This book outlines everything the beginning programmer needs to know to program with the Java programming language and the 1.4 Java Developer Kit. With the release of JDK 1.4, programmers can look forward to achieving better performance than ever.

Table of Contents

Beginning Java 2, SDK 1.4 Edition

Chapter 1 - Introducing Java

Chapter 2 - Programs, Data, Variables, and Calculation

Chapter 3 - Loops and Logic

Chapter 4 - Arrays and Strings

Chapter 5 - Defining Classes

Chapter 6 - Extending Classes and Inheritance

Chapter 7 - Exceptions

Chapter 8 - Understanding Streams

Chapter 9 - Accessing Files and Directories

Chapter 10 - Writing Files
<table>
<thead>
<tr>
<th>Ivor Horton</th>
<th>Steve Baker</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Robert Chang</td>
</tr>
<tr>
<td><strong>Technical Editors</strong></td>
<td><strong>Phil Powers De George</strong></td>
</tr>
<tr>
<td>Mankee Cheng</td>
<td>Justin Foley</td>
</tr>
<tr>
<td>Shivanand Nadkarni</td>
<td>David Hudson</td>
</tr>
<tr>
<td>Girish Sharangpani</td>
<td>Jim MacIntosh</td>
</tr>
<tr>
<td>Roberts Shaw</td>
<td>Craig McQueen</td>
</tr>
<tr>
<td>Andrew Tracey</td>
<td>Karen Ritchie</td>
</tr>
<tr>
<td></td>
<td>Gavin Smyth</td>
</tr>
<tr>
<td><strong>Commissioning Editors</strong></td>
<td><strong>John Timney</strong></td>
</tr>
<tr>
<td>Louay Fatoohi</td>
<td>David Whitney</td>
</tr>
<tr>
<td>Chanoch Wiggers</td>
<td></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td><strong>Production Coordinator</strong></td>
</tr>
<tr>
<td>John Collins</td>
<td>Natalie O'Donnell</td>
</tr>
<tr>
<td>Andrew Criddle</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Illustrations</strong></td>
</tr>
</tbody>
</table>
A Note from the Author

In all my *Beginning* books, my objective is to minimize what, in my judgment, are the three main hurdles the aspiring programmer must face: getting to grips with the jargon that pervades every programming language and environment, understanding the *use* of the language elements (as opposed to what they are), and appreciating how the language is applied in a practical context.

Jargon is an invaluable and virtually indispensable means of communication for the competent amateur as well as the expert professional, so it can't be avoided. My approach is to ensure that the beginner understands what the jargon means and gets comfortable with using it in context. In that way, they can use the documentation that comes along with most programming products more effectively, and can also feel competent to read and learn from the literature that surrounds most programming languages.

Comprehending the syntax and effects of the language elements are obviously essential to learning a language, but I believe illustrating *how* the language features work and *how* they are used are equally important. Rather than just use code fragments, I always try to provide the reader with practical working examples that show the relationship of each language feature to specific problems. These can then be a basis for experimentation, to see at first hand the effects of changing the code in various ways.

The practical context needs to go beyond the mechanics of applying individual language elements. To help the beginner gain the competence and confidence to develop their own applications, I aim to provide them with an insight into how things work in combination and on a larger scale than a simple example with a few lines of code. That's why I like to have at least one working example that builds over several chapters. In that way it's possible to show something of the approach to managing code as well as how language features can be applied together.
Finally, I know the prospect of working through a book of doorstop proportions can be quite daunting. For that reason it's important for the beginner to realize three things that are true for most programming languages. First, there is a lot to it, but this means there will be a greater sense of satisfaction when you've succeeded. Second, it's great fun, so you really will enjoy it. Third, it's a lot easier than you think, so you positively will make it.

Ivor Horton
Chapter 1: Introducing Java

Overview

This chapter will give you an appreciation of what the Java language is all about. Understanding the details of what we'll discuss in this chapter is not important at this stage; you will see all of them again in greater depth in later chapters of the book. The intent of this chapter is to introduce you to the general ideas that underpin what we'll be covering through the rest of the book, as well as the major contexts in which Java programs can be used and the kind of program that is applicable in each context.

In this chapter you will learn:

• The basic characteristics of the Java language.

• How Java programs work on your computer.

• Why Java programs are portable between different computers.

• The basic ideas behind object-oriented programming.

• How a simple Java program looks and how you can run it using the Java Development Kit.

• What HTML is and how it is used to include a Java program in a web page.
What is Java All About?

Java is an innovative programming language that has become the language of choice for programs that need to run-on a variety of different computer systems. First of all Java enables you to write small programs called applets. These are programs that you can embed in Internet web pages to provide some intelligence. Being able to embed executable code in a web page introduces a vast range of exciting possibilities. Instead of being a passive presentation of text and graphics, a web page can be interactive in any way that you want. You can include animations, games, interactive transaction processing the possibilities are almost unlimited.

Of course, embedding program code in a web page creates special security requirements. As an Internet user accessing a page with embedded Java code, you need to be confident that it will not do anything that might interfere with the operation of your computer, or damage the data you have on your system. This implies that execution of the embedded code must be controlled in such a way that it will prevent accidental damage to your computer environment, as well as ensure that any Java code that was created with malicious intent is effectively inhibited. Java implicitly incorporates measures to minimize the possibility of such occurrences arising with a Java applet.

Java also allows you to write large-scale application programs that you can run unchanged on any computer with an operating system environment in which the language is supported. This applies to the majority of computers in use today. You can even write programs that will work both as ordinary applications and as applets.

Java has matured immensely in recent years, particularly with the introduction of Java 2. The breadth of function provided by the standard core Java has grown incredibly. Java provides you with comprehensive facilities for building application with an interactive GUI, extensive image processing and graphics programming facilities, as well as support for accessing relational databases and communicating with remote computers over a network. Release 1.4 of Java added a very important additional capability, the ability to read and write XML. Just about any kind of application can now be programmed effectively in Java, with the implicit plus of complete portability.

Features of the Java Language

The most important characteristic of Java is that it was designed from the outset to be machine independent. Java programs can run unchanged on any operating system that supports Java. Of course there is still the slim possibility of the odd glitch as you are ultimately dependent on the implementation of Java on any particular machine, but Java programs are intrinsically more portable than programs written in other languages. An application written in Java will only require a single set of sourcecode, regardless of the number of different computer platforms on which it is run. In any other programming language, the application will frequently require the sourcecode to be tailored to accommodate different computer environments, particularly if there is an extensive graphical user interface involved. Java offers substantial savings in time and resources in developing, supporting, and maintaining major applications on several different hardware platforms and operating systems.

Possibly the next most important characteristic of Java is that it is object oriented. The object-oriented approach to programming is also an implicit feature of all Java programs, so we will be looking at what this implies later in this chapter. Object-oriented programs are easier to understand, and less time-consuming to maintain and extend than programs that have been written without the benefit of using objects.
Not only is Java object oriented, but it also manages to avoid many of the difficulties and complications that are inherent in some other object-oriented languages, making it easy to learn and very straightforward to use. It lacks the traps and 'gotchas' that arise in some other programming languages. This makes the learning cycle shorter and you need less real-world coding experience to gain competence and confidence. It also makes Java code easier to test.

Java has a built-in ability to support national character sets. You can write Java programs as easily for Greece or Japan, as you can for English speaking countries always assuming you are familiar with the national languages involved, of course. You can even build programs from the outset to support several different national languages with automatic adaptation to the environment in which the code executes.
Learning Java

Java is not difficult, but there is a great deal to it. The language itself is fairly compact, but very powerful. To be able to program effectively in Java, however, you also need to understand the libraries that go with the language, and these are very extensive. In this book, the sequence in which you learn how the language works, and how you apply it, has been carefully structured so that you can gain expertise and confidence with programming in Java through a relatively easy and painless process. As far as possible, each chapter avoids the use of things you haven't learned about already. A consequence, though, is that you won't be writing Java applications with a graphical user interface right away. While it may be an appealing idea, this would be a bit like learning to swim by jumping in the pool at the deep end. Generally speaking, there is good evidence that by starting in the shallow end of the pool and learning how to float before you try to swim, the chance of drowning is minimized, and there is a high expectation that you will end up a competent swimmer.

Java Programs

As we have already noted, there are two kinds of programs you can write in Java. Programs that are to be embedded in a web page are called Java applets, and normal standalone programs are called Java applications. You can further subdivide Java applications into console applications, which only support character output to your computer screen (to the command line on a PC under Windows, for example), and windowed Java applications that can create and manage multiple windows. The latter use the typical graphical user interface (GUI) mechanisms of window-based programs—menus, toolbars, dialogs and so on.

While we are learning the Java language basics, we will be using console applications as examples to illustrate how things work. These are application that use simple command line input and output. With this approach we can concentrate on understanding the specifics of the language, without worrying about any of the complexity involved in creating and managing windows. Once we are comfortable with using all the features of the Java language, we'll move on to windowed applications and applet examples.

Learning Java the Road Ahead

Before starting out, it is always helpful to have an idea of where you are heading and what route you should take, so let's take a look at a brief road map of where you will be going with Java. There are five broad stages you will progress through in learning Java using this book:

1. The first stage is this chapter. It sets out some fundamental ideas about the structure of Java programs and how they work. This includes such things as what object-oriented programming is all about, and how an executable program is created from a Java source file. Getting these concepts straight at the outset will make learning to write Java programs that much easier for you.
Next you will learn how statements are put together, what facilities you have for storing basic data in a program, how you perform calculations and how you make decisions based on the results of them. These are the nuts and bolts you need for the next stages.

3.

In the third stage you will learn about **classes**—how you define them and how you can use them. This is where you learn the object-oriented characteristics of the language. By the time you are through this stage you will have learned all the basics of how the Java language works so you will be ready to progress further into how you can use it.

4.

In the fourth stage, you will learn how you can segment the activities that your programs carry out into separate tasks that can execute concurrently. This is particularly important for when you want to include several applets in a web page, and you don't want one applet to have to wait for another to finish executing before it can start. You may want a fancy animation to continue running while you play a game, for example, with both programs sitting in the same web page.

5.

In the fifth stage you will learn in detail how you implement an application or an applet with a graphical user interface, and how you handle interactions with the user in this context. This amounts to applying the capabilities provided by the Java class libraries. When you finish this stage you will be equipped to write your own fully-fledged applications and applets in Java. At the end of the book, you should be a knowledgeable Java programmer. The rest is down to experience.

Throughout this book we will be using complete examples to explore how Java works. You should create and run all of the examples, even the simplest, preferably by typing them in yourself. Don't be afraid to experiment with them. If there is anything you are not quite clear on, try changing an example around to see what happens, or better still write an example of your own. If you are uncertain how some aspect of Java that you have already covered works, don't look it up right away—try it out. Making mistakes is a great way to learn.
The Java Environment

You can run Java programs on a wide variety of computers using a range of operating systems. Your Java programs will run just as well on a PC running Windows 95/98/NT/2000/XP as it will on Linux or a Sun Solaris workstation. This is possible because a Java program does not execute directly on your computer. It runs on a standardized hypothetical computer that is called the **Java virtual machine** or **JVM**, which is emulated inside your computer by a program.

A **Java compiler** converts the Java sourcecode that you write into a binary program consisting of **byte codes**. Byte codes are machine instructions for the Java virtual machine. When you execute a Java program, a program called the **Java interpreter** inspects and deciphers the byte codes for it, checks it out to ensure that it has not been tampered with and is safe to execute, and then executes the actions that the byte codes specify within the Java virtual machine. A Java interpreter can run standalone, or it can be part of a web browser such as Netscape Navigator or Microsoft Internet Explorer where it can be invoked automatically to run applets in a web page.

Because your Java program consists of byte codes rather than native machine instructions, it is completely insulated from the particular hardware on which it is run. Any computer that has the Java environment implemented will handle your program as well as any other, and because the Java interpreter sits between your program and the physical machine, it can prevent unauthorized actions in the program from being executed.

In the past there has been a penalty for all this flexibility and protection in the speed of execution of your Java programs. An interpreted Java program would typically run at only one tenth of the speed of an equivalent program using native machine instructions. With present Java machine implementations, much of the performance penalty has been eliminated, and in programs that are not computation intensive which is usually the case with the sort of program you would want to include in a web page, for example you really wouldn't notice this anyway. With the JVM that is supplied with the current Java 2 System Development Kit (SDK) available from the Sun web site, there are very few circumstances where you will notice any appreciable degradation in performance compared to a program compiled to native machine code.

Java Program Development
There are a number of excellent professional Java program development environments available, including products from Sun, Borland and Symantec. These all provide very friendly environments for creating and editing your source code, and compiling and debugging your programs. These are powerful tools for the experienced programmer, but for learning Java using this book, I recommend that you resist the temptation to use any of these, especially if you are relatively new to programming. Instead, stick to using the Java 2 SDK from Sun together with a suitable simple editor for creating your source code. The professional development systems tend to hide a lot of things you need to understand, and also introduce complexity that you really are better off without while you are learning. These products are intended primarily for knowledgeable and experienced programmers, so start with one when you get to the end of the book.

You can download the SDK from Sun for a variety of hardware platforms and operating systems, either directly from the Sun Java web site at http://java.sun.com (for Windows, Solaris, and Linux operating systems), or from sites that you can link to from there. The SDK we are going to use is available from http://java.sun.com/j2se/1.4. For instance, a version of the SDK for Mac OS is available from http://devworld.apple.com/java/.

There is one aspect of terminology that sometimes causes confusion—the SDK used to be known as the JDK—the Java Development Kit. If you see JDK this generally means the same as SDK. When you install the Java 2 SDK, you will see the old terminology survives in the name of the root directory where the SDK is installed, currently /jdk1.4.

I would urge you to install the SDK even if you do use one or other of the interactive development environments that are available. The SDK provides an excellent reference environment that you can use to check out problems that may arise. Not only that, your programs will only consist of the code that you write plus the classes from the Java libraries that you use. Virtually all commercial Java development systems provide pre-built facilities of their own to speed development. While this is very helpful for production program development, it really does get in the way when you are trying to learn Java.

A further consideration is that the version of Java supported by a commercial Java product is not always the most recent. This means that some features of the latest version of Java just won't work. If you really do prefer to work with a commercial Java development system for whatever reason, and you have problems with running a particular example from the book, try it out with the SDK. The chances are it will work OK.

To make use of the SDK you will need a plain text editor. Any editor will do as long as it does not introduce formatting codes into the contents of a file. There are quite a number of shareware and freeware editors around that are suitable, some of which are specific to Java, and you should have no trouble locating one. I find the JCreator editor is particularly good. There's a free version and a fee version with more functionality but the free version is perfectly adequate for learning. You can download a free copy from http://www.jcreator.com. A good place to start looking if you want to explore what is available is the http://www.download.com web site.

Installing the SDK

You can obtain detailed instructions on how to install the SDK for your particular operating system from the Sun web site, so I won't go into all the variations for different systems here. However, there are a few things to watch out for that may not leap out from the pages of the installation documentation.

First of all, the SDK and the documentation are separate and you install them separately. The SDK for Windows is distributed as a .exe file that you just execute to start installation. The documentation for the SDK consists of a large number of HTML files structured in a hierarchy that are distributed in a ZIP archive. You will find it easier to install the SDK first, followed by the documentation. If you install the SDK to drive C: under Windows, the directory
structure shown in the diagram will be created.

The jdk1.4 directory in the diagram is sometimes referred to as the root directory for Java. In some contexts it is also referred to as the Java home directory. If you want the documentation installed in the hierarchy shown above, then you should now extract the documentation from the archive to the jdk1.4 directory. This corresponds to C:\jdk1.4 if you installed the SDK to your C: drive. This will create a new subdirectory, docs, to the jdk1.4 root directory, and install the documentation files in that. To look at the documentation you just open the index.html file that is in the docs subdirectory.

You don't need to worry about the contents of most of these directories, at least not when you get started, but you should add the path for the jdk1.4\bin directory to the paths defined in your PATH environment variable. That way you will be able to run the compiler and the interpreter from anywhere without having to specify supplying the path to it. If you installed the SDK to C:, then you need to add the path C:\jdk1.4\bin. A word of warning if you have previously installed a commercial Java development product, check that it has not modified your PATH environment variable to include the path to its own Java executables.

If it has, when you try to run the Java compiler or interpreter, you are likely to get the versions supplied with the commercial product rather that those that came with the SDK. One way to fix this is to remove the path or paths that cause the problem. If you don't want to remove the paths that were inserted for the commercial product, you will have to use the full path specification when you want to run the compiler or interpreter from the SDK. The jre directory contains the Java Runtime facilities that are used when you execute a Java program. The classes in the Java libraries are stored in the jre\lib directory. They don't appear individually though. They are all packaged up in the archive, rt.jar. Leave this alone. The Java Runtime takes care of retrieving what it needs from the archive when your program executes.

The CLASSPATH environment variable is a frequent source of problems and confusion to newcomers to Java. The current SDK does NOT require CLASSPATH to be defined, and if it has been defined by some other Java version or system, it is likely to cause problems. Commercial Java development systems and versions of the Java Development Kit prior to 1.2 may well define the CLASSPATH environment variable, so check to see whether CLASSPATH has been defined on your system. If it has and you no longer have whatever defined it installed, you should delete it. If you have to keep the CLASSPATH environment variable maybe because you want to keep the system that defined it or you share the machine with someone who needs it you will have to use a command line option to define CLASSPATH temporarily whenever you compile or execute your Java code. We will see how to do this a little later in this chapter.

Extracting the Sourcecode for the Class Libraries
The sourcecode for the class libraries is included in the archive src.zip that you will find in the jdk1.4 root directory. Browsing this source can be very educational, and it can also be helpful when you are more experienced with Java in giving a better understanding of how things work or when they don’t, why they don’t. You can extract the source files from the archive using the Winzip utility or any other utility that will unpack .zip archives but be warned there’s a lot of it and it takes a while!

Extracting the contents of src.zip to the root directory \jdk1.4 will create a new subdirectory, src, and install the sourcecode in subdirectories to this. To look at the sourcecode, just open the .java file that you are interested in, using any plain text editor.

### Compiling a Java Program

Java sourcecode is always stored in files with the extension .java. Once you have created the sourcecode for a program and saved it in a .java file, you need to process the source using a Java compiler. Using the compiler that comes with the JDK, you would make the directory that contains your Java source file the current directory, and then enter the following command:

```
javac -source 1.4 MyProgram.java
```

Here, javac is the name of the Java compiler, and MyProgram.java is the name of the program source file. This command assumes that the current directory contains your source file. If it doesn’t the compiler won’t be able to find your source file. The -source command line option with the value 1.4 here tells the compiler that you want the code compiled with the SDK 1.4 language facilities. This causes the compiler to support a facility called assertions, and we will see what these are later on. If you leave this option out, the compiler will compile the code with SDK 1.3 capabilities so if the code uses assertions, these will be flagged as errors.

If you need to override an existing definition of the CLASSPATH environment variable perhaps because it has been set by a Java development system you have installed, the command would be:

```
javac -source 1.4 -classpath . MyProgram.java
```

The value of CLASSPATH follows the -classpath specification and is just a period. This defines just the path to the current directory, whatever that happens to be. This means that the compiler will look for your source file or files in the current directory. If you forget to include the period, the compiler will not be able to find your source files in the current directory. If you include the -classpath . command line option in any event, it will do no harm.

Note that you should avoid storing your source files within the directory structure that was created for the SDK, as this can cause problems. Set up a separate directory of your own to hold the sourcecode for a program and keep the code for each program in its own directory.

Assuming your program contains no errors, the compiler generates a byte code program that is the equivalent of your source code. The compiler stores the byte code program in a file with the same name as the source file, but with the extension .class. Java executable modules are always stored in a file with the extension .class. By default, the .class file will be stored in the same directory as the source file.

The command line options we have introduced here are by no means all the options you have available for the compiler. You will be able to compile all of the examples in the book just knowing about the options we have discussed. There is a comprehensive description of all the options within the documentation for the SDK. You can also specify the -help command line option to get a summary of the standard options you can use.

If you are using some other product to develop your Java programs, you will probably be using a much more user-friendly, graphical interface for compiling your programs that won’t involve entering commands such as that shown above. The file name extensions for your source file and the object file that results from it will be just the same
Executing a Java Application

To execute the byte code program in the .class file with the Java interpreter in the SDK, you make the directory containing the .class file current, and enter the command:

```
java -enableassertions MyProgram
```

Note that we use `MyProgram` to identify the program, *NOT* `MyProgram.class`. It is a common beginner's mistake to use the latter by analogy with the compile operation. If you put a .class file extension on `MyProgram`, your program won't execute and you will get an error message:

```
Exception in thread "main" java.lang.NoClassDefFoundError: MyProgram/class
```

While the compiler expects to find the name of your source file, the java interpreter expects the name of a class, which is `MyProgram` in this case, not the name of a file. The `MyProgram.class` file contains the `MyProgram` class. We will explain what a class is shortly.

The `enableassertions` option is necessary for SDK1.4 programs that use assertions, but since we will be using assertions once we have learned about them it's a good idea to get into the habit of always using this option. You can abbreviate the `-enableassertions` option to `-ea` if you wish.

If you want to override an existing `CLASSPATH` definition, the option is the same as with the compiler. You can also abbreviate `-classpath` to `-cp` with the Java interpreter, but strangely, this abbreviation does not apply to the compiler. Here's how the command would look:

```
java -ea -cp . MyProgram
```
Object-Oriented Programming in Java

As we said at the beginning of this chapter, Java is an object-oriented language. When you use a programming language that is not object oriented, you must express the solution to every problem essentially in terms of numbers and characters—the basic kinds of data that you can manipulate in the language. In an object-oriented language like Java, things are different. Of course, you still have numbers and characters to work with; these are referred to as the basic data types but you can define other kinds of entities that are relevant to your particular problem. You solve your problem in terms of the entities or objects that occur in the context of the problem. This not only affects how a program is structured, but also the terms in which the solution to your problem is expressed. If your problem concerns baseball players, your Java program is likely to have BaseballPlayer objects in it; if you are producing a program dealing with fruit production in California, it may well have objects that are Oranges in it. Apart from seeming to be inherently sensible, object-oriented programs are usually easier to understand.

In Java almost everything is an object. If you haven't delved into object-oriented programming before, or maybe because you have, you may feel this is a bit daunting. But fear not. Objects in Java are particularly easy. So easy, in fact, that we are going to start out by understanding some of the ideas behind Java objects right now. In that way you will be on the right track from the outset.

This doesn't mean we are going to jump in with all the precise nitty-gritty of Java that you need for describing and using objects. We are just going to get the concepts straight at this point. We will do this by taking a stroll through the basics using the odd bit of Java code where it helps the ideas along. All the code that we use here will be fully explained in later chapters. Concentrate on understanding the notion of objects first. Then we can ease into the specific practical details as we go along.

So What Are Objects?

 Anything can be thought of as an object. Objects are all around you. You can consider Tree to be a particular class of objects: trees in general; although it is a rather abstract class as you would be hard pushed to find an actual occurrence of a totally generic tree. Hence the Oak tree in my yard which I call myOak, the Ash tree in your yard which you call thatDarnedTree, and a generalSherman, the well-known redwood, are actual instances of specific types of tree, subclasses of Tree that in this case happen to be Oak, Ash, and Redwood. Note how we drop into the jargon here, class is a term that describes a specification for a collection of objects with common properties.

A class is a specification, or template expressed as a piece of program code which defines what goes to make up a particular sort of object. A subclass is a class that inherits all the properties of the parent class, but that also includes extra specialization. Of course, you will define a class specification to fit what you want to do. There are no absolutes here. For my trivial problem, the specification of a Tree class might just consist of its species and its height. If you are an arboriculturalist, then your problem with trees may require a much more complex class, or more likely a set of classes, that involve a mass of arboreal characteristics.

Every object that your program will use will have a corresponding class definition somewhere for objects of that type. This is true in Java as well as in other object-oriented languages. The basic idea of a class in programming parallels
that of classifying things in the real world. It is a convenient and well-defined way to group things together.

An instance of a class is a technical term for an existing object of that class. Ash is a specification for a type of object and yourAsh is an object constructed to that specification, so yourAsh would be an instance of the class Ash. Once you have a class defined, then you can come up with objects, or instances of that class. This raises the question of what differentiates an object of a given class, an Ash class object say, from a Redwood object. In other words, what sort of information defines a class?

**What Defines a Class of Objects?**

You may have already guessed the answer. A class definition lists all the parameters that you need to define an object of that particular class, at least, so far as your needs go. Someone else might choose a larger or smaller set of parameters to define the same sort of object. It all depends on what you want to do with the class. You will decide what aspects of the objects you need to include to define that particular class of object, and you will choose them depending on the kinds of problems that you want to address using the objects of the class. Let's think about a specific class of objects.

For a class Hat for example, you might use just two parameters in the definition. You could include the type of hat as a string of characters such as "Fedora" or "Baseball cap", and its size as a numeric value. These parameters that define an object of a class are referred to as instance variables or attributes of a class, or class fields. The instance variables can be basic types of data such as numbers, but they could also be other class objects. For example, the name of a Hat object could be of type String the class String defines objects that are strings of characters.

Of course there are lots of other things you could include to define a Hat if you wanted to, color for instance, which might be another string of characters such as "Blue". To specify a class you just decide what set of attributes suit your needs, and those are what you use. This is called data abstraction in the parlance of the object-oriented aficionado, because you just abstract the attributes you want to use from the myriad possibilities for a typical object.

In Java the definition of the class Hat would look something like:
Java Program Structure

Let's summarize the general nature of how a Java program is structured:

• A Java program always consists of one or more classes.

• You typically put the program code for each class in a separate file, and you must give each file the same name as that of the class that is defined within it.

• A Java source file must also have the extension .java.

Thus your file containing the class Hat will be called Hat.java and your file containing the class BaseballPlayer must have the file name BaseballPlayer.java.

A typical program will consist of several files as illustrated in the following diagram.

![Diagram of Java program structure]

This program clearly majors on apparel with four of the five classes representing clothing. Each source file will contain a class definition, and all of the files that go to make up the program will be stored in the same directory. The source files for your program will contain all the code that you wrote, but this is not everything that is ultimately included in the program. There will also be code from the Java standard class library, so let's take a peek at what that can do.

Java's Class Library

A library in Java is a collection of classes usually providing related facilities which you can use in your programs. The Java class library provides you with a whole range of goodies, some of which are essential for your programs to work at all, and some of which make writing your Java programs easier. To say that the standard class library covers a lot of ground would be something of an understatement so we won't be going into it in detail here, but we will be looking into how to apply many of the facilities it provides throughout the book.

Since the class library is a set of classes, it is stored in sets of files where each file contains a class definition. The classes are grouped together into related sets that are called packages, and each package is stored in a separate
directory. A class in a package can access any of the other classes in the package. A class in another package may or may not be accessible. We will learn more about this in Chapter 5.

The package name is based on the path to the directory in which the classes belonging to the package are stored. Classes in the package java.lang for example are stored in the directory path java\lang (or java/lang under Unix). This path is relative to a particular directory that is automatically known by the Java runtime environment that executes your code. You can also create your own packages that will contain classes of your own that you want to reuse in different contexts, and that are related in some way.

The SDK includes a growing number of standard packages well over 100 the last time I counted. Some of the packages you will meet most frequently are:

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang</td>
<td>These classes support the basic language features and the handling of arrays and strings. Classes in this package are always available directly in your programs by default because this package is always automatically loaded with your program.</td>
</tr>
<tr>
<td>java.io</td>
<td>Classes for data input and output operations.</td>
</tr>
<tr>
<td>java.util</td>
<td>This package contains utility classes of various kinds, including classes for managing data within collections or groups of data items.</td>
</tr>
<tr>
<td>javax.swing</td>
<td>These classes provide easy-to-use and flexible components for building graphical user interfaces (GUIs). The components in this package are referred to as Swing components.</td>
</tr>
<tr>
<td>java.awt</td>
<td>Classes in this package provide the original GUI components (JDK1.1) as well as some basic support necessary for Swing components.</td>
</tr>
<tr>
<td>java.awt.geom</td>
<td>These classes define 2-dimensional geometric shapes.</td>
</tr>
<tr>
<td>java.awt.event</td>
<td>The classes in this package are used in the implementation of windowed application to handle events in your program. Events are things like moving the mouse, pressing the left mouse button, or clicking on a menu item.</td>
</tr>
</tbody>
</table>

As noted above, you can use any of the classes from the java.lang package in your programs by default. To use classes from the other packages, you will typically use import statements to identify the names of the classes that you need from each package. This will allow you to reference the classes by the simple class name. Without an import statement you would need to specify the fully qualified name of each class from a package each time you refer to it. As we will see in a moment, the fully qualified name for a class includes the package name as well as the basic class name. Using fully qualified class names would make your program code rather cumbersome, and certainly less...
You can use an import statement to import the name of a single class from a package into your program, or all the class names. The two import statements at the beginning of the code for the applet you saw earlier in this chapter are examples of importing a single class name. The first was:

```java
import javax.swing.JApplet;
```

This statement imports the JApplet class name that is defined in the javax.swing package. Formally, the name of the JApplet class is not really JApplet—it is the fully qualified name javax.swing.JApplet. You can only use the unqualified name when you import the class or the complete package containing it into your program. You can still reference a class from a package even if you don't import it though; you just need to use the full class name, javax.swing.JApplet. You could try this out with the applet you saw earlier if you like. Just delete the two import statements from the file and use the full class names in the program. Then recompile it. It should work the same as before. Thus the fully qualified name for a class is the name of the package in which it is defined, followed by a period, followed by the name given to the class in its definition.

You could import the names of all the classes in the javax.swing package with the statement:

```java
import javax.swing.*;
```

The asterisk specifies that all the class names are to be imported. Importing just the class names that your source code uses makes compilation more efficient, but when you are using a lot of classes from a package you may find it more convenient to import all the names. This saves typing reams of import statements for one thing. We will do this with examples of Java code in the book to keep the number of lines to a minimum. However, there are risks associated with importing all the names in a package. There may be classes with names that are identical to names you have given to your own classes, which would obviously create some confusion when you compile your code.

**Important**

You will see more on how to use import statements in Chapter 5, as well as more about how packages are created and used, and you will be exploring the use of classes from the standard packages in considerable depth throughout the book.

As we indicated earlier, the standard classes do not appear as files or directories on your hard disk. They are packaged up in a single compressed file, rt.jar, that is stored in the jre/lib directory. This directory is created when you install the JDK on your computer. A .jar file is a Java archive—a compressed archive of Java classes. The standard classes that your executable program requires are loaded automatically from rt.jar, so you don't have to be concerned with it directly at all.

### Java Applications

Every Java application contains a class that defines a method called main(). The name of this class is the name that you use as the argument to the Java interpreter when you run the application. You can call the class whatever you want, but the method which is executed first in an application is always called main(). When you run your Java application the method main() will typically cause methods belonging to other classes to be executed, but the simplest possible Java application program consists of one class containing just the method main(). As we shall see below, the main() method has a particular fixed form, and if it is not of the required form, it will not be recognized by the Java interpreter as the method where execution starts.

We'll see how this works by taking a look at just such a Java program. You need to enter the program code using
The program consists of a definition for a class we have called OurFirstProgram. The class definition only contains one method, the method main(). The first line of the definition for the method main() is always of the form:
Java and Unicode

Programming to support languages that use anything other than the Latin character set has always been a major problem. There are a variety of 8-bit character sets defined for many national languages, but if you want to combine the Latin character set and Cyrillic in the same context, for example, things can get difficult. If you want to handle Japanese as well, it becomes impossible with an 8-bit character set because with 8 bits you only have 256 different codes so there just aren't enough character codes to go round. Unicode is a standard character set that was developed to allow the characters necessary for almost all languages to be encoded. It uses a 16-bit code to represent a character (so each character occupies two bytes), and with 16 bits up to 65,535 non-zero character codes can be distinguished. With so many character codes available, there is enough to allocate each major national character set its own set of codes, including character sets such as Kanji which is used for Japanese, and which requires thousand of character codes. It doesn't end there though. Unicode supports three encoding forms that allow up to a million additional characters to be represented.

As we shall see in Chapter 2, Java sourcecode is in Unicode characters. Comments, identifiers (names see Chapter 2), and character and string literals can all use any characters in the Unicode set that represent letters. Java also supports Unicode internally to represent characters and strings, so the framework is there for a comprehensive international language capability in a program. The normal ASCII set that you are probably familiar with corresponds to the first 128 characters of the Unicode set. Apart from being aware that each character occupies two bytes, you can ignore the fact that you are handling Unicode characters in the main, unless of course you are building an application that supports multiple languages from the outset.
Summary

In this chapter we have looked at the basic characteristics of Java, and how portability between different computers is achieved. We have also introduced the elements of object-oriented programming. There are bound to be some aspects of what we have discussed that you don't feel are completely clear to you. Don't worry about it. Everything we have discussed here we will be revisiting again in more detail later on in the book.

The essential points we have covered in this chapter are:

• Java applets are programs that are designed to be embedded in an HTML document. Java applications are standalone programs. Java applications can be console programs that only support text output to the screen, or they can be windowed applications with a GUI.

• Java programs are intrinsically object-oriented.

• Java sourcecode is stored in files with the extension .java.

• Java programs are compiled to byte codes, which are instructions for the Java Virtual Machine. The Java Virtual Machine is the same on all the computers on which it is implemented, thus ensuring the portability of Java programs.

• Java object code is stored in files with the extension .class.

• Java programs are executed by the Java interpreter, which analyses the byte codes and carries out the operations they specify.

• The Java System Development Kit (the SDK) supports the compilation and execution of Java applications and applets.

• Experience is what you get when you are expecting something else.
Resources

You can download the sourcecode for the examples in the book from any of:

- http://www.wrox.com
- ftp://www.wrox.com
- ftp://www.wrox.co.uk

The sourcecode download also includes ancillary files, such as .gif files containing icons for instance, where they are used in the examples.

If you have any questions on the fine formal detail of Java, the reference works we've used are:


Other sites of interest are:


- http://p2p.wrox.com for lists where you can get answers to your Java problems.


and for online magazine reading and opinion, check out:
We also like the Java Developer Connection, subscribe to it at http://java.sun.com/jdc
Chapter 2: Programs, Data, Variables, and Calculation

Overview

In this chapter we will look at the entities in Java that are not objects—numbers and characters. This will give you all the elements of the language you need to perform numerical calculations, and we will apply these in a few working examples.

By the end of this chapter you will have learnt:

• How to declare and define variables of the basic integer and floating point types

• How to write an assignment statement

• How integer and floating point expressions are evaluated

• How to output data from a console program

• How mixed integer and floating point expressions are evaluated

• What casting is and when you must use it

• What boolean variables are
What determines the sequence in which operators in an expression are executed

•

How to include comments in your programs
Data and Variables

A variable is a named piece of memory that you use to store information in your Java program—a piece of data of some description. Each named piece of memory that you define in your program will only be able to store data of one particular type. If you define a variable to store integers, for example, you cannot use it to store a value that is a decimal fraction, such as 0.75. If you have defined a variable that you will use to refer to a Hat object, you can only use it to reference an object of type Hat (or any of its subclasses, as we saw in Chapter 1). Since the type of data that each variable can store is fixed, whenever you use a variable in your program the compiler is able to check that it is not being used in a manner or a context that is inappropriate to its type. If a method in your program is supposed to process integers, the compiler will be able to detect when you inadvertently try to use the method with some other kind of data, for example, a string or a numerical value that is not integral.

Explicit data values that appear in your program are called literals. Each literal will also be of a particular type: 25, for instance, is an integer value of type int. We will go into the characteristics of the various types of literals that you can use as we discuss each variable type.

Before you can use a variable you must specify its name and type in a declaration statement. Before we look at how you write a declaration for a variable, we should consider what flexibility you have in choosing a name.

Variable Names

The name that you choose for a variable, or indeed the name that you choose for anything in Java, is called an identifier. An identifier can be any length, but it must start with a letter, an underscore (_), or a dollar sign ($). The rest of an identifier can include any characters except those used as operators in Java (such as +, -, or *), but you will be generally better off if you stick to letters, digits, and the underscore character.

Java is case sensitive, so the names republican and Republican are not the same. You must not include blanks or tabs in the middle of a name, so Betty May is out, but you could have BettyMay or even Betty_May. Note that you can't have 10Up as a name since you cannot start a name with a numeric digit. Of course, you could use tenUp as an alternative.

Subject to the restrictions we have mentioned, you can name a variable almost anything you like, except for two additional restraints—you can't use keywords in Java as a name for something, and a name can't be anything that is a constant value. Keywords are words that are an essential part of the Java language. We saw some keywords in the previous chapter and we will learn a few more in this chapter. If you want to know what they all are, a complete list appears in Appendix A. The restriction on constant values is there because, although it is obvious why a name can't be 1234 or 37.5, constants can also be alphabetic, such as true and false for example. We will see how we specify constant values later in this chapter. Of course, the basic reason for these rules is that the compiler has to be able to distinguish between your variables and other things that can appear in a program. If you try to use a name for a variable that makes this impossible, then it's not a legal name.

Clearly, it makes sense to choose names for your variables that give a good indication of the sort of data they hold. If
you want to record the size of a hat, for example, hatSize is not a bad choice for a variable name whereas qqq would be a bad choice. It is a common convention in Java to start variable names with a lower case letter and, where you have a name that combines several words, to capitalize the first letter of each word, as in hatSize or moneyWellSpent. You are in no way obliged to follow this convention but since almost all the Java world does, it helps to do so.

Note

If you feel you need more guidance in naming conventions (and coding conventions in general) take a look at http://www.javasoft.com/docs/codecon.

Variable Names and Unicode

Even though you are likely to be entering your Java programs in an environment that stores ASCII, all Java source code is in Unicode (subject to the reservations we noted in Chapter 1). Although the original source that you create is ASCII, it is converted to Unicode characters internally, before it is compiled. While you only ever need ASCII to write any Java language statement, the fact that Java supports Unicode provides you with immense flexibility. It means that the identifiers that you use in your source program can use any national language character set that is defined within the Unicode character set, so your programs can use French, Greek, or Cyrillic variable names, for example, or even names in several different languages, as long as you have the means to enter them in the first place. The same applies to character data that your program defines.

Variables and Types

As we mentioned earlier, each variable that you declare can store values of a type determined by the data type of that variable. You specify the type of a particular variable by using a type name in the variable declaration. For instance, here's a statement that declares a variable that can store integers:
Integer Data Types

There are four types of variables that you can use to store integer data. All of these are signed, that is, they can store both negative and positive values. The four integer types differ in the range of values they can store, so the choice of type for a variable depends on the range of data values you are likely to need.

The four integer types in Java are:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Variables of this type can have values from -128 to +127 and occupy 1 byte (8 bits) in memory</td>
</tr>
<tr>
<td>short</td>
<td>Variables of this type can have values from -32768 to 32767 and occupy 2 bytes (16 bits) in memory</td>
</tr>
<tr>
<td>int</td>
<td>Variables of this type can have values from -2147483648 to 2147483647 and occupy 4 bytes (32 bits) in memory</td>
</tr>
<tr>
<td>long</td>
<td>Variables of this type can have values from -9223372036854775808 to 9223372036854775807 and occupy 8 bytes (64 bits) in memory</td>
</tr>
</tbody>
</table>

Let's take a look at declarations of variables of each of these types:
Floating Point Data Types

Numeric values that are not integral are stored as floating point numbers. A floating point number has a fixed number of digits of accuracy but with a very wide range of values. You get a wide range of values, even though the number of digits is fixed, because the decimal point can "float". For example, the values 0.000005, 500.0, and 5000000000000.0 can be written as 5x10^-6, 5x10^2, and 5x10^12 respectively. We have just one digit '5' but we move the decimal point around.

There are two basic floating point types in Java, float and double. These give you a choice in the number of digits precision available to represent your data values, and in the range of values that can be accommodated:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Variables of this type can have values from -3.4E38 (-3.4x10^38) to +3.4E38 (+3.4x10^38) and occupy 4 bytes in memory. Values are represented with approximately 7 digits accuracy.</td>
</tr>
<tr>
<td>double</td>
<td>Variables of this type can have values from -1.7E308 (-1.7x10^308) to +1.7E308 (+1.7x10^308) and occupy 8 bytes in memory. Values are represented with approximately 17 digits accuracy. The smallest non-zero value that you can have is roughly 4.9x10^-324.</td>
</tr>
</tbody>
</table>

Important

All floating point operations and the definitions for values of type float and type double in Java conform to the IEEE 754 standard.

As with integer calculations, floating point calculations in Java will produce the same results on any computer.

Floating Point Values

When you are specifying floating point literals they are of type double by default, so 1.0 and 345.678 are both of type double. When you want to specify a value of type float, you just append an f, or an F, to the value, so 1.0f and 345.678F are both constants of type float. If you are new to programming it is important to note that you must not include commas as separators when specifying numerical values in your program code. Where you might normally write a value as 99,786.5, in your code you must write it without the comma, as 99786.5.

When you need to write very large or very small floating point values, you will usually want to write them with an exponent—that is, as a decimal value multiplied by a power of 10. You can do this in Java by writing the number as a decimal value followed by an E, or an e, preceding the power of 10 that you require. For example, the distance from the Earth to the Sun is approximately 149,600,000 kilometers, more conveniently written as 1.496E8. Since the E
(or e) indicates that what follows is the exponent, this is equivalent to $1.496\times10^8$. At the opposite end of the scale, the mass of an electron is around $0.000000000000000000000000009$ grams. This is much more convenient, not to say more readable, when it is written as $9.0\times10^{-28}$ grams.

### Declaring Floating Point Variables

You declare floating point variables in a similar way to that we've already used for integers. We can declare and initialize a variable of type double with the statement:
Arithmetic Calculations

You store the result of a calculation in a variable by using an assignment statement. An assignment statement consists of a variable name followed by an assignment operator, followed by an arithmetic expression, followed by a semicolon. Here is a simple example of an assignment statement:
Mixed Arithmetic Expressions

You can mix values of the basic types together in a single expression. The way mixed expressions are treated is governed by some simple rules that apply to each operator in such an expression. The rules, in the sequence in which they are checked, are:

- If either operand is of type double, the other is converted to double before the operation is carried out.

- If either operand is of type float, the other is converted to float before the operation is carried out.

- If either operand is of type long, the other is converted to long before the operation is carried out.

The first rule in the sequence that applies to a given operation is the one that is carried out. If neither operand is double, float, or long, they must be int, short, or byte, so they use 32-bit arithmetic as we saw earlier.

Explicit Casting

It may well be that the default treatment of mixed expressions listed above is not what you want. For example, if you have a double variable result, and you compute its value using two int variables three and two with the values 3 and 2 respectively, with the statement:
The op= Operators

The op= operators are used in statements of the form:

\[ lhs \ op= rhs; \]

where \( \op \) can be any of the operators +, -, *, /, %, plus some others you haven't seen yet. The above is basically a shorthand representation of the statement:

\[ lhs = lhs \ op (rhs); \]

The right hand side is in brackets because it is worked out first then the result is combined with the left hand side using the operation, \( \op \). Let's look at a few examples of this to make sure it's clear. To increment an int variable count by 5 you can write:
Mathematical Functions and Constants

Sooner or later you are likely to need mathematical functions in your programs, even if it's only obtaining an absolute value or calculating a square root. Java provides a range of methods that support such functions as part of the standard library stored in the package java.lang, and all these are available in your program automatically.

The methods that support various additional mathematical functions are implemented in the class Math as static methods, so to reference a particular function you can just write Math and a period in front of the name of the method you wish to use. For example, to use sqrt(), which calculates the square root of whatever you place between the parentheses, you would write Math.sqrt(aNumber) to produce the square root of the floating point value in the variable aNumber.

The class Math includes a range of methods for standard trigonometric functions. These are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(arg)</td>
<td>sine of the argument</td>
<td>double in radians</td>
<td>double</td>
</tr>
<tr>
<td>cos(arg)</td>
<td>cosine of the argument</td>
<td>double in radians</td>
<td>double</td>
</tr>
<tr>
<td>tan(arg)</td>
<td>tangent of the argument</td>
<td>double in radians</td>
<td>double</td>
</tr>
<tr>
<td>asin(arg)</td>
<td>sin-1 (arc sine) of the argument</td>
<td>double</td>
<td>double in radians with values from ( \frac{\pi}{2} ) to ( \frac{\pi}{2} ).</td>
</tr>
<tr>
<td>acos(arg)</td>
<td>cos-1 (arc cosine) of the argument</td>
<td>double</td>
<td>double in radians, with values from 0.0 to ( \pi ).</td>
</tr>
<tr>
<td>atan(arg)</td>
<td>tan-1 (arc tangent) of the argument</td>
<td>double</td>
<td>double in radians with values from ( \frac{\pi}{2} ) to ( \frac{\pi}{2} ).</td>
</tr>
<tr>
<td>atan2(arg1,arg2)</td>
<td>tan-1 (arc tangent) of arg1/arg2</td>
<td>Both double</td>
<td>double in radians with values from ( \pi ) to ( -\pi ).</td>
</tr>
</tbody>
</table>

As with all methods, the arguments that you put between the parentheses following the method name can be any
expression that produces a value of the required type. If you are not familiar with these trigonometric operations you can safely ignore them.

You also have a range of useful numerical functions that are implemented in the class Math. These are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Argument type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(arg)</td>
<td>Calculates the absolute value of the argument</td>
<td>int, long, float, or double</td>
<td>The same type as the argument</td>
</tr>
<tr>
<td>max (arg1, arg2)</td>
<td>Returns the larger of the two arguments, both of the same type</td>
<td>int, long, float, or double</td>
<td>The same type as the argument</td>
</tr>
<tr>
<td>min (arg1, arg2)</td>
<td>Returns the smaller of the two arguments, both of the same type</td>
<td>int, long, float, or double</td>
<td>The same type as the argument</td>
</tr>
<tr>
<td>ceil(arg)</td>
<td>Returns the smallest integer that is greater than or equal to the argument</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>floor(arg)</td>
<td>Returns the largest integer that is less than or equal to the argument</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>round(arg)</td>
<td>Calculates the nearest integer to the argument value</td>
<td>float or double</td>
<td>Of type int for a float argument, of type long for a double argument</td>
</tr>
<tr>
<td>rint(arg)</td>
<td>Calculates the nearest integer to the argument value</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>IEEEremainder (arg1, arg2)</td>
<td>Calculates the remainder when arg1 is divided by arg2</td>
<td>Both of type double</td>
<td>Of type double</td>
</tr>
</tbody>
</table>

The IEEEremainder() method produces the remainder from arg1 after dividing arg2 into arg1 the integral number of times that is closest to the exact value of arg1/arg2. This is somewhat different from the remainder operator. The operation arg1 % arg2 produces the remainder after dividing arg2 into arg1 the integral number of times that does not exceed the absolute value of arg1. In some situations this can result in markedly different results. For example, executing the expression 9.0 % 5.0 results in 4.0, whereas the expression Math.IEEEremainder(9.0, 5.0) results in 1.0. You can pick one approach to calculating the remainder or the other, to suit your requirements.

Where more than one type of argument is noted in the table, there are actually several methods, one for each type of argument, but all have the same name. We will see how this is possible in Java when we look at implementing class
methods in Chapter 5.

The mathematical functions available in the class Math are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Function</th>
<th>Argument Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(arg)</td>
<td>Calculates the square root of the argument</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>pow (arg1, arg2)</td>
<td>Calculates the first argument raised</td>
<td>Both double</td>
<td>double</td>
</tr>
<tr>
<td>exp(arg)</td>
<td>Calculates e raised to the power of the argument</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>log(arg)</td>
<td>Calculates the natural logarithm (base e) of the argument</td>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>random()</td>
<td>Returns a pseudo-random number greater than or equal to 0.0 and less than 1.0</td>
<td>None</td>
<td>double</td>
</tr>
</tbody>
</table>

The toRadians() method in the class Math will convert a double argument that is an angular measurement in degrees to radians. There is a complementary method, toDegrees(), to convert in the opposite direction. The Math class also defines double values for e and π, which you can access as Math.E and Math.PI respectively.

Let's try out a sample of the contents of the class Math in an example to make sure we know how they are used.

**Try It Out  The Math Class**

The following program will calculate the radius of a circle in feet and inches, given that it has an area of 100 square feet:
Storing Characters

Variables of the type char store a single character. They each occupy 16 bits, two bytes, in memory because all characters in Java are stored as Unicode. To declare and initialize a character variable myCharacter you would use the statement:
**Bitwise Operations**

As you already know, all these integer variables we have been talking about are represented internally as binary numbers. A value of type int consists of 32 binary digits, known to us computer fans as bits. You can operate on the bit values of integers using the bitwise operators, of which there are four available:

<table>
<thead>
<tr>
<th>&amp;</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OR</td>
</tr>
<tr>
<td>^</td>
<td>Exclusive OR</td>
</tr>
<tr>
<td>~</td>
<td>Complement</td>
</tr>
</tbody>
</table>

Each of these operators works with individual bits as follows:

- The bitwise AND operator, &, combines corresponding bits in its two operands such that if the first bit AND the second bit are 1, the result is 1 otherwise the result is 0.

- The bitwise OR operator, |, combines corresponding bits such that if either one bit OR the other is 1, then the result is 1. Only if both bits are 0 is the result 0.

- The bitwise exclusive OR (XOR) operator, ^, combines corresponding bits such that if both bits are the same the result is 0, otherwise the result is 1.

- The complement operator, ~, takes a single operand in which it inverts all the bits, so that each 1 bit becomes 0, and each 0 bit becomes 1.

You can see the effect of these operators in the following examples.
The illustration shows the binary digits that make up the operands and the results. Each of the three binary operations applies to corresponding individual pairs of bits from the operands separately. The complement operator flips the state of each bit in the operand.

Since you are concerned with individual bits with bitwise operations, writing a constant as a normal decimal value is not going to be particularly convenient. A much better way of writing binary values in this case is to express them as hexadecimal numbers, because you can convert from binary to hexadecimal, and vice versa, very quickly. There's more on this in Appendix B.

Converting from binary to hexadecimal is easy. Each group of four binary digits from the right corresponds to one hexadecimal digit. You just work out what the value of each four bits is and write the appropriate hexadecimal digit. For example, the value of a from the illustration is:

<table>
<thead>
<tr>
<th>Binary</th>
<th>0110</th>
<th>0110</th>
<th>1100</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal value</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>6</td>
<td>6</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

So the value of the variable a in hexadecimal is 0x66CD, where the 0x prefix indicates that this is a hexadecimal value. The variable b in the illustration has the hexadecimal value 0x000F. If you think of the variable b as a mask applied to a, you can view the & operator as keeping bits unchanged where the mask is 1 and setting the rest to 0. Mask is a term used to refer to a particular configuration of bits designed to select out specific bits when it is combined with a variable using a bitwise operator. So if you want to select a particular bit out of an integer variable, just AND it with a mask that has that bit set to 1 and all the others as 0.

You can also look at what the & operator does from another perspective - it forces a bit to 0, if the corresponding mask bit is 0. Similarly, the | operator forces a bit to be 1 when the mask bit is 1.

The & and | operators are the most frequently used, mainly for dealing with variables where the individual bits are used as state indicators of some kind for things that can be either true or false, or on or off. You could use a single bit as a state indicator determining whether something should be displayed, with the bit as 1, or not displayed, with the bit as 0. A single bit can be selected using the & operator - for example, to select the third bit in the int variable indicators, you can write:
Boolean Variables

Variables that can only have one of two values, true or false, are of type boolean, and the values true and false are boolean literals. You can define a boolean variable, called state, with the statement:
Operator Precedence

We have already introduced the idea of a pecking order for operators, which determines the sequence in which they are executed in a statement. A simple arithmetic expression such as $3 + 4 \times 5$ results in the value 23 because the multiply operation is executed first—it takes precedence over the addition operation. We can now formalize the position by classifying all the operators present in Java. Each operator in Java has a set priority or precedence in relation to the others, as shown in the following table. Operators with a higher precedence are executed before those of a lower precedence. Precedence is highest for operators in the top line in the table, down through to the operators in the bottom line, which have the lowest precedence:

<table>
<thead>
<tr>
<th>Operator Precedence Group</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(, [ ], . postfix ++, postfix --</td>
<td>left</td>
</tr>
<tr>
<td>unary +, unary -, prefix ++, prefix --, ~, !</td>
<td>right</td>
</tr>
<tr>
<td>(type), new</td>
<td>left</td>
</tr>
<tr>
<td>*, /, %</td>
<td>left</td>
</tr>
<tr>
<td>+, -</td>
<td>left</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;, &gt;&gt;&gt;</td>
<td>left</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=, instanceof</td>
<td>left</td>
</tr>
<tr>
<td>==, !=</td>
<td>left</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td>^</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>left</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>left</td>
</tr>
<tr>
<td>=, +=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &gt;&gt;&gt;=, &amp;=,</td>
<td>=, ^=</td>
</tr>
</tbody>
</table>

**Note**

Most of the operators that appear in the table you have not seen yet, but you will meet them all in this book eventually, and it is handy to have them all gathered together in a single precedence table that you can refer to when necessary.

By definition, the postfix ++ operator is executed after the other operators in the expression in which it appears, despite its high precedence. In this case, precedence determines what it applies to, in other words, the postfix ++ only acts on the variable that appears immediately before it. For this reason the expression oranges+++apples that we saw earlier is evaluated as (oranges++) + apples rather than oranges + (++apples).

The sequence of execution of operators with equal precedence in a statement is determined by a property called associativity. Each group of operators appearing on the same line in the table above are either left associative or right associative. A left associative operator attaches to its immediate left operand. This results in an expression involving several left associative operators with the same precedence in the same expression being executed in sequence starting with the leftmost and ending with the rightmost. Right associative operators of equal precedence in an expression bind to their right operand and consequently are executed from right to left. For example, if you write the statement:
Program Comments

We have been adding comments in all our examples so far, so you already know that // plus everything following in a line is ignored by the compiler (except when the // appears in a character string between double quotes of course). Another use for // is to comment out lines of code. If you want to remove some code from a program temporarily, you just need to add // at the beginning of each line you want to eliminate.

It is often convenient to include multiple lines of comment in a program, for example, at the beginning of a method to explain what it does. An alternative to using // at the beginning of each line in a block of comments is to put /* at the beginning of the first comment line and */ at the end of the last comment line. Everything between the /* and the next */ will be ignored. By this means you can annotate your programs, like this for example:
Summary

In this chapter you have seen all of the basic types of variables available in Java. The discussion of boolean variables will be more meaningful in the context of the next chapter since their primary use is in decision making and modifying the execution sequence in a program.

The important points you have learned in this chapter are:

- The integer types are byte, short, int, and long, occupying 1, 2, 4, and 8 bytes respectively.

- Variables of type char occupy 2 bytes and can store a single Unicode character code.

- Integer expressions are evaluated using 64-bit operations for variables of type long, and using 32-bit operations for all other integer types. You must therefore add a cast for all assignment operations storing a result of type byte, short, or char.

- A cast will be automatically supplied where necessary for op= assignment operations.

- The floating-point types are float and double, occupying 4 and 8 bytes respectively.

- Values that are outside the range of a floating-point type are represented by a special value that is displayed as either Infinity or -Infinity.

- Where the result of a floating point calculation is indeterminate, the value is displayed as NaN. Such values are referred to as not-a-number.

- Variables of type boolean can only have either the value true or the value false.

- The order of execution of operators in an expression is determined by their precedence. Where operators
are of equal precedence, the order of execution is determined by their associativity.
Exercises

1.

Write a console program to define and initialize a variable of type byte to 1, and then successively multiply it by 2 and display its value 8 times. Explain the reason for the last result.

2.

Write a console program to declare and initialize a double variable with some value such as 1234.5678. Then retrieve the integral part of the value and store it in a variable of type long, and the first four digits of the fractional part and store them in an integer of type short. Display the value of the double variable by outputting the two values stored as integers.

3.

Write a program that defines a floating point variable initialized with a dollar value for your income and a second floating point variable initialized with a value corresponding to a tax rate of 35 percent. Calculate and output the amount of tax you must pay with the dollars and cents stored as separate integer values (use two variables of type int to hold the tax, perhaps taxDollars and taxCents).

4.

The diameter of the Sun is approximately 865,000 miles. The diameter of the Earth is approximately 7600 miles. Use the methods in the class Math to calculate:

- the volume of the Earth in cubic miles
- the volume of the Sun in cubic miles
- the ratio of the volume of the Sun to the volume of the Earth
Chapter 3: Loops and Logic

Overview

In this chapter we'll look at how you make decisions and choices in your Java programs. You will also learn how to make your programs repeat a set of actions until a specific condition is met. We will cover:

- How you compare data values
- How you can define logical expressions
- How you can use logical expressions to alter the sequence in which program statements are executed
- How you can select different expressions depending on the value of a logical expression
- How to choose between options in a fixed set of alternatives
- How long your variables last
- How you can repeat a block of code a given number of times
- How you can repeat a block of code as long as a given logical expression is true
- How you can break out of loops and statement blocks
What assertions are and how you use them

All your programs of any consequence will use at least some, and often most, of the language capabilities and programming techniques we will cover in this chapter, so make sure you have a good grasp of them.

But first, how do we make decisions in code, and so affect the way the program runs?
Making Decisions

Making choices will be a fundamental element in all your programs. You need to be able to make decisions like, "If the bank balance is large, buy the car with the go-faster stripes, else renew the monthly bus ticket". In programming terms this requires the ability to make comparisons between variables, constants, and the values of expressions, then executing one group of statements or another, depending on the result of a given comparison. The first step to making decisions in a program is to look at how we make comparisons.

Making Comparisons

Java provides you with six relational operators for comparing two data values. Either of the data values you are comparing can be variables, constants or expressions drawn from Java's primitive data types byte, short, int, long, char, float or double.

<table>
<thead>
<tr>
<th>Relational Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Produces the value true if the left operand is greater than the right operand, and false otherwise.</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Produces the value true if the left operand is greater than or equal to the right operand, and false otherwise.</td>
</tr>
<tr>
<td>==</td>
<td>Produces the value true if the left operand is equal to the right operand, and false otherwise.</td>
</tr>
<tr>
<td>!=</td>
<td>Produces the value true if the left operand is not equal to the right operand, and false otherwise.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Produces the value true if the left operand is less than or equal to the right operand, and false otherwise.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Produces the value true if the left operand is less than the right operand, and false otherwise.</td>
</tr>
</tbody>
</table>

As you see, each operator produces either the value true, or the value false, and so is eminently suited to the business of making decisions.

If you wish to store the result of a comparison, you use a boolean variable. We saw how to declare these in the
previous chapter. For example you can define a boolean variable state and you can set its value in an assignment as follows:
Logical Operators

The tests we have put in the if expressions have been relatively simple so far, except perhaps for the last one. Real life is typically more complicated. You will often want to combine a number of conditions so that you execute a particular course, for example, if they are all true simultaneously. You can ride the roller coaster if you are over 12 years old, over four feet tall, and less than six feet six. Failure on any count and it's no go. Sometimes, though, you may need to test for any one of a number of conditions being true, for example, you get a lower price entry ticket if you are under 16, or over 65.

You can deal with both of these cases, and more, using logical operators to combine several expressions that have a value true or false. Because they operate on boolean values they are also referred to as boolean operators. There are five logical operators that operate on boolean values:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Long name</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>conditional AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical negation (NOT)</td>
</tr>
</tbody>
</table>

These are very simple, the only point of potential confusion being the fact that we have the choice of two operators for each of AND and OR. The extra operators are the bitwise & and | from the previous chapter that you can also apply to boolean values where they have an effect that is subtly different from && and ||. We'll first consider what each of these are used for in general terms, then we'll look at how we can use them in an example.

Boolean AND Operations

You can use either AND operator, && or & where you have two logical expressions that must both be true for the result to be true that is, you want to be rich and healthy. Either operator will produce the same result from the logical expression. We will come back to how they are different in a moment. First, let's explore how they are used. All of the following discussion applies equally well to & as well as &&.

Let's see how logical operators can simplify the last example. You could use the && operator if you were testing a
variable of type char to determine whether it contained an uppercase letter or not. The value being tested must be both greater than or equal to 'A' AND less than or equal to 'Z'. Both conditions must be true for the value to be a capital letter. Taking the example from our previous program, with a value stored in a char variable symbol, we could implement the test for an uppercase letter in a single if by using the && operator:
The Conditional Operator

The conditional operator is sometimes called a ternary operator because it involves three operands. It is best understood by looking at an example. Suppose we have two variables of type int, yourAge and myAge, and we want to assign the greater of the values stored in yourAge and myAge to a third variable also of type int, older. We can do this with the statement:
The switch Statement

The switch statement enables you to select from multiple choices based on a set of fixed values for a given expression. The expression must produce a result of type char, byte, short or int, but not long, otherwise the statement will not compile. In normal use it operates rather like a rotary switch in that you can select one of a fixed number of choices. For example, on some makes of washing machine you choose between the various possible machine settings in this way, with positions for cotton, wool, synthetic fiber and so on, which you select by turning the knob to point to the option that you want.

A switch statement reflecting this logic would be:
Variable Scope

The **scope** of a variable is the part of the program over which the variable name can be referenced, in other words, where you can use the variable in the program. Every variable that we have declared so far in program examples has been defined within the context of a method, the method main(). Variables that are declared within a method are called local variables, as they are only accessible within the confines of the method in which they are declared. However, they are not necessarily accessible everywhere in the code for the method in which they are declared. Look at the example in the illustration below that shows nested blocks inside a method.

```
Loops

A loop allows you to execute a statement or block of statements repeatedly. The need to repeat a block of code arises in almost every program. If you did the first exercise at the end of the last chapter, based on what you had learned up to that point you would have come up with a program along the lines of:
Assertions

Every so often you will find that the logic in your code leads to some logical condition that should always be true. If you test an integer and establish that it is odd, it is certainly true that it cannot be even for instance. You may also find yourself writing a statement or statements that, although they could be executed in theory, in practice they never really should be. I don't mean by this the usual sorts of errors that occur such as some incorrect data has been read in somehow. This should be handled ordinarily by the normal code. I mean circumstances where if the statements were to be executed, it would imply that something was very seriously wrong with the program or its environment. These are precisely the circumstances to which assertions apply.

A simple assertion is a statement of the form:
Summary

In this chapter you have learned about all of the essential mechanisms for making decisions in Java. You have also learned all of the looping facilities that you have available when programming. The essential points we have covered are:

• You can use relational operators to compare values, and such comparisons result in values of either true or false.

• You can combine basic comparisons and logical variables in more complex logical expressions by using logical operators.

• The if statement is a basic decision making tool in Java. It enables you to choose to execute a block of statements if a given logical expression has the value true. You can optionally execute another block of statements if the logical expression is false by using the else keyword.

• You can use the conditional operator to choose between two expressions depending on the value of a logical expression.

• You can use the switch statement to choose from a fixed number of alternatives.

• The variables in a method come into existence at the point at which you declare them, and cease to exist after the end of the block that immediately encloses their declaration. The program extent where the variable is accessible is the scope of the variable.

• You have three options for repeating a block of statements, a for loop, a while loop or a do while loop.

• The continue statement enables you to skip to the next iteration in the loop containing the continue statement.

• The labeled continue statement enables you to skip to the next iteration in a loop enclosing the labeled
continue that is identified by the label. The labeled loop need not be that immediately enclosing the labeled continue.

•

The break statement enables you to break out of a loop or block of statements in which it appears.

•

The labeled break statement enables you to break out of a loop or block of statements that encloses it that is identified by the label. This is not necessarily the block that encloses it directly.

•

You use an assertion statement to verify logical conditions that should always be true, or as code in parts of a program that should not be reached, but theoretically can be.

•

Horse sense is the thing a horse has which keeps it from betting on people.
Exercises

1.

Write a program to display a random choice from a set of six choices for breakfast (you could use any set, for example, scrambled eggs, waffles, fruit, cereal, toast or yogurt).

2.

When testing whether an integer is a prime, it is sufficient to try to divide by integers up to the square root of the number being tested. Rewrite the program example from this chapter to use this approach.

3.

A lottery requires that you select six different numbers from the integers 1 to 49. Write a program to do this for you and generate five sets of entries.

4.

Write a program to generate a random sequence of capital letters that does not include vowels.
Chapter 4: Arrays and Strings

Overview

In this chapter you will start to use Java objects. You will first be introduced to arrays, which enable you to deal with a number of variables of the same type through a single variable name, and then you will look at how to handle character strings. By the end of this chapter you will have learned:

• What arrays are and how you declare and initialize them.

• How you access individual elements of an array.

• How you can use individual elements of an array.

• How to declare arrays of arrays.

• How you can create arrays of arrays with different lengths.

• How to create String objects.

• How to create and use arrays of String objects.

• What operations are available for String objects.

• What StringBuffer objects are and how they relate to operations on String objects.
What operations are available for StringBuffer objects.

Some of what we discuss in this chapter relates to objects, and as we have not yet covered in detail how a class (or object definition) is defined we will have to skate over some points, but all will be revealed in Chapter 5.
Arrays

With the basic built-in Java data types we have seen in the previous chapters, each identifier corresponds to a single variable. But when you want to handle sets of values of the same type  the first 1000 primes for example  you really don't want to have to name them individually. What you need is an array.

An array is a named set of variables of the same type. Each variable in the array is called an array element. To reference a particular element in an array you use the array name combined with an integer value of type int, called an index. The index for an array element is the offset of that particular element from the beginning of the array. The first element will have an index of 0, the second will have an index of 1, the third an index of 2, and so on. The index value does not need to be an integer literal. It can be any expression that results in a value of type int equal to or greater than zero. Obviously a for loop control variable is going to be very useful for processing array elements which is why you had to wait until now to hear about arrays.

Array Variables

You are not obliged to create the array itself when you declare the array variable. The array variable is distinct from the array itself. You could declare the integer array variable primes with the statement:
Arrays of Characters

All our arrays have been numeric so far. You can also have arrays of characters. For example, we can declare an array variable of type char[] to hold 50 characters with the statement:
Strings

You will need to use character strings in most of your programs—headings, names, addresses, product descriptions...the list is endless. In Java, strings are objects of the class String. The String class is a standard class that comes with Java, and it is specifically designed for creating and processing strings.

String Literals

You have already made extensive use of string literals for output. Just about every time the println() method was used in an example, we used a string literal as the argument. A string literal is a sequence of characters between double quotes:

"This is a string literal!"

This is actually a String literal with a capital S—in other words, a constant object of the class String that the compiler creates for use in your program.

Some characters can't be entered explicitly from the keyboard for inclusion in a string literal. You can't include a double quote character as it is, for example, as this is used to indicate where a string literal begins and ends. You can't include a newline character by pressing the Enter key since this will move the cursor to a new line. As we saw back in Chapter 2, all of these characters are provided in the same way as char constants—you use an escape sequence. All the escape sequences you saw when we looked at char constants apply to strings. The statement:
Operations on Strings

There are many kinds of operations that can be performed on strings, but we can start with one you have used already, joining strings together, often called string concatenation.

Joining Strings

To join two String objects to form a single string you use the + operator, just as you have been doing with the argument to the println() method in the program examples thus far. The simplest use of this is to join two strings together:
StringBuffer Objects

String objects cannot be changed, but we have been creating strings that are combinations and modifications of existing String objects, so how is this done? Java has another standard class for defining strings, StringBuffer, and a StringBuffer object can be altered directly. Strings that can be changed are often referred to as **mutable strings** whereas a String object is an **immutable string**. Java uses objects of the class StringBuffer internally to perform many of the operations on String objects. You can use a StringBuffer object whenever you need a string that you can change directly.

So when do you use StringBuffer objects rather than String objects? StringBuffer objects come into their own when you are transforming strings - adding, deleting, or replacing substrings in a string. Operations will be faster and easier using StringBuffer objects. If you have static strings, which you occasionally need to concatenate, then String objects will be the best choice. Of course, if you want to you can mix the use of both in the same program.

Creating StringBuffer Objects

You can create a StringBuffer object that contains a given string with the statement:
Summary

You should now be thoroughly familiar with how to create and use arrays. Most people have little trouble dealing with one-dimensional arrays, but arrays of arrays are a bit more tricky so try to practice using these.

You have also acquired a good knowledge of what you can do with String and StringBuffer objects. Most operations with these objects are very straightforward and easy to understand. Being able to decide which methods you should apply to the solution of specific problems is a skill that will come with a bit of practice.

The essential points that we have discussed in this chapter are:

1. You use an array to hold multiple values of the same type, identified through a single variable name.

2. You reference an individual element of an array by using an index value of type int. The index value for an array element is the offset of that element from the first element in the array.

3. An array element can be used in the same way as a single variable of the same type.

4. You can obtain the number of elements in an array by using the length member of the array object.

5. An array element can also contain an array, so you can define arrays of arrays, or arrays of arrays of arrays.

6. A String object stores a fixed character string that cannot be changed. However, you can assign a given String variable to a different String object.

7. You can obtain the number of characters stored in a String object by using the length() method for the object.

8. The String class provides methods for joining, searching, and modifying strings - the modifications being achieved by creating a new String object.

9. A StringBuffer object can store a string of characters that can be modified.
You can get the number of characters stored in a StringBuffer object by calling its length() method, and you can find out the current maximum number of characters it can store by using its capacity() method.

You can change both the length and the capacity for a StringBuffer object.

The StringBuffer class contains a variety of methods for modifying StringBuffer objects.

You can create a String object from a StringBuffer object by using the toString() method of the StringBuffer object.
Exercises

1.

Create an array of String variables and initialize the array with the names of the months from January to December. Create an array containing 12 random decimal values between 0.0 and 100.0. Display the names of each month along with the corresponding decimal value. Calculate and display the average of the 12 decimal values.

2.

Write a program to create a rectangular array containing a multiplication table from 1x1 up to 12x12. Output the table as 13 columns with the numeric values right aligned in columns. (The first line of output will be the column headings, the first column with no heading, then the numbers 1 to 12 for the remaining columns. The first item in each of the succeeding lines is the row heading, which ranges from 1 to 12.)

3.

Write a program that sets up a String variable containing a paragraph of text of your choice. Extract the words from the text and sort them into alphabetical order. Display the sorted list of words. You could use a simple sorting method called the bubble sort. To sort an array into ascending order the process is as follows:

Starting with the first element in the array compare successive elements (0 and 1, 1 and 2, 2 and 3, and so on).

If the first element of any pair is greater than the second, interchange the two elements.

Repeat the process for the whole array until no interchanges are necessary. The array elements will now be in ascending order.

4.

Define an array of ten String elements each containing an arbitrary string of the form "month/day/year", for example,"10/29/99" or "12/5/01". Analyze each element in the array and output the date represented in the form 29th October 1999.

5.

Write a program that will reverse the sequence of letters in each word of your chosen paragraph from Exercise 3. For instance, "To be or not to be." would become "oT ebo ro ton ot eb."
Chapter 5: Defining Classes

Overview

In this chapter we will explore the heart of the Java language - classes. Classes specify the objects you use in object-oriented programming. These form the basic building blocks of any Java program, as we saw in Chapter 1. Every program in Java involves classes, since the code for a program can only appear within a class definition.

We will now explore the details of how a class definition is put together, how to create your own classes and how to use classes to solve your own computing problems. And in the next chapter we'll extend this to look at how object-oriented programming helps us work with related classes.

By the end of this chapter you will have learned:

• What a class is, and how you define one.

• How to implement class constructors.

• How to define class methods.

• What method overloading is.

• What a recursive method is and how it works.

• How to create objects of a class.

• What packages are and how you can create and use them.
• What access attributes are and how you should use them in your class definitions.

• When you should add the finalize() method to a class.

• What native methods are.
What is a Class?

As you saw in Chapter 1, a class is a prescription for a particular kind of object - it defines a new type. We can use the class definition to create objects of that class type, that is, to create objects that incorporate all the components specified as belonging to that class.

Note

In case that's too abstract, look back to the last chapter, where we used the String class. This is a comprehensive definition for a string object, with all the operations you are likely to need built in. This makes String objects indispensable and string handling within a program easy.

The String class lies towards one end of a spectrum in terms of complexity in a class. The String class is intended to be usable in any program. It includes facilities and capabilities for operating on String objects to cover virtually all circumstances in which you are likely to use strings. In most cases your own classes won't need to be this elaborate. You will typically be defining a class to suit your particular application. A very simple class for instance, a Plane or a Person, may well represent objects that can potentially be very complicated, if that fulfils your needs. A Person object might just contain a name, address, and phone number for example if you are just implementing an address book. In another context, in a payroll program perhaps, you might need to represent a Person with a whole host of properties, such as age, marital status, length of service, job code, pay rate, and so on. It all depends on what you intend to do with objects of your class.

In essence a class definition is very simple. There are just two kinds of things that you can include in a class definition:

• **Fields**
  These are variables that store data items that typically differentiate one object of the class from another. They are also referred to as data members of a class.

• **Methods**
  These define the operations you can perform for the class - so they determine what you can do to, or with, objects of the class. Methods typically operate on the fields - the variables of the class.

The fields in a class definition can be of any of the basic types, or they can be references to objects of any class type, including the one that you are defining.

The methods in a class definition are named, self-contained blocks of code that typically operate on the variables that appear in the class definition. Note though, that this doesn't necessarily have to be the case, as you might have guessed from the main() methods we have written in all our examples up to now.
Variables in a Class Definition

An object of a class is also referred to as an instance of that class. When you create an object, the object will contain all the variables that were included in the class definition. However, the variables in a class definition are not all the same - there are two kinds.

One kind of variable in a class is associated with each object uniquely - each instance of the class will have its own copy of each of these variables, with its own value assigned. These differentiate one object from another, giving an object its individuality - the particular name, address, and telephone number in a given Person object for instance. These are referred to as instance variables.

The other kind of class variable is associated with the class, and is shared by all objects of the class. There is only one copy of each of these kinds of variables no matter how many class objects are created, and they exist even if no objects of the class have been created. This kind of variable is referred to as a class variable because the variable belongs to the class, not to any particular object, although as we have said, all objects of the class will share it. These variables are also referred to as static fields because, as we will see, you use the keyword static when you declare them.

Because this is extremely important to understand, let's summarize the two kinds of variables that you can include in your classes:

- **Instance variables**
  Each object of the class will have its own copy of each of the instance variables that appear in the class definition. Each object will have its own values for each instance variable. The name 'instance variable' originates from the fact that an object is an 'instance' or an occurrence of a class and the values stored in the instance variables for the object differentiate the object from others of the same class type. An instance variable is declared within the class definition in the usual way, with a type name and a variable name, and can have an initial value specified.

- **Class variables**
  A given class will only have one copy of each of its class variables, and these will be shared between all the objects of the class. The class variables exist even if no objects of the class have been created. They belong to the class, and they can be referenced by any object or class, not just instances of that class. If the value of a class variable is changed, the new value is available in all the objects of the class. This is quite different from instance variables where changing a value for one object does not affect the values in other objects. A class variable must be declared using the keyword static preceding the type name.

Look at the following diagram, which illustrates the difference between the two:
This shows a schematic of a class Sphere with one class variable PI, and four instance variables, radius, xCenter, yCenter, and zCenter. Each of the objects, globe and ball, will have their own variables, radius, xCenter, yCenter, and zCenter, but both will share a single copy of the class variable PI.

Why would you need two kinds of variables in a class definition? The instance variables are clearly necessary since they are the parameters that distinguish a particular object. The radius and the coordinates of the center of the sphere are fundamental to determining how big a particular Sphere object is, and where it is in space. However, although the variable PI is a fundamental parameter for a sphere - to calculate the volume for example - it would be wasteful to store a value for PI in every object, since it is always the same. Incidentally, it is also available from the standard class Math so it is somewhat superfluous in this case, but you get the general idea. So one use for class variables is to hold constant values such as / that are common to all objects of the class.

Another use for class variables is to track data values that are common to all objects of a class, and that need to be available even when no objects have been defined. For example, if you wanted to keep a count of how many objects of a class have been created in your program, you would define the variable storing the count as a class variable. It would be essential to use a class variable, because you would still want to be able to use your count variable even when no objects have been declared.

Methods in a Class Definition

The methods that you define for a class provide the actions that can be carried out using the variables specified in the class definition.

Analogous to the variables in a class definition, there are two varieties of methods - instance methods and class methods. You can execute class methods even when no objects of a class exist, whereas instance methods can only be executed in relation to a particular object, so if no objects exist, there are no instance methods to be executed. Again, like class variables, class methods are declared using the keyword static so they are sometimes referred to as static methods. We saw in the previous chapter that the valueOf() method is a static member of the String class.

Since class methods can be executed when there are no objects in existence, they cannot refer to instance variables. This is quite sensible if you think about it - trying to operate with variables that might not exist is bound to cause trouble. In fact the Java compiler won't let you try. If you reference an instance variable in the code for a class method, it won't compile - you'll just get an error message. The method main(), where execution of a Java application starts, must always be declared as static, as you have seen. The reason for this should be apparent by now. Before an application starts execution, no objects exist, so in order to start execution, you need a method that is executable even though there are no objects - a static method therefore.

The class Sphere might well have an instance method volume() to calculate the volume of a particular object. It might also have a class method objectCount() to return the current count of how many objects of type Sphere have been
created. If no objects exist, you could still call this method and get the count 0.

Note

Note that, although instance methods are specific to objects of a class, there is only ever one copy of an instance method in memory that is shared by all objects of the class, as it would be extremely expensive to replicate all the instance methods for each object. There is a special mechanism that ensures that, each time you call a method the codes executes in a manner that is specific to an object, but we will defer exploring this until a little later in this chapter.

Apart from making the method main(), perhaps the most common use for class methods is when a class is just used to contain a bunch of utility methods, rather than as a specification for objects. All executable code in Java has to be within a class, but there are lots of general-purpose functions that you need that don't necessarily have an object association - calculating a square root, for instance, or generating a random number. For example, the mathematical functions that are implemented as class methods in the standard class Math, don't relate to class objects at all - they operate on values of the basic types. You don't need objects of type Math, you just want to use the methods from time to time, and you can do this as we saw in Chapter 2. The class Math also contains some class variables containing useful mathematical constants such as \( e \) and \( \pi \).

**Accessing Variables and Methods**

You will often want to access variables and methods, defined within a class, from outside it. We will see later that it is possible to declare class members with restrictions on accessing them from outside, but let's cover the principles that apply where the members are accessible. We need to consider accessing static members and instance members separately.

You can access a static member of a class using the class name, followed by a period, followed by the member name. With a class method you will also need to supply the parentheses enclosing any arguments to the method after the method name. The period here is called the dot operator. So, if you wanted to calculate the square root of \( \pi \) you could access the class method sqrt() and the class variable PI that are defined in the Math class as follows:
Defining Classes

To define a class you use the keyword class followed by the name of the class, followed by a pair of braces enclosing the details of the definition. Let's consider a concrete example to see how this works in practice. The definition of the Sphere class we mentioned earlier could be:
Defining Methods

We have been producing versions of the method main() since Chapter 1, so you already have an idea of how a method is constructed. Nonetheless, we will go through from the beginning to make sure everything is clear.

We'll start with the fundamental concepts. A method is a self-contained block of code that has a name, and has the property that it is reusable - the same method can be executed from as many different points in a program as you require. Methods also serve to break up large and complex calculations that might involve many lines of code into more manageable chunks. A method is executed by calling its name, as we will see, and the method may or may not return a value. Methods that do not return a value are called in a statement that just does the call. Methods that do return a value are usually called from within an expression, and the value that is returned by such a method is used in the evaluation of the expression.

The basic structure of a method is shown below.

```
<return_type> methodName( arg1, arg2, ..., argn ) {
    // Executable code goes here
}
```

When you specify the return type for a method, you are defining the type for the value that will be returned by the method when you execute it. The method must always return a value of this type. To define a method that does not return a value, you specify the return type as void. Something called an access attribute can optionally precede the return type in a method definition, but we will defer looking into this until later in this chapter.

The parameters to a method appear in its definition between parentheses following the method name. These specify what information is to be passed to the method when you execute it. Your methods do not have to have parameters specified. A method that does not require any information to be passed to it when it is executed has an empty pair of parentheses after the name.

Running from a Method

To return a value from a method when its execution is complete you use a return statement, for example:
Constructors

When you create an object of a class, a special kind of method called a **constructor** is always invoked. If you don't define any constructors for your class, the compiler will supply a **default constructor** in the class that does nothing. The primary purpose of a constructor is to provide you with the means of initializing the instance variables uniquely for the object that is being created. If you are creating a Person object with the name John Doe, then you want to be able to initialize the member holding the person's name to "John Doe". This is precisely what a constructor can do. Any initialization blocks that you have defined in a class are always executed before a constructor.

A constructor has two special characteristics that differentiate it from other class methods:

- A constructor never returns a value and you must not specify a return type - not even of type void.
- A constructor always has the same name as the class.

To see a practical example we could add a constructor to our Sphere class definition:

```java
class Sphere {
```
Defining and Using a Class

To put what we know about classes to use, we can use our Sphere class in an example.

You will be creating two source files. The first is the file CreateSpheres.java, which will contain the definition of the CreateSpheres class that will have the method main() defined as a static method. As usual, this is where execution of the program starts. The second file will be the file Sphere.java that contains the definition of the class Sphere that we have been assembling.

Both files will need to be in the same directory or folder - I suggest you name the directory CreateSpheres. Then copy or move the last version of Sphere.java to this directory.

Try It Out - Using the Sphere Class

Enter the following code for the file CreateSpheres.java:
Method Overloading

Java allows you to define several methods in a class with the same name, as long as each method has a set of parameters that is unique. This is called method overloading.

The name of a method together with the type and sequence of the parameters form the signature of the method - the signature of each method in a class must be distinct to allow the compiler to determine exactly which method you are calling at any particular point.

Note that the return type has no effect on the signature of a method. You cannot differentiate between two methods just by the return type. This is because the return type is not necessarily apparent when you call a method. For example, suppose you write a statement such as:
Using Objects

Let's create an example to do some simple 2D geometry. This will give us an opportunity to use more than one class. We will define two classes, a class of point objects and a class of line objects - we then use these to find the point at which the lines intersect. We will call the example TryGeometry, so this will be the name of the directory or folder in which you should save the program files. Quite a few lines of code are involved so we will put it together piecemeal, and try to understand how each piece works as we go.

Try It Out - The Point Class

We first define a basic class for point objects:
Recursion

The methods you have seen so far have been called from within other methods, but a method can also call itself something referred to as recursion. Clearly you must include some logic in a recursive method so that it will eventually stop calling itself. We can see how this might be done with a simple example.

We can write a method that will calculate integer powers of a variable, in other words, evaluate $x^n$, or $x \times \ldots \times x$ where $x$ is multiplied by itself $n$ times. We can use the fact that we can obtain $x^n$ by multiplying $x^{n-1}$ by $x$. To put this in terms of a specific example, we can calculate $2^4$ as $2^3$ multiplied by 2, and we can get $2^3$ by multiplying $2^2$ by 2, and $2^2$ is produced by multiplying $2^1$, which is 2 of course, by 2.

Try It Out   Calculating Powers

Here is the complete program including the recursive method, power():
Understanding Packages

Important

Packages are fundamental to Java programs so make sure you understand this section.

Packages are implicit in the organization of the standard classes as well as your own programs, and they influence the names you can use for classes and the variables and methods they contain. Essentially, a package is a unique named collection of classes. The purpose of grouping classes in a package with a unique name is to make it easy to add any or all of the classes in a package into your program code. One aspect of this is that the names used for classes in one package will not interfere with the names of classes in another package, or your program, because the class names in a package are qualified by the package name.

Every class in Java is contained in a package, including all those we have defined in our examples. You haven't seen many references to package names so far because we have been implicitly using the default package to hold our classes, and this doesn't have a name.

All of the standard classes in Java are contained within packages. The package that contains most of the standard classes that we have used so far is called java.lang. You haven't seen any explicit reference to this in your code either, because this package is automatically available to your programs. Things are arranged this way because some of the classes in java.lang, such as String, are used in every program. There are other packages containing standard classes that you will need to include explicitly when you use them, as we did in the previous chapter with the StringTokenizer class.

Packaging Up Your Classes

Putting one of your classes in a package is very simple. You just add a package statement as the first statement in the source file containing the class definition. Note that it must always be the first statement. Only comments or blank lines are allowed to precede the package statement. A package statement consists of the keyword, package, followed by the package name, and is terminated by a semi-colon. If you want the classes in a package to be accessible outside the package, you must declare the class using the keyword public in the first line of your class definition. Class definitions that aren't preceded by the keyword public are only accessible from methods in classes that belong to the same package.

For example, to include the class Sphere in a package called Geometry, the contents of the file Sphere.java would need to be:
Controlling Access to Class Members

We have not yet discussed in any detail how accessible class members are outside a class. You know that from inside a static class method you can refer to any of the static members of the class, and a non-static method can refer to any member of the class. The degree to which variables and methods within one class are accessible from other classes is more complicated. It depends on what access attributes you have specified for the members of a class, and whether the classes are in the same package. This is why we had to understand packages first.

Using Access Attributes

Let's start by considering classes in the same package. Within a given package, any class has direct access to any other class name for declaring variables or specifying method parameter types, for example but the variables and methods that are members of that other class are not necessarily accessible. The accessibility of these is controlled by access attributes. You have four possibilities when specifying an access attribute for a class member, including what we have used in our examples so far that is, not to specify anything at all and each possibility has a different effect overall. The options you have for specifying the accessibility of a variable or a method in a class are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Permitted access</th>
</tr>
</thead>
<tbody>
<tr>
<td>No access attribute</td>
<td>From methods in any class in the same package.</td>
</tr>
<tr>
<td>public</td>
<td>From methods in any class anywhere.</td>
</tr>
<tr>
<td>private</td>
<td>Only accessible from methods inside the class. No access from outside the class at all.</td>
</tr>
<tr>
<td>protected</td>
<td>From methods in any class in the same package and from any sub-class anywhere.</td>
</tr>
</tbody>
</table>

The table shows you how the access attributes you set for a class member determine the parts of the Java environment from which you can access it. We will discuss sub-classes in the next chapter, so don't worry about these for the moment. We will be coming back to how and when you use the protected attribute then. Note that public, private, and protected are all keywords. Specifying a member as public makes it completely accessible, and at the other extreme, making it private restricts access to members of the same class.

This may sound more complicated than it actually is. Look at the next diagram, which shows the access allowed between classes within the same package.
Within a package such as package1, only the private members of the class Class1 can't be directly accessed by a method in another class in the same package. Declaring a class member to be private limits its availability solely to methods in the same class.

We saw earlier that a class definition must have an access attribute of public if it is to be accessible from outside the package. The next diagram shows the situation where the classes, seeking access to the members of a public class, are in different packages.

Here access is more restricted. The only members of Class1 that can be accessed from an ordinary class, Class2, in another package are those specified as public. Keep in mind that the class, Class1, must also have been defined with the attribute public. From a sub-class of Class1 that is in another package, the members of Class1, without an access attribute, cannot be reached, and neither can the private members—these can never be accessed externally under any circumstances.

**Specifying Access Attributes**

As you probably gathered from the diagrams that we just looked at, to specify an access attribute for a class member, you just add the keyword to the beginning of the declaration. Here is the Point class you saw earlier, but now with access attributes defined for its members:

**Try It Out  Accessing the Point Class**

Make the following changes to your Point class. If you save it in a new directory, do make sure Line.java is copied there as well. It will be useful later if they are in a directory with the name Geometry.
Nested Classes

All the classes we have defined so far have been separate from each other, each stored away in its own file. Not all classes have to be defined like this. You can put the definition of one class inside the definition of another class. The inside class is called a **nested class**. A nested class can itself have another class nested inside it, if need be.

When you define a nested class, it is a member of the enclosing class in much the same way as other members. A nested class can have an access attribute just like other class members, and the accessibility from outside the enclosing class is determined by the attributes in the same way.
The finalize() Method

You have the option of including a method finalize() in a class definition. This method is called automatically by Java before an object is finally destroyed and the space it occupies in memory is released. Please note that this may be some time after the object is inaccessible in your program. When an object goes out of scope, it is dead as far as your program is concerned, but the Java Virtual Machine may not get around to disposing of the remains until later. When it does, it calls the finalize() method for the object. The form of the finalize() method is:
Native Methods

It is possible to include a method in a class that is implemented in some other programming language, such as C or C++, external to the Java Virtual Machine. To specify such a method within a class definition you use the keyword native in the declaration of the method. For example:
Summary

In this chapter you have learned all the essentials of defining your own classes. You can now create your own class types to fit the context of the problems you are dealing with. We will build on this in the next chapter to enable you to add more flexibility to the operations on your class objects by showing you how to realize polymorphism.

The important points covered in this chapter are:

- A class definition specifies the variables and methods that are members of the class.

- Each class must be saved in a file with the same name as the class, and with the extension .java.

- Class variables are declared using the keyword static, and one instance of each class variable is shared amongst all objects of a class.

- Each object of a class will have its own instance variables—these are variables declared without using the keyword static.

- Methods that are specified as static can be called even if no class objects exist, but a static method cannot refer to instance variables.

- Methods that are not specified as static can access any of the variables in the class directly.

- Recursive methods are methods that call themselves.

- Access to members of a class is determined by the access attributes that are specified for each of them. These can be public, private, protected, package private, or nothing at all.

- Classes can be grouped into a package. If a class in a package is to be accessible from outside the package the class must be declared using the keyword public.
To designate that a class is a member of a package you use a package statement at the beginning of the file containing the class definition.

•

To add classes from a package to a file you use an import statement immediately following any package statement in the file.

•

A native method is a method implemented in a language other than Java. Java programs containing native methods cannot be applets and are no longer portable.

•

A field does not get ploughed by turning it over in your mind.
Exercises

1. Define a class for rectangle objects defined by two points, the top-left and bottom-right corners of the rectangle. Include a constructor to copy a rectangle, a method to return a rectangle object, that encloses the current object and the rectangle passed as an argument, and a method to display the defining points of a rectangle. Test the class by creating four rectangles, and combining these cumulatively, to end up with a rectangle enclosing them all. Output the defining points of all the rectangles you create.

2. Define a class, mcmLength, to represent a length measured in meters, centimeters, and millimeters, each stored as integers. Include methods to add and subtract objects, to multiply and divide an object by an integer value, to calculate an area resulting from the product of two objects, and to compare objects. Include constructors that accept: three arguments – meters, centimeters, and millimeters; one integer argument in millimeters; one double argument in centimeters and no arguments, which creates an object with the length set to zero. Check the class by creating some objects and testing the class operations.

3. Define a class, tkgWeight, to represent a weight in tons, kilograms, and grams, and include a similar range of methods and constructors as the previous example. Demonstrate this class by creating and combining some class objects.

4. Put both the previous classes in a package called Measures. Import this package into a program that will calculate and display the total weight of the following: 200 carpets  size: 4 meters by 2 meters 9 centimeters, that weigh 1.25 kilograms per square meter; and 60 carpets  size: 3 meters 57 centimeters by 5 meters, that weigh 1.05 kilograms per square meter.
Chapter 6: Extending Classes and Inheritance

Overview

A very important part of object-oriented programming allows you to create a new class based on a class that has already been defined. The class that you use as the base for your new class can be either one you have defined, a standard class in Java, or a class defined by someone else—perhaps from a package supporting a specialized application area.

This chapter focuses on how you can reuse existing classes by creating new classes based on the ones you have, and explores the ramifications of using this facility, and the additional capabilities it provides. We will also delve into an important related topic—interfaces—and how you can use them.

In this chapter you will learn:

- How to reuse classes by defining a new class based on an existing class.
- What polymorphism is and how to define your classes to take advantage of it.
- What an abstract method is.
- What an abstract class is.
- What an interface is and how you can define your own interfaces.
- How to use interfaces in your classes.
How interfaces can help you implement polymorphic classes.
Using Existing Classes

Let's start by understanding the jargon. Defining a new class based on an existing class is called derivation. The new class, or derived class, is referred to as a direct subclass of the class from which it is derived. The original class is called a base class because it forms the base for the definition of the derived class. The original class is also referred to as a superclass of the derived class. You can also derive a new class from a derived class, which in turn was derived from some other derived class, and so on. This is illustrated in the following diagram:

![Diagram showing class hierarchy]

This shows just three classes in a hierarchy, but there can be as many as you like.

Let's consider a more concrete example. We could define a class Dog that could represent a dog of any kind.
Class Inheritance

In summary, when you derive a new class from a base class, the process is additive in terms of what makes up a class definition. The additional members that you define in the new class establish what makes a derived class object different from a base class object. Any members that you define in the new class are in addition to those that are already members of the base class. For our Spaniel class, derived from Dog, the data members to hold the name and the breed, that are defined for the class Dog, would automatically be in the class Spaniel. A Spaniel object will always have a complete Dog object inside it with all its data members and methods. This does not mean that all the members defined in the Dog class are available to methods that are specific to the Spaniel class. Some are and some aren't. The inclusion of members of a base class in a derived class so that they are accessible in that derived class is called class inheritance. An inherited member of a base class is one that is accessible within the derived class. If a base class member is not accessible in a derived class, then it is not an inherited member of the derived class, but base class members that are not inherited still form part of a derived class object.

An inherited member of a derived class is a full member of that class and is freely accessible to any method in the class. Objects of the derived class type will contain all the inherited members of the base class both fields and methods, as well as the members that are specific to the derived class. Note that a derived class object always contains a complete base class object within it, including all the fields and methods that are not inherited. We need to take a closer look at how inheritance works, and how the access attribute of a base class member affects its visibility in a derived class.

We need to consider several aspects of defining and using a derived class. First of all we need to know which members of the base class are inherited in the derived class. We will look at what this implies for data members and methods separately there are some subtleties here we should be quite clear on. We will also look at what happens when you create an object of the derived class. There are some wrinkles in this context that require closer consideration. Let's start by looking at the data members that are inherited from a base class.

Inheriting Data Members

The next diagram shows which access attributes permit a class member to be inherited in a subclass. It shows what happens when the subclass is defined in either the same package or a different package from that containing the base class. Remember that inheritance implies accessibility of the member in a derived class, not just presence.
Important

Note that a class itself can be specified as `public`. This makes the class accessible from any package anywhere. A class that is not declared as `public` can only be accessed from classes within the same package. This means for instance that you cannot define objects of a non-`public` class type within classes in other packages. It also means that to derive a new class in a package that does not contain the base class, the base class must be declared as `public`. If the base class is not declared as `public`, it cannot be reached directly from outside the package.

As you can see, a subclass that you define in the same package as its base inherits everything except for private data members of the base. If you define a subclass outside the package containing the base class, the private data members are not inherited, and neither are any data members in the base class that you have declared without access attributes. Members defined as private in the base class are never inherited under any circumstances.

You should also be able to see where the explicit access specifiers now sit in relation to one another. The public specifier is the least restrictive on class members since a public member is available everywhere, protected comes next, and prevents access from classes outside of a package, but does not limit inheritance provided the class itself is public. Putting no access specifier on a class member limits access to classes within the same package, and prevents inheritance in subclasses that are defined in a different package. The most restrictive is private since access is constrained to the same class.

The inheritance rules apply to class variables variables that you have declared as static as well as instance variables. You will recall that only one occurrence of each static variable exists, and is shared by all objects of the class, whereas each object has its own set of instance variables. So for example, a variable that you declare as private and static in the base class is not inherited in a derived class, whereas a variable that you declare as protected and static will be inherited, and will be shared between all objects of a derived class type, as well as objects of the base class type.

**Hiding Data Members**

You can define a data member in a derived class with the same name as a data member in the base class. This is not a recommended approach to class design generally, but it is possible that it can arise unintentionally. When it occurs, the base class data member may still be inherited, but will be hidden by the derived class member with the same name. The hiding mechanism applies regardless of whether the respective types or access attributes are the same or not the base class member will be hidden in the derived class if the names are the same.

Any use of the derived class member name will always refer to the member defined as part of the derived class. To refer to the inherited base class member, you must qualify it with the keyword `super` to indicate it is the member of the superclass that you want. Suppose you have a data member, `value`, as a member of the base class, and a data member with the same name in the derived class. In the derived class, the name `value` references the derived class member, and the name `super.value` refers to the member inherited from the base class. Note that you cannot use `super.super.something` to refer to a member name hidden in the base class of a base class.

In most situations you won't need to refer to inherited data members in this way as you would not deliberately set out to use duplicate names. The situation can commonly arise if you are using a class as a base that is subsequently modified by adding data members it could be a Java library class for instance, or some other class in a package designed and maintained by someone else. Since your code did not presume the existence of the base class member,
with the same name as your derived class data member, hiding the inherited member is precisely what you want. It allows the base class to be altered without breaking your code.

**Inherited Methods**

Ordinary methods in a base class, by which I mean methods that are not constructors, are inherited in a derived class in the same way as the data members of the base class. Those methods declared as private in a base class are not inherited, and those that you declare without an access attribute are only inherited if you define the derived class in the same package as the base class. The rest are all inherited.

Constructors are different from ordinary methods. Constructors in the base class are never inherited, regardless of their attributes. We can look into the intricacies of constructors in a class hierarchy by considering how derived class objects are created.

**Objects of a Derived Class**

We said at the beginning of this chapter that a derived class extends a base class. This is not just jargon—it really does do this. As we have said several times, inheritance is about what members of the base class are **accessible** in a derived class, not what members of the base class **exist** in a derived class object. An object of a subclass will contain **all** the members of the original base class, plus any new members defined in the derived class (see following diagram).

![Diagram showing inheritance and visibility of members in a class hierarchy]

The base members are all there in a derived class object—you just can't access some of them in the methods that you have defined for the derived class. The fact that you can't access some of the base class members does not mean that they are just excess baggage—they are essential members of your derived class objects. A Spaniel object needs all the Dog attributes that make it a Dog object, even though some of these may not be accessible to the Spaniel methods. Of course, the base class methods that are inherited in a derived class can access all the base class members, including those that are not inherited.

Though the base class constructors are not inherited in your derived class, you can still call them to initialize the base class members. More than that, if you don't call a base class constructor from your derived class constructor, the compiler will try to arrange to do it for you. The reasoning behind this is that since a derived class object has a base class object inside it, a good way to initialize the base part of a derived class object is using a base class constructor.

To understand this better, let's take a look at how it works in practice.

**Deriving a Class**
Let's take a simple example. Suppose we have defined a class to represent an animal as follows:
Choosing Base Class Access Attributes

You now know the options available to you in defining the access attributes for classes you expect to use to define subclasses. You know what effect the attributes have on class inheritance, but how do you decide which you should use?

There are no hard and fast rules what you choose will depend on what you want to do with your classes in the future, but there are some guidelines you should consider. They follow from basic object oriented principles:

• The methods that make up the external interface to a class should be declared as public. As long as there are no overriding methods defined in a derived class, public base class methods will be inherited and fully available as part of the external interface to the derived class. You should not normally make data members public unless they are constants intended for general use.

• If you expect other people will use your classes as base classes, your classes will be more secure if you keep data members private, and provide public methods for accessing and manipulating them. In this way you control how a derived class object can affect the base class data members.

• Making base class members protected allows them to be accessed from other classes in the same package, but prevents direct access from a class in another package. Base class members that are protected are inherited in a subclass and can, therefore, be used in the implementation of a derived class. You can use the protected option when you have a package of classes in which you want uninhibited access to the data members of any class within the same package, because they operate in a closely coupled way, for instance, but you want free access to be limited to subclasses in other packages.

• Omitting the access attribute for a class member makes it directly available to other classes in the same package, while preventing it from being inherited in a subclass that is not in the same package it is effectively private when viewed from another package.
Polymorphism

Class inheritance is not just about reusing classes that you have already defined as a basis for defining a new class. It also adds enormous flexibility to the way in which you can program your applications, with a mechanism called polymorphism. So what is polymorphism?

The word 'polymorphism' generally means the ability to assume several different forms or shapes. In programming terms it means the ability of a single variable of a given type to be used to reference objects of different types, and to automatically call the method that is specific to the type of object the variable references. This enables a single method call to behave differently, depending on the type of the object to which the call applies.

There are a few requirements that need to be fulfilled to get polymorphic behavior, so let's step through them.

First of all, polymorphism works with derived class objects. It also depends on a new capability that is possible within a class hierarchy that we haven't met before. Up to now, we have always been using a variable of a given type to reference objects of the same type. Derived classes introduce some new flexibility in this. Of course, we can store a reference to a derived class object in a variable of the derived class type, but we can also store it in a variable of any direct or indirect base class type. More than that, a reference to a derived class object must be stored in a variable of a direct or indirect class type for polymorphism to work. For instance, as shown in the previous diagram, a variable of type Dog can be used to store a reference to an object of any type derived from Dog. If the Dog class was derived from the Animal class here, a variable of type Animal could also be used to reference Spaniel, Chihuahua, or Collie objects.

Polymorphism means that the actual type of the object involved in a method call determines which method is called, rather than the type of the variable being used to store the reference to the object. In the previous diagram, if aDog contains a reference to a Spaniel object, the bark() method for that object will be called. If it contains a reference to a Collie object, the bark() method in the Collie class will be called. To get polymorphic operation when calling a method, the method must be a member of the base class - the class type of the variable you are using as well that of the class type of the object involved. So in our example, the Dog class must contain a bark() method, as must each of the derived classes. You cannot call a method for a derived class object using a variable of a base class type if the method is not a member of the base class. Any definition of the method in a derived class must have the same signature and the same return type as in the base class, and must have an access specifier that is not more restrictive. Indeed, if you define a method in a derived class with the same signature as a method in a base class, any attempt to
specify a different return type, or a more restrictive access specifier, will be flagged as an error by the compiler.

The conditions that need to be met if you want to use polymorphism can be summarized as:

- The method call for a derived class object must be through a variable of a base class type.
- The method called must also be a member of the base class.
- The method signature must be the same in the base and derived classes.
- The method return type must be the same in the base and derived classes.
- The method access specifier must be no more restrictive in the derived class than in the base.

When you call a method using a variable of a base class type, polymorphism results in the method that is called being selected based on the type of the object stored, not the type of the variable. Because a variable of a base type can store a reference to an object of any derived type, the kind of object stored will not be known until the program executes. Thus the choice of which method to execute has to be made dynamically when the program is running—it cannot be determined when the program is compiled. The bark() method that is called through the variable of type Dog in the earlier illustration, may do different things depending on what kind of object the variable references. As we will see, this introduces a whole new level of capability in programming using objects. It implies that your programs can adapt at runtime to accommodate and process different kinds of data quite automatically.

Note that polymorphism only applies to methods. It does not apply to data members. When you access a data member of a class object, the variable type always determines the class to which the data member belongs. This implies that a variable of type Dog can only be used to access data members of the Dog class. Even when it references an object of type Spaniel, for instance, you can only use it to access data members of the Dog part of a Spaniel object.

**Using Polymorphism**

As we have seen, polymorphism relies on the fact that you can assign an object of a subclass type to a variable that you have declared as being of a superclass type. Suppose you declare the variable:
Multiple Levels of Inheritance

As we indicated at the beginning of the chapter, there is nothing to prevent a derived class being used as a base class. For example, we could derive a class Spaniel from the class Dog without any problem:

Try It Out  A Spaniel Class

Start the Spaniel class off with this minimal code:
Abstract Classes

In the class Animal, we introduced a version of the method sound() that did nothing because we wanted to call the sound() method in the subclass objects dynamically. The method sound() has no meaning in the context of the generic class Animal, so implementing it does not make much sense. This situation often arises in object-oriented programming. You will often find yourself creating a superclass from which you will derive a number of subclasses, just to take advantage of polymorphism.

To cater for this, Java has abstract classes. An abstract class is a class in which one or more methods are declared, but not defined. The bodies of these methods are omitted, because, as in the case of the method sound() in our class Animal, implementing the methods does not make sense. Since they have no definition and cannot be executed, they are called abstract methods. The declaration for an abstract method ends with a semi-colon and you specify the method with the keyword abstract to identify it as such. To define an abstract class you use the keyword abstract in front of the class name.

We could have defined the class Animal as an abstract class by amending it as follows:
The Universal Superclass

I must now reveal something I have been keeping from you. All the classes that you define are subclasses by default whether you like it or not. All your classes have a standard class, Object, as a base, so Object is a superclass of every class. You never need to specify the class Object as a base in the definition of your classes—it happens automatically.

There are some interesting consequences of having Object as a universal superclass. For one thing, a variable of type Object can hold an object of any class type. This is useful when you want to write a method that needs to handle objects of unknown type. You can use a variable of type Object as a parameter to a method, to receive an object, and then include code in the method that figures out what kind of object it actually is (we will see something of the tools that will enable you to do this a little later in this chapter).

Of course, your classes will inherit members from the class Object. These all happen to be methods, of which seven are public, and two are protected. The seven public methods are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>toString()</td>
<td>This method returns a String object that describes the current object. In the inherited version of the method, this will be the name of the class, followed by '@' and the hexadecimal representation for the object. This method is called automatically when you concatenate objects with String variables using +. You can override this method in your classes to return your own String object for your class.</td>
</tr>
<tr>
<td>equals()</td>
<td>This compares the object passed as an argument with the current object, and returns true if they are the same object (not just equal—they must be one and the same object). It returns false if they are different objects, even if the objects have identical values for their data members.</td>
</tr>
<tr>
<td>getClass()</td>
<td>This method returns an object of type Class that identifies the class of the current object. We will see a little more about this later in this chapter.</td>
</tr>
<tr>
<td>hashCode()</td>
<td>This method calculates a hash code value for an object and returns it as type int. Hash code values are used in classes defined in the package java.util for storing objects in hash tables. We will see more about this in Chapter 12.</td>
</tr>
</tbody>
</table>
notify()  This is used to wake up a thread associated with the current object. We will discuss in Chapter 14 how threads work.

notifyAll()  This is used to wake up all threads associated with the current object. We will also discuss this in Chapter 14.

wait()  This method causes a thread to wait for a change in the current object. We will discuss this method in Chapter 14.

Note that getClass(), notify(), notifyAll(), and wait() cannot be overridden in your own class definitions they are 'fixed' with the keyword final in the class definition for Object (see the section on the final modifier later in this chapter).

It should be clear now why we could get polymorphic behavior with toString() in our derived classes when our base class did not define the method. There is always a toString() method in all your classes that is inherited from Object.

The two protected methods your classes inherit from Object are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone()</td>
<td>This will create an object that is a copy of the current object regardless of type. This can be of any type as an Object variable can refer to an object of any class. Note that this does not work with all class objects and does not always do precisely what you want, as we will see later in this section.</td>
</tr>
<tr>
<td>finalize()</td>
<td>This is the method that is called to clean up as an object is destroyed. As you have seen in the last chapter you can override this to add your own clean-up code.</td>
</tr>
</tbody>
</table>

Since all your classes will inherit the methods defined in the Object class we should look at them in a little more detail.

The toString() Method

We have already made extensive use of the toString() method and you know that it is used by the compiler to obtain a String representation of an object when necessary. It is obvious now why we must always declare the toString() method as public in a class. It is declared as such in the Object class and you can't declare it as anything else.

You can see what the toString() method, that is inherited from class Object, will output for an object of one of your classes by commenting out the toString() method in Animal class in the previous example. A typical sample of the output for an object is:
Casting Objects

You can cast an object to another class type, but only if the current object type and the new class type are in the same hierarchy of derived classes, and one is a superclass of the other. For example, earlier in this chapter we defined the classes Animal, Dog, Spaniel, Cat and Duck, and these classes are related in the hierarchy shown below:

![Class Hierarchy Diagram]

You can cast an object of a class upwards through its direct and indirect superclasses. For example, you could cast an object of type Spaniel directly to type Dog, type Animal or type Object. You could write:
Designing Classes

A basic problem in object-oriented programming is deciding how the classes in your program should relate to one another. One possibility is to create a hierarchy of classes by deriving classes from a base class that you have defined, and adding methods and data members to specialize the subclasses. Our Animal class and the subclasses derived from it are an example of this. Another possibility is to define a set of classes which are not hierarchical, but which have data members that are themselves class objects. A Zoo class might well have objects of types derived from Animal as members, for instance. You can have class hierarchies that contain data members that are class objects – we already have this with our classes derived from Animal since they have members of type String. The examples so far have been relatively clear-cut as to which approach to choose, but it is not always so evident. Quite often you will have a choice between defining your classes as a hierarchy, and defining classes that have members that are class objects. Which is the best approach to take?

Like all questions of this kind, there are no clear-cut answers. If object-oriented programming was a process that we could specify by a fixed set of rules that you could just follow blindly, we could get the computer to do it. There are some guidelines though, and some contexts in which the answer may be more obvious.

Aside from the desirability of reflecting real-world relationships between types of objects, the need to use polymorphism is a primary reason for using subclasses (or interfaces as we shall see shortly). This is the essence of object-oriented programming. Having a range of related objects that can be treated equivalently can greatly simplify your programs. You have seen how having various kinds of animals specified by classes derived from a common base class Animal allows us to act on different types of animal as though they are the same, producing different results depending on what kind of animal is being dealt with, and all this automatically.

A Classy Example

Many situations involve making judgments about the design of your classes. The way to go may well boil down to a question of personal preference. Let's try to see how the options look in practice by considering a simple example. Suppose we want to define a class PolyLine to represent lines consisting of one or more connected segments, as illustrated in the diagram:

This shows two polylines, one defined by four points, the other defined by seven points.
It seems reasonable to represent points as objects of a class Point. Points are well-defined objects that will occur in the context of all kinds of geometric entities. We have seen a class for points earlier that we put in the package Geometry. Rather than repeat the whole thing, we will define the bare bones we need in this context:
Using the final Modifier

We have already used the keyword final to fix the value of a static data member of a class. You can also apply this keyword to the definition of a method, and to the definition of a class.

It may be that you want to prevent a subclass from overriding a method in your class. When this is the case, simply declare that method as final. Any attempt to override a final method in a subclass will result in the compiler flagging the new method as an error. For example, you could declare the method addPoint() as final within the class, PolyLine, by writing its definition in the class as:
Interfaces

In the classes that we derived from the class Animal, we had a common method, sound(), that was implemented individually in each of the subclasses. The method signature was the same in each class, and the method could be called polymorphically. The main point to defining the class Animal first, and then subsequently the classes Dog, and Cat, and so on from it, was to be able to get polymorphic behavior. When all you want is a set of one or more methods to be implemented in a number of different classes so that you can call them polymorphically, you can dispense with the base class altogether.

You can achieve the same end result much more simply by using a Java facility called an interface. The name indicates its primary use - specifying a set of methods that represent a particular class interface, which can then be implemented appropriately in a number of different classes. All of the classes will then share this common interface, and the methods in it can be called polymorphically. This is just one aspect of what you can do using an interface. We will start by examining what an interface is from the ground up, and then look at what we can do with it.

An interface is essentially a collection of related constants and/or abstract methods, and in most cases it will contain just methods. An interface doesn’t define what a method does. It just defines its form - its name, its parameters, and its return type.

To make use of an interface, you implement the interface in a class - that is, you declare that the class implements the interface and you write the code for each of the methods declared in the interface as part of the class definition. When a class implements an interface, any constants that were defined in the interface definition are available directly in the class, just as though they were inherited from a base class. An interface can contain either constants, or abstract methods, or both.

The methods in an interface are always public and abstract, so you do not need to specify them as such - it is considered to be bad programming practice to specify any attributes for them and you definitely cannot add any attributes other than the defaults, public and abstract. This implies that methods declared in an interface can never be static so an interface always declares instance methods. The constants in an interface are always public, static and final, so you do not need to specify the attributes for these either.

An interface is defined just like a class, but using the keyword interface rather than the keyword class. You store an interface definition in a .java file with the same name as the interface. The name that you give to an interface must be different from that of any other interface or class in the same package. Just as for classes, the members of the interface - the constants and/or method declarations - appear between braces. Let’s start by looking at an interface that just defines constants.
Interfaces Containing Constants

Interfaces provide a very useful and easy to use mechanism for defining a set of constants and sharing them across several classes. Here is an interface containing only constants:
Anonymous Classes

There are occasions where you need to define a class for which you will only ever want to define one object in your program, and the only use for the object is to pass it directly as an argument to a method. In this case, as long as your class extends an existing class, or implements an interface, you have the option of defining the class as an anonymous class. The definition for an anonymous class appears in the new expression, in the statement where you create and use the object of the class, so that there is no necessity to provide a name for the class.

We will illustrate how this is done using an example. Supposing we want to define an object of a class that implements the interface ActionListener for one time use. We could do this as follows:
Summary

You should now understand polymorphism, and how to apply it. You will find that this technique can be utilized to considerable advantage in the majority of your Java programs. It will certainly appear in many of the examples in the remaining chapters.

The important points we have covered in this chapter are:

- An **abstract method** is a method that has no body defined for it, and is declared using the keyword `abstract`.

- An **abstract class** is a class that contains one or more abstract methods. It must be defined with the attribute `abstract`.

- You can define one class based on another. This is called class derivation or inheritance. The base class is called a **superclass** and the derived class is called a **subclass**. A superclass can also be a subclass of another superclass.

- A subclass inherits certain members of its superclass. An inherited member of a class can be referenced and used as though it was declared as a normal member of the class.

- A subclass does not inherit the superclass constructors.

- The private members of a superclass are not inherited in a subclass. If the subclass is not in the same package as the superclass, then members of the superclass that do not have an access attribute are not inherited.

- The first statement in the body of a constructor for a subclass should call a constructor for the superclass. If it does not, the compiler will insert a call for the default constructor for the superclass.

- A subclass can re-implement, or overload, the methods inherited from its superclass. If two or more subclasses, with a common base class, re-implement a common set of methods, these methods can be selected for execution at run-time.
• A variable of a superclass can point to an object of any of its subclasses. Such a variable can then be used to execute the subclass methods inherited from the superclass.

• A subclass of an abstract class must also be declared as abstract if it does not provide definitions for all of the abstract methods inherited from its superclass.

• A class defined inside another class is called a nested class or inner class. An inner class may itself contain inner classes.

• An interface can contain constants, abstract methods, and inner classes.

• A class can implement one or more interfaces by declaring them in the class definition, and including the code to implement each of the interface methods.

• A class that does not define all the methods for an interface it implements must be declared as abstract.

• If several classes implement a common interface, the methods declared as members of the interface can be executed polymorphically.
Exercises

1. Define an abstract base class Shape that includes protected data members for the (x, y) position of a shape, a public method to move a shape, and a public abstract method show() to output a shape. Derive subclasses for lines, circles and rectangles. Also define the class PolyLine that you saw in this chapter with Shape as its base class. You can represent: a line as two points, a circle as a center and a radius, and a rectangle as two points on diagonally opposite corners. Implement the toString() method for each class. Test the classes by selecting ten random objects of the derived classes, then invoking the show() method for each. Use the toString() methods in the derived classes.

2. Define a class, ShapeList, that can store an arbitrary collection of any objects of subclasses of the Shape class.

3. Implement the classes for shapes using an interface for the common methods, rather than inheritance from the superclass, while still keeping Shape as a base class.

4. Extend the LinkedList class that we defined in this chapter so that it supports traversing the list backwards as well as forwards.

5. Add methods to the class LinkedList to insert and delete elements at the current position.

6. Implement a method in the LinkedList class to insert an object following an object passed as an argument. (Assume the objects stored in the list implement an equals() method that compares the This object with an object passed as an argument, and returns true if they are equal.)
Chapter 7: Exceptions

Overview

Java uses exceptions as a way of signaling serious problems when you execute a program. The standard classes use them extensively. Since they arise in your Java programs when things go wrong, and if something can go wrong in your code, sooner or later it will, they are a very basic consideration when you are designing and writing your programs.

The reason we've been sidestepping the question of exceptions for the past six chapters is that you first needed to understand classes and inheritance before you could understand what an exception is, and appreciate what happens when an exception occurs. Now that you have a good grasp of these topics we can delve into how to use and deal with exceptions in a program.

In this chapter you will learn:

• What an exception is

• How you handle exceptions in your programs

• The standard exceptions in Java

• How to guarantee that a particular block of code in a method will always be executed

• How to define and use your own types of exceptions

• How to throw exceptions in your programs
The Idea Behind Exceptions

An exception usually signals an error, and is so-called because errors in your Java programs are bound to be the exception rather than the rule - by definition! An exception doesn't always indicate an error though - it can also signal some particularly unusual event in your program that deserves special attention.

If, in the midst of the code that deals with the normal operation of the program, you try to deal with the myriad, and often highly unusual, error conditions that might arise, your program structure will soon become very complicated and difficult to understand. One major benefit of having an error signaled by an exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly. Another positive aspect of exceptions is that they provide a way of enforcing a response to particular errors - with many kinds of exceptions, you must include code in your program to deal with them, otherwise your code will not compile.

One important idea to grasp is that not all errors in your programs need to be signaled by exceptions. Exceptions should be reserved for the unusual or catastrophic situations that can arise. A user entering incorrect input to your program for instance is a normal event, and should be handled without recourse to exceptions. The reason for this is that dealing with exceptions involves quite a lot of processing overhead, so if your program is handling exceptions a lot of the time it will be a lot slower than it needs to be.

An exception in Java is an object that's created when an abnormal situation arises in your program. This exception object has data members that store information about the nature of the problem. The exception is said to be thrown, that is, the object identifying the exceptional circumstance is tossed, as an argument, to a specific piece of program code that has been written specifically to deal with that kind of problem. The code receiving the exception object as a parameter is said to catch it.

The situations that cause exceptions are quite diverse, but they fall into four broad categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code or Data Errors</td>
<td>For example, you attempt an invalid cast of an object, you try to use an array index that's outside the limits for the array, or an integer arithmetic expression that has a zero divisor.</td>
</tr>
<tr>
<td>Standard Method Exceptions</td>
<td>For example, if you use the substring() method in the String class, it can throw a StringIndexOutOfBoundsException exception.</td>
</tr>
<tr>
<td>Throwing your own Exceptions</td>
<td>We'll see later in this chapter how you can throw a few of your own when you need to.</td>
</tr>
<tr>
<td>Java Errors</td>
<td>These can be due to errors in executing the Java Virtual Machine which runs your compiled program, but usually arise as a consequence of an error in your program.</td>
</tr>
</tbody>
</table>

Before we look at how you make provision in your programs for dealing with exceptions, we should understand what specific classes of exceptions could arise.
Types of Exceptions

An exception is always an object of some subclass of the standard class Throwable. This is true for exceptions that you define and throw yourself, as well as the standard exceptions that arise due to errors in your code. It’s also true for exceptions that are thrown by methods in one or other of the standard packages.

Two direct subclasses of the class Throwable - the class Error and the class Exception - cover all the standard exceptions. Both these classes themselves have subclasses which identify specific exception conditions.

Error Exceptions

The exceptions that are defined by the class Error, and its subclasses, are characterized by the fact that they all represent conditions that you aren't expected to do anything about and, therefore, you aren't expected to catch them. There are three direct subclasses of Error - ThreadDeath, LinkageError, and VirtualMachineError. The first of these sounds the most serious, but in fact it isn't. A ThreadDeath exception is thrown whenever an executing thread is deliberately stopped, and in order for the thread to be destroyed properly you should not catch this exception. There are circumstances where you might want to - for clean-up operations for instance - in which case you must be sure to rethrow the exception to allow the thread to die. When a ThreadDeath exception is thrown and not caught, it's the thread that ends, not the program. We will deal with threads in detail in Chapter 11.

The LinkageError exception class has subclasses that record serious errors with the classes in your program. Incompatibilities between classes or attempting to create an object of a non-existent class type are the sorts of things that cause these exceptions to be thrown. The VirtualMachineError class has four subclasses that specify exceptions that will be thrown when a catastrophic failure of the Java Virtual Machine occurs. You aren't prohibited from trying to deal with these exceptions but, in general, there's little point in attempting to catch them. The exceptions that correspond to objects of classes derived from LinkageError and VirtualMachineError are all the result of catastrophic events or conditions. There is little or nothing you can do to recover from them during the execution of the program. In these sorts of situations, all you can usually do is read the error message generated by the exception, and then, particularly in the case of a LinkageError exception, try to figure out what might be wrong with your code to cause the exception to be thrown.

RuntimeException Exceptions
For almost all the exceptions that are represented by subclasses of the Exception class, you must include code in your programs to deal with them if your code may cause them to be thrown. If a method in your program has the potential to generate an exception of some such class, you must either handle the exception within the method, or register that your method may throw such an exception. If you don't, your program will not compile. We'll see in a moment how to handle exceptions and how to specify that a method can throw an exception.

One group of subclasses of Exception that are exempted from this are those derived from RuntimeException. The reason that RuntimeException exceptions are treated differently, and that the compiler allows you to ignore them, is that they generally arise because of serious errors in your code. In most situations there is little you can do to recover the situation. However, in some contexts for some of these exceptions, this is not always the case, and you may well want to include code to recognize them. There are quite a lot of subclasses of RuntimeException that are used to signal problems in various packages in the Java class library. Let's look at the exception classes that have RuntimeException as a base that are defined in the java.lang package.

The subclasses of RuntimeException defined in the standard package java.lang are:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Exception Condition Represented</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>An invalid arithmetic condition has arisen such as an attempt to divide an integer value by zero.</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>You've attempted to use an index that is outside the bounds of the object it is applied to. This may be an array, a String object, or a Vector object. The class Vector is defined in the standard package, java.util. We will be looking into the Vector class in Chapter 10.</td>
</tr>
<tr>
<td>NegativeArraySizeException</td>
<td>You tried to define an array with a negative dimension.</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>You used an object variable containing null, when it should refer to an object for proper operation - for example, calling a method or accessing a data member.</td>
</tr>
<tr>
<td>ArrayStoreException</td>
<td>You've attempted to store an object in an array that isn't permitted for the array type.</td>
</tr>
<tr>
<td>ClassCastException</td>
<td>You've tried to cast an object to an invalid type - the object isn't of the class specified, nor is it a subclass, or a superclass, of the class specified.</td>
</tr>
<tr>
<td>Exception</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>You've passed an argument to a method which doesn't correspond with the parameter type.</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Your program has performed an illegal operation that is a security violation. This might be trying to read a file on the local machine from an applet.</td>
</tr>
<tr>
<td>IllegalMonitor</td>
<td>A thread has tried to wait on the monitor for an object that the thread doesn't own. (We'll look into threads in Chapter 11).</td>
</tr>
<tr>
<td>ckeditor_0</td>
<td>You tried to call a method at a time when it was not legal to do so.</td>
</tr>
<tr>
<td>Unsupported OperationException</td>
<td>Thrown if you request an operation to be carried out that is not supported.</td>
</tr>
</tbody>
</table>

In the normal course of events you shouldn't meet up with the last three of these. The ArithmeticException turns up quite easily in your programs, as does the IndexOutOfBoundsException. A mistake in a for loop limit will produce the latter. In fact there are two subclasses of IndexOutOfBoundsException that specify the type of exception thrown more precisely - ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException. A NullPointerException can also turn up relatively easily, as can ArrayStoreException, ClassCastException, and IllegalArgumentException surprisingly enough. The last three here arise when you are using a base class variable to call methods for derived class objects. Explicit attempts to perform an incorrect cast, or store a reference of an incorrect type or pass an argument of the wrong type to a method will all be picked up by the compiler. These exceptions can, therefore, only arise from using a variable of a base type to hold references to a derived class object.

The IllegalArgumentException class is a base class for two further exception classes, IllegalThreadStateException and NumberFormatException. The former arises when you attempt an operation that is illegal in the current thread state. The NumberFormatException exception is thrown by the valueOf(), or decode() method in the classes representing integers - that is, the classes Byte, Short, Integer, and Long. The parseXXX() methods in these classes can also throw this exception. The exception is thrown if the String object passed as an argument to the conversion method is not a valid representation of an integer - if it contains invalid characters for instance. In this case a special return value cannot be used, so throwing an exception is a very convenient way to signal that the argument is invalid.

We will try out some of the RuntimeException exceptions later in the chapter as some of them are so easy to generate, but let's see what other sorts of exception classes have Exception as a base.

**Other Subclasses of Exception**

For all the other classes derived from the class Exception, the compiler will check that you've either handled the exception in a method where the exception may be thrown, or you've indicated that the method can throw such an exception. If you do neither your code won't compile. We'll look more at how we ensure the code does compile in the next two sections.
Apart from a few that have `RuntimeException` as a base, all exceptions thrown by methods in the Java class library are of a type that you must deal with. In Chapter 8 we will be looking at input and output where the code will be liberally sprinkled with provisions for exceptions being thrown.

**Important**

We'll see later in this chapter that when you want to define your own exceptions, you do this by subclassing the `Exception` class. Wherever your exception can be thrown by a method, the compiler will verify either that it is caught in the method, or that the method definition indicates that it can be thrown by the method, just as it does for the built-in exceptions.
Dealing with Exceptions

As we discussed in the previous sections, if your code can throw exceptions other than those of type Error or type RuntimeException, (you can take it that we generally include the subclasses when we talk about Error and RuntimeException exceptions) you must do something about it. Whenever you write code that can throw an exception, you have a choice. You can supply code within the method to deal with any exception that is thrown, or you can essentially ignore it by enabling the method containing the exception throwing code to pass it on to the code that called the method.

Let's first see how you can pass an exception on.

Specifying the Exceptions a Method Can Throw

Suppose you have a method which can throw an exception that is neither a subclass of RuntimeException nor of Error. This could be an IOException for example, which can be thrown if your method involves some file input or output operations. If the exception isn’t caught and disposed of in the method, you must at least declare that the exception can be thrown. But how do you do that?

You do it simply by adding a throws clause in the definition of the method. Suppose we write a method that uses the methods from classes that support input/output that are defined in the package java.io. You'll see in the chapters devoted to I/O operations that some of these can throw exceptions represented by objects of classes IOException and FileNotFoundException. Neither of these are subclasses of RuntimeException or Error, and so the possibility of an exception being thrown needs to be declared. Since the method can't handle any exceptions it might throw, for the simple reason that we don't know how to do it yet, it must be defined as:
Exception Objects

Well, you now understand how to put try blocks together with catch blocks and finally blocks in your methods. You may be thinking, at this point, that it seems a lot of trouble to go to, just to display a message when an exception is thrown. You may be right, but whether you can do very much more depends on the nature and context of the problem. In many situations a message may be the best you can do, although you can produce messages that are a bit more informative than those we've used so far in our examples. For one thing we have totally ignored the exception object that is passed to the catch block.

The exception object that is passed to a catch block can provide additional information about the nature of the problem that originated it. To understand more about this, let's first look at the members of the base class for exceptions Throwable, because these will be inherited by all exception classes and are therefore contained in every exception object that is thrown.

The Class Throwable

The class Throwable is the class from which all Java exception classes are derived - that is, every exception object will contain the methods defined in this class. The class Throwable has two constructors, a default constructor, and a constructor that accepts an argument of type String. The String object that is passed to the constructor is used to provide a description of the nature of the problem causing the exception. Both constructors are public.

Objects of type Throwable contain two items of information about an exception:

• A message, that we have just referred to as being initialized by a constructor

• A record of the execution stack at the time the object was created

The execution stack keeps track of all the methods that are in execution at any given instant. It provides the means whereby executing a return gets back to the calling point for a method. The record of the execution stack that is stored in the exception object will consist of the line number in the source code where the exception originated followed by a trace of the method calls that immediately preceded the point at which the exception occurred. This is made up of the fully qualified name for each of the methods called, plus the line number in the source file where each method call occurred. The method calls are in sequence with the most recent method call appearing first. This will help you to understand how this point in the program was reached.

The Throwable class has the following public methods that enable you to access the message and the stack trace:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMessage()</td>
<td>This returns the contents of the message, describing the current exception. This will typically be the fully qualified name of the exception class (it will be a subclass of Throwable), and a brief description of the exception.</td>
</tr>
<tr>
<td>printStackTrace()</td>
<td>This will output the message and the stack trace to the standard error output stream - which is the screen in the case of a console program.</td>
</tr>
<tr>
<td>printStackTrace(PrintStream s)</td>
<td>This is the same as the previous method except that you specify the output stream as an argument. Calling the previous method for an exception object, e, is equivalent to:</td>
</tr>
<tr>
<td></td>
<td>e.printStackTrace(System.err);</td>
</tr>
</tbody>
</table>

There's another method, fillInStackTrace(), which will update the stack trace to the point at which this method is called. For example, if you put a call to this method in the catch block:
Defining Your Own Exceptions

There are two basic reasons for defining your own exception classes:

- You want to add information when a standard exception occurs, and you can do this by rethrowing an object of your own exception class.
- You may have error conditions that arise in your code that warrant the distinction of a special exception class.

However, you should bear in mind that there's a lot of overhead in throwing exceptions, so it is not a valid substitute for 'normal' recovery code that you would expect to be executed frequently. If you have recovery code that will be executed often, then it doesn't belong in a catch block, rather in something like an if-then-else loop.

Let's see how to create our own exceptions.

Defining an Exception Class

Your exception classes must always have Throwable as a superclass, otherwise they will not define an exception. Although you can derive them from any of the standard exceptions, your best policy is to derive them from the Exception class. This will allow the compiler to keep track of where such exceptions are thrown in your program, and check that they are either caught or declared as thrown in a method. If you use RuntimeException or one of its subclasses, the compiler checking for catch blocks of your exception class will be suppressed.

Let's go through an example of how you define an exception class:
Summary

In this chapter you have learned what exceptions are and how to deal with them in your programs. You should make sure that you consider exception handling as an integral part of developing your Java programs. The robustness of your program code depends on how effectively you deal with exceptions that can be thrown within it.

The important concepts we have explored in this chapter are:

• Exceptions identify errors that arise in your program.

• Exceptions are objects of subclasses of the class Throwable.

• Java includes a set of standard exceptions that may be thrown automatically, as a result of errors in your code, or may be thrown by methods in the standard classes in Java.

• If a method throws exceptions that aren't caught, and aren't represented by subclasses of the class Error, or by subclasses of the class RuntimeException, then you must identify the exception classes in a throws clause in the method definition.

• If you want to handle an exception in a method, you must place the code that may generate the exception in a try block. A method may have several try blocks.

• Exception handling code is placed in a catch block that immediately follows the try block that contains the code that can throw the exception. A try block can have multiple catch blocks that deal with different types of exception.

• A finally block is used to contain code that must be executed after the execution of a try block, regardless of how the try block execution ends. A finally block will always be executed before execution of the method ends.

• You can throw an exception by using a throw statement. You can throw an exception anywhere in a method. You can also rethrow an existing exception in a catch block to pass it to the calling method.
You can define your own exception classes that, in general, should be derived from the class Exception.
Exercises

1. Write a program that will generate exceptions of type NullPointerException, NegativeArraySizeException, and IndexOutOfBoundsException. Record the catching of each exception by displaying the message stored in the exception object, and the stack trace record.

2. Add an exception class to the last example that will differentiate between the index-out-of-bounds error possibilities, rethrow an appropriate object of this exception class in divide(), and handle the exception in main().

3. Write a program that calls a method which throws an exception of type ArithmeticException at a random iteration in a for loop. Catch the exception in the method, and pass the iteration count when the exception occurred to the calling method, by using an object of an exception class you define.

4. Add a finally block to the method in the previous example to output the iteration count when the method exits.
Chapter 8: Understanding Streams

Overview

This is the first of four chapters devoted to streams and file input/output. This chapter introduces streams, and deals with keyboard input, and output to the command line.

By the end of this chapter, you will have learned:

• What is a stream and what are the main classes that Java provides to support stream operations.

• What are stream readers and writers and what they are used for.

• How to read data from the keyboard.

• How to format data that you write to the command line.
Streams and the New I/O Capability

The package that supports stream input/output is java.io, and it is vast. It defines around fifty classes, many of which have a large number of methods. It is therefore quite impractical to go into them all in detail in a book of this kind. Refer to the java documentation for more information. Our strategy in this, and the following three chapters, will be to take a practical approach. The idea is to provide an overall grounding of the concepts involved, and to equip you with enough detailed knowledge to be able to do a number of specific, useful, and practical things in your programs. These are:

- To be able to read data of various kinds from the keyboard.
- To be able to create formatted output to the command line.
- To be able to read and write files containing basic data.
- To be able to read and write files containing objects.

To achieve this, we will give you an overview of what the important stream classes do, and how they interrelate, together with the classes that operate on streams. We will go into detail selectively, just exploring the classes and methods that we need to accomplish specific things. We will also be sticking to the latest and greatest I/O capability in the JDK 1.4, which makes it unnecessary to delve into a lot of the original stream classes.

Up to and including Java 1.3, the only way to read and write files was to use a stream. Java 1.4 introduced a new I/O capability in the java.nio and java.nio.channels packages for reading and writing files that contain data of the primitive Java types, including strings. This capability completely supersedes the stream I/O capability in this context. It is much more efficient, and in many ways easier to use, so we will limit our discussions of streams for handling files to the extent necessary to understand the new I/O capability. We will go into the new I/O capability in detail in the next two chapters.

Two areas where you must still use the facilities provided by the stream classes are reading from the keyboard, and writing to the command line. We will cover both of these in this chapter along with some general aspects of the stream classes and the relationships between them. The new I/O capability does not provide for objects to be written and read, so you still need to use streams for this. We will look into how we read and write objects to a file in Chapter 11 Serializing Objects.
Understanding Streams

A **stream** is an abstract representation of an input or output device that is a source of, or destination for, data. You can write data to a stream and read data from a stream. You can visualize a stream as a sequence of bytes that flows into or out of your program.

When you write data to a stream, the stream is called an **output stream**. The output stream can go to any device to which a sequence of bytes can be transferred, such as a file on a hard disk, or a phone line connecting your system to a remote system. An output stream can also go to your display screen, but only at the expense of limiting it to a fraction of its true capability. This is output to the command line. When you write to your display screen using a stream, it can only display characters, not graphical output. Graphical output requires more specialized support that we will discuss from **Chapter 15** onwards. Note that while a printer can be considered notionally as a stream, printing in Java does not work this way. A printer in Java is treated as a graphical device, so sending output to the printer is very similar to displaying graphical output on your display screen. You will learn how printing works in Java in **Chapter 20**.

You read data from an **input stream**. In principle, this can be any source of serial data, but is typically a disk file, the keyboard, or a remote computer.

Under normal circumstances, file input and output for the machine on which your program is executing is only available to Java applications. It is not available to Java applets except to a strictly limited extent. If this were not so, a malicious Java applet embedded in a web page could trash your hard disk. An IOException will normally be thrown by any attempted operation on disk files on the local machine in a Java applet. The directory containing the .class file for the applet, and its subdirectories, are freely accessible to the applet. Also, the security features in Java can be used to control what an applet (and an application running under a Security Manager) can access so that an applet can only access files or other resources *for which it has explicit permission*.

The main reason for using a stream as the basis for input and output operations is to make your program code for these operations independent of the device involved. This has two advantages. First, you don't have to worry about the detailed mechanics of each device, which are taken care of behind the scenes. Second, your program will work for a variety of input/output devices without any changes to the code.

Stream input and output methods generally permit very small amounts of data, such as a single character or byte, to
be written or read in a single operation. Transferring data to or from a stream like this may be extremely inefficient, so a stream is often equipped with a buffer in memory, in which case it is called a buffered stream. A buffer is simply a block of memory that is used to batch up the data that is transferred to or from an external device. Reading or writing a stream in reasonably large chunks will reduce the number of input/output operations necessary, and thus make the process more efficient.

When you write to a buffered output stream, the data is sent to the buffer, and not to the external device. The amount of data in the buffer is tracked automatically, and the data is usually sent to the device when the buffer is full. However, you will sometimes want the data in the buffer to be sent to the device before the buffer is full, and there are methods provided to do this. This operation is usually termed flushing the buffer.

Buffered input streams work in a similar way. Any read operation on a buffered input stream will read data from the buffer. A read operation for the device that is the source of data for the stream will only be read when the buffer is empty, and the program has requested data. When this occurs, a complete buffer-full of data will be read automatically from the device, if sufficient data is available.

Binary and Character Streams

The java.io package supports two types of streams, binary streams, which contain binary data, and character streams, which contain character data. Binary streams are sometimes referred to as byte streams. These two kinds of streams behave in different ways when you read and write data.

When you write data to a binary stream, the data is written to the stream as a series of bytes, exactly as it appears in memory. No transformation of the data takes place. Binary numerical values are just written as a series of bytes, four bytes for each value of type int, eight bytes for each value of type long, eight bytes for each value of type double, and so on. As we saw in Chapter 2, Java stores its characters internally as Unicode characters, which are 16-bit characters, so each Unicode character is written to a binary stream as two bytes, the high byte being written first.

Character streams are used for storing and retrieving text. You may also use character streams to read text files not written by a Java program. All binary numeric data has to be converted to a textual representation before being written to a character stream. This involves generating a character representation of the original binary data value. Reading numeric data from a stream that contains text involves much more work than reading binary data. When you read a value of type int from a binary stream, you know that it consists of four bytes. When you read an integer from a character stream, you have to determine how many characters make up the value. For each numerical value you read from a character stream, you have to be able to recognize where the value begins and ends, and then convert the token - the sequence of characters that represents the value - to its binary form. This is illustrated below:

![Diagram of stream contents and token recognition](image)

When you write strings to a stream as character data, by default, the Unicode characters are automatically converted to the local representation of the characters in the host machine, and these are then written to the stream. When you
read a string, the default mechanism is to convert the data from the stream back to Unicode characters from the local machine representation. With character streams, your program reads and writes Unicode characters, but the stream will contain characters in the equivalent character encoding used by the local computer.

You don't have to accept the default conversion process for character streams. Java allows named mappings between Unicode characters and sets of bytes to be defined, called **charsets**, and you can select an available charset that should apply when data is transferred to, or from, a particular character stream. We won't be going into this in detail, but you can find more information on defining and using charsets in the SDK documentation for the Charset class.
The Classes for Input and Output

There are quite a number of stream classes, but as you will see, they form a reasonably logical structure. Once you see how they are related, you shouldn't have much trouble using them. We will work through the class hierarchy from the top down, so you will be able to see how the classes hang together, and how you can combine them in different ways to suit different situations.

The package java.io contains the classes that provide the foundation for Java's support for stream I/O:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputStream</td>
<td>The base class for byte stream input operations.</td>
</tr>
<tr>
<td>OutputStream</td>
<td>The base class for byte stream output operations.</td>
</tr>
</tbody>
</table>

InputStream, and OutputStream are both abstract classes. As you are well aware by now, you cannot create instances of an abstract class—these classes only serve as a base from which to derive classes with more concrete input or output capabilities. However, both of the classes declare methods that define a basic set of operations for the streams they represent, so the fundamental characteristics of how a stream is accessed are set by these classes. Generally, the InputStream and OutputStream classes, and their subclasses, represent byte streams and provide the means of reading and writing binary data.

Basic Input Stream Operations

As we saw in the previous section, the InputStream class is abstract, so you cannot create objects of this class type. Nonetheless, input stream objects are often accessible via a reference of this type, so the methods identified in this class are what you get. The InputStream class includes three methods for reading data from a stream:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read()</td>
<td>This method is abstract in the InputStream class, so it has to be defined in a subclass. The method returns the next byte available from the stream as type int. If the end of the stream is reached, the method will return the value -1. An exception of type IOException will be thrown if an I/O error occurs.</td>
</tr>
</tbody>
</table>
read(byte[] array)

This method reads bytes from the stream into successive elements of array. The maximum of array.length bytes will be read. The method will not return until the input data is read, or the end of the stream is detected. The method returns the number of bytes read, or -1 if no bytes were read because the end of the stream was reached. If an I/O error occurs, an exception of type IOException will be thrown. If the argument to the method is null then a NullPointerException will be thrown. An input/output method that does not return until the operation is completed is referred to as a **blocking** method, and you say that the methods blocks until the operation is complete.

read(byte[] array, int offset, int length)

This works in essentially the same way as the previous method, except that up to length bytes are read into array starting with the element array[offset].

The data is read from the stream by these methods simply as bytes. No conversion is applied to the bytes read. If any conversion is required, for a stream containing bytes in the local character encoding for instance, you must provide a way to handle this. We will see how this might be done in a moment.

You can skip over bytes in an InputStream by calling its skip() method. You specify the number of bytes to be skipped as an argument of type long, and the actual number of bytes skipped is returned, also a value of type long. This method can throw an IOException if an error occurs.

You can close an InputStream by calling its close() method. Once you have closed an input stream, subsequent attempts to access or read from the stream will cause an IOException to be thrown.

The InputStream class has seven direct subclasses as shown in the diagram below.

We will be using the FileInputStream class in Chapter 10 Writing Files, and the ObjectInputStream class in Chapter 11 Reading Files.

The FilterInputStream class has a further nine direct subclasses that provide more specialized ways of filtering or transforming data from an input stream. We will only be using the BufferedInputStream class, but here's the complete set with an indication of what each of them does:
<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferedInputStream</td>
<td>Buffers input from another stream in memory to make the read operations more efficient.</td>
</tr>
<tr>
<td>DataInputStream</td>
<td>Reads data of primitive types from a binary stream.</td>
</tr>
<tr>
<td>CheckedInputStream</td>
<td>Reads an input stream and maintains a checksum for the data that is read to verify its integrity.</td>
</tr>
<tr>
<td>CipherInputStream</td>
<td>Reads data from an encrypted input stream.</td>
</tr>
<tr>
<td>DigestInputStream</td>
<td>Reads data from an input stream and updates an associated message digest. A message digest is a mechanism for combining an arbitrary amount of data from a stream into a fixed length value that can be used to verify the integrity of the data.</td>
</tr>
<tr>
<td>InflaterInputStream</td>
<td>Reads data from a stream that has been compressed, such as a ZIP file for example.</td>
</tr>
<tr>
<td>LineNumberInputStream</td>
<td>Reads data from a stream and keeps track of the current line number. The line number starts at 0 and is incremented each time a newline character is read.</td>
</tr>
<tr>
<td>ProgressMonitorInputStream</td>
<td>Reads data from an input stream and uses a progress-monitor to monitor reading the stream. If reading the stream takes a significant amount of time, a progress dialog will be displayed offering the option to cancel the operation. This is used in window-based applications for operations that are expected to be time consuming.</td>
</tr>
<tr>
<td>PushbackInputStream</td>
<td>Adds the capability to return the last byte that was read back to the input stream so you can read it again.</td>
</tr>
</tbody>
</table>

You can create a BufferedInputStream object from any other input stream, since the constructor accepts a reference of type InputStream as an argument. The BufferedInputStream class overrides the methods inherited from InputStream. For instance, in the following example:
The Standard Streams

Your operating system will typically define three standard streams that are accessible through members of the System class in Java:

- A **standard input stream** that usually corresponds to the keyboard by default. This is encapsulated by the in member of the System class, and is of type InputStream.

- A **standard output stream** that corresponds to output on the command line. This is encapsulated by the out member of the System class, and is of type PrintStream.

- A **standard error output stream** for error messages that usually maps to the command line output by default. This is encapsulated by the err member of the System class, and is also of type PrintStream.

You can reassign any of these to another stream within a Java application. The System class provides the static methods setIn(), setOut(), and setErr() for this purpose. The setIn() method requires an argument of type InputStream that specifies the new source of standard input. The other two methods expect an argument of type PrintStream.

Since the standard input stream is of type InputStream, we are not exactly overwhelmed by the capabilities for reading data from the keyboard in Java. Basically, we can read a byte or an array of bytes using a read() method as standard, and that's it. If you want more than that, reading integers, or decimal values, or strings as keyboard input, you're on your own. Let's see what we can do to remedy that.

Getting Data From the Keyboard

To get sensible input from the keyboard, you have to be able to scan the stream of characters and recognize what they are. When you read a numerical value from the stream, you have to look for the digits and possibly the sign and decimal point, figure out where the number starts and ends in the stream, and finally convert it to the appropriate value. To write the code to do this from scratch would take quite a lot of work. Fortunately, we can get a lot of help from the StreamTokenizer class in the java.io package.

The term **token** refers to a data item, such as a number or a string that will, in general, consist of several consecutive characters of a particular kind from the stream. For example, a number is usually a sequence of characters that consists of digits, maybe a decimal point, and sometimes a sign in front. The class has the name StreamTokenizer because it can read characters from a stream, and parse it into a series of tokens that it recognizes.
You create a StreamTokenizer object from a stream reader object that reads data from the underlying input stream. Since we want to read the standard input stream, System.in, we shall use an InputStreamReader that converts the raw bytes from the stream from the local character encoding to Unicode characters before the StreamTokenizer object sees them. In the interests of efficiency we will also buffer the data from the InputStreamReader through a BufferedReader object that will buffer the data in memory. We will therefore create our StreamTokenizer object like this:
Summary

In this chapter, we have discussed the facilities for inputting and outputting basic types of data to a stream. The important points we have discussed include:

•

A stream is an abstract representation of a source of serial input, or a destination for serial output.

•

The classes supporting stream operations are contained in the package java.io.

•

Two kinds of stream operations are supported, binary stream operations will result in streams that contain bytes, and character stream operations are for streams that contain characters in the local machine character encoding.

•

No conversion occurs when characters are written to, or read from, a byte stream. Characters are converted from Unicode to the local machine representation of characters when a character stream is written.

•

Byte streams are represented by sub classes of the classes InputStream and OutputStream.

•

Character stream operations are provided by sub classes of the Reader and Writer classes.

•

A nod is as good as a wink to a blind horse.
Exercises

1.

Modify the FormattedWriter class to support centered output of the value in the output field.

2.

Use a StreamTokenizer object to parse a string entered from the keyboard containing a series of data items separated by commas and output each of the items on a separate line.

3.

Create a class defining an object that will parse each line of input from the keyboard that contains items separated by an arbitrary delimiter (for example, a colon, or a comma, or a forward slash, etc.) and return the items as an array of type String[]. For example, the input might be:

1/one/2/two

The output would be returned as an array of type String[] containing "1", "one", "2", "two".
Chapter 9: Accessing Files and Directories

Overview

In this chapter, we will explore how we identify, access, and manipulate files and directories on your hard drive. This will include the ability to create new files and directories, but not to read or write files. We will get to that starting in the next chapter.

In this chapter you will learn:

• How you create File objects and use them to examine files and directories

• How you can use File class methods to examine the contents of the hard drives on your system

• How to create new files and directories on your hard drive

• How to create temporary files

• How you create FileOutputStream objects
Working with File Objects

It is easy to forget that a File object doesn't actually represent a file. You need to keep reminding yourself that a File object encapsulates a pathname or reference to what may or may not be a physical file or directory on your hard disk, not the physical file or directory itself. The fact that you create a File object does not determine that a file or directory actually exists. This is not as strange as it might seem at first sight. After all, you will often be defining a File object that encapsulates a path to a new file or directory that you intend to create.

As we will see, a File object serves two purposes:

• It enables you to check the pathname that it encapsulates against the physical file system and see whether it corresponds to a real file or directory.

• You can use it to create file stream objects.

The File class provides several methods for you to test the path that a File object encapsulates in various ways, as well as the physical file or directory it may represent. You can determine whether or not an object does represent a path to an existing file or directory, for example. If a File object does correspond to a real file, you have methods available that you can use to modify the file in a number of ways.

Creating File Objects

You have a choice of four constructors for creating File objects. The simplest accepts a String object as an argument that specifies a path for a file or a directory. For example, you could write the statement:
Creating File Output Streams

You use a FileOutputStream object when you want to write to a physical file on a disk. The FileOutputStream class is derived from the OutputStream class and therefore inherits the methods of that class for writing to a file. However, we won't bother going into detail on these, or the versions in the FileOutputStream class that override them, as we will be using the new file channel capability to write to a file.

There are five constructors for FileOutputStream objects:

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileOutputStream(String filename)</td>
<td>Creates an output stream for the file filename. The existing contents of the file will be overwritten. If the file cannot be opened for any reason, an exception of type FileNotFoundException will be thrown.</td>
</tr>
<tr>
<td>FileOutputStream(String filename, boolean append)</td>
<td>Creates an output stream for the file filename. Data written to the file will be appended following the existing contents if append is true. If append is false any existing file contents will be overwritten. If the file cannot be opened for any reason, an exception of type FileNotFoundException will be thrown.</td>
</tr>
<tr>
<td>FileOutputStream(File file)</td>
<td>Creates a file output stream for the file represented by the object file. Any existing file contents will be overwritten. If the file cannot be opened, an exception of type FileNotFoundException will be thrown.</td>
</tr>
<tr>
<td>FileOutputStream(File file, boolean append)</td>
<td>Creates a file output stream for the file represented by the object file. Data written to the file will be appended following the existing contents if append is true. If append is false any existing file contents will be overwritten. If the file cannot be opened for writing for any reason, an exception of type FileNotFoundException will be thrown.</td>
</tr>
<tr>
<td>FileOutputStream(FileDescriptor desc)</td>
<td>Creates an output stream corresponding to the argument desc. A FileDescriptor object represents an existing connection to a file so, since the file must exist, this constructor does not throw an exception of type FileNotFoundException.</td>
</tr>
</tbody>
</table>

The first four constructors will also create the file if it does not already exist, but only if the parent directory exists, so it is a good idea to check this. All of these constructors can throw a SecurityException if writing to the file is not authorized on your system, although by default there is no security manager for an application in which case there are no restrictions on writing files. Once you have created a FileOutputStream object, the physical file is automatically
opened, ready to be written. Once you have written the file, using a channel as we shall see in Chapter 10, you can close the file by calling the close() method for the FileOutputStream object. This also closes the file channel and releases all system resources associated with the stream.

To create a stream object of type FileOutputStream, you will typically create a File object first, and then pass that to a FileOutputStream constructor. This approach enables you to check the properties of the file using the File object before you try to create the stream and thus, avoid potential problems. Of course, you can create a FileOutputStream object directly from a String object that defines the path and file name, but it is generally much less convenient to do this. We will come back to the third possibility creating a file stream object from a FileDescriptor object in a moment.

In passing, here is how you would create a file output stream directly from the file name:
Summary

In this chapter, we have discussed the facilities for inspecting physical files and directories, and for writing basic types of data to a file. The important points we have discussed include:

- An object of the class File can encapsulate a file or directory path. The path encapsulated by a File object does not necessarily correspond to a physical file or directory.

- You can use a File object to test whether the path it encapsulates refers to a physical file or directory. If it does not, there are methods available to create it together with any directories that are part of the path that may also be required.

- The File class defines static methods for creating temporary files.

- An object of type FileDescriptor can also identify a physical file.

- A FileOutputStream object can be created from a File object and the file will be opened for writing. If the file does not exist it will be created where possible.

- You can lead a horse to water but you can't make him drink.
Exercises

1.

Modify the example that avoids overwriting a file to allow for file names that do not have extensions.

2.

Filenames on many systems are not of unlimited length, so appending _old to file names may break down at some point. Modify the example that avoids overwriting a file to append a three-digit numerical value to the file name to differentiate it from the existing file instead of just adding _old. The program should check for the presence of three digits at the end of the name for the existing file and replace this with a value incremented by an amount to make it unique. (That is, increment the last three digits by 1 until a unique file name is created.)

3.

Write a program that will list all the directories in a directory defined by a path supplied as a command line argument, or all the directories on a system if no command line argument is present. (Hint: The listRoots() method will give you the roots on a system and the listFiles() method will give you an array of File objects for the files and directories in any given directory including a root.)
Chapter 10: Writing Files

Overview

In this chapter, we will be looking at ways in which basic data can be written to a file using the new file I/O capability. The JDK 1.4 release introduced this new mechanism for file I/O that supersedes the read/write capability provided by readers and writers from the java.io package when applied to file streams. Since the new file I/O does everything that the old capability does, and does it better, we will focus just on that.

In this chapter you will learn:

- The principles of reading and writing files using the new I/O.
- How to obtain a file channel for a file.
- How to create a buffer and load it with data.
- What view buffers are and how you use them.
- How you use a channel object to write the contents of a buffer to a file.
File I/O Basics

If you are new to programming file operations, there are a couple of things that may not be apparent to you and can be a source of confusion so let's clarify them before we go any further.

Firstly, let's consider the nature of a file. Once you have written data to a file, you can regard it as just a linear sequence of bytes. The bytes in a file are referenced by their offset from the beginning, so the first byte is byte 0, the next byte is byte 1, the third byte is byte 2, and so on through to the end of the file. If there are n bytes in a file the last byte will be at offset n-1. There is no specific information in the file about how the data originated or what it represents unless you explicitly put it there. Even if there is, you need to know that it's there and read and interpret the data accordingly.

For instance if you write a series of 25 binary values of type int to a file, it will contain 100 bytes. There will be nothing in the file to indicate that the data consists of four byte integers so there is nothing to prevent you from reading the data back as 50 Unicode characters or 10 long values followed by a string, or any other arbitrary collection of data items that corresponds to 100 bytes. Of course, the result is unlikely to be very meaningful unless you interpret the data in the form in which it was written. This implies that to read data from a file correctly, you need to have prior knowledge of the structure and format of the data.

There are many ways in which the form of the data in the file may be recorded or implied. For instance, one way that the format of the data in a file can be communicated is to use an agreed file name extension for data of a particular kind, such as .java or .gif or .wav. Each type of file has a predefined structure so from the file extension you know how to interpret the data in the file. Another way is to use a generalized mechanism for communicating data and its structure such as XML. We will be looking into how we can work with XML in Java in Chapters 21 and 22.

You can access an existing file to read from or write to it in two different ways, described as sequential access or random access. The latter is sometimes referred to as direct access. Sequential access to a file is quite straightforward and works pretty much as you would expect. Sequential read access involves reading bytes from the file starting from the beginning with byte 0. Of course, if you are only interested in the file contents starting at byte 100, you can just read and ignore the first 100 bytes. Sequential write access involves writing bytes to the file either starting at the beginning if you are replacing the existing data or at the end if you are appending new data to the file.

The term random access is often misunderstood initially. Just like sequential access, random access is just a way of accessing data in a file and has nothing to do with how the data in the file is structured or how the physical file was originally written. You can access any file randomly for reading and/or writing. When you access a file randomly, you can read one or more bytes from the file starting at any point. For instance, you could read 20 bytes starting at the thirteenth byte in the file (which will be the byte at offset 12 of course), then read 50 bytes starting at the 101st byte or any other point that you choose. Similarly, you can update an existing file in random access mode by writing data starting at any point in the file. In random access mode, the choice of where to start reading or writing and how many bytes you read or write, is entirely up to you. You just need to know the offset for the byte where a read or write operation should start. Of course, for these to be sensible and successful operations, you have to have a clear idea of how the data in the file is structured.
Important

First a note of caution; before running any of the examples in this chapter, be sure to set up a separate directory for storing the files that you are using when you are testing programs. It's also not a bad idea to back up any files and directories on your system that you don't want to risk losing. But of course, you do back up your files regularly anyway—right?

The old adage, 'If anything can go wrong, it will,' applies particularly in this context, as does the complementary principle, 'If anything can't go wrong, it will.' Remember also that the probability of something going wrong increases in proportion to the inconvenience it is likely to cause.
File Input and Output

The new file I/O capabilities introduced in Java 1.4 provide the potential for substantially improved performance at the cost of some slight increase in complexity over the I/O facilities of previous releases. There are three kinds of objects involved in reading and writing files using the new I/O capability:

- A **file stream object** that encapsulates the physical file that you are working with. We saw how to create FileOutputStream objects at the end of the previous chapter and you use these for files to which you want to write. In the next chapter we will be using FileInputStream objects for files that we want to read.

- One or more **buffer objects** in which you put the data to be written to a file, or from which you get the data that has been read. We will learn about buffer objects in the next section.

- A **channel object** that provides the connection to the file and does the reading or writing of the data using one or more buffer objects. We will see how to obtain a channel from a file stream object later in this chapter.

The way in which these types of object work together is illustrated in the diagram below:

![Diagram of file I/O objects](image)

The process for writing and reading files is basically quite simple. To write to a file, you load data into one or more buffers that you have created, and then call a method for the channel object to write the data to the file that is encapsulated by the file stream. To read from a file, you call a method for the channel object to read data from the file into one or more buffers, and then retrieve the data from the buffers.

There are four classes defined in the java.io package that we will be using when we are working with files. As we have said, the FileInputStream and FileOutputStream classes define objects that provide access to a file for reading or writing respectively. You use an object of type RandomAccessFile when you want to access a file randomly, or when you want to use a single channel to both read from and write to a file. We will be exploring this, along with the FileInputStream class in the next chapter. You will see from the SDK documentation for the FileInputStream, FileOutputStream, and RandomAccessFile classes that they each provide methods for I/O operations. However, we
will ignore these, as we will be using the services of a file channel to perform operations with objects of these stream classes. The only method from these classes that we will be using is the close() method, that closes the file and any associated channel.
Channels

Channels were introduced in the 1.4 release of Java to provide a faster capability for input and output operations with files, network sockets, and piped I/O operations between programs than the methods provided by the stream classes. We will only be discussing channels in the context of files. The channel mechanism can take advantage of buffering and other capabilities of the underlying operating system and therefore is considerably more efficient than using the operations provided directly within the file stream classes. As we said earlier, a channel transfers data between a file and one or more buffers. We will take a quick look at the overall relationship between the various classes that define channels and buffers, and then look into the details of how you use channels with file streams.

There are a considerable number of classes and interfaces defining both channels and buffers. They also have similar names such as ByteBuffer and ByteChannel. Of course, File and file stream objects are also involved in file I/O operations so you will be using at least four different types of objects working together when you read from or write to files. Just to clarify what they all do, here's a summary of the essential role of each of them in file operations:

- A File object encapsulates a path to a file or a directory, and such an object encapsulating a file path can be used to construct a file stream object.

- A FileInputStream object encapsulates a file that can be read by a channel. A FileOutputStream object encapsulates a file that can be written by a channel. As we will see in the next chapter, a RandomAccessFile object can encapsulate a file that can be both read and written by a channel.

- A buffer just holds data in memory. You load into a buffer what is to be written to a file using the buffer's put() methods. You use a buffer's get() methods to retrieve data that has been read from a file.

- You obtain a FileChannel object from a file stream object or a RandomAccessFile object. A FileChannel object can read and write a file using read() and write() methods with a buffer or buffers as the source or destination of the data.

The channel interfaces and classes that we will be using are in the java.nio.channels package. The classes that define buffers are defined in the java.nio package. In a program that reads or writes files we will therefore need import statements for class names from at least three packages, the two packages we have just introduced plus the java.io package.

Channel Operations
There are a series of channel interfaces, each of which declares a set of one or more related operations that a channel may perform. They all extend a common interface, Channel, which declares two methods:

- The close() method that closes a channel
- The isOpen() method that tests the state of the channel, returning true if it is open and false otherwise

Note that closing a channel does not necessarily close the file that the channel is attached to but closing a file also closes its channel. The channel interfaces are related as illustrated in the hierarchy shown below:

Each arrow points from a given interface to an interface that it implements. The ByteChannel interface simply combines the operations specified by the ReadableByteChannel and WritableByteChannel interface without declaring any additional methods. The ScatteringByteChannel interface extends the ReadableByteChannel interface by adding methods that allow data to be read and distributed amongst several separate buffers in a single operation. The GatheringByteChannel adds methods to those of the WritableByteChannel to permit writing from a number of separate buffers in a single operation.

The methods that each channel interface declares are as follows:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadableByteChannel</td>
<td>int read(ByteBuffer input)</td>
</tr>
<tr>
<td></td>
<td>Reads bytes from a channel into the buffer specified by the argument and returns the number of bytes read, or -1 if the end of the stream is reached.</td>
</tr>
<tr>
<td>WritableByteChannel</td>
<td>int write(ByteBuffer output)</td>
</tr>
<tr>
<td></td>
<td>Writes bytes from the buffer specified by the argument to the channel, and returns the number of bytes written.</td>
</tr>
<tr>
<td>Class</td>
<td>Method Definition</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ByteChannel</td>
<td>This interface just inherits methods from the ReadableByteChannel and WritableByteChannel interfaces. No additional methods are declared.</td>
</tr>
<tr>
<td>ScatteringByteChannel</td>
<td>int read(ByteBuffer[] inputs)</td>
</tr>
<tr>
<td></td>
<td>int read(ByteBuffer[] inputs, int offset, int length)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>GatheringByteChannel</td>
<td>int write(ByteBuffer[] outputs)</td>
</tr>
<tr>
<td></td>
<td>int write(ByteBuffer[] outputs, int offset, int length)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All of these methods can throw exceptions of one kind or another and we will go into details on these when we come to apply them. Note that a channel only works with buffers of type ByteBuffer. There are other kinds of buffers as we shall see, but you can't use them directly with the read() and write() methods for a channel. We will see what determines the number of bytes read or written in an operation when we discuss buffers in detail.

**File Channels**

A FileChannel object defines a channel for a physical file, and provides an efficient mechanism for reading, writing, and manipulating the file. You can't create a FileChannel directly. You first have to create a file stream object for the file, then obtain a reference to the FileChannel object for the file by calling the getChannel() method for the file stream object. Here's how you would obtain the channel for a FileOutputStream object:

```java
File aFile = new File("C:/Beg Java Stuff/myFile.text");
```
Buffers

All the classes that define buffers have the abstract class Buffer as a base. The Buffer class therefore defines the fundamental characteristics common to all buffers. A particular buffer can store a sequence of elements of a given type, and an element can be of any primitive data type other than boolean. Thus, you can create buffers to store byte values, char values, short values, int values, long values, float values, or double values. The following classes in the java.nio package define these buffers:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteBuffer</td>
<td>A buffer that stores elements of type byte. You can also store the binary values of any of the other primitive types in this buffer, except for type boolean. Each binary value that you store will occupy a number of bytes in the buffer determined by the type values of type char or short will occupy two bytes, int values will occupy four bytes, and so on. Only buffers of this type can be used in a file I/O operation.</td>
</tr>
<tr>
<td>CharBuffer</td>
<td>A buffer that only stores elements of type char.</td>
</tr>
<tr>
<td>ShortBuffer</td>
<td>A buffer that only stores elements of type short.</td>
</tr>
<tr>
<td>IntBuffer</td>
<td>A buffer that only stores elements of type int.</td>
</tr>
<tr>
<td>LongBuffer</td>
<td>A buffer that only stores elements of type long.</td>
</tr>
<tr>
<td>FloatBuffer</td>
<td>A buffer that only stores elements of type float.</td>
</tr>
<tr>
<td>DoubleBuffer</td>
<td>A buffer that only stores elements of type double.</td>
</tr>
</tbody>
</table>

We keep repeating except for type boolean every so often, so we had better address that. The various types of buffers only provide for the numerical data types and type boolean does not fit into this category. Of course, you may actually want to record some boolean values in a file. In this case, you have to devise a suitable alternative representation. You could use integer values 0 and 1, or perhaps strings "true" and "false", or even characters 't' and 'f'. You could even represent a boolean value as a single bit and pack eight of them at a time into a single byte, but this is only likely to be worthwhile if you have a lot of them. Which approach you choose will depend on what is most convenient in the context in which you are using them.
While we have seven different classes defining buffers, a channel only uses buffers of type ByteBuffer to read or write data. The other types of buffers in the table above are called view buffers, because they are usually created as views of an existing buffer of type ByteBuffer. We will see how and why a little later in this chapter.

**Buffer Capacity**

Each type of buffer stores elements of a specific kind: a ByteBuffer object holds bytes, a LongBuffer object holds integers of type long, and so on for the other buffer types. The capacity of a buffer is the maximum number of elements it can contain, not the number of bytes unless, of course, it stores elements of type byte. The capacity of a buffer is fixed when you create it and cannot be changed subsequently. You can obtain the capacity for a buffer object as a value of type int by calling the capacity() method that it inherits from the Buffer class.

Of course, for a buffer that stores bytes, the capacity will be the maximum number of bytes it can hold, but for a buffer of type DoubleBuffer for instance that stores double values, the capacity will be the maximum number of double values you can put in it. Elements in a buffer are indexed from zero so the index position for referencing elements in a buffer runs from 0 to capacity-1.

**Buffer Position and Limit**

A buffer also has a limit and a position, both of which affect read/write operations executed by a channel using the buffer.

The position is the index position of the next buffer element to be read or written. This sounds a little strange, but keep in mind that a buffer can be for file input or output. For example, with a buffer used for output, the position identifies the next element to be written to the file. For a buffer used for file input, the position identifies where the next element read from the file will be stored.

The limit is the index position of the first element that should not be read or written. Thus, elements can be read or written starting with the element at position, and up to and including the element at limit-1. Thus if you want to fill all the elements in a buffer, the position must be at zero since this is where the first data item will go, and the limit must be equal to the capacity since the last data item has to be stored at the last element in the buffer, which is capacity-1.
A buffer's position and limit are used for determining what elements are involved in a read or write operation executed by a channel. How they affect I/O operations is easier to understand if we take a specific example. Let's first consider an operation that writes data from the buffer to a file.

When a file write operation is executed by a channel using a given buffer, elements from the buffer will be written to the file starting at the index specified by the position. Successive elements will be written to the file up to, and including, the element at index position \( limit - 1 \). For a read operation, data that is read from the file is stored in a buffer starting at the element index given by the buffer position. Elements will continue to be read, assuming they are available from the file, up to the index position \( limit - 1 \). Thus when you want to write all the data from a buffer, the limit will have to be equal to the capacity. In this case the limit will be an index value that is one beyond the index value for the last element in the buffer, so \( limit - 1 \) will refer to the last element.

The position and limit are involved when you load data into a buffer or retrieve data from it. The position specifies where the next element should be inserted in a buffer or retrieved from it. As we shall see, you will usually have the position automatically incremented to point to the next available position when you insert or extract elements in a buffer. The limit acts as a constraint to indicate where the data in a buffer ends, a bit like an end-of-file marker. You cannot insert or extract elements beyond the position specified by the limit.

Since a buffer's position is an index, it must be greater than or equal to zero. You can also deduce that it must also be less than or equal to the limit. Clearly, the limit cannot be greater than the capacity of a buffer. Otherwise, we could be trying to write elements to positions beyond the end of the buffer. However, as we have seen, it can be equal to it. These relationships can be expressed as:

\[
0 \leq \text{position} \leq \text{limit} \leq \text{capacity}
\]

As a general rule, if your code attempts to do things directly or indirectly that result in these relationships being violated, an exception will be thrown.

When you create a new independent buffer, its capacity will be fixed at a value that you specify. It will also have a position of zero and its limit will be set to its capacity. When you create a view buffer from an existing ByteBuffer, the contents of the view buffer starts at the current position for the ByteBuffer. The capacity and limit for the view buffer will be set to the limit for the original buffer, divided by the number of bytes in an element in the view buffer. The limit and position for the view buffer will subsequently be independent of the limit and position for the original buffer.

### Setting the Position and Limit

You can set the position and limit for a buffer explicitly by using the following methods defined in the Buffer class:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>

This document is created with the unregistered version of CHM2PDF Pilot
### position(int newPosition)
Sets the position to the index value specified by the argument. The new position value must be greater than or equal to zero, and not greater than the current limit, otherwise an exception of type IllegalArgumentException will be thrown. If the buffer's mark is defined (we will come to the mark in the [next section](#)) and greater than the new position, it will be discarded.

### limit(int newLimit)
Sets the limit to the index value specified by the argument. If the buffer's position is greater than the new limit it will be set to the new limit. If the buffer's mark is defined and exceeds the new limit it will be discarded. If the new limit value is negative or greater than the buffer's capacity, an exception of type IllegalArgumentException will be thrown.

Both of these methods return a reference of type Buffer for the object for which they were called. This enables you to chain them together in a single statement. For instance, given a buffer reference, buf, you could set both the position and the limit with the statement:
Writing To a File

To start with, we will be using the simplest write() method for a file channel that writes the data contained in a single ByteBuffer object to a file. The number of bytes written to the file is determined by the buffer's position and limit when the write() method executes. Bytes will be written starting with the byte at the buffer's current position. The number of bytes written is \( \text{limit-position} \), which is the number returned by the remaining() method for the buffer object. The write() method returns the number of bytes written as a value type int.

A channel write() operation can throw any of five different exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonWritableChannelException</td>
<td>Thrown if the channel was not opened for writing.</td>
</tr>
<tr>
<td>ClosedChannelException</td>
<td>Thrown if the channel is closed. Calling the close() method for the file channel will close the channel, as will calling the close() method for the file stream.</td>
</tr>
<tr>
<td>AsynchronousCloseException</td>
<td>Thrown if another thread closes the channel while the write operation is in progress.</td>
</tr>
<tr>
<td>ClosedByInterruptException</td>
<td>Thrown if another thread interrupts the current thread while the write operation is in progress.</td>
</tr>
<tr>
<td>IOException</td>
<td>Thrown if some other I/O error occurs.</td>
</tr>
</tbody>
</table>

The first of these is a subclass of RuntimeException, so you do not have to catch this exception. The other four are subclasses of IOException, which must be caught, so the write() method call must be in a try block. If you want to react specifically to one or other of these last four exceptions, you will need to add a catch block for that specific type. Otherwise you can just include a single catch block for type IOException to catch all four types of exception. For instance if you have set up a ByteBuffer, but, ready to be written, you might code the write operation like this:

```java
File aFile = new File("C:/Beg Java Stuff/myFile.text");
```
Summary

In this chapter, we have discussed the facilities for checking out physical files and directories, and writing basic types of data to a file. The important points we have discussed include:

• An object of the class File can represent the path to a physical file.

• An object of type FileDescriptor can also represent a physical file.

• A FileOutputStream object can be created from a File object and the file will be opened for writing. If the file does not exist it will be created where possible.

• A FileChannel object for a file is returned by the getChannel() method for a file stream object.

• A buffer contains data to be written to a file, or data that has been read from a file. Only ByteBuffer objects can be used directly in file I/O operations.

• A buffer's position is the index position of the first element in the buffer to be written or read. A buffer's limit specifies the index position of the first element that is not to be written or read.

• A view buffer is a buffer that allows the data in a backing byte buffer to be viewed as being of a particular basic type.

• You insert data into a buffer using its put() methods and retrieve data from it using its get() methods. Relative get() and put() methods increment the buffer's position, whereas absolute get() and put() methods do not.

• You write the contents of a ByteBuffer object to a file using a write() method belonging to the FileChannel object for the file.
The amount of data transferred between a buffer and a file in an I/O operation is determined by the buffer's position and limit. Data is read or written starting at the file's current position.

You can lead a horse to water but you can't make him drink.
Exercises

1. Modify the example that writes proverbs to a file to separate the proverbs using a delimiter character. You will need to choose a delimiter character that will not appear in normal text.

2. Write a program that, using an integer array of date values containing month, day, and year as integers for some number of dates (10 say, so the int array will be two dimensional with 10 rows and 3 columns), will write a file with a string representation of each date written as Unicode characters. For example, the date values 3,2,1990 would be written to the file as 2nd March 1990. Make sure that the date strings can be read back, either by using a separator character of some kind to mark the end of each string, or write the length of each string before you write the string itself.

3. Extend the previous example to write a second file at the same time as the first, but containing the month, day, and year values as binary data. You should have both files open and be writing to both at the same time.

4. Write a program that, for a given String object defined in the code, will write strings to a file in the local character encoding (as bytes) corresponding to all possible permutations of the words in the string. For example, for the string, the fat cat, you would write the strings: the fat cat, the cat fat, cat the fat, cat fat the, fat the cat, fat cat the, to the file, although not necessarily in that sequence. (Don't use very long strings; with \( n \) words in the string, the number of permutations is \( n! \).)
Chapter 11: Reading Files

Overview

In this chapter we will investigate how we can read files containing basic types of data. We will be exploring how to read files sequentially or at random, and how we can open a file for both read and write operations.

In this chapter you will learn:

* How to obtain a file channel for reading a file.
* How to use buffers in file channel read operations.
* How to read different types of data from a file.
* How to retrieve data from random positions in a file.
* How you can read and write the same file.
* How you can do direct data transfer between channels.
* What a memory-mapped file is and how you can access a memory-mapped file.
* What a file lock is and how you can lock all or part of a file.
File Read Operations

The process for reading a file parallels that of writing a file. You obtain a FileChannel object from a file from a file stream, and use the channel to read data from the file into one or more buffers. Initially we will be using a channel object from a FileInputStream object to read a file. Later when we want to read and write the same file we will be using a FileChannel object obtained from a RandomAccessFile object. Like the FileOutputStream class, the FileInputStream class defines its own methods for file input, as does the RandomAccessFile class. However, we will completely ignore these because the file channel methods for reading the file are much more efficient and will eventually obsolete the stream methods. In any event, if you are curious to see how the old stream input mechanism works you can find details of the methods that read from a file stream in the descriptions for the FileInputStream and RandomAccessFile classes in the documentation that accompanies the SDK.

The starting point for reading a file is to create a FileInputStream object. Creating FileInputStream objects is not very different from creating FileOutputStream objects so let's look into how we do that first.

Important

In our examples in this chapter we will be reading some of the files that we created in the last chapter so I hope that you kept them.

Creating File Input Streams

Since a FileInputStream object encapsulates a file that is essentially intended to be read and therefore must contain some data, a constructor for this class can only create an object for a file that already exists. There are three constructors for FileInputStream objects, each of which takes a single argument.

First of all you can create a FileInputStream object from a String object that specifies the file name. For example:
File Channel Read Operations

You obtain a reference to a FileChannel object that you can use to read a file by calling the getChannel() method of a FileInputStream object. Since a FileInputStream object opens a file as read-only, only channel read operations are legal. The channel returned by a FileInputStream object has three basic read operations available, each of which read starting at the byte indicated by the current position in the file. The file position will be incremented by the number of bytes read. The three read() methods for a FileChannel object are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(ByteBuffer buf)</td>
<td>Tries to read buf.remaining() bytes (equivalent to limit-position bytes) from the file into the buffer, buf, starting at the buffer's current position. The number of bytes read is returned as type int, or -1 if the channel reaches the end-of-file. The buffer position will be incremented by the number of bytes read and the buffer's limit will be left unchanged.</td>
</tr>
<tr>
<td>read(ByteBuffer[] buffers)</td>
<td>Tries to read bytes into each of the buffers in the buffers array in sequence. Bytes will be read into each buffer starting at the point defined by that buffer's position. The number of bytes read into each buffer is defined by the remaining() method for that buffer. The read() method returns the total number of bytes read as type int, or -1 if the channel reaches the end-of-file. Each buffer's position will be incremented by the number of bytes read into it. Each buffer's limit will be unchanged.</td>
</tr>
<tr>
<td>read(ByteBuffer[] buffers, int offset, int length)</td>
<td>This operates in the same way as the previous method except that bytes are read starting with the buffer buffers[offset], and up to and including the buffer buffers[offset+length-1]. This method will throw an exception of type IndexOutOfBoundsException if offset or offset+length-1 are not valid index values for the array buffers.</td>
</tr>
</tbody>
</table>

As you can see, all three read() methods read data into one or more buffers of type ByteBuffer. Since you can only use ByteBuffer objects to receive the data read from the file, you can only read data from a file via a channel as a series of bytes. How you interpret these bytes afterwards though is up to you.

All three methods can throw exceptions of any of the following types:

<table>
<thead>
<tr>
<th>Exception Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonReadableChannelException</td>
<td>Thrown if the file was not opened for reading.</td>
</tr>
<tr>
<td>ClosedChannelException</td>
<td>Thrown if the channel is closed.</td>
</tr>
<tr>
<td>Exception</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AsynchronousCloseException</td>
<td>Thrown if the channel is closed by another thread while the read operation is in progress.</td>
</tr>
<tr>
<td>ClosedByInterruptException</td>
<td>Thrown if another thread interrupts the current thread while the read operation is in progress.</td>
</tr>
<tr>
<td>IOException</td>
<td>Thrown if some other I/O error occurs.</td>
</tr>
</tbody>
</table>

The first of these is a subclass of RuntimeException so you are not obliged to catch this exception. If you don't need to identify the other exceptions individually, you can use a single catch block for exceptions of type IOException to catch any of them.

The FileChannel object keeps track of the file's current position, and this is initially set to zero, corresponding to the first byte available from the file. Each read operation increments the channel's file position by the number of bytes read, so the next read operation will start at that point, assuming you have not modified the file position by some other means. When you need to change the file position in the channel to reread the file for instance you just call the position() method for the FileChannel object, with the index position of the byte where you want the next read to start as the argument to the method. For example, with a reference to a FileChannel object stored in a variable inChannel, you could reset the file position back to the beginning of the file with the statements:
Reading a Text File

We can now attempt to read the very first file that we wrote in the previous chapter charData.txt. We wrote this file as Unicode characters so we must take this into account when interpreting the contents of the file.

We can set up a File object encapsulating the file path to start with and create a FileInputStream object from that. We can then obtain a reference to the channel that we will use to read the file. We won't include all the checking here that you should apply to validate the file path, as you know how to do that:
Reading Binary Data

When you read binary data you still read bytes from the file, so the process is essentially the same as we used in the previous example. To read a binary file, we create a FileInputStream object and get the FileChannel object from it, then read the data into a byte buffer. We could set up a file channel to read our primes.bin file like this:
Reading Mixed Data

The primes.txt file that we created in the previous chapter contains data of three different types. We have the string length as a binary value of type double of all things, followed by the string itself describing the prime value, followed by the binary prime value as type long. Reading this file is a little trickier than it looks at first sight.

To start with we will set up the file input stream and obtain the channel for the file. Since, apart from the name of the file, this is exactly as in the previous example we won't repeat it here. Of course, the big problem is that we don't know ahead of time exactly how long the strings are. We have two strategies to deal with this:

- We can read the string length in the first read operation, then read the string and the binary prime value in the next. The only downside to this approach is that it's not a particularly efficient way to read the file as we will have read operations that each read a very small amount of data.

- We can set up a sizable byte buffer of an arbitrary capacity and just fill it with bytes from the file. We can then sort out what we have in the buffer. The problem with this approach is that the buffer's contents may well end part way through one of the data items from the file. We will have to do some work to detect this and figure out what to do next but this will be much more efficient than the first approach since we will vastly reduce the number of read operations that are necessary to read the entire file.

Let's try the first approach first as it's easier.

To read the string length we need a byte buffer with a capacity to hold a single value of type double:
Copying Files

You probably don't need a file copy program as your operating system is bound to provide a facility for this. However, it is a useful way of demonstrating how a file channel for any input file can transfer data directly to a file channel for an output file without involving explicit buffers.

A file channel defines two methods for direct data transfer:

| transferTo(long position, long count, WritableByteChannel dst) | Attempts to transfer count bytes from this channel to the channel, dst. Bytes are read from this channel starting at the file position specified by position. The position of this channel is not altered by this operation but the position of dst will be incremented by the number of bytes written. Fewer than count bytes will be transferred if this channel's file has fewer than count bytes remaining, or if dst is non-blocking and has fewer than count bytes free in its system output buffer. The number of bytes transferred is returned as type int. |
| transferFrom(ReadableByteChannel src, long position, long count) | Attempts to transfer count bytes to this channel from the channel src. Bytes are written to this channel starting at the file position specified by position. The position of this channel is not altered by the operation but the position of src will be incremented by the number of bytes read from it. If position is greater than the size of the file, then no bytes will be transferred. Fewer than count bytes will be transferred if the file corresponding to src has fewer than count bytes remaining in the file or if it is non-blocking and has fewer than count bytes free in its system input buffer. The number of bytes transferred is returned as type int. |

A channel that was obtained from a FileInputStream object will only support the transferTo() method. Similarly, a channel that was obtained from a FileOutputStream object will only support the transferFrom() method. Both of these methods can throw any of the following flurry of exceptions:

| IllegalArgumentException | Thrown if either count or position is negative. |
### Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonReadableChannelException</td>
<td>Thrown if the operation attempts to read from a channel that was not opened for reading.</td>
</tr>
<tr>
<td>NonWritableChannelException</td>
<td>Thrown if the operation attempts to write to a channel that was not opened for writing.</td>
</tr>
<tr>
<td>ClosedChannelException</td>
<td>Thrown if either channel involved in the operation is closed.</td>
</tr>
<tr>
<td>AsynchronousCloseException</td>
<td>Thrown if either channel is closed by another thread while the operation is in progress.</td>
</tr>
<tr>
<td>ClosedByInterruptException</td>
<td>Thrown if another thread interrupts the current thread while the operation is in progress.</td>
</tr>
<tr>
<td>IOException</td>
<td>Thrown if some other I/O error occurs.</td>
</tr>
</tbody>
</table>

The value of these methods lies in the potential for using the I/O capabilities of the underlying operating system directly. Where this is possible, the operation is likely to be much faster than copying from one file to another in a loop using the read() and write() methods we have seen.

A file copy program is an obvious candidate for trying out these methods.

### Try It Out  Direct Data Transfer between Channels

We will put together a program that will copy the file that is specified by a command line argument. We will copy the file to a backup file that we will create in the same directory as the original. We will create the name of the new file by appending "_backup" to the original file name as many times as necessary to form a unique file name. That operation is a good candidate for writing a helper method:
Random Access to a File

We can already read or write a file at random. The FileChannel class defines both a read() and a write() method that operate at a specified position in the file:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(ByteBuffer buf, long position)</td>
<td>Reads bytes from the file into buf in the same way as we have seen previously except that bytes are read starting at the file position specified by the second argument. The channel's position is not altered by this operation. If position is greater than the number of bytes in the file then no bytes are read.</td>
</tr>
<tr>
<td>write(ByteBuffer buf, long position)</td>
<td>Writes bytes from buf to the file in the same way as we have seen previously except that bytes are written starting at the file position specified by the second argument. The channel's position is not altered by this operation. If position is less than the number of bytes in the file then bytes from that point will be overwritten. If position is greater than the number of bytes in the file then the file size will be increased to this point before bytes are written. In this case the bytes between the original end-of-file and where the new bytes are written will contain junk values.</td>
</tr>
</tbody>
</table>

These methods can throw the same exceptions as the corresponding method accepting a single argument, plus they may throw an exception of type IllegalArgumentException if a negative file position is specified.

Let's see how we can access a file randomly using the read() method above.

**Try It Out  Reading a File Randomly**

To show how easy it is to read from random positions in a file, we will write an example to extract a random selection of values from our primes.bin file. Here's the code:
Read/Write Operations with a Single File Channel

If you want to be able to read and write a file using a single channel, you must use the channel provided by a RandomAccessFile object. A RandomAccessFile object is not related to the other file stream classes since its only base class is Object. Its original purpose was to provide random access to a file, which the other file stream classes could not, but as we have seen, this capability has been usurped by a channel anyway.

There are two constructors available for creating RandomAccessFile objects, and both require two arguments. For one constructor, the first argument is a File object that identifies the file path, and the second is a String object that specifies the access mode. The other constructor offer the alternative of using a String object as the first argument specifying the file path, with the second argument defining the access mode as before.

The access mode can be "r", which indicates that you just want to read the file, or it can be "rw", which indicates that you want to open the file to allow both read and write operations. If you specify the mode as anything else, the constructor will throw an IllegalArgumentException.

To create a RandomAccessFile object, you could write:
Memory-Mapped Files

A memory-mapped file is a file that has its contents mapped into an area of virtual memory in your computer so you can reference or update the data directly without performing any explicit file read or write operations on the physical file yourself. The memory that a file maps to may be paged in or out by the operating system, just like any other memory in your computer, so its immediate availability in real memory is not guaranteed. Because of the potentially immediate availability of the data it contains, a memory-mapped file is particularly useful when you need to access the file randomly. Your program code can reference data in the file as though it were resident in memory.

The map() method for a FileChannel object will return a reference to a buffer of type MappedByteBuffer that will map to a specified part of the channel's file:

```
map(int mode, long position, long size)
```

Maps a region of the channel's file to a buffer of type MappedByteBuffer. The file region that is mapped starts at position in the file and is of length size bytes. The first argument, mode, specifies how the buffer's memory may be accessed and can be any of the following three constant values, defined in the FileChannel class:

- **MAP_RO.** This is valid if the channel was opened for reading the file, in other words, if the channel was obtained from a FileInputStream object or a RandomAccessFile object. In this mode the buffer is read-only. If you try to modify the buffer's contents a ReadOnlyBufferException will be thrown.

- **MAP_RW.** This is valid if the channel was obtained from a RandomAccessFile object with "rw" as its access mode. You can access and change the contents of the buffer and any changes to the contents will eventually be propagated to the file.

- **MAP_COW.** The COW part of the name is for **Copy On Write.** This option for mode is also only valid if the channel was obtained from a RandomAccessFile object with "rw" as its access mode. You can access or change the buffer but changes will not be propagated to the file. Private copies of modified portions of the buffer will be created and used for subsequent buffer accesses.

Because the MappedByteBuffer class is a subclass of the ByteBuffer class, you have all the ByteBuffer methods available for a MappedByteBuffer object. This implies that you can create view buffers for a MappedByteBuffer object, for instance.

The MappedByteBuffer class defines three methods of its own to add to those inherited from the ByteBuffer class:
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>force()</code></td>
<td>If the buffer was mapped in MAP_RW mode this method forces any changes made to the buffer's contents to be written to the file and returns a reference to the buffer. For buffers created with MAP_RO or MAP_COW mode this method has no effect.</td>
</tr>
<tr>
<td><code>load()</code></td>
<td>Loads the contents of the buffer into memory and returns a reference to the buffer.</td>
</tr>
<tr>
<td><code>isLoaded()</code></td>
<td>Returns true if this buffer's contents is available in physical memory and false otherwise.</td>
</tr>
</tbody>
</table>

Using a memory-mapped file through a MappedByteBuffer is simplicity itself, so let's try it.

**Try It Out Using a Memory-Mapped File**

We will access and modify the primes.bin file using a MappedByteBuffer. Here's the code:
Summary

In this chapter, we have discussed the various ways in which we can read basic types of data from a file. The important points we have discussed include:

- You can read a file using a FileChannel object obtained from a FileInputStream object or from a RandomAccessFile object.

- You can use a channel obtained from a RandomAccessFile object that was created with mode "rw" to both read and write the file.

- Data that are read from a file using a channel is stored in one or more buffers of type ByteBuffer.

- You can use view buffers to interpret the data read from a file as any basic type other than boolean.

- A memory-mapped file enables you to access data in the file as though it were resident in memory.

- Acquiring an exclusive lock on a file ensures that no other program can access the file while you hold the lock.

- Acquiring a shared lock on a file ensures your program has access to the file in circumstances where other programs may be accessing the same file.

- If you can't ride two horses at once, you shouldn't be in the circus.
Exercises

1. Write a program to read back and list the contents of the file written by the first exercise in the previous chapter.

2. Extend the ReadPrimes example that we produced in this chapter to optionally display the nth prime, when n is entered from the keyboard.

3. Extend the ReadPrimes program further to output a given number of primes, starting at a given number. For example, output 15 primes starting at the 30th. The existing capabilities should be retained.

4. Write a program that will output the contents of a file to the command line as groups of eight hexadecimal digits with five groups to a line, each group separated from the next by a space.

5. Write a program that will allow either one or more names and addresses to be entered from the keyboard and appended to a file, or the contents of the file to be read back and output to the command line.

6. Modify the previous example to store an index to the name and address file in a separate file. The index file should contain each person's second name, plus the position where the corresponding name and address can be found in the name and address file. Provide support for an optional command argument allowing a person's second name to be entered. When the command line argument is present, the program should then find the name and address and output it to the command line.
Chapter 12: Serializing Objects

Overview

In this chapter, we will see how you can transfer objects to and from a stream. By the end of this chapter you will have learned:

• What serialization is and how you make a class serializable.

• How to write objects to a file.

• What transient fields in a class are.

• How to write basic types of data to an object file.

• How to implement the Serializable interface.

• How to read objects from a file.

• How to implement serialization for classes containing objects that are not serializable by default.
Storing Objects in a File

The process of storing and retrieving objects in an external file is called **serialization**. Note that an array of any type is an object for the purposes of serialization, even an array of values of a primitive type, such as type int or type double. Writing an object to a file is referred to as **serializing** the object, and reading an object from a file is called **deserializing** an object. Serialization is concerned with writing objects and the fields they contain to a stream, so this excludes static members of a class. Static fields will have whatever values are assigned by default in the class definition.

I think you will be surprised at how easy this is. Perhaps the most impressive aspect of the way serialization is implemented in Java is that you can generally read and write objects of almost any class type, including objects of classes that you have defined yourself, without adding any code to the classes involved to support this mechanism. For the most part, everything is taken care of automatically:

Two classes from the `java.io` package are used for serialization. An `ObjectOutputStream` object manages the writing of objects to a file, and reading the objects back is handled by an object of the class `ObjectInputStream`. As we saw in *Chapter 8*, these are derived from `OutputStream` and `InputStream`, respectively.

### Writing an Object to a File

The constructor for the `ObjectOutputStream` class requires a reference to a `FileOutputStream` object as an argument that defines the stream for the file where you intend to store your objects. You could create an `ObjectOutputStream` object with the following statements:
Summary

In this chapter we have explored how we can write objects to a file and read them back. Making your class serializable makes it very easy to save your application data in a file. While what we have discussed is by no means exhaustive, you now know enough to deal with straightforward object serialization. The important points in this chapter are:

- To make objects of a class serializable the class must implement the Serializable interface.

- If a class has a superclass that does not implement the Serializable interface, then the superclass must have a public default constructor if it is to be possible to serialize the class.

- Objects are written to a file using an ObjectOutputStream object and read from a file using and ObjectInputStream object.

- Objects are written to a file by calling the writeObject() method for the ObjectOutputStream object corresponding to the file.

- Objects are read from a file by calling the readObject() method for the ObjectInputStream object corresponding to the file.

- When necessary, for instance if a superclass is not serializable, you can implement the readObject() and writeObject() methods for your classes.

- A good horse cannot be of a bad color.
Exercises

1. Define a Person class to encapsulate a person's name and address with the name and address being fields of type Name and Address. Write a program to allow names and addresses to be entered from the keyboard and stored as Person objects in a file. Once the file exists new entries should be appended to the file.

2. Extend the previous example to optionally list all the names and addresses contained within the file on the command line.

3. Extend the previous example to add an index based on the person's name for each person entered at the keyboard to locate the corresponding Person object in the object file. The index file will contain entries of type IndexEntry each of which encapsulates a name and a file position in the object file. The index file should be a separate file from the original file containing Person objects.

   Note: You will probably find it easiest to delete the previous file before you run this example so that the object file can be reconstructed along with the index file. You can get and set the file position from the underlying file input stream object.

4. Use the index file to provide random direct access to the object file for querying random names entered from the keyboard. Entering a name from the keyboard should result in the address for the individual, or a message indicating the entry is not present in the file. The process will be to first search the index file for an object with a name field matching the keyboard entry. When an IndexEntry is found, you use the file position it contains to retrieve data from the file containing the Person objects.
Chapter 13: Collection Classes

Overview

In this chapter we'll look at the collection classes that are defined in the java.util package. These provide you with a variety of ways for structuring and managing collections of objects in your programs. In particular the collection classes enable you to deal with situations where you don't know in advance how many objects you'll need to store, or where you need a bit more flexibility in the way in which you access an object from a collection than the indexing mechanism provided by an array.

In this chapter you will learn:

• What sets, lists, and maps are, and how they work

• What an Iterator object is used for

• Which container classes are available

• What a Vector is and how to use Vector objects in your programs

• How to manage Vector objects so that storing and retrieving elements is type safe

• What a Stack is and how you use it

• What LinkedLists are and how you use them
How you store and retrieve objects in a hash table represented by a HashMap object

- How you can generate hash codes for your own class objects
Understanding the Collection Classes

The collection classes in java.util support various ways for you to store and manage objects of any kind in memory. They include a professional implementation of a linked list that we took so much trouble to develop for ourselves back in Chapter 6. If you want an array that automatically expands to accommodate however many objects you throw into it, or you need to be able to store and retrieve an object based on what it is, rather than using an index or a sequence number, then look no further. You get all this and more in the collection classes.

We'll be exploring the following capabilities provided by the package:

<table>
<thead>
<tr>
<th>Class/Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Iterator interface</td>
<td>Declares methods for iterating through elements of a set, one at a time.</td>
</tr>
<tr>
<td>The Vector class</td>
<td>Supports an array-like structure for storing any type of object. The number of objects that you can store in a Vector object increases automatically as necessary.</td>
</tr>
<tr>
<td>The Stack class</td>
<td>Supports the storage of any type of object in a push down stack.</td>
</tr>
<tr>
<td>The LinkedList class</td>
<td>Supports the storage of any type of object in a doubly-linked list.</td>
</tr>
<tr>
<td>The HashMap class</td>
<td>Supports the storage of any type of object in a hash table, sometimes called a map.</td>
</tr>
</tbody>
</table>

We'll start by looking in general terms at various possible types of collections for objects.
Collections of Objects

Back in Chapter 6 we put together a basic class defining a linked list. An object of type LinkedList represented an example of a collection of objects that could be of any type. A collection is used as a generic term for any object that represents a set of objects grouped together by some means. A linked list is only one of a number of ways of grouping a number of objects together in a collection.

There are three main types of collections: sets, sequences, and maps. Let's first get an understanding of how these three types of collections work in principle, and then come back to look at the Java classes that implement versions of these. One point I would like to emphasize about the following discussion is that when we talk about a collection of objects, we mean a collection of references to objects. In Java collections only stores references the objects themselves are external to the collection.

Sets

A set is probably the simplest kind of collection you can have. Here the objects are not usually ordered in any particular way at all and objects are simply added to the set without any control over where they go. It's a bit like putting things in your pocket you just put things in and they rattle around inside your pocket in no particular order.

The principal access mechanism that you have for a set is simply to check whether a given object is a member of the set or not. For this reason, you cannot have duplicate objects in a set each object in the set must be unique. Of course, you can also remove a given object from a set, but only if you know what the object is in the first place in other words if you have a reference to the object in the set.

There are variations on the basic set that we have described here. For instance, sets can be ordered, so objects added to a set will be inserted into a sequence of objects ordered according to some criterion of comparison. Such sets require that the class that defines the objects to be stored implements suitable methods for comparing objects.

Sequences
A linked list is an example of a more general type of collection called a sequence or a list. A primary characteristic of a list is that the objects are stored in a linear fashion, not necessarily in any particular order, with a beginning and an end. This contrasts with a set where there is no order at all. An ordinary array is basically another example of a list, but is much more limited than a collection because it is fixed.

Collections generally have the ability to expand to accommodate as many elements as necessary. The Vector class, for example, is a collection class that provides similar functionality to an array, but which also has this ability to accommodate new elements as and when required.

Because a list is linear, you will only be able to add a new object to the list at the beginning, or at the end, or inserted following a given object in sequence after the fifth say. Generally, you can retrieve an object from a list in several ways. You can select the first or the last; you can get the object at a given position as in indexing an array; or you can search for an object identical to a given object by checking all the objects in the sequence either backwards or forwards. You can also iterate through the list backwards or forwards accessing each object in sequence. We didn't implement all these capabilities in our linked list class in Chapter 6, but we could have done.

You can delete objects from a list in the same sorts of ways that you retrieve an object; that is, you can remove the first or the last, an object at a particular position in the sequence, or an object that is equal to a given object. Sequences or lists have the facility that they can store several copies of the same object at different places in the sequence. This is not true of all types of collections, as we will see.

A stack, which is a last-in first-out storage mechanism, is also considered to be a list, as is a queue, which is a first-in first-out mechanism. It is easy to see that a linked list can act as a stack, since using the methods to add and remove objects at the end of a list makes the list operate as a stack. Similarly, only adding objects by using the method to add an object to the end of a linked list, and only retrieving objects from the head of the list, makes it operate as a queue.

**Maps**

A map is rather different from a set or a sequence because the entries involve pairs of objects. A map is also referred to sometimes as a dictionary because of the way it works. Each object that is stored in a map has an associated key object, and both the object and its key are stored as a pair. The key determines where the object is stored in the map, and when you want to retrieve an object you must supply the appropriate key so it acts as the equivalent of a
word that you look up in a regular dictionary.

A key can be any kind of object that you want to use to reference the object stored. Because the key has to uniquely identify the object, all the keys in a map must be different. To put this in context let’s take an example. Suppose you were creating a program to provide an address book. You might store all the details of each person— their name, address, phone number, or whatever—in a single object of type Entry perhaps, and store a reference to the object in a map. The key is the mechanism for retrieving objects, so assuming that all names are different, the name of the person would be a natural choice for the key. Thus the entries in the map in this case would be Name/Entry pairs. You would supply a Name object as the key, and get back the Entry object corresponding to the key. You might well have another map in this application where entries were keyed on the phone number. Then you could retrieve an entry corresponding to a given number. Of course, in practice, names are not unique hence the invention of such delightful attachments to the person as social security numbers.

**Hashing**

Where a key/object pair is stored in a map is determined from the key by a process known as **hashing**. Hashing processes the key object to produce an integer value called a **hash code**. The hashCode() method that is defined in the Object class produces a hash code of type int for an object. The hash code is typically used as an offset from the start of the memory that has been allocated within the map, to determine the location where the key/object pair is to be stored. Ideally the hashing process should result in values that are uniformly distributed within a given range, and every key should produce a different hash code. In general, this may not be the case, but there are ways around this so it is not a problem. We will look at keys and hash codes in a little more detail when we discuss using maps later in this chapter.

Now let’s look at how we can move through a collection.
Iterators

In the LinkedList class that we developed in Chapter 6 you might have thought that the mechanism for getting the objects from the list was a little cumbersome. It was necessary to obtain the first element by using one method, getFirst(), and successive elements by another method, getNext(). This makes the first element in a list a 'special case' so processing the elements has to take account of this and is a little more complicated than perhaps it needs to be.

A much better approach that can be used to process the elements from a collection sequentially involves something called an iterator.

Note

It is worth noting at this point that Java also provides something called an enumerator. An enumerator provides essentially the same capability as an iterator, but it is recommended in the Java documentation that you should use an iterator in preference to an enumerator for collections.

In general an iterator is an object that you can use to retrieve all the objects in a collection one by one. Someone dealing cards from a deck one by one is acting as an iterator for the card deck without the shuffle, of course.

In Java, an iterator is an interface that can be implemented by a collection class. Any collection object can create an object of type Iterator that encapsulates references to all the objects in the original collection in some sequence, and that can be accessed using the Iterator interface methods. In other words an iterator provides an easy way to get at all the objects in a collection one at a time. The basic mechanism for using an iterator in Java is illustrated below.

![Diagram of Iterator usage]

The Iterator interface in java.util declares just three methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>next()</td>
<td>Returns an object as type Object starting with the first, and sets the Iterator object to return the next object on the next call of this method. If there is no object to be returned the method throws a NoSuchElementException exception.</td>
</tr>
<tr>
<td>hasNext()</td>
<td>Returns true if there is a next object to be retrieved by a call to next().</td>
</tr>
<tr>
<td>remove()</td>
<td>Removes the last object returned by next() from the collection that supplied the Iterator object. If next() has not been called or if you call remove() twice after calling next(), an IllegalStateException will be thrown. Not all iterators support this method, in which case an UnsupportedOperationException exception will be thrown if you call it.</td>
</tr>
</tbody>
</table>

Since calling the `next()` method for an object that implements Iterator returns successive objects from the collection, starting with the first, you can progress through all the objects in a collection very easily with a loop such as:
# Collection Classes

You have a total of thirteen classes in java.util that you can use to manage collections of objects, and they support collections that are sets, lists, or maps, as follows:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sets:</strong></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>An implementation of a set that uses HashMap under the covers. Although a set is by definition unordered, there has to be <em>some</em> way to find an object reasonably efficiently. The use of a HashMap object to implement the set enables store and retrieve operations to be done in a constant time.</td>
</tr>
<tr>
<td>TreeSet</td>
<td>An implementation of a set that orders the objects in the set in ascending sequence. This means that an iterator obtained from a TreeSet object will provide the objects in ascending sequence. The TreeSet classes use a TreeMap under the covers.</td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>Implements a set using a hash table with all the entries linked in a doubly-linked list. This class can be used to make a copy of any set such that iteration ordering is preserved—something that does not apply to a HashSet.</td>
</tr>
<tr>
<td><strong>Lists:</strong></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>Implements a list as an array that automatically increases in size to accommodate as many elements as you need. Objects are stored and retrieved using an index as in a normal array. You can also use an iterator to retrieve objects from a Vector. The Vector is the only container class that is synchronized—that is, it is well-behaved when concurrently accessed by two or more threads. We will discuss threads and synchronization in the next chapter.</td>
</tr>
<tr>
<td>Stack</td>
<td>This class is derived from Vector and adds methods to implement a stack—a last-in first-out storage mechanism.</td>
</tr>
<tr>
<td>LinkedList</td>
<td>Implements a linked list. The linked list defined by this class can also be used as a stack or a queue.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ArrayList</td>
<td>Implements an array that can vary in size and can also be accessed as a linked list. This provides a similar function to the Vector class but is unsynchronized.</td>
</tr>
<tr>
<td>Maps:</td>
<td></td>
</tr>
<tr>
<td>Hashtable</td>
<td>Implements a map where all keys must be non-null. The class defining a key must implement the hashcode() method and the equals() method to work effectively. This class is a legacy of previous Java implementations and it is usually better to use the other classes that implement maps.</td>
</tr>
<tr>
<td>HashMap</td>
<td>Implements a map that allows null objects to be stored and allows a key to be null (only one of course, since keys must be unique).</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>Implements a map with all of its entries in a doubly-linked list. This class can be used to create a copy of a map of any type such that the order of the entries in the copy is the same as the original.</td>
</tr>
<tr>
<td>WeakHashMap</td>
<td>Implements a map such that if a key to an object is no longer referenced ordinarily, the key/object pair will be discarded. This contrasts with HashMap where the presence of the key in the map maintains the life of the key/object pair, even though the program using the map no longer has a reference to the key, and therefore cannot retrieve the object.</td>
</tr>
<tr>
<td>IdentityHashMap</td>
<td>Implements a map using a hash table where comparisons in searching the map for a key or a value compares references, not objects. This implies that two keys are equal only if they are the same key. The same applies to values.</td>
</tr>
<tr>
<td>TreeMap</td>
<td>Implements a map such that the objects are arranged in ascending key order.</td>
</tr>
</tbody>
</table>

We can't go into all these classes in detail, but to introduce you to how these can be applied we will explore the three that you are likely to find most useful, Vector, LinkedList, and HashMap. Before we get into the specifics of using the container classes we need to look at the interfaces they implement, since these provide the means of applying them.

**Collection Interfaces**

The java.util package defines six collection interfaces that determine the methods that you use to work with each type of collection class. There are three basic collection interfaces, the Set, List, and Map interfaces, which relate to the fundamental organization of objects in a collection. These are implemented amongst the classes as follows:
The relationships between the interfaces that are implemented by the collection classes are shown in the following diagram.

The Set and List interfaces both extend a common interface, Collection. Note that the Map interface does not extend Collection. Don't confuse the Collection interface with the Collections class (with an 's') that we will see later. The two other interfaces for collections are SortedSet that extends the Set interface, and SortedMap that extends the Map interface. The TreeSet class implements the SortedSet interface, and the SortedMap interface is implemented by the TreeMap class.

It is important to keep in mind that any collection class object that implements the Collection interface can be referenced using a variable of type Collection. This means that any of the list or set collections can be referenced in this way; only the map class types are excluded (but not entirely, as you can obtain a list from a map and the classes implementing a map can provide a view of the values stored as a Collection reference). You will see that using a parameter of type Collection is a standard way of passing a list or set to a method.

These interfaces involve quite a number of methods, so rather than go through them in the abstract, let's see them at work in the context of specific classes. We will look at the Vector class first since it is close to the notion of an array that you are already familiar with.
Using Vectors

The Vector class defines a collection of elements of type Object that works rather like an array, but with the additional feature that it can grow itself automatically when you need more capacity. It implements the List interface so it can be used as a list. Because it stores elements of type Object, and Object is a superclass of every object, you can store any type of object in a Vector. This also means that potentially you can use a single Vector object to store objects that are instances of a variety of different classes. This is another advantage the Vector class has over arrays, but the circumstances where this is desirable are relatively rare.

This ability to store diverse objects has a downside. It implies that it's very easy for you to store objects in a Vector by mistake. You can set up a Vector in which you plan to store a particular kind of object, but there's nothing to prevent the storage of some other kind of object in the Vector, or to signal that this may cause problems. If you need to protect against this kind of error, you must program for it yourself. This isn't terribly difficult. As you'll see later in this chapter, all you need to do is package your Vector as a private member of a class that you define, and then supply methods to store objects in the Vector that will only accept the type that you want.

Important

Like arrays, vectors only hold object references, not actual objects. To keep things simple we refer to a Vector as holding objects. We'll make the distinction only when it's important, but you should keep in mind that all the collection classes you're about to encounter hold object references.

Creating a Vector

There are four constructors for a Vector. The default constructor creates an empty Vector object with the capacity to store up to a default number of objects, and the Vector object will increase in size each time you add an element when the Vector is full. The default capacity of a Vector object is ten objects, and the Vector object will double in size when you add an object when it is full. For example:
Linked Lists

The LinkedList collection class implements a generalized linked list. We have already seen quite a few of the methods that the class implements as the members of the List interface implemented in the Vector class. Nonetheless, let's quickly go through the methods that the LinkedList class implements. There are two constructors a default constructor that creates an empty list, and a constructor that accepts a Collection argument that will create a LinkedList object containing the objects from the collection that is passed to it.

To add objects to a list you have the add() and addAll() methods exactly as we discussed for a Vector object. You can also add an object at the beginning of a list using the addFirst() method, and you can add one at the end using addLast(). Both methods accept an argument of type Object and do not return a value. Of course, the addLast() method provides the same function as the add() method.

To retrieve an object at a particular index position in the list you can use the get() method, as in the Vector class. You can also obtain references to the first and last objects in the list by using the getFirst() and getLast() methods, respectively. To remove an object you can use the remove() method with an argument that is either an index value or a reference to the object that is to be removed. The removeFirst() and removeLast() methods do what you would expect.

Replacing an existing element in the list at a given index position is achieved by using the set() method. The first argument is the index value and the second argument is the new object at that position. The old object is returned and the method will throw an IndexOutOfBoundsException if the index value is not within the limits of the list. The size() method returns the number of elements in the list.

As with a Vector object, you can obtain an Iterator object by calling iterator(), and you can obtain a ListIterator object by calling listIterator(). You will recall that an Iterator object only allows you to go forward through the elements, whereas a ListIterator enables you to iterate backwards or forwards.

We could change the TryPolyLine example from Chapter 6 to use a LinkedList collection object rather than our homemade version.

Try It Out Using a Genuine Linked List

We will put this example in a new directory, TryNewPolyLine. We can use the TryPolyLine class that contains main() and the Point class exactly as they are, so if you still have them, copy the source files to the new directory. We just need to change the PolyLine class definition:

```
import java.util.*;
```
Using Maps

As we saw at the beginning of this chapter, a map is a way of storing data that minimizes the need for searching when you want to retrieve an object. Each object is associated with a key that is used to determine where to store the reference to the object, and both the key and the object are stored in the map. Given a key, you can always go more or less directly to the object that has been stored in the map based on the key. It's important to understand a bit more about how the storage mechanism works for a map, and in particular what the implications of using the default hashing process are. We will explore the use of maps primarily in the context of the HashMap class.

The Hashing Process

A map sets aside an array in which it will store key and object pairs. The index to this array is produced from the key object by using the hash code for the object to compute an offset into the array for storing key/object pairs. By default, this uses the hashCode() method for the object that's used as a key. This is inherited in all classes from Object.

Note that, while every key must be unique, each key doesn't have to result in a unique hash code. When two or more different keys produce the same hash value, it's called a collision. A HashMap object deals with collisions by storing all the key/object pairs that have the same hash value in a linked list. If this occurs very often, it is obviously going to slow up the process of storing and retrieving data. Retrieving an object that resulted in a collision when it was stored will be a two-stage process. The key will be hashed to find the location where the key/object pair should be. The linked-list will then have to be searched to sort out the particular key we are searching on from all the others that have the same hash value.

There is therefore a strong incentive to minimize collisions and the price of reducing the possibility of collisions in a hash table is having plenty of empty space in the table.

The class Object defines the method hashCode() so any object can be used as a key and it will hash by default. The method as it is implemented in Object in Java, however, isn't a panacea. Since it usually uses the memory address where an object is stored to produce the hash value, distinct objects will always produce different hash values. In one sense this is a plus, because the more likely it is that a unique hash value will be produced for each key, the more efficient the operation of the hash map is going to be. The downside is that different object instances that have identical data will produce different hash values, so you can't compare them.
This becomes a nuisance if you use the default hashCode() method in objects that you're using as keys. In this case, an object stored in a hash map can never be retrieved using a different key object instance, even though that key object may be identical in all other respects. Yet this is precisely what you'll want to do in many cases.

Consider an application such as a simple address book. You might store map entries keyed on the names of the people to whom the entries relate, and you would want to search the map based on a name that was entered from the keyboard. However, the object representing the newly entered name is inevitably going to be distinct from that used as a key for the entry. Using the former, you will not be able to find the entry corresponding to the name.

The solution to this problem is somehow to make a hash of the instance variables of the object. Then, by comparing the values of the data members of the new name object with those for the name objects used as keys in the hash map, you'll be able to make a match.

### Using Your Own Class Objects as Keys

For objects of one of your own classes to be usable as keys in a hash table, you must override the equals() method of the Object class. In its default form, equals() accepts an object of the same class as an argument and returns a boolean value. The equals() method is used by methods in the HashMap class to determine when two keys are equal, so, in order to enable the changes discussed in the [previous section](#), your version of this method should return true when two different objects contain identical data values.

You can also override the default hashCode() method, which returns the hash value for the object as type int. The hashCode() method is used to generate the int value that is the key. Your hashCode() method should produce hash codes that are reasonably uniform over the possible range of keys and generally unique for each key.

### Generating Hash Codes

The various techniques for generating hash codes form a big topic, and we can only scratch the surface here. How you write the hashCode() method for your class is up to you, but it needs to meet certain requirements if it is to be effective. You should aim to return a number of type int for an object that has a strong probability of being unique to that object, and the numbers that you produce for several different objects should be as widely distributed across the range of int values as possible.

To achieve the uniqueness, you will typically want to combine the values of all the data members in an object to produce the hash code, so the first step is to produce an integer corresponding to each data member. You must then combine these integers to generate the return value that will be the hash code for the object. One technique you can use to do this is to multiply each of the integers corresponding to the data members by a different prime number and then sum the results. This should produce a reasonable distribution of values that have a good probability of being different for different objects. It doesn't matter which prime numbers you use as multipliers, as long as:

- They aren't so large as to cause the result to fall outside the range of type int
- You use a different one for each data member
So how do you get from a data member of a class to an integer? Generating an integer for data members of type `String` is easy: you just call the `hashCode()` method for the member. This has been implemented in the `String` class to produce good hash code values that will be the same for identical strings (take a look at the source code if you want to see how). You can use integer data members as they are, but floating point data members need a bit of judgment. If they have a small range in integer terms, you need to multiply them by a value that's going to result in a unique integer when they are cast to type `int`. If they have a very large range in integer terms you may need to scale them down.

Suppose you intended to use a `Person` object as a key in a hash table, and the class data members were `firstName` and `surname` of type `String`, and `age` of type `int`. You could implement the `hashCode()` method for the class as:
Summary

All of the classes in this chapter will be useful sooner or later when you're writing your own Java programs. We'll be applying many of them in examples throughout the remainder of the book.

The important elements we've covered are:

• You can use a Vector object as a kind of flexible array that expands automatically to accommodate any number of objects stored.

• The Stack class is derived from the Vector class and implements a pushdown stack.

• The HashMap class defines a hash map in which objects are stored based on an associated key.

• An Iterator is an interface for retrieving objects from a collection sequentially. An Iterator object allows you to access all the objects it contains serially but only once. There's no way to go back to the beginning.

• The ListIterator interface provides methods for traversing the objects in a collection backwards or forwards.

• Objects stored in any type of collection can be accessed using Iterator objects.

• Objects stored in a Vector, a Stack, or a LinkedList can be accessed using ListIterator objects.
Exercises

1. Implement a version of the program to calculate prime numbers that we saw in Chapter 4 to use a Vector object instead of an array to store the primes. (Hint: remember the Integer class.)

2. Write a program to store a deck of 52 cards in a linked list in random sequence using a Random class object. You can represent a card as a two character string: "1C" for the ace of clubs, "JD" for the jack of diamonds, and so on. Output the cards from the stack as four hands of 13 cards.

3. Extend the program from the chapter that used a map to store names and telephone numbers such that you can enter a number to retrieve the name.

4. Implement a phone book so that just a surname can be used to search and have all the entries corresponding to the name display.
Chapter 14: A Collection of Useful Classes

Overview

In this chapter we'll be looking at some more very useful classes in the java.util package but this time they are not collection classes, just a collection of classes. We will also be looking at the facilities provided by classes in the java.util.regex package that implements regular expressions in Java. Support for regular expressions is a very powerful and important addition to Java.

In this chapter you will learn:

- How to use the static methods in the Arrays class for filling, comparing, sorting, and searching arrays

- How to use the Observable class and the Observer interface to communicate between objects

- What facilities the Random class provides

- How to create and use Date and Calendar objects

- What regular expressions are and how you can create and use them
Utility Methods for Arrays

The `Arrays` class in `java.util` provides you with a set of static methods for operating on arrays. There are methods for sorting and searching arrays, as well as methods for comparing two arrays of elements of a basic type. You also have methods for filling arrays of elements with a given value. Let's look at the simplest method first, the `fill()` method for filling an array.

Filling an Array

The need to fill an array with a specific value arises quite often. The `fill()` method comes in a number of overloaded versions of the form:

```
fill(type [] array, type value)
```

Here `type` is a placeholder for the types supported by various versions of the method. The method stores value in each element of array. The return type is void so there is no return value. There are versions supporting type as any of the following:

<table>
<thead>
<tr>
<th>boolean</th>
<th>byte</th>
<th>char</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>int</td>
<td>long</td>
<td>Object</td>
<td></td>
</tr>
</tbody>
</table>

The version of `fill()` accepting an array argument of type `Object[]` will obviously process an array of any class type.

Here's how you could fill an array of integers with a particular value:
Observable and Observer Objects

The class Observable provides you with an interesting mechanism for communicating a change in one class object to a number of other class objects. One use for this mechanism is in GUI programming where you often have one object representing all the data for the application—a text document, for instance, or a geometric model of a physical object and several other objects that represent views of the data that are displayed in separate windows, where each shows a different representation or perhaps a subset of the data. This is referred to as the document/view architecture for an application, or sometimes the model/view architecture. This is a contraction of something referred to as the model/view/controller architecture and we will come back to this when we discuss creating Graphical User Interfaces (GUI). The document/view terminology is applied to any collection of application data—geometry, bitmaps, or whatever. It isn't restricted to what is normally understood by the term 'document'.

When the Document object changes, all the views need to be notified that a change has occurred, since they may well need to update what they display. The document is observable and all the views are observers. This is exactly what the Observable class is designed to achieve, when used in combination with an interface, Observer. A document can be considered to be an Observable object, and a view can be thought of as an Observer object. This enables the view to respond to changes in the document.

The document/view architecture portrays a many-to-many relationship. A document may have many observers, and a view may observe many documents.

Defining Classes of Observable Objects

You use the Observable class in the definition of a class of objects that may be observed. You simply derive the class for objects to be monitored, Document say, from the class Observable.

Any class that may need to be notified when a Document object has been changed must implement the interface Observer. This doesn't in itself cause the Observer objects to be notified when a change in an observed object occurs; it just establishes the potential for this to happen. You need to do something else to link the observers to the observable, which we'll come to in a moment.

The definition of the class for observed objects could be of the form:
Generating Random Numbers

We have already used the Random class a little, but let's investigate this in more detail. The class Random enables you to create multiple random number generators that are independent of one another. Each object of the class is a separate random number generator. Any Random object can generate pseudo-random numbers of types int, long, float, or double. These numbers are created using an algorithm that takes a 'seed' and 'grows' a sequence of numbers from it. Initializing the algorithm twice with the same seed would produce the same sequence because the algorithm is deterministic.

The integer values generated will be uniformly distributed over the complete range for the type, and the floating point values will be uniformly distributed over the range 0.0 to 1.0 for both types. You can also generate numbers of type double with a Gaussian (or normal) distribution that has a mean of 0.0 and a standard deviation of 1.0. This is the typical bell-shaped curve that represents the probability distribution for many random events.

There are two constructors for a Random object. The default constructor will create an object that uses the current time from your computer clock as the seed value for generating pseudo-random numbers. The other constructor accepts an argument of type long that will be used as the seed.
Dates and Times

There are quite a few classes in the java.util package that are involved with dates and times, including the Date class, the Calendar class, and the GregorianCalendar class. In spite of the class name, a Date class object actually defines a particular instant in time to the nearest millisecond, measured from January 1, 1970, 00:00:00 GMT. Since it is relative to a particular instant in time, it also corresponds to a date. The Calendar class is the base class for GregorianCalendar, which represents the sort of day/month/year calendar everybody is used to, and also provides methods for obtaining day, month, and year information from a Date object. A Calendar object is always set to a particular date — a particular instant on a particular date to be precise — but you can change it by various means. From this standpoint a GregorianCalendar object is more like one of those desk calendars that just show one date, and you can flip over the days, months, or years to show another date.

There is also the TimeZone class that defines a time zone that can be used in conjunction with a calendar, and that you can use to specify the rules for clock changes due to daylight saving time. The ramifications of handling dates and times are immense so we will only be able to dabble here, but at least you will get the basic ideas. Let’s take a look at Date objects first.

The Date Class

With the Date class you can create an object that represents a given date and time. You have two ways to do this using the following constructors:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date()</td>
<td>Creates an object based on the current time from your computer clock to the nearest millisecond.</td>
</tr>
<tr>
<td>Date(long time)</td>
<td>Creates an object based on the time value in milliseconds since 00:00:00 GMT on January 1, 1970 that is passed as an argument.</td>
</tr>
</tbody>
</table>

With either constructor you create a Date object that represents a specific instant in time to the nearest millisecond. Carrying dates around as the number of milliseconds since the dawn of the year 1970 won't grab you as being incredibly user-friendly but we'll come back to how we can interpret a Date object better in a moment. The Date class provides three methods for comparing Date objects:

<table>
<thead>
<tr>
<th>Comparison Methods</th>
<th>Description</th>
</tr>
</thead>
</table>


after (Date earlier)  Returns true if the current object represents a date that's later than the date represented by the argument earlier, and false otherwise.

before (Date later)  Returns true if the current object represents a date that's earlier than the date represented by the argument later, and false otherwise.

equals (Object aDate)  Returns true if the current object and the argument represent the same date and time, and false otherwise. This implies that they would both return the same value from getTime().

The equals() method returns true if two different Date objects represent the same date and time. Since the hashCode() method is also implemented for the class, you have all you need to use Date objects as keys in a hash table.

Interpreting Date Objects

The DateFormat class is an abstract class that you can use to create meaningful String representations of Date objects. It isn't in the java.util package though it's defined in the package java.text. There are four standard representations for the date and the time that are identified by constants defined in the DateFormat class. The effects of these will vary in different countries, because the representation for the date and the time will reflect the conventions of those countries. The constants in the DateFormat class defining the four formats are:

<table>
<thead>
<tr>
<th>Date Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHORT</td>
<td>A completely numeric representation for a date or a time, such as 2/2/97 or 4:15 am.</td>
</tr>
<tr>
<td>MEDIUM</td>
<td>A longer representation than SHORT, such as 5-Dec-97.</td>
</tr>
<tr>
<td>LONG</td>
<td>A longer representation than MEDIUM, such as December 5, 1997.</td>
</tr>
<tr>
<td>FULL</td>
<td>A comprehensive representation of the date or the time such as Friday, December 5, 1997 AD or 4:45:52 PST (Pacific Standard Time).</td>
</tr>
</tbody>
</table>

A Locale object identifies information that is specific to a country, a region, or a language. You can define a Locale object for a specific country, for a specific language, for a country and a language, or for a country and a language and a variant, the latter being a vendor or browser specific code such as WIN or MAC. When you are creating a Locale object you use ISO codes to specify the language and/or the country. The language codes are defined by ISO-639. Countries are specified by the country codes in the standard ISO-3166. You can find the country codes on the Internet at:
You can find the language codes at:


or at:

http://lcweb.loc.gov/standards/iso639-2/langhome.html

For some countries, the easiest way to specify the locale, if you don't have the ISO codes on the tip of your tongue, is to use the Locale objects defined within the Locale class. In Java 2 these are:

<table>
<thead>
<tr>
<th>US</th>
<th>CANADA</th>
<th>CANADA_FRENCH</th>
<th>PRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>UK</td>
<td>GERMANY</td>
<td>FRANCE</td>
<td>ITALY</td>
</tr>
<tr>
<td>JAPAN</td>
<td>KOREA</td>
<td>CHINA</td>
<td>TAIWAN</td>
</tr>
</tbody>
</table>

Because the DateFormat class is abstract, you can't create objects of the class directly, but you can obtain DateFormat objects by using any of the following static methods, each of which returns a value of type DateFormat:

<table>
<thead>
<tr>
<th>Static Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getTimeInstance()</td>
<td>Returns a time formatter for the default locale that uses the default style for the time.</td>
</tr>
<tr>
<td>getTimeInstance (int timeStyle)</td>
<td>Returns a time formatter for the default locale that uses the style for the time that is specified by the argument.</td>
</tr>
<tr>
<td>getTimeInstance (int style, Locale aLocale)</td>
<td>Returns a time formatter for the locale specified by the second argument that uses the style for the time that is specified by the first argument.</td>
</tr>
<tr>
<td>getDateInstance()</td>
<td>Returns a date formatter for the default locale that uses the default style for the date.</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getDateInstance (int dateStyle)</td>
<td>Returns a date formatter for the default locale that uses the style for the date specified by the argument.</td>
</tr>
<tr>
<td>getDateInstance (int dateStyle, Locale aLocale)</td>
<td>Returns a date formatter for the locale specified by the second argument that uses the style for the date that is specified by the first argument.</td>
</tr>
<tr>
<td>getInstance()</td>
<td>Returns a default date and time formatter that uses the SHORT style for both the date and the time.</td>
</tr>
<tr>
<td>getDateTimeInstance()</td>
<td>Returns a date and time formatter for the default locale that uses the default style for both the date and the time.</td>
</tr>
<tr>
<td>getDateTimeInstance (int dateStyle, int timeStyle)</td>
<td>Returns a date and time formatter for the current locale that uses the styles for the date and the time specified by the arguments.</td>
</tr>
<tr>
<td>getDateTimeInstance (int dateStyle, int timeStyle, Locale aLocale)</td>
<td>Returns a date and time formatter for aLocale with the styles for the date and the time as specified by the first two arguments.</td>
</tr>
</tbody>
</table>

When you've obtained a DateFormat object for the country and the style that you want, and the sort of data you want to format the date or the time or both you're ready to produce a String from the Date object.

All you need to do is to pass the Date object to the format() method for the DateFormat object. For example:
Regular Expressions

We have seen some elementary capability for searching strings when we discussed the String class back in Chapter 4. From Java 1.4, we have had much more sophisticated facilities for analyzing strings by searching for patterns known as regular expressions. Regular expressions are not unique to Java. Perl is perhaps better known for its support of regular expressions, many word processors, especially on Unix, and there are specific utilities for regular expressions too.

So what is a regular expression? A regular expression is simply a string that describes a pattern that is to be used to search for matches within some other string. It's not simply a passive sequence of characters to be matched, though. A regular expression is essentially a mini-program for a specialized kind of computer called a state-machine. This isn't a real machine but a piece of software specifically designed to interpret a regular expression and analyze a given string based on that.

The regular expression capability in Java is implemented through two classes in the java.util.regex package: the Pattern class that defines objects that encapsulate regular expressions, and the Matcher class that defines an object that encapsulates a state-machine that can search a particular string using a given Pattern object. The java.util.regex package also defines the PatternSyntaxException class that defines exception objects thrown when a syntax error is found when compiling a regular expression to create a Pattern object.

Using regular expressions in Java is basically very simple:

1. You create a Pattern object by passing a string containing a regular expression to the static compile() method in the Pattern class.

2. You then obtain a Matcher object, which can search a given string for the pattern, by calling the matcher() method for the Pattern object with the string that is to be searched as the argument.

3. You call the find() method (or some other methods as we shall see) for the Matcher object to search the string.

4. If the pattern is found, you query the matcher object to discover the whereabouts of the pattern in the string and other information relating to the match.

While this is a straightforward process that is easy to code, the hard work is in defining the pattern to achieve the result that you want. This is an extensive topic since in their full glory regular expressions are immensely powerful and can get very complicated. There are books devoted entirely to this so our aim will be to get enough of a bare bones understanding of how regular expressions work so you will be in a good position to look into the subject in more depth if you need to. Although regular expressions can look quite fearsome, don't be put off. They are always built step-by-step, so although the end result may look complicated and obscure, they are not at all difficult to put together. Regular expressions are a lot of fun and a sure way to impress your friends and maybe confound your enemies.
Defining Regular Expressions

You may not have heard of regular expressions before reading this book and therefore may think you have never used them. If so, you are almost certainly wrong. Whenever you search a directory for files of a particular type, "*.java" for instance, you are using a form of regular expression. However, to say that regular expressions can do much more than this is something of an understatement. To get an understanding of what we can do with regular expressions, we will start at the bottom with the simplest kind of operation and work our way up to some of the more complex problems they can solve.

Creating a Pattern

In its most elementary form, a regular expression just does a simple search for a substring. For example, if we want to search a string for the word had, the regular expression is exactly that. So the string defining this particular regular expression is "had". Let's use this as a vehicle for understanding the programming mechanism for using regular expressions. We can create a Pattern object for our expression "had" with the statement:
Summary

Regular expressions are a very powerful capability that we only touched on in this chapter.

The important elements we've covered are:

- The java.util.Arrays class provides static methods for sorting, searching, filling, and comparing arrays.

- Objects of type Random can generate pseudo-random numbers of type int, long, float, and double. The integers are uniformly distributed across the range of the type int or long. The floating point numbers are between 0.0 and 1.0. You can also generate numbers of type double with a Gaussian distribution with a mean of 0.0 and a standard deviation of 1.0, and random boolean values.

- Classes derived from the Observable class can signal changes to classes that implement the Observer interface. You define the Observer objects that are to be associated with an Observable class object by calling the addObserver() method. This is primarily intended to be used to implement the document/view architecture for applications in a GUI environment.

- You can create Date objects to represent a date and time that you specify in milliseconds since January 1, 1970, 00:00:00 GMT, or the current date and time from your computer clock.

- You can use a DateFormat object to format the date and time for a Date object as a string. The format will be determined by the style and the locale that you specify.

- A GregorianCalendar object represents a calendar set to an instant in time on a given date.

- A regular expression defines a pattern that is used for searching text.

- In Java a regular expression is compiled into a Pattern object that you can then use to obtain a Matcher object that will scan a given string looking for the pattern.
The `appendReplacement()` method for a Matcher object enables you to make substitutions for patterns found in the input text.

- A capturing group in a regular expression records the text that matches a sub-pattern.

- By using capturing groups you can rearrange the sequence of substrings in a string matching a pattern.
Exercises

1. Define a static method to fill an array of type char[] with a given value passed as an argument to the method.

2. For the adventurous gambler use a stack and a Random object in a program to simulate a game of Blackjack for one player using two decks of cards.

3. Write a program to display the sign of the Zodiac corresponding to a birth date entered through the keyboard.

4. Write a program using regular expressions to remove spaces from the beginning and end of each line in a file.

5. Write a program using a regular expression to reproduce a file with a sequential line number starting at "0001" inserted at the beginning of each line in the original file. You can use a copy of your Java source file as the input to test this.

6. Write a program using a regular expression to eliminate any line numbers that appear at the beginning of lines in a file. You can use the output from the previous exercise as a test for your program.
Chapter 15: Threads

Overview

In this chapter we'll investigate the facilities Java has to enable you to overlap the execution of segments of a single program. As well as ensuring your programs run more efficiently, this capability is particularly useful when your program must, of necessity, do a number of things at the same time: for example, a server program on a network that needs to communicate with multiple clients.

In this chapter you will learn:

• What a thread is and how you can create threads in your programs.

• How to control interactions between threads.

• What synchronization means and how to apply it in your code.

• What deadlocks are, and how to avoid them.

• How to set thread priorities.

• How to get information about the threads in your programs.
Understanding Threads

Many programs, of any size, contain some code segments that are more or less independent of one another, and that may execute more efficiently if the code segments could be overlapped in time. Threads provide a way to do this. Of course, if like most people your computer only has one processor, you can't execute more than one computation at any instant, but you can overlap input/output operations with processing.

Another reason for using threads is to allow processes in a program that need to run continuously, such as a continuously running animation, to be overlapped with other activities in the same program. Java applets in a web page are executed under the control of your browser, and threads make it possible for multiple applets to be executing concurrently. If you only have one processor, this is an illusion created by your operating system since only one thread can actually be executing instructions at any given instant, but it's a very effective illusion. To produce animation, you typically put some code that draws a succession of still pictures in a loop that runs indefinitely.

The code to draw the picture generally runs under the control of a timer so that it executes at a fixed rate, for example, 20 times per second. Of course, nothing else can happen in the same thread while the loop is running. If you want to have another animation running, it must be in a separate thread. Then the multitasking capability of your operating system can allow the two threads to share the available processor time, thus allowing both animations to run.

Let's get an idea of the principles behind how threads operate. Consider a very simple program that consists of three activities:

- Reads a number of blocks of data from a file
- Performs some calculation on each block of data
- Writes the results of the calculation to another file

You could organize the program as a single sequence of activities. In this case the activities read file, process, write file run in sequence, and the sequence is repeated for each block to be read and processed. You could also organize the program so that reading a block from the file is one activity, performing the calculation is a second activity, and writing the results is a third activity. Both of these situations are illustrated below.
Once a block of data has been read, the computation process can start, and as soon as the computation has been completed, the results can be written out. With the program executing each step in sequence (that is, as a single thread), as shown in the top half of the diagram, the total time for execution will be the sum of the times for each of the individual activities. However, suppose we were able to execute each of the activities independently, as illustrated in the lower half of the diagram. In this case, reading the second block of data can start as soon as the first block has been read, and in theory we can have all three activities executing concurrently. This is possible even though you only have one processor, because the input and output operations are likely to require relatively little processor time while they are executing, so the processor can be doing other things while they are in progress. This can have the effect of reducing the total execution time for the program.

These three processes that we have identified that run more or less independently of one another— one to read the file, another to process the data, and a third to write the results— are called threads. Of course, the first example at the top of the diagram has just one thread that does everything in sequence. Every Java program has at least one thread. However, the three threads in the lower example aren’t completely independent of one another. After all, if they were, you might as well make them independent programs. There are practical limitations too   the potential for overlapping these threads will be dependent on the capabilities of your computer, and of your operating system. However, if you can get some overlap in the execution of the threads, the program is going to run faster. There's no magic in using threads, though. Your computer has only a finite capacity for executing instructions, and if you have many threads running you may in fact increase the overall execution time because of the overhead implicit in managing the switching of control between threads.

An important consideration when you have a single program running as multiple threads is that the threads are unlikely to have identical execution times, and, if one thread is dependent on another, you can't afford to have one overtaking the other   otherwise you'll have chaos. Before you can start calculating in the example in the diagram, you need to be sure that the data block the calculation uses has been read, and before you can write the output, you need to know that the calculation is complete. This necessitates having some means for the threads to communicate with one another.

The way we have shown the threads executing in the previous diagram isn't the only way of organizing the program. You could have three threads, each of which reads the file, calculates the results, and writes the output, as shown here.
Now there's a different sort of contention between the threads. They are all competing to read the file and write the results, so there needs to be some way of preventing one thread from getting at the input file while another thread is already reading from it. The same goes for the output file. There's another aspect of this arrangement that is different from the previous version. If one thread, thread1 say, reads a block, block4 perhaps, that needs a lot of time to compute the results, another thread, thread2 say, could conceivably read a following block, block5 maybe, and calculate and write the results for block5 before thread1 has written the results for block4. If you don't want the results appearing in a different sequence from the input, you should do something about this. Before we delve into the intricacies of making sure our threads don't get knotted, let's first look at how we create a thread.

Creating Threads

Your program always has at least one thread: the one created when the program begins execution. With a program, this thread starts at the beginning of main(). With an applet, the browser is the main thread. That means that when your program creates a thread, it is in addition to the main thread of execution that created it. As you might have guessed, creating an additional thread involves using an object of a class, and the class you use is java.lang.Thread. Each additional thread that your program creates is represented by an object of the class Thread, or of a subclass of Thread. If your program is to have three additional threads, you will need to create three such objects.

To start the execution of a thread, you call the start() method for the Thread object. The code that executes in a new thread is always a method called run(), which is public, accepts no arguments, and doesn't return a value. Threads other than the main thread in a program always start in the run() method for the object that represents the thread. A program that creates three threads is illustrated diagrammatically here:

For a class representing a thread in your program to do anything, you must implement the run() method as the version defined in the Thread class does nothing. Your implementation of run() can call any other methods you want. Our illustration shows main() creating all three threads, but that doesn't have to be the case. Any thread can create more threads.

Now here comes the bite; you don't call the run() method to start a thread, you call the start() method for the object representing the thread and that causes the run() method to be called. When you want to stop the execution of a thread that is running, you call the stop() method for the Thread object.

The reason for this is somewhat complex but basically boils down to this: Threads are always owned and managed by the operating system and a new thread can only be created and started by the operating system. If you were to call the run() method yourself, it would simply operate like any other method, running in the same thread as the program that calls it.

When you call the start() method for a Thread object, you are calling a native code method that causes the operating system to initiate another thread from which the run() method for the Thread object executes.
In any case, it is not important to understand exactly how this works. Just remember: always start your thread by calling the `start()` method. If you try to call the `run()` method directly yourself, then you will not have created a new thread and your program will not work as you intended.

There are two ways in which you can define a class that is to represent a thread. One way is to define your class as a subclass of `Thread` and provide a definition of the method `run()` that overrides the inherited method. The other possibility is to define your class as implementing the interface `Runnable`, which declares the method `run()`, and then create a `Thread` object in your class when you need it. We will look at and explore the advantages of each approach in a little more detail.

Try It Out  Deriving a Subclass of Thread

We can see how this works by using an example. We'll define a single class, `TryThread`, which we'll derive from `Thread`. Execution starts in the method `main()`:
Managing Threads

In both the examples we've seen in this chapter, the threads are launched and then left to compete for computer resources. Because all three threads compete in an uncontrolled way for the processor, the output from the threads gets muddled. This isn't normally a desirable feature in a program. In most instances where you use threads, the way in which they execute will need to be managed so that they don't interfere with each other.

Of course, in our examples, the programs are deliberately constructed to release control of the processor part way through outputting a name. While this is very artificial, similar situations can arise in practice, particularly where threads are involved in a repetitive operation. It is important to appreciate that a thread can be interrupted while a source statement is executing. For instance, imagine that a bank teller is crediting a check to an account and at the same time the customer with that account is withdrawing some cash through an ATM machine. This might happen in the following way:

- The bank teller checks the balance of the customer's account, which is $500.
- The ATM machine asks for the account balance.
- The teller adds the value of the check, $100, to the account balance to give a figure of $600.
- The ATM machine takes $50 off the balance of $500, which gives a figure of $450, and spits out 5 $10 bills.
- The teller assigns the value of $600 to the account balance.
- The ATM machine assigns the value $450 to the account balance.

Here you can see the problem very well. Asking the account for its balance and assigning a new balance to the account are two different operations. As long as this is the case, we can never guarantee that this type of problem will not occur.

Where two or more threads share a common resource, such as a file or a block of memory, you'll need to take steps to ensure that one thread doesn't modify a resource while that resource is still being used by another thread. Having one thread update a record in a file while another thread is part way through retrieving the same record is a recipe for disaster. One way of managing this sort of situation is to use synchronization for the threads involved.
Synchronization

The objective of synchronization is to ensure that, when several threads want access to a single resource, only one thread can access it at any given time. There are two ways in which you can use synchronization to manage your threads of execution:

• You can manage code at the method level this involves synchronizing methods.

• You can manage code at the block level using synchronizing blocks.

We'll look at how we can use synchronized methods first.

Synchronized Methods

You can make a subset (or indeed all) of the methods for any class object mutually exclusive, so that only one of the methods can execute at any given time. You make methods mutually exclusive by declaring them in the class using the keyword synchronized. For example:
Thread Priorities

All threads have a priority that determines which thread is executed when several threads are waiting for their turn. This makes it possible to give one thread more access to processor resources than another. Let's consider an elementary example of how this could be used. Suppose you have one thread in a program that requires all the processor resources—some solid long running calculation—and some other threads that require relatively few resources. By making the thread that requires all the resources a low priority thread, you ensure that the other threads get executed promptly, while the processor bound thread can make use of the processor cycles that are left over after the others have had their turn.

The possible values for thread priority are defined in static data members of the class Thread. These members are of type int, and are declared as final. The maximum thread priority is defined by the member MAX_PRIORITY, which has the value 10. The minimum priority is MIN_PRIORITY, defined as 1. The value of the default priority that is assigned to the main thread in a program is NORM_PRIORITY, which is set to 5. When you create a thread, its priority will be the same as that of the thread that created it.

You can modify the priority of a thread by calling the setPriority() method for the Thread object. This method accepts an argument of type int that defines the new priority for the thread. An IllegalArgumentException will be thrown if you specify a priority that is less than MIN_PRIORITY or greater than MAX_PRIORITY.

If you're going to be messing about with the priorities of the threads in your program, you need to be able to find out the current priority for a thread. You can do this by calling the getPriority() method for the Thread object. This will return the current priority for the thread as a value of type int.

Using Thread Priorities

In the last example, you could set priorities for the threads by adding statements to main():
Summary

In this chapter you have learned about threads and how you can create and manage them. We will be using threads from time to time in examples later in this book so be sure you don't move on from here without being comfortable with the basic ideas of how you create and start a thread.

The essential points that we have covered in this chapter are:

•

Threads are subtasks in a program that can be in execution concurrently.

•

A thread is represented by an object of the class Thread. Execution of a thread begins with the execution of the `run()` method defined in the class `Thread`.

•

You define the code to be executed in a thread by implementing the `run()` method in a class derived from `Thread`, or in a class that implements the interface `Runnable`.

•

A thread specified as `daemon` will cease execution when the thread that created it ends.

•

A thread that isn't a daemon thread is called a `user thread`. A user thread will not be terminated automatically when the thread that created it ends.

•

You start execution of a thread by calling the `start()` method for its `Thread` object. If you need to halt a thread before normal completion you can stop execution of a thread by calling the `interrupt()` method for its `Thread` object.

•

Methods can be declared as synchronized. Only one synchronized instance method for an object can execute at any given time. Only one synchronized static method for a class can execute at one time.

•

A code block can be declared as synchronized on an object. Only one synchronized code block for an object can execute at any given time.
In a synchronized method or code block, you can call the `wait()` method inherited from the class `Object` to halt execution of a thread. Execution of the waiting thread will continue when the `notify()` or `notifyAll()` method inherited from `Object` is called by a thread synchronized on the same object.

The `notify()` or `notifyAll()` method can only be called from a method or code block that is synchronized to the same object as the method or block that contains the `wait()` method that halted the thread.

You can modify the relative priority of a thread by calling its `setPriority()` method. This only has an effect on execution in environments that support priority scheduling.
Exercises

1.

Modify the last example in the chapter so that each transaction is a debit or a credit at random.

2.

Modify the result of the previous exercise to incorporate an array of clerks, each running in their own thread, and each able to handle both debits and credits.

3.

Extend the result of the previous exercise to incorporate two supervisors for two teams of clerks, where the supervisors each run in their own thread. The supervisor threads should originate transactions and pass them to the clerks they supervise.
Chapter 16: Creating Windows

Overview

Until now, the programs we have been creating have perhaps not been what you may instinctively think of as a program. We can't expect a user to know about classpaths and so on in order to run one of our programs. More traditionally, an application consists of one or more windows that the user can interact with. These windows, and the environment that the user interacts with, is known as the **Graphical User Interface (GUI)**.

In this chapter, we will investigate how to create a window for a Java application and we will take a first look at some of the components we can put together to create a graphical user interface in Java.

You will learn:

- How to create a resizable window.
- What components and containers are.
- How you can add components to a window.
- How to control the layout of components.
- How to create a menu bar and menus for a window.
- What a menu shortcut is and how you can add a shortcut for a menu item.
- What the restrictions on the capabilities of an applet are.
How to convert an application into an applet.
Graphical User Interfaces in Java

There is a vast amount of functionality in the Java class libraries devoted to supporting graphical user interface (GUI) creation and management, far more than it is feasible to cover in a single book— even if it is big. Just the JFrame class, which we will be exploring in a moment, contains more than 200 methods when one includes those inherited from superclasses! We will therefore have to be selective in what we go into in detail, in terms of both the specific classes we discuss and their methods. We will however cover the basic operations that you need to understand to create your own applications and applets. With a good grasp of the basics, you should be able to explore other areas of the Java class library beyond those discussed without too much difficulty.

The fundamental elements that you need in order to create a GUI reside in two packages, java.awt and javax.swing. The java.awt package was the primary repository for classes you would use to create a GUI in Java 1.1— 'awt' being an abbreviation for Abstract Windowing Toolkit, but many of the classes it defines have been superseded in Java 2 by javax.swing. Most of the classes in the java.awt package define GUI elements, referred to as Swing components, that provide much-improved alternatives to components defined by classes in java.awt. We will be looking into the JButton class in the Swing set that defines a button, rather than the Button class in java.awt. However, the Swing component classes are generally derived from, and depend on, fundamental classes within java.awt, so you can't afford to ignore these.

The Swing classes are part of a more general set of GUI programming capabilities that are collectively referred to as the Java Foundation Classes, or JFC for short. JFC covers not only the Swing component classes, such as those defining buttons and menus, but also classes for 2D drawing from the java.awt.geom package, and classes that support drag-and-drop capability in the java.awt.dnd package. The JFC also includes an API defined in the javax.accessibility package that allows applications to be implemented that provide for users with disabilities.

The Swing component classes are more flexible than the component classes defined in the java.awt package because they are implemented entirely in Java. The java.awt components depend on native code to a great extent, and are, therefore, restricted to a 'lowest common denominator' set of interface capabilities. Because Swing components are pure Java, they are not restricted by the characteristics of the platform on which they run. Apart from the added function and flexibility of the Swing components, they also provide a feature called pluggable look-and-feel that makes it possible to change the appearance of a component. You can programatically select the look-and-feel of a component from those implemented as standard, or you can create your own look-and-feel for components if you wish. The pluggable look-and-feel of the Swing components has been facilitated by designing the classes in a particular way, called the Model View Control architecture.

Model-View-Controller (MVC) Architecture

The design of the Swing component classes is loosely based on something called the Model-View-Controller architecture, or MVC. This is not of particular consequence in the context of applying the Swing classes, but it's important to be aware of it if you want to modify the pluggable look-and-feel of a component. MVC is not new, and did not originate with Java. In fact the idea of MVC emerged some time ago within the context of the Smalltalk programming language. MVC is an idealized way of modeling a component as three separate parts:
The **model** that stores the data that defines the component.


The **view** that creates the visual representation of the component from the data in the model.


The **controller** that deals with user interaction with the component and modifies the model and/or the view in response to a user action as necessary.


In object-oriented terms, each of the three logical parts for a component—the model, the view, and the controller—is ideally represented by a different class type. In practice this turns out to be difficult because of the dependencies between the view and the controller. Since the user interacts with the physical representation of the component, the controller operation is highly dependent on the implementation of the view. For this reason, the view and controller are typically represented by a single composite object that corresponds to a view with an integrated controller. In this case the MVC concept degenerates into the document/view architecture that we introduced when we discussed the Observable class and Observer interface. Sun call it the **Separable Model architecture**.


The way the Swing components provide for a pluggable look-and-feel is to make the visual appearance of a component and the interface to the user the responsibility of an independent object called the **UI delegate**. This is the view+controller part of the MVC model. Thus a different UI delegate can provide a component with a new look-and-feel.

The details of how you modify the look-and-feel of a component is beyond the scope of this book. It is, however, as well to be aware of the MVC architecture on which the Swing components are based since it appears quite often in the literature around Java, and should you want to change the look-and-feel of a component at some time.
Creating a Window

A basic window in Java is represented by an object of the class Window in the package java.awt. Objects of the class Window are hardly ever used directly since borders and a title bar are fairly basic prerequisites for a typical application window, and this class provides neither. The library class JFrame, defined in javax.swing, is a much more useful class for creating a window, since as well as a title bar and a border, it provides a wealth of other facilities. Its superclasses are shown below.

The Component class is the grandmother of all component classes—it defines the basic properties and methods shared by all components. We will see later that all the Swing components have the Component class as a base. The Container class adds the capability for a Component object to contain other components, which is a frequent requirement. Since JFrame has Container as a superclass, a JFrame object can contain other components. Beyond the obvious need for a window to be able to contain the components that represent the GUI, a menu bar should contain menus, for instance, which in turn will contain menu items: a toolbar will obviously contain toolbar buttons, and there are many other examples. For this reason the Container class is also a base for all the classes that define Swing components.

The Window class adds methods to the Container class that are specific to a window, such as the ability to handle events arising from user interaction with the window. The Frame class is the original class in java.awt that provided a proper window, with a title bar and a border, with which everyone is familiar. The JFrame class adds functionality to the Frame class to support much more sophisticated facilities for drawing and displaying other components. You can deduce from the hierarchy in the diagram how a JFrame object can easily end up with its 200+ methods as it has five superclasses from which it inherits members. We aren't going to trawl through all these classes and methods. We'll just look into the ones we need in context as we go along, and then see how they are applied for real. This will hopefully teach you the most important methods in this class.

You can display an application window simply by creating an object of type JFrame, calling a method for the object to set the size of the window, and then calling a method to display the window. Let's try that right away.

Try It Out   Framing a Window

Here's the code:
Components and Containers

A component represents a graphical entity of one kind or another that can be displayed on the screen. A component is any object of a class that is a subclass of Component. As we have seen, a JFrame window is a component, but there are many others. Before getting into specifics, let's first get a feel for the general relationship between the groups of classes that represent components. Part of the class hierarchy with Component as a base is shown overleaf. The arrows in the diagram point towards the superclass.

This shows some of the subclasses of Component—the ones that are important to us at the moment. We discussed the chain through to JFrame earlier, but the other branches are new. The classes that we will be using directly are all the most commonly derived classes.

Let's summarize how you would typically use the key classes in this hierarchy:

<table>
<thead>
<tr>
<th>Class</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFrame</td>
<td>This is used as the basic Java application window. An object of this class has a title bar and provision for adding a menu. You can also add other components to it. You will usually subclass this class to create a window class specific to your application. It is then possible to add GUI components or draw in this window if required, as we will see.</td>
</tr>
<tr>
<td>JDialog</td>
<td>You use this class to define a dialog window that is used for entering data into a program in various ways. You usually code the creation of a dialog in response to some menu item being selected.</td>
</tr>
<tr>
<td>JApplet</td>
<td>This is the base class for a Java 2 applet—a program designed to run embedded in a web page. All your Java 2 applets will have this class as a base. You can draw in a JApplet and also add menus and other components.</td>
</tr>
</tbody>
</table>
JComponent

The subclasses of JComponent define a range of standard components such as menus, buttons, checkboxes, and so on. You will use these classes to create the GUI for your application or applet.

All the classes derived from Container can contain other objects of any of the classes derived from Component, and are referred to generically as containers. Since the Container class is a sub class of the Component class, every container object is a Component too, so a container can contain other containers. The exception is the Window class and its subclasses, as objects of type Window (or of a subclass type) can't be contained in another container. If you try to do this, an exception will be thrown. The JComponent class is the base for all the Swing components used in a window as part of the GUI, so, since this class is derived from Container, all of the Swing components are also containers.

As you can see, the JApplet class, which is a base class for all Swing applets, is derived from Component via the Container class. An applet will, therefore, also inherit the methods from the Container and Component classes. It also inherits methods from the old Applet class, which it extends and improves upon. You should note that the JApplet, JFrame, and JDialog classes, and the JComponent class and its subclasses, are all in the package javax.swing. The Applet class is in java.applet, and all the others are in java.awt. The package java.applet is tiny; it only contains the one class plus three related interfaces, but we won't need to use it directly. We will always be using the JApplet class to define an applet, as it's significantly better than Applet.

Window and Frame Components

The basic difference between a JFrame object and a Window object is that a JFrame object represents the main window for an application, whereas a Window object does not you always need a JFrame object before you can create a Window object.

Since the JDialog class is derived directly from the Window class, you can only create a JDialog object in an application in the context of a JFrame object. Apart from the default constructor, the constructors for the JDialog class generally require a JFrame object to be passed as an argument. This JFrame object is referred to as the parent of the JDialog object. A JFrame object has a border, is resizable, and has the ability to hold a built-in menu bar. Since a JFrame object is the top-level window in an application, its size and location are defined relative to the screen. A JDialog object with a JFrame object as a parent will be located relative to its parent.

Note that while we will discuss applets based on the JApplet class in this book, there is still a significant role for applets based on the more restricted capabilities of the Applet class. This is because as yet browsers do not support Java 2 applets by default. Both Netscape Navigator and Microsoft Internet Explorer require the Java Plug-In from Sun to be installed before a Java2 applet can be executed. You can download the Java Plug-In from the Sun Java web site at http://java.sun.com/products.

As we said, the JApplet, JFrame, and JDialog classes are all containers because they have Container as a base class and therefore, in principle, can contain any kind of component. They are also all components themselves since they are derived ultimately from the Component class. However, things are not quite as simple as that. You don't add the components for your application or applet GUI directly to the JFrame or JApplet object for your program. Let's look at how it actually works in practice.

Window Panes
When you want to add GUI components or draw in a window displayed from a JFrame object, you add the components to, or draw on, a window pane that is managed by the JFrame object. The same goes for an applet. Broadly speaking, window panes are container objects that represent an area of a window, and they come in several different types.

You will use a window pane called the **content pane** most of the time, but there are others. The relationship between the contentPane object, other window panes, and the application window itself is shown here.

![Diagram showing the relationship between window panes](image)

As you see, the area below the title bar in a JFrame window corresponds to a JRootPane object. This contains another pane, the layeredPane object in the illustration, which is of type JLayeredPane. This pane corresponds to the whole of the area occupied by the JRootPane object in the window and manages the menu bar if the window has one. The area in the layeredPane below the menu bar corresponds to the contentPane object, and it's here that you typically add GUI components. You also display text or do any drawing in the area covered by the content pane.

The layeredPane object has special properties for advanced applications that permit groups of components to be managed in separate layers that overlay one another within the pane. With this capability you can control how components are displayed relative to one another, because the layers are displayed in a particular order from back to front. The components in a layer at the front will appear on the screen in front of those in a layer that is towards the back.

There is also an additional pane not shown in the diagram. This is the glassPane object, and this also corresponds to the complete JRootPane area. The contents of the glassPane object displays on top of all the other panes, so this is used to display components that you always want to display on top of anything else displayed in the window such as pop-up menus. You can also use the glassPane to display graphics that need to be updated relatively frequently such as when you create an animation. When part of what is displayed is to be animated, a static background can be displayed independently via the contentPane. Since this doesn't need to be reprocessed each time the animated objects need to be redrawn, the whole process can be much more efficient.

The JFrame class defines methods to provide you with a reference to any of the panes:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRootPane()</td>
<td>Returns the root pane as type JRootPane.</td>
</tr>
</tbody>
</table>
getLayeredPane()  
Returns the layered pane as type JLayeredPane.

getContentPane()  
Returns the content pane as type Container. This is the method you will use most frequently, since you normally add components to the content pane.

getGlassPane()  
Returns the glass pane as type Component.

All the classes discussed here that represent panes are themselves Swing components, defined in the javax.swing package. A JApplet object has the same arrangement of panes as a JFrame object, so adding components to an applet, or drawing on it, works in exactly the same way. An applet defined as a JApplet object can also have a menu bar just like an application window.

All the panes, as well as the menu bar, are components, so before we start delving into how to add a menu bar or other components to a window, let's unearth a little more about the make-up of components in general.
Basics of Components

There's quite a lot of basic stuff that is common to all components that we have to examine before we can apply components properly. It also has applications in many different contexts. While this may seem like something of a catalog of classes and methods, without much in the way of practical application, please stay with it. We will be using most of these capabilities in a practical context later. To understand the fundamental things we can do with Swing components, we will examine what functionality they inherit from the Component and Container classes.

When a component is contained within another component, the outer object is referred to as the parent. You can find out what the parent of any given component is by calling its getParent() method. This method is inherited from the Component class and it returns the parent as type Container, since only a subclass of Container can hold other components. If there is no parent, as is the case with a JFrame component, this method will return null.

Component Attributes

The Component class defines attributes, which record the following information about an object:

- The **position** is stored as (x, y) coordinates. This fixes where the object is in relation to its container in the coordinate system of the container object.

- The **name** of the component is stored as a String object.

- The **size** is recorded as values for the width and the height of the object.

- The **foreground color** and **background color** that apply to the object. These color values are used when the object is displayed.

- The **font** used by the object when text is displayed.

- The **cursor** for the object. This defines the appearance of the cursor when it is over the object.

- Whether the object is **enabled** or not. When a component is enabled its enabled state is true and it has a normal appearance. When a component is disabled it is grayed out. Note that a disabled component can still
originate events.

Whether the object is **visible** on the screen or not if an object is not marked as visible it is not drawn on the screen.

Whether the object is **valid** or not if an object is not valid, layout of the entities that make up the object has not been determined. This is the case before an object is made visible. You can make a Container object invalid by changing its contents. It will then need to be validated before it is displayed correctly.

You can only modify the characteristics of a Component object by calling its methods or affecting it indirectly in some way since none of the data members that store its characteristics are directly accessible they are all private. For example, you can change the name of a Component object myWindow with the statement:
Using Containers

A container is any component with the Container class as a base; so all the Swing components are containers. The Container class is the direct base class for the Window class and it provides the ability to contain other components. Since the Container class is an abstract class, you cannot create instances of Container. Instead it is objects of the subclasses such as Window, JFrame, or JDialog that inherit the ability to contain other components.

**Important**

Note that a container cannot contain an object of the class `Window`, or an object of any of the classes derived from `Window`. An object of any other class that is derived from `Component` can be contained.

The components within a container are displayed within the area occupied by the container on the display screen. A dialog box, for example, might contain a JList object offering some choices: JCheckbox objects offering other options and JButton objects representing buttons enabling the user to end the dialog or enter the selections; all these components would appear within the boundaries of the dialog box. Of course, for the contained components to be visible the container must itself be displayed, as the container effectively ‘owns’ its components. The container also controls how its embedded components are laid out by means of a **layout manager**.

Before we look at what a layout manager is, and how the layout of the components in a container is determined, let’s look into the basic methods defined in the Container class, and therefore available to all containers.

You can find out about the components in a container object by using the following methods defined in the Container class:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getComponentCount()</code></td>
<td>Returns a count of the number of components contained by the current component.</td>
</tr>
<tr>
<td><code>Component getComponent (int index)</code></td>
<td>Returns the component identified by the index value. The index value is an array index so it must be between 0 and one less than the number of components contained, otherwise an ArrayIndexOutOfBoundsException will be thrown.</td>
</tr>
<tr>
<td><code>Component[]getComponents()</code></td>
<td>Returns an array of all the components in the current container.</td>
</tr>
</tbody>
</table>
If we have a Container object, content, perhaps the content pane of a JFrame window, we could iterate through the components in the Container with the following statements:
Container Layout Managers

An object called a layout manager determines the way that components are arranged in a container. All containers will have a default layout manager but you can choose a different layout manager when necessary. There are many layout manager classes provided in the java.awt and javax.swing packages, so we will introduce those that you are most likely to need. It is possible to create your own layout manager classes, but creating layout managers is beyond the scope of this book. The layout manager for a container determines the position and size of all the components in the container: you should not change the size and position of such components yourself. Just let the layout manager take care of it.

Since the classes that define layout managers all implement the LayoutManager interface, you can use a variable of type LayoutManager to store any of them if necessary. We will look at six layout manager classes in a little more detail. The names of these classes and the basic arrangements that they provide are as follows:

<table>
<thead>
<tr>
<th>Layout Manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FlowLayout</td>
<td>Places components in successive rows in a container, fitting as many on each row as possible, and starting on the next row as soon as a row is full. This works in much the same way as your text processor placing words on a line. Its primary use is for arranging buttons although you can use it with other components. It is the default layout manager for JPanel objects.</td>
</tr>
<tr>
<td>BorderLayout</td>
<td>Places components against any of the four borders of the container and in the center. The component in the center fills the available space. This layout manager is the default for the contentPane in a JFrame, JDialog, or JApplet object.</td>
</tr>
<tr>
<td>CardLayout</td>
<td>Places components in a container one on top of the other like a deck of cards. Only the 'top' component is visible at any one time.</td>
</tr>
<tr>
<td>GridLayout</td>
<td>Places components in the container in a rectangular grid with the number of rows and columns that you specify.</td>
</tr>
<tr>
<td>GridBagLayout</td>
<td>This also places the components into an arrangement of rows and columns but the rows and columns can vary in length. This is a complicated layout manager with a lot of flexibility in how you control where components are placed in a container.</td>
</tr>
</tbody>
</table>
BoxLayout

This arranges components either in a row or in a column. In either case the components are clipped to fit if necessary, rather than wrapping to the next row or column. The BoxLayout manager is the default for the Box container class.

SpringLayout

Allows components to have their position defined by 'springs' or 'struts' fixed to an edge of the container or another component in the container.

The BoxLayout, SpringLayout, and Box classes are defined in the javax.swing package. The other layout manager classes in the list above are defined in java.awt.

One question to ask is why do we need layout managers at all? Why don't we just place components at some given position in a container? The basic reason is to ensure that the GUI elements for your Java program are displayed properly in every possible Java environment. Layout managers automatically adjust components to fit the space available. If you fix the size and position of each of the components, they could run into one another and overlap if the screen area available to your program is reduced.

To set the layout manager of a container, you can call the setLayout() method for the container. For example, you could change the layout manager for the container object aWindow of type JFrame to flow layout with the statements:
Adding a Menu to a Window

As we have already discussed, a JMenuBar object represents the menu bar that is placed at the top of a window. You can add JMenu or JMenuItem objects to a JMenuBar object and these will be displayed on the menu bar. A JMenu object is a menu item with a label that can display a pull-down menu when clicked. A JMenuItem object represents a simple menu item with a label that results in some program action when clicked such as opening a dialog. A JMenuItem can have an icon in addition to, or instead of, a String label. Each item on the pull-down menu for an object of type JMenu, can be an object of either type JMenu, JMenuItem, JCheckBoxMenuItem, or JRadioButtonMenuItem.

A JCheckBoxMenuItem is a simple menu item with a checkbox associated with it. The checkbox can be checked and unchecked and typically indicates that that menu item was selected last time the pull-down menu was displayed. You can also add separators in a pull-down menu. These are simply bars to separate one group of menu items from another. A JRadioButtonMenuItem is a menu item much like a radio button in that it is intended to be one of a group of like menu items added to a ButtonGroup object. Both JCheckBoxMenuItem and JRadioButtonMenuItem objects can have icons.

Creating JMenu and JMenuItem

To create a JMenu object you call a JMenu class constructor and pass a String object to it that contains the label for the menu. For example, to create a File menu you would write:
More on Applets

Applets are a peculiar kind of program as they are executed in the context of a web browser. This places some rather severe restrictions on what you can do in an applet, to protect the environment in which they execute. Without these restrictions they would be a very direct means for someone to interfere with your system — in short, a virus delivery vehicle.

System security in Java programs is managed by a **security manager**. This is simply an object that provides methods for setting and checking security controls that determine what is and what is not allowed for a Java program. What an applet can and cannot do is determined by both the security manager that the browser running the applet has installed, and the security policy that is in effect for the system.

Unless permitted explicitly by the security policy in effect, the main default limitations on an applet are:

- An applet cannot have any access to files on the local computer.
- An applet cannot invoke any other program on the local computer.
- An applet cannot communicate with any computer other than the computer from which the HTML page containing the applet was downloaded.

Obviously there will be circumstances where these restrictions are too stringent. In this case you can set up a security policy that allows certain operations for specific trusted programs, applets, or sites, by authorizing them explicitly in a **policy file**. A policy file is an ASCII text file that defines what is permitted for a particular code source. We won't be going into details on this, but if you need to set up a policy file for your system, it is easiest to use the policytool program supplied with the JDK.

Because they are normally shipped over the Internet as part of an HTML page, applets should be compact. This doesn't mean that they are inevitably simple or unsophisticated. Because they can access the host computer from which they originated, they can provide a powerful means of enabling access to files on that host, but they are usually relatively small to allow them to be easily downloaded.

The `JApplet` class includes the following methods, which are all called automatically by the browser or applet viewer controlling the applet:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>

This document is created with the unregistered version of CHM2PDF Pilot
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init()</td>
<td>You implement this method to do any initialization that is necessary for the applet. This method is called once by the browser when the applet starts execution.</td>
</tr>
<tr>
<td>void start()</td>
<td>You implement this method to start the processing for the applet. For example, if your applet displays an animated image, you would start a thread for the animation in this method. This method is called by the browser immediately after init(). It is also called if the user returns to the current .html page after leaving it.</td>
</tr>
<tr>
<td>void stop()</td>
<td>This method is called by the browser when the user moves off the page containing the applet. You implement this to stop any operations that you started in the start() method.</td>
</tr>
<tr>
<td>void destroy()</td>
<td>This method is called after the stop() method when the browser is shut down. In this method you can release any resources your applet uses that are managed by the local operating system. This includes such things as resources used to display a window.</td>
</tr>
</tbody>
</table>

These are the basic methods you need to implement in the typical applet. We really need some graphics knowledge to go further with implementing an applet, so we will return to the practical application of these methods in Chapter 13.

**Converting an Application to an Applet**

Subject to the restrictions described in the previous section, you can convert an application to an applet relatively easily. You just need to be clear about how each part of program executes. You know that an application is normally started in the method main(). The method main() is not called for an applet but the method init() is, so one thing you should do is add an init() method to the application class. The other obvious difference is that an applet always extends the JApplet class.

We can demonstrate how to convert an application so that it also works as an applet, by changing the definition of the Sketcher class. This doesn't make a very sensible applet, but you can see the principles at work.

**Try It Out  Running Sketcher as an Applet**

You need to modify the contents of Sketcher.java so that it contains the following:
Summary

In this chapter you have learned how to create an application window, and how to use containers in the creation of the GUI for a program. We discussed the following important points:

- The package javax.swing provides classes for creating a graphical user interface (GUI).

- A component is an object that is used to form part of the GUI for a program. All components have the class Component as a superclass.

- A container is a component that can contain other components. A container object is created with a class that is a subclass of Container. The classes JPanel, JApplet, JWindow, JFrame, and JDialog are containers.

- The class JApplet is the base class for an applet. The JFrame class is a base class for an application window with a title bar, borders, and a menu.

- The arrangement of components in a container is controlled by a layout manager.

- The default layout manager for the content pane of JFrame, JApplet, and JDialog objects is BorderLayout.

- The GridBagLayout provides the most flexible control of the positioning of components in a container. The position of a component in a GridBagLayout is controlled by a GridBagConstraints object.

- A Box container can be used to arrange components or containers in rows and columns. You can use multiple nested Box containers in combination to create more complex arrangements quite easily, that otherwise might require GridBagLayout to be used.

- A layout manager of type SpringLayout arranges components by applying constraints in the form of Spring objects to their edges.
A menu bar is represented by a JMenuBar object. Menu items can be objects of type JMenuItem, JMenu, JCheckBoxMenuItem, or JRadioButtonMenuItem.

- You associate a pull-down menu with an item of type JMenuItem.

- You can create a shortcut for a menu by calling its setMnemonic() method, and you can create an accelerator key combination for a menu item by calling its setAccelerator() method.

In the next chapter we will move on to look at events—that is, how we associate program actions with menu items and components within a window, and how to close a window when the close icon is clicked.
Exercises

1. Create an application, with a square window in the center of the screen that is half the height of the screen by deriving your own window class from JFrame.

2. Add six buttons to the application in the previous example in a vertical column on the left side of the application window.

3. Add a menu bar containing the items File, Edit, Window, and Help.

4. Add a pull-down menu for Edit containing the two groups of items of your own choice with a separator between them.

5. Add another item to the Edit pull-down menu which itself has a pull-down menu, and provide accelerators for the items in the menu.

6. Here's an exercise to tickle the brain cells: use a SpringLayout to obtain the button arrangement shown below in an application window.
Chapter 17: Handling Events

Overview

In this chapter you will learn how a window-based Java application is structured, and how to respond to user actions in an application or an applet. This is the fundamental mechanism you will be using in virtually all of your graphical Java programs. Once you understand how user actions are handled in Java, you will be equipped to implement the application-specific code that is necessary to make your program do what you want.

In this chapter you will learn:

• What an event is.

• What an event-driven program is and how it is structured.

• How events are handled in Java.

• How events are categorized in Java.

• How components handle events.

• What an event listener is, and how you create one.

• What an adapter class is and how you can use it to make programming the handling of events easier.

• What actions are and how you use them.
How to create a toolbar.
Window-based Java Programs

Before we get into the programming specifics of window-based programs, we need to understand a little of how such programs are structured, and how they work. There are fundamental differences between the console programs that we have been producing up to now, and a window-based Java program. With a console program, you start the program, and the program code determines the sequences of events. Generally everything is predetermined. You enter data when required and the program will output data when it wants. At any given time, the specific program code to be executed next is generally known.

A window-based application, or an applet come to that, is quite different. The operation of the program is driven by what you do with the GUI. Selecting menu items or buttons using the mouse, or through the keyboard, causes particular actions within the program. At any given moment you have a whole range of possible interactions available to you, each of which will result in a different program action. Until you do something, the specific program code that is to be executed next is not known.

Event-driven Programs

Your actions when you're using the GUI for a window-based program or an applet, clicking a menu item or a button, moving the mouse and so on, are first identified by the operating system. For each action, the operating system determines which of the programs currently running on your computer should know about it, and passes the action on to that program. When you click a mouse button, it's the operating system that registers this and notes the position of the mouse cursor on the screen. It then decides which application controls the window where the cursor was when you pressed the button, and communicates the mouse button-press to that program. The signals that a program receives from the operating system as a result of your actions are called events.

A program is not obliged to respond to any particular event. If you just move the mouse, for instance, the program need not invoke any code to react to that. If it doesn't, the event is quietly disposed of. Each event that the program does recognize is associated with one or more methods, and when the event occurs, when you click a menu item, for example, the appropriate methods will be called automatically. A window-based program is called an event-driven program, because the sequence of events created as a result of your interaction with the GUI drives and determines what happens in the program.

Important

Events are not limited to window-based applications. They are a quite general concept. Most programs that control or monitor things are event-driven. Any occurrence external to a program such as a switch closing, or a preset temperature being reached, can be registered as an event. In Java you can even create events within your program to signal some other part of the code that something noteworthy has happened. However, we're going to concentrate on the kinds of events that occur when you interact as a user with a program.
The Event-handling Process

To manage the user's interaction with the components that make up the GUI for a program, we must understand how events are handled in Java. To get an idea of how this works, let's consider a specific example. Don't worry too much about the class names and other details here. Just try to get a feel for how things connect together.

Suppose the user clicks a button in the GUI for your program. The button is the source of this event. The event generated as a result of the mouse click is associated with the JButton object in your program that represents the button on the screen. An event always has a source object in this case the JButton object. When the button is clicked, it will create a new object that represents and identifies this event in this case an object of type ActionEvent. This object will contain information about the event and its source. Any event that is passed to a Java program will be represented by a particular event object and this object will be passed as an argument to the method that is to handle the event.

The event object corresponding to the button click will be passed to any listener object that has previously registered an interest in this kind of event a listener object being simply an object that listens for particular events. A listener is also called a target for an event. Here, 'passing the event to the listener' just means the event source calling a particular method in the listener object and passing the event object to it as an argument. A listener object can listen for events for a particular object just a single button for instance, or it can listen for events for several different objects a group of menu items for example. Which approach you take depends on the context, and which is most convenient from a programming point of view. Your programs will often involve both.

So how do you define a listener? You can make the objects of any class listener objects by making the class implement a listener interface. There are quite a variety of listener interfaces, to cater for different kinds of events. In the case of our button click, the ActionListener interface needs to be implemented to receive the event from the button. The code that is to receive this event object and respond to the event is implemented in a method declared in the listener interface. In our example, the actionPerformed() method in the ActionListener interface is called when the event occurs, and the event object is passed as an argument. Each kind of listener interface defines particular methods for receiving the events that that listener has been designed to deal with.

Simply implementing a listener interface isn't sufficient to link the listener object to an event source. You still have to connect the listener to the source, or sources, of the events that you want it to deal with. You register a listener object with a source by calling a particular method in the source object. In this case, we call the addActionListener() method for the JButton object, and pass the listener object as an argument to the method.

This mechanism for handling events using listeners is very flexible, and very efficient, particularly for GUI events. Any number of listeners can receive a particular event. However, a particular event is only passed to the listeners that have registered to receive it, so only interested parties are involved in responding to each event. Since being a listener just requires a suitable interface to be implemented, you can receive and handle events virtually anywhere you like. The way in which events are handled in Java, using listener objects, is referred to as the delegation event model. This is because the responsibility for responding to events that originate with a component, such as a button or a menu item, is not handled by the objects that originated the events themselves but is delegated to separate listener objects.
Not all event handling necessarily requires a separate listener. A component can handle its own events, as we shall see a little later in this chapter.

A very important point to keep in mind when writing code to handle events is that all such code executes in the same thread, the event-dispatching thread. This implies that while your event-handling code is executing, no other events can be processed. The code to handle the next event will only start executing when the current event-handler finishes. Thus the responsiveness of your program to the user is dependent on how long your event-handling code takes to execute. For snappy performance, your event handlers must take as little time as possible to execute.

Let's now get down to looking at the specifics of what kinds of events we can expect, and the range of listener interfaces that process them.
Event Classes

There are many different kinds of events to which your program may need to respond from menus, buttons, from the mouse, from the keyboard, and a number of others. In order to have a structured approach to handling events, these are broken down into subsets. At the topmost level, there are two broad categories of events in Java:

- **Low-level Events** these are events that arise from the keyboard or from the mouse, or events associated with operations on a window such as reducing it to an icon or closing it. The meaning of a low-level event is something like 'the mouse was moved', 'this window has been closed' or 'this key was pressed.'

- **Semantic Events** these are specific component-related events such as pressing a button by clicking it to cause some program action, or adjusting a scrollbar. They originate, and you interpret them, in the context of the GUI you have created for your program. The meaning of a semantic event is typically along the lines of: 'the OK button was pressed', or 'the Save menu item was selected'. Each kind of component, a button, or a menu item for example, can generate a particular kind of semantic event.

These two categories can seem to be a bit confusing as they overlap in a way. If you click a button, you create a semantic event as well as a low level event. The click produces a low-level event object in the form of 'the mouse was clicked' as well as a semantic event 'the button was pushed'. In fact it produces more than one mouse event, as we shall see. Whether your program handles the low-level events or the semantic event, or possibly both kinds of event, depends on what you want to do.

Most of the events relating to the GUI for a program are represented by classes defined in the package java.awt.event. This package also defines the listener interfaces for the various kinds of events that it defines. The package javax.swing.event defines classes for events that are specific to Swing components.

Low-level Event Classes

There are four kinds of low-level events that you can elect to handle in your programs. They are represented by the following classes in the java.awt.event package:

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FocusEvent</td>
<td>Objects of this class represent events that originate when a component gains or loses the keyboard focus. Only the component that has the focus can receive input from the keyboard, so it will usually be highlighted or have the cursor displayed.</td>
</tr>
</tbody>
</table>
MouseEvent

Objects of this class represent events that result from user actions with the mouse such as moving the mouse or pressing a mouse button.

KeyEvent

Objects of this class represent events that arise from pressing keys on the keyboard.

WindowEvent

Objects of this class represent events that relate to a window, such as activating or deactivating a window, reducing a window to its icon, or closing a window. These events relate to objects of the Window class or any subclass of Window.

The MouseEvent class has two subclasses that identify more specialized mouse events. One is the MenuDragMouseEvent class that defines event objects signaling when the mouse has been dragged over a menu item. The other is the MouseWheelEvent class that defines event objects indicating when the mouse wheel is rotated.

Important

Just so that you know, this isn't an exhaustive list of all of the low-level event classes. It's a list of the ones you need to know about. For example, there's the PaintEvent class that is concerned with the internals of how components get painted on the screen. There's also another low-level event class, ContainerEvent, which defines events relating to a container such as adding or removing components. You can ignore this class as these events are handled automatically.

Each of these classes defines methods that enable you to analyze the event. For a MouseEvent object, for example, you can get the coordinates of the cursor when the event occurred. These low-level event classes also inherit methods from their superclasses, and are related in the manner shown in the diagram:

The class AWTEvent is itself a subclass of java.util.EventObject. The EventObject class implements the Serializable interface, so all objects of the event classes in the diagram are serializable. It also defines a method, getSource(), which returns the object that is the source of an event as type Object. All the event classes shown inherit this method.

The class AWTEvent defines constants, which are public final values identifying the various kinds of events. These constants are named for the sake of consistency as the event name in capital letters, followed by _MASK. The constants identifying the low-level events that you are most likely to be interested in are:

<table>
<thead>
<tr>
<th>MOUSE_EVENT_MASK</th>
<th>MOUSE_MOTION_EVENT_MASK</th>
<th>MOUSE_WHEEL_EVENT_MASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY_EVENT_MASK</td>
<td>ADJUSTMENT_EVENT_MASK</td>
<td>TEXT_EVENT_MASK</td>
</tr>
<tr>
<td>WINDOW_EVENT_MASK</td>
<td>WINDOW_FOCUS_EVENT_MASK</td>
<td>WINDOW_STATUS_EVENT_MASK</td>
</tr>
<tr>
<td>ITEM_EVENT_MASK</td>
<td>FOCUS_EVENT_MASK</td>
<td></td>
</tr>
</tbody>
</table>
Each of these constants is a value of type long with a single bit being set to 1 and all the remaining set to 0. Because they are defined this way you can combine them using a bitwise OR operator and you can separate a particular constant out from a combination by using a bitwise AND.

Important

The list of event masks above is not exhaustive. There are masks for component events represented by objects of the class ComponentEvent, and for container events. These events occur when a component is moved or resized, or a component is added to a container, for example. There is also a mask for events associated with components that receive text input. You won't normally need to get involved in these events so we won't be discussing them further.

You use the identifiers for event masks to enable a particular group of events in a component object. You call the enableEvents() method for the component, and pass the variable for the events you want enabled as an argument. However, you only do this when you aren't using a listener. Registering a listener automatically enables the events that the listener wants to hear, so you don't need to call the enableEvents() method. The circumstance when you might do this is when you want an object to handle some of its own events although you can achieve the same result using a listener.

### Making a Window Handle its own Events

Using listeners is the preferred way of handling events since it is easier than enabling events directly for an object, and the code is clearer. Nonetheless, we should take a look at how you deal with events after calling enableEvents(). An example of where you might want to call enableEvents() exists in our SketchFrame class in the Sketcher program.

As you may recall from the previous chapter, we used the setDefaultCloseOperation() method to determine what happened when you close the window by clicking on the icon. Although the EXIT_ON_CLOSE argument value that we used disposed of the frame and closed the application, it didn't provide any opportunity to do any checking or clean-up before causing the program to exit. We can respond to the close icon being clicked in the program ourselves, rather than letting the JFrame facilities handle the associated event within the window object itself. This will eventually enable us to prompt the user to save any data that has been created, and then shut down the application ourselves when a close event occurs, so let's give it a try.

### Try It Out  Closing a Window

We need to modify the SketchFrame class definition from the previous chapter as follows:

```java
// Frame for the Sketcher application
```
Semantic Event Handling in Applets

Event handling in an applet is exactly the same as in an application, but we ought to see it for ourselves. Let's see how we would handle events for buttons in an applet. We can create an applet that uses some buttons that have listeners. To make this example a bit more gripping, we'll throw in the possibility of monetary gain.

That's interesting to almost everybody. Let's suppose we want to have an applet to create random numbers for a lottery. The requirement is to generate six different random numbers between 1 and 49. It would also be nice to be able to change a single number if you don't like it, so we'll add that capability as well. Since your local lottery may not be like this, we will implement the applet to make it easy for you to adapt the applet to fit your requirements.

By displaying the six selected numbers on buttons, we can provide for changing one of the choices by processing the action event for that button. Thus, clicking a button will provide another number. We'll also add a couple of control buttons, one to make a new selection for a complete set of lottery numbers, and another just for fun to change the button color. Here's how the applet will look when running under appletviewer:

Try It Out  A Lottery Applet

We can outline the broad structure of the applet's code as follows:
An obvious candidate for implementing semantic event listeners is in the Sketcher program, to support the operation of the menu bar in the class SketchFrame. When we click on an item in one of the pull-down menus, a semantic event will be generated that we can listen for and then use to determine the appropriate program action.

Listening to Menu Items

We will start with the Elements menu. This is concerned with identifying the type of graphic element to be drawn next, and the color in which it will be drawn. We won't be drawing them for a while, but we can put in the infrastructure to set the type and color for an element without worrying about how it will actually be created and drawn.

To identify the type of element, we can define constants that will act as IDs for the four types of element we have provided for in the menu so far. This will help us with the operation of the listeners for the menu item as well as provide a way to identify a particular type of element. Since we will accumulate quite a number of application wide constants, it will be convenient to define them in an interface that can be implemented by any class that refers to any of the constants. Here's the initial definition including constants to define line, rectangle, circle and curve elements:
Using Actions

One difficulty with the code that we have added to support the menus is that it is very menu specific. What I mean by this is that if we are going to do a proper job on the Sketcher application, we will undoubtedly want it to have a toolbar. The toolbar will surely have a whole bunch of buttons that perform exactly the same actions as the menu items we have just implemented, so we will be in the business of doing the same thing over again in the toolbar context. Of course, the only reason I brought it up, as I'm sure you anticipated, is that there is another way of working with menus, and that is to use an action object.

An action object is a bit of a strange beast, and it can be quite hard to understand at first so we will take it slowly. First of all let's look at what we mean by an 'action' here, as it is a precise term in this context. An action is an object of any class that implements the Action interface. This interface declares methods that operate on an action object, for example storing properties relating to the action, enabling it and disabling it. The Action interface happens to extend the ActionListener interface so an action object is a listener as well as an action. Now that we know an Action object can get and set properties, and is also a listener, how does that help us in implementing the Sketcher GUI?

The answer is in the last capability of an Action object. Some Swing components, such as those of type JMenu and JToolBar, have an add() method that accepts an argument of type Action. When you add an Action object to these using the add() method, the method creates a component from the Action object that is automatically of the right type. If you add an Action object to a JMenu object, a JMenuItem will be created and returned by the add() method. On the other hand, when you add exactly the same Action object to a JToolBar object, an object of type JButton will be created and returned. This means that you can add the very same Action object to both a menu and a toolbar, and since the Action object is its own listener you automatically get both supporting the same action. Clever, eh?

First, we should look at the Action interface.

The Action Interface

In general, properties are items of information that relate to a particular object and are stored as part of the object. Properties are often stored in a map, where a key identifies a particular property, and the value corresponding to that property can be stored in association with the key. The Properties class that is defined in the java.util package does exactly that. The Action interface has provision for storing seven basic standard properties that relate to an Action object:

- A **name** a String object that is used as the label for a menu item or a toolbar button.
- A **small icon** an Icon object to be displayed on a toolbar button.
• A short description of the action a String object to be used as a tooltip.

• An accelerator key for the action defined by a KeyStroke object.

• A long description of the action a String object that is intended to be used as context sensitive help.

• A mnemonic key for the action this is a key code of type int.

• An action command key defined by an entry in a KeyMap object associated with a component.

Just so you are aware of them I have included the complete set here, but we will concentrate on just using the first three. We haven't met Icon objects before, but we will get to them a little later in this chapter.

You are not obliged to provide for all of these properties in your action classes, but the interface provides the framework for it. These properties are stored internally in a map collection in your action class, so the Action interface defines constants that you use as keys for each of the standard properties. These constants are all of type String, and the ones we are interested in are NAME, SMALL_ICON, and SHORT_DESCRIPTION. The others are ACCELERATOR_KEY, LONG_DESCRIPTION, MNEMONIC_KEY, and ACTION_COMMAND_KEY. There is another constant of type String defined in the interface with the name DEFAULT. This is for you to use to store a default property for the action.

The Action interface also declares the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| void putValue(String key, Object value) | Stores the value with the key key in the map supported by the action class. To store the name of an action within a class method you might write:  

putValue(NAME, theName);

This uses the standard key, NAME to store the object theName. |
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `Object getValue(String key)` | This retrieves the object from the map corresponding to the key key. To retrieve a small icon within an action class method you might write:  
`Icon lineIcon = (Icon)getValue(SMALL_ICON);` |
| `boolean isEnabled()` | Returns true if the action object is enabled and false otherwise. |
| `void setEnabled(boolean state)` | Sets the action object as enabled if the argument state is true and disabled if it is false. This operates on both the toolbar button and the menu item if they have been created using the same object. |
| `void addPropertyChangeListener(PropertyChangeListener listener)` | This adds the listener passed as an argument that listens for changes to properties such as the enabled state of the object. This is used by a container for an action object to track property changes. |
| `void removePropertyChangeListener(PropertyChangeListener listener)` | This removes the listener passed as an argument. This is also for use by a Container object.  
Of course, since the Action interface extends the ActionListener interface, it also incorporates the ActionPerformed() method that you are already familiar with. |

So far, all we seem to have with this interface is a license to do a lot of work in implementing it but it's not as bad as that. The javax.swing package defines a class, AbstractAction, that already implements the Action interface. If you extend this class to create your own action class, you get a basic infrastructure for free. Let's try it out in the context of Sketcher.

**Using Actions as Menu Items**

This will involve major surgery on our SketchFrame class. Although we'll be throwing away all those fancy varieties of menu items we spent so much time putting together, at least you know how they work now, and we'll end up with much less code after re-engineering the class, as you'll see. As the saying goes, you've got to crack a few eggs to make a soufflé.

We'll go back nearly to square one and reconstruct the class definition. First we will delete a lot of code from the existing class definition. Comments show where we will add code to re-implement the menus using actions. Get your definition of SketchFrame to the following state:

```java
// Frame for the Sketcher application
```
Adding a Toolbar

A toolbar is a bar, usually positioned below the menu bar, which contains a row of buttons that typically provides a more direct route to menu options. We could add a toolbar to the Sketcher program for the menu items that are likely to be most popular. Just so that you know where we are heading, the kind of toolbar we will end up with ultimately is shown below.

The four buttons in the first group are for the most used functions in the file menu. The other two groups of four buttons select the element type and element color respectively. So how are we going to put this toolbar together?

Adding the toolbar itself couldn't be easier. A toolbar is a Swing component defined by the JToolBar class. You can add a member to the SketchFrame class for a toolbar by adding the following field to the class definition:
Summary

In this chapter you have learned how to handle events in your applications and in your applets. Events are fundamental to all window-based applications, as well as most applets, so you will be applying the techniques from this chapter throughout the rest of the book.

The most important points we have discussed in this chapter are:

- A user interaction generates an event in the context of a component.

- There are two categories of events associated with a component: **low-level events** from the mouse, keyboard or window system events such as opening or closing a window, and **semantic events** which represent component actions such as pressing a button or selecting a menu item.

- Both low-level and semantic events can arise simultaneously.

- An event for a component can be handled by the component object itself, or by a separate object that implements a listener interface corresponding to the event type.

- A component that is to handle its own events does so by calling its enableEvents() method and implementing the class method to process the kind of event that has been enabled.

- A listener object that is registered with a component will receive notification of the events originating with the component that correspond to the type(s) of events the listener can handle.

- A listener interface for low-level events requires several event-handling methods to be implemented.

- A listener interface for semantic events declares a single event handling method.

- An adapter class defines a set of empty methods for one or more low-level event interfaces. You can create your own class defining a low-level event listener by deriving your class from an adapter class, and then
implementing the event handling methods in which you are interested.

- Events in applications and in applets are handled in exactly the same way.

- An Action object is an object of a class that implements the Action interface. Action objects can be used to create menu items and associated toolbar buttons.

- An Action object is automatically the listener for the menu item and toolbar button that are created from it.
Exercises

1. Modify Sketcher to add an Exit action for the File menu and the toolbar.

2. Modify the lottery applet to present the six numbers selected in ascending sequence.

3. Replace the action listener for the selection buttons in the Lottery applet with a mouse listener, and use the mousePressed() method to update the selection with a new value.

4. Modify the Lottery applet to implement the mouse listener for a selection button as an inner class to the Lottery class.

5. Modify the Lottery applet to implement the control buttons on a toolbar based on Action objects.

6. Change the Lottery applet to handle the MOUSE_ENTERED and MOUSE_EXITED events within the toolbar buttons you added in the previous exercise.

7. Add tooltips to the lucky number buttons and the toolbar buttons in the Lottery applet. (You can make the tooltip the same for each of the lucky number buttons.)
Chapter 18: Drawing in a Window

Overview

In this chapter we will look at how you can draw using the Java 2D facilities that are part of JFC. We will see how to draw in an applet and in an application. We will investigate how we can combine the event-handling capability that we learned about in the previous chapter with the drawing facilities we'll explore in this chapter, so that we can implement an interactive Graphical User Interface for creating a sketch.

By the end of this chapter you will have learned:

• What components are available for creating a GUI.

• How coordinates are defined for drawing on a component.

• How you implement drawing on a component.

• How to structure the components in a window for drawing.

• What kinds of shapes you can draw on a component.

• How you implement mouse listener methods to enable interactive drawing operations.
Using the Model/View Architecture

We need to develop an idea of how we're going to manage the data for a sketch in the Sketcher program before we start drawing a sketch, because this will affect where and how we handle events. We already have a class that defines an application window, SketchFrame, but this class would not be a very sensible place to store the underlying data that defines a sketch. For one thing, we'll want to save a sketch in a file, and serialization is the easiest way to do that. If we're going to use serialization to store a sketch, we won't want all the stuff in the implementation of the SketchFrame class muddled up with the data relating to the sketch we have created.

For another, it will make the program easier to implement if we separate out the basic data defining a sketch from the definition of the GUI. This will be along the lines of the MVC architecture that we first mentioned in Chapter 15, a variant of which is used in the definition of Swing components.

Ideally, we should manage the sketch data in a class designed specifically for that purpose — this class will be the model for a sketch.

A class representing a view of the data in the model class will display the sketch and handle user interactions — so this class will combine viewing functions and a sketch controller. The general GUI creation and operations not specific to a view will be dealt with in the SketchFrame class. This is not the only way of implementing the things we want in the Sketcher program, but it's quite a good way.

Our model object will contain a mixture of text and graphics that will go to make up a sketch. We'll call our model class SketchModel and we'll call the class representing a view of the model SketchView, although we won't be adding the view to the program until the next chapter. The following diagram illustrates the relationships between the classes we will have in Sketcher.

The application object will have overall responsibility for managing links between the other objects involved in the program. Any object that has access to the application object will be able to communicate with any other object as long as the application class has methods to make each of the objects available. Thus the application object will act as the communication channel between objects.

Note that SketchFrame is not the view class — it just defines the application window and the GUI components associated with that. When we create a SketchView object in the next chapter, we'll arrange to insert the SketchView object into the content pane of the SketchFrame object, and manage it using the layout manager for the content pane. By defining the view class separately from the application class, we separate the view of a sketch from the menus and other components we use to interact with the program. One benefit of this is that the area in which we display the document has its own coordinate system, independent of that of the application window.
To implement the foundations for the model/view design in Sketcher we need to define classes for the model and the view, at least in outline. The class to contain the data defining a sketch we can define in skeleton form as:
Coordinate Systems in Components

In Chapter 15, we saw how your computer screen has a coordinate system that is used to define the position and size of a window. We also saw how we can add components to a container with their position established by a layout manager. This coordinate system is analogous to the screen coordinate system. The origin is at the top-left corner of the container, with the positive $x$-axis running horizontally from left to right, and the positive $y$-axis running from top to bottom. The positions of buttons in a JWindow or a JFrame object are specified as a pair of $(x, y)$ pixel coordinates, relative to the origin at the top-left corner of the container object on the screen. Below you can see the coordinate system for the Sketcher application window.

![Coordinate System for Sketcher](image)

Of course, the layered pane for the window object will have its own coordinate system with the origin in the top-left corner of the pane, and this is used to position the menu and the content pane. The content pane will have its own coordinate system too that will be used to position the components it contains.

It's not just containers and windows that have their own coordinate system: each JButton object also has its own system, as do JToolBar objects. In fact, every component has its own coordinate system.

![Coordinate Systems for JButton and JToolBar](image)

It's clear that a container needs a coordinate system for specifying the positions of the components it contains. You also need a coordinate system to draw on a component to draw a line for instance you need to be able to specify where it begins and ends in relation to the component and while the coordinate system here is similar to that used for positioning components in a container, it's not exactly the same. It's more complicated when you are drawing but for very good reasons. Let's see how the coordinate system for drawing works.
Drawing on a Component

Before we get into the specifics of how you draw on a component, let's understand the principle ideas behind it. When you draw on a component using the Java 2D capabilities, there are two coordinate systems involved. When you draw something—a line or a curve for instance—you specify the line or the curve in a device-independent logical coordinate system called the user coordinate system for the component, or user space. By default this coordinate system has the same orientation as the system that we discussed for positioning components in containers. The origin is at the top-left corner; the positive x-axis runs from left to right, and the positive y-axis from top to bottom.

Coordinates are usually specified as floating point values, although integers can also be used.

A particular graphical output device will have its own device coordinate system, or device space. This has the same orientation as the default user coordinate system, but the coordinate units depend on the characteristics of the device. Your display, for instance, will have a different device coordinate system for each configuration of the screen resolution, so the coordinate system when your display is set to 1024x768 resolution will be different from the coordinate system for 800x600 pixels.

Note

Incidentally, the drawing process is often referred to as rendering, since graphical output devices are generally raster devices and the drawing elements such as lines, rectangles, text, and so on need to be rendered into a rasterized representation before they can be output to the device.

Having a device-independent coordinate system for drawing means that you can use essentially the same code for outputting graphics to a variety of different devices to your display screen, for example, or to your printer even though these devices themselves have quite different coordinate systems with different resolutions. The fact that your screen might have 90 pixels per inch while your printer may have 600 dots per inch is automatically taken care of. Java 2D will deal with converting your user coordinates to the device coordinate system that is specific to the output device you are using.

With the default mapping from user coordinates to device coordinates, the units for user coordinates are assumed to be 1/72 of an inch. Since for most screen devices the pixels are approximately 1/72 inch apart, the conversion amounts to an identity transformation. If you want to use user coordinates that are in some other units, you have to
provide for this yourself. We will look into the mechanism that you would use to do this when we discuss transformations in the next chapter.

**Graphics Contexts**

The user coordinate system for drawing on a component using Java 2D is encapsulated in an object of type Graphics2D, which is usually referred to as a graphics context. It provides all the tools you need to draw whatever you want on the surface of the component. A graphics context enables you to draw lines, curves, shapes, filled shapes, as well as images, and gives you a great deal of control over the drawing process.

The Graphics2D class is derived from the Graphics class that defined device contexts in earlier versions of Java, so if you feel the need to use the old drawing methods, they are all inherited in the Graphics2D class. We will be concentrating on the new more powerful and flexible facilities provided by Graphics2D but, as you will see, references to graphics contexts are usually passed around as type Graphics so you need to be aware of it. Note that both the Graphics and Graphics2D classes are abstract classes, so you can't create objects of either type directly. An object representing a graphics context is entirely dependent on the component to which it relates, so a graphics context is always obtained for use with a particular component.

The Graphics2D object for a component takes care of mapping user coordinates to device coordinates, so it contains information about the device that is the destination for output as well as the user coordinates for the component. The information required for converting user coordinates to device coordinates is encapsulated in three different kinds of object:

- A GraphicsEnvironment object encapsulates all the graphics devices (as GraphicsDevice objects) and fonts (as Font objects) that are available on your computer.

- A GraphicsDevice object encapsulates information about a particular device such as a screen or a printer, and stores it in one or more GraphicsConfiguration objects.

- A GraphicsConfiguration object defines the characteristics of a particular device such as a screen or a printer. Your display screen will typically have several GraphicsConfiguration objects associated with it, each corresponding to a particular combination of screen resolution and number of displayable colors.

The graphics context also maintains other information necessary for drawing operations such as the drawing color, the line style and the specification of the fill color and pattern for filled shapes. We will see how to work with these attributes in examples later in this chapter.

Since a graphics context defines the drawing context for a specific component, before you can draw on a component you must have a reference to its graphics context object. For the most part, you will draw on a component by implementing the paint() method that is called whenever the component needs to be reconstructed. An object representing the graphics context for the component is passed as an argument to the paint() method, and you use this to do the drawing. The graphics context includes all the methods that you use to draw on a component and we will be looking into many of these in this chapter.
The paint() method is not the only way of drawing on a component. You can obtain a graphics context for a component at any time just by calling its getGraphics() method.

There are occasions when you want to get a component redrawn while avoiding a direct call of the paint() method. In such cases you should call repaint() for the component. There are five versions of this method that you can use; we'll look at four:

<table>
<thead>
<tr>
<th>repaint() Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>repaint()</td>
<td>Causes the entire component to be repainted by calling its paint() method after all of the currently outstanding events have been processed.</td>
</tr>
<tr>
<td>repaint(long msec)</td>
<td>Requests that the entire component is repainted within msec milliseconds.</td>
</tr>
<tr>
<td>repaint(int msec, int x, int y, int width, int height)</td>
<td>Adds the region specified by the arguments to the dirty region list if the component is visible. The dirty region list is simply a list of areas of the component that need to be repainted. The component will be repainted by calling its paint() method when all currently outstanding events have been processed or within msec milliseconds. The region is the rectangle at position (x, y) with the width and height as specified by the last two arguments.</td>
</tr>
<tr>
<td>repaint(Rectangle rect)</td>
<td>Adds the rectangle specified by rect to the dirty region list if the component is visible.</td>
</tr>
</tbody>
</table>

You will find that the first and the last methods are the ones you use most of the time.

That's enough theory for now. Time to get a bit of practice.

Let's get an idea of how we can draw on a component by drawing on the SketchView object that we added to Sketcher. All we need to do is implement the paint() method in the SketchView class.

**Try It Out  Drawing in a View**

Add the following implementation of the method to the SketchView class:

```java
import javax.swing.JComponent;
```
Shapes

Classes that define geometric shapes are contained in the java.awt.geom package, so to use them in a class we will need an import statement for this package at the beginning of the class file. You can add one to SketchView.java right now if you like. While the classes that define shapes are in java.awt.geom, the Shape interface is defined in java.awt, so you will usually need to import class names from both packages into your source file.

Any class that implements the Shape interface defines a shape visually it will be some composite of straight lines and curves. Straight lines, rectangles, ellipses and curves are all shapes.

A graphic context knows how to draw Shape objects. To draw a shape on a component, you just need to pass the object defining the shape to the draw() method for the Graphics2D object for the component. To look at this in detail, we'll split the shapes into three groups, straight lines and rectangles, arcs and ellipses, and freeform curves. First though, we must take a look at how points are defined.

Classes Defining Points

There are two classes in the java.awt.geom package that define points, Point2D.Float and Point2D.Double. From the class names you can see that these are both inner classes to the class Point2D, which also happens to be an abstract base class for both too. The Point2D.Float class defines a point from a pair of (x,y) coordinates of type float, whereas the Point2D.Double class defines a point as a coordinate pair of type double. The Point class in the java.awt package also defines a point, but in terms of a coordinate pair of type int. This class also has Point2D as a base.

The Point class actually predates the Point2D class, but the class was redefined to make it a subclass of Point2D when Point2D was introduced, hence the somewhat unusual class hierarchy with only two of the subclasses as inner classes. The merit of this arrangement is that all of the subclasses inherit the methods defined in the Point2D class, so operations on each of the three kinds of point are the same.

The three subclasses of Point2D define a default constructor that defines the point 0,0, and a constructor that accept a pair of coordinates of the type appropriate to the class type.

The operations that each of the three concrete point classes inherit are:

1. **Accessing coordinate values:**
The `getX()` and `getY()` methods return the x and y coordinates of a point as type double, regardless of how the coordinates are stored. These are abstract methods in the `Point2D` class so they are defined in each of the subclasses. Although you get coordinates as double values from all three concrete classes via these methods you can always access the coordinates with their original type directly since the coordinates are stored in public fields with the same names, x and y, in each case.

2.

**Calculating the distance between two points:**

You have no less than three overloaded versions of the `distance()` method for calculating the distance between two points, and returning it as type double:

<table>
<thead>
<tr>
<th>Method Description</th>
<th>Method Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a static version of the method that calculates the distance between the points <code>x1,y1</code> and <code>x2,y2</code>.</td>
<td><code>distance(double x1, double y1, double x2, double y2)</code></td>
</tr>
<tr>
<td>Calculates the distance from the current point (the object for which the method is called) and the point <code>xNext,yNext</code>.</td>
<td><code>distance(double xNext, double yNext)</code></td>
</tr>
<tr>
<td>Calculates the distance from the current point to the point, <code>nextPoint</code>. The argument can be any of the subclass types, <code>Point, Point2D.Float</code> or <code>Point2D.Double</code>.</td>
<td><code>distance(Point2D nextPoint)</code></td>
</tr>
</tbody>
</table>

Here’s how you might calculate the distance between two points:
Filling Shapes

Once you know how to create and draw a shape, filling it is easy. You just call the fill() method for the Graphics2D object and pass a reference of type Shape to it. This works for any shape but for sensible results the boundary should be closed.

Let's try it out by modifying the applet example that displayed stars.

Try It Out   Filling Stars

To fill the stars we just need to call the fill() method for each star in the paint() method of the StarPane object. Modify the paint() method as follows:

    public void paint(Graphics g) {
Managing Shapes

When we create shapes in Sketcher, we'll have no idea of the sequence of shape types that will occur. This is determined totally by the person using the program to produce a sketch. We'll therefore need to be able to draw shapes and perform other operations on them without knowing what they are - and polymorphism can help here.

We don't want to use the shape classes defined in java.awt.geom directly as we will want to add our own attributes such as color or line style, and store them as part of the object. We could consider using the shape classes as base classes for our shapes, but we couldn't use the GeneralPath class in this scheme of things because, as we have already seen, it's final and we might not want this restriction. We could consider defining an interface that all our shape classes would implement. However, there will be some methods that have a common implementation in all our shape classes so we would need to repeat this code in every class.

Taking all of this into account, the easiest approach might be to define a common base class for our shape classes, and include a member in each class to store a shape object of one kind or another. We'll then be able to include a polymorphic method to return a reference to a shape as type Shape for use with the draw() method of a Graphics2D object.

We can start by defining a base class, Element, from which we'll derive the classes defining specific types of shapes. The Element class will have data members that are common to all types of shapes, and we can put the methods that we want to execute polymorphically in this class too. All we need to do is make sure that each shape class that is derived from the Element class has its own implementation of these methods.

The diagram shows the initial members that we will declare in the Element base class. The only data member for now is the color member to store the color of a shape. The getShape() and getBounds() methods will be abstract here since the Element class is not intended to define a shape, but we will be able to implement the getColor() method in this class. The other methods will be implemented by the subclasses of Element.

Initially, we'll define the five classes shown in the diagram that represent shapes, with the Element class as a base. They provide objects that represent straight lines, rectangles, circles, freehand curves and blocks of text. These classes will all inherit the data members that we define for the Element class. As you can see from the names of our shape classes, they are all nested classes to the class Element. The Element class will serve as the base class, as well as house our shape classes. This will avoid any possible confusion with other classes that might have names such as Line or Circle for instance. Since there will be no Element objects around, we will declare our shape classes as static members of the Element class.

We can now define the base class, Element. Note that this won't be the final version, as we'll be adding more
functionality in later chapters. Here's the code that needs to go in Element.java in the Sketcher directory:
Drawing Using the Mouse

We've drawn shapes so far using data internal to the program. In our Sketcher program we want to be able to draw a shape using the mouse in the view, and then store the finished shape in the model. We want the process to be as natural as possible, so we'll implement a mechanism that allows you to draw by pressing the left mouse button (more accurately, button 1) and dragging the cursor to draw the selected type of shape. So for a line, the point where you depress the mouse button will be the start point for the line, and the point where you release the button will be the end point.

As you drag the mouse with the button down, we'll display the line as it looks at that point. Thus the line will be displayed dynamically all the time the mouse cursor is being dragged and the left button remains pressed. This process is called rubber-banding.

We can use essentially the same process of pressing the mouse button and dragging the cursor for all four of our shapes. Thus two points will define each shape - the cursor position where the mouse button is pressed, and the cursor position where the mouse button is released, (plus the color for the shape, of course). This implies that our shape constructors will all have three parameters, the two points and the color. Let's look at how we handle mouse events to make this work.

Handling Mouse Events

Because all the drawing operations for a sketch will be accomplished using the mouse, we must implement the process for creating elements within the methods that will handle the mouse events. The mouse events we're interested in will originate in the SketchView object because that's where we'll be drawing shapes. We will make the view responsible for handling all its own events, which will include events that occur in the drawing process as well as interactions with existing shapes.

Drawing a shape, such as a line, interactively will involve us in handling three different kinds of mouse event. Let's summarize what they are, and what we need to do when they occur:

<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Event</td>
<td>Action</td>
</tr>
<tr>
<td>------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Left Button (Button 1) pressed</td>
<td>Save the cursor position somewhere as the starting point for the line. We'll store this in a data member of the inner class to SketchView that we'll create to define listeners for mouse events.</td>
</tr>
<tr>
<td>Mouse dragged</td>
<td>Save the current cursor position somewhere as the end point for the line. Erase any previously drawn temporary line, and create a new temporary line from the starting point that was saved initially. Draw the new temporary line.</td>
</tr>
<tr>
<td>Left Button (Button 1) released</td>
<td>If there's a reference to a temporary line stored, add it to the sketch model, and redraw it.</td>
</tr>
</tbody>
</table>

You'll remember from the previous chapter that there are two mouse listener interfaces: MouseListener which has methods for handling events that occur when the mouse buttons are pressed or released and MouseMotionListener which has methods for handling events that arise when the mouse is moved. You will also recall that the MouseInputAdapter class implements both, and since we need to implement methods from both interfaces, we'll add an inner class to the SketchView class that extends the MouseInputAdapter class.

Since there's quite a lot of code involved in this, we will first define the bare bones of the class to handle mouse events, and then build in the detail until it does what we want.

**Try It Out  Implementing a Mouse Listener**

Add the following class outline as an inner class to SketchView:

```java
import javax.swing.JComponent;
```
Defining Our Own Shape Classes

All the classes that define shapes in Sketcher will be static nested classes of the Element class. As we have already said, as well as being a convenient way to keep our shape class definitions together, this will also avoid possible conflict with classes such as the Rectangle class in the Java class library.

We can start with the simplest — a class representing a line.

Defining Lines

A line will be defined by two points and its color. We can define the Line class as a nested class in the base class Element, as follows:

```java
import java.awt.Color;
```
Summary

In this chapter you have learned how to draw on components and how you can use mouse listeners to implement a drawing interface. The important points we have covered in this chapter are:

- A Graphics2D component represents the drawing surface of the component.

- You draw on a component by calling methods for its Graphics2D object.

- The user coordinate system for drawing on a component has the origin in the top-left corner of the component by default with the positive x-axis from left to right, and the positive y-axis from top to bottom. This is automatically mapped to the device coordinate system, which is in the same orientation.

- You normally draw on a component by implementing its paint() method. The paint() method is passed a Graphics2D object that is the graphics context for the component but as type Graphics. You must cast the Graphics object to type Graphics2D to be able to access the Graphics2D class methods. The paint() method is called whenever the component needs to be redrawn.

- You can't create a Graphics2D object. If you want to draw on a component outside of the paint() method, you can obtain a Graphics2D object for the component by calling its getGraphics() method.

- There is more than one drawing mode that you can use. The default mode is paint mode, where drawing overwrites the background pixels with pixels of the current color. Another mode is XOR mode where the current color is combined with the background color. This is typically used to alternate between the current color and a color passed to the setXORMode() method.

- The Graphics2D class defines methods for drawing outline shapes as well as filled shapes.

- The java.awt.geom package defines classes that represent 2D shapes.
Exercises

1. Add the code to the Sketcher program to support drawing an ellipse.

2. Modify the Sketcher program to include a button for switching fill mode on and off.

3. Extend the classes defining rectangles, circles and ellipses to support filled shapes.

4. Extend the curve class to support filled shapes.

5. (Harder for curve enthusiasts!) Implement an applet to display a curve as multiple CubicCurve2D objects from points on the curve entered by clicking the mouse. The applet should have two buttons—one to clear the window and allow points on the curve to be entered, and the other to display the curve. Devise your own scheme for default control points.

6. (Also harder!) Modify the previous example to ensure that the curve is continuous—this implies that the control points either side of an interior point, and the interior point itself, should be on a straight line. Allow control points to be dragged with the mouse, but still maintaining the continuity of the curve.
Chapter 19: Extending the GUI

Overview

In this chapter we will investigate how we can improve the GUI for Sketcher. After adding a status bar, we will investigate how to create dialogs, and how we can use them to communicate with the user and to manage input. Another GUI capability we will be exploring is pop-up menus, and we will be using these to enhance the Sketcher application. All of this will give you a lot more practice in implementing event listeners and more besides.

In this chapter you will learn:

• How to create a status bar
• How to create a dialog
• What a modal dialog is and how it differs from a non-modal dialog
• How to create a message box dialog
• How you can use components in a dialog to receive input
• What a pop-up menu is
• What context menus are and how you can implement them
Creating a Status Bar

One limitation of the Sketcher program as it stands is that you have no direct feedback on what current element type and color have been selected. As a gentle start to this chapter, let's fix that now. A window status bar is a common and very convenient way of displaying the status of various application parameters, each in its own pane.

We can make up our own class, StatusBar for instance, that will define a status bar. Ideally we would design a class for a generic status bar and customize it for Sketcher, but we will take the simple approach of designing a class that is specific to Sketcher. The JPanel class would be a good base for our StatusBar class since it represents a panel, and we can add objects representing status bar panes to it. We can use the JLabel class as a base for defining status bar panes and add sunken borders to them for a distinct appearance.

Let's start with a status bar at the bottom of Sketcher with two panes to show the current element type and color. Then we will know exactly what we are about to draw. We can start by defining the StatusBar class that will represent the status bar in the application window, and we'll define the StatusPane class as an inner class to StatusBar.

Try It Out   Defining a Status Bar Class

Here's an initial stab at the definition for the StatusBar class:
Using Dialogs

A dialog is a window that is displayed within the context of another window—its parent. You use dialogs to manage input that can't be handled conveniently through interaction with the view: selecting from a range of options for instance, or enabling data to be entered from the keyboard. You can also use dialogs for information messages or warnings. The JDialog class in the javax.swing package defines dialogs, and a JDialog object is a specialized sort of Window. A JDialog object will typically contain one or more components for displaying information or allowing data to be entered, plus buttons for selection of dialog options (including closing the dialog), so there's quite a bit of work involved in putting one together. However, for many of the typical dialogs you will want to use, the JOptionPane class provides an easy shortcut to creating dialogs. Below is a dialog that we'll create later in this chapter using just one statement.

We'll use this dialog to provide a response to clicking on a Help/About menu item that we will add to Sketcher in a moment. First though, we need to understand a little more about how dialogs work.

Modal and Non-modal Dialogs

There are two different kinds of dialog that you can create, and they have distinct operating characteristics. You have a choice of creating either a modal dialog or a non-modal dialog. When you display a modal dialog by selecting a menu item or clicking a button, it inhibits the operation of any other windows in the application until you close the dialog. The dialog opposite that displays a message is a modal dialog. The dialog window retains the focus as long as it is displayed and operation of the application cannot continue until you click the OK button. Modal dialogs that manage input will normally have at least two buttons, an OK button that you use to accept whatever input has been entered and then close the dialog, and a Cancel button to just close the dialog and abort the entry of the data. Dialogs that manage input are almost always modal dialogs, simply because you won't generally want to allow other interactions to be triggered until your user's input is complete.

A non-modal dialog can be left on the screen for as long as you want, since it doesn't block interaction with other windows in the application. You can also switch the focus back and forth between using a non-modal dialog and using any other application windows that are on the screen.

Whether you create a modal or a non-modal dialog is determined either by an argument to a dialog class constructor, or by which constructor you choose, since some of them create non-modal dialogs by default. The default, no-argument JDialog constructor creates a non-modal dialog with an empty title bar. Since you have no provision for specifying a parent window for the dialog, a shared hidden frame will be used as the parent in this case. You have a choice of five constructors for creating a JDialog object where the parent can be a window of type Frame or JFrame:
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDialog(Frame parent)</td>
<td>(empty)</td>
<td>parent</td>
</tr>
<tr>
<td>JDialog(Frame parent, String title)</td>
<td>title</td>
<td>parent</td>
</tr>
<tr>
<td>JDialog(Frame parent, boolean modal)</td>
<td>(empty)</td>
<td>parent</td>
</tr>
<tr>
<td>JDialog(Frame parent, String title, boolean modal)</td>
<td>title</td>
<td>parent</td>
</tr>
<tr>
<td>JDialog(Frame parent, String title, boolean modal, GraphicsConfiguration gc)</td>
<td>title</td>
<td>parent</td>
</tr>
</tbody>
</table>

Clearly since the first parameter is of type Frame, you can supply a reference of type Frame or type JFrame. There are a further five constructors for creating JDialog objects with a Dialog or JDialog object as the parent. The only difference between these and the ones in the table overleaf is that the type of the first parameter is Dialog rather than Frame. Any of these constructors can throw an exception of type HeadlessException if the system on which the code is executing does not have a display attached.

After you've created a JDialog object using any of the constructors, you can change the kind of dialog window it will produce from modal to non-modal, or vice versa, by calling the setModal() method for the object. If you specify the argument to the method as true, the dialog will be modal, and a false argument will make it non-modal. You can also check whether a JDialog object is modal or not. The isModal() method for the object will return true if it represents a modal dialog, and false otherwise.

All JDialog objects are initially invisible, so to display them you must call the setVisible() method for the JDialog object with the argument true. This method is inherited from the Component class via the Container and Window classes. If you call setVisible() with the argument false, the dialog window is removed from the screen. Once you've
displayed a modal dialog window, the user can't interact with any of the other application windows until you call setVisible() for the dialog object with the argument false, so you typically do this in the event handler which is called to close the dialog. Note that the setVisible() method only affects the visibility of the dialog. You still have a perfectly good JDialog object so that when you want to display the dialog again, you just call its setVisible() method with an argument set to true. Of course, if you call dispose() for the JDialog object, or set the default close operation to DISPOSE_ON_CLOSE, then you won't be able to use the JDialog object again.

To set or change the title bar for a dialog, you just pass a String object to the setTitle() method for the JDialog object. If you want to know what the current title for a dialog is, you can call the getTitle() method which will return a String object containing the title bar string.

Dialog windows are resizable by default, so you can normally change the size of a dialog window by dragging its boundaries. If you don't want to allow a dialog window to be resized, you can inhibit this by calling the setResizable() for the JDialog object with the argument as false. An argument value of true re-enables the resizing capability.

A Simple Modal Dialog

The simplest kind of dialog is one that just displays some information. We could see how this works by adding a Help menu with an About menu item, and then displaying an About dialog to provide information about the application.

Let's derive our own dialog class from JDialog so we can create an About dialog.

Try It Out   Defining the AboutDialog Class

The constructor for our AboutDialog class will need to accept three arguments - the parent Frame object, which will be the application window in Sketcher, a String object defining what should appear on the title bar and a String object for the message we want to display. We'll only need one button in the dialog window, an OK button to close the dialog. We can make the whole thing self-contained by making the AboutDialog class the action listener for the button, and since it's only relevant in the context of the SketchFrame class, we can define it as an inner class.

```
public class SketchFrame extends Jframe implements Constants {
```
# Instant Dialogs

The JOptionPane class in the javax.swing package defines a number of static methods that will create and display standard modal dialogs for you. The simplest dialog you can create this way is a message dialog rather like our About message dialog. The following methods produce message dialogs:

<table>
<thead>
<tr>
<th>Dial-a-Dialog Methods</th>
<th>Description</th>
</tr>
</thead>
</table>
| showMessageDialog (Component parent, Object message) | This method displays a modal dialog with the default title "Message". The first argument is the parent for the dialog. The Frame object containing the component will be used to position the dialog. If the first argument is null, a default Frame object will be created and that will be used to position the dialog centrally on the screen.

The second argument specifies what is to be displayed in addition to the default OK button. This can be a String object specifying the message or an Icon object defining an icon to be displayed. It can also be a Component, in which case the component will be displayed. If some other type of object is passed as the second argument, its toString() method will be called, and the String object that is returned will be displayed. You will usually get a default icon for an information message along with your message.

You can pass multiple items to be displayed by passing an array of type Object[] for the second argument. Each array element will be processed as above according to its type and they will be arranged in a vertical stack. Clever, eh? |
| showMessageDialog (Component parent, Object message, String title, int messageType) | This displays a dialog as above, but with the title specified by the third argument. The fourth argument, messageType, can be: |

- ERROR_MESSAGE
- INFORMATION_MESSAGE
- WARNING_MESSAGE
- QUESTION_MESSAGE
- PLAIN_MESSAGE

These determine the style of the message constrained by the current look and feel. This will usually include a default icon, such as a question mark for QUESTION_MESSAGE. |
showMessageDialog
(Component parent, Object message, String title, int messageType, Icon icon)

This displays a dialog as above except that the icon will be what you pass as the fifth argument. Specifying a null argument for the icon will produce a dialog the same as the previous version of the method.

We could have used one of these for the About dialog instead of all that messing about with inner classes. Let's see how.

Try It Out  An Easy About Dialog

Delete the inner class, AboutDialog, from SketchFrame we won't need that any longer. Change the implementation of the actionPerformed() method in the SketchFrame class to:

```java
public void actionPerformed(ActionEvent e) {
```
Pop-up Menus

The javax.swing package defines a class, JPopupMenu, that represents a menu that you can pop up at any position within a component, but conventionally you display it at the current mouse cursor position when a particular mouse button is pressed. There are two constructors in the PopupMenu class: one to which you pass a String object that defines a name for the menu, and a default constructor that defines a menu without a name. If you specify a name for a pop-up menu with a statement such as,
Transforming the User Coordinate System

We said when we started learning how to draw on a component that the drawing operations are specified in a user-coordinate system, and the user coordinates are converted to a device coordinate system. The conversion of coordinates from user system to device system is taken care of by the methods in the graphics context object that we use to do the drawing, and they do this by applying a transformation to the user coordinates. The term 'transformation' refers to the computational operations that result in the conversion.

By default, the origin, the (0, 0) point, in the user coordinate system corresponds to the (0, 0) point in the device coordinates system. The axes are also coincident too, with positive $x$ heading from left to right, and positive $y$ from top to bottom. However you can move the origin of the user coordinate system relative to its default position. Such a move is called a translation.

A fixed value, deltaX say, is added to each $x$ coordinate, and another value, deltaY say, is added to every $y$ coordinate and the effect of this is to move the origin of the user coordinate system relative to the device coordinate system: everything will be shifted to the right and down compared to where it would have been without the translation. Of course, the deltaX and deltaY values can be negative, in which case it would shift things to the left and up.

A translation is one kind of affine transformation. Affine is a funny word. Some say it goes back to Laurel and Hardy where Ollie says, "This is affine mess you've got us into", but I don't subscribe to that. An affine transformation is actually a linear transformation that leaves straight lines still straight and parallel lines still parallel. As well as translations, there are other kinds of affine transformation that you can define:

- **Rotation** the user coordinates system is rotated through a given angle about its origin.

- **Scale** the $x$ and $y$ coordinates are each multiplied by a scaling factor, and the multipliers for $x$ and $y$ can be different. This enables you to enlarge or reduce something in size. If the scale factor for one coordinate axis is negative, then objects will be reflected in the other axis. Setting the scale factor for $x$ coordinates to 1, for example, will make all positive coordinates negative and vice versa so everything is reflected in the $y$ axis.
Shear  this is perhaps a less familiar operation. It adds a value to each \( x \) coordinate that depends on the \( y \) coordinate, and adds a value to each \( y \) coordinate that depends on the \( x \) coordinate. You supply two values to specify a shear, \( s_X \) and \( s_Y \) say, and they change the coordinates in the following way:

Each \( x \) coordinate becomes \((x + s_X \times y)\)
Each \( y \) coordinate becomes \((y + s_Y \times x)\)

The effect of this can be visualized most easily if you first imagine a rectangle that is drawn normally. A shearing transform can squash it by tilting the sides rather like when you flatten a carton but keep opposite sides straight and parallel.

The illustration shows:

- A rotation of \(-\pi/4\) radians, which is 45 degrees. Rotation angles are expressed in radians and a positive angle rotates everything from the positive x-axis towards the positive y-axis therefore clockwise. The rotation in the illustration is negative and therefore counterclockwise.

- A scaling transformation corresponding to an \( x \) scale of 2.5 and a \( y \) scale of 1.5.

- A shearing operation where only the \( x \) coordinates have a shear factor. The factor for the \( y \) coordinates is 0 so they are unaffected and the transformed shape is the same height as the original.

The AffineTransform Class

In Java, the AffineTransform class in the java.awt.geom package represents an affine transformation. Every Graphics2D graphics context has one. The default AffineTransform object in a graphics context is the identity transform, which leaves user coordinates unchanged. It is applied to the user coordinate system anyway for everything you draw, but all the coordinates are unaltered by default. You can retrieve a copy of the current transform for a graphics context object by calling its getTransform() method. For example:
Choosing Custom Colors

We made provision in the status bar for showing a custom color. It would be a shame not to make use of this, so let's add a dialog to enable any color to be chosen. This is going to be a lot easier than you imagine.

To keep it simple, we will implement this as a facility on the general pop-up menu, although in practice you would probably want it accessible from the main menu and the toolbar. We will add a member to the SketchFrame class for the menu item:
Summary

In this chapter you have learned how to use dialogs to manage data input. You have also learned how to implement context menus, which can bring a professional feel to the GUI in your applications. You have applied scrollbars to varying data values as well as scrolling a window, so you should be in a position to use them in whatever context you need.

The important points we have covered in this chapter are:

•

A modal dialog blocks input from other windows in the same application as long as it is displayed.

•

A non-modal dialog does not block input to other windows. You can switch the focus between a non-modal dialog and other windows in the application whenever necessary.

•

The JOptionPane class provides static methods for creating simple dialogs.

•

A pop-up menu is a menu that can be displayed at any point within the coordinate system of a component.

•

A context menu is a pop-up menu that is specific to what lies at the point where the menu is displayed so the contents of the menu depend on the context.

•

A context menu is displayed as a result of a pop-up trigger, which is usually a right mouse button click.

•

The AffineTransform class defines an affine transformation that can be applied to a graphics context and to a Shape object.

•

A Graphic2D object always contains an AffineTransform object, and the default transform leaves coordinates unchanged.
The transform for a graphics context is applied immediately before user coordinates for a shape are converted to device coordinates.

- There are four kinds of transform you can create: translations, rotations, scaling, and shearing.

- You can combine any number of transformations in a single AffineTransform object.
Exercises

1. Implement a dialog initiated from a toolbar button to select the current element color.

2. Add a menu item to the Element context menu that will display information about the element at the cursor in a dialog—what it is and its basic defining data.

3. Display a special context menu when the cursor is over a TEXT object that provides a menu option to edit the text through a dialog.

4. Change the implementations of the element classes to make use of the combined translate and rotate operation.

5. Add a toolbar button to switch highlighting on and off. The same button should turn it on when it is off and vice versa, so you need to change the button label appropriately.

6. Add a Scale menu item to the element context menu that will allow an element to be scaled by dragging the mouse cursor.

7. Implement a main menu item and a toolbar button for choosing a custom color.
Chapter 20: Filing and Printing Documents

Overview

In this chapter we will explore serializing and printing documents in an application, and adding these as the finishing touches to our Sketcher program. These capabilities are not available to an untrusted applet for security reasons, so everything we will cover here only applies to applications and trusted applets. Although we have already covered serialization in Chapter 11, you will find that there is quite a difference between understanding how the basic methods for object input and output work, and applying them in a practical context.

In this chapter you will learn:

• How to use the JFileChooser class

• How to save a sketch in a file as objects

• How to implement the Save As menu mechanism

• How to open a sketch stored in a file and integrate it into the application

• How to create a new sketch and integrate it into the application

• How to ensure that the current sketch is saved before the application is closed or a new sketch is loaded

• How printing in Java works
• How to print in landscape orientation rather than portrait orientation

• How to implement multipage printing

• How to output components to your printer
Serializing the Sketch

Our Sketcher program can only be considered to be a practical application if we can save sketches in a file and retrieve them later—in other words we need to implement serialization for a SketchModel object and use that to make the File menu work. Ideally, we want to be able to write the model for a sketch to a file and be able to read it back at a later date and reconstruct exactly the same model object. This naturally leads us to choose serialization as the way to do this, because the primary purpose of serialization is the accurate storage and retrieval of objects.

We've seen how to serialize objects, back in Chapter 11. All we need to do to serialize a sketch document in our Sketcher program is to apply what we learned there. Of course, there are quite a few classes involved in a sketch document but it will be remarkably easy, considering the potential complexity of a sketch—I promise!

Of course, saving a sketch on disk and reading it back from a file will be significantly more work than implementing serialization for the document. The logic of opening and saving files so as not to lose anything accidentally can get rather convoluted. Before we get into that, there is a more fundamental point we should address—our sketch doesn't have a name. We should at least make provision for assigning a file name to a sketch, and maybe display the name in the title bar of the application window.

Try It Out  Assigning a Document Name

Since the sketch is going to have a name, because we intend to store it somewhere, let's define a default directory to hold sketches. Add the following lines to the end of the Constants interface (Constants.java):
Implementing the Serializable Interface

As I hope you still remember, the fundamental step in making objects serializable is to implement the Serializable interface in every class that defines objects we want written to a file. We need a methodical approach here, so how about top-down starting with the SketchModel class.

Try It Out  Serializing SketchModel Objects

This is where we get a great deal from astonishingly little effort. To implement serialization for the SketchModel class you must first modify the class definition header to:
Supporting the File Menu

To support the menu items in the File menu, we must add some code to the actionPerformed() method in the FileAction class. We can try to put a skeleton together but a problem presents itself immediately: the ultimate source of an event will be either a toolbar button (a JButton object) or a menu item (a JMenuItem object) that was created from a FileAction object. How do we figure out what the action was that originated the event? We only have one definition of the actionPerformed() method shared amongst all FileAction class objects so we need a way to determine which particular FileAction object caused the event. That way we can decide what we should do in response to the event.

Each FileAction object stores a String that was passed to the constructor as the name argument, and was then passed on to the base class constructor. If only we had thought of saving it, we could compare the name for the current object with the name for each of the FileAction objects in the SketchFrame class. Then we could tell which object the actionPerformed() method was called for.

All is not lost though. We can call the getValue() method for the ActionEvent object to retrieve the name for the action object that caused the event. We can then compare that with the name for each of the FileAction objects that we store as members of the SketchFrame class. We can therefore implement the actionPerformed() member of the FileAction class like this:

```java
public void actionPerformed(ActionEvent e) {
```
Printing in Java

Printing is always a messy business—inevitably so, because you have to worry about tedious details such as the size of a page, the margin sizes, and how many pages you're going to need for your output. As you might expect, the process for printing an image is different from printing text and you may also have the added complication of several printers with different capabilities being available, so with certain types of documents you need to select an appropriate printer. The way through this is to take it one step at a time. Let's understand the general principles first.

There are five packages dedicated to supporting printing capabilities in Java. These are:

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.print</td>
<td>Defines classes and interfaces that enable you to determine what printers are available and what their capabilities are. It also enables you to identify types of documents to be printed.</td>
</tr>
<tr>
<td>javax.print.attribute</td>
<td>Defines classes and interfaces supporting the definition of sets of printing attributes. For example, you can define a set of attributes required by a particular document when it is printed, such as color output and two-sided printing, for instance.</td>
</tr>
<tr>
<td>javax.print.attribute.standard</td>
<td>Defines classes that identify a set of standard printing attributes.</td>
</tr>
<tr>
<td>javax.print.event</td>
<td>Defines classes that identify events that can occur while printing and interfaces that identify listeners for printing events.</td>
</tr>
<tr>
<td>java.awt.print</td>
<td>Defines classes and attributes for expediting the printing of 2D graphics and text.</td>
</tr>
</tbody>
</table>

The first four of these packages make up what is called the Print Service API that was added to Java in SDK 1.4. This allows printing on all Java platforms and has facilities for discovering and using multiple printers with varying capabilities. Since in all probability you have just a single printer available, we will concentrate in the first instance on understanding the classes and interfaces defined in the java.awt.print package that carry out print operations on a given printer, and stray into classes and interfaces from the other packages when necessary.

There are four classes in the java.awt.print package and we will be using all of them eventually:
An object of this class type controls printing to a particular print service (such as a printer or fax capability).

An object of this class type defines the size and orientation of a page that is to be printed.

An object of this class type defines the size and printable area of a sheet of paper.

An object of this class type defines a multipage document where pages may have different formats and require different rendering processes.

The PrinterJob class here drives the printing process. Don't confuse this with the PrintJob class in the java.awt package; this is involved in the old printing process introduced in Java 1.1 and the PrinterJob class now supersedes this. A PrinterJob class object provides the interface to a printer in your environment, and you use PrinterJob class methods to set up and initiate the printing process for a particular document. You start printing off one or more pages in a document by calling the print() method for the PrinterJob object.

A PageFormat object encapsulates information about a page, such as its dimensions, margin sizes, and orientation. An object of type Paper describes the characteristics of a physical sheet of paper that will be part of a PageFormat object. A Book object encapsulates a document consisting of a collection of pages that are typically processed in an individual way. We will be getting into the detail of how you work with these a little later in this chapter.

There are three interfaces in the java.awt.print package:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printable</td>
<td>Implemented by a class to print a single page.</td>
</tr>
<tr>
<td>Pageable</td>
<td>Implemented by a class to print a multipage document where each page may be printed by a different Printable object.</td>
</tr>
<tr>
<td>Printer Graphics</td>
<td>Declares a method for obtaining a reference to the PrinterJob object for use in a method that is printing a page.</td>
</tr>
</tbody>
</table>

When you print a page, an object of a class that implements the Printable interface determines what is actually printed. Such an object is referred to as a page painter.

The Printable interface only defines one method, print(), which is called by a PrinterJob object when a page should be printed. Therefore the print() method does the printing of a page. Note that we have mentioned two print() methods one defined in the PrinterJob class that you call to starting the printing process, and another declared in the Printable interface that you implement in your class that is to do the printing legwork for a single page.
The printing operation you must code when you implement the `print()` method declared in the `Printable` interface works through a graphics context object that provides the means for writing data to your printer. The first argument that is passed to your `print()` method when it is called by a `PrinterJob` object is a reference of type `Graphics` that represents the graphics context for the printer. The object that it references is actually of type `Graphics2D`, which parallels the process you are already familiar with for drawing on a component. Just as with writing to the display, you use the methods defined in the `Graphics` and `Graphics2D` classes to print what you want, and the basic mechanism for printing 2D graphics or text on a page is identical to drawing on a component. The `Graphics` object for a printer also happens to implement the `PrinterGraphics` interface (not to be confused with the `PrintGraphics` interface in the `java.awt` package!) that declares just one method, `getPrinterJob()`. You call this method to obtain a reference to the object that is managing the print process. You would do this if you need to call `PrinterJob` methods to extract information about the print job, such as the job name or the user name.

A class that implements the `Pageable` interface defines an object that represents a set of pages to be printed rather than a single page. You would implement this interface for more complicated printing situations where a different page painter may print each page using an individual `PageFormat` object. It's the job of the `Pageable` object to supply information to the `PrinterJob` object about which page painter and `PageFormat` object should be used to print each page. The `Pageable` interface declares three methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getNumberOfPages()</code></td>
<td>Returns the number of pages to be printed as type <code>int</code>, or the constant value <code>Pageable.UNKNOWN_NUMBER_OF_PAGES</code> if the number of pages is not known.</td>
</tr>
<tr>
<td><code>getPageFormat(int pageIndex)</code></td>
<td>Returns the <code>PageFormat</code> object describing the size and orientation of the page specified by the argument. An exception of type <code>IndexOutOfBoundsException</code> will be thrown if the page does not exist.</td>
</tr>
<tr>
<td><code>getPrintable(int pageIndex)</code></td>
<td>Returns a reference to the <code>Printable</code> object responsible for printing the page specified by the argument. An exception of type <code>IndexOutOfBoundsException</code> will be thrown if the page does not exist.</td>
</tr>
</tbody>
</table>

A `Book` object also encapsulates a document that consists of a number of pages, each of which may be processed individually for printing. The difference between this and an object of a class that implements `Pageable` is that you can add individual pages to a `Book` object programmatically, whereas a class implementing `Pageable` encapsulates all the pages. We will look at how both of these options work later in this chapter.

### Creating and Using `PrinterJob` Objects

Because the `PrinterJob` class encapsulates and manages the printing process for a given printer that is external to the JVM, you can't create an object of type `PrinterJob` directly using a constructor. You can obtain a reference to a `PrinterJob` object for the default printer on a system by calling the static method `getPrinterJob()` that is defined in the `PrinterJob` class:
Summary

In this chapter you have added full support for the File menu to the Sketcher application, for both sketch storage and retrieval, and for printing. You should find that the techniques that you have used here are readily applicable in other Java applications. The approach to saving and restoring a model object is not usually dependent on the kind of data it contains. Of course, if your application is a word processor, you will have a little more work to do taking care that the number of lines printed on each page is a whole number of lines. In other words you will have to make sure you avoid having the top half of a line of text on one page, and the bottom half on the next. There are other Java classes to help with that, however, and we don't really have the space to discuss them here but look them up the javax.swing.text package is a veritable gold mine for text handling!

If you have been following all the way with Sketcher, you now have an application that consists of well over a thousand lines of code, so you should be pretty pleased with yourself.

The important points we have covered in this chapter are:

- You can implement writing your model object to a file and reading it back by making it serializable.

- The JFileChooser class provides a generalized way for displaying a dialog to enable a file to be chosen.

- A printing operation is initiated by creating a PrinterJob object. This object encapsulates the interface to your printer and is used to manage the printing process.

- A PrintService object encapsulates a printer.

- A PageFormat object defines the format for a page, and methods for this object can provide information on the paper size and orientation and the printable area on the page.

- An object of type Paper defines a page.

- You can display a print dialog by calling the printDialog() method for a PrinterJob object. The no argument version of printDialog() will display the native print dialog, whereas the version accepting a single argument of
type PrintRequestAttributeSet displays a Java print dialog.

- Printing a page is always done by an object of a class that implements the Printable interface.

- You print a page by calling methods for the Graphics object passed to the print() method in the Printable interface by the PrinterJob object.

- You can manage multipage print jobs by implementing the Pageable interface in a class. This will enable different types of class object to be used to print different pages.

- A Book object can encapsulate a series of pages to be printed. Each Printable object that is appended to a book prints one or more pages in a given format.
Exercises

1. At the moment in Sketcher, you can close a file selection dialog with the approve button without a file being selected. Modify Sketcher so that the dialog will continue to be displayed until you select a file or close the dialog with the Cancel button.

2. Modify the Sketcher program to print the title at the top of the page on which the sketch is printed.

3. Modify the printing of a sketch so that a solid black boundary line is drawn around the sketch on the page.

4. Modify Sketcher to print a single sketch laid out on four pages. The sketch should be enlarged to provide the best fit on the four pages without distorting circles that is, the same scale should be applied to the $x$ and $y$ axes.

5. Use a Book object to print a cover page plus the sketch spread over four pages as in the previous exercise.
Chapter 21: Java and XML

Overview

Release 1.4 of the SDK introduced capabilities for processing Extensible Markup Language (XML) documents as part of the standard set of class libraries. These are collectively referred to as JAXP, the Java API for XML. In this chapter and the next we will be exploring not only how we can read XML documents, but also how we can create and modify them. This chapter provides a brief outline of XML and some related topics, plus a practical introduction to reading XML documents from within your Java programs using one of the two available mechanisms for this. In the next chapter we will discuss the second approach to reading XML documents, as well as how we can modify them and create new ones. Inevitably, we can only skim the surface in a lot of areas since XML itself is a huge topic. However, you should find enough in this chapter and the next to give you a good feel for what XML is about, and how you can handle XML documents in Java.

In this chapter you will learn:

- What a well-formed XML document is
- What constitutes a valid XML document
- What the components in an XML document are, and how they are used
- What a DTD is and how it is defined
- What namespaces are and why you use them
- What the SAX and DOM APIs are, and how they different.
How you read documents using SAX
XML

XML, or the eXtensible M arkup L anguage to give it its full title, is a system and hardware independent language for expressing data and its structure within an XML d ocument. An XML d ocument is a Unicode text file that contains the data together with markup that defines the structure of the data. Because an XML document is a text file, you can create XML using any plain text editor, although an editor designed for creating and editing XML will obviously make things easier. The precise definition of XML is in the hands of the World Wide Web Consortium (W3C), and if you want to consult the XML 1.0 specification, you can find it at http://www.w3.org/XML.

The term 'markup' derives from a time when the paper draft of a document to be printed was marked up by hand to indicate to the typesetter how the printed form of the document should look. Indeed the ancestry of XML can be traced back to a system that was originally developed by IBM in the 1960s to automate and standardize markup for system reference manuals for IBM hardware and software products. XML markup looks similar to HTML in that it consists of tags and attributes added to the text in a file. However, the superficial appearance is where the similarity between XML and HTML ends. XML and HTML are profoundly different in purpose and capability.

Firstly, although an XML document can be created, read, and understood by a person, XML is primarily for communicating data from one computer to another. XML documents will therefore more typically be generated and processed by computer programs. An XML document defines the structure of the data it contains so a program that receives it can properly interpret it. Thus XML is a tool for transferring information and its organization between computer programs. The purpose of HTML on the other hand is solely the description of how data should look when it is displayed or printed. The only structuring information that generally appears in an HTML document relates to the appearance of the data as a visible image. The purpose of HTML is data presentation.

Secondly, HTML provides you with a set of tags that is essentially fixed and geared to the presentation of data. XML is a language in which you can define new sets of tags and attributes to suit different kinds of data indeed to suit any kind of data including your particular data. Because XML is extensible, it is often described as a meta-language a language for defining new languages in other words. The first step in using XML to exchange data is to define the language that you intend to use for that purpose - in XML.

Of course, if I invent a set of XML markup to describe data of a particular kind, you will need to know the rules for creating XML documents of this type if you want to create, receive, or modify them. As we shall see, the definition of the markup that has been used within an XML document can be included as part of the document. It also can be provided as a separate entity, in a file identified by a URI for instance, that can be referenced within any document of that type. The use of XML has already been standardized for very diverse types of data. There are XML languages for describing the structures of chemical compounds and, musical scores, as well as plain old text such as in this book.

The Java API for XML P rocessing (JAXP) provides you with the means for reading, creating, and modifying XML documents from within your Java programs. In order to understand and use this API there are two basic topics you need to be reasonably familiar with:

-
What an XML document is for and what it consists of.

What a DTD is and how it relates to an XML document.

You also need to be aware of what an XML namespace is, if only because JAXP has methods relating to handling these. You can find more information on JAXP at http://sun.java.com/xml/jaxp/.

Just in case you are new to XML, we will briefly explore the basic characteristics of XML and DTDs before we start applying the classes and methods provided by JAXP to process XML documents. We will also briefly explore what XML namespaces are for. If you are already comfortable with these topics you can skip most of this chapter and pick up where we start talking about SAX. Let's start by looking into the general organization of an XML document.
XML Document Structure

An XML document basically consists of two parts, a **prolog** and a **document body**:

- **The prolog** provides information necessary for the interpretation of the contents of the document body. It contains two optional components, and since you can omit both, the prolog itself is optional. The two components of the prolog, in the sequence in which they must appear, are:

  - An **XML declaration** that defines the version of XML that applies to the document, and may also specify the particular Unicode character encoding used in the document and whether the document is standalone or not. Either the character encoding or the standalone specification can be omitted from the XML declaration but if they do appear they must be in the given sequence.

  - A **document type declaration** specifying an external **Document Type Definition** (DTD) that identifies markup declarations for the elements used in the body of the document, or explicit markup declarations, or both.

- **The document body** contains the data. It comprises one or more elements where each element is defined by a begin tag and an end tag. The elements in the document body define the structure of the data. There is always a single root element that contains all the other elements. All of the data within the document is contained within the elements in the document body.

**Processing instructions (PI)** for the document may also appear at the end of the prolog and at the end of the document body. Processing instructions are instructions intended for an application that will process the document in some way. You can include comments that provide explanations or other information for human readers of the XML document as part of the prolog and as part of the document body.

When an XML document is said to be **well-formed**, it just means that it conforms to the rules for writing XML, as defined by the XML specification. Essentially an XML document is well-formed if its prolog and body are consistent with the rules for creating these. In a well-formed document there must be only one root element and all elements must be properly nested. We will summarize more specifically what is required to make a document well-formed a little later in this chapter, after we have looked into the rules for writing XML.

An **XML processor** is a software module that is used by an application to read an XML document and gain access to the data and its structure. An XML processor also determines whether an XML document is well-formed or not. Processing instructions are passed through to an application without any checking or analysis by the XML processor. The XML specification describes how an XML processor should behave when reading XML documents, including what information should be made available to an application for various types of document content.
Here's an example of a well-formed XML document:

```xml
<proverb>Too many cooks spoil the broth.</proverb>
```

The document just consists of a root element that defines a proverb. There is no prolog and, formally, you don't have to supply one, but it would be much better if the document did include at least the XML version that is applicable, like this:

```xml
<?xml version="1.0"?>
```
Data Structure in XML

The ability to nest elements is fundamental to defining the structure of the data in a document. We can easily represent the structure of the data in our XML fragment defining an address, as shown below.

The structure follows directly from the nesting of the elements. The <address> element contains all of the others directly, so the nested elements are drawn as subsidiary or child elements of the <address> element. The items that appear within the tree structure, the elements and the data items, are referred to as nodes.

The diagram below shows the structure of the first circle definition in XML that we created in the previous section. Even though there's an extra level of elements in this diagram, there are strong similarities to the structure above.

We can see that both structures have a single root element, <address> in the first example and <circle> in the second. We can also see that each element contains either other elements or some data that is a segment of the document content. In both diagrams all the document content lies at the bottom. Nodes at the extremities of a tree are referred to as leaf nodes.

In fact an XML document always has a structure similar to this. Each element in a document can contain other elements, or text, or elements and text, or it can be empty.
Document Type Definitions

We have seen several small examples of XML and in each case it was fairly obvious what the content was meant to represent, but where are the rules that ensure such data is represented consistently and correctly in different documents? Do the <radius> and <position> elements have to be in that sequence in a <circle> element and could we omit one or other of them?

Clearly there has to be a way to determine what is correct and what is incorrect for any particular element in a document. As we mentioned earlier, a Document Type Definition (DTD) defines how valid elements are constructed for a particular type of document, so the XML for purchase order documents in a company could be defined by one DTD, and sales invoice documents by another. The document type definition for a document is specified in a document type declaration commonly known as a DOCTYPE declaration that appears in the document prolog following any XML declaration. A DTD essentially defines a vocabulary for describing data of a particular kind—the set of elements that you use to identify the data in other words. It also defines the possible relationships between these elements—how they can be nested. The contents of a document of the type identified by a particular DTD must be defined and structured according to rules that make up the DTD. Any document of a given type can be checked for validity against its DTD.

A DTD can be an integral part of a document but it is usually, and more usefully, defined separately. Including a DTD in an XML document makes the document self-contained, but it does increase its bulk. It also means that the DTD has to appear within each document of the same type. A separate DTD that is external to a document avoids this and provides a single reference point for all documents of a particular type. An external DTD also makes maintenance of the DTD for a document type easier as it only needs to be changed in one place for all documents that make use of it.

Let's look at how we identify the DTD for a document and then investigate some of the ways in which elements and their attributes can be defined in a DTD.

Declaring a DTD

You use a document type declaration (a DOCTYPE declaration) in the prolog of an XML document to specify the DTD for the document. An XML 1.0 document can only have one DOCTYPE declaration. You can include the markup declarations for elements used in the document explicitly within the DOCTYPE statement, in which case the declarations are referred to as the internal subset. You can also specify a URI that identifies the DTD for the document, usually in the form of a URL. In this case the set of declarations is referred to as the external subset. If you include explicit declarations as well as a URI referencing an external DTD, the document has both an internal and an external subset. Here is an example of an XML document that has an external subset:

<?xml version="1.0"?>
Rules for a Well-Formed Document

Now that we know a bit more about XML elements and what goes into a DTD, we can formulate what you must do to ensure your XML document is well-formed. The rules for a document to be well-formed are quite simple:

1. If the XML declaration appears in the prolog, it must include the XML version. Other specifications in the XML document must be in the prescribed sequence: character encoding then standalone specification.

2. If the document type declaration appears in the prolog the DOCTYPE name must match that of the root element and the markup declarations in the DTD must be according to the rules for writing markup declarations.

3. The body of the document must contain at least one element, the root element, which contains all the other elements, and an instance of the root element must not appear in the content of another element. All elements must be properly nested.

4. Elements in the body of the document must be consistent with the markup declarations identified by the DOCTYPE declaration.

The rules for writing an XML document are absolutely strict. Break one rule and your document is not well formed and will not be processed. This strict application of the rules is essential because we are communicating data and its structure. If any laxity were permitted it would open the door to uncertainty about how the data should be interpreted. HTML used to be quite different from XML in this respect. Until recently, the rules for writing HTML were only loosely applied by HTML readers such as web browsers.

For instance, even though a paragraph in HTML should be defined using a begin tag, `<p>`, and an end tag, `</p>`, you can usually get away with omitting the end tag, and you can use both capital and lower-case p, and indeed close a capital-case P paragraph with a lower-case p, and vice versa. You can often have overlapping tags in HTML and get away with that too. While it is not to be recommended, a loose application of the rules for HTML is not so harmful since HTML is only concerned with data presentation. The worst that can happen is that the data does not display quite as you intended.

Recently, the W3C has released a number of specifications that make HTML an XML language, and we can expect compliance within the next few years. The enduring problem is, of course, that the Internet has many years of material that is still very useful but that will never be well-formed XML, so browsers may never be fully XML compliant.
XML Namespaces

This is the last topic we need a little insight into before we get back into Java programming. Even though they are very simple, XML namespaces can be very confusing. The confusion arises because it is so easy to make assumptions about what they imply when you first meet them. Let's look briefly at why we have XML namespaces in the first place and then see what an XML namespace actually is.

We saw earlier that an XML document can only have one DOCTYPE declaration. This can identify an external DTD by a URI or include explicit markup declarations, or it may do both. What happens if we want to combine two or more XML documents that each have their own DTD into a single document? The short answer is we can't easily anyway. Since the DTD for each document will have been defined without regard for the other, element name collisions are a real possibility. It may be impossible to differentiate between different elements that share a common name and in this case major revisions of the documents' contents as well as a new DTD will be necessary to deal with this. It won't be easy.

XML namespaces are intended to help deal with this problem. They enable names used in markup to be qualified so that you can make duplicate names used in different markup unique by putting them in separate namespaces. An XML namespace is just a collection of element and attribute names that is identified by a URI. Each name in an XML namespace is qualified by the URI that identifies the namespace. Thus different XML namespaces may contain common names without causing confusion since each name is notionally qualified by the unique URI for the namespace that contains it.

I say 'notionally qualified' because you don't actually qualify names using the URI directly. You use another name called a namespace prefix whose value is the URI for the namespace. For example, I could have a namespace that is identified by the URI http://www.wrox.com/Toys and a namespace prefix, toys, that contains a declaration for the name rubber_duck. I could have a second namespace with the URI http://www.wrox.com/BathAccessories and the namespace prefix BathAccessories that also defines the name rubber_duck.

The rubber_duck name from the first namespace is referred to as Toys:rubber_duck and that from the second namespace is BathAccessories:rubber_duck so there is no possibility of confusing them. The colon is used in the qualified name to separate the namespace prefix from the local name, which is why we said earlier you should avoid the use of colons in ordinary XML names.

Let's come back to the confusing aspects of namespaces for a moment. There is a temptation to imagine that the URI that identifies an XML namespace also identifies a document somewhere that specifies the names that are in the namespace. This is not required by the namespace specification. The URI is just a unique identifier for the namespace and a unique qualifier for a set of names. It does not necessarily have any other purpose, or even have to refer to a real document. It only needs to be unique. The definition of how names within a given namespace relate to one another and the rules for markup that uses them is an entirely separate question. This may be provided by a DTD or some other mechanism such as an XML Schema.

Namespace Declarations
A namespace is associated with a particular element in a document, which of course can be, but does not have to be, the root element. A typical namespace declaration in an XML document looks like this:

```xml
<sketcher:sketch xmlns:sketcher="http://www.wrox.com/dtds/sketches"/>
```
Working with XML Documents

Right at the beginning of this chapter we introduced the notion of an XML processor as a module that is used by an application to read XML documents. An XML processor parses the contents of a document and makes the elements together with their attributes and content available to the application, so it is also referred to as an XML parser. In case you haven't met the term before a parser is just a program module that breaks text in a given language down into its component parts. A natural language processor would have a parser that identifies the grammatical segments in each sentence. A compiler has a parser that identifies variables, constants, operators, etc. in a program statement. An application accesses the content of a document through an API provided by an XML parser and the parser does the job of figuring out what the document consists of.

Java supports two complementary APIs for processing an XML document:

- **SAX**, which is the Simple API for XML parsing.
- **DOM**, which is the Document Object Model for XML.

The support in SDK 1.4 is for DOM level 2 version 1 and for SAX version 2. SDK 1.4 also supports XSLT version 1.0 where XSL is the eXtensible Stylesheet Language and T is Transformations a language for transforming one XML document into another, or into some other textual representation such as HTML. However, we will concentrate on the basic application of DOM and SAX. XSLT is such an extensive topic that there are several books devoted entirely to it.

**Note**


Before we get into detail on these APIs, let's look at the broad differences between SAX and DOM, and get an idea of the circumstances in which you might choose to use one rather than the other.

**SAX Processing**

SAX uses an event-based process for reading an XML document that is implemented through a callback mechanism. This is very similar to the way in which we handle GUI events in Java. As the parser reads a document, each parsing event, such as recognizing the start or end of an element, results in a call to a particular method that is associated with that event. Such a method is often referred to as a handler. It is up to you to implement these methods to respond appropriately to the event. Each of your methods then has the opportunity to react to the event...
that will result in it being called in any way that you wish. Below you can see the events that would arise from the
XML document example that we saw earlier.

<table>
<thead>
<tr>
<th>XML</th>
<th>Events in your Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xml version=&quot;1.0&quot;&gt;</code></td>
<td>Start document</td>
</tr>
<tr>
<td><code>&lt;circle</code></td>
<td>Start element</td>
</tr>
<tr>
<td>15</td>
<td>Start element</td>
</tr>
<tr>
<td>&lt;/circle&gt;</td>
<td>End element</td>
</tr>
<tr>
<td><code>&lt;position</code></td>
<td>Start element</td>
</tr>
<tr>
<td><code>30</code></td>
<td>Start element</td>
</tr>
<tr>
<td><code>&lt;/x-coordinate&gt;</code></td>
<td>End element</td>
</tr>
<tr>
<td><code>&lt;y-coordinate&gt;</code></td>
<td>End element</td>
</tr>
<tr>
<td><code>&lt;/position&gt;</code></td>
<td>End element</td>
</tr>
<tr>
<td><code>&lt;/circle&gt;</code></td>
<td>End element</td>
</tr>
</tbody>
</table>

SAX Processing of XML

Each type of event results in a different method in your program being called. There are, for instance, different events
for registering the beginning and end of a document. You can also see that the start and end of each element result in
two further kinds of events, and another type of event occurs for each segment of document data. Thus this particular
document will involve five different methods in your program being called, some of them more than once, of course,
so there is one method for each type of event.

Because of the way SAX works, your application inevitably receives the document a piece at a time, with no
representation of the whole document. This means that if you need to have the whole document available to your
program with its elements and content properly structured you have to assemble it yourself from the information
supplied to your callback methods.

Of course, it also means that you don't have to keep the entire document in memory if you don't need it, so if you are
just looking for particular information from a document, all `<phonenumber>` elements for instance, you can just save
those as you receive them through the callback mechanism, and discard the rest. As a consequence, SAX is a
particularly fast and memory efficient way of selectively processing the contents of an XML document.

First of all, SAX itself is not an XML document parser; it is a public domain definition of an interface to an XML
parser, where the parser is an external program. The public domain part of the SAX API is in three packages that
are shipped as part of the SDK:

- org.xml.sax this defines the Java interfaces specifying the SAX API and the InputSource class that
  encapsulates a source of an XML document to be parsed.

- org.xml.sax.helpers this defines a number of helper classes for interfacing to a SAX parser.

- org.xml.sax.ext this defines interfaces representing optional extensions to SAX2 to obtain information about
  a DTD, or to obtain information about comments and CDATA sections in a document.

In addition to these, the javax.xml.parsers package provides factory classes that you use to gain access to a parser
and the javax.xml.transform package defines interfaces and classes for XSLT 1.0 processing of an XML document.

In Java terms there are several interfaces involved. The XMLReader interface that is defined in the org.xml.sax
package specifies the methods that the SAX parser will call as it recognizes elements, attributes, and other components of an XML document. You must provide a class that implements these methods and responds to the method calls in the way that you want.

**DOM Processing**

DOM works quite differently to SAX. When an XML document is parsed, the whole document tree is assembled in memory and returned to your application as an object of type Document that encapsulates it, as illustrated below.

Once you have the Document object available you can call the object's methods to navigate through the elements in the document tree starting with the root element. With DOM, the entire document is available to you for you to process as often and in as many ways as you want. This is a major advantage over SAX processing. The downside to this is the amount of memory occupied by the document there is no choice, you get it all no matter how big it is. With some documents the amount of memory required may be prohibitively large.

DOM has one other unique advantage over SAX. It allows you to modify existing documents or create new ones. If you want to create an XML document programmatically and then transfer it to an external destination such as a file or another computer, DOM is the API for this since SAX has no direct provision for creating or modifying XML documents. We will go into detail on how we can use a DOM parser in the next chapter.
Accessing Parsers

The javax.xml.parsers package defines four classes supporting the processing of XML documents:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAXParserFactory</td>
<td>Enables you to create a configurable factory object that you can use to</td>
</tr>
<tr>
<td></td>
<td>create a SAXParser object encapsulating a SAX-based parser.</td>
</tr>
<tr>
<td>SAXParser</td>
<td>Defines an object that wraps a SAX-based parser.</td>
</tr>
<tr>
<td>DocumentBuilderFactory</td>
<td>Enables you to create a configurable factory object that you can use to</td>
</tr>
<tr>
<td></td>
<td>create a DocumentBuilder object encapsulating a DOM-based parser.</td>
</tr>
<tr>
<td>DocumentBuilder</td>
<td>Defines an object that wraps a DOM-based parser.</td>
</tr>
</tbody>
</table>

All four classes are abstract. This is because JAXP is designed to allow different parsers and their factory classes to be plugged in. Both DOM and SAX parsers are developed independently of the Java SDK so it is important to be able to integrate new parsers as they come along. As we will see, the Crimson parser that is currently distributed with the SDK is controlled and developed by the Apache Project, but you may want to take advantage of the new features provided by the Xerces parser from the same organization.

These abstract classes act as wrappers for the specific factory and parser objects that you need to use for a particular parser and insulate your code from a particular parser implementation. An instance of a factory object that can create an instance of a parser is created at runtime, so your program can use a different parser without changing or even recompiling your code. Now that you have a rough idea of the general principles, let's get down to specifics and practicalities, starting with SAX.
Using SAX

To process an XML document with SAX, you first have to establish contact with the parser that you want to use. The first step towards this is to create a SAXParserFactory object like this:
Summary

In this chapter we have discussed the fundamental characteristics of XML and how Java supports the analysis and synthesis of XML documents. The key points we have covered include the following:

- XML is a language for describing data that is to be communicated from one computer to another. Data is described in the form of text that contains the data plus markup that defines the structure of the data.
- XML is also a meta-language because you can use XML to create new languages for defining and structuring data.
- Markup consists of XML elements that may also include attributes, where an attribute is a name/value pair.
- The structure and meaning of a particular type of document can be defined within a Document Type Definition (DTD).
- A DTD can be defined in an external file or it can be part of a document.
- A DTD is identified by a DOCTYPE declaration in a document.
- An XML namespace defines a set of names qualified by a prefix that corresponds to a URI.
- The SAX API defines a simple event-driven mechanism for analyzing XML documents.
- A SAX parser is a program that parses an XML document and identifies each element in a document by calling a particular method in your program. The methods that are called are those defined by the SAX API.
Exercises

1.

Write a program using SAX that will count the number of occurrences of each element type in an XML document and display them. The document file to be processed should be identified by the first command line argument.

2.

Modify the program resulting from the previous exercise so that it will accept optional additional command line arguments that are the names of elements. When there are two or more command line arguments, the program should only count and report on the elements identified by the second and subsequent command line arguments.
Chapter 22: Creating and Modifying XML Documents

Overview

In this chapter we will concentrate on understanding what we can do with the DOM API. As we outlined in the previous chapter, DOM uses a mechanism that is completely different to SAX. As well as providing an alternative mechanism for parsing XML documents, DOM also adds the capability for you to modify them and create new ones.

In this chapter you will learn:

- What a document object model is
- How you create a DOM parser
- How you access the contents of a document using DOM
- How you create and update a new XML document
- How to modify Sketcher to read and write sketches as XML documents
The Document Object Model (DOM)

As we saw in the previous chapter, a DOM parser presents you with an object encapsulating the entire XML structure. You can then call methods belonging to this object to navigate through the document tree and process the elements and attributes in the document in whatever way you want. This is quite different to SAX as we have already noted, but nonetheless there is quite a close relationship between DOM and SAX.

The mechanism for getting access to a DOM parser is very similar to what we used to obtain a SAX parser. You start with a factory object that you obtain like this:
Summary

In this chapter we have discussed how we can use a DOM parser to analyze XML and how JAXP supports the synthesis and modification of XML documents using DOM. The key points we have covered include the following:

- An object of type DocumentBuilder encapsulates a DOM parser.
- You create an object encapsulating a DOM parser by using a DocumentBuilderFactory object that you obtain by calling the static newInstance() method that is defined in the DocumentFactoryBuilder class.
- You can parse an XML document by passing the document as an argument to the parse() method for a DocumentBuilder object.
- A DOM parser creates a Document object that encapsulates an entire XML document as a tree of Node objects.
- The DOM API defines the methods that a Document object has that enable you to analyze an XML document by navigating through the nodes in the Document object.
- The DOM API also defines methods for creating a new XML document encapsulated by a Document object.
- When you want to create a new XML document that includes a DTD you should use the createDocument() method for a DOMImplementation object rather than the newDocument() method for a DocumentBuilder object.
Exercises

1.

Write a program using DOM that will count the number of occurrences of each element type in an XML document and display them. The document file should be identified by the first command line argument. The program should also accept optional additional command line arguments that are the names of elements. When there are two or more command line arguments, the program should only count and report on the elements identified by the second and subsequent command line arguments.

2.

Implement the XML Import capability in Sketcher using SAX rather than DOM.
Appendix A: Keywords

The following keywords are reserved in Java, so you must not use them as names in your programs:
| `abstract`       | `int`          |
| `assert`         | `interface`   |
| `boolean`        | `long`        |
| `break`          | `native`      |
| `byte`           | `new`         |
| `case`           | `package`     |
| `catch`          | `private`     |
| `char`           | `protected`   |
| `class`          | `public`      |
| `const`          | `return`      |
| `continue`       | `short`       |
| `default`        | `static`      |
| `do`             | `strictfp`    |
| `double`         | `super`       |
| `else`           | `switch`      |
| `extends`        | `synchronized`|
| `final`          | `this`        |
| `finally`        | `throw`       |
| `float`          | `throws`      |
| `for`            | `transient`   |
| `goto`           | `try`         |
| `if`             | `void`        |
| `implements`     | `volatile`    |
| `import`         | `while`       |
| `instanceof`     |               |

You should also not attempt to use the boolean values true and false, or null as names in your programs.
Appendix B: Computer Arithmetic

In the chapters of this book, we have deliberately kept discussion of arithmetic to a minimum. However, it is important overall and fundamental to understanding how some operators work, so I have included a summary of the subject in this appendix. If you feel confident about your math knowledge, this will all be old hat to you and you need read no further. If you find the math parts tough, then this section should show you how easy it really is.

Binary Numbers

First let's consider what we mean when we write a common everyday number such as 321 or 747. Put more precisely we mean:

321 is:

\[3 \times 10 \times 10 + 2 \times 10 + 1\]

and 747 is:

\[7 \times 10 \times 10 + 4 \times 10 + 7\]

Because it is built around powers of ten, we call this the decimal system (derived from the Latin *decimalis* meaning of tithes, which was a tax of 10% - ah, those were the days...).

Representing numbers in this way is very handy for people with ten fingers and ten toes, or creatures with ten of any kind of appendage for that matter. However, your PC is quite unhandy in this context, being built mainly of switches that are either on or off. This is OK for counting up to two, but not spectacular at counting to ten. For this reason your computer represents numbers to base 2 rather than base 10. This is called the binary system of counting, analogous to the bicycle (two wheels). With the decimal system, to base 10, the digits used can be from 0 to 9. In the binary system, to base 2, the digits can only be 0 or 1, ideal when you only have on/off switches to represent them. Each digit in the binary system is called a bit, being an abbreviation for binary digit. In an exact analogy to our usual base 10 counting, the binary number 1101 is therefore:

\[1 \times 2 \times 2 \times 2 + 1 \times 2 \times 2 + 0 \times 2 + 1\]
which amounts to 13 in the decimal system. In the following figure you can see the decimal equivalents of 8-bit binary numbers illustrated.

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>0</td>
<td>1000 0000</td>
<td>128</td>
</tr>
<tr>
<td>0000 0001</td>
<td>1</td>
<td>1000 0001</td>
<td>129</td>
</tr>
<tr>
<td>0000 0010</td>
<td>2</td>
<td>1000 0010</td>
<td>130</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0001 0000</td>
<td>16</td>
<td>1001 0000</td>
<td>144</td>
</tr>
<tr>
<td>0001 0001</td>
<td>17</td>
<td>1001 0001</td>
<td>145</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0111 1100</td>
<td>124</td>
<td>1111 1100</td>
<td>252</td>
</tr>
<tr>
<td>0111 1101</td>
<td>125</td>
<td>1111 1101</td>
<td>253</td>
</tr>
<tr>
<td>0111 1110</td>
<td>126</td>
<td>1111 1110</td>
<td>254</td>
</tr>
<tr>
<td>0111 1111</td>
<td>127</td>
<td>1111 1111</td>
<td>255</td>
</tr>
</tbody>
</table>

Note that using just 7 bits we can represent all the decimal numbers from 0 to 127, which is a total of 27, or 128 numbers, and using all 8 bits we get 256, or 28 numbers. In general, if we have n bits we can represent 2^n positive integers with values from 0 to 2^n-1.
Hexadecimal Numbers

When we get to larger binary numbers, for example:

```
1111 0101 1011 1001 1110 0001
```

the notation starts to be a little cumbersome, particularly when you consider that if you apply the same method to work out what this in decimal, it's only 16,103,905, a miserable 8 decimal digits. You can sit more angels on a pinhead than that. Well, as it happens, we have an excellent alternative.

Arithmetic to base 16 is a very convenient option. Each digit can have values from 0 to 15 (the digits from 10 to 15 being represented by the letters A to F as shown in the next figure) and values from 0 to 15 correspond quite nicely with the range of values that four binary digits can represent.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
<td>1010</td>
</tr>
<tr>
<td>B</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>1100</td>
</tr>
</tbody>
</table>
Since a hexadecimal digit corresponds exactly to 4 binary bits, we can represent the binary number above as a hexadecimal number just by taking successive groups of four binary digits starting from the right, and writing the equivalent base 16 digit for each group. The binary number:

```
1111 0101 1011 1001 1110 0001
```

will therefore come out as:

F5B9E1

We have six hexadecimal digits corresponding to the six groups of four binary digits. Just to show it all works out with no cheating, we can convert this number directly from hexadecimal to decimal, by again using the analogy with the meaning of a decimal number, as follows:

F5B9E1 is:

```
15 x 16 x 16 x 16 x 16 + 5 x 16 x 16 x 16 x 16 + 11 x 16 x 16 x 16 + 9 x 16 x 16 + 14 x16 + 1
```

This in turn turns out to be:

```
15,728,640 + 327,680 + 45,056 + 2304 + 224 + 1
```

which fortunately totals to the same number we got when we converted the equivalent binary number to a decimal value.
Negative Binary Numbers

There is another aspect to binary arithmetic that you need to understand - negative numbers. So far we have assumed everything is positive - the optimist's view if you will - our glass is still half full. But we can't avoid the negative side of life forever - the pessimist's perspective that our glass is already half empty. How do we indicate a negative number? Well, we only have binary digits at our disposal and indeed they contain the solution.

For numbers that we want to have the possibility of negative values (referred to as **signed** numbers) we must first decide on a fixed length (in other words, the number of binary digits) and then designate the leftmost binary digit as a sign bit. We have to fix the length in order to avoid any confusion about which bit is the sign bit as opposed to bits that are digits. A single bit is quite capable of representing the sign of a number because a number can be either positive - corresponding to a sign bit being 0, or negative - indicated by the sign bit being 1.

Of course, we can have some numbers with 8 bits, and some with 16 bits, or whatever, as long as we know what the length is in each case. If the sign bit is 0 the number is positive, and if it is 1 it is negative. This would seem to solve our problem, but not quite. If we add -8 in binary to +12 we would really like to get the answer +4. If we do that simplistically, just putting the sign bit of the positive value to 1 to make it negative, and then doing the arithmetic with conventional carries, it doesn't quite work:

<table>
<thead>
<tr>
<th>12 in binary is</th>
<th>0000 1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8 in binary we suppose is</td>
<td>1000 1000</td>
</tr>
</tbody>
</table>

since +8 is 0000 1000. If we now add these together we get:

1001 0100

This seems to be -20, which is not what we wanted at all. It's definitely not +4, which we know is 0000 0100. Ah, I hear you say, you can't treat a sign just like another digit. But that is just what we do have to do when dealing with computers because, dumb things that they are, they have trouble coping with anything else. So we really need a different representation for negative numbers. Well, we could try subtracting +12 from +4 since the result should be -8:

<table>
<thead>
<tr>
<th>+4 is</th>
<th>0000 0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take away +12</td>
<td>0000 1100</td>
</tr>
</tbody>
</table>
and we get

| 1111 1000 |

For each digit from the fourth from the right onwards we had to borrow 1 to do the sum, analogously to our usual decimal method for subtraction. This supposedly is -8, and even though it doesn't look like it, it is. Just try adding it to +12 or +15 in binary and you will see that it works. So what is it? It turns out that the answer is what is called the two's complement representation of negative binary numbers.

Now here we are going to demand a little faith on your part and avoid getting into explanations of why it works. We will just show you how the 2's complement form of a negative number can be constructed from a positive value, and that it does work so you can prove it to yourself. Let's return to our previous example where we need the 2's complement representation of -8. We start with +8 in binary:

0000 1000

We now flip each digit - if it is one make it zero, and vice versa:

1111 0111

This is called the 1's complement form, and if we now add 1 to this we will get the 2's complement form:

| 1111 0111 |
| Add one to this | 0000 0001 |
| and we get: | 1111 1000 |

Now this looks pretty similar to our representation of -8 we got from subtracting +12 from +4. So just to be sure, let's try the original sum of adding -8 to +12:

| +12 is | 0000 1100 |
| Our version of -8 is | 1111 1000 |
| and we get: | 0000 0100 |

So the answer is 4 - magic! It works! The carry propagates through all the leftmost 1's, setting them back to zero. One fell off the end, but we shouldn't worry about that. It's probably the one we borrowed from off the end in the subtraction sum we did to get -8. In fact what is happening is that we are making the assumption that the sign bit, 1 or
0, repeats forever to the left. Try a few examples of your own, you will find it always works quite automatically. The really great thing is, it makes arithmetic very easy (and fast) for your computer.
Floating Point Numbers

We often have to deal with very large numbers: the number of protons in the universe, for example, which needs around 79 decimal digits. Clearly there are lots of situations where we need more than the 10 decimal digits we get from a 4 byte binary number. Equally, there are lots of very small numbers. The amount of time in minutes it takes the typical car salesman to accept your offer on his 1982 Ford LTD (and only covered 380,000 miles...). A mechanism for handling both these kinds of numbers is - as you will have guessed from the title of this section - floating-point numbers.

A floating-point representation of a number is a decimal point followed by a fixed number of digits, multiplied by a power of 10 to get the number you want. It's easier to demonstrate than explain, so let's take some examples. The number 365 in normal decimal notation would be written in floating point form as:

\[0.365 \times 10^3\]

where the \(E\) stands for "exponent" and is the power of ten that the 0.365 (the mantissa) is multiplied by, to get the required value. That is:

\[0.365 \times 10 \times 10 \times 10\]

which is clearly 365.

Now let's look at a smallish number:

\[0.365 \times 10^{-4}\]

This is evaluated as \(.365 \times 10^{-4}\), which is \(0.000365\) - exactly the time in minutes required by the car salesman to accept your cash.

The number of digits in the mantissa of a floating-point number depends on the type of the floating-point number that you are using. The Java type \texttt{float} provides the equivalent of approximately 7 decimal digits, and the type \texttt{double} provides around 17 decimal digits. The number of digits is approximate because the mantissa is binary, not decimal, and there's not an exact mapping between binary and decimal digits.

Suppose we have a large number such as 2,134,311,179. How does this look as a floating-point number? Well, as type \texttt{float} it looks like:

\[0.2134311 \times 10^{10}\]

It's not quite the same. We have lost three low order digits so we have approximated our original value as 2,134,311,000. This is a small price to pay for being able to handle such a vast range of numbers, typically from 10-38 to 10+38 either positive or negative, as well having an extended representation that goes from a minute 10-308 to a mighty 10+308. As you can see, they are called floating-point numbers for the fairly obvious reason that the decimal point "floats" depending on the exponent value.

Aside from the fixed precision limitation in terms of accuracy, there is another aspect you may need to be conscious of. You need to take great care when adding or subtracting numbers of significantly different magnitudes. A simple example will demonstrate the kind of problem that can arise. We can first consider adding \(.365\times10^{-3}\) to \(.365\times10^{7}\). We
can write this as a decimal sum:

\[.000365 + 3,650,000\]

This produces the result:

\[3,650,000.000365\]

Which when converted back to floating point becomes:

\[.3650000E+7\]

So we might as well not have bothered. The problem lies directly with the fact that we only carry 7 digits precision. The 7 digits of the larger number are not affected by any of the digits of the smaller number because they are all further to the left. Funnily enough, you must also take care when the numbers are very nearly equal. If you compute the difference between such numbers you may end up with a result that only has one or two digits precision. It is quite easy in such circumstances to end up computing with numbers that are total garbage.
Index of Data types, Keywords and Operators

This is not a comprehensive list of Data types, Keywords and Operators. For the Guide to the Index' please refer to Main Index.

Data types

byte, 40, 41, 43  
char, 62  
double, 44  
float, 44  
int, 40, 41  
long, 40, 41, 42  
short, 40, 41, 43
Index

Keywords

abstract, 252
assert, 114
catch, 301
class, 21, 176, 278
extends, 236, 241, 282
final, 176, 180, 276, 277
for, 99
implements, 257
import, 209
interface, 278
native, 231
new, 188
package, 206
private, 24, 213, 244
protected, 213, 245
public, 31, 206, 209, 213, 244, 280
return, 178
static, 31, 176, 180, 184, 226
super, 239, 242, 244
synchronized, 613, 628
this, 182
throw, 315
throws, 300
try, 301
void, 31, 177
Index

Operators

+ (addition), 138
+= (add=), 138
add= (+=), 57
AND (&), 64, 86
assignment (=), 67
complement (~), 64
conditional AND (&&), 86
conditional OR (||), 88
decrement (--), 51, 55
divide= (/=), 57
increment (+=), 51, 55
modulus (%), 51, 55
modulus= (%=), 57
multiply= (*=), 57
NOT (!), 89
OR (|), 64, 88
shift left (<<), 68
shift right (>>, 68
shift right fill (>>>), 68
subtract= (-=), 57
XOR (^), 64
Index of Classes, Exceptions, Interfaces and Methods

This is not a comprehensive list of Classes, Exceptions, Interfaces and Methods. For the Guide to the Index' please refer to Main Index.

Classes

AbstractAction, 766, 767
AbstractButton, 718
ActionEvent, 745, 876
AdjustmentEvent, 745
AffineTransform, 902
Arc2D, 803
Array, 551
ArrayList, 506
AWTEvent, 732
Book, 953, 985
Boolean, 212
BorderFactory, 690
BorderLayout, 682, 683
Box, 688
BoxLayout, 688
Buffer, 437, 447
Byte, 212
ByteBuffer, 387, 440, 1082
Calendar, 573
CardLayout, 684
Character, 90, 212
CharBuffer, 387, 437
Circle, 1073, 1090
Class, 254, 485
Collections, 638
Color, 662
Component, 646
ComponentEvent, 733
Container, 646
ContainerEvent, 732
CubicCurve2D, 806
Cursor, 664
Curve, 1074, 1091
Date, 367, 565
This document is created with the unregistered version of CHM2PDF Pilot
Rectangle, 657, 659, 798, 840, 1072, 1090
Rectangle2D, 798
RoundRectangle2D, 798
SAXParser, 1025, 1031
SAXParserFactory, 1025, 1026, 1027, 1030
Short, 212
ShortBuffer, 387
SpinnerDateModel, 882
SpinnerListModel, 882
SpinnerModel, 882
SpinnerNumberModel, 882
Stack, 498, 506, 525
StreamTokenizer, 338
String, 134, 1082
StringBuffer, 160
StringTokenizer, 157
System, 31, 320
SystemColor, 664
Text, 865, 866, 937, 1075, 1091
Thread, 602
Throwable, 296, 316
Toolkit, 655
TreeMap, 506
TreeSet, 505
TrySAXHandler, 1038, 1041
Vector, 498, 499, 506, 508, 637
WeakHashMap, 506
Window, 646
WindowAdapter, 743
WindowEvent, 731
Writer, 335
XMLParser, 1031
Index

Exceptions

ArithmeticException, 298, 301, 305, 309, 312, 317, 321, 322
ArrayIndexOutOfBoundsException, 299, 309, 311, 313, 318, 512, 513
ArrayStoreException, 298, 515
AsynchronousCloseException, 403, 405, 434, 451
BufferUnderflowException, 440, 442
ClassCastException, 298, 485, 515
ClassNotFoundException, 482
CloneNotSupportedException, 259
ClosedByInterruptException, 403, 405, 434, 451
ClosedChannelException, 403, 405, 434, 451
DOMException, 1050, 1066
EmptyStackException, 525
EOFException, 484
FileNotFoundException, 374, 375, 409
IllegalArgument Exception, 298, 391, 405, 451, 456, 461, 527, 538, 637, 1050
IllegalMonitorStateException, 298, 632
IllegalStateException, 298, 503
IllegalThreadStateException, 299, 606
IndexOutOfBoundsException, 124, 298, 305, 309, 397, 425, 514, 531, 956
InterruptedException, 607, 608, 609
InvalidClassException, 476, 482
InvalidMarkException, 398
InvalidUserInputException, 345
IOException, 300, 311, 329, 403, 405, 434, 451, 461, 482, 943, 1050
LinkageError, 297
NegativeArraySizeException, 298
NonReadableChannelException, 434, 451
NonWritableChannelException, 403, 405, 451
NotSerializableException, 476, 931
NullPointerException, 298, 515, 538, 839
NumberFormatException, 212, 299
OptionalDataException, 482
ParseException, 569
ParserConfigurationException, 1027, 1041
ParserConfigurationException, 1027, 1041
PatternSyntaxException, 576
ArchivoException, 956, 958
ReadOnlyBufferException, 398, 399
Runtime Exception, 298, 309
SAXException, 1048, 1050
SAXNotRecognizedException, 1030
SAXNotSupportedException, 1030
SecurityException, 298, 363, 373
StreamCorruptedException, 482
StringIndexOutOfBoundsException, 149, 163, 167, 296, 299
ThreadDeath, 297
UnsupportedOperationException, 298, 397, 503, 538
VirtualMachineError, 297
Index

Interfaces

Action, 764, 765, 766, 771, 781
ActionListener, 291, 685, 730, 746, 752, 753, 755, 757, 761, 762, 764, 858, 860, 876, 887, 888, 897-99, 912, 917, 921
Attr, 1060
ByteChannel, 385
ChangeListener, 883
Cloneable, 257, 282
Collection, 507, 538
Comparable, 522, 539
Constants, 835
ContentHandler, 1032, 1033
ConversionFactors, 278, 282
Conversions, 280, 281, 282
Document, 1050
DocumentFragment, 1052
DocumentType, 1052, 1061
DOMImplementation, 1065
DTDHandler, 1033
Element, 1052, 1068
EntityResolver, 1032, 1033
ErrorHandler, 1033, 1041, 1056
EventListener, 737
Externalizable, 493
FileFilter, 367
FilenameFilter, 367, 368, 370
FocusListener, 740
GatheringByteChannel, 385
Icon, 775
ItemListener, 746, 885
Iterator, 498, 502, 514
KeyListener, 740
List, 507
ListIterator, 503, 514
ListSelectionListener, 879, 880
Map, 507, 537
Map.Entry, 538
MouseListener, 739
MouseMotionListener, 739, 893
MouseWheelListener, 739
NamedNodeMap, 1059
Node, 1051
NodeList, 1053
Observer, 556, 560, 786, 836, 899
Pageable, 954, 976, 979, 981, 984
PathIterator, 934, 936
Printable, 954, 958-60, 962, 964, 967, 969, 970, 972, 975, 983, 988
PrinterGraphics, 954
PrintRequestAttribute, 974
ReadableByteChannel, 385
Runnable, 603, 610, 619
ScatteringByteChannel, 385
Serializable, 475, 477, 486, 732, 931
Set, 507, 538
Shape, 795, 800
SortedMap, 507
SortedSet, 507
WindowFocusListener, 738
WindowListener, 737, 742, 951
WindowStateListener 738
WritableByteChannel, 385
Index

Methods

abs(), 59, 564
AbstractAction(), 767
accept(), 367, 368, 370, 944, 946
acos(), 59
actionPerformed(), 685, 730, 746, 752, 55, 757, 762, 768, 769, 853, 858, 860, 876, 887, 888, 897, 99, 912, 917, 921, 937, 943, 944, 946, 950, 957, 958, 961, 962, 970, 973, 983, 988, 1079, 1085
add(), 504, 510, 512, 519, 529, 531, 532, 573, 638, 660, 675, 682, 685, 713, 714, 716, 764, 769, 771, 772, 76, 778, 809, 828, 836, 888, 897, 900, 963
addActionListener(), 291, 685, 730, 858, 860
addAll(), 531
addChoosableFileFilter(), 946
addFirst(), 531
addItem(), 274, 276
addLast(), 531
addListSelectionListener(), 879
addMotionListener(), 832
addMouseListener(), 832
addObserver(), 557
addPoint(), 270, 276, 532
addPropertyChangeListener(), 766
addSeparator(), 714, 717
addWindowListener(), 742, 744
adjustmentValueChanged(), 746
after(), 565, 574, 576
allocate(), 392, 413
allocateDirect(), 414, 450
append(), 164, 165, 985, 986
appendChild(), 1068
appendReplacement(), 588, 590, 593, 596
appendTail(), 590
array(), 397
asCharBuffer(), 393
asDoubleBuffer(), 393
asFloatBuffer(), 393
asin(), 59, 918
asIntBuffer(), 393
asLongBuffer(), 393, 440
asReadOnlyBuffer(), 393
asShortBuffer(), 393
atan(), 59
atan2(), 59
This document is created with the unregistered version of CHM2PDF Pilot
This document is created with the unregistered version of CHM2PDF Pilot
getAllFonts(), 666, 669
getAscent(), 669
getAttributes(), 1059
getAvailableFontFamilyNames(), 878
getAvailableLocales(), 569
getBackground(), 662
getBlue(), 663
getBounds(), 655, 826, 828, 838, 895, 909, 912, 920, 963
getBytes(), 160, 411, 1082
getBoundingBox(), 801
getCenterPoint(), 656
getChannel(), 386, 409
getChar(), 437
getChars(), 159, 167
getChildren(), 1052, 1057
getClass(), 253, 254, 485, 1056
getClickCount(), 872
getColor(), 826
columnNumber(), 1042
getComponent(), 674
getComponents(), 674
getConstraint(), 703
getContentPane(), 653
getCurrentPoint(), 816, 820
getDateTimeInstance(), 567
getDebugElement(), 1088
getDOMImplementation(), 1065
getDouble(), 445
getElementsByTagName(), 1089, 1091
tagsByTagNameNS(), 1089
tagsByTagNameNS(), 1062
gotD(), 378
getFeature(), 1030
getFileComponent(), 461
getFileSelectionMode(), 940
getFirst(), 274, 276, 501, 531
gotFirstChild(), 1059, 1088
gotFirstChild(), 879
getFont(), 662, 669
getFontMetrics(), 669
getFontMetrics(), 662
getGlassPane(), 653
getGraphics(), 792
getGreen(), 663
getHeight(), 669, 775, 799, 978, 989
getHgap(), 684
getID(), 735
getImageableHeight(), 960, 967, 978
getImageableWidth(), 960, 966, 967, 978
This document is created with the unregistered version of CHM2PDF Pilot
isDone(), 934
isEmpty(), 517, 661, 800
isEnabled(), 654, 765
isFile(), 362, 367
isFloatable(), 775
isHidden(), 362
isInfinite(), 212
isInterface(), 256
isInterrupted(), 608
isLetter(), 666
isLetterOrDigit(), 90
isLoaded(), 462
isLowerCase(), 90
isModal(), 856
isNamespaceAware(), 1026, 1050
isNaN(), 212
isOpen(), 384
isPlain(), 666
isPopupTrigger(), 889, 891, 896
isUpperCase(), 90
isValid(), 466, 654
isValidating(), 1026
isVisible(), 654
isWhitespace(), 90, 151
item(), 1053, 1060, 1061
itemStateChanged(), 746, 885
iterator(), 503, 513, 531, 532, 538
join(), 608
keyPressed(), 740
keyReleased(), 740
keySet(), 537, 546
keyTyped(), 740
last(), 685
lastElement(), 513
lastIndexOf(), 151, 53, 155
limit(), 391
lineTo(), 816, 845, 935, 1091
matcher(), 578
matches(), 581
mark(), 398
matcher(), 578
matches(), 581
lookupPrintServices(), 956
lowerCaseMode(), 341
main(), 30, 49, 82, 83, 96, 99, 109, 137, 172, 174, 452
map(), 462, 464
mark(), 462, 464
println(), 31, 49, 50, 61, 135, 302, 346, 839
printlnStacktrace(), 316, 318
processEvent(), 736, 737
processFocusEvent(), 736, 737
ProcessingInstruction(), 1034
processKeyEvent(), 737
processMouseEvent(), 736
processMouseMotionEvent(), 737
processMouseWheelEvent(), 737
processWindowEvent(), 735, 736
ptLineDist(), 917
push(), 525
pushback(), 341
put(), 397, 399, 423, 536, 542
putAll(), 536
putChar(), 400
putConstraint(), 706
putDouble(), 400
putFloat(), 400
putInt(), 400, 413
putLong(), 400
putShort(), 400
putValue(), 765
quadTo(), 816, 935
quoteChar(), 341
random(), 60, 82, 131, 138, 185
read(), 310, 311, 385, 433, 456, 606
readObject(), 481, 482, 484, 488, 489, 932, 950
Rectangle(), 659
relativeCCW(), 918
release(), 466
remaining(), 392, 403, 437, 450
remove(), 503, 504, 515, 531, 536, 639, 828, 898, 900
removeAll(), 516
removeChoosableFileFilter(), 946
removeElementAt(), 516
removeFirst(), 531
removeLast(), 531
removePropertyChangeListener(), 766
renameTo(), 370
repaint(), 792, 814, 837, 879, 916, 950
replace(), 158
replenish(), 450
rescanCurrentDirectory(), 940
reset(), 398, 492, 579, 817
resetSyntax(), 341
resolveEntity(), 1033
reverse(), 167
rewind(), 403
rint(), 60
roll(), 573
rotate(), 904, 918, 920
round(), 60, 61
run(), 602, 603, 605 11, 614, 619, 634 36, 638
scale(), 904, 967
search(), 526
seek(), 461
set(), 505, 512, 531, 571
setAccelerator(), 718, 719
setAcceptAllFileFilter(), 946
setApproveButtonMnemonic(), 939
desApproveButtonText(), 939
setApproveButtonText(), 939
setAcceleratorText(), 939
setAttribute(), 1069
setAttributeNode(), 1069
setAttributeNodeNS(), 1069
setAttributeNS(), 1069
setBackground(), 662
setBorder(), 681, 750
setBottomComponent(), 881
setBounds(), 648, 649, 655, 656
setBounds(), 558, 560, 828
setCharAt(), 167
setClip(), 794, 971
setCoalescing(), 1049
setComposite(), 794
setAttributeNode(), 881
setComponent(), 940
setCurrentDirectory(), 940
setCursor(), 662, 665
detDaemon(), 605, 606, 611
setDefaultCloseOperation(), 648, 649, 733
setDialogTitle(), 939
SetDocumentLocator(), 1035
setFeature(), 1030, 1041
setFileFilter(), 946
setFileSelectionMode(), 940
setFloatable(), 774
setFont(), 662, 794, 867, 879
setForeground(), 662
setHgap(), 680, 684
setHorizontalComponent(), 851
setIcon(), 778, 779
setIgnoringComments(), 1049
setIgnoringElementContentWhitespace(), 1049
setImageableArea(), 978
setJMenuBar(), 713
setLayout(), 677, 678, 851
setLeftComponent(), 881
setLength(), 163
setLocation(), 655, 658, 659, 797, 803, 812
setMnemonic(), 718, 719, 860
setModal(), 856
setMultiSelectionEnabled(), 940
setName(), 611, 654
setNamespaceAware(), 1026, 1030, 1038, 1049
setOrientation(), 881, 968, 978
setPageable(), 976, 983, 985
writeInt(), 476
writeLong(), 476
writeObject(), 475, 77, 480, 481, 488, 489, 932
writeShort(), 476
yield(), 609