK
> save_config
OK
> reconnect
OK
> quit
```

TIP

If you are still using WEP, it is possible to connect your Raspberry Pi, but I don't recommend it. WEP has been proven insecure and can be cracked in under a minute by low-end hardware and freely available software. There are also some new cryptographic attacks against WPA involving vulnerabilities in TKIP. In short, upgrading your network to WPA2 is an important security practice.

Updating and Upgrading

Now that you are on the network, the very first thing you should do is to update your Raspbian distribution. This will ensure you have the latest package listing and current versions of all of the core files. To do this, first execute the following command to download the latest package listing:

```
sudo apt-get update
```

Then you can perform an upgrade of your Raspberry Pi distribution by using this additional command:

```
sudo apt-get upgrade
```

Depending upon how old the NOOBS distribution you originally used was, and how fast your network connection and SD card are, this could take quite a while. This might be a good opportunity to brew another cup of coffee.

Setting Up a Hostname

If your Raspberry Pi gets its IP address from DHCP, the address can change on every reboot. If you are running the Raspberry Pi headless (without a monitor or display), this can make it a chore to search for the new IP address. A good alternative is to use Bonjour/Zeroconf, which broadcasts your hostname over multicast. This way you can refer to your Raspberry Pi as raspberrypi.local (or, in general, hostname.local) from anywhere on your local network.

The first step is to set a unique hostname. This can be done from the Advanced Options in the Raspberry Pi configuration utility. To bring up the configuration utility from the command line, type the following:

```
sudo raspi-config
```

After setting the hostname, you will be asked to reboot the Pi to update the network configuration. After reboot, you can install Bonjour on the Raspberry Pi by running the following command:

```
sudo apt-get install libnss-mdns
```

After this command completes, you are ready to access the Pi on the network. From any computer on the same network where multicast packets reach, you can replace the IP address of your Pi with hostname.local. For example, Listing 1-2 shows the output of pinging my Raspberry Pi with hostname nighthackingpi.

Listing 1-2 *Pinging nighthackingpi via Bonjour*

```
NightHacking-Presenter:~ sjc$ ping nighthackingpi.local
PING nighthackingpi.local (192.168.1.10): 56 data bytes
64 bytes from 192.168.1.10: icmp_seq=0 ttl=64 time=76.379 ms
64 bytes from 192.168.1.10: icmp_seq=1 ttl=64 time=93.390 ms
```

Notice that it automatically translates from the hostname to an IP address of 192.168.1.10. However, if this address changed in the future, I could use the same command to access my Pi.

Bonjour is installed by default on OS X and Ubuntu Linux. If you are running on Windows, you already have Bonjour installed if you have previously installed iTunes. Otherwise, the easiest way to get it is to install Bonjour Print Services for Windows from Apple: http://support.apple.com/kb/DL999.

Connecting to Your Raspberry Pi with SSH

Using an SSH client from your computer is a convenient and secure way of interacting with your Raspberry Pi. Once you have networking configured on both machines, this is as simple as connecting with the hostname or IP address.

For Unix or OS X you can simply use a terminal window and the version of SSH that ships with your operating system. Figure 1-11 shows an example of an SSH login from an OS X computer.

```
sjc — pi@raspberrypi: ~ — ssh — 80×23 — ⌘1
NightHacking-Presenter:~ sjc$
NightHacking-Presenter:~ sjc$
NightHacking-Presenter:~ sjc$
NightHacking-Presenter:~ sjc$ ssh pi@192.168.1.10
The authenticity of host '192.168.1.10 (192.168.1.10)' can't be established.
RSA key fingerprint is 0c:b4:c5:3c:1c:2b:8e:ef:fe:97:26:18:a1:33:1b:bf.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.10' (RSA) to the list of known hosts.
pi@192.168.1.10's password:
Linux raspberrypi 3.12.35+ #730 PREEMPT Fri Dec 19 18:31:24 GMT 2014 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jan 13 02:11:13 2015 from 192.168.1.164
pi@raspberrypi ~ $ java -version
java version "1.8.0"
Java(TM) SE Runtime Environment (build 1.8.0-b132)
Java HotSpot(TM) Client VM (build 25.0-b70, mixed mode)
pi@raspberrypi ~ $
```

FIGURE 1-11. *SSH from OS X*

To connect via SSH on the command line, simply issue the following ssh command:

```
ssh user@hostname
```

where "user" is your username (most likely, pi) and "hostname" is your Pi's network name or IP address (for example, 192.168.0.2).

If this is the first time you are connecting, you may be asked to verify the RSA key fingerprint. This is a security measure to ensure that the device to which you are creating an encrypted connection is in fact the device you intended to communicate with. If your network has been compromised (or you are on a public network), then it is possible for someone to launch a man-in-the-middle attack and spoof as your device.

To verify the RSA key fingerprint, physically log on to the Raspberry Pi and type the following command:

```
ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key.pub
```

This returns a fingerprint that you can verify against the one returned by the SSH tunnel, which will look something like the following:

```
2048 0c:b4:c5:3c:1c:2b:8e:ef:fe:97:26:18:a1:33:1b:bf  root@raspberrypi (RSA)
```

CAUTION
Checking the RSA key fingerprint after logging on to the Raspberry Pi is as good as not checking at all. Once someone else has established a man-in-the-middle attack, they can simply intercept the command and return a matching fingerprint.

Now that the connection has been established as secure, SSH will ask for your password. Once authenticated, you can issue commands just as if you were physically at the keyboard. This is often more convenient, and it lets you interact with a headless Raspberry Pi to do redeployment, diagnostics, or troubleshooting.

On Windows you will have to install an SSH client yourself. A well-known and free SSH client is PuTTY, which is maintained by a small team based in Cambridge, England. You can find the PuTTY downloads here: www.chiark.greenend.org.uk/~sgtatham/putty/download.html.

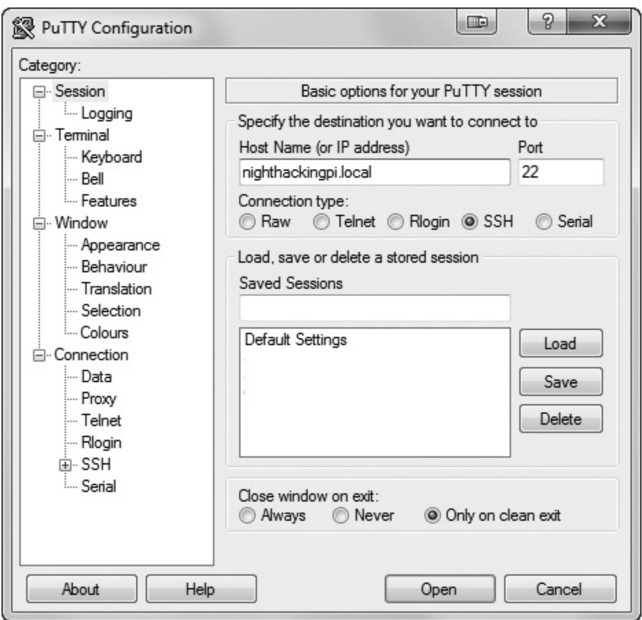


FIGURE 1-12. *PuTTY SSH client for Windows*

The initial configuration screen of PuTTY is shown in Figure 1-12. Simply enter the IP address or hostname of your Raspberry Pi, make sure SSH is selected, and click Open. This will start a secure connection that prompts you to verify the RSA key fingerprint (as just discussed) and then lets you connect with your username and password.

Creating a Simple Raspberry Pi Application

Now that you have a convenient SSH prompt to access your Raspberry Pi, you are ready to try running Java remotely. In the next chapter you will install a full-featured integrated development environment (IDE) to speed up development, but for this simple `HelloRaspberryPi` application, it is easy enough to type it in on the command line.

To create the application, you use the echo and append (>) commands to generate a simple Java class. Listing 1-3 shows the commands (in bold) you type into the SSH sessions.

Listing 1-3 *Creation of the `HelloRaspberryPi` class*

```
pi@nighthackingpi ~ $ echo "class HelloRaspberryPi {
>   public static void main(String[] args) {
>     System.out.println(\"Hello Raspberry Pi\");
>   }
> }" > HelloRaspberryPi.java
```

Notice that you can continue a command within quotation marks on the next line simply by pressing ENTER. The command prompt (>) on each line is automatically typed by the system, and in this example I used spaces for indentation. The only other difference from normal Java code is that the double quotes (" ") need to be escaped with a preceding backslash (\).

The last line writes this application to a file called HelloRaspberryPi.java that you can compile by using `javac` with the following command:

```
pi@nighthackingpi ~ $ javac HelloRaspberryPi.java
```

Executing the application is as simple as running `java` in the same directory with the main class name:

```
pi@nighthackingpi ~ $ java HelloRaspberryPi
```

My shell console is shown in Figure 1-13 along with the output of the program.

```
sjc — pi@nighthackingpi: ~ — ssh — 56×11 — ⌘1
pi@nighthackingpi ~ $ echo "class HelloRaspberryPi {
>   public static void main(String[] args) {
>     System.out.println(\"Hello Raspberry Pi\");
>   }
> }" > HelloRaspberryPi.java
pi@nighthackingpi ~ $ javac HelloRaspberryPi.java
pi@nighthackingpi ~ $ java HelloRaspberryPi
Hello Raspberry Pi
pi@nighthackingpi ~ $
```

FIGURE 1-13. *Output of the `HelloRaspberryPi` application*

Congratulations on setting up your first Raspberry Pi and running a simple Java application on it! The work you completed in this chapter on hardware, configuration, and networking has set the foundation for the rest of your Raspberry Pi projects. In the next chapter we will explore the visual capabilities of the Raspberry Pi and set up a full Java IDE to streamline future projects.

6

Flow Control and Exceptions

CERTIFICATION OBJECTIVES

- Use if and switch Statements
- Develop for, do, and while Loops
- Use break and continue Statements
- Use try, catch, and finally Statements
- State the Effects of Exceptions
- Recognize Common Exceptions

✓ Two-Minute Drill

Q&A Self Test

Can you imagine trying to write code using a language that didn't give you a way to execute statements conditionally? Flow control is a key part of most any useful programming language, and Java offers several ways to accomplish it. Some statements, such as `if` statements and `for` loops, are common to most languages. But Java also throws in a couple of flow control features you might not have used before—exceptions and assertions. (We'll discuss assertions in the next chapter.)

The `if` statement and the `switch` statement are types of conditional/decision controls that allow your program to behave differently at a "fork in the road," depending on the result of a logical test. Java also provides three different looping constructs—`for`, `while`, and `do`—so you can execute the same code over and over again depending on some condition being true. Exceptions give you a clean, simple way to organize code that deals with problems that might crop up at runtime.

With these tools, you can build a robust program that can handle any logical situation with grace. Expect to see a wide range of questions on the exam that include flow control as part of the question code, even on questions that aren't testing your knowledge of flow control.

CERTIFICATION OBJECTIVE

Using if and switch Statements (OCA Objectives 3.4 and 3.5—also Upgrade Objective 1.1)

3.4 Create if and if-else constructs.

3.5 Use a switch statement.

The `if` and `switch` statements are commonly referred to as decision statements. When you use decision statements in your program, you're asking the program to evaluate a given expression to determine which course of action to take. We'll look at the `if` statement first.

if-else Branching

The basic format of an `if` statement is as follows:

```
if (booleanExpression) {
  System.out.println("Inside if statement");
}
```

The expression in parentheses must evaluate to (a `boolean`) `true` or `false`. Typically you're testing something to see if it's true, and then running a code block (one or more statements) if it is true and (optionally) another block of code if it isn't. The following code demonstrates a legal `if-else` statement:

```
if (x > 3) {
  System.out.println("x is greater than 3");
} else {
  System.out.println("x is not greater than 3");
}
```

The `else` block is optional, so you can also use the following:

```
if (x > 3) {
  y = 2;
}
z += 8;
a = y + x;
```

The preceding code will assign 2 to `y` if the test succeeds (meaning `x` really is greater than 3), but the other two lines will execute regardless. Even the curly braces are optional if you have only one statement to execute within the body of the conditional block. The following code example is legal (although not recommended for readability):

```
if (x > 3)    // bad practice, but seen on the exam
  y = 2;
z += 8;
a = y + x;
```

Most developers consider it good practice to enclose blocks within curly braces, even if there's only one statement in the block. Be careful with code like the preceding, because you might think it should read as

"If `x` is greater than 3, then set `y` to 2, `z` to `z` + 8, and `a` to `y` + `x`."

But the last two lines are going to execute no matter what! They aren't part of the conditional flow. You might find it even more misleading if the code were indented as follows:

```
if (x > 3)
  y = 2;
  z += 8;
  a = y + x;
```

You might have a need to nest `if-else` statements (although, again, it's not recommended for readability, so nested `if` tests should be kept to a minimum). You can set up an `if-else` statement to test for multiple conditions. The following

example uses two conditions so that if the first test fails, we want to perform a second test before deciding what to do:

```
if (price < 300) {
  buyProduct();
} else {
  if (price < 400) {
    getApproval();
  }
  else {
    dontBuyProduct();
  }
}
```

This brings up the other `if-else` construct, the `if`, `else if`, `else`. The preceding code could (and should) be rewritten like this:

```
if (price < 300) {
  buyProduct();
} else if (price < 400) {
    getApproval();
} else {
    dontBuyProduct();
}
```

There are a couple of rules for using `else` and `else if`:

- You can have zero or one `else` for a given `if`, and it must come after any `else ifs`.
- You can have zero to many `else ifs` for a given `if` and they must come before the (optional) `else`.
- Once an `else if` succeeds, none of the remaining `else ifs` nor the `else` will be tested.

The following example shows code that is horribly formatted for the real world. As you've probably guessed, it's fairly likely that you'll encounter formatting like this on the exam. In any case, the code demonstrates the use of multiple `else ifs`:

```
int x = 1;
if ( x == 3 ) { }
else if (x < 4) {System.out.println("<4"); }
else if (x < 2) {System.out.println("<2"); }
else { System.out.println("else"); }
```

It produces this output:

```
<4
```

(Notice that even though the second `else if` is true, it is never reached.)

Sometimes you can have a problem figuring out which `if` your `else` should pair with, as follows:

```
if (exam.done())
if (exam.getScore() < 0.61)
System.out.println("Try again.");
// Which if does this belong to?
else System.out.println("Java master!");
```

We intentionally left out the indenting in this piece of code so it doesn't give clues as to which `if` statement the `else` belongs to. Did you figure it out? Java law decrees that an `else` clause belongs to the innermost `if` statement to which it might possibly belong (in other words, the closest preceding `if` that doesn't have an `else`). In the case of the preceding example, the `else` belongs to the second `if` statement in the listing. With proper indenting, it would look like this:

```
if (exam.done())
  if (exam.getScore() < 0.61)
    System.out.println("Try again.");
  // Which if does this belong to?
  else
    System.out.println("Java master!");
```

Following our coding conventions by using curly braces, it would be even easier to read:

```
if (exam.done()) {
  if (exam.getScore() < 0.61) {
    System.out.println("Try again.");
  // Which if does this belong to?
  } else {
    System.out.println("Java master!");
  }
}
```

Don't get your hopes up about the exam questions being all nice and indented properly. Some exam takers even have a slogan for the way questions are presented on the exam: Anything that can be made more confusing, will be.

Be prepared for questions that not only fail to indent nicely, but intentionally indent in a misleading way. Pay close attention for misdirection like the following:

```
if (exam.done())
  if (exam.getScore() < 0.61)
    System.out.println("Try again.");
else
  System.out.println("Java master!"); // Hmmmmm… now where does
                                      // it belong?
```

Of course, the preceding code is exactly the same as the previous two examples, except for the way it looks.

Legal Expressions for if Statements

The expression in an `if` statement must be a `boolean` expression. Any expression that resolves to a `boolean` is fine, and some of the expressions can be complex. Assume `doStuff()` returns `true`,

```
int y = 5;
int x = 2;
if (((x > 3) && (y < 2)) | doStuff()) {
  System.out.println("true");
}
```

which prints

```
true
```

You can read the preceding code as, "If both `(x > 3)` and `(y < 2)` are `true`, or if the result of `doStuff()` is `true`, then print `true`." So, basically, if just `doStuff()` alone is `true`, we'll still get `true`. If `doStuff()` is `false`, though, then both `(x > 3)` and `(y < 2)` will have to be `true` in order to print `true`. The preceding code is even more complex if you leave off one set of parentheses as follows:

```
int y = 5;
int x = 2;
if ((x > 3) && (y < 2) | doStuff()) {
  System.out.println("true");
}
```

This now prints…nothing! Because the preceding code (with one less set of parentheses) evaluates as though you were saying, "If `(x > 3)` is `true`, and either `(y < 2)` or the result of `doStuff()` is `true`, then print `true`. So if `(x > 3)` is not `true`, no point in looking at the rest of the expression." Because of the short-circuit `&&`, the expression is evaluated as though there were parentheses around `(y < 2) | doStuff()`. In other words, it is evaluated as a single expression before the `&&` and a single expression after the `&&`.

Remember that the only legal expression in an `if` test is a `boolean`. In some languages, 0 == false, and 1 == true. Not so in Java! The following code shows `if` statements that might look tempting but are illegal, followed by legal substitutions:

```
int trueInt = 1;
int falseInt = 0;
if (trueInt)              // illegal
if (trueInt == true)      // illegal
if (1)                    // illegal
if (falseInt == false)    // illegal
if (trueInt == 1)         // legal
if (falseInt == 0)        // legal
```

exam watch

One common mistake programmers make (and that can be difficult to spot), is assigning a `boolean` variable when you meant to test a `boolean` variable. Look out for code like the following:

```
boolean boo = false;
if (boo = true) { }
```

You might think one of three things:

1. ***The code compiles and runs fine, and the `if` test fails because `boo` is `false`.***
2. ***The code won't compile because you're using an assignment (=) rather than an equality test (==).***
3. ***The code compiles and runs fine, and the `if` test succeeds because `boo` is SET to `true` (rather than TESTED for `true`) in the `if` argument!***

Well, number 3 is correct—pointless, but correct. Given that the result of any assignment is the value of the variable after the assignment, the expression `(boo = true)` has a result of `true`. Hence, the `if` test succeeds. But the only variables that can be assigned (rather than tested against something else) are a `boolean` or a `Boolean`; all other assignments will result in something non-`boolean`, so they're not legal, as in the following:

```
int x = 3;
if (x = 5) { }  // Won't compile because x is not a boolean!
```

Because `if` tests require `boolean` expressions, you need to be really solid on both logical operators and `if` test syntax and semantics.

switch Statements (OCA, OCP, and Upgrade Topic)

You've seen how `if` and `else-if` statements can be used to support both simple and complex decision logic. In many cases, the `switch` statement provides a cleaner way to handle complex decision logic. Let's compare the following `if-else if` statement to the equivalently performing `switch` statement:

```
int x = 3;
if(x == 1) {
  System.out.println("x equals 1");
}
else if(x == 2) {
  System.out.println("x equals 2");
}
else {
  System.out.println("No idea what x is");
}
```

Now let's see the same functionality represented in a `switch` construct:

```
int x = 3;
switch (x) {
  case 1:
    System.out.println("x equals 1");
    break;
  case 2:
    System.out.println("x equals 2");
    break;
  default:
    System.out.println("No idea what x is");
}
```

Note: The reason this `switch` statement emulates the `if` is because of the `break` statements that were placed inside of the `switch`. In general, `break` statements are optional, and as you will see in a few pages, their inclusion or exclusion causes huge changes in how a `switch` statement will execute.

Legal Expressions for switch and case

The general form of the `switch` statement is

```
switch (expression) {
  case constant1: code block
  case constant2: code block
  default: code block
}
```

A `switch`'s expression must evaluate to a `char`, `byte`, `short`, `int`, an `enum` (as of Java 5), and a `String` (as of Java 7). That means if you're not using an `enum` or a `String`, only variables and values that can be automatically promoted (in other words, implicitly cast) to an `int` are acceptable. You won't be able to compile if you use anything else, including the remaining numeric types of `long`, `float`, and `double`.

Note: For OCA candidates, `enum`s are not covered on your exam, and you won't encounter any questions related to `switch` statements that use `enum`s.

A `case` constant must evaluate to the same type that the `switch` expression can use, with one additional—and big—constraint: the `case` constant must be a compile-time constant! Since the `case` argument has to be resolved at compile time, you can use only a constant or `final` variable that is immediately initialized with a literal value. It is not enough to be `final`; it must be a compile time *constant*. Here's an example:

```
final int a = 1;
final int b;
b = 2;
int x = 0;
switch (x) {
  case a:     // ok
  case b:     // compiler error
```

Also, the `switch` can only check for equality. This means that the other relational operators such as greater than are rendered unusable in a `case`. The following is an example of a valid expression using a method invocation in a `switch` statement. Note that for this code to be legal, the method being invoked on the object reference must return a value compatible with an `int`.

```
String s = "xyz";
switch (s.length()) {
  case 1:
    System.out.println("length is one");
    break;
  case 2:
    System.out.println("length is two");
    break;
  case 3:
    System.out.println("length is three");
    break;
  default:
    System.out.println("no match");
}
```

One other rule you might not expect involves the question, "What happens if I `switch` on a variable smaller than an `int`?" Look at the following `switch`:

```
byte g = 2;
switch(g) {
  case 23:
  case 128:
}
```

This code won't compile. Although the `switch` argument is legal—a byte is implicitly cast to an `int`—the second `case` argument (128) is too large for a `byte`, and the compiler knows it! Attempting to compile the preceding example gives you an error something like this:

```
Test.java:6: possible loss of precision
found   : int
required: byte
    case 128:
         ^
```

It's also illegal to have more than one `case` label using the same value. For example, the following block of code won't compile because it uses two `cases` with the same value of 80:

```
int temp = 90;
switch(temp) {
  case 80 :  System.out.println("80");
  case 80 :  System.out.println("80");   // won't compile!
  case 90 :  System.out.println("90");
  default :  System.out.println("default");
}
```

It *is* legal to leverage the power of boxing in a `switch` expression. For instance, the following is legal:

```
switch(new Integer(4)) {
  case 4: System.out.println("boxing is OK");
}
```

exam watch

Look for any violation of the rules for `switch` and `case` arguments. For example, you might find illegal examples like the following snippets:

```
switch(x) {
  case 0 {
    y = 7;
  }
}

switch(x) {
  0: { }
  1: { }
}
```

In the first example, the `case` uses a curly brace and omits the colon. The second example omits the keyword `case`.

An Intro to String "equality"

As we've been discussing, the operation of `switch` statements depends on the expression "matching" or being "equal" to one of the cases. We've talked about how we know when primitives are equal, but what does it mean for objects to be equal? This is another one of those surprisingly tricky topics, and for those of you who

intend to take the OCP exam, we'll spend a lot of time discussing "object equality" in Part II. For you OCA candidates, all you have to know is that for a `switch` statement, two `String`s will be considered "equal" if they have the same case-sensitive sequence of characters. For example, in the following partial `switch` statement, the expression would match the case:

```
String s = "Monday";
switch(s) {
  case "Monday":   // matches!
```

But the following would NOT match:

```
String s = "MONDAY";
switch(s) {
  case "Monday":   // Strings are case-sensitive, DOES NOT match
```

Break and Fall-Through in switch Blocks

We're finally ready to discuss the `break` statement and offer more details about flow control within a `switch` statement. The most important thing to remember about the flow of execution through a `switch` statement is this:

> `case` constants are evaluated from the top down, and the first `case` constant that matches the `switch`'s expression is the execution *entry point*.

In other words, once a `case` constant is matched, the Java Virtual Machine (JVM) will execute the associated code block and ALL subsequent code blocks (barring a `break` statement) too! The following example uses a `String` in a `case` statement:

```
class SwitchString {
  public static void main(String [] args) {
    String s = "green";
    switch(s) {
      case "red": System.out.print("red ");
      case "green": System.out.print("green ");
      case "blue": System.out.print("blue ");
      default: System.out.println("done");
    }
  }
}
```

In this example `case "green":` matched, so the JVM executed that code block and all subsequent code blocks to produce the output:

```
green blue done
```

Again, when the program encounters the keyword `break` during the execution of a `switch` statement, execution will immediately move out of the `switch` block to

the next statement after the `switch`. If `break` is omitted, the program just keeps executing the remaining `case` blocks until either a `break` is found or the `switch` statement ends. Examine the following code:

```
int x = 1;
switch(x) {
  case 1:  System.out.println("x is one");
  case 2:  System.out.println("x is two");
  case 3:  System.out.println("x is three");
}
System.out.println("out of the switch");
```

The code will print the following:

```
x is one
x is two
x is three
out of the switch
```

This combination occurs because the code didn't hit a `break` statement; execution just kept dropping down through each `case` until the end. This dropping down is actually called "fall-through," because of the way execution falls from one `case` to the next. Remember, the matching `case` is simply your entry point into the `switch` block! In other words, you must *not* think of it as, "Find the matching `case`, execute just that code, and get out." That's *not* how it works. If you do want that "just the matching code" behavior, you'll insert a `break` into each `case` as follows:

```
int x = 1;
switch(x) {
  case 1:  {
    System.out.println("x is one");  break;
  }
  case 2:  {
    System.out.println("x is two");  break;
  }
  case 3:  {
    System.out.println("x is two");  break;
  }
}
System.out.println("out of the switch");
```

Running the preceding code, now that we've added the `break` statements, will print this:

```
x is one
out of the switch
```

And that's it. We entered into the `switch` block at `case 1`. Because it matched the `switch()` argument, we got the `println` statement and then hit the `break` and jumped to the end of the `switch`.

An interesting example of this fall-through logic is shown in the following code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
  case 2:
  case 4:
  case 6:
  case 8:
  case 10: {
    System.out.println("x is an even number");  break;
  }
}
```

This `switch` statement will print `x is an even number` or nothing, depending on whether the number is between one and ten and is odd or even. For example, if `x` is 4, execution will begin at `case 4`, but then fall down through `6`, `8`, and `10`, where it prints and then breaks. The `break` at `case 10`, by the way, is not needed; we're already at the end of the `switch` anyway.

Note: Because fall-through is less than intuitive, Oracle recommends that you add a comment such as `// fall through` when you use fall-through logic.

The Default Case

What if, using the preceding code, you wanted to print `x is an odd number` if none of the `cases` (the even numbers) matched? You couldn't put it after the `switch` statement, or even as the last `case` in the `switch`, because in both of those situations it would always print `x is an odd number`. To get this behavior, you'd use the `default` keyword. (By the way, if you've wondered why there is a `default` keyword even though we don't use a modifier for default access control, now you'll see that the `default` keyword is used for a completely different purpose.) The only change we need to make is to add the `default` case to the preceding code:

```
int x = someNumberBetweenOneAndTen;

switch (x) {
  case 2:
  case 4:
  case 6:
  case 8:
  case 10: {
    System.out.println("x is an even number");
    break;
  }
  default: System.out.println("x is an odd number");
}
```

The `default case` doesn't have to come at the end of the `switch`. Look for it in strange places such as the following:

```
int x = 2;
switch (x) {
  case 2:  System.out.println("2");
  default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

Running the preceding code prints this:

```
2
default
3
4
```

And if we modify it so that the only match is the `default case`, like this,

```
int x = 7;
switch (x) {
  case 2:  System.out.println("2");
  default: System.out.println("default");
  case 3: System.out.println("3");
  case 4: System.out.println("4");
}
```

then running the preceding code prints this:

```
default
3
4
```

The rule to remember is that `default` works just like any other `case` for fall-through!

EXERCISE 6-1

Creating a switch-case Statement

Try creating a `switch` statement using a `char` value as the `case`. Include a default behavior if none of the `char` values match.

- Make sure a `char` variable is declared before the `switch` statement.
- Each `case` statement should be followed by a `break`.
- The `default case` can be located at the end, middle, or top.

CERTIFICATION OBJECTIVE

Creating Loops Constructs (OCA Objectives 5.1, 5.2, 5.3, 5.4, and 5.5)

5.1 *Create and use while loops.*

5.2 *Create and use for loops including the enhanced for loop.*

5.3 *Create and use do/while loops.*

5.4 *Compare loop constructs.*

5.5 *Use break and continue.*

Java loops come in three flavors: `while`, `do`, and `for` (and as of Java 5, the `for` loop has two variations). All three let you repeat a block of code as long as some condition is true, or for a specific number of iterations. You're probably familiar with loops from other languages, so even if you're somewhat new to Java, these won't be a problem to learn.

Using while Loops

The `while` loop is good when you don't know how many times a block or statement should repeat, but you want to continue looping as long as some condition is true. A `while` statement looks like this:

```
while (expression) {
  // do stuff
}
```

Or this:

```
int x = 2;
while(x == 2) {
  System.out.println(x);
  ++x;
}
```

In this case, as in all loops, the expression (test) must evaluate to a `boolean` result. The body of the `while` loop will execute only if the expression (sometimes called the "condition") results in a value of `true`. Once inside the loop, the loop body will repeat until the condition is no longer met because it evaluates to `false`. In the previous example, program control will enter the loop body because `x` is equal to 2. However, `x` is incremented in the loop, so when the condition is checked again it will evaluate to `false` and exit the loop.

Any variables used in the expression of a `while` loop must be declared before the expression is evaluated. In other words, you can't say this:

```
while (int x = 2) { }   // not legal
```

Then again, why would you? Instead of testing the variable, you'd be declaring and initializing it, so it would always have the exact same value. Not much of a test condition!

The key point to remember about a `while` loop is that it might not ever run. If the test expression is `false` the first time the `while` expression is checked, the loop body will be skipped and the program will begin executing at the first statement *after* the `while` loop. Look at the following example:

```
int x = 8;
while (x > 8) {
  System.out.println("in the loop");
  x = 10;
}
System.out.println("past the loop");
```

Running this code produces

```
past the loop
```

Because the expression `(x > 8)` evaluates to `false`, none of the code within the `while` loop ever executes.

Using do Loops

The do loop is similar to the while loop, except that the expression is not evaluated until after the do loop's code is executed. Therefore, the code in a do loop is guaranteed to execute at least once. The following shows a do loop in action:

```
do {
  System.out.println("Inside loop");
} while(false);
```

The System.out.println() statement will print once, even though the expression evaluates to false. Remember, the do loop will always run the code in the loop body at least once. Be sure to note the use of the semicolon at the end of the while expression.

exam watch

As with if tests, look for while loops (and the while test in a do loop) with an expression that does not resolve to a boolean. Take a look at the following examples of legal and illegal while expressions:

```
int x = 1;
while (x) { }          // Won't compile; x is not a boolean
while (x = 5) { }      // Won't compile; resolves to 5
                       // (as the result of assignment)
while (x == 5) { }     // Legal, equality test
while (true) { }       // Legal
```

Using for Loops

As of Java 5, the for loop took on a second structure. We'll call the old style of for loop the "basic for loop," and we'll call the new style of for loop the "enhanced for loop" (it's also sometimes called the for-each). Depending on what documentation you use, you'll see both terms, along with for-in. The terms for-in, for-each, and "enhanced for" all refer to the same Java construct.

The basic for loop is more flexible than the enhanced for loop, but the enhanced for loop was designed to make iterating through arrays and collections easier to code.

The Basic for Loop

The `for` loop is especially useful for flow control when you already know how many times you need to execute the statements in the loop's block. The `for` loop declaration has three main parts, besides the body of the loop:

- Declaration and initialization of variables
- The `boolean` expression (conditional test)
- The iteration expression

The three `for` declaration parts are separated by semicolons. The following two examples demonstrate the `for` loop. The first example shows the parts of a `for` loop in a pseudocode form, and the second shows a typical example of a `for` loop:

```
for (/*Initialization*/ ; /*Condition*/ ;  /* Iteration */) {
  /* loop body */
}

for (int i = 0; i<10; i++) {
  System.out.println("i is " + i);
}
```

The Basic for Loop: Declaration and Initialization

The first part of the `for` statement lets you declare and initialize zero, one, or multiple variables of the same type inside the parentheses after the `for` keyword. If you declare more than one variable of the same type, you'll need to separate them with commas as follows:

```
for (int x = 10, y = 3; y > 3; y++) { }
```

The declaration and initialization happens before anything else in a `for` loop. And whereas the other two parts—the `boolean` test and the iteration expression—will run with each iteration of the loop, the declaration and initialization happens just once, at the very beginning. You also must know that the scope of variables declared in the `for` loop ends with the `for` loop! The following demonstrates this:

```
for (int x = 1; x < 2; x++) {
  System.out.println(x);  // Legal
}
System.out.println(x);    // Not Legal! x is now out of scope
                          // and can't be accessed.
```

If you try to compile this, you'll get something like this:

```
Test.java:19: cannot resolve symbol
symbol  : variable x
location: class Test
  System.out.println(x);
                     ^
```

Basic for Loop: Conditional (boolean) Expression

The next section that executes is the conditional expression, which (like all other conditional tests) must evaluate to a `boolean` value. You can have only one logical expression, but it can be very complex. Look out for code that uses logical expressions like this:

```
for (int x = 0; (((x < 10) && (y-- > 2)) | x == 3)); x++) { }
```

The preceding code is legal, but the following is not:

```
for (int x = 0; (x > 5), (y < 2); x++) { } // too many
                                           // expressions
```

The compiler will let you know the problem:

```
TestLong.java:20: ';' expected
for (int x = 0; (x > 5), (y < 2); x++) { }
                       ^
```

The rule to remember is this: *You can have only one test expression.*

In other words, you can't use multiple tests separated by commas, even though the other two parts of a `for` statement can have multiple parts.

Basic for Loop: Iteration Expression

After each execution of the body of the `for` loop, the iteration expression is executed. This is where you get to say what you want to happen with each iteration of the loop. Remember that it always happens after the loop body runs! Look at the following:

```
for (int x = 0; x < 1; x++) {
  // body code that doesn't change the value of x
}
```

This loop executes just once. The first time into the loop, `x` is set to 0, then `x` is tested to see if it's less than 1 (which it is), and then the body of the loop executes. After the body of the loop runs, the iteration expression runs, incrementing `x` by 1.

Next, the conditional test is checked, and since the result is now `false`, execution jumps to below the `for` loop and continues on.

Keep in mind that barring a forced exit, evaluating the iteration expression and then evaluating the conditional expression are always the last two things that happen in a `for` loop!

Examples of forced exits include a `break`, a return, a `System.exit()`, and an exception, which will all cause a loop to terminate abruptly, without running the iteration expression. Look at the following code:

```
static boolean doStuff() {
  for (int x = 0; x < 3; x++) {
    System.out.println("in for loop");
    return true;
  }
  return true;
}
```

Running this code produces

```
in for loop
```

The statement prints only once, because a `return` causes execution to leave not just the current iteration of a loop, but the entire method. So the iteration expression never runs in that case. Table 6-1 lists the causes and results of abrupt loop termination.

Basic for Loop: for Loop Issues

None of the three sections of the `for` declaration are required! The following example is perfectly legal (although not necessarily good practice):

```
for( ; ; ) {
  System.out.println("Inside an endless loop");
}
```

In this example, all the declaration parts are left out, so the `for` loop will act like an endless loop.

TABLE 6-1 Causes of Early Loop Termination

Code in Loop	What Happens
`break`	Execution jumps immediately to the first statement after the `for` loop.
`return`	Execution jumps immediately back to the calling method.
`System.exit()`	All program execution stops; the VM shuts down.

For the exam, it's important to know that with the absence of the initialization and increment sections, the loop will act like a `while` loop. The following example demonstrates how this is accomplished:

```
int i = 0;

for (;i<10;) {
  i++;
  // do some other work
}
```

The next example demonstrates a `for` loop with multiple variables in play. A comma separates the variables, and they must be of the same type. Remember that the variables declared in the `for` statement are all local to the `for` loop and can't be used outside the scope of the loop.

```
for (int i = 0,j = 0; (i<10) && (j<10); i++, j++) {
  System.out.println("i is " + i + " j is " +j);
}
```

exam watch

Variable scope plays a large role in the exam. You need to know that a variable declared in the `for` loop can't be used beyond the `for` loop. But a variable only initialized in the `for` statement (but declared earlier) can be used beyond the loop. For example, the following is legal:

```
int x = 3;
for (x = 12; x < 20; x++) { }
System.out.println(x);
```

But this is not:

```
for (int x = 3; x < 20; x++) { } System.out.println(x);
```

The last thing to note is that all three sections of the `for` loop are independent of each other. The three expressions in the `for` statement don't need to operate on the same variables, although they typically do. But even the iterator expression, which many mistakenly call the "increment expression," doesn't need to increment or set

anything; you can put in virtually any arbitrary code statements that you want to happen with each iteration of the loop. Look at the following:

```
int b = 3;
for (int a = 1;  b != 1; System.out.println("iterate")) {
  b = b - a;
}
```

The preceding code prints

```
iterate
iterate
```

exam watch

Many questions in the Java 7 exams list "Compilation fails" and "An exception occurs at runtime" as possible answers. This makes them more difficult, because you can't simply work through the behavior of the code. You must first make sure the code isn't violating any fundamental rules that will lead to a compiler error, and then look for possible exceptions. Only after you've satisfied those two should you dig into the logic and flow of the code in the question.

The Enhanced for Loop (for Arrays)

The enhanced `for` loop, new as of Java 5, is a specialized `for` loop that simplifies looping through an array or a collection. In this chapter we're going to focus on using the enhanced `for` to loop through arrays. In Chapter 11 we'll revisit the enhanced `for` as we discuss collections—where the enhanced `for` really comes into its own.

Instead of having *three* components, the enhanced `for` has *two*. Let's loop through an array the basic (old) way, and then using the enhanced `for`:

```
int [] a = {1,2,3,4};
for(int x = 0; x < a.length; x++)   // basic for loop
  System.out.print(a[x]);
for(int n : a)                      // enhanced for loop
  System.out.print(n);
```

This produces the following output:

```
12341234
```

More formally, let's describe the enhanced `for` as follows:

```
for(declaration : expression)
```

The two pieces of the `for` statement are

- **`declaration`** The *newly declared* block variable, of a type compatible with the elements of the array you are accessing. This variable will be available within the `for` block, and its value will be the same as the current array element.
- **`expression`** This must evaluate to the array you want to loop through. This could be an array variable or a method call that returns an array. The array can be any type: primitives, objects, or even arrays of arrays.

Using the preceding definitions, let's look at some legal and illegal enhanced `for` declarations:

```
int x;
long x2;
long [] la = {7L, 8L, 9L};
int [][] twoDee = {{1,2,3}, {4,5,6}, {7,8,9}};
String [] sNums = {"one", "two", "three"};
Animal [] animals = {new Dog(), new Cat()};

// legal 'for' declarations
for(long y : la ) ;        // loop thru an array of longs
for(int[] n : twoDee) ;    // loop thru the array of arrays
for(int n2 : twoDee[2]) ;  // loop thru the 3rd sub-array
for(String s : sNums) ;    // loop thru the array of Strings
for(Object o : sNums) ;    // set an Object reference to
                           // each String
for(Animal a : animals) ;  // set an Animal reference to each
                           // element

// ILLEGAL 'for' declarations
for(x2 : la) ;             // x2 is already declared
for(int x2 : twoDee) ;     // can't stuff an array into an int
for(int x3 : la) ;         // can't stuff a long into an int
for(Dog d : animals) ;     // you might get a Cat!
```

The enhanced `for` loop assumes that, barring an early exit from the loop, you'll always loop through every element of the array. The following discussions of `break` and `continue` apply to both the basic and enhanced `for` loops.

Using break and continue

The `break` and `continue` keywords are used to stop either the entire loop (`break`) or just the current iteration (`continue`). Typically, if you're using `break` or `continue`, you'll do an `if` test within the loop, and if some condition becomes `true` (or `false` depending on the program), you want to get out immediately. The difference between them is whether or not you continue with a new iteration or jump to the first statement below the loop and continue from there.

exam watch

Remember, `continue` statements must be inside a loop; otherwise, you'll get a compiler error. `break` statements must be used inside either a loop or a `switch` statement.

The `break` statement causes the program to stop execution of the innermost loop and start processing the next line of code after the block.

The `continue` statement causes only the current iteration of the innermost loop to cease and the next iteration of the same loop to start if the condition of the loop is met. When using a `continue` statement with a `for` loop, you need to consider the effects that `continue` has on the loop iteration. Examine the following code:

```
for (int i = 0; i < 10; i++) {
  System.out.println("Inside loop");
  continue;
}
```

The question is, is this an endless loop? The answer is no. When the `continue` statement is hit, the iteration expression still runs! It runs just as though the current iteration ended "in the natural way." So in the preceding example, `i` will still increment before the condition `(i < 10)` is checked again.

Most of the time, a `continue` is used within an `if` test as follows:

```
for (int i = 0; i < 10; i++) {
  System.out.println("Inside loop");
  if (foo.doStuff() == 5) {
    continue;
  }
  // more loop code, that won't be reached when the above if
  // test is true
}
```

Unlabeled Statements

Both the `break` statement and the `continue` statement can be unlabeled or labeled. Although it's far more common to use `break` and `continue` unlabeled, the exam expects you to know how labeled `break` and `continue` statements work. As stated before, a `break` statement (unlabeled) will exit out of the innermost looping construct and proceed with the next line of code beyond the loop block. The following example demonstrates a `break` statement:

```
boolean problem = true;
while (true) {
  if (problem) {
    System.out.println("There was a problem");
    break;
  }
}
// next line of code
```

In the previous example, the `break` statement is unlabeled. The following is an example of an unlabeled `continue` statement:

```
while (!EOF) {
                      // read a field from a file
  if (wrongField) {
    continue;         // move to the next field in the file
  }
                      // otherwise do other stuff with the field
}
```

In this example, a file is being read one field at a time. When an error is encountered, the program moves to the next field in the file and uses the `continue` statement to go back into the loop (if it is not at the end of the file) and keeps reading the various fields. If the `break` command were used instead, the code would stop reading the file once the error occurred and move on to the next line of code after the loop. The `continue` statement gives you a way to say, "This particular iteration of the loop needs to stop, but not the whole loop itself. I just don't want the rest of the code in this iteration to finish, so do the iteration expression and then start over with the test, and don't worry about what was below the `continue` statement."

Labeled Statements

Although many statements in a Java program can be labeled, it's most common to use labels with loop statements like `for` or `while`, in conjunction with `break` and

continue statements. A label statement must be placed just before the statement being labeled, and it consists of a valid identifier that ends with a colon (:).

You need to understand the difference between labeled and unlabeled break and continue. The labeled varieties are needed only in situations where you have a nested loop, and they need to indicate which of the nested loops you want to break from, or from which of the nested loops you want to continue with the next iteration. A break statement will exit out of the labeled loop, as opposed to the innermost loop, if the break keyword is combined with a label.

Here's an example of what a label looks like:

```
foo:
  for (int x = 3; x < 20; x++) {
    while(y > 7) {
      y--;
    }
  }
```

The label must adhere to the rules for a valid variable name and should adhere to the Java naming convention. The syntax for the use of a label name in conjunction with a break statement is the break keyword, then the label name, followed by a semicolon. A more complete example of the use of a labeled break statement is as follows:

```
boolean isTrue = true;
outer:
  for(int i=0; i<5; i++) {
    while (isTrue) {
      System.out.println("Hello");
      break outer;
    }     // end of inner while loop
    System.out.println("Outer loop."); // Won't print
  }       // end of outer for loop
System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Good-Bye
```

In this example, the word Hello will be printed one time. Then, the labeled break statement will be executed, and the flow will exit out of the loop labeled outer. The next line of code will then print out Good-Bye.

Let's see what will happen if the continue statement is used instead of the break statement. The following code example is similar to the preceding one, with the exception of substituting continue for break:

```
outer:
  for (int i=0; i<5; i++) {
    for (int j=0; j<5; j++) {
      System.out.println("Hello");
      continue outer;
    }    // end of inner loop
    System.out.println("outer"); // Never prints
  }
System.out.println("Good-Bye");
```

Running this code produces

```
Hello
Hello
Hello
Hello
Hello
Good-Bye
```

In this example, `Hello` will be printed five times. After the `continue` statement is executed, the flow continues with the next iteration of the loop identified with the label. Finally, when the condition in the outer loop evaluates to `false`, this loop will finish and `Good-Bye` will be printed.

EXERCISE 6-2

Creating a Labeled while Loop

Try creating a labeled `while` loop. Make the label `outer` and provide a condition to check whether a variable `age` is less than or equal to 21. Within the loop, increment `age` by 1. Every time the program goes through the loop, check whether `age` is 16. If it is, print the message "get your driver's license" and continue to the outer loop. If not, print "Another year."

- The `outer` label should appear just before the `while` loop begins.
- Make sure `age` is declared outside of the `while` loop.

Labeled `continue` and `break` statements must be inside the loop that has the same label name; otherwise, the code will not compile.

CERTIFICATION OBJECTIVE

Handling Exceptions (OCA Objectives 8.1, 8.2, 8.3, and 8.4)

8.1 Differentiate among checked exceptions, RuntimeExceptions, and errors.

8.2 Create a try-catch block and determine how exceptions alter normal program flow.

8.3 Describe what exceptions are used for in Java.

8.4 Invoke a method that throws an exception.

An old maxim in software development says that 80 percent of the work is used 20 percent of the time. The 80 percent refers to the effort required to check and handle errors. In many languages, writing program code that checks for and deals with errors is tedious and bloats the application source into confusing spaghetti. Still, error detection and handling may be the most important ingredient of any robust application. Java arms developers with an elegant mechanism for handling errors that produces efficient and organized error-handling code: exception handling.

Exception handling allows developers to detect errors easily without writing special code to test return values. Even better, it lets us keep exception-*handling* code cleanly separated from exception-*generating* code. It also lets us use the same exception-handling code to deal with a range of possible exceptions.

Java 7 added several new exception-handling capabilities to the language. For our purposes, Oracle split the various exception-handling topics into two main parts:

1. The OCA exam covers the Java 6 version of exception handling.
2. The OCP exam adds the new exception features added in Java 7.

In order to mirror Oracle's objectives, we split exception handling into two chapters. This chapter will give you the basics—plenty to handle the OCA exam. Chapter 7 (which also marks the beginning of the OCP part of the book) will pick up where we left off by discussing the new Java 7 exception handling features.

Catching an Exception Using try and catch

Before we begin, let's introduce some terminology. The term "exception" means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs. When an exceptional event occurs in Java, an exception is said to be "thrown." The code that's responsible for doing something about the exception is called an "exception handler," and it "catches" the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. For example, if you call a method that opens a file but the file cannot be opened, execution of that method will stop, and code that you wrote to deal with this situation will be run. Therefore, we need a way to tell the JVM what code to execute when a certain exception happens. To do this, we use the `try` and `catch` keywords. The `try` is used to define a block of code in which exceptions may occur. This block of code is called a "guarded region" (which really means "risky code goes here"). One or more `catch` clauses match a specific exception (or group of exceptions—more on that later) to a block of code that handles it. Here's how it looks in pseudocode:

```
 1. try {
 2.   // This is the first line of the "guarded region"
 3.   // that is governed by the try keyword.
 4.   // Put code here that might cause some kind of exception.
 5.   // We may have many code lines here or just one.
 6. }
 7. catch(MyFirstException) {
 8.   // Put code here that handles this exception.
 9.   // This is the next line of the exception handler.
10.   // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13.   // Put code here that handles this exception
14. }
15.
16.   // Some other unguarded (normal, non-risky) code begins here
```

In this pseudocode example, lines 2 through 5 constitute the guarded region that is governed by the `try` clause. Line 7 is an exception handler for an exception of type `MyFirstException`. Line 12 is an exception handler for an exception of type `MySecondException`. Notice that the `catch` blocks immediately follow the `try` block. This is a requirement; if you have one or more `catch` blocks, they must immediately follow the `try` block. Additionally, the `catch` blocks must all follow

each other, without any other statements or blocks in between. Also, the order in which the `catch` blocks appear matters, as we'll see a little later.

Execution of the guarded region starts at line 2. If the program executes all the way past line 5 with no exceptions being thrown, execution will transfer to line 15 and continue downward. However, if at any time in lines 2 through 5 (the `try` block) an exception of type `MyFirstException` is thrown, execution will immediately transfer to line 7. Lines 8 through 10 will then be executed so that the entire `catch` block runs, and then execution will transfer to line 15 and continue.

Note that if an exception occurred on, say, line 3 of the `try` block, the rest of the lines in the `try` block (4 and 5) would never be executed. Once control jumps to the `catch` block, it never returns to complete the balance of the `try` block. This is exactly what you want, though. Imagine that your code looks something like this pseudocode:

```
try {
  getTheFileFromOverNetwork
  readFromTheFileAndPopulateTable
}
catch(CantGetFileFromNetwork) {
  displayNetworkErrorMessage
}
```

This pseudocode demonstrates how you typically work with exceptions. Code that's dependent on a risky operation (as populating a table with file data is dependent on getting the file from the network) is grouped into a `try` block in such a way that if, say, the first operation fails, you won't continue trying to run other code that's also guaranteed to fail. In the pseudocode example, you won't be able to read from the file if you can't get the file off the network in the first place.

One of the benefits of using exception handling is that code to handle any particular exception that may occur in the governed region needs to be written only once. Returning to our earlier code example, there may be three different places in our `try` block that can generate a `MyFirstException`, but wherever it occurs it will be handled by the same `catch` block (on line 7). We'll discuss more benefits of exception handling near the end of this chapter.

Using finally

Although `try` and `catch` provide a terrific mechanism for trapping and handling exceptions, we are left with the problem of how to clean up after ourselves if an exception occurs. Because execution transfers out of the `try` block as soon as an exception is thrown, we can't put our cleanup code at the bottom of the `try` block

and expect it to be executed if an exception occurs. Almost as bad an idea would be placing our cleanup code in each of the `catch` blocks—let's see why.

Exception handlers are a poor place to clean up after the code in the `try` block because each handler then requires its own copy of the cleanup code. If, for example, you allocated a network socket or opened a file somewhere in the guarded region, each exception handler would have to close the file or release the socket. That would make it too easy to forget to do cleanup and also lead to a lot of redundant code. To address this problem, Java offers the `finally` block.

A `finally` block encloses code that is always executed at some point after the `try` block, whether an exception was thrown or not. Even if there is a `return` statement in the `try` block, the `finally` block executes right after the `return` statement is encountered and before the `return` executes!

This is the right place to close your files, release your network sockets, and perform any other cleanup your code requires. If the `try` block executes with no exceptions, the `finally` block is executed immediately after the `try` block completes. If there was an exception thrown, the `finally` block executes immediately after the proper `catch` block completes. Let's look at another pseudocode example:

```
 1: try {
 2:   // This is the first line of the "guarded region".
 3: }
 4: catch(MyFirstException) {
 5:   // Put code here that handles this exception
 6: }
 7: catch(MySecondException) {
 8:   // Put code here that handles this exception
 9: }
10: finally {
11:   // Put code here to release any resource we
12:   // allocated in the try clause
13: }
14:
15:   // More code here
```

As before, execution starts at the first line of the `try` block, line 2. If there are no exceptions thrown in the `try` block, execution transfers to line 11, the first line of the `finally` block. On the other hand, if a `MySecondException` is thrown while the code in the `try` block is executing, execution transfers to the first line of that exception handler, line 8 in the `catch` clause. After all the code in the `catch` clause is executed, the program moves to line 11, the first line of the `finally` clause. Repeat after me: `finally` always runs! Okay, we'll have to refine that a little, but for now, start burning in the idea that `finally` always runs. If an exception is thrown, `finally` runs. If an exception is not thrown, `finally` runs. If the exception is

caught, `finally` runs. If the exception is not caught, `finally` runs. Later we'll look at the few scenarios in which `finally` might not run or complete.

Remember, `finally` clauses are not required. If you don't write one, your code will compile and run just fine. In fact, if you have no resources to clean up after your `try` block completes, you probably don't need a `finally` clause. Also, because the compiler doesn't even require `catch` clauses, sometimes you'll run across code that has a `try` block immediately followed by a `finally` block. Such code is useful when the exception is going to be passed back to the calling method, as explained in the next section. Using a `finally` block allows the cleanup code to execute even when there isn't a `catch` clause.

The following legal code demonstrates a `try` with a `finally` but no `catch`:

```
try {
  // do stuff
} finally {
  // clean up
}
```

The following legal code demonstrates a `try`, `catch`, and `finally`:

```
try {
  // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}
```

The following ILLEGAL code demonstrates a `try` without a `catch` or `finally`:

```
try {
  // do stuff
}
  // need a catch or finally here
System.out.println("out of try block");
```

The following ILLEGAL code demonstrates a misplaced `catch` block:

```
try {
  // do stuff
}
  // can't have code between try/catch
System.out.println("out of try block");
catch(Exception ex) { }
```

exam watch

It is illegal to use a `try` clause without either a `catch` clause or a `finally` clause. A `try` clause by itself will result in a compiler error. Any `catch` clauses must immediately follow the `try` block. Any `finally` clause must immediately follow the last `catch` clause (or it must immediately follow the `try` block if there is no `catch`). It is legal to omit either the `catch` clause or the `finally` clause, but not both.

Propagating Uncaught Exceptions

Why aren't `catch` clauses required? What happens to an exception that's thrown in a `try` block when there is no `catch` clause waiting for it? Actually, there's no requirement that you code a `catch` clause for every possible exception that could be thrown from the corresponding `try` block. In fact, it's doubtful that you could accomplish such a feat! If a method doesn't provide a `catch` clause for a particular exception, that method is said to be "ducking" the exception (or "passing the buck").

So what happens to a ducked exception? Before we discuss that, we need to briefly review the concept of the call stack. Most languages have the concept of a method stack or a call stack. Simply put, the call stack is the chain of methods that your program executes to get to the current method. If your program starts in method `main()` and `main()` calls method `a()`, which calls method `b()`, which in turn calls method `c()`, the call stack consists of the following:

```
c
b
a
main
```

We will represent the stack as growing upward (although it can also be visualized as growing downward). As you can see, the last method called is at the top of the stack, while the first calling method is at the bottom. The method at the very top of the stack trace would be the method you were currently executing. If we move back down the call stack, we're moving from the current method to the previously called method. Figure 6-1 illustrates a way to think about how the call stack in Java works.

Now let's examine what happens to ducked exceptions. Imagine a building, say, five stories high, and at each floor there is a deck or balcony. Now imagine that on each deck, one person is standing holding a baseball mitt. Exceptions are like balls dropped from person to person, starting from the roof. An exception is first thrown

FIGURE 6-1

The Java method call stack

1) The call stack while `method3()` is running.

4	`method3()`	`method2` invokes `method3`
3	`method2()`	`method1` invokes `method2`
2	`method1()`	`main` invokes `method1`
1	`main()`	`main` begins

The order in which methods are put on the call stack

2) The call stack after `method3()` completes
Execution returns to `method2()`

1	`method2()`	`method2()` will complete
2	`method1()`	`method1()` will complete
3	`main()`	`main()` will complete and the JVM will exit

The order in which methods complete

from the top of the stack (in other words, the person on the roof), and if it isn't caught by the same person who threw it (the person on the roof), it drops down the call stack to the previous method, which is the person standing on the deck one floor down. If not caught there by the person one floor down, the exception/ball again drops down to the previous method (person on the next floor down), and so on until it is caught or until it reaches the very bottom of the call stack. This is called "exception propagation."

If an exception reaches the bottom of the call stack, it's like reaching the bottom of a very long drop; the ball explodes, and so does your program. An exception that's never caught will cause your application to stop running. A description (if one is available) of the exception will be displayed, and the call stack will be "dumped." This helps you debug your application by telling you what exception was thrown, from what method it was thrown, and what the stack looked like at the time.

exam watch

You can keep throwing an exception down through the methods on the stack. But what happens when you get to the `main()` method at the bottom? You can throw the exception out of `main()` as well. This results in the JVM halting, and the stack trace will be printed to the output. The following code throws an exception:

```
class TestEx {
  public static void main (String [] args) {
    doStuff();
  }
  static void doStuff() {
    doMoreStuff();
  }
  static void doMoreStuff() {
    int x = 5/0;  // Can't divide by zero!
                  // ArithmeticException is thrown here
  }
}
```

It prints out a stack trace something like this:

```
 %java TestEx
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestEx.doMoreStuff(TestEx.java:10)
at TestEx.doStuff(TestEx.java:7)
at TestEx.main(TestEx.java:3)
```

EXERCISE 6-3

Propagating and Catching an Exception

In this exercise you're going to create two methods that deal with exceptions. One of the methods is the `main()` method, which will call another method. If an exception is thrown in the other method, `main()` must deal with it. A `finally` statement will be included to indicate that the program has completed. The method that `main()` will call will be named `reverse`, and it will reverse the order of the characters in a `String`. If the `String` contains no characters, `reverse` will propagate an exception up to the `main()` method.

1. Create a class called `Propagate` and a `main()` method, which will remain empty for now.
2. Create a method called `reverse`. It takes an argument of a `String` and returns a `String`.
3. In `reverse`, check whether the `String` has a length of 0 by using the `String.length()` method. If the length is 0, the `reverse` method will throw an exception.

4. Now include the code to reverse the order of the `String`. Because this isn't the main topic of this chapter, the reversal code has been provided, but feel free to try it on your own.

```
String reverseStr = "";
for(int i=s.length()-1;i>=0;--i) {
  reverseStr += s.charAt(i);
}
return reverseStr;
```

5. Now in the `main()` method you will attempt to call this method and deal with any potential exceptions. Additionally, you will include a `finally` statement that displays when `main()` has finished.

Defining Exceptions

We have been discussing exceptions as a concept. We know that they are thrown when a problem of some type happens, and we know what effect they have on the flow of our program. In this section we will develop the concepts further and use exceptions in functional Java code.

Earlier we said that an exception is an occurrence that alters the normal program flow. But because this is Java, anything that's not a primitive must be…an object. Exceptions are no exception to this rule. Every exception is an instance of a class that has class `Exception` in its inheritance hierarchy. In other words, exceptions are always some subclass of `java.lang.Exception`.

When an exception is thrown, an object of a particular `Exception` subtype is instantiated and handed to the exception handler as an argument to the `catch` clause. An actual `catch` clause looks like this:

```
try {
  // some code here
}
catch (ArrayIndexOutOfBoundsException e) {
  e.printStackTrace();
}
```

In this example, `e` is an instance of the `ArrayIndexOutOfBoundsException` class. As with any other object, you can call its methods.

Exception Hierarchy

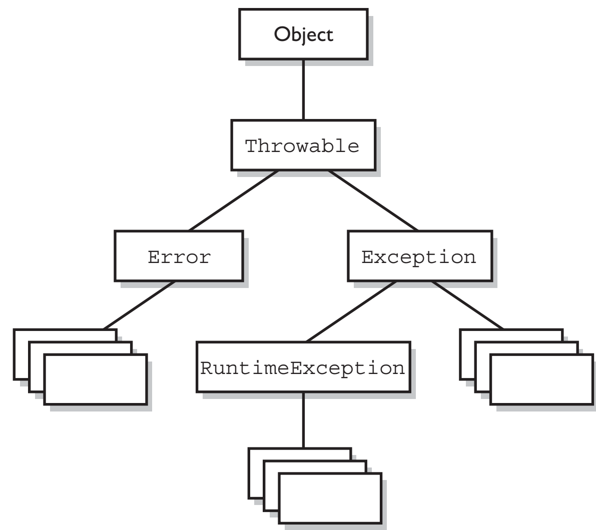
All exception classes are subtypes of class `Exception`. This class derives from the class `Throwable` (which derives from the class `Object`). Figure 6-2 shows the hierarchy for the exception classes.

As you can see, there are two subclasses that derive from `Throwable`: `Exception` and `Error`. Classes that derive from `Error` represent unusual situations that are not caused by program errors and indicate things that would not normally happen during program execution, such as the JVM running out of memory. Generally, your application won't be able to recover from an `Error`, so you're not required to handle them. If your code does not handle them (and it usually won't), it will still compile with no trouble. Although often thought of as exceptional conditions, `Errors` are technically not exceptions because they do not derive from class `Exception`.

In general, an exception represents something that happens not as a result of a programming error, but rather because some resource is not available or some other condition required for correct execution is not present. For example, if your application is supposed to communicate with another application or computer that is not answering, this is an exception that is not caused by a bug. Figure 6-2 also shows a subtype of `Exception` called `RuntimeException`. These exceptions are a special case because they sometimes do indicate program errors. They can also represent rare, difficult-to-handle exceptional conditions. Runtime exceptions are discussed in greater detail later in this chapter.

FIGURE 6-2

Exception class hierarchy



Java provides many exception classes, most of which have quite descriptive names. There are two ways to get information about an exception. The first is from the type of the exception itself. The next is from information that you can get from the exception object. Class `Throwable` (at the top of the inheritance tree for exceptions) provides its descendants with some methods that are useful in exception handlers. One of these is `printStackTrace()`. As you would expect, if you call an exception object's `printStackTrace()` method, as in the earlier example, a stack trace from where the exception occurred will be printed.

We discussed that a call stack builds upward with the most recently called method at the top. You will notice that the `printStackTrace()` method prints the most recently entered method first and continues down, printing the name of each method as it works its way down the call stack (this is called "unwinding the stack") from the top.

exam watch

For the exam, you don't need to know any of the methods contained in the `Throwable` classes, including `Exception` and `Error`. You are expected to know that `Exception`, `Error`, `RuntimeException`, and `Throwable` types can all be thrown using the `throw` keyword and can all be caught (although you rarely will catch anything other than `Exception` subtypes).

Handling an Entire Class Hierarchy of Exceptions

We've discussed that the `catch` keyword allows you to specify a particular type of exception to catch. You can actually catch more than one type of exception in a single `catch` clause. If the exception class that you specify in the `catch` clause has no subclasses, then only the specified class of exception will be caught. However, if the class specified in the `catch` clause does have subclasses, any exception object that subclasses the specified class will be caught as well.

For example, class `IndexOutOfBoundsException` has two subclasses, `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`. You may want to write one exception handler that deals with exceptions produced by either type of boundary error, but you might not be concerned with which exception you actually have. In this case, you could write a `catch` clause like the following:

```
try {
  // Some code here that can throw a boundary exception
}
catch (IndexOutOfBoundsException e) {
  e.printStackTrace();
}
```

If any code in the `try` block throws `ArrayIndexOutOfBoundsException` or `StringIndexOutOfBoundsException`, the exception will be caught and handled. This can be convenient, but it should be used sparingly. By specifying an exception class's superclass in your `catch` clause, you're discarding valuable information about the exception. You can, of course, find out exactly what exception class you have, but if you're going to do that, you're better off writing a separate `catch` clause for each exception type of interest.

Resist the temptation to write a single catchall exception handler such as the following:

```
try {
  // some code
}
catch (Exception e) {
  e.printStackTrace();
}
```

This code will catch every exception generated. Of course, no single exception handler can properly handle every exception, and programming in this way defeats the design objective. Exception handlers that trap many errors at once will probably reduce the reliability of your program, because it's likely that an exception will be caught that the handler does not know how to handle.

Exception Matching

If you have an exception hierarchy composed of a superclass exception and a number of subtypes, and you're interested in handling one of the subtypes in a special way but want to handle all the rest together, you need write only two `catch` clauses.

When an exception is thrown, Java will try to find (by looking at the available `catch` clauses from the top down) a `catch` clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception. If it does not find a `catch` clause that matches a supertype for the exception, then the exception is propagated down the call stack. This process is called "exception matching." Let's look at an example.

```
 1: import java.io.*;
 2: public class ReadData {
 3:   public static void main(String args[]) {
 4:     try {
 5:       RandomAccessFile raf =
 6:         new RandomAccessFile("myfile.txt", "r");
 7:       byte b[] = new byte[1000];
 8:       raf.readFully(b, 0, 1000);
 9:     }
10:     catch(FileNotFoundException e) {
11:       System.err.println("File not found");
12:       System.err.println(e.getMessage());
13:       e.printStackTrace();
14:     }
15:     catch(IOException e) {
16:       System.err.println("IO Error");
17:       System.err.println(e.toString());
18:       e.printStackTrace();
19:     }
20:   }
21: }
```

This short program attempts to open a file and to read some data from it. Opening and reading files can generate many exceptions, most of which are some type of `IOException`. Imagine that in this program we're interested in knowing only whether the exact exception is a `FileNotFoundException`. Otherwise, we don't care exactly what the problem is.

`FileNotFoundException` is a subclass of `IOException`. Therefore, we could handle it in the `catch` clause that catches all subtypes of `IOException`, but then we would have to test the exception to determine whether it was a `FileNotFoundException`. Instead, we coded a special exception handler for the `FileNotFoundException` and a separate exception handler for all other `IOException` subtypes.

If this code generates a `FileNotFoundException`, it will be handled by the `catch` clause that begins at line 10. If it generates another `IOException`—perhaps `EOFException`, which is a subclass of `IOException`—it will be handled by the `catch` clause that begins at line 15. If some other exception is generated, such as a runtime exception of some type, neither `catch` clause will be executed and the exception will be propagated down the call stack.

Notice that the `catch` clause for the `FileNotFoundException` was placed above the handler for the `IOException`. This is really important! If we do it the opposite way, the program will not compile. The handlers for the most specific exceptions must always be placed above those for more general exceptions. The following will not compile:

```
try {
  // do risky IO things
} catch (IOException e) {
  // handle general IOExceptions
} catch (FileNotFoundException ex) {
  // handle just FileNotFoundException
}
```

You'll get a compiler error something like this:

```
TestEx.java:15: exception java.io.FileNotFoundException has
 already been caught
} catch (FileNotFoundException ex) {
  ^
```

If you think back to the people with baseball mitts (in the section "Propagating Uncaught Exceptions"), imagine that the most general mitts are the largest and can thus catch many different kinds of balls. An `IOException` mitt is large enough and flexible enough to catch any type of `IOException`. So if the person on the fifth floor (say, Fred) has a big ol' `IOException` mitt, he can't help but catch a `FileNotFoundException` ball with it. And if the guy (say, Jimmy) on the second floor is holding a `FileNotFoundException` mitt, that `FileNotFoundException` ball will never get to him, since it will always be stopped by Fred on the fifth floor, standing there with his big-enough-for-any-`IOException` mitt.

So what do you do with exceptions that are siblings in the class hierarchy? If one `Exception` class is not a subtype or supertype of the other, then the order in which the `catch` clauses are placed doesn't matter.

Exception Declaration and the Public Interface

So, how do we know that some method throws an exception that we have to catch? Just as a method must specify what type and how many arguments it accepts and what is returned, the exceptions that a method can throw must be *declared* (unless the exceptions are subclasses of `RuntimeException`). The list of thrown exceptions is part of a method's public interface. The `throws` keyword is used as follows to list the exceptions that a method can throw:

```
void myFunction() throws MyException1, MyException2 {
  // code for the method here
}
```

This method has a `void` return type, accepts no arguments, and declares that it can throw one of two types of exceptions: either type `MyException1` or type `MyException2`.

(Just because the method declares that it throws an exception doesn't mean it always will. It just tells the world that it might.)

Suppose your method doesn't directly throw an exception, but calls a method that does. You can choose not to handle the exception yourself and instead just declare it, as though it were your method that actually throws the exception. If you do declare the exception that your method might get from another method, and you don't provide a `try/catch` for it, then the method will propagate back to the method that called your method and will either be caught there or continue on to be handled by a method further down the stack.

Any method that might throw an exception (unless it's a subclass of `RuntimeException`) must declare the exception. That includes methods that aren't actually throwing it directly, but are "ducking" and letting the exception pass down to the next method in the stack. If you "duck" an exception, it is just as if you were the one actually throwing the exception. `RuntimeException` subclasses are exempt, so the compiler won't check to see if you've declared them. But all non-`RuntimeExceptions` are considered "checked" exceptions, because the compiler checks to be certain you've acknowledged that "bad things could happen here."

Remember this:

> Each method must either handle all checked exceptions by supplying a `catch` clause or list each unhandled checked exception as a thrown exception.

This rule is referred to as Java's "handle or declare" requirement (sometimes called "catch or declare").

exam watch

Look for code that invokes a method declaring an exception, where the calling method doesn't handle or declare the checked exception. The following code (which uses the `throw` keyword to throw an exception manually—more on this next) has two big problems that the compiler will prevent:

```
void doStuff() {
  doMore();
}
void doMore() {
  throw new IOException();
}
```

First, the `doMore()` method throws a checked exception but does not declare it! But suppose we fix the `doMore()` method as follows:

```
void doMore() throws IOException { … }
```

The `doStuff()` method is still in trouble because it, too, must declare the `IOException`, unless it handles it by providing a `try/catch`, with a `catch` clause that can take an `IOException`.

Again, some exceptions are exempt from this rule. An object of type `RuntimeException` may be thrown from any method without being specified as part of the method's public interface (and a handler need not be present). And even if a method does declare a `RuntimeException`, the calling method is under no obligation to handle or declare it. `RuntimeException`, `Error`, and all of their subtypes are unchecked exceptions, and unchecked exceptions do not have to be specified or handled. Here is an example:

```
import java.io.*;
class Test {
  public int myMethod1() throws EOFException {
    return myMethod2();
  }
  public int myMethod2() throws EOFException {
    // code that actually could throw the exception goes here
    return 1;
  }
}
```

Let's look at `myMethod1()`. Because `EOFException` subclasses `IOException`, and `IOException` subclasses `Exception`, it is a checked exception and must be declared as an exception that may be thrown by this method. But where will the exception actually come from? The public interface for method `myMethod2()` called here declares that an exception of this type can be thrown. Whether that method actually throws the exception itself or calls another method that throws it is unimportant to us; we simply know that we either have to catch the exception or declare that we threw it. The method `myMethod1()` does not catch the exception, so it declares that it throws it. Now let's look at another legal example, `myMethod3()`:

```
public void myMethod3() {
  // code that could throw a NullPointerException goes here
}
```

According to the comment, this method can throw a `NullPointerException`. Because `RuntimeException` is the superclass of `NullPointerException`, it is an unchecked exception and need not be declared. We can see that `myMethod3()` does not declare any exceptions.

Runtime exceptions are referred to as *unchecked* exceptions. All other exceptions are *checked* exceptions, and they don't derive from `java.lang.RuntimeException`. A checked exception must be caught somewhere in your code. If you invoke a method that throws a checked exception but you don't catch the checked exception somewhere, your code will not compile. That's why they're called checked exceptions: the compiler checks to make sure that they're handled or declared. A number of the methods in the Java API throw checked exceptions, so you will often write exception handlers to cope with exceptions generated by methods you didn't write.

You can also throw an exception yourself, and that exception can be either an existing exception from the Java API or one of your own. To create your own exception, you simply subclass `Exception` (or one of its subclasses) as follows:

```
class MyException extends Exception { }
```

And if you throw the exception, the compiler will guarantee that you declare it as follows:

```
class TestEx {
  void doStuff() {
    throw new MyException();  // Throw a checked exception
  }
}
```

The preceding code upsets the compiler:

```
TestEx.java:6: unreported exception MyException; must be caught or
declared to be thrown
  throw new MyException();
  ^
```

You need to know how an `Error` compares with checked and unchecked exceptions. Objects of type `Error` are not `Exception` objects, although they do represent exceptional conditions. Both `Exception` and `Error` share a common superclass, `Throwable`; thus both can be thrown using the `throw` keyword. When an `Error` or a subclass of `Error` (like `RuntimeException`) is thrown, it's unchecked. You are not required to catch `Error` objects or `Error` subtypes. You can also throw

exam watch

When an object of a subtype of `Exception` is thrown, it must be handled or declared. These objects are called checked exceptions and include all exceptions except those that are subtypes of `RuntimeException`, which are unchecked exceptions. Be ready to spot methods that don't follow the "handle or declare" rule, such as this:

```
class MyException extends Exception {
  void someMethod () {
    doStuff();
  }
  void doStuff() throws MyException {
    try {
      throw new MyException();
    }
    catch(MyException me) {
      throw me;
    }
  }
}
```

You need to recognize that this code won't compile. If you try, you'll get this:

```
MyException.java:3: unreported exception MyException;
must be caught or declared to be thrown
doStuff();
      ^
```

Notice that `someMethod()` fails either to handle or declare the exception that can be thrown by `doStuff()`.

an `Error` yourself (although, other than `AssertionError`, you probably won't ever want to), and you can catch one, but again, you probably won't. What, for example, would you actually do if you got an `OutOfMemoryError`? It's not like you can tell the garbage collector to run; you can bet the JVM fought desperately to save itself (and reclaimed all the memory it could) by the time you got the error. In other words, don't expect the JVM at that point to say, "Run the garbage collector? Oh, thanks so much for telling me. That just never occurred to me. Sure, I'll get right on it." Even better, what would you do if a `VirtualMachineError` arose? Your program is toast by the time you'd catch the error, so there's really no point in trying to catch

one of these babies. Just remember, though, that you can! The following compiles just fine:

```
class TestEx {
  public static void main (String [] args) {
    badMethod();
  }
  static void badMethod() {  // No need to declare an Error
    doStuff();
  }
  static void doStuff() {    // No need to declare an Error
    try {
      throw new Error();
    }
    catch(Error me) {
      throw me;             // We catch it, but then rethrow it
    }
  }
}
```

If we were throwing a checked exception rather than `Error`, then the `doStuff()` method would need to declare the exception. But remember, since `Error` is not a subtype of `Exception`, it doesn't need to be declared. You're free to declare it if you like, but the compiler just doesn't care one way or another when or how the `Error` is thrown, or by whom.

Because Java has checked exceptions, it's commonly said that Java forces developers to handle exceptions. Yes, Java forces us to write exception handlers for each exception that can occur during normal operation, but it's up to us to make the exception handlers actually do something useful. We know software managers who melt down when they see a programmer write something like this:

```
try {
  callBadMethod();
} catch (Exception ex) { }
```

Notice anything missing? Don't "eat" the exception by catching it without actually handling it. You won't even be able to tell that the exception occurred, because you'll never see the stack trace.

Rethrowing the Same Exception

Just as you can throw a new exception from a `catch` clause, you can also throw the same exception you just caught. Here's a `catch` clause that does this:

```
catch(IOException e) {
  // Do things, then if you decide you can't handle it…
  throw e;
}
```

All other `catch` clauses associated with the same `try` are ignored; if a `finally` block exists, it runs, and the exception is thrown back to the calling method (the next method down the call stack). If you throw a checked exception from a `catch` clause, you must also declare that exception! In other words, you must handle *and* declare, as opposed to handle *or* declare. The following example is illegal:

```
public void doStuff() {
  try {
    // risky IO things
  } catch(IOException ex) {
    // can't handle it
     throw ex;  // Can't throw it unless you declare it
  }
}
```

In the preceding code, the `doStuff()` method is clearly able to throw a checked exception—in this case an `IOException`—so the compiler says, "Well, that's just peachy that you have a `try/catch` in there, but it's not good enough. If you might rethrow the `IOException` you catch, then you must declare it (in the method signature)!"

EXERCISE 6-4

Creating an Exception

In this exercise we attempt to create a custom exception. We won't put in any new methods (it will have only those inherited from `Exception`), and because it extends `Exception`, the compiler considers it a checked exception. The goal of the program is to determine whether a command-line argument representing a particular food (as a string) is considered bad or okay.

1. Let's first create our exception. We will call it `BadFoodException`. This exception will be thrown when a bad food is encountered.

2. Create an enclosing class called `MyException` and a `main()` method, which will remain empty for now.
3. Create a method called `checkFood()`. It takes a `String` argument and throws our exception if it doesn't like the food it was given. Otherwise, it tells us it likes the food. You can add any foods you aren't particularly fond of to the list.
4. Now in the `main()` method, you'll get the command-line argument out of the `String` array and then pass that `String` on to the `checkFood()` method. Because it's a checked exception, the `checkFood()` method must declare it, and the `main()` method must handle it (using a `try/catch`). Do not have `main()` declare the exception, because if `main()` ducks the exception, who else is back there to catch it? (Actually, `main()` can legally declare exceptions, but don't do that in this exercise.)

As nifty as exception handling is, it's still up to the developer to make proper use of it. Exception handling makes organizing our code and signaling problems easy, but the exception handlers still have to be written. You'll find that even the most complex situations can be handled, and your code will be reusable, readable, and maintainable.

CERTIFICATION OBJECTIVE

Common Exceptions and Errors (OCA Objective 8.5)

8.5 Recognize common exception classes and categories.

Exception handling is another area that the exam creation team decided to expand for the OCJP 5, OCJP 6, and both Java 7 exams. The intention of this objective is to make sure that you are familiar with some of the most common exceptions and errors you'll encounter as a Java programmer.

exam watch

The questions from this section are likely to be along the lines of, "Here's some code that just did something bad, which exception will be thrown?" Throughout the exam, questions will present some code and ask you to determine whether the code will run, or whether an exception will be thrown. Since these questions are so common, understanding the causes for these exceptions is critical to your success.

This is another one of those objectives that will turn up all through the real exam (does "An exception is thrown at runtime" ring a bell?), so make sure this section gets a lot of your attention.

Where Exceptions Come From

Jump back a page and take a look at the last sentence. It's important that you understand what causes exceptions and errors, and where they come from. For the purposes of exam preparation, let's define two broad categories of exceptions and errors:

- **JVM exceptions** Those exceptions or errors that are either exclusively or most logically thrown by the JVM
- **Programmatic exceptions** Those exceptions that are thrown explicitly by application and/or API programmers

JVM Thrown Exceptions

Let's start with a very common exception, the `NullPointerException`. As we saw in earlier chapters, this exception occurs when you attempt to access an object using a reference variable with a current value of `null`. There's no way that the compiler can hope to find these problems before runtime. Take a look at the following:

```
class NPE {
  static String s;
  public static void main(String [] args) {
    System.out.println(s.length());
  }
}
```

Surely, the compiler can find the problem with that tiny little program! Nope, you're on your own. The code will compile just fine, and the JVM will throw a `NullPointerException` when it tries to invoke the `length()` method.

Earlier in this chapter we discussed the call stack. As you recall, we used the convention that `main()` would be at the bottom of the call stack, and that as `main()` invokes another method, and that method invokes another, and so on, the stack grows upward. Of course the stack resides in memory, and even if your OS gives you a gigabyte of RAM for your program, it's still a finite amount. It's possible to grow the stack so large that the OS runs out of space to store the call stack. When this happens, you get (wait for it...) a `StackOverflowError`. The most common way for this to occur is to create a recursive method. A recursive method invokes itself in the method body. Although that may sound weird, it's a very common and useful technique for such things as searching and sorting algorithms. Take a look at this code:

```
void go() {    // recursion gone bad
   go();
}
```

As you can see, if you ever make the mistake of invoking the `go()` method, your program will fall into a black hole—`go()` invoking `go()` invoking `go()`, until, no matter how much memory you have, you'll get a `StackOverflowError`. Again, only the JVM knows when this moment occurs, and the JVM will be the source of this error.

Programmatically Thrown Exceptions

Now let's look at programmatically thrown exceptions. Remember we defined "programmatically" as meaning something like this:

Created by an application and/or API developer.

For instance, many classes in the Java API have methods that take `String` arguments and convert these `String`s into numeric primitives. A good example of these classes are the so-called "wrapper classes" that OCP candidates will study in Chapter 8. Even though we haven't talked about wrapper classes yet, the following example should make sense.

At some point long ago, some programmer wrote the `java.lang.Integer` class and created methods like `parseInt()` and `valueOf()`. That programmer wisely decided that if one of these methods was passed a `String` that could not be

converted into a number, the method should throw a `NumberFormatException`. The partially implemented code might look something like this:

```
int parseInt(String s) throws NumberFormatException {
  boolean parseSuccess = false;
  int result = 0;
  // do complicated parsing
  if (!parseSuccess)   // if the parsing failed
    throw new NumberFormatException();
  return result;
}
```

Other examples of programmatic exceptions include an `AssertionError` (okay, it's not an exception, but it IS thrown programmatically), and throwing an `IllegalArgumentException`. In fact, our mythical API developer could have used `IllegalArgumentException` for her `parseInt()` method. But it turns out that `NumberFormatException` extends `IllegalArgumentException` and is a little more precise, so in this case, using `NumberFormatException` supports the notion we discussed earlier: that when you have an exception hierarchy, you should use the most precise exception that you can.

Of course, as we discussed earlier, you can also make up your very own special custom exceptions and throw them whenever you want to. These homemade exceptions also fall into the category of "programmatically thrown exceptions."

A Summary of the Exam's Exceptions and Errors

OCA Objective 8.5 does not list specific exceptions and errors; it says "recognize common exceptions…." Table 6-2 summarizes the ten exceptions and errors that are a part of the SCJP 6 exam; it will cover OCA Objective 8.5, too.

End of Part I—OCA

Barring our standard end-of-chapter stuff, such as mock exam questions, you've reached the end of the OCA part of the book. If you've studied these six chapters carefully, and then taken and reviewed the end-of-chapter mock exams and the OCA master exams and done well on them, we're confident that you're a little bit over-prepared for the official Oracle OCA exam. (Not "way" over-prepared—just a little.) Good luck, and we hope to see you back here for Part II, Chapter 7, in which we'll explore the exception handling features added in Java 7.

TABLE 6-2 Descriptions and Sources of Common Exceptions

Exception	Description	Typically Thrown
`ArrayIndexOutOfBoundsException` (Chapter 5)	Thrown when attempting to access an array with an invalid index value (either negative or beyond the length of the array).	By the JVM
`ClassCastException` (Chapter 2)	Thrown when attempting to cast a reference variable to a type that fails the IS-A test.	By the JVM
`IllegalArgumentException`	Thrown when a method receives an argument formatted differently than the method expects.	Programmatically
`IllegalStateException`	Thrown when the state of the environment doesn't match the operation being attempted—for example, using a scanner that's been closed.	Programmatically
`NullPointerException` (Chapter 3)	Thrown when attempting to invoke a method on, or access a property from, a reference variable whose current value is `null`.	By the JVM
`NumberFormatException` (this chapter)	Thrown when a method that converts a `String` to a number receives a `String` that it cannot convert.	Programmatically
`AssertionError`	Thrown when an `assert` statement's `boolean` test returns `false`.	Programmatically
`ExceptionInInitializerError` (Chapter 2)	Thrown when attempting to initialize a static variable or an initialization block.	By the JVM
`StackOverflowError` (this chapter)	Typically thrown when a method recurses too deeply. (Each invocation is added to the stack.)	By the JVM
`NoClassDefFoundError`	Thrown when the JVM can't find a class it needs, because of a command-line error, a classpath issue, or a missing `.class` file.	By the JVM

CERTIFICATION SUMMARY

This chapter covered a lot of ground, all of which involved ways of controlling your program flow, based on a conditional test. First you learned about `if` and `switch` statements. The `if` statement evaluates one or more expressions to a `boolean` result. If the result is `true`, the program will execute the code in the block that is encompassed by the `if`. If an `else` statement is used and the `if` expression evaluates to `false`, then the code following the `else` will be performed. If no `else` block is defined, then none of the code associated with the `if` statement will execute.

You also learned that the `switch` statement can be used to replace multiple `if-else` statements. The `switch` statement can evaluate integer primitive types that can be implicitly cast to an `int` (those types are `byte`, `short`, `int`, and `char`), or it can evaluate `enums`, and as of Java 7, it can evaluate `Strings`. At runtime, the JVM will try to find a match between the expression in the `switch` statement and a constant in a corresponding `case` statement. If a match is found, execution will begin at the matching case and continue on from there, executing code in all the remaining `case` statements until a `break` statement is found or the end of the `switch` statement occurs. If there is no match, then the `default` case will execute, if there is one.

You've learned about the three looping constructs available in the Java language. These constructs are the `for` loop (including the basic `for` and the enhanced `for`, which was new to Java 5), the `while` loop, and the `do` loop. In general, the `for` loop is used when you know how many times you need to go through the loop. The `while` loop is used when you do not know how many times you want to go through, whereas the `do` loop is used when you need to go through at least once. In the `for` loop and the `while` loop, the expression will have to evaluate to `true` to get inside the block and will check after every iteration of the loop. The `do` loop does not check the condition until after it has gone through the loop once. The major benefit of the `for` loop is the ability to initialize one or more variables and increment or decrement those variables in the `for` loop definition.

The `break` and `continue` statements can be used in either a labeled or unlabeled fashion. When unlabeled, the `break` statement will force the program to stop processing the innermost looping construct and start with the line of code following the loop. Using an unlabeled `continue` command will cause the program to stop execution of the current iteration of the innermost loop and proceed with the next iteration. When a `break` or a `continue` statement is used in a labeled manner, it will perform in the same way, with one exception: the statement will not apply to the innermost loop; instead, it will apply to the loop with the label. The `break` statement is used most often in conjunction with the `switch` statement. When there is a match between the `switch` expression and the `case` constant, the code following the `case` constant will be performed. To stop execution, a `break` is needed.

You've seen how Java provides an elegant mechanism in exception handling. Exception handling allows you to isolate your error-correction code into separate blocks so that the main code doesn't become cluttered by error-checking code. Another elegant feature allows you to handle similar errors with a single error-handling block, without code duplication. Also, the error handling can be deferred to methods further back on the call stack.

You learned that Java's `try` keyword is used to specify a guarded region—a block of code in which problems might be detected. An exception handler is the code that is executed when an exception occurs. The handler is defined by using Java's `catch` keyword. All `catch` clauses must immediately follow the related `try` block.

Java also provides the `finally` keyword. This is used to define a block of code that is always executed, either immediately after a `catch` clause completes or immediately after the associated `try` block in the case that no exception was thrown (or there was a `try` but no `catch`). Use `finally` blocks to release system resources and to perform any cleanup required by the code in the `try` block. A `finally` block is not required, but if there is one, it must immediately follow the last `catch`. (If there is no `catch` block, the `finally` block must immediately follow the `try` block.) It's guaranteed to be called except when the `try` or `catch` issues a `System.exit()`.

An exception object is an instance of class `Exception` or one of its subclasses. The `catch` clause takes, as a parameter, an instance of an object of a type derived from the `Exception` class. Java requires that each method either catches any checked exception it can throw or else declares that it throws the exception. The exception declaration is part of the method's signature. To declare that an exception may be thrown, the `throws` keyword is used in a method definition, along with a list of all checked exceptions that might be thrown.

Runtime exceptions are of type `RuntimeException` (or one of its subclasses). These exceptions are a special case because they do not need to be handled or declared, and thus are known as "unchecked" exceptions. Errors are of type `java.lang.Error` or its subclasses, and like runtime exceptions, they do not need to be handled or declared. Checked exceptions include any exception types that are not of type `RuntimeException` or `Error`. If your code fails either to handle a checked exception or declare that it is thrown, your code won't compile. But with unchecked exceptions or objects of type `Error`, it doesn't matter to the compiler whether you declare them or handle them, do nothing about them, or do some combination of declaring and handling. In other words, you're free to declare them and handle them, but the compiler won't care one way or the other. It's not good practice to handle an `Error`, though, because you can rarely recover from one.

Finally, remember that exceptions can be generated by the JVM, or by a programmer.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. You might want to loop through them several times.

Writing Code Using if and switch Statements (OCA Objectives 3.4 and 3.5)

- The only legal expression in an `if` statement is a `boolean` expression—in other words, an expression that resolves to a `boolean` or a `Boolean` reference.
- Watch out for `boolean` assignments (`=`) that can be mistaken for `boolean` equality (`==`) tests:

```
boolean x = false;
if (x = true) { } // an assignment, so x will always be true!
```

- Curly braces are optional for `if` blocks that have only one conditional statement. But watch out for misleading indentations.
- `switch` statements can evaluate only to `enums` or the `byte`, `short`, `int`, `char`, and, as of Java 7, `String` data types. You can't say this:

```
long s = 30;
switch(s) { }
```

- The `case` constant must be a literal or `final` variable, or a constant expression, including an `enum` or a `String`. You cannot have a case that includes a non-final variable or a range of values.
- If the condition in a `switch` statement matches a `case` constant, execution will run through all code in the `switch` following the matching `case` statement until a `break` statement or the end of the `switch` statement is encountered. In other words, the matching `case` is just the entry point into the `case` block, but unless there's a `break` statement, the matching `case` is not the only `case` code that runs.
- The `default` keyword should be used in a `switch` statement if you want to run some code when none of the `case` values match the conditional value.
- The `default` block can be located anywhere in the `switch` block, so if no preceding `case` matches, the `default` block will be entered, and if the `default` does not contain a `break`, then code will continue to execute (fall-through) to the end of the `switch` or until the `break` statement is encountered.

Writing Code Using Loops (OCA Objectives 5.1, 5.2, 5.3, and 5.4)

- A basic `for` statement has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
- If a variable is incremented or evaluated within a basic `for` loop, it must be declared before the loop or within the `for` loop declaration.
- A variable declared (not just initialized) within the basic `for` loop declaration cannot be accessed outside the `for` loop—in other words, code below the `for` loop won't be able to use the variable.
- You can initialize more than one variable of the same type in the first part of the basic `for` loop declaration; each initialization must be separated by a comma.
- An enhanced `for` statement (new as of Java 5) has two parts: the *declaration* and the *expression*. It is used only to loop through arrays or collections.
- With an enhanced `for`, the *expression* is the array or collection through which you want to loop.
- With an enhanced `for`, the *declaration* is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
- You cannot use a number (old C-style language construct) or anything that does not evaluate to a `boolean` value as a condition for an `if` statement or looping construct. You can't, for example, say `if(x)`, unless `x` is a `boolean` variable.
- The `do` loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue (OCA Objective 5.5)

- An unlabeled `break` statement will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
- An unlabeled `continue` statement will cause the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
- If the `break` statement or the `continue` statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

Handling Exceptions (OCA Objectives 8.1, 8.2, 8.3, and 8.4)

- Exceptions come in two flavors: checked and unchecked.
- Checked exceptions include all subtypes of `Exception`, excluding classes that extend `RuntimeException`.
- Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using `throws`, or handle the exception with an appropriate `try/catch`.
- Subtypes of `Error` or `RuntimeException` are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them or to declare them, but the compiler doesn't care one way or the other.
- If you use an optional `finally` block, it will always be invoked, regardless of whether an exception in the corresponding `try` is thrown or not, and regardless of whether a thrown exception is caught or not.
- The only exception to the `finally`-will-always-be-called rule is that a `finally` will not be invoked if the JVM shuts down. That could happen if code from the `try` or `catch` blocks calls `System.exit()`.
- Just because `finally` is invoked does not mean it will complete. Code in the `finally` block could itself raise an exception or issue a `System.exit()`.
- Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to `main()`, and `main()` is "ducking" the exception by declaring it).
- You can create your own exceptions, normally by extending `Exception` or one of its subtypes. Your exception will then be considered a checked exception (unless you are extending from `RuntimeException`), and the compiler will enforce the handle or declare rule for that exception.
- All `catch` blocks must be ordered from most specific to most general. If you have a `catch` clause for both `IOException` and `Exception`, you must put the `catch` for `IOException` first in your code. Otherwise, the `IOException` would be caught by `catch(Exception e)`, because a `catch` argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining `catch` clauses that can never be reached.
- Some exceptions are created by programmers, and some by the JVM.

SELF TEST

1. (Also an Upgrade topic) Given:

```
public class Flipper {
  public static void main(String[] args) {
    String o = "-";
    switch("FRED".toLowerCase().substring(1,3)) {
      case "yellow":
        o += "y";
      case "red":
        o += "r";
      case "green":
        o += "g";
    }
    System.out.println(o);
  }
}
```

What is the result?

A. `-`

B. `-r`

C. `-rg`

D. Compilation fails

E. An exception is thrown at runtime

2. Given:

```
class Plane {
  static String s = "-";
  public static void main(String[] args) {
    new Plane().s1();
    System.out.println(s);
  }
  void s1() {
    try { s2(); }
    catch (Exception e) { s += "c"; }
  }
  void s2() throws Exception  {
    s3();  s += "2";
    s3();  s += "2b";
  }
  void s3() throws Exception {
    throw new Exception();
  }
}
```

What is the result?

A. `-`

B. `-c`

C. `-c2`

D. `-2c`

E. `-c22b`

F. `-2c2b`

G. `-2c2bc`

H. Compilation fails

3. Given:

```
try { int x = Integer.parseInt("two"); }
```

Which could be used to create an appropriate `catch` block? (Choose all that apply.)

A. `ClassCastException`

B. `IllegalStateException`

C. `NumberFormatException`

D. `IllegalArgumentException`

E. `ExceptionInInitializerError`

F. `ArrayIndexOutOfBoundsException`

4. Given:

```
public class Flip2 {
  public static void main(String[] args) {
    String o = "-";
    String[] sa = new String[4];
    for(int i = 0; i < args.length; i++)
      sa[i] = args[i];
    for(String n: sa) {
      switch(n.toLowerCase()) {
        case "yellow": o += "y";
        case "red":    o += "r";
        case "green":  o += "g";
      }
    }
    System.out.print(o);
  }
}
```

And given the command-line invocation:

```
Java Flip2 RED Green YeLLow
```

Which are true? (Choose all that apply.)

A. The string rgy will appear somewhere in the output

B. The string rgg will appear somewhere in the output

C. The string gyr will appear somewhere in the output

D. Compilation fails

E. An exception is thrown at runtime

5. Given:

```
1. class Loopy {
2.   public static void main(String[] args) {
3.     int[] x = {7,6,5,4,3,2,1};
4.     // insert code here
5.       System.out.print(y + " ");
6.     }
7.   }
8. }
```

Which, inserted independently at line 4, compiles? (Choose all that apply.)

A. `for(int y : x) {`

B. `for(x : int y) {`

C. `int y = 0; for(y : x) {`

D. `for(int y=0, z=0; z<x.length; z++) { y = x[z];`

E. `for(int y=0, int z=0; z<x.length; z++) { y = x[z];`

F. `int y = 0; for(int z=0; z<x.length; z++) { y = x[z];`

6. Given:

```
class Emu {
  static String s = "-";
  public static void main(String[] args) {
    try {
      throw new Exception();
    } catch (Exception e) {
        try {
          try { throw new Exception();
          } catch (Exception ex) { s += "ic "; }
          throw new Exception(); }
        catch (Exception x) { s += "mc "; }
        finally { s += "mf "; }
    } finally { s += "of "; }
    System.out.println(s);
} }
```

What is the result?

A. `-ic of`

B. `-mf of`

C. `-mc mf`

D. `-ic mf of`

E. `-ic mc mf of`

F. `-ic mc of mf`

G. Compilation fails

7. Given:

```
 3. class SubException extends Exception { }
 4. class SubSubException extends SubException { }
 5.
 6. public class CC { void doStuff() throws SubException { } }
 7.
 8. class CC2 extends CC { void doStuff() throws SubSubException { } }
 9.
10. class CC3 extends CC { void doStuff() throws Exception { } }
11.
12. class CC4 extends CC { void doStuff(int x) throws Exception { } }
13.
14. class CC5 extends CC {  void doStuff()  { } }
```

What is the result? (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails due to an error on line 8

C. Compilation fails due to an error on line 10

D. Compilation fails due to an error on line 12

E. Compilation fails due to an error on line 14

8. (OCP only) Given:

```
 3. public class Ebb {
 4.   static int x = 7;
 5.   public static void main(String[] args) {
 6.     String s = "";
 7.     for(int y = 0; y < 3; y++) {
 8.       x++;
 9.       switch(x) {
10.         case 8: s += "8 ";
11.         case 9: s += "9 ";
12.         case 10: { s+= "10 "; break; }
13.         default: s += "d ";
14.         case 13: s+= "13 ";
15.       }
```

```
16.     }
17.     System.out.println(s);
18.   }
19.   static { x++; }
20. }
```

What is the result?

A. `9 10 d`

B. `8 9 10 d`

C. `9 10 10 d`

D. `9 10 10 d 13`

E. `8 9 10 10 d 13`

F. `8 9 10 9 10 10 d 13`

G. Compilation fails

9. Given:

```
 3. class Infinity { }
 4. public class Beyond extends Infinity {
 5.   static Integer i;
 6.   public static void main(String[] args) {
 7.     int sw = (int)(Math.random() * 3);
 8.     switch(sw) {
 9.       case 0: {  for(int x = 10; x > 5; x++)
10.                    if(x > 10000000) x = 10;
11.                  break; }
12.       case 1: {  int y = 7 * i;  break;  }
13.       case 2: {  Infinity inf = new Beyond();
14.                  Beyond b = (Beyond)inf;  }
15.     }
16.   }
17. }
```

And given that line 7 will assign the value 0, 1, or 2 to `sw`, which are true? (Choose all that apply.)

A. Compilation fails

B. A `ClassCastException` might be thrown

C. A `StackOverflowError` might be thrown

D. A `NullPointerException` might be thrown

E. An `IllegalStateException` might be thrown

F. The program might hang without ever completing

G. The program will always complete without exception

10. Given:

```
 3. public class Circles {
 4.   public static void main(String[] args) {
 5.     int[] ia = {1,3,5,7,9};
 6.     for(int x : ia) {
 7.       for(int j = 0; j < 3; j++) {
 8.         if(x > 4 && x < 8) continue;
 9.         System.out.print(" " + x);
10.         if(j == 1) break;
11.         continue;
12.       }
13.       continue;
14.     }
15.   }
16. }
```

What is the result?

A. `1 3 9`

B. `5 5 7 7`

C. `1 3 3 9 9`

D. `1 1 3 3 9 9`

E. `1 1 1 3 3 3 9 9 9`

F. Compilation fails

11. Given:

```
 3. public class OverAndOver {
 4.   static String s = "";
 5.   public static void main(String[] args) {
 6.     try {
 7.       s += "1";
 8.       throw new Exception();
 9.     } catch (Exception e) { s += "2";
10.     } finally { s += "3"; doStuff(); s += "4";
11.     }
12.     System.out.println(s);
13.   }
14.   static void doStuff() { int x = 0; int y = 7/x; }
15. }
```

What is the result?

A. `12`

B. `13`

C. `123`

D. `1234`

E. Compilation fails

F. `123` followed by an exception

G. `1234` followed by an exception

H. An exception is thrown with no other output

12. Given:

```
 3. public class Wind {
 4.   public static void main(String[] args) {
 5.     foreach:
 6.     for(int j=0; j<5; j++) {
 7.       for(int k=0; k< 3; k++) {
 8.         System.out.print(" " + j);
 9.         if(j==3 && k==1) break foreach;
10.         if(j==0 || j==2) break;
11.       }
12.     }
13.   }
14. }
```

What is the result?

A. `0 1 2 3`

B.`1 1 1 3 3`

C. `0 1 1 1 2 3 3`

D. `1 1 1 3 3 4 4 4`

E. `0 1 1 1 2 3 3 4 4 4`

F. Compilation fails

13. Given:

```
 3. public class Gotcha {
 4.   public static void main(String[] args) {
 5.     // insert code here
 6.
 7.   }
 8.   void go() {
 9.     go();
10.   }
11. }
```

And given the following three code fragments:

```
I.   new Gotcha().go();

II.  try { new Gotcha().go(); }
     catch (Error e) { System.out.println("ouch"); }

III. try { new Gotcha().go(); }
     catch (Exception e) { System.out.println("ouch"); }
```

When fragments I–III are added, independently, at line 5, which are true? (Choose all that apply.)

A. Some will not compile
B. They will all compile
C. All will complete normally
D. None will complete normally
E. Only one will complete normally
F. Two of them will complete normally

14. Given the code snippet:

```
String s = "bob";
String[] sa = {"a", "bob"};
final String s2 = "bob";
StringBuilder sb = new StringBuilder("bob");

// switch(sa[1]) {           // line 1
// switch("b" + "ob") {      // line 2
// switch(sb.toString()) {   // line 3

// case "ann":  ;            // line 4
// case s:      ;            // line 5
// case s2:     ;            // line 6
}
```

And given that the numbered lines will all be tested by un-commenting one `switch` statement and one `case` statement together, which line(s) will FAIL to compile? (Choose all that apply.)

A. line 1
B. line 2
C. line 3
D. line 4
E. line 5
F. line 6
G. All six lines of code will compile

15. Given:

```
1. public class Frisbee {
2.   // insert code here
3.     int x = 0;
4.     System.out.println(7/x);
5.   }
6. }
```

And given the following four code fragments:

```
I.   public static void main(String[] args) {
II.  public static void main(String[] args) throws Exception {
III. public static void main(String[] args) throws IOException {
IV.  public static void main(String[] args) throws RuntimeException {
```

If the four fragments are inserted independently at line 2, which are true? (Choose all that apply.)

A. All four will compile and execute without exception

B. All four will compile and execute and throw an exception

C. Some, but not all, will compile and execute without exception

D. Some, but not all, will compile and execute and throw an exception

E. When considering fragments II, III, and IV, of those that will compile, adding a `try/catch` block around line 4 will cause compilation to fail

16. Given:

```
 2. class MyException extends Exception { }
 3. class Tire {
 4.   void doStuff() {  }
 5. }
 6. public class Retread extends Tire {
 7.   public static void main(String[] args) {
 8.     new Retread().doStuff();
 9.   }
10.   // insert code here
11.     System.out.println(7/0);
12.   }
13. }
```

And given the following four code fragments:

```
I.   void doStuff() {
II.  void doStuff() throws MyException {
III. void doStuff() throws RuntimeException {
IV.  void doStuff() throws ArithmeticException {
```

When fragments I–IV are added, independently, at line 10, which are true? (Choose all that apply.)

A. None will compile

B. They will all compile

C. Some, but not all, will compile

D. All of those that compile will throw an exception at runtime

E. None of those that compile will throw an exception at runtime

F. Only some of those that compile will throw an exception at runtime

SELF TEST ANSWERS

1. ☑ **A** is correct. As of Java 7 the code is legal, but the `substring()` method's second argument is exclusive. If the invocation had been `substring(1,4)`, the output would have been `-rg`. Note: We hope you won't have too many exam questions that focus on API trivia like this one. If you knew the switch was legal, give yourself "almost full credit."
   ☒ **B, C, D,** and **E** are incorrect based on the above. (OCA Objectives 2.7 and 3.5, and Upgrade Objective 1.1)

2. ☑ **B** is correct. Once `s3()` throws the exception to `s2()`, `s2()` throws it to `s1()`, and no more of `s2()`'s code will be executed.
   ☒ **A, C, D, E, F, G,** and **H** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

3. ☑ **C** and **D** are correct. `Integer.parseInt` can throw a `NumberFormatException`, and `IllegalArgumentException` is its superclass (that is, a broader exception).
   ☒ **A, B, E,** and **F** are not in `NumberFormatException`'s class hierarchy. (OCA Objective 8.5)

4. ☑ **E** is correct. As of Java 7 the syntax is legal. The `sa[]` array receives only three arguments from the command line, so on the last iteration through `sa[]`, a `NullPointerException` is thrown.
   ☒ **A, B, C,** and **D** are incorrect based on the above. (OCA Objectives 3.5, 5.2, and 8.5, and Upgrade Objective 1.1)

5. ☑ **A, D,** and **F** are correct. **A** is an example of the enhanced `for` loop. **D** and **F** are examples of the basic `for` loop.
   ☒ **B, C,** and **E** are incorrect. **B** is incorrect because its operands are swapped. **C** is incorrect because the enhanced `for` must declare its first operand. **E** is incorrect syntax to declare two variables in a `for` statement. (OCA Objective 5.2)

6. ☑ **E** is correct. There is no problem nesting `try/catch` blocks. As is normal, when an exception is thrown, the code in the `catch` block runs, and then the code in the `finally` block runs.
   ☒ **A, B, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

7. ☑ **C** is correct. An overriding method cannot throw a broader exception than the method it's overriding. Class CC4's method is an overload, not an override.
   ☒ **A, B, D,** and **E** are incorrect based on the above. (OCA Objectives 8.2 and 8.4)

8. ☑ **D** is correct. Did you catch the static initializer block? Remember that switches work on "fall-through" logic, and that fall-through logic also applies to the default case, which is used when no other case matches.
   ☒ **A, B, C, E, F,** and **G** are incorrect based on the above. (OCA Objective 3.5)

9. ☑ **D** and **F** are correct. Because `i` was not initialized, case 1 will throw a `NullPointerException`. Case 0 will initiate an endless loop, not a stack overflow. Case 2's downcast will *not* cause an exception.
☒ **A, B, C, E,** and **G** are incorrect based on the above. (OCA Objectives 3.5 and 8.4)

10. ☑ **D** is correct. The basic rule for unlabeled `continue` statements is that the current iteration stops early and execution jumps to the next iteration. The last two `continue` statements are redundant!
☒ **A, B, C, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

11. ☑ **H** is correct. It's true that the value of `String` s is `123` at the time that the divide-by-zero exception is thrown, but `finally()` is *not* guaranteed to complete, and in this case `finally()` never completes, so the `System.out.println` (S.O.P) never executes.
☒ **A, B, C, D, E, F,** and **G** are incorrect based on the above. (OCA Objective 8.2)

12. ☑ **C** is correct. A `break` breaks out of the current innermost loop and carries on. A labeled `break` breaks out of and terminates the labeled loops.
☒ **A, B, D, E,** and **F** are incorrect based on the above. (OCA Objectives 5.2 and 5.5)

13. ☑ **B** and **E** are correct. First off, `go()` is a badly designed recursive method, guaranteed to cause a `StackOverflowError`. Since `Exception` is not a superclass of `Error`, catching an `Exception` will not help handle an `Error`, so fragment III will not complete normally. Only fragment II will catch the `Error`.
☒ **A, C, D,** and **F** are incorrect based on the above. (OCA Objectives 8.1, 8.2, and 8.4)

14. ☑ **E** is correct. A `switch`'s `cases` must be compile-time constants or `enum` values.
☒ **A, B, C, D, F,** and **G** are incorrect based on the above. (OCA Objective 3.5 and Upgrade Objective 1.1)

15. ☑ **D** is correct. This is kind of sneaky, but remember that we're trying to toughen you up for the real exam. If you're going to throw an `IOException`, you have to import the java.io package or declare the exception with a fully qualified name.
☒ **A, B, C,** and **E** are incorrect. **A, B,** and **C** are incorrect based on the above. **E** is incorrect because it's okay both to handle and declare an exception. (OCA Objectives 8.2 and 8.5)

16. ☑ **C** and **D** are correct. An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However, an overriding method *can* throw `RuntimeExceptions` not thrown by the overridden method.
☒ **A, B, E,** and **F** are incorrect based on the above. (OCA Objective 8.1)

1

Packaging, Compiling, and Interpreting Java Code

CERTIFICATION OBJECTIVES

- The Java Platform
- Understand Packages
- Understand Package-Derived Classes
- Understand Class Structure
- Compile and Interpret Java Code

✓ Two-Minute Drill

Q&A Self Test

Since you are holding this book, or reading an electronic version of it, you must have an affinity for Java. You must also have the desire to let everyone know through the Oracle Certified Associate, Java SE 8 Programmer (OCA), certification process that you are truly Java savvy. As such, you should either be—or have the desire to be—a Java programmer, and in the long term, a true Java developer. You may be or plan to be a project manager heading up a team of Java programmers and/or developers. In this case, you will need to acquire a basic understanding of the Java language and its technologies. In either case, this book is for you.

To start, you may be wondering about the core functional elements provided by the basic Java Standard Edition (SE) platform with regard to libraries and utilities, and how these elements are organized. This chapter answers these questions by discussing Java packages and classes, along with their packaging, structuring, compilation, and interpretation processes.

When you have finished this chapter, you will have a firm understanding of packaging Java classes, high-level details of common Java SE packages, and the fundamentals of Java's compilation and interpretation tools.

CERTIFICATION OBJECTIVE

The Java Platform

Exam Objective Compare and contrast the features and components of Java, such as platform independence, object orientation, encapsulation, and so on

The Java language was first released in 1995 as a beta. At that time the Java team had a radical vision. They envisioned a language that was independent of the platform it was running on. They also wanted to create a language that was object oriented at its core and that used all the principles that this implied. Encapsulation, polymorphism, inheritance, and abstraction are all basic concepts upon which Java is built. This section will review the core philosophy that makes up the Java language.

Platform Independence

When the Java language is compiled, it is targeted for execution on the Java virtual machine, or JVM, instead of a specific hardware architecture. The compiled Java code is called *bytecode*. This is why it is possible to compile the Java language on a

Windows PC and execute the output on a Linux server. The only requirement for the code to work on any computer is the presence of a compatible JVM.

The Java language extends past the PC and server, however. Many mobile phones have embraced the power of Java as their recommended language for apps. This allows the hardware manufacturer to change the hardware between models without breaking the compatibility of the software. Java is even present in embedded systems and appliances. On devices such as Blu-ray players and car infotainment systems, Java software is often present.

It is important that you understand that platform independence does not mean your server code will run on your Blu-ray player. Java has a few different JVM specs for devices with different capabilities. For example, embedded systems use a JVM with only a subset of features, and mobile phones typically use a JVM with mobile optimized user interface libraries. All of these JVMs share a common Java core, but platform independence is limited to compatible versions.

Java's Object-Oriented Philosophy

Java was conceived as an object-oriented language, in contrast to the C language, which is procedural. An object-oriented language organizes related data and code together—a process called *encapsulation*. A properly encapsulated object uses data protection and exposes only some of its data and methods. The data and methods that are designed for internal use in the object are not exposed to other objects.

Object-oriented design also encourages *abstraction*, the ability to generalize algorithms. Abstraction facilitates code reuse and flexibility. These concepts are at the heart of the Java language. Inheritance and polymorphism are key concepts in creating reusable code. Both are covered in much more depth in Chapters 7 and 8 of this book.

Robust and Secure

Security and robustness were major design goals when Java was created. C and C++ suffered from the misuse of pointers, memory management, and buffer overruns. Java was architected to overcome these issues and many more.

Java was designed not to have explicit pointers. In the C language family, pointers store a memory address to an object. This memory address can be directly altered. Java variables store references to objects but do not allow access to, or modification of, the memory address stored in the reference. This simplified development and removed a level of complexity that was often the source of application instability.

Memory management was addressed in Java with the JVM's built-in garbage collector. When Java was introduced, many languages relied on explicit memory

management. This meant that the developer was responsible for both allocating and deallocating the memory that was used for objects. This process could become tedious. If it was done incorrectly, the application could leak memory and/or crash. With Java, the JVM periodically runs the garbage collector, which looks for any objects that have gone out of scope or that are no longer referenced, and it automatically deallocates their memory. This frees the developer from this manual, error-prone task and increases robustness by ensuring that memory is properly managed.

Buffer overruns are a common exploit vector found in software that does not check for them. In a C program, when an array is created, the index used is never automatically checked to ensure it is in bounds. In fact, an out-of-bounds index may not even crash the program. The software will read or write to the memory address whether it is in bounds or out, and this can create unpredictable behavior. This can be used maliciously to alter the program in ways the developer never intended. Java automatically checks the bounds of arrays. If an index is out-of-bounds, an exception is thrown. This level of checking helps create both more robust and secure software.

CERTIFICATION OBJECTIVE

Understand Packages

Exam Objective Import other Java packages to make them accessible in your code

Packaging is a common approach used to organize related classes and interfaces. Most reusable code is packaged. Unpackaged classes are commonly found in books and online tutorials, as well as in software applications with a narrow focus. This section will show you how and when to package your Java classes and how to import external classes from your Java packages. The following topics will be covered:

- Package design
- Package and import statements

Package Design

Packages are considered containers for classes, but they actually define where classes will be located in the hierarchical directory structure. Packaging is encouraged by Java coding standards to decrease the likelihood of classes colliding in the same

TABLE 1-1

Package Attribute Considerations

Package Attribute	Benefits of Applying the Package Attribute
Class coupling	Package dependencies are reduced with class coupling.
System coupling	Package dependencies are reduced with system coupling.
Package size	Typically, larger packages support reusability, whereas smaller packages support maintainability.
Maintainability	Often, software changes can be limited to a single package when the package houses focused functionality.
Naming	Consider conventions when naming your packages. Use a reverse domain name for the package structure. Use lowercase characters delimited with underscores to separate words in package names.

namespace. The package name plus the class names creates the *fully qualified class name*. Packaging your classes also promotes code reuse, maintainability, and the object-oriented principle of encapsulation and modularity.

When you design Java packages, such as the grouping of classes, consider the key areas shown in Table 1-1.

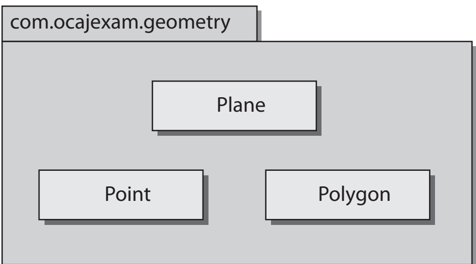
Let's take a look at a real-world example. As program manager, suppose you need two sets of classes with unique functionality that will be used by the same end product. You task Developer A to build the first set and Developer B to build the second. You do not define the names of the classes, but you do define the purpose of the package and what it must contain. Developer A is to create several geometry-based classes, including a point class, a polygon class, and a plane class. Developer B is to build classes that will be included for simulation purposes, including objects such as hot air balloons, helicopters, and airplanes. You send them off to build their classes (without having them package their classes).

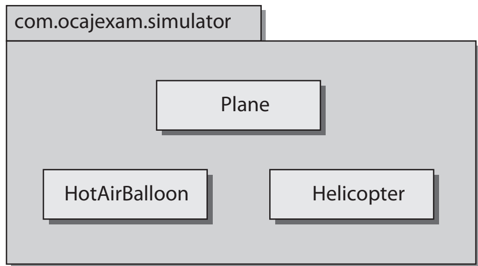
Come delivery time, they both give you a class named `Plane.java`—that is, one for the geometry plane class and one for the airplane class. Now you have a problem, because both of these source files (class files, too) cannot coexist in the same directory because they have the same name. The solution is packaging. If you had designated package names to the developers, this conflict never would have happened (as shown in Figure 1-1). The lesson learned is this: Always package your code, unless your coding project is trivial in nature.

package and import Statements

You should now have a general idea of when and why to package your source files. Next, you need to know exactly how to do this. To place a source file into a package, you use the `package` statement at the beginning of that file. You may use zero or

FIGURE 1-1 Separate packaging of classes with the same names





one `package` statements per source file. To import classes from other packages into your source file, you may use the `import` statement or you may precede each class name with its package name. The `java.lang` package that houses the core language classes is imported by default.

The following code listing shows usage of the `package` and `import` statements. You can return to this listing as we discuss the `package` and `import` statements in detail throughout the chapter.

```
package com.ocaj.exam.tutorial; // Package statement
/* Imports class ArrayList from the java.util package */
import java.util.ArrayList;
/* Imports all classes from the java.io package */
import java.io.*;
public class MainClass {
  public static void main(String[] args) {
    /* Creates console from java.io package - run outside your
IDE */
    Console console = System.console();
    String planet = console.readLine(" \nEnter your favorite
    planet: " );
    /* Creates list for planets */
    ArrayList planetList = new ArrayList();
    planetList.add(planet); // Adds users input to the list
    planetList.add("Gliese 581 c"); // Adds a string to the list
    System.out.println(" \nTwo cool planets: "  + planetList);
  }
}
$ Enter your favorite planet: Jupiter
$ Two cool planets: [Jupiter, Gliese 581 c]
```

The package Statement

The `package` statement includes the `package` keyword, followed by the package path delimited with periods. Table 1-2 shows valid examples of `package` statements. `package` statements have the following attributes:

- They are optional.
- They are limited to one per source file.
- Standard coding convention for `package` statements reverses the domain name of the organization or group creating the package. For example, the owners of the domain name `ocajexam.com` may use the following package name for a utilities package: `com.ocajexam.utilities`.
- Package names equate to directory structures. The package name `com.ocajexam.utils` would equate to the directory com/ocajexam/utils. If a class includes a `package` statement that does not map to the relative directory structure, the class will not be usable.
- The package names beginning with `java.*` and `javax.*` are reserved.
- Package names should be lowercase. Individual words within the package name should be separated by underscores.

The Java SE API contains several packages. These packages are detailed in Oracle's Online Javadoc documentation at http://docs.oracle.com/javase/8/docs/api/.

On the exam, you will see packages for the Java Abstract Window Toolkit API, the Java Swing API, the Java Basic Input/Output API, the Java Networking API, the Java Utilities API, and the core Java Language API. You will need to know the basic functionality that each package/API contains.

The import Statement

An `import` statement enables you to include source code from other classes into a source file at compile time. The `import` statement includes the `import` keyword followed by the package path delimited with periods and ending with a class

TABLE 1-2

Valid package Statements

Package Statement	Related Directory Structure
`package java.net;`	*[directory_path]\java\net*
`package com.ocajexam.utilities;`	*[directory_path]\com\ocajexam\utilities*
`package package_name;`	*[directory_path]\package_name*

TABLE 1-3 Valid import Statements

Import Statement	Definition
`import java.net.*;`	Imports all the classes from the package `java.net`
`import java.net.URL;`	Imports only the `URL` class from the package `java.net`
`import static java.awt .Color.*;`	Imports all static members of the `Color` class of the package `java.awt` (J2SE 5.0 onward only)
`import static java.awt. color.ColorSpace.CS_GRAY;`	Imports the static member `CS_GRAY` of the `ColorSpace` class of the package `java.awt` (J2SE 5.0 onward only)

name or an asterisk, as shown in Table 1-3. These `import` statements occur after the optional `package` statement and before the class definition. Each `import` statement can relate to one package only.

For maintenance purposes, it is better that you import your classes explicitly. This will allow the programmer to determine quickly which external classes are used throughout the class. For example, rather than using `import java .util.*`, use `import java.util.Vector`. In this real-world example, the coder would quickly see (with the latter approach) that the class imports only one class and it is a collection type. In this case, it is a legacy type and the determination to update the class with a newer collection type could be done quickly.

SCENARIO & SOLUTION

To paint basic graphics and images, which package should you use?	Use the Java AWT API package. `import java.awt.*;`
To use data streams, which package should you use?	Use the Java Basic I/O package. `import java.io.*;`
To develop a networking application, which package should you use?	Use the Java Networking API package. `import java.net.*;`
To work with the collections framework, event model, and date/time facilities, which package should you use?	Use the Java Utilities API package. `import java.util.*;`
To utilize the core Java classes and interfaces, which package should you use?	Use the core Java Language package, which is imported by default. `import java.lang.*;`

C and C++ programmers will see some look-and-feel similarities between Java's `import` statement and C/C++'s `#include` statement, even though there is no direct mapping in functionality.

The static import Statement

Static `import` statements were introduced in Java SE 5.0. Simply put, static `import` statements allow you to import static members. The following example statements demonstrate this:

```
/* Import static member ITALY */
import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
...

/* Imports all static members in class Locale */
import static java.util.Locale.*;
...
System.out.println("Locale: " + ITALY); // Prints "Local: it_IT"
System.out.println("Locale: " + GERMANY); // Prints "Local: de_DE"
System.out.println("Locale: " + JAPANESE); // Print "Local: ja"
...
```

Without the static `import` statements shown in the example, the direct references to `ITALY`, `GERMANY`, and `JAPANESE` would be invalid and would cause compilation issues.

```
// import static java.util.Locale.ITALY;
...
System.out.println("Locale: " + ITALY); // Won't compile
```

EXERCISE 1-1

Replacing Implicit import Statements with Explicit import Statements

Consider the following sample application:

```
import java.io.*;
import java.text.*;
import java.time.*;
import java.time.format.*;
import java.util.*;
import java.util.logging.*;
```

```
public class TestClass {
  public static void main(String[] args) throws IOException {
    /* Ensure directory has been created */
    Files.createDirectories(Paths.get("logs"));
    /* Get the date to be used in the filename */
    DateTimeFormatter df
      = DateTimeFormatter.ofPattern("yyyyMMdd_hhmm");
    LocalDateTime now = LocalDateTime.now();
    String date = now.format(df);
    /* Set up the filename in the logs directory */
    String logFileName = "logs\\testlog-" + date + ".txt";
    /* Set up Logger */
    FileHandler myFileHandler = new FileHandler(logFileName);
    myFileHandler.setFormatter(new SimpleFormatter());
    Logger ocajLogger = Logger.getLogger("OCAJ Logger");
    ocajLogger.setLevel(Level.ALL);
    ocajLogger.addHandler(myFileHandler);
    /* Log Message */
    ocajLogger.info("\nThis is a logged information message. ");
    /* Close the file */
    myFileHandler.close();
  }
}
```

There can be implicit `import` statements that allow all necessary classes of a package to be imported:

```
import java.io.* ; // Implicit import example
```

There can be explicit `import` statements that allow only the designated class or interface of a package to be imported:

```
import java.io.File ; // Explicit import example
```

This exercise will have you using explicit `import` statements in lieu of the implicit `import` statements for all of the necessary classes of the sample application. If you are unfamiliar with compiling and interpreting Java programs, finish reading this chapter and then come back to this exercise. Otherwise, let's begin.

1. Type the sample application into a new file and name it *TestClass.java*. Save the file.
2. Compile and run the application to ensure that you have created the file contents without error: `javac TestClass.java` to compile, `java TestClass` to run. Verify that the log message prints to the screen. Also verify that a file has been created in the logs subdirectory with the same message in it.

3. Comment out all of the `import` statements:

```
//import java.io.*;
//import java.text.*;
// import java.time.*;
// import java.time.format.*;
//import java.util.*;
//import java.util.logging.*;
```

4. Compile the application: `javac TestClass.java`. You will be presented with several compiler errors related to the missing class imports. As an example, the following illustration demonstrates the errors that are displayed when only the `java.io` package has been commented out:

```
C:\Windows\system32\cmd.exe

c:\code>javac TestClass.java
TestClass.java:7: error: cannot find symbol
public static void main (String [] args) throws IOException {
                                                ^
  symbol:   class IOException
  location: class TestClass
TestClass.java:9: error: cannot find symbol
new File ("TEST").mkdir();
    ^
  symbol:   class File
  location: class TestClass
2 errors

c:\code>
```

5. For each class that cannot be found, use the online Java Specification API to determine which package it belongs to and then update the source file with the necessary explicit `import` statement. Once completed, you will have replaced the four *implicit* `import` statements with nine *explicit* `import` statements.
6. Run the application again to ensure that the application works with the explicit `import` statements the same way it did with the implicit `import` statements.

Understand Package-Derived Classes

Oracle includes more than 200 packages in the Java SE 8 API. Each package has a specific focus. Fortunately, you need to be familiar with only a few of them for the OCA exam. These may include packages for Java utilities, basic input/output, networking, Abstract Window Toolkit (AWT), Swing, and data/time. The java data/time classes will be covered in more detail in Chapter 10.

The following sections address these APIs:

- Java Utilities API
- Java Basic Input/Output API
- Java Networking API
- Java Abstract Window Toolkit API
- Java Swing API
- JavaFX

Java Utilities API

The Java Utilities API is contained in the package `java.util`. This API provides functionality for a variety of utility classes. The API's key classes and interfaces can be divided into several categories. Categories of classes that may be seen on the exam include the Java Collections Framework, date and time facilities, internationalization, and some miscellaneous utility classes.

Of these categories, the Java Collections Framework pulls the most weight because it is frequently used and provides the fundamental data structures necessary to build valuable Java applications. Table 1-4 details the classes and interfaces of the Collections API that you may see referenced on the exam.

To assist collections in sorting where the ordering is not natural, the Collections API provides the `Comparator` interface. Similarly, the `Comparable` interface that resides in the `java.lang` package is used to sort objects by their natural ordering.

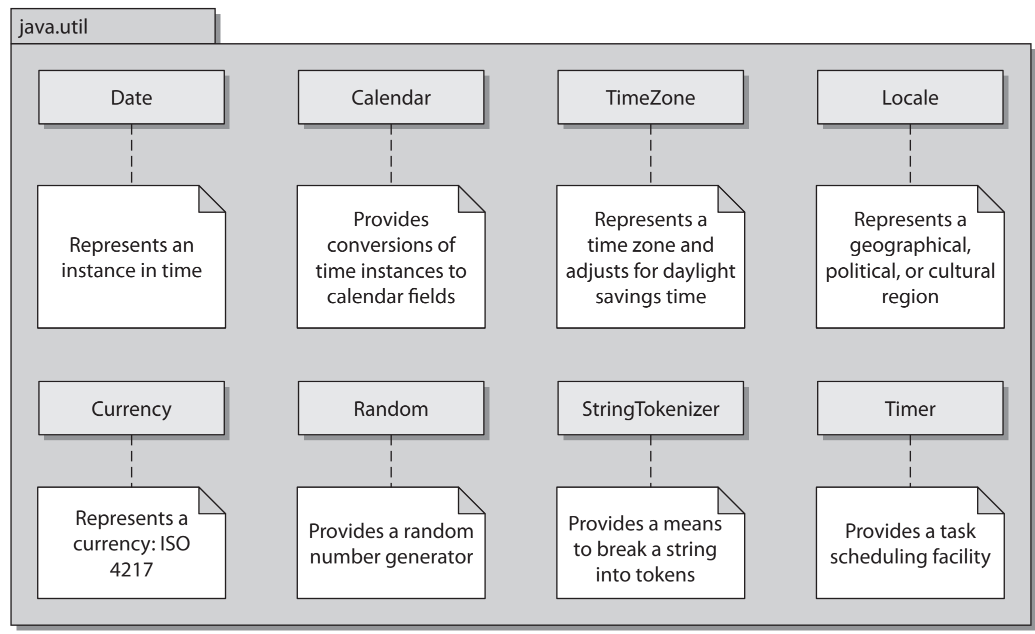
TABLE 1-4 Various Classes of the Java Collections Framework

Interface	Implementations	Description
`List`	`ArrayList`, `LinkedList`, `Vector`	Data structures based on positional access.
`Map`	`HashMap`, `Hashtable`, `LinkedHashMap`, `TreeMap`	Data structures that map keys to values.
`Set`	`HashSet`, `LinkedHashSet`, `TreeSet`	Data structures based on element uniqueness.
`Queue`	`PriorityQueue`	Queues typically order elements in a first in, first out (FIFO) manner. Priority queues order elements according to a supplied comparator.

Various other classes and interfaces reside in the `java.util` package. Legacy date and time facilities are represented by the `Date`, `Calendar`, and `TimeZone` classes. Geographical regions are represented by the `Locale` class. The `Currency` class represents currencies per the ISO 4217 standard. A random-number generator is provided by the `Random` class. And `StringTokenizer` breaks strings into tokens. Several other classes exist within `java.util`, and these (and the collection interfaces and classes) are classes that you may find yourself commonly using on the job. These utilities classes are represented in Figure 1-2.

Many packages have related classes and interfaces with unique functionality, so they are included in their own subpackages. For example, regular expressions are stored in a subpackage of the Java utilities (`java.util`) package. The subpackage is named `java.util.regex` and houses the `Matcher` and `Pattern` classes. Where needed, consider creating subpackages for your own projects.

FIGURE 1-2 Various utility classes



Java Basic Input/Output API

The Java Basic Input/Output API is contained in the package `java.io`. This API provides functionality for general system input and output in relation to data streams, serialization, and the file system. Data-stream classes include byte-stream subclasses of the `InputStream` and `OutputStream` classes. Data-stream classes also include character-stream subclasses of the `Reader` and `Writer` classes. Figure 1-3 depicts part of the class hierarchy for the `Reader` and `Writer` abstract classes.

Other important `java.io` classes and interfaces include `File`, `FileDescriptor`, `FilenameFilter`, and `RandomAccessFile`. The `File` class provides a representation of file and directory pathnames. The `FileDescriptor` class provides a means to function as a handle for opening files and sockets. The `FilenameFilter` interface, as its name implies, defines the functionality to filter filenames. The `RandomAccessFile` class allows for the reading and writing of files to specified locations.

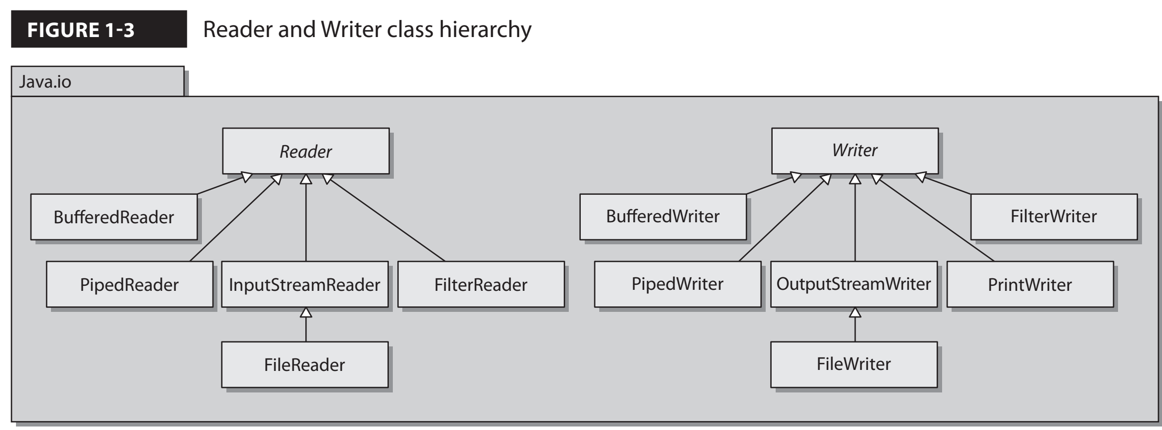
In JDK 7, the NIO.2 API was introduced in the package `java.nio`. This included the useful `Paths` interface, the `Path` class, and the `Files` class. The `Files` class has `lines`, `list`, `walk`, and `find` methods that work hand-in-hand with the Stream API. All of this is beyond the scope of the exam but is useful to know. The following code snippet provides a quick look at what you can do with the API and other new features of Java. You'll see the lambda expression-related code in Chapter 11 (for example, `p -> { statements; }`).

```
// Print out .txt file names in a given folder
try {
  Files.walk(Paths.get("C:\\opt\\dnaProg\\users\\docs")).forEach(p -> {
    if (p.getFileName().toString().endsWith(".txt")) {
      System.out.println("Text doc:" + p.getFileName());
    }
   });
  } catch (IOException e) {
  e.printStackTrace();
}
```

The Java Networking API

The Java Networking API is contained in the package `java.net`. This API provides functionality in support of creating network applications. The API's key classes and interfaces are represented in Figure 1-4. You will probably see few, if any, of these classes on the exam, but the figure will help you conceptualize

FIGURE 1-3 Reader and Writer class hierarchy



what's in the `java.net` package. The improved performance I/O API (`java.nio`) package, which provides for nonblocking networking and the socket factory support package (`javax.net`), is not included on the exam.

Java Abstract Window Toolkit API

The Java Abstract Window Toolkit API is contained in the package `java.awt`. This API provides functionality for creating heavyweight components with regard to creating user interfaces and painting associated graphics and images. The AWT API was Java's original GUI API and has been superseded by the Swing API. Where Swing has been recommended over AWT, certain pieces of the AWT API still

FIGURE 1-4 Various classes of the Networking API

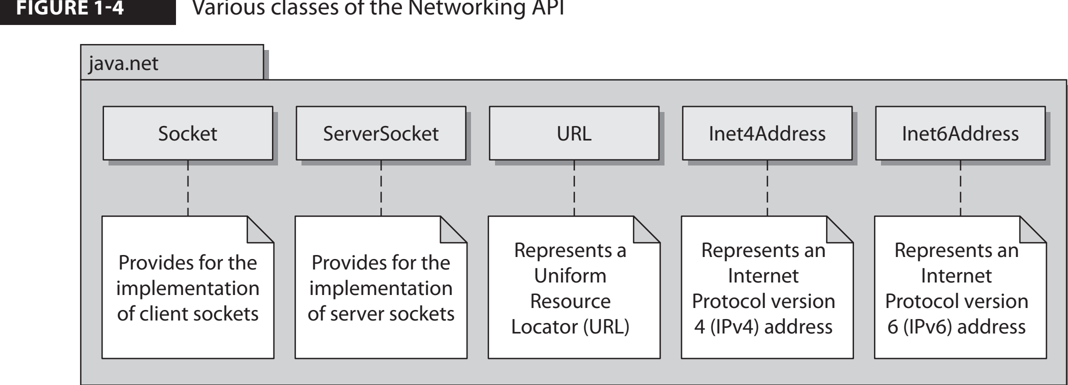
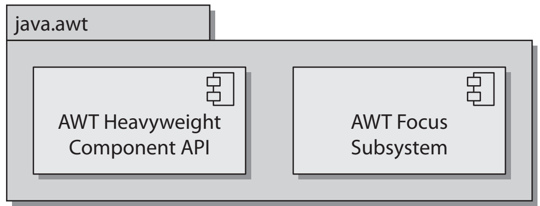


FIGURE 1-5

AWT major elements



remain commonly used, such as the AWT Focus subsystem that was reworked in J2SE 1.4. The AWT Focus subsystem provides for navigation control between components. Figure 1-5 depicts these major AWT elements.

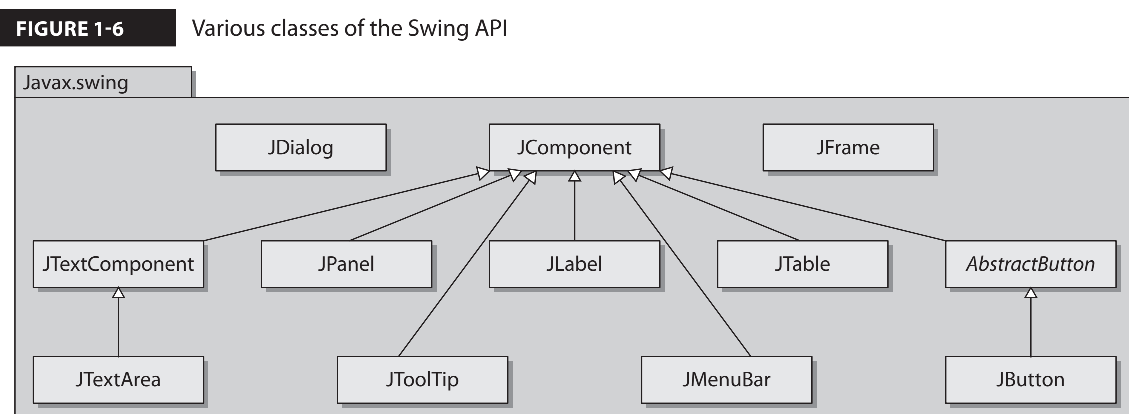
Java Swing API

The Java Swing API is contained in the package `javax.swing`. This API provides functionality for creating lightweight (pure-Java) containers and components. The Swing API, providing a more sophisticated set of GUI components, supersedes the AWT API. Many of the Swing classes are simply prefaced with the addition of "J" in contrast to the legacy AWT component equivalent. For example, Swing uses the class `JButton` to represent a button container, whereas AWT uses the class `Button`.

Swing also provides look-and-feel support, allowing for universal style changes of the GUI's components. Other features include tooltips, accessibility functionality, an event model, and enhanced components such as tables, trees, text components,

SCENARIO & SOLUTION	
You need to create basic Java Swing components such as buttons, panes, and dialog boxes. Provide the code to import the necessary classes of a package.	`// Java Swing API package` `import javax.swing.*;`
You need to support text-related aspects of your Swing components. Provide the code to import the necessary classes of a package.	`// Java Swing API text subpackage` `import javax.swing.text.*;`
You need to implement and configure basic pluggable look-and-feel support. Provide the code to import the necessary classes of a package.	`// Java Swing API plaf subpackage` `import javax.swing.plaf.*;`
You need to use Swing event listeners and adapters. Provide the code to import the necessary classes of a package.	`// Java Swing API event subpackage` `import javax.swing.event.*;`

FIGURE 1-6 Various classes of the Swing API



sliders, and progress bars. Some of the Swing API's key classes are represented in Figure 1-6.

The Swing API makes excellent use of subpackages, with 18 of them in Java SE 8. As mentioned earlier, when common classes are separated into their own packages, code usability and maintainability are enhanced.

Swing takes advantage of the model-view-controller (MVC) architecture. The *model* represents the current state of each component. The *view* is the representation of the components on the screen. The *controller* is the functionality that ties the UI components to events. Although understanding the underlying architecture of Swing is important, it's not necessary for the exam. For comprehensive information on the Swing API, look to the book *Swing: A Beginner's Guide*, by Herbert Schildt (McGraw-Hill Professional).

It's good to be familiar with the package prefixes `java` and `javax`. The prefix `java` is commonly used for the core packages. The prefix `javax` is commonly used for packages that comprise Java standard extensions. Take special notice of the prefix usage in the AWT and Swing APIs: `java.awt` and `javax.swing`. Also note that JavaFX will be replacing Swing as the GUI toolkit for Java SE. Its prefix is `javafx`.

JavaFX API

JavaFX is Java's latest technology for creating rich user interfaces. It is designed to provide lightweight, hardware-accelerated interfaces. JavaFX provides a similar set of features to the Swing library. JavaFX is intended to replace Swing in the same manner that Swing replaced AWT. The JavaFX libraries are part of the `javafx` package.

JavaFX best practices suggest that the MVC architecture be used when designing applications. FXML, an XML-based markup language, has been created for defining user interfaces. Many of the more than 60 UI controls can be styled by using Cascading Style Sheets (CSS). These features together represent a powerful new way to create user interfaces. JavaFX makes going from whiteboard design to implemented software faster than ever before. A great reference for JavaFX is *Introducing JavaFX 8 Programming,* by Herbert Schildt (Oracle Press).

JavaFX is the latest technology for creating user interfaces. Oracle is actively promoting this technology as the go-to tool kit. However, the Swing libraries are not going away anytime soon. In Java 8, both JavaFX and Swing are fully supported and can be used interchangeably. The `SwingNode` class allows Swing elements to be embedded in JavaFX. The `JFXPanel` will allow the reverse so that JavaFX elements can be used in a Swing applications.

CERTIFICATION OBJECTIVE

Understand Class Structure

Exam Objective Define the structure of a Java class

You must understand the structure of a Java class to do well on the exam and to have a promising career with Java. It would help to have a fundamental knowledge of Java naming conventions as well as knowledge of the typical separators that are seen in Java source code (such as comment separators and brackets for enclosing entities). These topics are covered in the following sections:

- Naming Conventions
- Separators and Other Java Source Symbols
- Java Class Structure

Naming Conventions

Naming conventions are rules for the usage and application of characters in creation of identifiers, methods, class names, and so forth, throughout your code base. If some of your team members are not applying naming conventions to their code, you should encourage them to do so, for the good of the effort and for maintainability aspects for after the code is deployed.

The popular article, "How to Write Unmaintainable Code," by Roedy Green, is worth reading (http://thc.org/root/phun/unmaintain.html). It brings to light, in a comical way, the challenges that can occur with maintaining code when there is a blatant or intentional disregard to software development best practices. On the flip-side, *The Passionate Programmer: Creating a Remarkable Career in Software Development* ***, by Chad Fowler (Pragmatic Bookshelf, 2009), encourages the software developer to be the best that he or she can be.***

You may encounter people who will come up with their own naming conventions. Although this is better than not applying any convention, an outsider trying to maintain that person's code would need to learn the original convention and apply it as well for consistency. Fortunately, the Java community does subscribe to a shared thought on how naming conventions should be applied to the many different elements in Java. Table 1-5 describes these conventions in a simple manner. When applying

TABLE 1-5 Java Naming Conventions

Element	Lettering	Characteristic	Example
Class name	Begins uppercase, continues CamelCase	Noun	`SpaceShip`
Interface name	Begins uppercase, continues CamelCase	Adjective ending with "able" or "ible" when providing a capability; otherwise a noun	`Dockable`
Method name	Begins lowercase, continues CamelCase	Verb, may include adjective or noun	`orbit`
Instance and static variables names	Begins lowercase, continues CamelCase	Noun	`moon`
Parameters and local variables	Begins lowercase, continues CamelCase if multiple words are necessary	Single words, acronyms, or abbreviations	`lop` (line of position)
Generic type parameters	Single uppercase letter	The letter *T* is recommended	`T`
Constant	All uppercase letters	Multiple words separated by underscores	`LEAGUE`
Enumeration	Begins uppercase, continues CamelCase; the set of objects should be all uppercase	Noun	`enum Occupation {MANNED, SEMI_MANNED, UNMANNED}`
Package	All lowercase letters	Public packages should be the reversed domain name of the org	com.ocajexam.sim

naming conventions, you should strive to use meaningful and unambiguous names. And remember that naming conventions exist for the primary goal of making Java programs more readable, and therefore maintainable. The practice of using CamelCase—using uppercase letters for the first characters in compound words—is part of the Java naming conventions.

Separators and Other Java Source Symbols

The Java programming language makes use of several separators and symbols to aid in the structuring of the source code in a software program. Table 1-6 details these separators and symbols.

TABLE 1-6 Symbols and Separators

Symbol	Description	Purpose
( )	Parentheses	Encloses set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions
{ }	Braces	Encloses blocks of codes, initializes arrays
[ ]	Box brackets	Declares array types, initializes arrays
< >	Angle brackets	Encloses generics
;	Semicolon	Terminates statement at the end of a line
,	Comma	Separates identifiers in variable declarations, separates values, separates expressions in a for loop
.	Period	Delineates package names, selects an object's field or method, supports method chaining
:	Colon	Follows loop labels
' '	Single quotes	Defines a single character
->	Arrow operator	Separates left-side parameters from the right-side expression
" "	Double quotes	Defines a string of characters
//	Forward slashes	Indicates a single-line comment
/* */	Forward slashes with asterisks	Indicates a blocked comment for multiple lines
/** */	Forward slashes with a double and single asterisk	Indicates Javadoc comments

Java Class Structure

Every Java program has at least one class. A Java class has a signature, optional constructors, optional data members (fields), and optional methods, as outlined here:

```
[modifiers] class classIdentifier [extends superClassIdentifier]
[implements interfaceIdentifier1, interfaceIdentifier2, etc.] {
      [data members]
      [constructors]
      [methods]
}
```

Each class may extend one and only one superclass. Each class may implement one or more interfaces. Interfaces are separated by commas.

The following `SpaceShip` class shows typical elements annotated with comments. The file containing this `SpaceShip` class must be called `SpaceShip.java`. Note that the class declaration extends the `Ship` class and implements the `Dockable` interface. The `Dockable` interface includes the `dockShip` method, which is overridden here. `Ship` class methods would be inherited by the `SpaceShip` class. Chapters 4–7 go into more comprehensive details about creating and working with classes.

The following code shows the structure of a typical class:

```
package com.ocajexam.craft_simulator;

public class SpaceShip extends Ship implements Dockable {

  // Data Members
  public enum ShipType {
    FRIGATE, BATTLESHIP, MINELAYER, ESCORT, DEFENSE
  }
  ShipType shipType = ShipType.BATTLESHIP;

  // Constructors
  public SpaceShip() {
    System.out.println("\nSpaceShip created with default ship type.");
  }
  public SpaceShip(ShipType shipType) {
    System.out.println("\nSpaceShip created with specified ship type.");
    this.shipType = shipType;
  }

  // Methods
  @Override
```

```
  public void dockShip () {
    // TODO
  }
  @Override
  public String toString() {
    String shipTypeRefined = this.shipType.name().toLowerCase();
    return "The pirate ship is a " + shipTypeRefined + " ship.";
  }
}
```

This `SpaceShip` class can be instantiated as demonstrated in the following code:

```
package com.ocajexam.craft_simulator;
import com.ocajexam.craft_simulator.PirateShip.ShipType;
public class SpaceShipSimulator {

  public static void main(String[] args) {

    // Create SpaceShip object with default ship type
    SpaceShip ship1 = new SpaceShip ();
    // Prints "The pirate ship is a battleship."
    System.out.println(ship1);

    // Create SpaceShip object with specified ship type
    SpaceShip ship2 = new SpaceShip (ShipType.FRIGATE);
    // Prints "The pirate ship is a frigate ship."
    System.out.println(ship2);
    }
}
```

The override annotation (`@Override`) indicates that a method declaration intends on overriding a method declaration in the class's supertype.

CERTIFICATION OBJECTIVE

Compile and Interpret Java Code

Exam Objective Create executable Java applications with a main method, run a program from the command line, including console output

The Java Development Kit (JDK) includes several utilities for compiling, debugging, and running Java applications. This section details two utilities from the kit: the Java

compiler and the Java interpreter. For more information on the JDK and its other utilities, see Chapter 10.

Java Compiler

Because we'll need a sample application to use for our Java compiler and interpreter exercises, we'll employ the simple `GreetingsUniverse.java` source file, shown in the following listing, throughout the section. This sample includes the main method used as the entry point of the executed code. When the program is started, this is the first method to be called by the JVM. The main method shown here contains one line of code. This line,

```
System.out.println("Greetings, Universe!")
```

will print

```
Greetings, Universe!
```

to standard output. This output would typically be displayed on a Java application that was started from a console.

```
public class GreetingsUniverse {
  public static void main(String[] args) {
    System.out.println("Greetings, Universe!");
  }
}
```

Let's take a look at compiling and interpreting simple Java programs along with their most basic command-line options.

Compiling Your Source Code

The Java compiler is only one of several tools in the JDK. When you have time, inspect the other tools resident in the JDK's bin folder, as shown in Figure 1-7. For the scope of the OCA exam, you will need to know the details surrounding only the compiler and interpreter.

The Java compiler simply converts Java source files into bytecode. The Java compiler's usage is as follows:

```
javac [options] [source files]
```

The most straightforward way to compile a Java class is to preface the Java source files with the compiler utility from the command line: `javac.exe FileName.java`. The `.exe` is the standard executable file extension on Windows machines

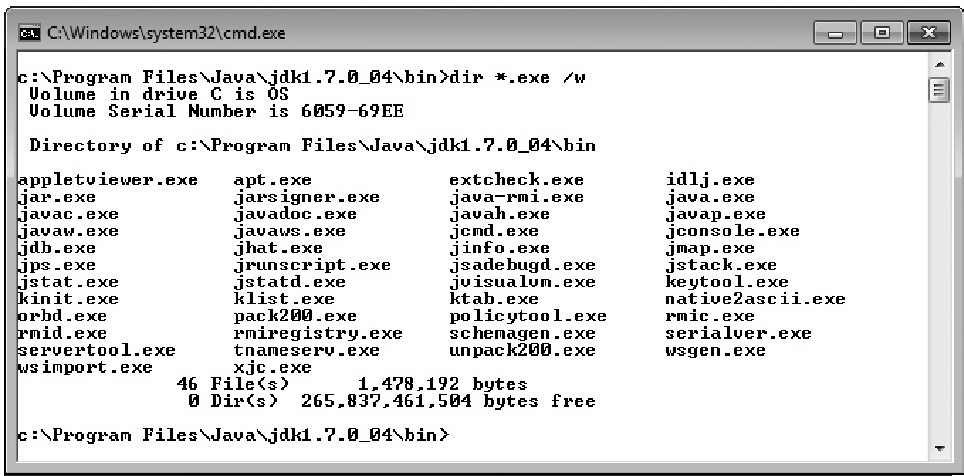


FIGURE 1-7 Java Development Kit utilities

and is optional. The `.exe` extension is not present on executables on UNIX-like systems.

```
javac GreetingsUniverse.java
```

This will result in a bytecode file being produced with the same preface, such as `GreetingsUniverse.class`. This bytecode file will be placed into the same folder as the source code, unless the code is packaged and/or it's been told via a command-line option to be placed somewhere else.

You will find that many projects use Apache Ant and/or Maven build environments. Understanding the fundamentals of the command-line tools is necessary for writing and maintaining the scripts associated with these build products.

Compiling Your Source Code with the -d Option

You may want to specify explicitly where you would like the compiled bytecode class files to go. You can accomplish this by using the `-d` option:

```
javac -d classes GreetingsUniverse.java
```

This command-line structure will place the class file into the classes directory, and since the source code was packaged (that is, the source file included a `package` statement), the bytecode will be placed into the relative subdirectories.

```
[present working directory]\classes\com\ocajexam\tutorial\
    GreetingsUniverse.class
```

INSIDE THE EXAM

Command-Line Tools

Most projects use integrated development environments (IDEs) to compile and execute code. The clear benefit in using IDEs is that building and running code can be as easy as stepping through a couple of menu options or clicking a hot key. The disadvantage is that even though you may establish your settings through a configuration dialog and see the commands and subsequent arguments in one of the workspace windows, you are not getting direct experience in repeatedly creating the complete structure of the commands and associated arguments by hand. The exam is structured to validate that you have experience in scripting compiler and interpreter invocations. Do not take this prerequisite lightly. Take the exam only after you have mastered when and how to use the tools, switches, and associated arguments. At a later time, you can consider taking advantage of the "shortcut" features of popular IDEs such as those provided by NetBeans, Eclipse, IntelliJ IDEA, and JDeveloper.

Compiling Your Code with the -classpath Option

If you want to compile your application with user-defined classes and packages, you may need to tell the JVM where to look by specifying them in the classpath. This classpath inclusion is accomplished by telling the compiler where the desired classes and packages are with the `-cp` or `-classpath` command-line option. In the following compiler invocation, the compiler includes in its compilation any source files that are located under the 3rdPartyCode\classes directory, as well as any classes located in the present working directory (the period). The `-d` option (again) will place the compiled bytecode into the classes directory.

```
javac -d classes -cp 3rdPartyCode\classes\;. GreetingsUniverse
   .java
```

Note that you do not need to include the classpath option if the classpath is defined with the `CLASSPATH` environment variable, or if the desired files are in the present working directory.

On Windows systems, classpath directories are delimited with backward slashes and paths are delimited with semicolons:

```
-classpath .;\dir_a\classes_a\;\dir_b\classes_b\
```

On POSIX-based systems, classpath directories are delimited with forward slashes and paths are delimited with colons:

```
-classpath .:/dir_a/classes_a/:/dir_b/classes_b/
```

Again, the period represents the present (or current) working directory.

exam watch

Know your switches. The designers of the exam will try to throw you by presenting answers with mix-matching compiler and interpreter switches. You may even see some make-believe switches that do not exist anywhere. For additional preparation, query the commands' complete set of switches by typing `java -help` or `javac -help`. Switches are also known as command-line parameters, command-line switches, options, and flags.

Java Interpreter

Interpreting the Java files is the basis for creating the Java application, as shown in Figure 1-8. Let's examine how to invoke the interpreter and its command-line options.

```
java [-options] class [args…]
```

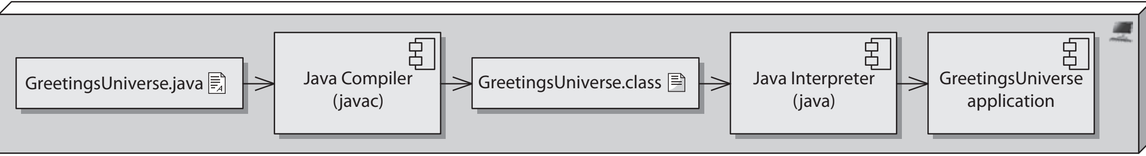
Interpreting Your Bytecode

The Java interpreter is invoked with the `java[.exe]` command. Use it to interpret bytecode and execute your program.

You can easily invoke the interpreter on a class that's not packaged, as follows:

```
java MainClass
```

FIGURE 1-8 Bytecode conversion



You can optionally start the program with the `javaw` command on Microsoft Windows to exclude the command window. This is a nice feature with GUI-based applications, because the console window is often not necessary.

```
javaw.exe MainClass
```

Similarly, on POSIX-based systems, you can use the ampersand to run the application as a background process:

```
java MainClass &
```

Interpreting Your Code with the -classpath Option

When interpreting your code, you may need to define where certain classes and packages are located. You can find your classes at runtime when you include the `-cp` or `-classpath` option with the interpreter. If the classes you want to include are packaged, then you can start your application by pointing the full path of the application to the base directory of classes, as in the following interpreter invocation:

```
java -cp classes com.ocajexam.tutorial.MainClass
```

The delimitation syntax is the same for the `-cp` and `-classpath` options, as defined earlier in the "Compiling Your Code with the -classpath Option" section.

Interpreting Your Bytecode with the -D Option

The `-D` command-line option allows for the setting of new property values. The usage is as follows:

```
java -D<name>=<value> class
```

The following single-file application comprising the `PropertiesManager` class prints out all of the system properties:

```
import java.util.Properties;
public class PropertiesManager {
  public static void main(String[] args) {
    if (args.length == 0) {System.exit(0);}
    Properties props = System.getProperties();
      /* New property example */

      props.setProperty("new_property2", "new_value2");
      switch (args[0]) {
        case "-list_all":
          props.list(System.out); // Lists all properties
```

```
          break;
        case "-list_prop":
          /* Lists value */
          System.out.println(props.getProperty(args[1]));
          break;
        default:
         System.out.println("Usage: java
           PropertiesManager [-list_all]");
         System.out.println("       java
           PropertiesManager [-list_prop [property]]");
         break;
      }
    }
  }
```

Let's run this application while setting a new system property called `new_property1` to the value of `new_value1`:

```
java -Dnew_property1=new_value1 PropertiesManager -list_all
```

You'll see in the standard output that the listing of the system properties includes the new property that we set and its value:

```
...
new_property1=new_value1
java.specification.name=Java Platform API Specification
...
```

Optionally, you can set a value by instantiating the `Properties` class and then setting a property and its value with the `setProperty` method.

To help you conceptualize system properties a little better, Table 1-7 details a subset of the standard system properties.

Retrieving the Version of the Interpreter with the -version Option

The `-version` command-line option is used with the Java interpreter to return the version of the JVM and exit. Don't take the simplicity of the command for granted, as the designers of the exam may try to trick you by including additional arguments after the command. Take the time to toy with the command by adding arguments and putting the `-version` option in various places. Do not make any assumptions about how you think the application will respond. Figure 1-9 demonstrates varying results based on where the `-version` option is used.

Check out the other JDK utilities at your disposal. Java Flight Recorder and Java Mission Control in particular are valuable GUI-based tools that are used to monitor, profile, and collect runtime information.

TABLE 1-7 Subset of System Properties

System Property	Property Description
`file.separator`	The platform-specific file separator (/ for POSIX, \ for Windows)
`java.class.path`	The classpath as defined for the system's environment variable
`java.class.version`	The Java class version number
`java.home`	The directory of the Java installation
`java.vendor`	The vendor supplying the Java platform
`java.vendor.url`	The vendor's Uniform Resource Locator
`java.version`	The version of the Java Interpreter/JVM
`line.separator`	The platform-specific line separator (`\r` on Mac OS 9, `\n` for POSIX, `\r\n` for Microsoft Windows)
`os.arch`	The architecture of the operating system
`os.name`	The name of the operating system
`os.version`	The version of the operating system
`path.separator`	The platform-specific path separator (`:` for POSIX, `;` for Windows)
`user.dir`	The current working directory of the user
`user.home`	The home directory of the user
`user.language`	The language code of the default locale
`user.name`	The username for the current user
`user.timezone`	The system's default time zone

FIGURE 1-9

The -version command-line option

```
C:\Windows\system32\cmd.exe

c:\code>java -version
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java -version INVALID_ARGUMENT
java version "1.7.0_04"
Java(TM) SE Runtime Environment (build 1.7.0_04-b20)
Java HotSpot(TM) 64-Bit Server VM (build 23.0-b21, mixed mode)

c:\code>java HelloWorld -version
Hello, World!

c:\code>_
```

EXERCISE 1-2

Compiling and Interpreting Packaged Software

When you compile and run packaged software from an IDE, the execution process can be as easy as clicking a run icon, as the IDE will maintain the classpath for you and will also let you know if anything is out of sorts. When you try to compile and interpret the code yourself from the command line, you will need to know exactly how to path your files. Consider our sample application that is placed in the `com.ocajexam.tutorial` package (that is, the com/ocajexam/tutorial directory).

```
package com.ocajexam.tutorial;
public class GreetingsUniverse {
  public static void main(String[] args) {
    System.out.println("Greetings, Universe!");
  }
}
```

This exercise will have you compiling and running the application with new classes created in a separate package.

1. Compile the program:

   ```
   javac -d . GreetingsUniverse.java
   ```

2. Run the program to ensure it is error free:

   ```
   java -cp . com.ocajexam.tutorial.GreetingsUniverse
   ```

3. Create three classes named `Earth`, `Mars`, and `Venus` and place them in the `com.ocajexam.tutorial.planets` package. Create constructors that will print the names of the planets to standard output. The details for the `Earth` class are given here as an example of what you will need to do:

   ```
   package com.ocajexam.tutorial.planets;
   public class Earth {
     public Earth {
       System.out.println("Hello from Earth!");
     }
   }
   ```

4. Instantiate each class from the main program by adding the necessary code to the `GreetingsUniverse` class.

   ```
   Earth e = new Earth();
   ```

5. Ensure that all of the source code is in the paths src/com/ocajexam/tutorial/ and src/com/ocajexam/tutorial/planets/.
6. Determine the command-line arguments needed to compile the complete program. Compile the program, and debug where necessary.
7. Determine the command-line arguments needed to interpret the program. Run the program.

   The standard output will read as follows:

```
$ Greetings, Universe!
Hello from Earth!
Hello from Mars!
Hello from Venus!
```

CERTIFICATION SUMMARY

This chapter discussed packaging, structuring, compiling, and interpreting Java code. The chapter started with a discussion on the importance of organizing your classes into packages as well as using the `package` and `import` statements to define and include different pieces of source code. Through the middle of the chapter, we discussed the key features of commonly used Java packages: `java.awt`, `javax.swing`, `java.net`, `java.io`, and `java.util`. We discussed the basic structure of a Java class. We then concluded the chapter by providing detailed information on how to compile and interpret Java source and class files and how to work with their command-line options. At this point, you should be able to (outside of an IDE) package, build, and run basic Java programs independently.

TWO-MINUTE DRILL

Understand Packages

- ☐ Packages are containers for classes.
- ☐ A `package` statement defines the directory path where files are stored.
- ☐ A `package` statement uses periods for delimitation.
- ☐ Package names should be lowercase and separated with underscores between words.
- ☐ Package names beginning with `java.*` and `javax.*` are reserved.
- ☐ There can be zero or one package statement per source file.
- ☐ An `import` statement is used to include source code from external classes.
- ☐ An `import` statement occurs after the optional `package` statement and before the class definition.
- ☐ An `import` statement can define a specific class name to import.
- ☐ An `import` statement can use an asterisk to include all classes within a given package.

Understand Package-Derived Classes

- ☐ The Java Abstract Window Toolkit API is included in the `java.awt` package and subpackages.
- ☐ The `java.awt` package includes GUI creation and painting graphics and images functionality.
- ☐ The Java Swing API is included in the `javax.swing` package and subpackages.
- ☐ The `javax.swing` package includes classes and interfaces that support lightweight GUI component functionality.
- ☐ The Java Basic Input/Output-related classes are contained in the `java.io` package.
- ☐ The `java.io` package includes classes and interfaces that support input/output functionality of the file system, data streams, and serialization.
- ☐ Java networking classes are included in the `java.net` package.

- ❑ The `java.net` package includes classes and interfaces that support basic networking functionality that is also extended by the `javax.net` package.
- ❑ Fundamental Java utilities are included in the `java.util` package.
- ❑ The `java.util` package and subpackages include classes and interfaces that support the Java Collections Framework, legacy collection classes, event model, date and time facilities, and internationalization functionality.

Understand Class Structure

- ❑ Naming conventions are used to make Java programs more readable and maintainable.
- ❑ Naming conventions are applied to several Java elements, including class names, interface names, method names, instance and static variable names, parameter and local variable names, generic type parameter names, constant names, enumeration names, and package names.
- ❑ The preferred order of presenting elements in a class is data members, followed by constructors, followed by methods. Note that the inclusion of each type of element is optional.

Compile and Interpret Java Code

- ❑ The Java compiler is invoked with the `javac[.exe]` command.
- ❑ The `.exe` extension is optional on Microsoft Windows machines and is not present on UNIX-like systems.
- ❑ The compiler's `-d` command-line option defines where compiled class files should be placed.
- ❑ The compiler's `-d` command-line option will include the package location if the class has been declared with a `package` statement.
- ❑ The compiler's `-classpath` command-line option defines directory paths in search of classes.
- ❑ The Java interpreter is invoked with the `java[.exe]` command.
- ❑ The interpreter's `-classpath` switch defines directory paths to use at runtime.
- ❑ The interpreter's `-D` command-line option allows for the setting of system property values.

- ☐ The interpreter's syntax for the `-D` command-line option is `-Dproperty=value`.
- ☐ The interpreter's `-version` command-line option is used to return the version of the JVM and exit.
- ☐ The `-h` command-line option can be applied either to the compiler or the interpreter to print out the tool's usage information.

SELF TEST

Understanding Packages

1. Which two `import` statements will allow for the import of the `HashMap` class?

A. `import java.util.HashMap;`

B. `import java.util.*;`

C. `import java.util.HashMap.*;`

D. `import java.util.hashMap;`

2. Which statement would designate that your file belongs in the package `com.ocajexam.utilities`?

A. `pack com.ocajexam.utilities;`

B. `package com.ocajexam.utilities.*`

C. `package com.ocajexam.utilities.*;`

D. `package com.ocajexam.utilities;`

3. Which of the following is the only Java package that is imported by default?

A. `java.awt`

B. `java.lang`

C. `java.util`

D. `java.io`

Understand Package-Derived Classes

4. The `JCheckBox` and `JComboBox` classes belong to which package?

A. `java.awt`

B. `javax.awt`

C. `java.swing`

D. `javax.swing`

5. Which package contains the Java Collections Framework?

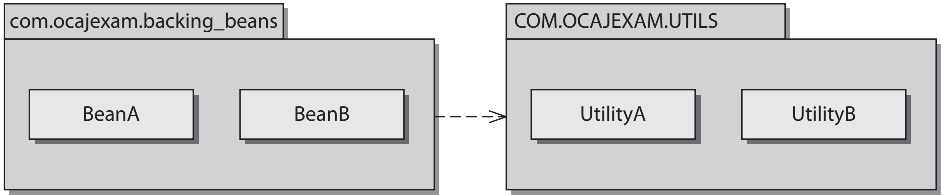
A. `java.io`

B. `java.net`

C. `java.util`

D. `java.utils`

6. The Java Basic I/O API contains what types of classes and interfaces?
   A. Internationalization
   B. RMI, JDBC, and JNDI
   C. Data streams, serialization, and file system
   D. Collection API and data streams
7. Which API provides a lightweight solution for GUI components?
   A. AWT
   B. Abstract Window Toolkit
   C. Swing
   D. AWT and Swing
8. Consider the following illustration. What problem exists with the packaging? You may wish to reference Appendix G of the Unified Modeling Language (UML) for assistance.



   A. You can have only one class per package.
   B. Packages cannot have associations between them.
   C. Package `com.ocajexam.backing_beans` fails to meet the appropriate package naming conventions.
   D. Package `COM.OCAJEXAM.UTILS` fails to meet the appropriate package naming conventions.

Understand Class Structure

9. When apply naming conventions, which Java elements should start with an uppercase letter and continue on using the CamelCase convention?
   A. Class names
   B. Interface names
   C. Constant names
   D. Package names
   E. All of the above

10. When instantiating an object with generics, should you use angle brackets, box brackets, curly brackets, or double-quotation marks to enclose the generic type? Select the appropriate answer.

A. List<Integer> a = new ArrayList<Integer>();

B. List[Integer] a = new ArrayList[Integer]();

C. List{Integer} a = new ArrayList{Integer}();

D. List"Integer" a = new ArrayList"Integer"();

11. When you're organizing the elements in a class, which order is preferred?

A. Data members, methods, constructors

B. Data members, constructors, methods

C. Constructors, methods, data members

D. Constructors, data members, methods

E. Methods, constructors, data members

Compile and Interpret Java Code

12. Which usage represents a valid way of compiling a Java class?

A. java MainClass.class

B. javac MainClass

C. javac MainClass.source

D. javac MainClass.java

13. Which two command-line invocations of the Java interpreter return the version of the interpreter?

A. java -version

B. java --version

C. java -version ProgramName

D. java ProgramName -version

14. Which two command-line usages appropriately identify the classpath?

A. javac -cp /project/classes/ MainClass.java

B. javac -sp /project/classes/ MainClass.java

C. javac -classpath /project/classes/ MainClass.java

D. javac -classpaths /project/classes/ MainClass.java

15. Which command-line usages appropriately set a system property value?

A. `java -Dcom.ocajexam.propertyValue=003 MainClass`

B. `java -d com.ocajexam.propertyValue=003 MainClass`

C. `java -prop com.ocajexam.propertyValue=003 MainClass`

D. `java -D:com.ocajexam.propertyValue=003 MainClass`

SELF TEST ANSWERS

Understand Packages

1. Which two `import` statements will allow for the import of the `HashMap` class?

A. `import java.util.HashMap;`

B. `import java.util.*;`

C. `import java.util.HashMap.*;`

D. `import java.util.hashMap;`

Answer:

☑ **A** and **B.** The `HashMap` class can be imported directly via `import java.util.HashMap` or with a wildcard via `import java.util.*;`.

☒ **C** and **D** are incorrect. **C** is incorrect because the answer is a static `import` statement that imports static members of the `HashMap` class, and not the class itself. **D** is incorrect because class names are case sensitive, so the class name `hashMap` does not equate to `HashMap`.

2. Which statement would designate that your file belongs in the package `com.ocajexam.utilities`?

A. `pack com.ocajexam.utilities;`

B. `package com.ocajexam.utilities.*`

C. `package com.ocajexam.utilities.*;`

D. `package com.ocajexam.utilities;`

Answer:

☑ **D.** The keyword `package` is appropriately used, followed by the package name delimited with periods and followed by a semicolon.

☒ **A, B,** and **C** are incorrect. **A** is incorrect because the word `pack` is not a valid keyword. **B** is incorrect because a `package` statement must end with a semicolon, and you cannot use asterisks in `package` statements. **C** is incorrect because you cannot use asterisks in `package` statements.

3. Which of the following is the only Java package that is imported by default?

A. `java.awt`

B. `java.lang`

C. `java.util`

D. `java.io`

Answer:

☑ **B.** The `java.lang` package is the only package that has all of its classes imported by default.

☒ **A, C,** and **D** are incorrect. The classes of packages `java.awt`, `java.util`, and `java.io` are not imported by default.

Understand Package-Derived Classes

4. The `JCheckBox` and `JComboBox` classes belong to which package?

A. `java.awt`

B. `javax.awt`

C. `java.swing`

D. `javax.swing`

Answer:

☑ **D.** Components belonging to the Swing API are generally prefaced with a uppercase *J*. Therefore, `JCheckBox` and `JComboBox` would be part of the Java Swing API and not the Java AWT API. The Java Swing API base package is `javax.swing`.

☒ **A, B,** and **C** are incorrect. **A** is incorrect because the package `java.awt` does not include the `JCheckBox` and `JComboBox` classes since they belong to the Java Swing API. Note that the package `java.awt` includes the `CheckBox` class, as opposed to the `JCheckBox` class. **B** and **C** are incorrect because the package names `javax.awt` and `java.swing` do not exist.

5. Which package contains the Java Collections Framework?

A. `java.io`

B. `java.net`

C. `java.util`

D. `java.utils`

Answer:

☑ **C.** The Java Collections Framework is part of the Java Utilities API in the `java.util` package.
☒ **A, B,** and **D** are incorrect. **A** is incorrect because the Java Basic I/O API's base package is named `java.io` and does not contain the Java Collections Framework. **B** is incorrect because the Java Networking API's base package is named `java.net` and also does not contain the Collections Framework. **D** is incorrect because there is no package named `java.utils`.

6. The Java Basic I/O API contains what types of classes and interfaces?

A. Internationalization
B. RMI, JDBC, and JNDI
C. Data streams, serialization, and file system
D. Collection API and data streams

Answer:

☑ **C.** The Java Basic I/O API contains classes and interfaces for data streams, serialization, and the file system.
☒ **A, B,** and **D** are incorrect. Internationalization (i18n), RMI, JDBC, JNDI, and the Collections framework are not included in the Basic I/O API.

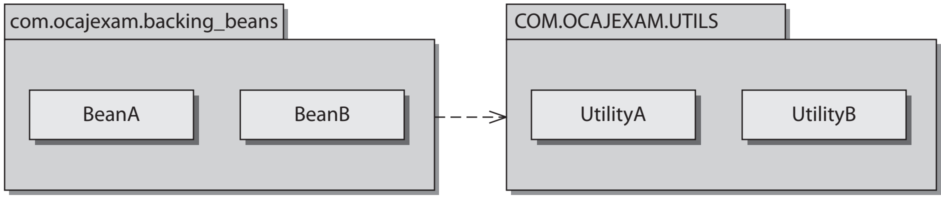
7. Which API provides a lightweight solution for GUI components?

A. AWT
B. Abstract Window Toolkit
C. Swing
D. AWT and Swing

Answer:

☑ **C.** The Swing API provides a lightweight solution for GUI components, meaning that the Swing API's classes render using pure Java code and not native platform widgets.
☒ **A, B,** and **D** are incorrect. AWT and the Abstract Window Toolkit are one and the same and provide a heavyweight solution for GUI components.

8. Consider the following illustration. What problem exists with the packaging? You may wish to reference Appendix G of the Unified Modeling Language (UML) for assistance.



A. You can have only one class per package.

B. Packages cannot have associations between them.

C. Package `com.ocajexam.backing_beans` fails to meet the appropriate package naming conventions.

D. Package `COM.OCAJEXAM.UTILS` fails to meet the appropriate package naming conventions.

Answer:

☑ **D.** `COM.OCAJEXAM.UTILS` fails to meet the appropriate package naming conventions. Package names should be lowercase and should use an underscore between words. However, the words in `ocajexam` are joined in the URL; therefore, excluding the underscore here is acceptable. The package name should read `com.ocajexam.utils`.

☒ **A, B,** and **C** are incorrect. **A** is incorrect because being restricted to having one class in a package is ludicrous. There is no limit. **B** is incorrect because packages can and frequently do have associations with other packages. **C** is incorrect because `com.ocajexam.backing_beans` meets appropriate packaging naming conventions.

Understand Class Structure

9. When apply naming conventions, which Java elements should start with an uppercase letter and continue on using the CamelCase convention?

A. Class names

B. Interface names

C. Constant names

D. Package names

E. All of the above

Answer:

☑ **A** and **B.** Class names and interface names should start with an uppercase letter and continue on using the CamelCase convention.
☒ **C** and **D** are incorrect. **C** is incorrect because constant names should be all uppercase letters separated by underscores. **D** is incorrect because package names do not include uppercase letters, nor do they subscribe to the CamelCase convention.

10. When instantiating an object with generics, should you use angle brackets, box brackets, parentheses, or double-quotation marks to enclose the generic type? Select the appropriate answer.

A. List<Integer> a = new ArrayList<Integer>();

B. List[Integer] a = new ArrayList[Integer]();

C. List{Integer} a = new ArrayList{Integer}();

D. List"Integer" a = new ArrayList"Integer"();

Answer:

☑ **A.** Generics use angle brackets.
☒ **B, C,** and **D** are incorrect. Box brackets (**B**), curly brackets (**C**), and double quotation marks (**D**) are not used to enclose the generic type.

11. When you're organizing the elements in a class, which order is preferred?

A. Data members, methods, constructors

B. Data members, constructors, methods

C. Constructors, methods, data members

D. Constructors, data members, methods

E. Methods, constructors, data members

Answer:

☑ **B.** The preferred order in presenting elements in a class is to present the data members first, followed by constructors, followed by methods.
☒ **A, C, D,** and **E** are incorrect. Although ordering the elements in these manners will not cause any functional or compilation errors, none of these is the preferred order.

Compile and Interpret Java Code

12. Which usage represents a valid way of compiling a Java class?

A. `java MainClass.class`

B. `javac MainClass`

C. `javac MainClass.source`

D. `javac MainClass.java`

Answer:

☑ **D.** The compiler is invoked by the `javac` command. When compiling a Java class, you must include the filename, which houses the main classes, including the `.java` extension.

☒ **A, B,** and **C** are incorrect. **A** is incorrect because `MainClass.class` is bytecode that is already compiled. **B** is incorrect because `MainClass` is missing the `.java` extension. **C** is incorrect because `MainClass.source` is not a valid name for any type of Java file.

13. Which two command-line invocations of the Java interpreter return the version of the interpreter?

A. `java -version`

B. `java --version`

C. `java -version ProgramName`

D. `java ProgramName -version`

Answer:

☑ **A** and **C.** The `-version` flag should be used as the first argument. The application will return the appropriate strings to standard output with the version information and then immediately exit. The second argument is ignored.

☒ **B** and **D** are incorrect. **B** is incorrect because the version flag does not allow double dashes. You may see double dashes for flags in utilities, especially those following the GNU license. However, the double dashes do not apply to the version flag of the Java interpreter. **D** is incorrect because the version flag must be used as the first argument or its functionality will be ignored.

14. Which two command-line usages appropriately identify the classpath?

A. `javac -cp /project/classes/ MainClass.java`

B. `javac -sp /project/classes/ MainClass.java`

C. `javac -classpath /project/classes/ MainClass.java`

D. `javac -classpaths /project/classes/ MainClass.java`

Answer:

☑ **A** and **C.** The option flag that is used to specify the classpath is `-cp` or `-classpath`.
☒ **B** and **D** are incorrect. The option flags `-sp` (**B**) and `-classpaths` (**D**) are invalid.

15. Which command-line usages appropriately set a system property value?

A. `java -Dcom.ocajexam.propertyValue=003 MainClass`

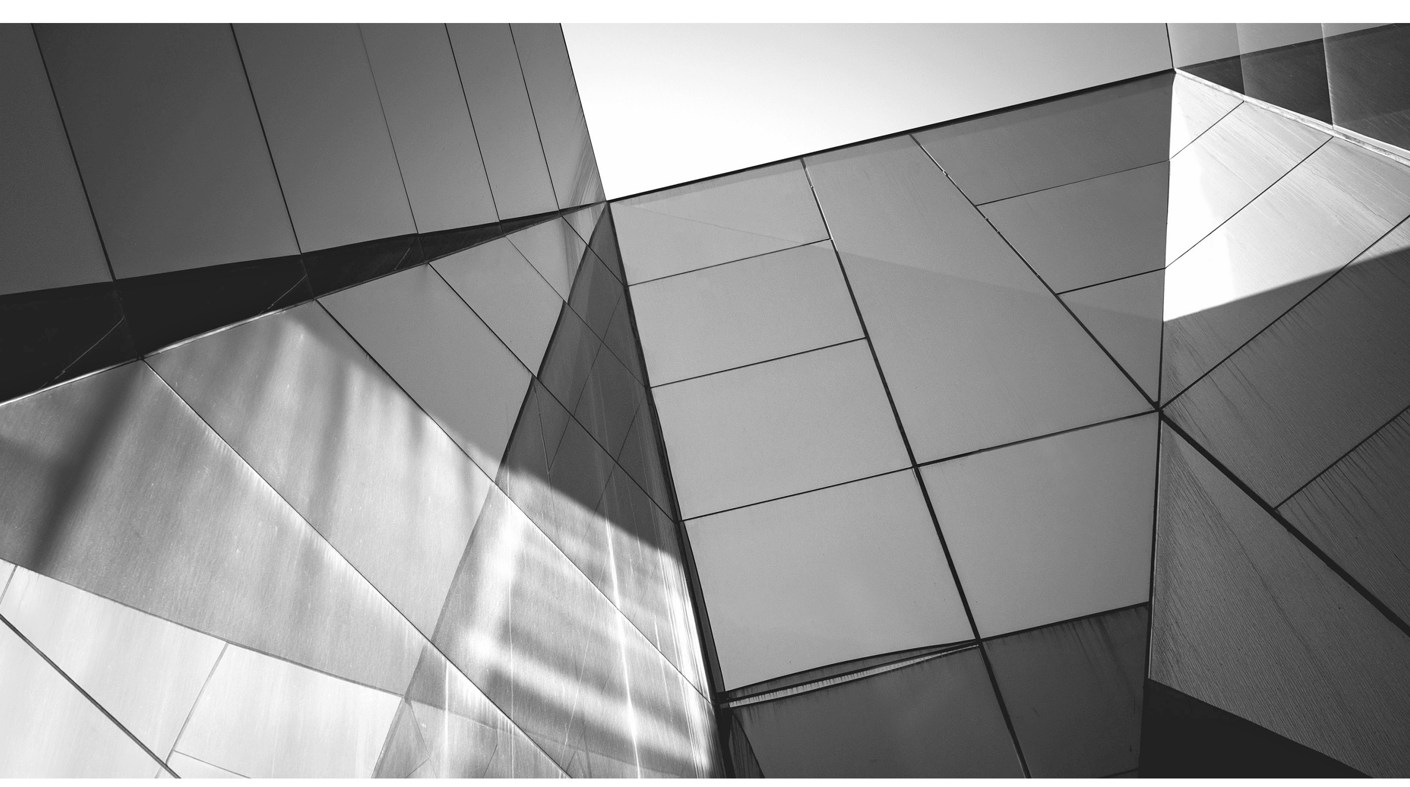
B. `java -d com.ocajexam.propertyValue=003 MainClass`

C. `java -prop com.ocajexam.propertyValue=003 MainClass`

D. `java -D:com.ocajexam.propertyValue=003 MainClass`

Answer:

☑ **A.** The property setting is used with the interpreter, not the compiler. The property name must be sandwiched between the `-D` flag and the equal sign. The desired value should immediately follow the equal sign.
☒ **B, C,** and **D** are incorrect. The `-d` (**B**), `-prop` (**C**), and `-D:` (**D**) flags are invalid ways to designate a system property.



CHAPTER 1

Taking Java to the Next Level

The changes in Java 8 are the biggest in the history of the language, combining coordinated changes to the language, the libraries, and the virtual machine. They promise to alter the way we think about the execution of Java programs and to make the language fit for use in a world, soon to arrive, of massively parallel hardware. Yet for such an important innovation, the actual changes to the language seem quite minor. What is it about these apparently minor modifications that will make such a big difference? And why should we change a programming model that has served us so well throughout the lifetime of Java, and indeed for much longer before that? In this chapter we will explore some of the limitations of that model and see how the lambda-related features of Java 8 will enable Java to evolve to meet the challenges of a new generation of hardware architectures.

1.1 From External to Internal Iteration

Let's start with code that simply iterates over a collection of mutable objects, calling a single method on each of them. The following code fragment constructs a collection of `java.awt.Point` objects (`Point` is a conveniently simple library class, consisting only of a pair (`x`,`y`) of coordinates). Our code then iterates over the collection, translating (i.e., moving) each `Point` by a distance of 1 on both the x and y axes.

```
List<Point> pointList = Arrays.asList(new Point(1, 2), new Point(2, 3));
for (Point p : pointList) {
    p.translate(1, 1);
}
```

Before Java 5 introduced the for-each loop, we would have written the loop like this:

```
for (Iterator pointItr = pointList.iterator(); pointItr.hasNext(); ) {
    ((Point) pointItr.next()).translate(1, 1);
}
```

Or, in a clearer idiom (though less favored because it increases the scope of `pointItr`):

```
Iterator pointItr = pointList.iterator();
while (pointItr.hasNext()) {
    ((Point) pointItr.next()).translate(1, 1);
}
```

Here we are asking `pointList` to create an `Iterator` object on our behalf, and we are then using that object to access the elements of `pointList` in turn. This version is still relevant, because today this is the code that the Java compiler generates to implement the for-each loop. Its key aspect for us is that the order of access to the

elements of `pointList` is controlled by the `Iterator`—there is nothing that we can do to change it. The `Iterator` for an `ArrayList`, for example, will return the elements of the list in sequential order.

Why is this problematic? After all, when the Java Collections Framework was designed in 1998, it seemed perfectly reasonable to dictate the access order of list elements in this way. What has changed since then?

Part of the answer lies in how hardware has been evolving. Workstations and servers have been equipped with multiple processors for a long time, but between the design of the Java Collections Framework in 1998 and the appearance of the first dual-core processors in personal computers in 2005, a revolution had taken place in chip design. A 40-year trend of exponentially increasing processor speed had been halted by inescapable physical facts: signal leakage, inadequate heat dissipation, and the hard truth that, even at the speed of light, data cannot cross a chip quickly enough for further processor speed increases.

But clock speed limitations notwithstanding, the density of chip components continued to increase. So, since it wasn't possible to offer a 6 GHz core, the chip vendors instead began to offer dual-core processors, each core running at 3 GHz. This trend has continued, with currently no end in sight; at the time of the Java 8 ship date (March 2014) quad-core processors have become mainstream, eight-core processors are appearing in the commodity hardware market, and specialist servers have long been available with dozens of cores per processor. The direction is clear, and any programming model that doesn't adapt to it will fail in the face of competition from models that do adapt. Adaptation would mean providing developers with an accessible way of making use of the processing power of multiple cores by distributing tasks over them to be executed in parallel.[1] Failing to adapt, on the other hand, would mean that Java programs, bound by default to a single core, would run at a progressively greater speed disadvantage compared to programs in languages that had found ways to assist users in easily parallelizing their code.

The need for change is shown by the code at the start of this section, which could only access list elements one at a time in the order dictated by the iterator. Collection processing is not the only processor-intensive task that programs have to carry out, but it is one of the most important. The model of iteration embodied in Java's loop constructs forces collection element processing into a serial straitjacket, and that is a serious problem at a time when the most pressing requirement for runtimes—at least as far as performance is concerned—is precisely the opposite: to distribute processing over multiple cores. Although we will see in Chapter 6 that by no means every problem will benefit from parallelization, the best cases give us speedup that is nearly linear in the number of cores.

[1] The distribution of a processing task over multiple processors is often called *parallelization*. Even if we dislike this word, it's a useful shorthand that will sometimes make explanations shorter and more readable.

1.1.1 Internal Iteration

The intrusiveness of the serial model of iteration becomes obvious when we imagine imposing it on a real-world situation. If someone were to ask you to mail some letters with the instruction "repeat the following action: if you have any more letters, take the next one in alphabetical order of addressee's surname and put it in the mailbox," your kindest thought would probably be that they have overspecified the task. You would know that ordering doesn't matter in this task, and neither does the mode—sequential or parallel—of execution, yet it would seem you aren't allowed to ignore them. In this situation you might feel some sympathy with a collection forced by external iteration to process elements serially and in a fixed order when much better strategies may be available.

In reality, all you need to know for that real-world task is that every letter in a bundle needs mailing; exactly how to do that should be up to you. And in the same way, we ought to be able to tell collections *what* should be done to each element they contain, rather than specifying *how*, as external iteration does. If we could do that, what would the code look like? Collections would have to expose a method accepting the "what," namely the task to be executed on each element; an obvious name for this method is `forEach`. With it, we can imagine replacing the iterative code from the start of this section with this:

```
pointList.forEach(/*translate the point by (1,1)*/);
```

Before Java 8 this would have been a strange suggestion, since `java.util.List` (which is the type of `pointList`) has no `forEach` method and, as an interface, cannot have one added. However, in Chapter 7 we'll see that Java 8 overcomes this problem with the introduction of non-abstract interface methods.

The new method `Collection.forEach` (actually inherited by `Collection` from its superinterface `Iterable`) is an example of *internal iteration*, so called because, although the explicit iterative code is no longer obvious, iteration is still taking place internally. It is now managed by the `forEach` method, which applies its *behavioral parameter* to each element of its collection.

The change from external to internal iteration may seem a small one, simply a matter of moving the work of iteration across the client-library boundary. But the consequences are not small. The parallelization work that we require can now be defined in the collection class instead of repeatedly in every client method that must iterate over the collection. Moreover, the implementation is free to use additional techniques such as laziness and out-of-order execution—and, indeed, others yet to be discovered—to get to the answer faster.

So internal iteration is necessary if a programming model is to allow collection library writers the freedom to choose, for each collection, the best way of implementing bulk processing. But what is to replace the comment in the call of `forEach`—how can the collection's methods be told what task is to be executed on each element?

1.1.2 The Command Pattern

There's no need to go outside traditional Java mechanisms to find an answer to this question. For example, we routinely create `Runnable` instances and pass them as arguments. If you think of a `Runnable` as an object representing a task to be executed when its `run` method is called, you can see that what we now require is very similar. For another example, the Swing framework allows the developer to define an action that will be executed in response to a number of different events—menu item selection, button press, etc.—on the user interface. If you are familiar with classical design patterns, you will recognize this loose description of the Command Pattern.

In the case we're considering, what command is needed? Our starting point was a call to the `translate` method of every `Point` in a `List`. So for this example, it appears that `forEach` should accept as its argument an object exposing a method that will call `translate` on each element of the list. If we make this object an instance of a more general interface, `PointAction` say, then we can define different implementations of `PointAction` for different actions that we want to have iteratively executed on `Point` collections:

```
public interface PointAction {
    void doForPoint(Point p);
}
```

Right now, the implementation we want is

```
class TranslateByOne implements PointAction {
    public void doForPoint(Point p) {
        p.translate(1, 1);
    }
}
```

Now we can sketch a naïve implementation of `forEach`:

```
public class PointArrayList extends ArrayList<Point> {
    public void forEach(PointAction t) {
        for (Point p : this) {
            t.doForPoint(p);
        }
    }
}
```

and if we make `pointList` an instance of `PointArrayList`, our goal of internal iteration is achieved with this client code:

```
pointList.forEach(new TranslateByOne());
```

Of course, this toy code is absurdly specialized; we aren't really going to write a new interface for every element type we need to work with. Fortunately, we don't need to; there is nothing special about the names `PointAction` and `doForPoint`; if we simply replace them consistently with other names, nothing changes. In the Java 8 collections library they are called `Consumer` and `accept`. So our `PointAction` interface becomes:

```
public interface Consumer<T> {
    void accept(T t);
}
```

Parameterizing the type of the interface allows us to dispense with the specialized `ArrayList` subclass and instead add the method `forEach` directly to the class itself, as is done by inheritance in Java 8. This method takes a `java.util.function.Consumer`, which will receive and process each element of the collection.

```
public class ArrayList<E> {
    ...
    public void forEach(Consumer<E> c) {
        for (E e : this) {
            c.accept(e);
        }
    }
}
```

Applying these changes to the client code, we get

```
class TranslateByOne implements Consumer<Point> {
    public void accept(Point p) {
        p.translate(1, 1);
    }
}
...
pointList.forEach(new TranslateByOne());
```

You may think that this code is still pretty clumsy. But notice that the clumsiness is now concentrated in the representation of each command by an instance of a class. In many cases, this is overkill. In the present case, for example, all that `forEach` really needs is the *behavior* of the single method `accept` of the object that has been supplied to it. State and all the other apparatus that make up the object are included only because method arguments in Java, if not primitives, have to be object references. But we have always needed to specify this apparatus—until now.

1.1.3 Lambda Expressions

The code that concluded the previous section is not idiomatic Java for the command pattern. When, as in this case, a class is both small and unlikely to be reused, a more common usage is to define an *anonymous inner class*:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

Experienced Java developers are so accustomed to seeing code like this that we have often forgotten how we felt when we first encountered it. Common first reactions to the syntax for anonymous inner classes used in this way are that it is ugly, verbose, and difficult to understand quickly, even though all it is really doing is to say "do this for each element." You don't have to agree completely with these judgements to accept that any attempt to persuade developers to rely on this idiom for every collection operation is unlikely to be very successful. And this, at last, is our cue for the introduction of lambda expressions.[2]

To reduce the verbosity of this call, we should try to identify those places where we are supplying information that the compiler could instead infer from the context. One such piece of information is the name of the interface being implemented by the anonymous inner class. It's enough for the compiler to know that the declared type of the parameter to `forEach` is `Consumer<T>`; that is sufficient information to allow the supplied argument to be checked for type compatibility. Let's de-emphasize the code that the compiler can infer:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

Second, what about the name of the method being overridden—in this case, `accept`? There's no way that the compiler can infer that in general. But in the case of `Consumer` there is no need to infer the name, because the interface has only a single method. This "one method interface" pattern is so useful for defining callbacks that it has an official status: any object to be used in the abbreviated form that we are developing must implement an interface like this, exposing a single abstract method (this is called

[2] People are often curious about the origin of the name. The idea of lambda expressions comes from a model of computation developed in the 1930s by the American mathematician Alonzo Church, in which the Greek letter λ (lambda) represents functional abstraction. But why that particular letter? Church seems to have liked to tease: asked about his choice, his usual explanation involved accidents of typesetting, but in later years he had an alternative answer: "Eeny, meeny, miny, moe."

a *functional interface*, or sometimes a *SAM* interface). That gives the compiler a way to choose the correct method without ambiguity. Again let's de-emphasize the code that can be inferred in this way:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

Finally, the instantiated type of `Consumer` can often be inferred from the context, in this case from the fact that when the `forEach` method calls `accept`, it supplies it with an element of `pointList`, previously declared as a `List<Point>`. That identifies the type parameter to `Consumer` as `Point`, allowing us to omit the explicit type declaration of the argument to `accept`.

This is what's left when we de-emphasize this last component of the `forEach` call:

```
pointList.forEach(new Consumer<Point>() {
    public void accept(Point p) {
        p.translate(1, 1);
    }
});
```

The argument to `forEach` represents an object, implementing the interface (`Consumer`) required by `forEach`, such that when `accept` (the only abstract method on that interface) is called for a `pointList` element `p`, the effect will be to call `p.translate(1, 1)`.

Some extra syntax ("`->`") is required to separate the *parameter list* from the *expression body*. With that addition, we finally get the simple form for a lambda expression. Here it is, being used in internal iteration:

```
pointList.forEach(p -> p.translate(1, 1));
```

If you are unused to reading lambda expressions, you may find it helpful for the moment to continue to think of them as an abbreviation for a method declaration, mentally mapping the parameter list of the lambda to that of the imaginary method, and its body (often preceded by an added `return`) to the method body. In the next chapter, we will see that it is going to be necessary to vary the simple syntax of the preceding example for lambda expressions with multiple parameters and with more elaborate bodies and in cases where the compiler cannot infer parameter types. But if you have followed the reasoning that brought us to this point, you should have a basic understanding of the motivation for the introduction of lambda expressions and of the form that they have taken.

This section has covered a lot of ground. To summarize: we began by considering the adaptations that our programming model needs to make in order to accommodate the requirements of changing hardware architectures; this brought us to a review of processing of collection elements, which in turn made us aware of the need to have a concise way of defining behavior for collections to execute; finally, paring away the excess text from anonymous inner class definitions brought us to a simple syntax for lambda expressions.

In the remaining sections of this chapter, we will look at some of the new idioms that lambda expressions make possible. We will see that bulk processing of collection elements can be written in a much more expressive style, that these changes in idiom make it much easier for library writers to incorporate parallel algorithms to take advantage of new hardware architectures, and finally that emphasizing functional behavior can improve the design of APIs. It's an impressive list of achievements for such an innocuous-looking change!

1.2 From Collections to Streams

Let's extend the example of the previous section a little. In real-life programs, it's common to process collections in a number of stages: a collection is iteratively processed to produce a new collection, which in turn is iteratively processed, and so on. Here is an example—rather artificial, in the interests of simplicity—which starts with a collection of `Integer` instances, then applies an arbitrary transformation to produce a collection of `Point` instances, and finally finds the maximum among the distances of each `Point` from the origin.

```
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 1));
}
double maxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
```

Although it could certainly be improved, this is idiomatic Java—most developers have seen many examples of code in this pattern—but if we look at it with fresh eyes, some unpleasant features stand out at once. Firstly, it is very verbose, taking nine lines of code to carry out only three operations. Secondly, the collection `pointList`, required only as intermediate storage, is an overhead on the operation of the program; if the intermediate storage is very large, creating it would at best add to garbage collection overheads, and at worst would exhaust available heap space. Thirdly, there is

an implicit assumption, difficult to spot, that the minimum value of an empty list is `Double.MIN_VALUE`. But the worst aspect of all is the gap between the developer's intentions and the way that they are expressed in code. To understand this program, you have to work out what it's doing, then guess the developer's intention (or, if you're very fortunate, read the comments), and only then check its correctness by matching the operation of the program to the informal specification you deduced.[3] All this work is slow and error-prone—indeed, the very purpose of a high-level language is supposed to be to minimize it by supporting code that is as close as possible to the developer's mental model. So how can the gap be closed?

Let's restate the problem specification:

> "Apply a transformation to each one of a collection of `Integer` instances to produce a `Point`, then find the greatest distance of any of these `Points` from the origin."

If we de-emphasize the parts of the preceding code that do not correspond to the elements of this informal specification, we see what a poor match there is between code and problem specification. Omitting the first line, in which the list `intList` is initially created, we get:

```
List<Point> pointList = new ArrayList<>();
for (Integer i : intList) {
    pointList.add(new Point(i % 3, i / 3));
}
double maxDistance = Double.MIN_VALUE;
for (Point p : pointList) {
    maxDistance = Math.max(p.distance(0, 0), maxDistance);
}
```

This suggests a new, data-oriented way of looking at the program, one that will look familiar if you are used to Unix pipes and filters: we can follow the progress of a single value from the source collection, viewing it as being transformed first from an `Integer` to a `Point` and second from a `Point` to a `double`. Both of these transformations can take place in isolation, without any reference to the other values being processed—exactly the requirement for parallelization. Only with the third step, finding the greatest distance, is it necessary for the values to interact (and even then, there are techniques for efficiently computing this in parallel).

This data-oriented view can be represented diagrammatically, as in Figure 1-1. In this figure it is clear that the rectangular boxes represent operations. The connecting

[3] The situation is better than it used to be. Some of us are old enough to remember how much of this kind of work was involved in writing big programs in assembler (*really* low-level languages, not far removed from machine code). Programming languages have become much more expressive since then, but there is still plenty of room for progress.

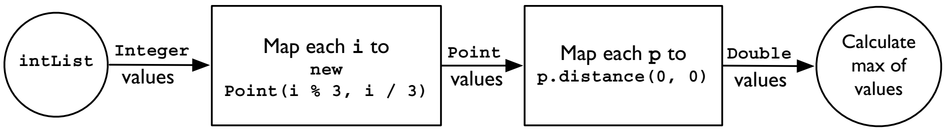


FIGURE 1-1. *Composing streams into a data pipeline*

lines represent something new, a way of delivering a sequence of values to an operation. This is different from any kind of collection, because at a given moment the values to be delivered by a connector may not all have been generated yet. These value sequences are called *streams*. Streams differ from collections in that they provide an (optionally) ordered sequence of values without providing any storage for those values; they are "data in motion," a means for expressing bulk data operations. If the idea of streams is new to you, it may help to imagine a kind of iterator on which the only operation is like `next`, except that besides returning the next value, it can signal that there are no more values to get. In the Java 8 collections API, streams are represented by interfaces—`Stream` for reference values, and `IntStream`, `LongStream`, and `DoubleStream` for streams of primitive values—in the package `java.util.stream`.

In this view, the operations represented by the boxes in Figure 1-1 are operations on streams. The boxes in this figure represent two applications of an operation called `map`; it transforms each stream element using a systematic rule. Looking at `map` alone, we might think that we were dealing with operations on individual stream elements. But we will soon meet other stream operations that can reorder, drop, or even insert values; each of these operations can be described as taking a stream and transforming it in some way. Each rectangular box represents an *intermediate operation*, one that is not only defined on a stream, but that also returns a stream as its output. For example, assuming for a moment that a stream `intStream` forms the input to the first operation, the transformations made by the intermediate operations of Figure 1-1 can be represented in code as:

```
Stream<Point> points = intStream.map(i -> new Point(i % 3, i / 3));
DoubleStream distances = points.mapToDouble(p -> p.distance(0, 0));
```

The circle at the end of the pipeline represents the *terminal operation* `max`. Terminal operations consume a stream, optionally returning a single value, or—if the stream is empty—nothing, represented by an empty `Optional` or one of its specializations (see p. 65):

```
OptionalDouble maxDistance = distances.max();
```

Pipelines like that in Figure 1-1 have a beginning, a middle, and an end. We have seen the operations that defined the middle and the end; what about the beginning? The values flowing into streams can be supplied by a variety of sources—collections, arrays, or generating functions. In practice, a common use case will be feeding the contents of a collection into a stream, as here. Java 8 collections expose a new method `stream()` for this purpose, so the start of the pipeline can be represented as:

```
Stream<Integer> intStream = intList.stream();
```

And the complete code with which this section began has become:

```
OptionalDouble maxDistance =
    intList.stream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p -> p.distance(0, 0))
        .max();
```

This style, often called *fluent* because "the code flows," is unfamiliar in the context of collection processing and may seem initially difficult to read in this context. However, compared to the successive iterations in the code that introduced this section, it provides a nice balance of conciseness with a close correspondence to the problem statement: "map each integer in the source `intList` to a corresponding `Point`, map each `Point` in the resulting list to its distance from the origin, then find the maximum of the resulting values." The structure of the code highlights the key operations, rather than obscuring them as in the original.

As a bonus, the performance overhead of creating and managing intermediate collections has disappeared as well: executed sequentially, the stream code is more than twice as fast as the loop version. Executed in parallel, virtually perfect speedup is achieved on large data sets (for more details of the experiment, see p. 148).

1.3 From Sequential to Parallel

This chapter began with the assertion that Java now needs to support parallel processing of collections, and that lambdas are an essential step in providing this support. We've come most of the way by seeing how lambdas make it easy for client code developers to make use of internal iteration. The last step is to see how internal iteration of the collection classes actually implements parallelism. It's useful to know the principles of how this will work, although you don't need them for everyday use—the complexity of the implementations is well hidden from developers of client code.

Independent execution on multiple cores is accomplished by assigning a different thread to each core, each thread executing a subtask of the work to be done—in this case, a subset of the collection elements to be processed. For example, given a four-

core processor and a list of *N* elements, a program might define a `solve` algorithm to break the task down for parallel execution in the following way:

```
if the task list contains more than N/4 elements {
    leftTask = task.getLeftHalf()
    rightTask = task.getRightHalf()
    doInParallel {
        leftResult = leftTask.solve()
        rightResult = rightTask.solve()
    }
    result = combine(leftResult, rightResult)
} else {
    result = task.solveSequentially()
}
```

The preceding pseudocode is a highly simplified description of parallel processing using a list specialization of the pattern of *recursive decomposition*—recursively splitting large tasks into smaller ones, to be executed in parallel, until the subtasks are "small enough" to be executed in serial. Implementing recursive decomposition requires knowing how to split tasks in this way, how to execute sufficiently small ones without further splitting, and how to then combine the results of these smaller executions. The technique for splitting depends on the source of the data; in this case, splitting a list has an obvious implementation. Combining the results of subtasks is often achieved by applying the pipeline terminal operation to them; for the example of this chapter, it involves taking the maximum of two subtask results.

The Java fork/join framework uses this pattern, allocating threads from its pool to new subtasks rather than creating new ones. Clearly, reimplementing this pattern is far more coding work than can realistically be expected of developers every time a collection is to be processed. This is library work—or it certainly should be!

In this case, the library class is the collection; from Java 8 onward, the collections library classes will be able to use the fork/join framework in this way, so that client developers can put parallelization, essentially a performance issue, to the back of their minds and get on with solving business problems. For our current example, the only change necessary to the client code is emphasized here:

```
OptionalDouble maxDistance =
    intList.parallelStream()
        .map(i -> new Point(i % 3, i / 3))
        .mapToDouble(p -> p.distance(0, 0))
        .max();
```

This illustrates what is meant by the slogan for the introduction of parallelism in Java 8: *explicit but unobtrusive*. Parallel execution is achieved by breaking the initial list of `Integer` values down recursively, as in the pseudocode for `solve`, until the

sublists are small enough, then executing the entire pipeline serially, and finally combining the results with `max`. The process for deciding what is "small enough" takes into account the number of cores available and, sometimes, characteristics of the list. Figure 1-2 shows the decomposition of a list for processing by four cores: in this case, "small enough" is just the list size divided by four. (A connected problem is deciding when a list is "big enough" to make it worthwhile to incur the overhead of executing in parallel. Chapter 6 will explore this problem in detail.)

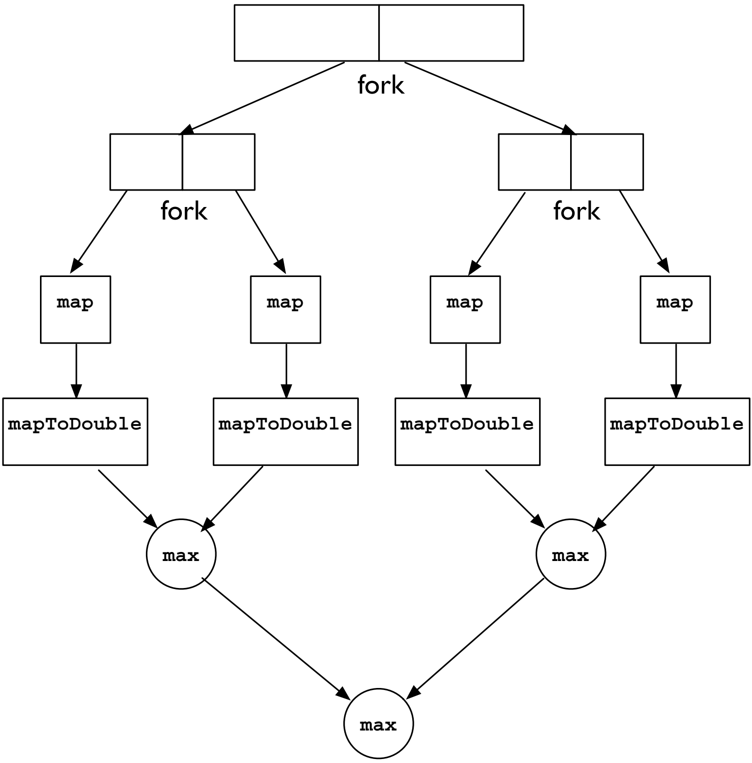


FIGURE 1-2. *Recursive decomposition of a list processing task*

Unobtrusive parallelism is an example of one of the key themes of Java 8; the API changes that it enables give much greater freedom to library developers. One important way in which they can use it is to explore the many opportunities for performance improvement that are provided by modern—and future—machine architectures.

1.4 Composing Behaviors

Earlier in this chapter we saw how functionally similar lambda expressions are to anonymous inner classes. But writing them so differently leads to different ways of thinking about them. Lambda expressions *look* like functions, so it's natural to ask whether we can make them *behave* like functions. That change of perspective will encourage us to think about working with behaviors rather than objects, and that in turn will lead in the direction of some very different programming idioms and library APIs.

For example, a core operation on functions is *composition*: combining together two functions to make a third, whose effect is the same as applying its two components in succession. Composition is not an idea that arises at all naturally in connection with anonymous inner classes, but in a generalized form it corresponds very well to the construction of traditional object-oriented programs. And just as object-oriented programs are broken down by decomposition, the reverse of composition will work for functions too.

Suppose, for example, that we want to sort a list of `Point` instances in order of their `x` coordinate. The standard Java idiom for a "custom" sort[4] is to create a `Comparator`:

```
Comparator<Point> byX = new Comparator<Point>(){
    public int compare(Point p1, Point p2) {
        return Double.compare(p1.getX(), p2.getX());
    }
};
```

Substituting a lambda expression for the anonymous inner class declaration, as described in the previous section, improves the readability of the code:

```
Comparator<Point> byX =
    (p1, p2) -> Double.compare(p1.getX(), p2.getX());            ❶
```

But that doesn't help with another very significant problem: `Comparator` is monolithic. If we wanted to define a `Comparator` that compared on `y` instead of `x` coordinates, we would have to copy the entire declaration, substituting `getY` for `getX` everywhere. Good programming practice should lead us to look for a better solution, and a moment's reflection shows that `Comparator` is actually carrying out two functions—extracting sort keys from its arguments and then comparing those keys. We should be able to improve the code of ❶ by building a `Comparator` function parameterized on these two components. We'll now evolve the code to do that. The intermediate stages may seem awkward and verbose, but persist: the conclusion will be worthwhile.

[4] Two ways of comparing and sorting objects are standard in the Java platform: a class can have a *natural order*; in this case, it implements the interface `Comparable` and so exposes a `compareTo` method that an object can use to compare itself with another. Or a `Comparator` can be created for the purpose, as in this case.

To start, let's turn the two concrete component functions that we have into lambda form. We know the type of the functional interface for the key extractor function—`Comparator`—but we also need the type of the functional interface corresponding to the function `p -> p.getX()`. Looking in the package devoted to the declaration of functional interfaces, `java.util.function`, we find the interface `Function`:

```
public interface Function<T,R> {
    public R apply(T t);
}
```

So we can now write the lambda expressions for both key extraction and key comparison:

```
Function<Point,Double> keyExtractor = p -> p.getX();
Comparator<Double> keyComparer = (d1, d2) -> Double.compare(d1, d2);
```

And our version of `Comparator<Point>` can be reassembled from these two smaller functions:

```
Comparator<Point> compareByX = (p1, p2) -> keyComparer.compare(
    keyExtractor.apply(p1), keyExtractor.apply(p2));                ❷
```

This matches the form of ❶ but represents an important improvement (one that would be much more significant in a larger example): you could plug in any `keyComparer` or `keyExtractor` that had previously been defined. After all, that was the whole purpose of seeking to parameterize the larger function on its smaller components.

But although recasting the `Comparator` in this way has improved its structure, we have lost the conciseness of ❶. We can recover that in the special but very common case where `keyComparer` expresses the natural ordering on the extracted keys. Then ❷ can be rewritten as:

```
Comparator<Point> compareByX = (p1, p2) ->
    keyExtractor.apply(p1).compareTo(keyExtractor.apply(p2));       ❸
```

And, noticing the importance of this special case, the platform library designers added a static method `comparing` to the interface `Comparator`; given a key extractor, it creates the corresponding `Comparator`[5] using natural ordering on the keys. Here is its method declaration, in which generic type parameters have been simplified for this explanation:

```
public static <T,U extends Comparable<U>>
                  Comparator<T> comparing(Function<T,U> keyExtractor) {
```

[5] Other overloads of `comparing` can create `Comparators` for primitive types in the same way, but since natural ordering can't be used, they instead use the `compare` methods exposed by the wrapper classes.

```
    return (c1, c2) ->
        keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Using that method allows us to write the following (assuming a static import declaration of `Comparators.comparing`) instead of ❸:

```
Comparator<Point> compareByX = comparing(p -> p.getX());                    ❹
```

Compared to ❶, ❹ is a big improvement: more concise and more immediately understandable because it isolates and lays emphasis on the important element, the key extractor, in a way that is possible only because `comparing` accepts a simple behavior and uses it to build a more complex one from it.

To see the improvement in action, imagine that our problem changes slightly so that instead of finding the single point that is furthest from the origin, we decide to print all the points in ascending order of their distance. It is straightforward to capture the necessary ordering:

```
Comparator<Point> byDistance = comparing(p -> p.distance(0, 0));
```

And to implement the changed problem specification, the stream pipeline needs only a small corresponding change:

```
intList.stream()
    .map(i -> new Point(i % 3, i / 3))
    .sorted(comparing(p -> p.distance(0, 0)))
    .forEach(p -> System.out.printf("(%f, %f)", p.getX(), p.getY()));
```

The change needed to accommodate the new problem statement illustrates some of the advantages that lambdas will bring. Changing the `Comparator` was straightforward because it is being created by composition and we needed to specify only the single component being changed. The use of the new comparator fits smoothly with the existing stream operations, and the new code is again close to the problem statement, with a clear correspondence between the changed part of the problem and the changed part of the code.

1.5 Conclusion

It should be clear by now why the introduction of lambda expressions has been so keenly awaited. In the earlier sections of this chapter we saw the possibilities they will create for performance improvement, by allowing library developers to enable automatic parallelization. Although this improvement will not be universally available—one purpose of this book is to help you to understand exactly when your application will benefit from "going parallel"—it represents a major step in the right direction,

of starting to make the improved performance of modern hardware accessible to the application programmer.

In the last section, we saw how lambdas will encourage the writing of better APIs. The signature of `Comparator.comparing` is a sign of things to come: as client programmers become comfortable with supplying behaviors like the key extraction function that `comparing` accepts, fine-grained library methods like `comparing` will become the norm and, with them, corresponding improvements in the style and ease of client coding.