Table of Contents

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Chapter 5. The Java Platform

Chapters Chapter 2, Chapter 3, and Chapter 4 documented the Java programming language. This chapter switches gears and covers the Java platform—a vast collection of predefined classes available to every Java program, regardless of the underlying host system on which it is running. The classes of the Java platform are collected into related groups, known as *packages*. This chapter begins with an overview of the packages of the Java platform that are documented in this book. It then moves on to demonstrate, in the form of short examples, the most useful classes in these packages. Most of the examples are code snippets only, not full programs you can compile and run. For fully fleshed-out, real-world examples, see *Java Examples in a Nutshell* (O'Reilly). That book expands greatly on this chapter and is intended as a companion to this book.

5.1. Java Platform Overview

Table 5-1 summarizes the key packages of the Java platform that are covered in this book.

Table 5-1. Key packages of the Java platform

Package	Description
`java.io`	Classes and interfaces for input and output. Although some of the classes in this package are for working directly with files, most are for working with streams of bytes or characters.
`java.lang`	The core classes of the language, such as `String`, `Math`, `System`, `Thread`, and `Exception`.
`java.lang.annotation`	Annotation types and other supporting types for the Java 5.0 annotation feature. (See Chapter 4.)
`java.lang.instrument`	Support classes for Java virtual machine instrumentation agents, which are allowed to modify the byte code of the program the JVM is running. New in Java 5.0.
`java.lang.management`	A framework for monitoring and managing a running Java virtual machine. New in Java 5.0.
`java.lang.ref`	Classes that define weak references to objects. A weak reference is one that does not prevent the referent object from being garbage-collected.
`java.lang.reflect`	Classes and interfaces that allow Java programs to reflect on themselves by examining the constructors, methods, and fields of classes.
`java.math`	A small package that contains classes for arbitrary-precision integer and floating-point arithmetic.
`java.net`	Classes and interfaces for networking with other systems.
`java.nio`	Buffer classes for the New I/O API. Added in Java 1.4.
`java.nio.channels`	Channel and selector interfaces and classes for high-performance, nonblocking I/O.
`java.nio.charset`	Character set encoders and decoders for converting Unicode strings to and from bytes.
`java.security`	Classes and interfaces for access control and authentication. This package and its subpackages support cryptographic message digests and digital signatures.
`java.text`	Classes and interfaces for working with text in internationalized applications.
`java.util`	Various utility classes, including the powerful collections framework for working with collections of objects.
`java.util.concurrent`	Thread pools and other utility classes for concurrent programming. Subpackages support atomic variables and locks. New in Java 5.0.
`java.util.jar`	Classes for reading and writing JAR files.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Package	Description
`java.util.logging`	A flexible logging facility. Added in Java 1.4.
`java.util.prefs`	An API to read and write user and system preferences. Added in Java 1.4.
`java.util.regex`	Text pattern matching using regular expressions. Added in Java 1.4.
`java.util.zip`	Classes for reading and writing ZIP files.
`javax.crypto`	Classes and interfaces for encryption and decryption of data.
`javax.net`	Defines factory classes for creating sockets and server sockets. Enables the creation of socket types other than the default.
`javax.net.ssl`	Classes for encrypted network communication using the Secure Sockets Layer (SSL).
`javax.security.auth`	The top-level package for the JAAS API for authentication and authorization. Various subpackages hold most of the actual classes. Added in Java 1.4.
`javax.xml.parsers`	A high-level API for parsing XML documents using pluggable DOM and SAX parsers.
`javax.xml.transform`	A high-level API for transforming XML documents using a pluggable XSLT transformation engine and for converting XML documents between streams, DOM trees, and SAX events. Subpackages provide support for DOM, SAX and stream transformations. Added in Java 1.4.

Table 5-1 does not list all the packages in the Java platform, only the most important of those documented in this book. Java also defines numerous packages for graphics and graphical user interface programming and for distributed, or enterprise, computing. The graphics and GUI packages are `java.awt` and `javax.swing` and their many subpackages. These packages are documented in *Java Foundation Classes in a Nutshell* and *Java Swing*, both from O'Reilly. The enterprise packages of Java include `java.rmi`, `java.sql`, `javax.jndi`, `org.omg.CORBA`, `org.omg.CosNaming`, and all of their subpackages. These packages, as well as several standard extensions to the Java platform, are documented in *Java Enterprise in a Nutshell* (O'Reilly).

5.2. Text

Most programs manipulate text in one form or another, and the Java platform defines a number of important classes and interfaces for representing, formatting, and scanning text. The sections that follow provide an overview.

5.2.1. The String Class

Strings of text are a fundamental and commonly used data type. In Java, however, strings are not a primitive type, like `char`, `int`, and `float`. Instead, strings are represented by the `java.lang.String` class, which defines many useful methods for manipulating strings. `String` objects are *immutable*: once a `String` object has been created, there is no way to modify the string of text it represents. Thus, each method that operates on a string typically returns a new `String` object that holds the modified string.

This code shows some of the basic operations you can perform on strings:

```
// Creating strings
String s = "Now";               // String objects have a special literal syntax
String t = s + " is the time."; // Concatenate strings with + operator
String t1 = s + " " + 23.4;     // + converts other values to strings
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
t1 = String.valueOf('c');       // Get string corresponding to char value
t1 = String.valueOf(42);        // Get string version of integer or any value
t1 = object.toString();         // Convert objects to strings with toString()

// String length
int len = t.length();           // Number of characters in the string: 16

// Substrings of a string
String sub = t.substring(4);    // Returns char 4 to end: "is the time."
sub = t.substring(4, 6);        // Returns chars 4 and 5: "is"
sub = t.substring(0, 3);        // Returns chars 0 through 2: "Now"
sub = t.substring(x, y);        // Returns chars between pos x and y-1
int numchars = sub.length();    // Length of substring is always (y-x)

// Extracting characters from a string
char c = t.charAt(2);           // Get the 3rd character of t: w
char[] ca = t.toCharArray();    // Convert string to an array of characters
t.getChars(0, 3, ca, 1);        // Put 1st 3 chars of t into ca[1]-ca[3]

// Case conversion
String caps = t.toUpperCase();  // Convert to uppercase
String lower = t.toLowerCase(); // Convert to lowercase

// Comparing strings
boolean b1 = t.equals("hello");         // Returns false: strings not equal
boolean b2 = t.equalsIgnoreCase(caps);  // Case-insensitive compare: true
boolean b3 = t.startsWith("Now");       // Returns true
boolean b4 = t.endsWith("time.");       // Returns true
int r1 = s.compareTo("Pow");            // Returns < 0: s comes before "Pow"
int r2 = s.compareTo("Now");            // Returns 0: strings are equal
int r3 = s.compareTo("Mow");            // Returns > 0: s comes after "Mow"
r1 = s.compareToIgnoreCase("pow");      // Returns < 0 (Java 1.2 and later)

// Searching for characters and substrings
int pos = t.indexOf('i');         // Position of first 'i': 4
pos = t.indexOf('i', pos+1);      // Position of the next 'i': 12
pos = t.indexOf('i', pos+1);      // No more 'i's in string, returns -1
pos = t.lastIndexOf('i');         // Position of last 'i' in string: 12
pos = t.lastIndexOf('i', pos-1);  // Search backwards for 'i' from char 11

pos = t.indexOf("is");            // Search for substring: returns 4
pos = t.indexOf("is", pos+1);     // Only appears once: returns -1
pos = t.lastIndexOf("the ");      // Search backwards for a string
String noun = t.substring(pos+4); // Extract word following "the"

// Replace all instances of one character with another character
String exclaim = t.replace('.', '!');  // Works only with chars, not substrings

// Strip blank space off the beginning and end of a string
String noextraspaces = t.trim();

// Obtain unique instances of strings with intern()
String s1 = s.intern();        // Returns s1 equal to s
String s2 = "Now";             // String literals are automatically interned
boolean equals = (s1 == s2);   // Now can test for equality with ==
```

5.2.2. The Character Class

As you know, individual characters are represented in Java by the primitive `char` type. The Java platform also defines a `Character` class, which contains useful class methods for checking the type of a character and for converting the case of a character. For example:

```
char[] text;  // An array of characters, initialized somewhere else
int p = 0;    // Our current position in the array of characters
// Skip leading whitespace
while((p < text.length) && Character.isWhitespace(text[p])) p++;
// Capitalize the first word of text
while((p < text.length) && Character.isLetter(text[p])) {
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    text[p] = Character.toUpperCase(text[p]);
    p++;
  }
```

5.2.3. The StringBuffer Class

Since `String` objects are immutable, you cannot manipulate the characters of an instantiated `String`. If you need to do this, use a `java.lang.StringBuffer` or `java.lang.StringBuilder` instead. These two classes are identical except that `StringBuffer` has `synchronized` methods. `StringBuilder` was introduced in Java 5.0 and you should use it in preference to `StringBuffer` unless it might actually be manipulated by multiple threads. The following code demonstrates the `StringBuffer` API but could be easily changed to use `StringBuilder`:

```
// Create a string buffer from a string
StringBuffer b = new StringBuffer("Mow");

// Get and set individual characters of the StringBuffer
char c = b.charAt(0);        // Returns 'M': just like String.charAt()
b.setCharAt(0, 'N');         // b holds "Now": can't do that with a String!

// Append to a StringBuffer
b.append(' ');               // Append a character
b.append("is the time.");    // Append a string
b.append(23);                // Append an integer or any other value

// Insert Strings or other values into a StringBuffer
b.insert(6, "n't");          // b now holds: "Now isn't the time.23"

// Replace a range of characters with a string (Java 1.2 and later)
b.replace(4, 9, "is");       // Back to "Now is the time.23"

// Delete characters
b.delete(16, 18);            // Delete a range: "Now is the time"
b.deleteCharAt(2);           // Delete 2nd character: "No is the time"
b.setLength(5);              // Truncate by setting the length: "No is"

// Other useful operations
b.reverse();                 // Reverse characters: "si oN"
String s = b.toString();     // Convert back to an immutable string
s = b.substring(1,2);        // Or take a substring: "i"
b.setLength(0);              // Erase buffer; now it is ready for reuse
```

5.2.4. The CharSequence Interface

As of Java 1.4, both the `String` and the `StringBuffer` classes implement the `java.lang.CharSequence` interface, which is a standard interface for querying the length of and extracting characters and subsequences from a readable sequence of characters. This interface is also implemented by the `java.nio.CharBuffer` interface, which is part of the New I/O API that was introduced in Java 1.4. `CharSequence` provides a way to perform simple operations on strings of characters regardless of the underlying implementation of those strings. For example:

```
/**
 * Return a prefix of the specified CharSequence that starts at the first
 * character of the sequence and extends up to (and includes) the first
 * occurrence of the character c in the sequence. Returns null if c is
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
 * not found. s may be a String, StringBuffer, or java.nio.CharBuffer.
 */
public static CharSequence prefix(CharSequence s, char c) {
  int numChars = s.length();          // How long is the sequence?
  for(int i = 0; i < numChars; i++) { // Loop through characters in sequence
    if (s.charAt(i) == c)             // If we find c,
      return s.subSequence(0,i+1);    // then return the prefix subsequence
  }
  return null;                        // Otherwise, return null
}
```

5.2.5. The Appendable Interface

`Appendable` is a Java 5.0 interface that represents an object that can have a `char` or a `CharSequence` appended to it. Implementing classes include `StringBuffer`, `StringBuilder`, `java.nio.CharBuffer`, `java.io.PrintStream`, and `java.io.Writer` and all of its character output stream subclasses, including `PrintWriter`. Thus, the `Appendable` interface represents the common appendability of the text buffer classes and the text output stream classes. As we'll see below, a `Formatter` object can send its output to any `Appendable` object.

5.2.6. String Concatenation

The `+` operator concatenates two `String` objects or one `String` and one value of some other type, producing a new `String` object. Be aware that each time a string concatenation is performed and the result stored in a variable or passed to a method, a new `String` object has been created. In some circumstances, this can be inefficient and can result in poor performance. It is especially important to be careful when doing string concatenation within a loop. The following code is inefficient, for example:

```
// Inefficient: don't do this
public String join(List<String> words) {
    String sentence = "";
    // Each iteration creates a new String object and discards an old one.
    for(String word: words) sentence += word;
    return sentence;
}
```

When you find yourself writing code like this, switch to a `StringBuffer` or a `StringBuilder` and use the `append()` method:

```
// This is the right way to do it
public String join(List<String> words) {
    StringBuilder sentence = new StringBuilder();
    for(String word: words) sentence.append(word);
    return sentence.toString();
}
```

There is no need to be paranoid about string concatenation, however. Remember that string literals are concatenated by the compiler rather than the Java interpreter. Also, when a single expression contains multiple string concatenations, these are compiled efficiently

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

using a `StringBuilder` (or `StringBuffer` prior to Java 5.0) and result in the creation of only a single new `String` object.

5.2.7. String Comparison

Since strings are objects rather than primitive values, they cannot, in general, be compared for equality with the = = operator. == compares references and can determine if two expressions evaluate to a reference to the same string. It cannot determine if two distinct strings contain the same text. To do that, use the `equals( )` method. In Java 5.0 you can compare the content of a string to any other `CharSequence` with the `contentEquals( )` method.

Similarly, the < and > relational operators do not work with strings. To compare the order of strings, use the `compareTo()` method, which is defined by the `Comparable<String>` interface and is illustrated in the sample code above. To compare strings without taking the case of the letters into account, use `compareToIgnoreCase( )`.

Note that `StringBuffer` and `StringBuilder` do not implement `Comparable` and do not override the default versions of `equals( )` and `hashCode()` that they inherit from `Object`. This means that it is not possible to compare the text held in two `StringBuffer` or `StringBuilder` objects for equality or for order.

One important, but little understood method of the `String` class is `intern( )`. When passed a string `s`, it returns a string `t` that is guaranteed to have the same content as `s`. What's important, though, is that for any given string content, it always returns a reference to the same `String` object. That is, if `s` and `t` are two `String` objects such that `s.equals(t)`, then:

```
s.intern() == t.intern()
```

This means that the `intern( )` method provides a way of doing fast string comparisons using ==. Importantly, string literals are always implicitly interned by the Java VM, so if you plan to compare a string `s` against a number of string literals, you may want to intern `s` first and then do the comparison with = =.

The `compareTo()` and `equals( )` methods of the `String` class allow you to compare strings. `compareTo( )` bases its comparison on the character order defined by the Unicode encoding while `equals( )` defines string equality as strict character-by-character equality. These are not always the right methods to use, however. In some languages, the character ordering imposed by the Unicode standard does not match the dictionary ordering used when alphabetizing strings. In Spanish, for example, the letters

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

"ch" are considered a single letter that comes after "c" and before "d." When comparing human-readable strings in an internationalized application, you should use the `java.text.Collator` class instead:

```
import java.text.*;

// Compare two strings; results depend on where the program is run
// Return values of Collator.compare() have same meanings as String.compareTo()
Collator c = Collator.getInstance();      // Get Collator for current locale
int result = c.compare("chica", "coche"); // Use it to compare two strings
```

5.2.8. Supplementary Characters

Java 5.0 has adopted the Unicode 4.0 standard, which, for the first time, has defined codepoints that fall outside the 16-bit range of the `char` type. When working with these "supplementary characters" (which are primarily Han ideographs), you must use `int` values to represent the individual character. In `String` objects, or for any other type that represents text as a sequence of `char` values, these supplementary characters are represented as a series of two `char` values known as a *surrogate pair*.

Although readers of the English edition of this book are unlikely to ever encounter supplementary characters, you should be aware of them if you are working on programs that might be localized for use in China or another country that uses Han ideographs. To help you work with supplementary characters, the `Character`, `String`, `StringBuffer`, and `StringBuilder` classes have been extended with new methods that operate on `int` codepoints rather than `char` values. The following code illustrates some of these methods. You can find other, similar methods in the reference section and read about them in the online javadoc documentation.

```
int codepoint = 0x10001;  // This codepoint doesn't fit in a char
// Get the UTF-16 surrogate pair of chars for the codepoint
char[] surrogatePair = Character.toChars(codepoint);
// Convert the chars to a string.
String s = new String(surrogatePair);

// Print string length in characters and codepoints
System.out.println(s.length());
System.out.println(s.codePointCount(0, s.length()-1));

// Print encoding of first character, then encoding of first codepoint.
System.out.println(Integer.toHexString(s.charAt(0)));
System.out.println(Integer.toHexString(s.codePointAt(0)));

// Here's how to safely loop through a string that may contain
// supplementary characters
String tricky = s + "Testing" + s + "!";
int i = 0, n = tricky.length();
while(i < n) {
    // Get the codepoint at the current position
    int cp = tricky.codePointAt(i);
    if (cp < '\uffff') System.out.println((char) cp);
    else System.out.println("\\u" + Integer.toHexString(cp));

    // Increment the string index by one codepoint (1 or 2 chars).
    i = tricky.offsetByCodePoints(i, 1);
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.2.9. Formatting Text with printf() and format( )

A common task when working with text output is to combine values of various types into a single block of human-readable text. One way to accomplish this relies on the string-conversion power of Java's string concatenation operator. It results in code like this:

```
System.out.println(username + " logged in after " + numattempts +
                   "attempts. Last login at: " + lastLoginDate);
```

Java 5.0 introduces an alternative that is familiar to C programmers: a `printf( )` method. "printf" is short for "print formatted" and it combines the printing and formatting functions into one call. The `printf( )` method has been added to the `PrintWriter` and `PrintStream` output stream classes in Java 5.0. It is a varargs method that expects one or more arguments. The first argument is the "format string." It specifies the text to be printed and typically includes one or more "format specifiers," which are escape sequences beginning with character `%`. The remaining arguments to `printf( )` are values to be converted to strings and substituted into the format string in place of the format specifiers. The format specifiers constrain the types of the remaining arguments and specify exactly how they are converted to strings. The string concatenation shown above can be rewritten as follows in Java 5.0:

```
System.out.printf("%s logged in after %d attempts. Last login at: %tc%n",
                  username, numattempts, lastLoginDate);
```

The format specifier `%s` simply substitutes a string. `%d` expects the corresponding argument to be an integer and displays it as such. `%tc` expects a `Date`, `Calendar`, or number of milliseconds and converts that value to text representation of the full date and time. `%n` performs no conversion: it simply outputs the platform-specific line terminator, just as the `println( )` method does.

The conversions performed by `printf()` are all properly localized. Times and dates are displayed with locale-appropriate punctuation, for example. And if you request that a number be displayed with a thousands separator, you'll get locale-specific punctuation there, too (a comma in England and a period in France, for example).

In addition to the basic `printf( )` method, `PrintWriter` and `PrintStream` also define a synonymous method named `format()` : it takes exactly the same arguments and behaves in exactly the same way. The `String` class also has a `format()` method in Java 5.0. This static `String.format()` method behaves like `PrintWriter.format( )` except that instead of printing the formatted string to a stream, it simply returns it:

```
// Format a string, converting a double value to text using two decimal
// places and a thousands separator.
double balance = getBalance();
String msg = String.format("Account balance: $%,.2f", balance);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The `java.util.Formatter` class is the general-purpose formatter class behind the `printf()` and `format( )` utility methods. It can format text to any `Appendable` object or to a named file. The following code uses a `Formatter` object to write a file:

```
public static void writeFile(String filename, String[] lines)
    throws IOException
{
    Formatter out = new Formatter(filename);  // format to a named file
    for(int i = 0; i < lines.length; i++) {
        // Write a line of the file
        out.format("%d: %s%n", i, lines[i]);
        // Check for exceptions
        IOException e = out.ioException();
        if (e != null) throw e;
    }
    out.close();
}
```

When you concatenate an object to a string, the object is converted to a string by calling its `toString()` method. This is what the `Formatter` class does by default as well. Classes that want more precise control over their formatting can implement the `java.util.Formattable` interface in addition to implementing `toString()`.

We'll see additional examples of formatting with `printf( )` when we cover the APIs for working with numbers, dates, and times. See `java.util.Formatter` for a complete list of available format specifiers and options.

5.2.10. Logging

Simple terminal-based programs can send their output and error messages to the console with `System.out.println()` or `System.out.print( )`. Server programs that run unattended for long periods need a different solution for output: the hardware they run on may not have a display terminal attached, and, if it does, there is unlikely to be anyone looking at it. Programs like this need *logging* functionality in which output messages are sent to a file for later analysis or through a network socket for remote monitoring. Java 1.4 provides a logging API in the `java.util.logging` package.

Typically, the application developer uses a `Logger` object associated with the class or package of the application to generate log messages at any of seven severity levels (see `java.util.logging.Level`). These messages may report errors and warnings or provide informational messages about interesting events in the application's life cycle. They can include debugging information or even trace the execution of important methods within the program.

The system administrator or end user of the application is responsible for setting up a logging configuration file that specifies where log messages are directed (the console, a file, a network socket, or a combination of these), how they are formatted (as plain text or XML documents), and at what severity threshold they are logged (log messages with a severity below the specified threshold are discarded with very little overhead and should not significantly impact the performance of the application). The logging level severity threshold can be configured independently so that `Logger` objects associated with different classes or packages can be "tuned in" or "tuned out." Because of this end-user configurability, you should feel free to use logging output liberally in your program. In normal operation, most log messages will be discarded efficiently and automatically. During program development, or when diagnosing a problem in a deployed application, however, the log messages can prove very valuable.

For most applications, using the Logging API is quite simple. Obtain a named `Logger` object whenever necessary by calling the static `Logger.getLogger( )` method, passing the class or package name of the application as the logger name. Then, use one of the many `Logger` instance methods to generate log messages. The easiest methods to use have names that correspond to severity levels, such as `severe()` , `warning( )`, and `info()`. Here is some sample code:

```
import java.util.logging.*;

// Get a Logger object named after the current package
Logger logger = Logger.getLogger("com.davidflanagan.servers.pop");
logger.info("Starting server.");       // Log an informational message
ServerSocket ss;                       // Do some stuff
try { ss = new ServerSocket(110); }
catch(Exception e) {                   // Log exceptions
  logger.log(Level.SEVERE, "Can't bind port 110", e); // Complex log message
  logger.warning("Exiting");                          // Simple warning
  return;
}
logger.fine("got server socket"); // Fine-detail (low-severity) debug message
```

5.2.11. Pattern Matching with Regular Expressions

In Java 1.4 and later, you can perform textual pattern matching with regular expressions. Regular expression support is provided by the `Pattern` and `Matcher` classes of the `java.util.regex` package, but the `String` class defines a number of convenient methods that allow you to use regular expressions even more simply. Regular expressions use a fairly complex grammar to describe patterns of characters. The Java implementation uses the same regex syntax as the Perl 5 programming language. See the `java.util.regex.Pattern` class in the reference section for a summary of this syntax or consult a good Perl programming book for further details. For a complete tutorial on Perl-style regular expressions, see *Mastering Regular Expressions* (O'Reilly).

The simplest `String` method that accepts a regular expression argument is `matches( )`; it returns `true` if the string matches the pattern defined by the specified regular expression:

```
// This string is a regular expression that describes the pattern of a typical
// sentence. In Perl-style regular expression syntax, it specifies
// a string that begins with a capital letter and ends with a period,
// a question mark, or an exclamation point.
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
String pattern = "^[A-Z].*[\\.?!]$";
String s = "Java is fun!";
s.matches(pattern);  // The string matches the pattern, so this returns true.
```

The `matches( )` method returns `true` only if the entire string is a match for the specified pattern. Perl programmers should note that this differs from Perl's behavior, in which a match means only that some portion of the string matches the pattern. To determine if a string or any substring matches a pattern, simply alter the regular expression to allow arbitrary characters before and after the desired pattern. In the following code, the regular expression characters `.*` match any number of arbitrary characters:

```
s.matches(".*\\bJava\\b.*"); // True if s contains the word "Java" anywhere
                             // The b specifies a word boundary
```

If you are already familiar with Perl's regular expression syntax, you know that it relies on the liberal use of backslashes to escape certain characters. In Perl, regular expressions are language primitives and their syntax is part of the language itself. In Java, however, regular expressions are described using strings and are typically embedded in programs using string literals. The syntax for Java string literals also uses the backslash as an escape character, so to include a single backslash in the regular expression, you must use two backslashes. Thus, in Java programming, you will often see double backslashes in regular expressions.

In addition to matching, regular expressions can be used for search-and-replace operations. The `replaceFirst( )` and `replaceAll( )` methods search a string for the first substring or all substrings that match a given pattern and replace the string or strings with the specified replacement text, returning a new string that contains the replacements. For example, you could use this code to ensure that the word "Java" is correctly capitalized in a string `s`:

```
s.replaceAll("(?i)\\bjava\\b",// Pattern: the word "java", case-insensitive
             "Java");         // The replacement string, correctly capitalized
```

The replacement string passed to `replaceAll()` and `replaceFirst( )` need not be a simple literal string; it may also include references to text that matched parenthesized subexpressions within the pattern. These references take the form of a dollar sign followed by the number of the subexpression. (If you are not familiar with parenthesized subexpressions within a regular expression, see `java.util.regex.Pattern` in the reference section.) For example, to search for words such as JavaBean, JavaScript, JavaOS, and JavaVM (but not Java or Javanese) and to replace the Java prefix with the letter J without altering the suffix, you could use code such as:

```
s.replaceAll("\\bJava([A-Z]\\w+)",  // The pattern
             "J$1");      // J followed by the suffix that matched the
                          // subexpression in parentheses: [A-Z]\\w+
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The other `String` method that uses regular expressions is `split()`, which returns an array of the substrings of a string, separated by delimiters that match the specified pattern. To obtain an array of words in a string separated by any number of spaces, tabs, or newlines, do this:

```
String sentence = "This is a\n\ttwo-line sentence";
String[] words = sentence.split("[ \t\n\r]+");
```

An optional second argument specifies the maximum number of entries in the returned array.

The `matches( )`, `replaceFirst()`, `replaceAll( )`, and `split()` methods are suitable for when you use a regular expression only once. If you want to use a regular expression for multiple matches, you should explicitly use the `Pattern` and `Matcher` classes of the `java.util.regex` package. First, create a `Pattern` object to represent your regular expression with the static `Pattern.compile()` method. (Another reason to use the `Pattern` class explicitly instead of the `String` convenience methods is that `Pattern.compile( )` allows you to specify flags such as `Pattern.CASE_INSENSITIVE` that globally alter the way the pattern matching is done.) Note that the `compile()` method can throw a `PatternSyntaxException` if you pass it an invalid regular expression string. (This exception is also thrown by the various `String` convenience methods.) The `Pattern` class defines `split()` methods that are similar to the `String.split()` methods. For all other matching, however, you must create a `Matcher` object with the `matcher()` method and specify the text to be matched against:

```
import java.util.regex.*;

Pattern javaword = Pattern.compile("\\bJava(\\w*)", Pattern.CASE_INSENSITIVE);
Matcher m = javaword.matcher(sentence);
boolean match = m.matches();  // True if text matches pattern exactly
```

Once you have a `Matcher` object, you can compare the string to the pattern in various ways. One of the more sophisticated ways is to find all substrings that match the pattern:

```
String text = "Java is fun; JavaScript is funny.";
m.reset(text);  // Start matching against a new string
// Loop to find all matches of the string and print details of each match
while(m.find()) {
  System.out.println("Found '" + m.group(0) + "' at position " + m.start(0));
  if (m.start(1) < m.end(1)) System.out.println("Suffix is " + m.group(1));
}
```

The `Matcher` class has been enhanced in several ways in Java 5.0. The most important of these is the ability to save the results of the most recent match in a `MatchResult` object. The previous algorithm that finds all matches in a string could be rewritten in Java 5.0 as follows:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
import java.util.regex.*;
import java.util.*;

public class FindAll {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile(args[0]);
        String text = args[1];

        List<MatchResult> results = findAll(pattern, text);
        for(MatchResult r : results) {
            System.out.printf("Found '%s' at (%d,%d)%n",
                              r.group(), r.start(), r.end());
        }
    }

    public static List<MatchResult> findAll(Pattern pattern, CharSequence text)
    {
        List<MatchResult> results = new ArrayList<MatchResult>();
        Matcher m = pattern.matcher(text);
        while(m.find()) results.add(m.toMatchResult());
        return results;
    }
}
```

5.2.12. Tokenizing Text

`java.util.Scanner` is a general purpose text tokenizer, added in Java 5.0 to complement the `java.util.Formatter` class described earlier in this chapter. `Scanner` takes full advantage of Java regular expressions and can take its input text from a string, file, stream, or any object that implements the `java.lang.Readable` interface. `Readable` is also new in Java 5.0 and is the opposite of the `Appendable` interface.

A `Scanner` can break its input text into tokens separated by whitespace or any desired delimiter character or regular expression. It implements the `Iterator<String>` interface, which allows for simple looping through the returned tokens. `Scanner` also defines a variety of convenience methods for parsing tokens as `boolean`, integer, or floating-point values, with locale-sensitive number parsing. It has `skip( )` methods for skipping input text that matches a specified pattern and also has methods for searching ahead in the input text for text that matches a specified pattern.

Here's how you could use a `Scanner` to break a `String` into space-separated words:

```
public static List<String> getTokens(String line) {
    List<String> result = new ArrayList<String>();
    for(Scanner s = Scanner.create(line); s.hasNext(); )
        result.add(s.next());
    return result;
}
```

Here's how you might use a `Scanner` to break a file into lines:

```
public static void printLines(File f) throws IOException {
    Scanner s = Scanner.create(f);
    // Use a regex to specify line terminators as the token delimiter
    s.useDelimiter("\r\n|\n|\r");
    while(s.hasNext()) System.out.println(s.next());
}
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

The following method uses `Scanner` to parse an input line in the form `x + y = z`. It demonstrates the ability of a `Scanner` to scan numbers. Note that `Scanner` does not just parse Java-style integer literals: it supports thousands separators and does so in a locale-sensitive way—for example, it would parse the integer 1,234 for an American user and 1.234 for a French user. This code also demonstrates the `skip()` method and shows that a `Scanner` can scan text directly from an `InputStream`.

```
public static boolean parseSum() {
    System.out.print("enter sum> "); // Prompt the user for input
    System.out.flush();          // Make sure prompt is visible immediately

    try {
        // Read and parse the user's input from the console
        Scanner s = Scanner.create(System.in);
        s.useDelimiter("");    // Don't require spaces between tokens
        int x = s.nextInt();   // Parse an integer
        s.skip("\\s*\\+\\s*"); // Skip optional space and literal +
        int y = s.nextInt();   // Parse another integer
        s.skip("\\s*=\\s*");   // Skip optional space and literal =
        int z = s.nextInt();   // Parse a third integer

        return x + y == z;
    }
    catch(InputMismatchException e) { // pattern does not match
        throw new IllegalArgumentException("syntax error");
    }
    catch(NoSuchElementException e) { // no more input available
        throw new IllegalArgumentException("syntax error");
    }
}
```

5.2.13. StringTokenizer

A number of other Java classes operate on strings and characters. One notable class is `java.util.StringTokenizer`, which you can use to break a string of text into its component words:

```
String s = "Now is the time";
java.util.StringTokenizer st = new java.util.StringTokenizer(s);
while(st.hasMoreTokens()) {
  System.out.println(st.nextToken());
}
```

You can even use this class to tokenize words that are delimited by characters other than spaces:

```
String s = "a:b:c:d";
java.util.StringTokenizer st = new java.util.StringTokenizer(s, ":");
```

`java.io.StreamTokenizer` is another tokenizing class. It has a more complicated API and has more powerful features than `StringTokenizer`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.3. Numbers and Math

Java provides the `byte`, `short`, `int`, `long`, `float`, and `double` primitive types for representing numbers. The `java.lang` package includes the corresponding `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` classes, each of which is a subclass of `Number`. These classes can be useful as object wrappers around their primitive types, and they also define some useful constants:

```
// Integral range constants: Integer, Long, and Character also define these
Byte.MIN_VALUE      // The smallest (most negative) byte value
Byte.MAX_VALUE      // The largest byte value
Short.MIN_VALUE     // The most negative short value
Short.MAX_VALUE     // The largest short value

// Floating-point range constants: Double also defines these
Float.MIN_VALUE     // Smallest (closest to zero) positive float value
Float.MAX_VALUE     // Largest positive float value

// Other useful constants
Math.PI             // 3.14159265358979323846
Math.E              // 2.7182818284590452354
```

5.3.1. Mathematical Functions

The `Math` class defines a number of methods that provide trigonometric, logarithmic, exponential, and rounding operations, among others. This class is primarily useful with floating-point values. For the trigonometric functions, angles are expressed in radians. The logarithm and exponentiation functions are base *e*, not base 10. Here are some examples:

```
double d = Math.toRadians(27);       // Convert 27 degrees to radians
d = Math.cos(d);                     // Take the cosine
d = Math.sqrt(d);                    // Take the square root
d = Math.log(d);                     // Take the natural logarithm
d = Math.exp(d);                     // Do the inverse: e to the power d
d = Math.pow(10, d);                 // Raise 10 to this power
d = Math.atan(d);                    // Compute the arc tangent
d = Math.toDegrees(d);               // Convert back to degrees
double up = Math.ceil(d);            // Round to ceiling
double down = Math.floor(d);         // Round to floor
long nearest = Math.round(d);        // Round to nearest
```

In Java 5.0, several new functions have been added to the `Math` class, including the following:

```
double d = 27;
d = Math.cbrt(d);      // cube root
d = Math.log10(d);     // base-10 logarithm
d = Math.sinh(d);      // hyperbolic sine.  Also cosh() and tanh()
d = Math.hypot(3, 4);  // Hypotenuse
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.3.2. Random Numbers

The `Math` class also defines a rudimentary method for generating pseudo-random numbers, but the `java.util.Random` class is more flexible. If you need *very* random pseudo-random numbers, you can use the `java.security.SecureRandom` class:

```
// A simple random number
double r = Math.random();     // Returns d such that: 0.0 <= d < 1.0

// Create a new Random object, seeding with the current time
java.util.Random generator = new java.util.Random(System.currentTimeMillis());
double d = generator.nextDouble();   // 0.0 <= d < 1.0
float f = generator.nextFloat();     // 0.0 <= f < 1.0
long l = generator.nextLong();       // Chosen from the entire range of long
int i = generator.nextInt();         // Chosen from the entire range of int
i = generator.nextInt(limit);        // 0 <= i < limit (Java 1.2 and later)
boolean b = generator.nextBoolean(); // true or false (Java 1.2 and later)
d = generator.nextGaussian();        // Mean value: 0.0; std. deviation: 1.0
byte[] randomBytes = new byte[128];
generator.nextBytes(randomBytes);    // Fill in array with random bytes

// For cryptographic strength random numbers, use the SecureRandom subclass
java.security.SecureRandom generator2 = new java.security.SecureRandom();
// Have the generator generate its own 16-byte seed; takes a *long* time
generator2.setSeed(generator2.generateSeed(16)); // Extra random 16-byte seed
// Then use SecureRandom like any other Random object
generator2.nextBytes(randomBytes);   // Generate more random bytes
```

5.3.3. Big Numbers

The `java.math` package contains the `BigInteger` and `BigDecimal` classes. These classes allow you to work with arbitrary-size and arbitrary-precision integers and floating-point values. For example:

```
import java.math.*;

// Compute the factorial of 1000
BigInteger total = BigInteger.valueOf(1);
for(int i = 2; i <= 1000; i++)
  total = total.multiply(BigInteger.valueOf(i));
System.out.println(total.toString());
```

In Java 1.4, `BigInteger` has a method to randomly generate large prime numbers, which is useful in many cryptographic applications:

```
BigInteger prime =
  BigInteger.probablePrime(1024,        // 1024 bits long
                           generator2); // Source of randomness. From above.
```

The `BigDecimal` class has been overhauled in Java 5.0 and is much more usable in this release. In addition to its utility for representing very large or very precise floating point numbers, it is also useful for financial calculations because it relies on a decimal representation of fractions rather than a binary representation. `float` and `double` values cannot precisely represent a number as simple as 0.1, and this can cause rounding errors that are often unacceptable when representing monetary values. `BigDecimal` and its associated `MathContext` and `RoundingMode` types provide a solution. For example:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Compute monthly interest payments on a loan
public static BigDecimal monthlyPayment(int amount, // amount of loan
                                        int years,  // term in years
                                        double apr) // annual interest %
{
    // Convert the loan amount to a BigDecimal
    BigDecimal principal = new BigDecimal(amount);

    // Convert term of loan in years to number of monthly payments
    int payments=years*12;

    // Convert interest from annual percent to a monthly decimal
    BigDecimal interest = BigDecimal.valueOf(apr);
    interest = interest.divide(new BigDecimal(100));     // as fraction
    interest = interest.divide(new BigDecimal(12));      // monthly

    // The monthly payment computation
    BigDecimal x = interest.add(BigDecimal.ONE).pow(payments);
    BigDecimal y = principal.multiply(interest).multiply(x);
    BigDecimal monthly = y.divide(x.subtract(BigDecimal.ONE),
                                  MathContext.DECIMAL64);  // note context

    // Convert to two decimal places
    monthly = monthly.setScale(2, RoundingMode.HALF_EVEN);

    return monthly;
}
```

5.3.4. Converting Numbers from and to Strings

A Java program that operates on numbers must get its input values from somewhere. Often, such a program reads a textual representation of a number and must convert it to a numeric representation. The various `Number` subclasses define useful conversion methods:

```
String s = "-42";
byte b = Byte.parseByte(s);             // s as a byte
short sh = Short.parseShort(s);         // s as a short
int i = Integer.parseInt(s);            // s as an int
long l = Long.parseLong(s);             // s as a long
float f = Float.parseFloat(s);          // s as a float (Java 1.2 and later)
f = Float.valueOf(s).floatValue();      // s as a float (prior to Java 1.2)
double d = Double.parseDouble(s);       // s as a double (Java 1.2 and later)
d = Double.valueOf(s).doubleValue();    // s as a double (prior to Java 1.2)

// The integer conversion routines handle numbers in other bases
byte b = Byte.parseByte("1011", 2);       // 1011 in binary is 11 in decimal
short sh = Short.parseShort("ff", 16);    // ff in base 16 is 255 in decimal

// The valueOf() method can handle arbitrary bases between 2 and 36
int i = Integer.valueOf("egg", 17).intValue();   // Base 17!

// The decode() method handles octal, decimal, or hexadecimal, depending
// on the numeric prefix of the string
short sh = Short.decode("0377").byteValue();   // Leading 0 means base 8
int i = Integer.decode("0xff").shortValue();   // Leading 0x means base 16
long l = Long.decode("255").intValue();        // Other numbers mean base 10

// Integer class can convert numbers to strings
String decimal = Integer.toString(42);
String binary = Integer.toBinaryString(42);
String octal = Integer.toOctalString(42);
String hex = Integer.toHexString(42);
String base36 = Integer.toString(42, 36);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.3.5. Formatting Numbers

The `printf()` and `format( )` methods of Java 5.0 described earlier in this chapter work well for formatting numbers. The `%d` format specifier is for formatting integers in decimal format:

```
// Format int, long and BigInteger to the string "1 10 100"
String s = String.format("%d %d %d", 1, 10L, BigInteger.TEN.pow(2));
// Add thousands separators
s = String.format("%,d", Integer.MAX_VALUE); // "2,147,483,647"
// Output value right-justified in a field 8 characters wide
s = String.format("%8d", 123);              // "     123"
// Pad on the left with zeros to make 5 digits total
s = String.format("%05d", 123);             // "00123"
```

Floating-point numbers can be formatted using `%f`, `%e`, or `%g` format specifiers, which differ in whether and when exponential notation is used:

```
double x = 1.234E9; // (1.234 billion)
// returns "1234000000.000000 1.234000e+09 1.234000e+09 1234.000000"
s = String.format("%f %e %g %g", x, x, x, x/1e6);
```

You'll notice that the numbers above are all formatted with six digits following the decimal point. This default can be altered by specifying a *precision* in the format string:

```
// display a BigDecimal with 2 significant digits
s = String.format("%.2f", new BigDecimal("1.234"));  // "1.23"
```

Other flags can be applied to floating-point conversions as well. The following code formats a column of numbers right-justified within a field 10 characters wide. Each number has two digits following the decimal place and includes thousands separators when necessary. Negative values are formatted in parentheses, a common formatting convention in accounting.

```
// A column of 4 numbers. %n is newline.
s = String.format("%(,10.2f%n%(,10.2f%n%(,10.2f%n%(,10.2f%n",
                  BigDecimal.TEN,                    //      10.00
                  BigDecimal.TEN.movePointRight(3),  //  10,000.00
                  BigDecimal.TEN.movePointLeft(3),   //       0.01
                  BigDecimal.TEN.negate());          //    (10.00)
```

See `java.util.Formatter` in the reference section for complete details on supported format specifiers and formatting options.

Prior to Java 5.0, numbers can be formatted using the `java.text.NumberFormat` class:

```
import java.text.*;

// Use NumberFormat to format and parse numbers for the current locale
NumberFormat nf = NumberFormat.getNumberInstance();  // Get a NumberFormat
System.out.println(nf.format(9876543.21)); // Format number for current locale
try {
  Number n = nf.parse("1.234.567,89");      // Parse strings according to locale
} catch (ParseException e) { /* Handle exception */ }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Monetary values are sometimes formatted differently than other numbers
NumberFormat moneyFmt = NumberFormat.getCurrencyInstance();
System.out.println(moneyFmt.format(1234.56)); // Prints $1,234.56 in U.S.
```

5.4. Dates and Times

Java allows dates and times to be represented and manipulated in three forms: as `long` values or as `java.util.Date` or `java.util.Calendar` objects. Java 5.0 introduces the enumerated type `java.util.concurrent.TimeUnit`. The values of this type represent time granularities or units: seconds, milliseconds, microseconds, and nanoseconds. They have useful convenience methods but do not themselves represent a time value.

5.4.1. Milliseconds and Nanoseconds

At the lowest level, dates and times are represented as a `long` value that holds the positive or negative number of milliseconds since midnight on January 1, 1970. This special date and time is known as the epoch and is measured in Greenwich Mean Time (GMT) or Universal Time (UTC). To query the current time in this millisecond representation, use `System.currentTimeMillis()`

```
long now = System.currentTimeMillis();
```

In Java 5.0 and later, you can use `System.nanoTime()` to query time in nanoseconds. This method returns a `long` number of nanoseconds `long`. Unlike `currentTimeMillis( )`, the `nanoTime()` does not return a time relative to any defined epoch. `nanoTime()` is good for measuring relative or elapsed time (as long as the elapsed time is not more than 292 years) but is not suitable for absolute time:

```
long start = System.nanoTime();
doSomething();
long end = System.nanoTime();
long elapsedNanoSeconds = end - start;
```

5.4.2. The Date Class

`java.util.Date` is an object wrapper around a `long` that holds a number of milliseconds since the epoch. Using a `Date` object instead of a `long` allows simple conversion to a nonlocalized string with the `toString` method. `Date` objects can be compared for equality with the `equals( )` method and they can be compared for order with the `compareTo()` method or the `before( )` and `after()` methods.

The no-argument version of the `Date( )` constructor creates a `Date` that represents the current time. You can also pass a `long` number of milliseconds to create a `Date` that represents some other time. `getTime( )` returns the millisecond representation of the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`Date`. `Date` is a mutable class, so you can also pass a number of milliseconds to `setTime( )`.

`Date` has a number of methods for querying and setting the year, month, day, hour, minute, and second. All of these methods have been deprecated, however, in favor of the `Calendar` class, described next.

5.4.3. The Calendar Class

The `java.util.Calendar` class is a properly localized version of `Date`. It is simply a wrapper around a `long` number of milliseconds but can represent that instant in time according to the calendar of the current locale (usually a Gregorian calendar) and the time zone of the current locale. Furthermore, it has methods for querying, setting, and doing arithmetic on the various fields of the date and time.

The code below shows common uses of the `Calendar` class. Note that the `set( )`, `get()`, and `add( )` methods all take an initial argument that specifies what field of the date or time is being set, queried, or added to. Fields such as year, day of month, day of week, hour, minute, and second are defined by integer constants in the class. Other integer constants define values for the months and weekdays of the Gregorian calendar. The month constant `UNDECIMBER` represents a 13th month used in lunar calendars.

```
// Get a Calendar for current locale and time zone
Calendar cal = Calendar.getInstance();

// Figure out what day of the year today is
cal.setTimeInMillis(System.currentTimeMillis()); // Set to the current time
int dayOfYear = cal.get(Calendar.DAY_OF_YEAR);  // What day of the year is it?

// What day of the week does the leap day in the year 2008 occur on?
cal.set(2008, Calendar.FEBRUARY, 29);            // Set year, month, day fields
int dayOfWeek = cal.get(Calendar.DAY_OF_WEEK);   // Query a different field

// What day of the month is the 3rd Thursday of May, 2005?
cal.set(Calendar.YEAR, 2005);                    // Set the year
cal.set(Calendar.MONTH, Calendar.MAY);           // Set the month
cal.set(Calendar.DAY_OF_WEEK,Calendar.THURSDAY); // Set the day of week
cal.set(Calendar.DAY_OF_WEEK_IN_MONTH, 3);       // Set the week
int dayOfMonth = cal.get(Calendar.DAY_OF_MONTH); // Query the day in month

// Get a Date object that represents three months from now
cal.setTimeInMillis(System.currentTimeMillis()); // Current time
cal.add(Calendar.MONTH, 3);                      // Add 3 months
Date expiration = cal.getTime();                 // Retrieve result as a Date
long millis = cal.getTimeInMillis();             // or get it as a long
```

5.4.4. Formatting Dates and Times

The `toString()` method of `Date` produces a textual representation of a date and time but does no localization and allows no customization of which fields (day, month and year or hours and minutes, for example) are to be displayed. The `toString( )` method should be used only to produce a machine-readable timestamp, not a human-readable string.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Like numbers, dates and times can be converted to strings using the `String.format( )` method and the related `java.util.Formatter` class of Java 5.0. Format strings for displaying dates and times are all two-character sequences that begin with the letter t. The second letter of each sequence specifies the field or set of fields of the date or time to display. For example `%tR` displays the hours and minutes fields using 24-hour time, and `%tD` displays the month, day, and year fields separated by slashes. `String.format( )` can format a date or time specified as a `long`, a `Date`, or a `Calendar`:

```
// current hours and minutes
long now = System.currentTimeMillis();
String s = String.format("%tR", now);         // "15:12"

// Current month/day/year
Date d = new Date(now);
s = String.format("%tD", d);                  // "07/13/04"

// Hours and minutes using 12-hour clock
Calendar c = Calendar.getInstance();
c.setTime(d);
s = String.format("%tl:%tM %tp", now, d, c);  // "3:12 pm"
```

Prior to Java 5.0 and its `Formatter` class, you can format dates and times using the `java.text.DateFormat` class, which automatically handles locale-specific conventions for date and time formatting. `DateFormat` even works correctly in locales that use a calendar other than the common era (Gregorian) calendar in use throughout much of the world:

```
import java.util.Date;
import java.text.*;

// Display today's date using a default format for the current locale
DateFormat defaultDate = DateFormat.getDateInstance();
System.out.println(defaultDate.format(new Date()));

// Display the current time using a short time format for the current locale
DateFormat shortTime = DateFormat.getTimeInstance(DateFormat.SHORT);
System.out.println(shortTime.format(new Date()));

// Display date and time using a long format for both
DateFormat longTimestamp =
  DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
System.out.println(longTimestamp.format(new Date()));

// Use SimpleDateFormat to define your own formatting template
// See java.text.SimpleDateFormat for the template syntax
DateFormat myformat = new SimpleDateFormat("yyyy.MM.dd");
System.out.println(myformat.format(new Date()));
try {   // DateFormat can parse dates too
  Date leapday = myformat.parse("2000.02.29");
}
catch (ParseException e) { /* Handle parsing exception */ }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.5. Arrays

The `java.lang.System` class defines an `arraycopy( )` method that is useful for copying specified elements in one array to a specified position in a second array. The second array must be the same type as the first, and it can even be the same array:

```
char[] text = "Now is the time".toCharArray();
char[] copy = new char[100];
// Copy 10 characters from element 4 of text into copy, starting at copy[0]
System.arraycopy(text, 4, copy, 0, 10);

// Move some of the text to later elements, making room for insertions
System.arraycopy(copy, 3, copy, 6, 7);
```

In Java 1.2 and later, the `java.util.Arrays` class defines useful array-manipulation methods, including methods for sorting and searching arrays:

```
import java.util.Arrays;

int[] intarray = new int[] { 10, 5, 7, -3 }; // An array of integers
Arrays.sort(intarray);                       // Sort it in place
int pos = Arrays.binarySearch(intarray, 7);  // Value 7 is found at index 2
pos = Arrays.binarySearch(intarray, 12);     // Not found: negative return value

// Arrays of objects can be sorted and searched too
String[] strarray = new String[] { "now", "is", "the", "time" };
Arrays.sort(strarray);   // sorted to: { "is", "now", "the", "time" }

// Arrays.equals() compares all elements of two arrays
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone);  // Yes, they're equal

// Arrays.fill() initializes array elements
byte[] data = new byte[100];          // An empty array; elements set to 0
Arrays.fill(data, (byte) -1);         // Set them all to -1
Arrays.fill(data, 5, 10, (byte) -2);  // Set elements 5, 6, 7, 8, 9 to -2
```

Arrays can be treated and manipulated as objects in Java. Given an arbitrary object `o`, you can use code such as the following to find out if the object is an array and, if so, what type of array it is:

```
Class type = o.getClass();
if (type.isArray()) {
  Class elementType = type.getComponentType();
}
```

5.6. Collections

The Java Collections Framework is a set of important utility classes and interfaces in the `java.util` package for working with collections of objects. The Collections Framework defines two fundamental types of collections. A `Collection` is a group of objects while a `Map` is a set of mappings, or associations, between objects. A `Set` is a type of `Collection` with no duplicates, and a `List` is a `Collection` in which the elements are ordered. `SortedSet` and `SortedMap` are specialized sets and maps that maintain their

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

elements in a sorted order. `Collection` , `Set`, `List`, `Map`, `SortedSet`, and `SortedMap` are all interfaces, but the `java.util` package also defines various concrete implementations, such as lists based on arrays and linked lists, and maps and sets based on hashtables or binary trees. Other important interfaces are `Iterator` and `ListIterator`, which allow you to loop through the objects in a collection. The Collections Framework was added in Java 1.2, but prior to that release you can use `Vector` and `Hashtable`, which are approximately the same as `ArrayList` and `HashMap`.

In Java 1.4, the Collections API added the `RandomAccess` marker interface, which is implemented by `List` implementations that support efficient random access (i.e., it is implemented by `ArrayList` and `Vector` but not by `LinkedList`). Java 1.4 also introduced `LinkedHashMap` and `LinkedHashSet`, which are hashtable-based maps and sets that preserve the insertion order of elements. Finally, `IdentityHashMap` is a hashtable-based `Map` implementation that uses the == operator to compare key objects rather than using the `equals()` method to compare them.

The Collections Framework has been overhauled in Java 5.0 to use generics (see Chapter 4). Java 5.0 also adds `EnumSet` and `EnumMap` classes that are specialized for working with enumerated values (see Chapter 4) and the `java.lang.Iterable` interface used by the new `for/in` looping statement. Finally, Java 5.0 adds the `Queue` interface. Most of the interesting `Queue` implementations are `BlockingQueue` implementations in `java.util.concurrent`.

5.6.1. The Collection Interface

`Collection<E>` is a parameterized interface that represents a generic group of objects of type `E`. The group may or may not allow duplicate elements and may or may not impose an ordering on the elements. Methods are defined for adding and removing objects from the group, testing an object for membership in the group, and iterating through all elements in the group. Additional methods return the elements of the group as an array and return the size of the collection.

The Java Collections Framework does not provide any implementations of `Collection`, but this interface is still very important because it defines the features common to all `Set` , `List`, and `Queue` implementations. The following code illustrates the operations you can perform on `Collection` objects:

```
// Create some collections to work with.
Collection<String> c = new HashSet<String>();  // An empty set
// We'll see these utility methods later
Collection<String> d = Arrays.asList("one", "two");    // immutable
Collection<String> e = Collections.singleton("three"); // immutable
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Add elements to a collection. These methods return true if the collection
// changes, which is useful with Sets that don't allow duplicates.
c.add("zero");           // Add a single element
c.addAll(d);             // Add a collection of elements

// Copy a collection: most implementations have a copy constructor
Collection<String> copy = new ArrayList<String>(c);

// Remove elements from a collection.
// All but clear() return true if the collection changes.
c.remove("zero");        // Remove a single element
c.removeAll(e);          // Remove a collection of elements
c.retainAll(d);          // Remove all elements that are not in e
c.clear();               // Remove all elements from the collection

// Querying collection size
boolean b = c.isEmpty(); // Collection is now empty
int s = c.size();        // Collection size is now 0.

// Restore collection from the copy we made
c.addAll(copy);

// Test membership in the collection.  Membership is based on the equals()
// method, not the == operator.
b = c.contains("zero");  // true
b = c.containsAll(d);    // true

// Iterate through collection elements with a while loop.
// Some implementations (such as lists) guarantee an order of iteration
// Others make no guarantees.
Iterator<String> iterator = c.iterator();
while(iterator.hasNext()) System.out.println(iterator.next());

// Iteration with a for loop
for(Iterator<String> i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());

// Java 5.0 iteration using a for/in loop
for(String word : c) System.out.println(word);

// Most Collection implementations have a useful toString() method
System.out.println(c);   // As an alternative to the iterations above

// Obtain an array of collection elements.  If the iterator guarantees
// an order, this array has the same order. The array is a copy, not a
// reference to an internal data structure.
Object[] elements = c.toArray();

// If we want the elements in a String[], we must pass one in
String[] strings = c.toArray(new String[c.size()]);

// Or we can pass an empty String[] just to specify the type and
// the toArray() method will allocate an array for us
strings = c.toArray(new String[0]);
```

Remember that you can use any of the methods shown above with any `Set`, `List`, or `Queue`. These subinterfaces may impose membership restrictions or ordering constraints on the elements of the collection but still provide the same basic methods. Methods such as `add( )` , `remove()`, `clear( )`, and `retainAll()` that alter the collection are optional, and read-only implementations may throw `UnsupportedOperationException`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`Collection`, `Map`, and their subinterfaces do *not* extend the `Cloneable` or `Serializable` interfaces. All of the collection and map implementation classes provided in the Java Collections Framework, however, do implement these interfaces.

Some collection implementations place restrictions on the elements that they can contain. An implementation might prohibit `null` as an element, for example. And `EnumSet` restricts membership to the values of a specified enumerated type. Attempting to add a prohibited element to a collection always throws an unchecked exception such as `NullPointerException` or `ClassCastException`. Checking whether a collection contains a prohibited element may also throw such an exception, or it may simply return `false`.

5.6.2. The Set Interface

A *set* is a collection of objects that does not allow duplicates: it may not contain two references to the same object, two references to `null`, or references to two objects `a` and `b` such that `a.equals(b)`. Most general-purpose `Set` implementations impose no ordering on the elements of the set, but ordered sets are not prohibited (see `SortedSet` and `LinkedHashSet`). Sets are further distinguished from ordered collections like lists by the general expectation that they have an efficient `contains( )` method that runs in constant or logarithmic time.

`Set` defines no additional methods beyond those defined by `Collection` but places additional restrictions on those methods. The `add( )` and `addAll()` methods of a `Set` are required to enforce the no-duplicates rules: they may not add an element to the `Set` if the set already contains that element. Recall that the `add()` and `addAll( )` methods defined by the `Collection` interface return `true` if the call resulted in a change to the collection and `false` if it did not. This return value is relevant for `Set` objects because the no-duplicates restriction means that adding an element does not always result in a change to the set.

Table 5-2 lists the implementations of the `Set` interface and summarizes their internal representation, ordering characteristics, member restrictions, and the performance of the basic `add( )`, `remove( )`, and `contains()` operations as well as iteration performance. You can read more about each class in the reference section. Note that `CopyOnWriteArraySet` is in the `java.util.concurrent` package; all the other implementations are part of `java.util`. Also note that `java.util.BitSet` is not a `Set` implementation. This legacy class is useful as a compact and efficient list of `boolean` values but is not part of the Java Collections Framework.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Table 5-2. Set Implementations

Class	Internal represen-tation	Element order	Member restric-tions	Basic opera-tions	Iteration perfor-mance	Notes
HashSet	hashtable	none	none	O(1)	O(capacity)	Best general-purpose implementation.
LinkedHashSet	linked hashtable	insertion order	none	O(1)	O(n)	Preserves insertion order.
EnumSet	bit fields	enum declaration	enum values	O(1)	O(n)	Holds non-`null` enum values only.
TreeSet	red-black tree	sorted ascending	comparable	O(log(n))	O(n)	`Comparable` elements or `Comparator`.
CopyOnWriteArraySet	array	insertion order	none	O(n)	O(n)	Threadsafe without synchronized methods.

The `TreeSet` implementation uses a red-black tree data structure to maintain a set that is iterated in ascending order according to the natural ordering of `Comparable` objects or according to an ordering specified by a `Comparator` object. `TreeSet` actually implements the `SortedSet` interface, which is a subinterface of `Set`.

`SortedSet` offers several interesting methods that take advantage of its sorted nature. The following code illustrates:

```
public static void testSortedSet(String[] args) {
    // Create a SortedSet
    SortedSet<String> s = new TreeSet<String>(Arrays.asList(args));

    // Iterate set: elements are automatically sorted
    for(String word : s) System.out.println(word);

    // Special elements
    String first = s.first();  // First element
    String last = s.last();    // Last element
    // Subrange views of the set
    SortedSet<String> tail = s.tailSet(first+'\0'); // all elements but first
    SortedSet<String> head = s.headSet(last);       // all elements but last
    SortedSet<String> middle = s.subSet(first+'\0', // all but ends
                                        last);
}
```

5.6.3. The List Interface

A `List` is an ordered collection of objects. Each element of a list has a position in the list, and the `List` interface defines methods to query or set the element at a particular position, or *index*. In this respect a `List` is like an array whose size changes as needed to accommodate the number of elements it contains. Unlike sets, lists allow duplicate elements.

In addition to its index-based `get( )` and `set()` methods, the `List` interface defines methods to add or remove an element at a particular index and also defines methods to return the index of the first or last occurrence of a particular value in the list. The

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`add( )` and `remove()` methods inherited from `Collection` are defined to append to the list and to remove the first occurrence of the specified value from the list. The inherited `addAll()` appends all elements in the specified collection to the end of the list, and another version inserts the elements at a specified index. The `retainAll()` and `removeAll( )` methods behave as they do for any `Collection`, retaining or removing multiple occurrences of the same value, if needed.

The `List` interface does not define methods that operate on a range of list indexes. Instead it defines a single `subList` method that returns a `List` object that represents just the specified range of the original list. The sublist is backed by the parent list, and any changes made to the sublist are immediately visible in the parent list. Examples of `subList( )` and the other basic `List` manipulation methods are below.

```
// Create lists to work with
List<String> l = new ArrayList<String>(Arrays.asList(args));
List<String> words = Arrays.asList("hello", "world");

// Querying and setting elements by index
String first = l.get(0);             // First element of list
String last = l.get(l.size()-1);     // Last element of list
l.set(0, last);                      // The last shall be first

// Adding and inserting elements.  add() can append or insert
l.add(first);       // Append the first word at end of list
l.add(0, first);    // Insert first word at the start of the list again
l.addAll(words);    // Append a collection at the end of the list
l.addAll(1, words); // Insert collection after first word

// Sublists: backed by the original list
List<String> sub = l.subList(1,3);  // second and third elements
sub.set(0, "hi");                   // modifies 2nd element of l
// Sublists can restrict operations to a subrange of backing list
String s = Collections.min(l.subList(0,4));
Collections.sort(l.subList(0,4));
// Independent copies of a sublist don't affect the parent list.
List<String> subcopy = new ArrayList<String>(l.subList(1,3));

// Searching lists
int p = l.indexOf(last);  // Where does the last word appear?
p = l.lastIndexOf(last);  // Search backward

// Print the index of all occurrences of last in l.  Note subList()
int n = l.size();
p = 0;
do {
    // Get a view of the list that includes only the elements we
    // haven't searched yet.
    List<String> list = l.subList(p, n);
    int q = list.indexOf(last);
    if (q == -1) break;
    System.out.printf("Found '%s' at index %d%n", last, p+q);
    p += q+1;
} while(p < n);

// Removing elements from a list
l.remove(last);         // Remove first occurrence of the element
l.remove(0);            // Remove element at specified index
l.subList(0,2).clear(); // Remove a range of elements using subList()
l.retainAll(words);     // Remove all but elements in words
l.removeAll(words);     // Remove all occurrences of elements in words
l.clear();              // Remove everything
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

A general expectation of `List` implementations is that they can be efficiently iterated, typically in time proportional to the size of the list. Lists do not all provide efficient random-access to the elements at any index, however. Sequential-access lists, such as the `LinkedList` class, provide efficient insertion and deletion operations at the expense of random access performance. In Java 1.4 and later, implementations that provide efficient random access implement the `RandomAccess` marker interface, and you can test for this interface with `instanceof` if you need to ensure efficient list manipulations:

```
List<?> l = ...;  // Arbitrary list we're passed to manipulate
// Ensure we can do efficient random access.  If not, use a copy constructor
// to make a random-access copy of the list before manipulating it.
if (!(l instanceof RandomAccess)) l = new ArrayList<?>(l);
```

The `Iterator` returned by the `iterator()` method of a `List` iterates the list elements in the order that they occur in the list. `List` implements `Iterable`, and lists can be iterated with a `for/in` loop just as any other collection can.

To iterate just a portion of a list, you can use the `subList( )` method to create a sublist view:

```
List<String> words = ...;  // Get a list to iterate

// Iterate just all elements of the list but the first
for(String word : words.subList(1, words.size()))
    System.out.println(word);
```

In addition to normal iteration, lists also provide enhanced bidirectional iteration using a `ListIterator` object returned by the `listIterator()` method. To iterate backward through a `List`, for example, start with a `ListIterator` with its cursor positioned after the end of the list:

```
ListIterator<String> li = words.listIterator(words.size());
while(li.hasPrevious()) {
    System.out.println(li.previous());
}
```

Table 5-3 summarizes the five general-purpose `List` implementations in the Java platform. `Vector` and `Stack` are legacy implementations left over from Java 1.0. `CopyOnWriteArrayList` is a new in Java 5.0 and is part of the `java.util.concurrent` package.

Table 5-3. List implementations

Class	Representation	Random access	Notes
ArrayList	array	yes	Best all-around implementation.
LinkedList	double-linked list	no	Efficient insertion and deletion.
CopyOnWriteArrayList	array	yes	Threadsafe; fast traversal, slow modification.
Vector	array	yes	Legacy class; synchronized method.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Class	Representation	Random access	Notes
Stack	array	yes	Extends `Vector`; adds `push( )`, `pop()`, `peek( )`.

5.6.4. The Map Interface

A *map* is a set of *key* objects and a mapping from each member of that set to a *value* object. The `Map` interface defines an API for defining and querying mappings. `Map` is part of the Java Collections Framework, but it does not extend the `Collection` interface, so a `Map` is a little-c collection, not a big-C `Collection`. `Map` is a parameterized type with two type variables. Type variable `K` represents the type of keys held by the map, and type variable `V` represents the type of the values that the keys are mapped to. A mapping from `String` keys to `Integer` values, for example, can be represented with a `Map<String,Integer>`.

The most important `Map` methods are `put()`, which defines a key/value pair in the map, `get( )`, which queries the value associated with a specified key, and `remove( )`, which removes the specified key and its associated value from the map. The general performance expectation for `Map` implementations is that these three basic methods are quite efficient: they should usually run in constant time and certainly no worse than in logarithmic time.

An important feature of `Map` is its support for "collection views." Although a `Map` is not a `Collection`, its keys can be viewed as a `Set`, its values can be viewed as a `Collection`, and its mappings can be viewed as a `Set` of `Map.Entry` objects. (`Map.Entry` is a nested interface defined within `Map`: it simply represents a single key/value pair.)

The sample code below shows the `get( )`, `put()`, `remove( )`, and other methods of a `Map` and also demonstrates some common uses of the collection views of a `Map`:

```
// Create maps to work with
Map<String,Integer> m = new HashMap<String,Integer>();  // New, empty map
// Immutable Map containing a single key-value pair
Map<String,Integer> singleton = Collections.singletonMap("testing", -1);
// Note this rarely-used syntax to explicitly specify the parameter
// types of the generic emptyMap() method.  The returned map is immutable
Map<String,Integer> empty = Collections.<String,Integer>emptyMap();

// Populate the map using the put() method to define mappings from array
// elements to the index at which each element appears
String[] words = { "this", "is", "a", "test" };
for(int i = 0; i < words.length; i++)
    m.put(words[i], i);  // Note autoboxing of int to Integer

// Each key must map to a single value. But keys may map to the same value
for(int i = 0; i < words.length; i++)
    m.put(words[i].toUpperCase(), i);

// The putAll() method copies mappings from another Map
m.putAll(singleton);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Query the mappings with the get() method
for(int i = 0; i < words.length; i++)
    if (m.get(words[i]) != i) throw new AssertionError();

// Key and value membership testing
m.containsKey(words[0]);        // true
m.containsValue(words.length);  // false

// Map keys, values, and entries can be viewed as collections
Set<String> keys = m.keySet();
Collection<Integer> values = m.values();
Set<Map.Entry<String,Integer>> entries = m.entrySet();

// The Map and its collection views typically have useful toString() methods
System.out.printf("Map: %s%nKeys: %s%nValues: %s%nEntries: %s%n",
                  m, keys, values, entries);

// These collections can be iterated.
// Most maps have an undefined iteration order (but see SortedMap)
for(String key : m.keySet()) System.out.println(key);
for(Integer value: m.values()) System.out.println(value);

// The Map.Entry<K,V> type represents a single key/value pair in a map
for(Map.Entry<String,Integer> pair : m.entrySet()) {
    // Print out mappings
    System.out.printf("'%s' ==> %d%n", pair.getKey(), pair.getValue());
    // And increment the value of each Entry
    pair.setValue(pair.getValue() + 1);
}

// Removing mappings
m.put("testing", null);   // Mapping to null can "erase" a mapping:
m.get("testing");         // Returns null
m.containsKey("testing"); // Returns true: mapping still exists
m.remove("testing");      // Deletes the mapping altogether
m.get("testing");         // Still returns null
m.containsKey("testing"); // Now returns false.

// Deletions may also be made via the collection views of a map.


// Additions to the map may not be made this way, however.
m.keySet().remove(words[0]);  // Same as m.remove(words[0]);
m.values().remove(2);         // Remove one mapping to the value 2
m.values().removeAll(Collections.singleton(4)); // Remove all mappings to 4
m.values().retainAll(Arrays.asList(2, 3));      // Keep only mappings to 2 & 3

// Deletions can also be done via iterators
Iterator<Map.Entry<String,Integer>> iter = m.entrySet().iterator();
while(iter.hasNext()) {
    Map.Entry<String,Integer> e = iter.next();
    if (e.getValue() == 2) iter.remove();
}

// Find values that appear in both of two maps.  In general, addAll() and
// retainAll() with
keySet() and values() allow union and intersection
Set<Integer> v = new HashSet<Integer>(m.values());
v.retainAll(singleton.values());

// Miscellaneous methods
m.clear();                // Deletes all mappings
m.size();                 // Returns number of mappings: currently 0
m.isEmpty();              // Returns true
m.equals(empty);          // true: Maps implementations override equals
```

The `Map` interface includes a variety of general-purpose and special-purpose implementations, which are summarized in Table 5-4. As always, complete details are in

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

the reference section. All classes in Table 5-4 are in the `java.util` package except `ConcurrentHashMap`, which is part of `java.util.concurrent`.

Table 5-4. Map implementations

Class	Representation	Since	null keys	null values	Notes
HashMap	hashtable	1.2	yes	yes	General-purpose implementation.
ConcurrentHashMap	hashtable	5.0	no	no	General-purpose threadsafe implementation; see `ConcurrentMap` interface.
EnumMap	array	5.0	no	yes	Keys are instances of an enum.
LinkedHashMap	hashtable plus list	1.4	yes	yes	Preserves insertion or access order.
TreeMap	red-black tree	1.2	no	yes	Sorts by key value. Operations are O(log(n)). See `SortedMap`.
IdentityHashMap	hashtable	1.4	yes	yes	Compares with = = instead of `equals( )`.
WeakHashMap	hashtable	1.2	yes	yes	Doesn't prevent garbage collection of keys.
Hashtable	hashtable	1.0	no	no	Legacy class; synchronized methods.
Properties	hashtable	1.0	no	no	Extends `Hashtable` with `String` methods.

The `ConcurrentHashMap` class of the `java.util.concurrent` package implements the `ConcurrentMap` interface of the same package. `ConcurrentMap` extends `Map` and defines some additional atomic operations that are important in multithreaded programming. For example, the `putIfAbsent()` method is like `put( )` but adds the key/value pair to the map only if the key is not already mapped.

`TreeMap` implements the `SortedMap` interface, which extends `Map` to add methods that take advantage of the sorted nature of the map. `SortedMap` is quite similar to the `SortedSet` interface. The `firstKey()` and `lastKey( )` methods return the first and last keys in the `keySet()`. And `headMap()` , `tailMap( )`, and `subMap()` return a restricted range of the original map.

5.6.5. The Queue and BlockingQueue Interfaces

A *queue* is an ordered collection of elements with methods for extracting elements, in order, from the *head* of the queue. Queue implementations are commonly based on insertion order as in first-in, first-out (FIFO) queues or last in, first-out queues (LIFO queues are also known as stacks). Other orderings are possible, however: a *priority queue* orders its elements according to an external `Comparator` object, or according to the natural ordering of `Comparable` elements. Unlike a `Set`, `Queue` implementations typically allow duplicate elements. Unlike `List`, the `Queue` interface does not define methods for manipulating queue elements at arbitrary positions. Only the element at the head of the queue is available for examination. It is common for `Queue` implementations to have a fixed capacity: when a queue is full, it is not possible to add more elements. Similarly, when a queue is empty, it is not possible to remove any more elements. Because full and empty conditions are a normal part of many queue-based algorithms, the

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`Queue` interface defines methods that signal these conditions with return values rather than by throwing exceptions. Specifically, the `peek()` and `poll( )` methods return `null` to indicate that the queue is empty. For this reason, most `Queue` implementations do not allow `null` elements.

A *blocking queue* is a type of queue that defines blocking `put( )` and `take()` methods. The `put( )` method adds an element to the queue, waiting, if necessary, until there is space in the queue for the element. And the `take( )` method removes an element from the head of the queue, waiting, if necessary, until there is an element to remove. Blocking queues are an important part of many multithreaded algorithms, and the `BlockingQueue` interface (which extends `Queue`) is defined as part of the `java.util.concurrent` package. `Queue`, `BlockingQueue`, and their implementations are new in Java 5.0. See Section 5.7.7 later in this chapter for a list of `BlockingQueue` implementations.

Queues are not nearly as commonly used as sets, lists, and maps, except perhaps in certain multithreaded programming styles. In lieu of example code here, we'll try to clarify the confusing array of queue insertion and removal operations:

- Adding elements to queues

`add()`

This `Collection` method simply adds an element in the normal way. In bounded queues, this method may throw an exception if the queue is full.

`offer()`

This `Queue` method is like `add( )` but returns `false` instead of throwing an exception if the element cannot be added because a bounded queue is full.

`BlockingQueue` defines a timeout version of `offer( )` that waits up to a specified amount of time for space to become available in a full queue. Like the basic version of the method, it returns `true` if the element was inserted and `false` otherwise.

`put( )`

This `BlockingQueue` method blocks: if the element cannot be inserted because the queue is full, `put( )` waits until some other thread removes an element from the queue, and space becomes available for the new element.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

- Removing elements from queues

`remove()`

In addition to the `Collection.remove( )` method, which removes a specified element from the queue, the `Queue` interface defines a no-argument version of `remove( )` that removes and returns the element at the head of the queue. If the queue is empty, this method throws a `NoSuchElementException`.

`poll( )`

This `Queue` method removes and returns the element at the head of the queue, like `remove( )` does but returns `null` if the queue is empty instead of throwing an exception.

`BlockingQueue` defines a timeout version of `poll()` that waits up to a specified amount of time for an element to be added to an empty queue.

`take()`

This `BlockingQueue` method removes and returns the element at the head of the queue. If the queue is empty, it blocks until some other thread adds an element to the queue.

`drainTo( )`

This `BlockingQueue` method removes all available elements from the queue and adds them to a specified `Collection`. It does not block to wait for elements to be added to the queue. A variant of the method accepts a maximum number of elements to drain.

- Querying the element at the head, without removing it from the queue

`element( )`

This `Queue` method returns the element at the head of the queue but does not remove that element from the queue. If the queue is empty, it throws `NoSuchElementException`.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`peek( )`

This `Queue` method is like `element()` but returns `null` if the queue is empty.

The `LinkedList` class has been retrofitted, in Java 5.0, to implement `Queue`. It provides unbounded FIFO (first in, first out) ordering, and insertion and removal operations require constant time. `LinkedList` allows `null` elements, although their use is discouraged when the list is being used as a queue.

The only other `Queue` implementation in the `java.util` package is `PriorityQueue`, which orders its elements according to a `Comparator` or orders `Comparable` elements according to the order defined by their `compareTo( )` methods. The head of a `PriorityQueue` is always the the smallest element according to the defined ordering.

The `java.util.concurrent` package contains a number of `BlockingQueue` implementations; they are described later in the chapter. This package also contains `ConcurrentLinkedQueue`, an efficient threadsafe `Queue` implementation that does not suffer the overhead of `synchronized` methods.

5.6.6. Collection Wrappers

The `java.util.Collections` class is home to quite a few static utility methods designed for use with collections. One important group of these methods are the collection *wrapper* methods: they return a special-purpose collection wrapped around a collection you specify. The purpose of the wrapper collection is to wrap additional functionality around a collection that does not provide it itself. Wrappers exist to provide thread-safety, write-protection and runtime type checking. Wrapper collections are always *backed by* the original collection, which means that the methods of the wrapper simply dispatch to the equivalent methods of the wrapped collection. This means that changes made to the collection through the wrapper are visible through the wrapped collection and vice versa.

The first set of wrapper methods provides threadsafe wrappers around collections. Except for the legacy classes `Vector` and `Hashtable`, the collection implementations in `java.util` do not have `synchronized` methods and are not protected against concurrent access by multiple threads. If you need threadsafe collections, create them with code like this:

```
List<String> list = Collections.synchronizedList(new ArrayList<String>());
Set<Integer> set = Collections.synchronizedSet(new HashSet<Integer>());
Map<String,Integer> map =
    Collections.synchronizedMap(new HashMap<String,Integer>());
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

A second set of wrapper methods provides collection objects through which the underlying collection cannot be modified. They return a read-only view of a collection: any attempt to change the content of the collection results in an `UnsupportedOperationException`. These wrappers are useful when you must pass a collection to a method that must not be allowed to modify or mutate the content of the collection in any way:

```
List<Integer> primes = new ArrayList<Integer>();
List<Integer> readonly = Collections.unmodifiableList(primes);
// We can modify the list through primes
primes.addAll(Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19));
// But we can't modify through the read-only wrapper
readonly.add(23);  // UnsupportedOperationException
```

The final set of wrapper methods provides runtime type checking of any values added to the collection. They were added in Java 5.0 to complement the compile-time type safety provided by generics. These wrappers are helpful when working with legacy code that has not been converted to use generics. If you have a `SortedSet<String>`, for example, and must pass it to a method that expects a `Set`, you can use a checked wrapper to ensure that that method cannot add anything to the set that is not a `String`:

```
SortedSet<String> words = new TreeSet<String>(); // A set
SortedSet<String> checkedWords =                 // A checked set
    Collections.checkedSortedSet(words, String.class);
addWordsFromFile(checkedWords, filename);        // Passed to legacy method
```

5.6.7. Special-Case Collections

In addition to its wrapper methods, the `java.util.Collections` class also defines utility methods for creating immutable collection instances that contain a single element and other methods for creating empty collections. `singleton()` , `singletonList( )`, and `singletonMap()` return immutable `Set` , `List`, and `Map` objects that contain a single specified object or a single key/value pair. These methods are useful, for example, when you need to pass a single object to a method that expects a collection.

The `Collections` class also includes methods that return empty collections. If you are writing a method that returns a collection, it is usually best to handle the no-values-to-return case by returning an empty collection instead of a special-case value like `null`:

```
Set<Integer> si = Collections.emptySet();
List<String> ss = Collections.emptyList();
Map<String,Integer> m = Collections.emptyMap();
```

Finally, `nCopies()` returns an immutable `List` that contains a specified number of copies of a single specified object:

```
List<Integer> tenzeros = Collections.nCopies(10, 0);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.6.8. Converting to and from Arrays

Arrays of objects and collections serve similar purposes. It is possible to convert from one to the other:

```
String[] a ={ "this", "is", "a", "test" };  // An array
List<String> l = Arrays.asList(a);          // View array as an ungrowable list
List<String> m = new ArrayList<String>(l);  // Make a growable copy of the view

// In Java 5.0, asList() is a varargs method so we can do this, too:
Set<Character> abc = new HashSet<Character>(Arrays.asList('a', 'b', 'c'));

// Collection defines the toArray() method.  The no-args version creates
// an Object[] array, copies collection elements to it and returns it
Object[] members = set.toArray();         // Get set elements as an array
Object[] items = list.toArray();          // Get list elements as an array
Object[] keys = map.keySet().toArray();   // Get map key objects as an array
Object[] values = map.values().toArray(); // Get map value objects as an array

// If you want the return value to be something other than Object[], pass
// in an array of the appropriate type.  If the array is not big enough,
// another one of the same type will be allocated.  If the array is too big,
// the collection elements copied to it will be null-terminated
String[] c = l.toArray(new String[0]);
```

5.6.9. Collections Utility Methods

Just as the `java.util.Arrays` class defined methods to operate on arrays, the `java.util.Collections` class defines methods to operate on collections. Most notable are methods to sort and search the elements of collections:

```
Collections.sort(list);
int pos = Collections.binarySearch(list, "key"); // list must be sorted first
```

Here are some other interesting `Collections` methods:

```
Collections.copy(list1, list2); // Copy list2 into list1, overwriting list1
Collections.fill(list, o);      // Fill list with Object o
Collections.max(c);             // Find the largest element in Collection c
Collections.min(c);             // Find the smallest element in Collection c

Collections.reverse(list);      // Reverse list
Collections.shuffle(list);      // Mix up list
```

5.6.10. Implementing Collections

The Java Collections Framework provides abstract classes that make it simple to implement common types of collections. The following code extends `AbstractList` to define a `QuadraticSequence`, a list implementation that computes list values on demand rather than actually storing them in memory anywhere. See also `AbstractSet`, `AbstractMap`, `AbstractQueue`, and `AbstractSequentialList`.

```
import java.util.*;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
/** An immutable List<Double> representing the sequence ax^2 + bx + c */
public class QuadraticSequence extends AbstractList<Double> {
    final int size;
    final double a, b, c;

    QuadraticSequence(double a, double b, double c, int size) {
        this.a = a; this.b = b; this.c = c; this.size = size;
    }

    @Override public int size() { return size; }

    @Override public Double get(int index) {
        if (index<0 || index>=size) throw new ArrayIndexOutOfBoundsException();
        return a*index*index + b*index + c;
    }
}
```

5.7. Threads and Concurrency

The Java platform has supported multithreaded or *concurrent* programming with the `Thread` class and `Runnable` interface since Java 1.0. Java 5.0 bolsters that support with a comprehensive set of new utilities for concurrent programming.

5.7.1. Creating, Running, and Manipulating Threads

Java makes it easy to define and work with multiple threads of execution within a program. `java.lang.Thread` is the fundamental thread class in the Java API. There are two ways to define a thread. One is to subclass `Thread`, override the `run( )` method and then instantiate your `Thread` subclass. The other is to define a class that implements the `Runnable` method (i.e., define a `run()` method) and then pass an instance of this `Runnable` object to the `Thread()` constructor. In either case, the result is a `Thread` object, where the `run()` method is the body of the thread. When you call the `start()` method of the `Thread` object, the interpreter creates a new thread to execute the `run()` method. This new thread continues to run until the `run( )` method exits. Meanwhile, the original thread continues running itself, starting with the statement following the `start()` method. The following code demonstrates:

```
final List list; // Some long unsorted list of objects; initialized elsewhere

/** A Thread class for sorting a List in the background */
class BackgroundSorter extends Thread {
  List l;
  public BackgroundSorter(List l) { this.l = l; }    // Constructor
  public void run() { Collections.sort(l); }         // Thread body
}

// Create a BackgroundSorter thread
Thread sorter = new BackgroundSorter(list);
// Start it running; the new thread runs the run() method above while
// the original thread continues with whatever statement comes next.
sorter.start();

// Here's another way to define a similar thread
Thread t = new Thread(new Runnable() {          // Create a new thread
  public void run() { Collections.sort(list); } // to sort the list of objects.
});
t.start();                                      // Start it running
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.7.1.1. Thread lifecycle

A thread can be in one of six states. In Java 5.0, these states are represented by the `Thread.State` enumerated type, and the state of a thread can be queried with the `getState( )` method. A listing of the `Thread.State` constants provides a good overview of the lifecycle of a thread:

`NEW`

The `Thread` has been created but its `start( )` method has not yet been called. All threads start in this state.

`RUNNABLE`

The thread is running or is available to run when the operating system schedules it.

`BLOCKED`

The thread is not running because it is waiting to acquire a lock so that it can enter a `synchronized` method or block. We'll see more about `synchronized` methods and blocks later in this section.

`WAITING`

The thread is not running because it has called `Object.wait()` or `Thread.join( )`.

`TIMED_WAITING`

The thread is not running because it has called `Thread.sleep()` or has called `Object.wait( )` or `Thread.join()` with a timeout value.

`TERMINATED`

The thread has completed execution. Its `run( )` method has exited normally or by throwing an exception.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.7.1.2. Thread priorities

Threads can run at different priority levels. A thread at a given priority level does not typically run unless no higher-priority threads are waiting to run. Here is some code you can use when working with thread priorities:

```
// Set a thread t to lower-than-normal priority
t.setPriority(Thread.NORM_PRIORITY-1);

// Set a thread to lower priority than the current thread
t.setPriority(Thread.currentThread().getPriority() - 1);

// Threads that don't pause for I/O should explicitly yield the CPU
// to give other threads with the same priority a chance to run.
Thread t = new Thread(new Runnable() {
  public void run() {
    for(int i = 0; i < data.length; i++) {  // Loop through a bunch of data
      process(data[i]);                     // Process it
      if ((i % 10) == 0)                    // But after every 10 iterations,
        Thread.yield();                     // pause to let other threads run.
    }
  }
});
```

5.7.1.3. Handling uncaught exceptions

A thread terminates normally when it reaches the end of its `run( )` method or when it executes a `return` statement in that method. A thread can also terminate by throwing an exception, however. When a thread exits in this way, the default behavior is to print the name of the thread, the type of the exception, the exception message, and a stack trace. In Java 5.0, you can install a custom handler for uncaught exceptions in a thread. For example:

```
// This thread just throws an exception
Thread t = new Thread() {
        public void run() {throw new UnsupportedOperationException();}
    };

// Giving threads a name helps with debugging
t.setName("My Broken Thread");

// Here's a handler for the error.
t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
        public void uncaughtException(Thread t, Throwable e) {
            System.err.printf("Exception in thread %d '%s':" +
                              "%s at line %d of %s%n",
                              t.getId(),    // Thread id
                              t.getName(),  // Thread name
                              e.toString(), // Exception name and message
                              e.getStackTrace()[0].getLineNumber(), // line #
                              e.getStackTrace()[0].getFileName());  // filename
        }
    });
```

5.7.2. Making a Thread Sleep

Often, threads are used to perform some kind of repetitive task at a fixed interval. This is particularly true when doing graphical programming that involves animation or similar effects. The key to doing this is making a thread *sleep*, or stop running, for a specified

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

amount of time. This is done with the static `Thread.sleep( )` method, or, in Java 5.0, with utility methods of enumerated constants of the `TimeUnit` class:

```
import static java.util.concurrent.TimeUnit.SECONDS;  // utility class

public class Clock extends Thread {
    // This field is volatile because two different threads may access it
    volatile boolean keepRunning = true;

    public Clock() {     // The constructor
        setDaemon(true); // Daemon thread: interpreter can exit while it runs
    }

    public void run() {        // The body of the thread
        while(keepRunning) {   // This thread runs until asked to stop
            long now = System.currentTimeMillis();    // Get current time
            System.out.printf("%tr%n", now);          // Print it out
            try { Thread.sleep(1000); }               // Wait 1000 milliseconds
            catch (InterruptedException e) { return; }// Quit on interrupt
        }
    }

    // Ask the thread to stop running.  An alternative to interrupt().
    public void pleaseStop() { keepRunning = false; }

    // This method demonstrates how to use the Clock class
    public static void main(String[] args) {
        Clock c = new Clock();                 // Create a Clock thread
        c.start();                             // Start it
        try {  SECONDS.sleep(10); }            // Wait 10 seconds
        catch(InterruptedException ignore) {}  // Ignore interrupts
        // Now stop the clock thread.  We could also use c.interrupt()
        c.pleaseStop();
    }
}
```

Notice the `pleaseStop()` method in this example: it is designed to stop the clock thread in a controlled way. The example is coded so that it can also be stopped by calling the `interrupt()` method it inherits from `Thread`. The `Thread` class defines a `stop()` method, but it is deprecated.

5.7.3. Running and Scheduling Tasks

Java provides a number of ways to run tasks asynchronously or to schedule them for future execution without having to explicitly create `Thread` objects. The following sections illustrate the `Timer` class added in Java 1.3 and the executors framework of the Java 5.0 `java.util.concurrent` package.

5.7.3.1. Scheduling tasks with Timer

Added in Java 1.3, the `java.util.Timer` and `java.util.TimerTask` classes make it easy to run repetitive tasks. Here is some code that behaves much like the `Clock` class shown earlier:

```
import java.util.*;

// Define the time-display task
TimerTask displayTime = new TimerTask() {
  public void run() { System.out.printf("%tr%n", System.currentTimeMillis()); }
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
};
// Create a timer object to run the task (and possibly others)
Timer timer = new Timer();
// Now schedule that task to be run every 1,000 milliseconds, starting now
timer.schedule(displayTime, 0, 1000);

// To stop the time-display task
displayTime.cancel();
```

5.7.3.2. The Executor interface

In Java 5.0, the `java.util.concurrent` package includes the `Executor` interface. An `Executor` is an object that can execute a `Runnable` object. A user of an `Executor` often does not need to be aware of just how the `Executor` accomplishes this: it just needs to know that the `Runnable` will, at some point, run. `Executor` implementations can be created to use a number of different threading strategies, as the following code makes clear. (Note that this example also demonstrates the use of a `BlockingQueue`.)

```
import java.util.concurrent.*;

/** Execute a Runnable in the current thread. */
class CurrentThreadExecutor implements Executor {
    public void execute(Runnable r) { r.run(); }
}

/** Execute each Runnable using a newly created thread */
class NewThreadExecutor implements Executor {
    public void execute(Runnable r) { new Thread(r).start(); }
}

/**
 * Queue up the Runnables and execute them in order using a single thread
 * created for that purpose.
 */
class SingleThreadExecutor extends Thread implements Executor {
    BlockingQueue<Runnable> q = new LinkedBlockingQueue<Runnable>();

    public void execute(Runnable r) {
        // Don't execute the Runnable here; just put it on the queue.
        // Our queue is effectively unbounded, so this should never block.
        // Since it never blocks, it should never throw InterruptedException.
        try { q.put(r); }
        catch(InterruptedException never) { throw new AssertionError(never); }
    }

    // This is the body of the thread that actually executes the Runnables
    public void run() {
        for(;;) {                   // Loop forever
            try {
                Runnable r = q.take();  // Get next Runnable, or wait
                r.run();                // Run it!
            }
            catch(InterruptedException e) {
                // If interrupted, stop executing queued Runnables.
                return;
            }
        }
    }
}
```

These sample implementations help demonstrate how an `Executor` works and how it separates the notion of executing a task from the scheduling policy and threading details

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

of the implementation. It is rarely necessary to actually implement your own `Executor`, however, since `java.util.concurrent` provides the flexible and powerful `ThreadPoolExecutor` class. This class is typically used via one of the static factory methods in the `Executors` class:

```
Executor oneThread = Executors.newSingleThreadExecutor(); // pool size of 1
Executor fixedPool = Executors.newFixedThreadPool(10); // 10 threads in pool
Executor unboundedPool = Executors.newCachedThreadPool(); // as many as needed
```

In addition to these convenient factory methods, you can also explicitly create a `ThreadPoolExecutor` if you want to specify a minimum and maximum size for the thread pool or want to specify the queue type (bounded, unbounded, priority-sorted, or synchronized, for example) to use for tasks that cannot immediately be run by a thread.

5.7.3.3. ExecutorService

If you've looked at the signature for `ThreadPoolExecutor` or for the `Executors` factory methods cited above, you'll see that it is an `ExecutorService` . The `ExecutorService` interface extends `Executor` and adds the ability to execute `Callable` objects. `Callable` is something like a `Runnable`. Instead of encapsulating arbitrary code in a `run()` method, however, a `Callable` puts that code in a `call()` method. `call( )` differs from `run()` in two important ways: it returns a result, and it is allowed to throw exceptions.

Because `call()` returns a result, the `Callable` interface takes the result type as a parameter. A time-consuming chunk of code that computes a large prime number, for example, could be wrapped in a `Callable<BigInteger>`:

```
import java.util.concurrent.*;
import java.math.BigInteger;
import java.util.Random;
import java.security.SecureRandom;

/** This is a Callable implementation for computing big primes. */
public class RandomPrimeSearch implements Callable<BigInteger> {
    static Random prng = new SecureRandom();  // self-seeding
    int n;
    public RandomPrimeSearch(int bitsize) { n = bitsize; }
    public BigInteger call() { return BigInteger.probablePrime(n, prng); }
}
```

You can invoke the `call( )` method of any `Callable` object directly, of course, but to execute it using an `ExecutorService`, you pass it to the `submit()` method. Because `ExecutorService` implementations typically run tasks asynchronously, the `submit()` method cannot simply return the result of the `call( )` method. Instead, `submit()` returns a `Future` object. A `Future` is simply the promise of a result sometime in the future. It is parameterized with the type of the result, as shown in this code snippet:

```
// Try to compute two primes at the same time
ExecutorService threadpool = Executors.newFixedThreadPool(2);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
Future<BigInteger> p = threadpool.submit(new RandomPrimeSearch(512));
Future<BigInteger> q = threadpool.submit(new RandomPrimeSearch(512));
```

Once you have a `Future` object, what can you do with it? You can call `isDone()` to see if the `Callable` has finished running. You can call `cancel()` to cancel execution of the `Callable` and can call `isCancelled()` to see if the `Callable` was canceled before it completed. But most of the time, you simply call `get( )` to get the result of the `call( )` method. `get()` blocks, if necessary, to wait for the `call( )` method to complete. Here is code you might use with the `Future` objects shown above:

```
BigInteger product = p.get().multiply(q.get());
```

Note that the `get()` method may throw an `ExecutionException`. Recall that `Callable.call( )` can throw any kind of exception. If this happens, the `Future` wraps that exception in an `ExecutionException` and throws it from `get()`. Note that the `Future.isDone()` method considers a `Callable` to be "done," even if the `call( )` method terminated abnormally with an exception.

5.7.3.4. ScheduledExecutorService

`ScheduledExecutorService` is an extension of `ExecutorService` that adds `Timer`-like scheduling capabilities. It allows you to schedule a `Runnable` or `Callable` to be executed once after a specified time delay or to schedule a `Runnable` for repeated execution. In each case, the result of scheduling a task for future execution is a `ScheduledFuture` object. This is simply a `Future` that also implements the `Delay` interface and provides a `getDelay( )` method that can be used to query the remaining time before execution of the task.

The easiest way to obtain a `ScheduledExecutorService` is with factory methods of the `Executors` class. The following code uses a `ScheduledExecutorService` to repeatedly perform an action and also to cancel the repeated action after a fixed interval.

```
/**
 * Print random ASCII characters at a rate of cps characters per second
 * for a total of totalSeconds seconds.
 */
public static void spew(int cps, int totalSeconds) {
    final Random rng = new Random(System.currentTimeMillis());
    final ScheduledExecutorService executor =
        Executors.newSingleThreadScheduledExecutor();
    final ScheduledFuture<?> spewer =
        executor.scheduleAtFixedRate(new Runnable() {
                public void run() {
                    System.out.print((char)(rng.nextInt('~' - ' ') + ' '));
                    System.out.flush();
                }
            },
                                     0, 1000000/cps, TimeUnit.MICROSECONDS);
    executor.schedule(new Runnable() {
            public void run() {
                spewer.cancel(false);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
                executor.shutdown();
                System.out.println();
            }
        },
                    totalSeconds, TimeUnit.SECONDS);
}
```

5.7.4. Exclusion and Locks

When using multiple threads, you must be very careful if you allow more than one thread to access the same data structure. Consider what would happen if one thread was trying to loop through the elements of a `List` while another thread was sorting those elements. Preventing this kind of unwanted concurrency is one of the central problems of multithreaded computing. The basic technique for preventing two threads from accessing the same object at the same time is to require a thread to obtain a lock on the object before the thread can modify it. While any one thread holds the lock, another thread that requests the lock has to wait until the first thread is done and releases the lock. Every Java object has the fundamental ability to provide such a locking capability.

The easiest way to keep objects threadsafe is to declare all sensitive methods `synchronized`. A thread must obtain a lock on an object before it can execute any of its `synchronized` methods, which means that no other thread can execute any other `synchronized` method at the same time. (If a `static` method is declared `synchronized`, the thread must obtain a lock on the class, and this works in the same manner.) To do finer-grained locking, you can specify `synchronized` blocks of code that hold a lock on a specified object for a short time:

```
// This method swaps two array elements in a synchronized block
public static void swap(Object[] array, int index1, int index2) {
  synchronized(array) {
    Object tmp = array[index1];
    array[index1] = array[index2];
    array[index2] = tmp;
  }
}

// The Collection, Set, List, and Map implementations in java.util do
// not have synchronized methods (except for the legacy implementations
// Vector and Hashtable). When working with multiple threads, you can
// obtain synchronized wrapper objects.
List synclist = Collections.synchronizedList(list);
Map syncmap = Collections.synchronizedMap(map);
```

5.7.4.1. The java.util.concurrent.locks package

Note that when you use the `synchronized` modifier or statement, the lock you acquire is block-scoped, and is automatically released when the thread exits the method or block. The `java.util.concurrent.locks` package in Java 5.0 provides an alternative: a `Lock` object that you explicitly lock and unlock. `Lock` objects are not automatically block-scoped and you must be careful to use `try/finally` constructs to ensure that locks are

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

always released. On the other hand, `Lock` enables algorithms that are simply not possible with block-scoped locks, such as the following "hand-over-hand" linked list traversal:

```
import java.util.concurrent.locks.*;  // New in Java 5.0

/**
 * A partial implementation of a linked list of values of type E.
 * It demonstrates hand-over-hand locking with Lock
 */
public class LinkList<E> {
    E value;                // The value of this node of the list
    LinkList<E> rest;       // The rest of the list
    Lock lock;              // A lock for this node

    public LinkList(E value) {  // Constructor for a list
        this.value = value;          // Node value
        rest = null;                 // This is the only node in the list
        lock = new ReentrantLock();  // We can lock this node
    }

    /**
     * Append a node to the end of the list, traversing the list using
     * hand-over-hand locking. This method is threadsafe: multiple threads
     * may traverse different portions of the list at the same time.
     **/
    public void append(E value) {
        LinkList<E> node = this;  // Start at this node
        node.lock.lock();         // Lock it.

        // Loop 'till we find the last node in the list
        while(node.rest != null) {
            LinkList<E> next = node.rest;

            // This is the hand-over-hand part.  Lock the next node and then
            // unlock the current node.  We use a try/finally construct so
            // that the current node is unlocked even if the lock on the
            // next node fails with an exception.
            try { next.lock.lock(); }  // lock the next node
            finally { node.lock.unlock(); } // unlock the current node
            node = next;
        }

        // At this point, node is the final node in the list, and we have
        // a lock on it.  Use a try/finally to ensure that we unlock it.
        try {
            node.rest = new LinkList<E>(value); // Append new node
        }
        finally { node.lock.unlock(); }
    }
}
```

5.7.4.2. Deadlock

When you are using locking to prevent threads from accessing the same data at the same time, you must be careful to avoid *deadlock*, which occurs when two threads end up waiting for each other to release a lock they need. Since neither can proceed, neither one can release the lock it holds, and they both stop running. The following code is prone to deadlock. Whether or not a deadlock actually occurs may vary from system to system and from execution to execution.

```
// When two threads try to lock two objects, deadlock can occur unless
// they always request the locks in the same order.
final Object resource1 = new Object();   // Here are two objects to lock
final Object resource2 = new Object();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
Thread t1 = new Thread(new Runnable() {  // Locks resource1 then resource2
  public void run() {
    synchronized(resource1) {
      synchronized(resource2) { compute(); }
    }
  }
});

Thread t2 = new Thread(new Runnable() {  // Locks resource2 then resource1
  public void run() {
    synchronized(resource2) {
      synchronized(resource1) { compute(); }
    }
  }
});

t1.start();  // Locks resource1
t2.start();  // Locks resource2 and now neither
thread can progress!
```

5.7.5. Coordinating Threads

It is common in multithreaded programming to require one thread to wait for another thread to take some action. The Java platform provides a number of ways to coordinate threads, including methods built into the `Object` and `Thread` classes, as well as "synchronizer" utility classes introduced in Java 5.0.

5.7.5.1. wait( ) and notify()

Sometimes a thread needs to stop running and wait until some kind of event occurs, at which point it is told to continue running. This is done with the `wait()` and `notify( )` methods. These aren't methods of the `Thread` class, however; they are methods of `Object`. Just as every Java object has a lock associated with it, every object can maintain a list of waiting threads. When a thread calls the `wait()` method of an object, any locks the thread holds are temporarily released, and the thread is added to the list of waiting threads for that object and stops running. When another thread calls the `notifyAll( )` method of the same object, the object wakes up the waiting threads and allows them to continue running:

```
import java.util.*;

/**
 * A queue. One thread calls push() to put an object on the queue.
 * Another calls pop() to get an object off the queue. If there is no
 * data, pop() waits until there is some, using wait()/notify().
 * wait() and notify() must be used within a synchronized method or
 * block. In Java 5.0, use a java.util.concurrent.BlockingQueue instead.
 */
public class WaitingQueue<E> {
    LinkedList<E> q = new LinkedList<E>();  // Where objects are stored
    public synchronized void push(E o) {
        q.add(o);         // Append the object to the end of the list
        this.notifyAll(); // Tell waiting threads that data is ready
    }
    public synchronized E pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException ignore) {}
        }
        return q.remove(0);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    }
}
```

Note that such a class is not necessary in Java 5.0 because `java.util.concurrent` defines the `BlockingQueue` interface and general-purpose implementations such as `ArrayBlockingQueue`.

5.7.5.2. Waiting on a Condition

Java 5.0 provides an alternative to the `wait()` and `notifyAll( )` methods of `Object`. `java.util.concurrent.locks` defines a `Condition` object with `await()` and `signalAll()` methods. `Condition` objects are always associated with `Lock` objects and are used in much the same way as the locking and waiting capability built into each Java object. The primary benefit is that it is possible to have more than one `Condition` for each `Lock`, something that is not possible with `Object`-based locking and waiting.

5.7.5.3. Waiting for a thread to finish

Sometimes one thread needs to stop and wait for another thread to complete. You can accomplish this with the `join( )` method:

```
List list;  // A long list of objects to be sorted; initialized elsewhere

// Define a thread to sort the list: lower its priority, so it runs only
// when the current thread is waiting for I/O and then start it running.
Thread sorter = new BackgroundSorter(list);               // Defined earlier
sorter.setPriority(Thread.currentThread.getPriority()-1); // Lower priority
sorter.start();                                           // Start sorting

// Meanwhile, in this original thread, read data from a file
byte[] data = readData();  // Method defined elsewhere

// Before we can proceed, we need the list to be fully sorted, so
// we must wait for the sorter thread to exit, if it hasn't already.
try { sorter.join(); } catch(InterruptedException e) {}
```

5.7.5.4. Synchronizer utilities

`java.util.concurrent` includes four " synchronizer" classes that help to synchronize the state of a concurrent program by making threads wait until certain conditions hold:

`Semaphore`

> The `Semaphore` class models semaphores, a traditional concurrent programming construct. Conceptually, a semaphore represents one or more "permits." A thread that needs a permit calls `acquire( )` and then calls `release()` when done with it. `acquire()` blocks if no permits are available, suspending the thread until another thread releases a permit.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`CountDownLatch`

A *latch* is conceptually any variable or concurrency construct that has two possible states and transitions from its initial state to its final state only once. Once the transition occurs, it remains in that final state forever. `CountDownLatch` is a concurrency utility that can exist in two states, closed and open. In its initial closed state, any threads that call the `await( )` method block and cannot proceed until it transitions to its latched open state. Once this transition occurs, all waiting threads proceed, and any threads that call `await()` in the future will not block at all. The transition from closed to open occurs when a specified number of calls to `countDown()` have occurred.

`Exchanger`

An `Exchanger` is a utility that allows two threads to *rendezvous* and exchange values. The first thread to call the `exchange( )` method blocks until a second thread calls the same method. When this happens, the argument passed to the `exchange()` method by the first thread becomes the return value of the method for the second thread and vice-versa. When the two `exchange()` invocations return, both threads are free to continue running concurrently. `Exchanger` is a generic type and uses its type parameter to specify the type of values to be exchanged.

`CyclicBarrier`

A `CyclicBarrier` is a utility that enables a group of N threads to wait for each other to reach a synchronization point. The number of threads is specified when the `CyclicBarrier` is first created. Threads call the `await( )` method to block until the last thread calls `await( )`, at which point all threads resume again. Unlike a `CountDownLatch`, a `CyclicBarrier` resets its count and is ready for immediate reuse. `CyclicBarrier` is useful in parallel algorithms in which a computation is decomposed into parts, and each part is handled by a separate thread. In such algorithms, the threads must typically rendezvous so that their partial solutions can be merged into a complete solution. To facilitate this, the `CyclicBarrier` constructor allows you to specify a `Runnable` object to be executed by the last thread that calls `await( )` before any of the other threads are woken up and allowed to resume. This `Runnable` can provide the coordination required to assemble a solution from the threads computations or to assign a new computation to each of the threads.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.7.6. Thread Interruption

In the examples illustrating the `sleep( )`, `join()`, and `wait( )` methods, you may have noticed that calls to each of these methods are wrapped in a `try` statement that catches an `InterruptedException`. This is necessary because the `interrupt()` method allows one thread to interrupt the execution of another. The outcome of an interrupt depends on how you handle the `InterruptedException`. The response that is usually preferred is for an interrupted thread to stop running. On the other hand, if you simply catch and ignore the `InterruptedException`, an interrupt simply stops a thread from blocking.

If the `interrupt( )` method is called on a thread that is not blocked, the thread continues running, but its "interrupt status" is set to indicate that an interrupt has been requested. A thread can test its own interrupt status by calling the static `Thread.interrupted()` method, which returns `true` if the thread has been interrupted and, as a side effect, clears the interrupt status. One thread can test the interrupt status of another thread with the instance method `isInterrupted( )`, which queries the status but does not clear it.

If a thread calls `sleep( )`, `join()`, or `wait()` while its interrupt status is set, it does not block but immediately throws an `InterruptedException` (the interrupt status is cleared as a side effect of throwing the exception). Similarly, if the `interrupt( )` method is called on a thread that is already blocked in a call to `sleep( )`, `join()`, or `wait()`, that thread stops blocking by throwing an `InterruptedException`.

One of the most common times that threads block is while doing input/output; a thread often has to pause and wait for data to become available from the filesystem or from the network. (The `java.io`, `java.net`, and `java.nio` APIs for performing I/O operations are discussed later in this chapter.) Unfortunately, the `interrupt()` method does not wake up a thread blocked in an I/O method of the `java.io` package. This is one of the shortcomings of `java.io` that is cured by the New I/O API in `java.nio`. If a thread is interrupted while blocked in an I/O operation on any channel that implements `java.nio.channels.InterruptibleChannel`, the channel is closed, the thread's interrupt status is set, and the thread wakes up by throwing a `java.nio.channels.ClosedByInterruptException`. The same thing happens if a thread tries to call a blocking I/O method while its interrupt status is set. Similarly, if a thread is interrupted while it is blocked in the `select( )` method of a `java.nio.channels.Selector` (or if it calls `select( )` while its interrupt status is set), `select( )` will stop blocking (or will never start) and will return immediately. No exception is thrown in this case; the interrupted thread simply wakes up, and the `select()` call returns.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.7.7. Blocking Queues

As noted in Section 5.6.5 earlier in this chapter, a *queue* is a collection in which elements are inserted at the "tail" and removed at the "head." The `Queue` interface and various implementations were added to `java.util` as part of Java 5.0. `java.util.concurrent` extends the `Queue` interface: `BlockingQueue` defines `put()` and `take( )` methods that allow you to add and remove elements of the queue, blocking if necessary until the queue has room, or until there is an element to be removed. The use of blocking queues is a common pattern in multithreaded programming: one thread produces objects and places them on a queue for consumption by another thread which removes them from the queue.

`java.util.concurrent` provides five implementations of `BlockingQueue`:

`ArrayBlockingQueue`

: This implementation is based on an array, and, like all arrays, has a fixed capacity established when it is created. At the cost of reduced throughput, this queue can operate in a "fair" mode in which threads blocking to `put()` or `take( )` an element are served in the order in which they arrived.

`LinkedBlockingQueue`

: This implementation is based on a linked-list data structure. It may have a maximum size specified, but, by default, it is essentially unbounded.

`PriorityBlockingQueue`

: This unbounded queue does not implement FIFO (first-in, first-out) ordering. Instead, it orders its elements based on a specified `Comparator` object, or based on their natural ordering if they are `Comparable` objects and no `Comparator` is specified. The element returned by `take()` is the smallest element according to the `Comparator` or `Comparable` ordering. See also `java.util.PriorityQueue` for a nonblocking version.

`DelayQueue`

: A `DelayQueue` is like a `PriorityBlockingQueue` for elements that implement the `Delayed` interface. `Delayed` is `Comparable` and orders elements by how long they are delayed. But `DelayQueue` is more than just an unbounded queue that sorts its

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

elements. It also restricts `take( )` and related methods so that elements cannot be removed from the queue until their delay has elapsed.

`SynchronousQueue`

This class implements the degenerate case of a `BlockingQueue` with a capacity of zero. A call to `put()` blocks until some other thread calls `take( )`, and a call to `take( )` blocks until some other thread calls `put()`.

5.7.8. Atomic Variables

The `java.util.concurrent.atomic` package contains utility classes that permit *atomic* operations on fields without locking. An atomic operation is one that is indivisible: no other thread can observe an atomic variable in the middle of an atomic operation on it. These utility classes define `get()` and `set( )` accessor methods that have the properties of `volatile` fields but also define compound operations such as compare-and-set and get-and-increment that behave atomically. The code below demonstrates the use of `AtomicInteger` and contrasts it with the use of a traditional `synchronized` method:

```
// The count1(), count2() and count3() methods are all threadsafe.  Two
// threads can call these methods at the same time, and they will never
// see the same return value.
public class Counters {
    // A counter using a synchronized method and locking
    int count1 = 0;
    public synchronized int count1() { return count1++; }

    // A counter using an atomic increment on an AtomicInteger
    AtomicInteger count2 = new AtomicInteger(0);
    public int count2() { return count2.getAndIncrement(); }

    // An optimistic counter using compareAndSet()
    AtomicInteger count3 = new AtomicInteger(0);
    public int count3() {
        // Get the counter value with get() and set it with compareAndSet().
        // If compareAndSet() returns false, try again until we get
        // through the loop without interference.
        int result;
        do {
            result = count3.get();
        } while(!count3.compareAndSet(result, result+1));
        return result;
    }
}
```

5.8. Files and Directories

The `java.io.File` class represents a file or a directory and defines a number of important methods for manipulating files and directories. Note, however, that none of these methods allow you to read the contents of a file; that is the job of

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`java.io.FileInputStream`, which is just one of the many types of I/O streams used in Java and discussed in the next section. Here are some things you can do with `File`:

```
import java.io.*;
import java.util.*;

// Get the name of the user's home directory and represent it with a File
File homedir = new File(System.getProperty("user.home"));
// Create a File object to represent a file in that directory
File f = new File(homedir, ".configfile");

// Find out how big a file is and when it was last modified
long filelength = f.length();
Date lastModified = new java.util.Date(f.lastModified());

// If the file exists, is not a directory, and is readable,
// move it into a newly created directory.
if (f.exists() && f.isFile() && f.canRead()) {       // Check config file
  File configdir = new File(homedir, ".configdir"); // A new config directory
  configdir.mkdir();                                // Create that directory
  f.renameTo(new File(configdir, ".config"));       // Move the file into it
}

// List all files in the home directory
String[] allfiles = homedir.list();

// List all files that have a ".java" suffix
String[] sourcecode = homedir.list(new FilenameFilter() {
  public boolean accept(File d, String name) { return name.endsWith(".java"); }
});
```

The `File` class gained some important additional functionality as of Java 1.2:

```
// List all filesystem root directories; on Windows, this gives us
// File objects for all drive letters (Java 1.2 and later).
File[] rootdirs = File.listRoots();

// Atomically, create a lock file, then delete it (Java 1.2 and later)
File lock = new File(configdir, ".lock");
if (lock.createNewFile()) {
  // We successfully created the file.  Now arrange to delete it on exit
  lock.deleteOnExit();

  // Now run the application secure in the knowledge that no one else
  // is running it at the same time
  ...
}
else {
  // We didn't create the file; someone else has a lock
  System.err.println("Can't create lock file; exiting.");
  System.exit(1);
}

// Create a temporary file to use during processing (Java 1.2 and later)
File temp = File.createTempFile("app", ".tmp");  // Filename prefix and suffix
// Do something with the temp file
  ...
// And delete it when we're done
temp.delete();
```

5.8.1. RandomAccessFile

The `java.io` package also defines a `RandomAccessFile` class that allows you to read binary data from arbitrary locations in a file. This can be useful in certain situations, but

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

most applications read files sequentially, using the stream classes described in the next section. Here is a short example of using `RandomAccessFile`:

```
// Open a file for read/write ("rw") access
File datafile = new File(configdir, "datafile");
RandomAccessFile f = new RandomAccessFile(datafile, "rw");
f.seek(100);                 // Move to byte 100 of the file
byte[] data = new byte[100];   // Create a buffer to hold data
f.read(data);                // Read 100 bytes from the file
int i = f.readInt();         // Read a 4-byte integer from the file
f.seek(100);                 // Move back to byte 100
f.writeInt(i);               // Write the integer first
f.write(data);               // Then write the 100 bytes
f.close();                   // Close file when done with it
```

5.9. Input/Output with java.io

The `java.io` package defines a large number of classes for reading and writing streaming, or sequential, data. The `InputStream` and `OutputStream` classes are for reading and writing streams of bytes while the `Reader` and `Writer` classes are for reading and writing streams of characters. Streams can be nested, meaning you might read characters from a `FilterReader` object that reads and processes characters from an underlying `Reader` stream. This underlying `Reader` stream might read bytes from an `InputStream` and convert them to characters.

5.9.1. Reading Console Input

You can perform a number of common operations with streams. One is to read lines of input the user types at the console:

```
import java.io.*;

BufferedReader console = new BufferedReader(new InputStreamReader(System.in));
System.out.print("What is your name: ");
String name = null;
try {
  name = console.readLine();
}
catch (IOException e) { name = "<" + e + ">"; }  // This should never happen
System.out.println("Hello " + name);
```

5.9.2. Reading Lines from a Text File

Reading lines of text from a file is a similar operation. The following code reads an entire text file and quits when it reaches the end:

```
String filename = System.getProperty("user.home") + File.separator + ".cshrc";
try {
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String line;
  while((line = in.readLine()) != null) {  // Read line, check for end-of-file
    System.out.println(line);              // Print the line
  }
  in.close();    // Always close a stream when you are done with it
}
catch (IOException e) {
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    // Handle FileNotFoundException, etc. here
  }
```

5.9.3. Writing Text to a File

Throughout this book, you've seen the use of the `System.out.println()` method to display text on the console. `System.out` simply refers to an output stream. You can print text to any output stream using similar techniques. The following code shows how to output text to a file:

```
try {
  File f = new File(homedir, ".config");
  PrintWriter out = new PrintWriter(new FileWriter(f));
  out.println("## Automatically generated config file. DO NOT EDIT!");
  out.close();  // We're done writing
}
catch (IOException e) { /* Handle exceptions */ }
```

5.9.4. Reading a Binary File

Not all files contain text, however. The following lines of code treat a file as a stream of bytes and read the bytes into a large array:

```
try {
  File f;                              // File to read; initialized elsewhere
  int filesize = (int) f.length();     // Figure out the file size
  byte[] data = new byte[filesize];    // Create an array that is big enough
  // Create a stream to read the file
  DataInputStream in = new DataInputStream(new FileInputStream(f));
  in.readFully(data);  // Read file contents into array
  in.close();
}
catch (IOException e) { /* Handle exceptions */ }
```

5.9.5. Compressing Data

Various other packages of the Java platform define specialized stream classes that operate on streaming data in some useful way. The following code shows how to use stream classes from `java.util.zip` to compute a checksum of data and then compress the data while writing it to a file:

```
import java.io.*;
import java.util.zip.*;

try {
  File f;                          // File to write to; initialized elsewhere
  byte[] data;                     // Data to write; initialized elsewhere
  Checksum check = new Adler32(); // An object to compute a simple checksum

  // Create a stream that writes bytes to the file f
  FileOutputStream fos = new FileOutputStream(f);
  // Create a stream that compresses bytes and writes them to fos
  GZIPOutputStream gzos = new GZIPOutputStream(fos);
  // Create a stream that computes a checksum on the bytes it writes to gzos
  CheckedOutputStream cos = new CheckedOutputStream(gzos, check);

  cos.write(data);              // Now write the data to the nested streams
  cos.close();                  // Close down the nested chain of streams
  long sum = check.getValue(); // Obtain the computed checksum
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
}
catch (IOException e) { /* Handle exceptions */ }
```

5.9.6. Reading ZIP Files

The `java.util.zip` package also contains a `ZipFile` class that gives you random access to the entries of a ZIP archive and allows you to read those entries through a stream:

```
import java.io.*;
import java.util.zip.*;

String filename;   // File to read; initialized elsewhere
String entryname;  // Entry to read from the ZIP file; initialized elsewhere
ZipFile zipfile = new ZipFile(filename);        // Open the ZIP file
ZipEntry entry = zipfile.getEntry(entryname);   // Get one entry
InputStream in = zipfile.getInputStream(entry); // A stream to read the entry
BufferedInputStream bis = new BufferedInputStream(in); // Improves efficiency
// Now read bytes from bis...
// Print out contents of the ZIP file
for(java.util.Enumeration e = zipfile.entries(); e.hasMoreElements();) {
  ZipEntry zipentry = (ZipEntry) e.nextElement();
  System.out.println(zipentry.getName());
}
```

5.9.7. Computing Message Digests

If you need to compute a cryptographic-strength checksum (also known as a message digest), use one of the stream classes of the `java.security` package. For example:

```
import java.io.*;
import java.security.*;
import java.util.*;


File f;          // File to read and compute digest on; initialized elsewhere
List text = new ArrayList();  // We'll store the lines of text here

// Get an object that can compute an SHA message digest
MessageDigest digester = MessageDigest.getInstance("SHA");
// A stream to read bytes from the file f
FileInputStream fis = new FileInputStream(f);
// A stream that reads bytes from fis and computes an SHA message digest
DigestInputStream dis = new DigestInputStream(fis, digester);
// A stream that reads bytes from dis and converts them to characters
InputStreamReader isr = new InputStreamReader(dis);
// A stream that can read a line at a time
BufferedReader br = new BufferedReader(isr);
// Now read lines from the stream
for(String line; (line = br.readLine()) != null; text.add(line)) ;
// Close the streams
br.close();
// Get the message digest
byte[] digest = digester.digest();
```

5.9.8. Streaming Data to and from Arrays

So far, we've used a variety of stream classes to manipulate streaming data, but the data itself ultimately comes from a file or is written to the console. The `java.io` package defines other stream classes that can read data from and write data to arrays of bytes or strings of text:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
import java.io.*;

// Set up a stream that uses a byte array as its destination
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(baos);
out.writeUTF("hello");          // Write some string data out as bytes
out.writeDouble(Math.PI);       // Write a floating-point value out as bytes
byte[] data = baos.toByteArray(); // Get the array of bytes we've written
out.close();                      // Close the streams

// Set up a stream to read characters from a string
Reader in = new StringReader("Now is the time!");
// Read characters from it until we reach the end
int c;
while((c = in.read()) != -1) System.out.print((char) c);
```

Other classes that operate this way include `ByteArrayInputStream`, `StringWriter`, `CharArrayReader`, and `CharArrayWriter`.

5.9.9. Thread Communication with Pipes

`PipedInputStream` and `PipedOutputStream` and their character-based counterparts, `PipedReader` and `PipedWriter`, are another interesting set of streams defined by `java.io`. These streams are used in pairs by two threads that want to communicate. One thread writes bytes to a `PipedOutputStream` or characters to a `PipedWriter`, and another thread reads bytes or characters from the corresponding `PipedInputStream` or `PipedReader`:

```
// A pair of connected piped I/O streams forms a pipe. One thread writes
// bytes to the PipedOutputStream, and another thread reads them from the
// corresponding PipedInputStream. Or use PipedWriter/PipedReader for chars.
final PipedOutputStream writeEndOfPipe = new PipedOutputStream();
final PipedInputStream readEndOfPipe = new PipedInputStream(writeEndOfPipe);

// This thread reads bytes from the pipe and discards them
Thread devnull = new Thread(new Runnable() {
  public void run() {
    try { while(readEndOfPipe.read() != -1); }
    catch (IOException e) {}  // ignore it
  }
});
devnull.start();
```

5.10. Networking with java.net

The `java.net` package defines a number of classes that make writing networked applications surprisingly easy. Various examples follow.

5.10.1. Networking with the URL Class

The easiest networking class to use is `URL`, which represents a uniform resource locator. Different Java implementations may support different sets of URL protocols, but, at a minimum, you can rely on support for the `http://`, `ftp://`, and `file://` protocols. As

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

of Java 1.4, secure HTTP is also supported with the `https://` protocol. Here are some ways you can use the `URL` class:

```
import java.net.*;
import java.io.*;

// Create some URL objects
URL url=null, url2=null, url3=null;
try {
  url = new URL("http://www.oreilly.com");        // An absolute URL
  url2 = new URL(url, "catalog/books/javanut4/"); // A relative URL
  url3 = new URL("http:", "www.oreilly.com", "index.html");
} catch (MalformedURLException e) { /* Ignore this exception */ }

// Read the content of a URL from an input stream
InputStream in = url.openStream();

// For more control over the reading process, get a URLConnection object
URLConnection conn = url.openConnection();

// Now get some information about the URL
String type = conn.getContentType();
String encoding = conn.getContentEncoding();
java.util.Date lastModified = new java.util.Date(conn.getLastModified());
int len = conn.getContentLength();

// If necessary, read the contents of the URL using this stream
InputStream in = conn.getInputStream();
```

5.10.2. Working with Sockets

Sometimes you need more control over your networked application than is possible with the `URL` class. In this case, you can use a `Socket` to communicate directly with a server. For example:

```
import java.net.*;
import java.io.*;

// Here's a simple client program that connects to a web server,
// requests a document and reads the document from the server.
String hostname = "java.oreilly.com";  // The server to connect to
int port = 80;                         // Standard port for HTTP
String filename = "/index.html";       // The file to read from the server
Socket s = new Socket(hostname, port); // Connect to the server

// Get I/O streams we can use to talk to the server
InputStream sin = s.getInputStream();
BufferedReader fromServer = new BufferedReader(new InputStreamReader(sin));
OutputStream sout = s.getOutputStream();
PrintWriter toServer = new PrintWriter(new OutputStreamWriter(sout));

// Request the file from the server, using the HTTP protocol
toServer.print("GET " + filename + " HTTP/1.0\r\n\r\n");
toServer.flush();

// Now read the server's response, assume it is a text file, and print it out
for(String l = null; (l = fromServer.readLine()) != null; )
  System.out.println(l);

// Close everything down when we're done
toServer.close();
fromServer.close();
s.close();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.10.3. Secure Sockets with SSL

In Java 1.4, the Java Secure Socket Extension, or JSSE, was added to the core Java platform in the packages `javax.net` and `javax.net.ssl`.[1] This API enables encrypted network communication over sockets that use the SSL (Secure Sockets Layer, also known as TLS) protocol. SSL is widely used on the Internet: it is the basis for secure web communication using the `https://` protocol. In Java 1.4 and later, you can use `https://` with the `URL` class as previously shown to securely download documents from web servers that support SSL.

[1] An earlier version of JSSE using different package names is available as a separate download for use with Java 1.2 and Java 1.3. See http://java.sun.com/products/jsse/.

Like all Java security APIs, JSSE is highly configurable and gives low-level control over all details of setting up and communicating over an SSL socket. The `javax.net` and `javax.net.ssl` packages are fairly complex, but in practice, you need only a few classes to securely communicate with a server. The following program is a variant on the preceding code that uses HTTPS instead of HTTP to securely transfer the contents of the requested URL:

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;
import java.security.cert.*;

/**
 * Get a document from a web server using HTTPS. Usage:
 *   java HttpsDownload <hostname> <filename>
 **/
public class HttpsDownload {
    public static void main(String[] args) throws IOException {
        // Get a SocketFactory object for creating SSL sockets
        SSLSocketFactory factory =
            (SSLSocketFactory) SSLSocketFactory.getDefault();

        // Use the factory to create a secure socket connected to the
        // HTTPS port of the specified web server.
        SSLSocket sslsock=(SSLSocket)factory.createSocket(args[0], // Hostname
                                                          443); // HTTPS port

        // Get the certificate presented by the web server
        SSLSession session = sslsock.getSession();
        X509Certificate cert;
        try { cert = (X509Certificate)session.getPeerCertificates()[0]; }
        catch(SSLPeerUnverifiedException e) { // If no or invalid certificate
            System.err.println(session.getPeerHost() +
                               " did not present a valid certificate.");
            return;
        }

        // Display details about the certificate
        System.out.println(session.getPeerHost() +
                           " has presented a certificate belonging to:");
        System.out.println("\t[" + cert.getSubjectDN().getName() + "]");
        System.out.println("The certificate bears the valid signature of:");
        System.out.println("\t[" + cert.getIssuerDN().getName() + "]");

        // If the user does not trust the certificate, abort
        System.out.print("Do you trust this certificate (y/n)? ");
        System.out.flush();
        BufferedReader console =
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        new BufferedReader(new InputStreamReader(System.in));
    if (Character.toLowerCase(console.readLine().charAt(0)) != 'y') return;

    // Now use the secure socket just as you would use a regular socket
    // First, send a regular HTTP request over the SSL socket
    PrintWriter out = new PrintWriter(sslsock.getOutputStream());
    out.print("GET " + args[1] + " HTTP/1.0\r\n\r\n");
    out.flush();

    // Next, read the server's response and print it to the console
    BufferedReader in =
     new BufferedReader(new InputStreamReader(sslsock.getInputStream()));
    String line;
    while((line = in.readLine()) != null) System.out.println(line);

    // Finally, close the socket
    sslsock.close();
  }
}
```

5.10.4. Servers

A client application uses a `Socket` to communicate with a server. The server does the same thing: it uses a `Socket` object to communicate with each of its clients. However, the server has an additional task in that it must be able to recognize and accept client connection requests. This is done with the `ServerSocket` class. The following code shows how you might use a `ServerSocket`. The code implements a simple HTTP server that responds to all requests by sending back (or mirroring) the exact contents of the HTTP request. A dummy server like this is useful when debugging HTTP clients:

```
import java.io.*;
import java.net.*;

public class HttpMirror {
  public static void main(String[] args) {
    try {
      int port = Integer.parseInt(args[0]);       // The port to listen on
      ServerSocket ss = new ServerSocket(port);   // Create a socket to listen
      for(;;) {                                   // Loop forever
        Socket client = ss.accept();              // Wait for a connection
        ClientThread t = new ClientThread(client);// A thread to handle it
        t.start();                                // Start the thread running
      }                                           // Loop again
    }
    catch (Exception e) {
      System.err.println(e.getMessage());
      System.err.println("Usage: java HttpMirror <port>;");
    }
  }

  static class ClientThread extends Thread {
    Socket client;
    ClientThread(Socket client) { this.client = client; }
    public void run() {
      try {
        // Get streams to talk to the client
        BufferedReader in =
          new BufferedReader(new InputStreamReader(client.getInputStream()));
        PrintWriter out =
          new PrintWriter(new OutputStreamWriter(client.getOutputStream()));

        // Send an HTTP response header to the client
        out.print("HTTP/1.0 200\r\nContent-Type: text/plain\r\n\r\n");

        // Read the HTTP request from the client and send it right back
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        // Stop when we read the blank line from the client that marks
        // the end of the request and its headers.
        String line;
        while((line = in.readLine()) != null) {
          if (line.length() == 0) break;
          out.println(line);
        }

        out.close();
        in.close();
        client.close();
      }
      catch (IOException e) { /* Ignore exceptions */ }
    }
  }
}
```

This server code could be modified using JSSE to support SSL connections. Making a server secure is more complex than making a client secure, however, because a server must have a certificate it can present to the client. Therefore, server-side JSSE is not demonstrated here.

5.10.5. Datagrams

Both `URL` and `Socket` perform networking on top of a stream-based network connection. Setting up and maintaining a stream across a network takes work at the network level, however. Sometimes you need a low-level way to speed a packet of data across a network, but you don't care about maintaining a stream. If, in addition, you don't need a guarantee that your data will get there or that the packets of data will arrive in the order you sent them, you may be interested in the `DatagramSocket` and `DatagramPacket` classes:

```
import java.net.*;

// Send a message to another computer via a datagram
try {
  String hostname = "host.example.com";    // The computer to send the data to
  InetAddress address =                    // Convert the DNS hostname
    InetAddress.getByName(hostname);       // to a lower-level IP address.
  int port = 1234;                         // The port to connect to
  String message = "The eagle has landed.";  // The message to send
  byte[] data = message.getBytes();          // Convert string to bytes
  DatagramSocket s = new DatagramSocket();   // Socket to send message with
  DatagramPacket p =                         // Create the packet to send
    new DatagramPacket(data, data.length, address, port);
  s.send(p);                                // Now send it!
  s.close();                                // Always close sockets when done
}
catch (UnknownHostException e) {}  // Thrown by InetAddress.getByName()
catch (SocketException e) {}       // Thrown by new DatagramSocket()
catch (java.io.IOException e) {}   // Thrown by DatagramSocket.send()

// Here's how the other computer can receive the datagram
try {
  byte[] buffer = new byte[4096];              // Buffer to hold data

  DatagramSocket s = new DatagramSocket(1234); // Socket that receives it
                                               // through
  DatagramPacket p =
    new DatagramPacket(buffer, buffer.length); // The packet that receives it
  s.receive(p);                                // Wait for a packet to arrive
  String msg =                                 // Convert the bytes from the
    new String(buffer, 0, p.getLength());      // packet back to a string.
  s.close();                                   // Always close the socket
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
}
catch (SocketException e) {}         // Thrown by new DatagramSocket()
catch (java.io.IOException e) {}     // Thrown by DatagramSocket.receive()
```

5.10.6. Testing the Reachability of a Host

In Java 5.0 the `InetAddress` class has an `isReachable( )` method that attempts to determine whether the host is reachable. The following code uses it in a naive Java implementation of the Unix *ping* utility:

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;

public class Ping {
    public static void main(String[] args) throws IOException {
        try {
            String hostname = args[0];
            int timeout = (args.length > 1)?Integer.parseInt(args[1]):2000;
            InetAddress[] addresses = InetAddress.getAllByName(hostname);
            for(InetAddress address : addresses) {
                if (address.isReachable(timeout))
                    System.out.printf("%s is reachable%n", address);
                else
                    System.out.printf("%s could not be contacted%n", address);
            }
        }
        catch (UnknownHostException e) {
            System.out.printf("Unknown host: %s%n", args[0]);
        }
        catch(IOException e) { System.out.printf("Network error: %s%n", e); }
        catch (Exception e) {
            // ArrayIndexOutOfBoundsException or NumberFormatException
            System.out.println("Usage: java Ping <hostname> [timeout in ms]");
        }
    }
}
```

5.11. I/O and Networking with java.nio

Java 1.4 introduced an entirely new API for high-performance, nonblocking I/O and networking. This API consists primarily of three new packages. `java.nio` defines `Buffer` classes that are used to store sequences of bytes or other primitive values. `java.nio.channels` defines *channels* through which data can be transferred between a buffer and a data source or sink, such as a file or a network socket. This package also contains important classes used for nonblocking I/O. Finally, the `java.nio.charset` package contains classes for efficiently converting buffers of bytes into buffers of characters. The sections that follow contain examples of using all three of these packages as well as examples of specific I/O tasks with the New I/O API.

5.11.1. Basic Buffer Operations

The `java.nio` package includes an abstract `Buffer` class, which defines generic operations on buffers. This package also defines type-specific subclasses such as

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

ByteBuffer, CharBuffer, and IntBuffer. (See Buffer and ByteBuffer in the reference section for details on these classes and their various methods.) The following code illustrates typical sequences of buffer operations on a ByteBuffer. The other type-specific buffer classes have similar methods.

```
import java.nio.*;

// Buffers don't have public constructors. They are allocated instead.
ByteBuffer b = ByteBuffer.allocate(4096);  // Create a buffer for 4,096 bytes
// Or do this to try to get an efficient buffer from the low-level OS
ByteBuffer buf2 = ByteBuffer.allocateDirect(65536);
// Here's another way to get a buffer: by "wrapping" an array
byte[] data;  // Assume this array is created and initialized elsewhere
ByteBuffer buf3 = ByteBuffer.wrap(data); // Create buffer that uses the array
// It is also possible to create a "view buffer" to view bytes as other types
buf3.order(ByteOrder.BIG_ENDIAN);   // Specify the byte order for the buffer
IntBuffer ib = buf3.asIntBuffer();  // View those bytes as integers

// Now store some data in the buffer
b.put(data);       // Copy bytes from array to buffer at current position
b.put((byte)42);   // Store another byte at the new current position
b.put(0, (byte)9); // Overwrite first byte in buffer. Don't change position.
b.order(ByteOrder.BIG_ENDIAN);  // Set the byte order of the buffer
b.putChar('x');       // Store the two bytes of a Unicode character in buffer
b.putInt(0xcafebabe); // Store four bytes of an int into the buffer

// Here are methods for querying basic numbers about a buffer
int capacity = b.capacity();  // How many bytes can the buffer hold? (4,096)
int position = b.position();  // Where will the next byte be written or read?
// A buffer's limit specifies how many bytes of the buffer can be used.
// When writing into a buffer, this should be the capacity. When reading data
// from a buffer, it should be the number of bytes that were previously
// written.
int limit = b.limit();         // How many should be used?
int remaining = b.remaining(); // How many left? Return limit-position.
boolean more=b.hasRemaining(); // Test if there is still room in the buffer

// The position and limit can also be set with methods of the same name
// Suppose you want to read the bytes you've written into the buffer
b.limit(b.position());         // Set limit to current position
b.position(0);                 // Set limit to 0; start reading at beginning

// Instead of the two previous calls, you usually use a convenience method
b.flip();   // Set limit to position and position to 0; prepare for reading
b.rewind(); // Set position to 0; don't change limit; prepare for rereading
b.clear();  // Set position to 0 and limit to capacity; prepare for writing

// Assuming you've called flip(), you can start reading bytes from the buffer
buf2.put(b);        // Read all bytes from b and put them into buf2
b.rewind();         // Rewind b for rereading from the beginning
byte b0 = b.get();  // Read first byte; increment buffer position
byte b1 = b.get();  // Read second byte; increment buffer position
byte[] fourbytes = new byte[4];
b.get(fourbytes);   // Read next four bytes, add 4 to buffer position
byte b9 = b.get(9); // Read 10th byte, without changing current position
int i = b.getInt(); // Read next four bytes as an integer; add 4 to position

// Discard bytes you've already read; shift the remaining ones to the
// beginning of the buffer; set position to new limit and limit to capacity,
// preparing the buffer for writing more bytes into it.
b.compact();
```

You may notice that many buffer methods return the object on which they operate. This is done so that method calls can be "chained" in code, as follows:

```
ByteBuffer bb=ByteBuffer.allocate(32).order(ByteOrder.BIG_ENDIAN).putInt(1234);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

Many methods throughout `java.nio` and its subpackages return the current object to enable this kind of method chaining. Note that the use of this kind of chaining is a stylistic choice (which I have avoided in this chapter) and does not have any significant impact on efficiency.

`ByteBuffer` is the most important of the buffer classes. However, another commonly used class is `CharBuffer`. `CharBuffer` objects can be created by wrapping a string and can also be converted to strings. `CharBuffer` implements the new `java.lang.CharSequence` interface, which means that it can be used like a `String` or `StringBuffer` in certain applications (e.g., for regular expression pattern matching).

```
// Create a read-only CharBuffer from a string
CharBuffer cb = CharBuffer.wrap("This string is the data for the CharBuffer");
String s = cb.toString();  // Convert to a String with toString() method
System.out.println(cb);    // or rely on an implicit call to toString().
char c = cb.charAt(0);     // Use CharSequence methods to get characters
char d = cb.get(1);        // or use a CharBuffer absolute read.
// A relative read that reads the char and increments the current position
// Note that only the characters between the position and limit are used when
// a CharBuffer is converted to a String or used as a CharSequence.
char e = cb.get();
```

Bytes in a `ByteBuffer` are commonly converted to characters in a `CharBuffer` and vice versa. We'll see how to do this when we consider the `java.nio.charset` package.

5.11.2. Basic Channel Operations

Buffers are not all that useful on their own—there isn't much point in storing bytes into a buffer only to read them out again. Instead, buffers are typically used with channels: your program stores bytes into a buffer, then passes the buffer to a channel, which reads the bytes out of the buffer and writes them to a file, network socket, or some other destination. Or, in the reverse, your program passes a buffer to a channel, which reads bytes from a file, socket, or other source and stores those bytes into the buffer, where they can then be retrieved by your program. The `java.nio.channels` package defines several channel classes that represent files, sockets, datagrams, and pipes. (We'll see examples of these concrete classes later in this chapter.) The following code, however, is based on the capabilities of the various channel interfaces defined by `java.nio.channels` and should work with any `Channel` object:

```
Channel c;  // Object that implements Channel interface; initialized elsewhere
if (c.isOpen()) c.close();  // These are the only methods defined by Channel

// The read() and write() methods are defined by the
// ReadableByteChannel and WritableByteChannel interfaces.
ReadableByteChannel source;      // Initialized elsewhere
WritableByteChannel destination; // Initialized elsewhere
ByteBuffer buffer = ByteBuffer.allocateDirect(16384); // Low-level 16 KB buffer

// Here is the basic loop to use when reading bytes from a source channel and
// writing them to a destination channel until there are no more bytes to read
// from the source and no more buffered bytes to write to the destination.
while(source.read(buffer) != -1 || buffer.position() > 0) {
  // Flip buffer: set limit to position and position to 0. This prepares
```

```
        // the buffer for reading (which is done by a channel *write* operation).
        buffer.flip();
        // Write some or all of the bytes in the buffer to the destination
        destination.write(buffer);
        // Discard the bytes that were written, copying the remaining ones to
        // the start of the buffer. Set position to limit and limit to capacity,
        // preparing the buffer for writing (done by a channel *read* operation).
        buffer.compact();
      }

      // Don't forget to close the channels
      source.close();
      destination.close();
```

In addition to the `ReadableByteChannel` and `WritableByteChannel` interfaces illustrated in the preceding code, `java.nio.channels` defines several other channel interfaces. `ByteChannel` simply extends the readable and writable interfaces without adding any new methods. It is a useful shorthand for channels that support both reading and writing. `GatheringByteChannel` is an extension of `WritableByteChannel` that defines `write()` methods that *gather* bytes from more than one buffer and write them out. Similarly, `ScatteringByteChannel` is an extension of `ReadableByteChannel` that defines `read()` methods that read bytes from the channel and *scatter* or distribute them into more than one buffer. The gathering and scattering `write( )` and `read()` methods can be useful when working with network protocols that use fixed-size headers that you want to store in a buffer separate from the rest of the transferred data.

One confusing point to be aware of is that a channel read operation involves writing (or putting) bytes into a buffer, and a channel write operation involves reading (or getting) bytes from a buffer. Thus, when I say that the `flip( )` method prepares a buffer for reading, I mean that it prepares a buffer for use in a channel `write( )` operation! The reverse is true for the buffer's `compact()` method.

5.11.3. Encoding and Decoding Text with Charsets

A `java.nio.charset.Charset` object represents a character set plus an encoding for that character set. `Charset` and its associated classes, `CharsetEncoder` and `CharsetDecoder`, define methods for encoding strings of characters into sequences of bytes and decoding sequences of bytes into strings of characters. Since these classes are part of the New I/O API, they use the `ByteBuffer` and `CharBuffer` classes:

```
// The simplest case. Use Charset convenience routines to convert.
Charset charset = Charset.forName("ISO-8859-1"); // Get Latin-1 Charset
CharBuffer cb = CharBuffer.wrap("Hello World");  // Characters to encode
// Encode the characters and store the bytes in a newly allocated ByteBuffer
ByteBuffer bb = charset.encode(cb);
// Decode these bytes into a newly allocated CharBuffer and print them out
System.out.println(charset.decode(bb));
```

Note the use of the ISO-8859-1 (a.k.a. Latin-1) charset in this example. This 8-bit charset is suitable for most Western European languages, including English. Programmers who

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

work only with English may also use the 7-bit US-ASCII charset. The `Charset` class does not do encoding and decoding itself, and the previous convenience routines create `CharsetEncoder` and `CharsetDecoder` classes internally. If you plan to encode or decode multiple times, it is more efficient to create these objects yourself:

```
Charset charset = Charset.forName("US-ASCII");   // Get the charset
CharsetEncoder encoder = charset.newEncoder();   // Create an encoder from it
CharBuffer cb = CharBuffer.wrap("Hello World!"); // Get a CharBuffer
WritableByteChannel destination;                 // Initialized elsewhere
destination.write(encoder.encode(cb));           // Encode chars and write
```

The preceding `CharsetEncoder.encode( )` method must allocate a new `ByteBuffer` each time it is called. For maximum efficiency, you can call lower-level methods to do the encoding and decoding into an existing buffer:

```
ReadableByteChannel source;                      // Initialized elsewhere
Charset charset = Charset.forName("ISO-8859-1"); // Get the charset
CharsetDecoder decoder = charset.newDecoder();   // Create a decoder from it
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // Buffer to hold bytes
CharBuffer cb = CharBuffer.allocate(2048);       // Buffer to hold characters

while(source.read(bb) != -1) {  // Read bytes from the channel until EOF
  bb.flip();                    // Flip byte buffer to prepare for decoding
  decoder.decode(bb, cb, true); // Decode bytes into characters
  cb.flip();                    // Flip char buffer to prepare for printing
  System.out.print(cb);         // Print the characters
  cb.clear();                   // Clear char buffer to prepare for decoding
  bb.clear();                   // Prepare byte buffer for next channel read
}
source.close();                 // Done with the channel, so close it
System.out.flush();             // Make sure all output characters appear
```

The preceding code relies on the fact that ISO-8859-1 is an 8-bit encoding charset and that there is one-to-one mapping between characters and bytes. For more complex charsets, such as the UTF-8 encoding of Unicode or the EUC-JP charset used with Japanese text; however, this does not hold, and more than one byte is required for some (or all) characters. When this is the case, there is no guarantee that all bytes in a buffer can be decoded at once (the end of the buffer may contain a partial character). Also, since a single character may encode to more than one byte, it can be tricky to know how many bytes a given string will encode into. The following code shows a loop you can use to decode bytes in a more general way:

```
ReadableByteChannel source;                      // Initialized elsewhere
Charset charset = Charset.forName("UTF-8");      // A Unicode encoding
CharsetDecoder decoder = charset.newDecoder();   // Create a decoder from it
ByteBuffer bb = ByteBuffer.allocateDirect(2048); // Buffer to hold bytes
CharBuffer cb = CharBuffer.allocate(2048);       // Buffer to hold characters

// Tell the decoder to ignore errors that might result from bad bytes
decoder.onMalformedInput(CodingErrorAction.IGNORE);
decoder.onUnmappableCharacter(CodingErrorAction.IGNORE);

decoder.reset();                 // Reset decoder if it has been used before
while(source.read(bb) != -1) {   // Read bytes from the channel until EOF
  bb.flip();                     // Flip byte buffer to prepare for decoding
  decoder.decode(bb, cb, false); // Decode bytes into characters
  cb.flip();                     // Flip char buffer to prepare for printing
  System.out.print(cb);          // Print the characters
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    cb.clear();                    // Clear the character buffer
    bb.compact();                  // Discard already decoded bytes
  }
  source.close();                  // Done with the channel, so close it

  // At this point, there may still be some bytes in the buffer to decode
  bb.flip();                       // Prepare for decoding
  decoder.decode(bb, cb, true);    // Pass true to indicate this is the last call
  decoder.flush(cb);               // Output any final characters
  cb.flip();                       // Flip char buffer
  System.out.print(cb);            // Print the final characters
```

5.11.4. Working with Files

`FileChannel` is a concrete `Channel` class that performs file I/O and implements `ReadableByteChannel` and `WritableByteChannel` (although its `read()` method works only if the underlying file is open for reading, and its `write( )` method works only if the file is open for writing). Obtain a `FileChannel` object by using the `java.io` package to create a `FileInputStream`, a `FileOutputStream`, or a `RandomAccessFile` and then call the `getChannel()` method (added in Java 1.4) of that object. As an example, you can use two `FileChannel` objects to copy a file:

```
String filename = "test";   // The name of the file to copy
// Create streams to read the original and write the copy
FileInputStream fin = new FileInputStream(filename);
FileOutputStream fout = new FileOutputStream(filename + ".copy");
// Use the streams to create corresponding channel objects
FileChannel in = fin.getChannel();
FileChannel out = fout.getChannel();
// Allocate a low-level 8KB buffer for the copy
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
while(in.read(buffer) != -1 || buffer.position() > 0) {
  buffer.flip();      // Prepare to read from the buffer and write to the file
  out.write(buffer);  // Write some or all buffer contents
  buffer.compact();   // Discard all bytes that were written and prepare to
}                     // read more from the file and store them in the buffer.
in.close();           // Always close channels and streams when done with them
out.close();
fin.close();          // Note that closing a FileChannel does not
fout.close();         // automatically close the underlying stream.
```

`FileChannel` has special `transferTo( )` and `transferFrom()` methods that make it particularly easy (and on many operating systems, particularly efficient) to transfer a specified number of bytes from a `FileChannel` to some other specified channel, or from some other channel to a `FileChannel`. These methods allow us to simplify the preceding file-copying code to the following:

```
FileChannel in, out;              // Assume these are initialized as in the
                                  // preceding example.
long numbytes = in.size();        // Number of bytes in original file
in.transferTo(0, numbytes, out);  // Transfer that amount to output channel
```

This code could be equally well-written using `transferFrom( )` instead of `transferTo()` (note that these two methods expect their arguments in different orders):

```
long numbytes = in.size();
out.transferFrom(in, 0, numbytes);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`FileChannel` has other capabilities that are not shared by other channel classes. One of the most important is the ability to "memory map" a file or a portion of a file, i.e., to obtain a `MappedByteBuffer` (a subclass of `ByteBuffer`) that represents the contents of the file and allows you to read (and optionally write) file contents simply by reading from and writing to the buffer. Memory mapping a file is a somewhat expensive operation, so this technique is usually efficient only when you are working with a large file to which you need repeated access. Memory mapping offers you yet another way to perform the same file-copy operation shown previously:

```
long filesize = in.size();
ByteBuffer bb = in.map(FileChannel.MapMode.READ_ONLY, 0, filesize);
while(bb.hasRemaining()) out.write(bb);
```

The channel interfaces defined by `java.nio.channels` include `ByteChannel` but not `CharChannel`. The channel API is low-level and provides methods for reading bytes only. All of the previous examples have treated files as binary files. It is possible to use the `CharsetEncoder` and `CharsetDecoder` classes introduced earlier to convert between bytes and characters, but when you want to work with text files, the `Reader` and `Writer` classes of the `java.io` package are usually much easier to use than `CharBuffer`. Fortunately, the `Channels` class defines convenience methods that bridge between the old and new APIs. Here is code that wraps a `Reader` and a `Writer` object around input and output channels, reads lines of Latin-1 text from the input channel, and writes them back out to the output channel, with the encoding changed to UTF-8:

```
ReadableByteChannel in;      // Assume these are initialized elsewhere
WritableByteChannel out;
// Create a Reader and Writer from a FileChannel and charset name
BufferedReader reader=new BufferedReader(Channels.newReader(in, "ISO-8859-1"));
PrintWriter writer = new PrintWriter(Channels.newWriter(out, "UTF-8"));
String line;
while((line = reader.readLine()) != null) writer.println(line);
reader.close();
writer.close();
```

Unlike the `FileInputStream` and `FileOutputStream` classes, the `FileChannel` class allows random access to the contents of the file. The zero-argument `position()` method returns the *file pointer* (the position of the next byte to be read), and the one-argument `position( )` method allows you to set this pointer to any value you want. This allows you to skip around in a file in the way that the `java.io.RandomAccessFile` does. Here is an example:

```
// Suppose you have a file that has data records scattered throughout, and the
// last 1,024 bytes of the file are an index that provides the position of
// those records. Here is code that reads the index of the file, looks up the
// position of the first record within the file and then reads that record.
FileChannel in = new FileInputStream("test.data").getChannel(); // The channel
ByteBuffer index = ByteBuffer.allocate(1024);   // A buffer to hold the index
long size = in.size();                          // The size of the file
in.position(size - 1024);                       // Position at start of index
in.read(index);                                 // Read the index
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
int record0Position = index.getInt(0);          // Get first index entry
in.position(record0Position);                   // Position file at that point
ByteBuffer record0 = ByteBuffer.allocate(128);  // Get buffer to hold data
in.read(record0);                               // Finally, read the record
```

The final feature of `FileChannel` that we'll consider here is its ability to lock a file or a portion of a file against all concurrent access (an exclusive lock) or against concurrent writes (a shared lock). (Note that some operating systems strictly enforce all locks while others provide only an advisory locking facility that requires programs to cooperate and to attempt to acquire a lock before reading or writing portions of a shared file.) In the previous random-access example, suppose we wanted to ensure that no other program was modifying the record data while we read it. We could acquire a shared lock on that portion of the file with the following code:

```
FileLock lock = in.lock(record0Position, // Start of locked region
                        128,     // Length of locked region
                        true);   // Shared lock: prevent concurrent updates
                                 // but allow concurrent reads.
in.position(record0Position);    // Move to start of index
in.read(record0);                // Read the index data
lock.release();                  // You're done with the lock, so release it
```

5.11.5. Client-Side Networking

The New I/O API includes networking capabilities as well as file-access capabilities. To communicate over the network, you can use the `SocketChannel` class. Create a `SocketChannel` with the static `open()` method, then read and write bytes from and to it as you would with any other channel object. The following code uses `SocketChannel` to send an HTTP request to a web server and saves the server's response (including all of the HTTP headers) to a file. Note the use of `java.net.InetSocketAddress`, a subclass of `java.net.SocketAddress`, to tell the `SocketChannel` what to connect to. These classes were also introduced as part of the New I/O API.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

// Create a SocketChannel connected to the web server at www.oreilly.com
SocketChannel socket =
  SocketChannel.open(new InetSocketAddress("www.oreilly.com",80));
// A charset for encoding the HTTP request
Charset charset = Charset.forName("ISO-8859-1");
// Send an HTTP request to the server. Start with a string, wrap it to
// a CharBuffer, encode it to a ByteBuffer, then write it to the socket.
socket.write(charset.encode(CharBuffer.wrap("GET / HTTP/1.0\r\n\r\n")));
// Create a FileChannel to save the server's response to
FileOutputStream out = new FileOutputStream("oreilly.html");
FileChannel file = out.getChannel();
// Get a buffer for holding bytes while transferring from socket to file
ByteBuffer buffer = ByteBuffer.allocateDirect(8192);
// Now loop until all bytes are read from the socket and written to the file
while(socket.read(buffer) != -1 || buffer.position() > 0) {  // Are we done?
  buffer.flip();       // Prepare to read bytes from buffer and write to file
  file.write(buffer);  // Write some or all bytes to the file
  buffer.compact();    // Discard those that were written
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
}
socket.close();        // Close the socket channel
file.close();          // Close the file channel
out.close();           // Close the underlying file
```

Another way to create a `SocketChannel` is with the no-argument version of `open()`, which creates an unconnected channel. This allows you to call the `socket()` method to obtain the underlying socket, configure the socket as desired, and connect to the desired host with the connect method. For example:

```
SocketChannel sc = SocketChannel.open();  // Open an unconnected socket channel
Socket s = sc.socket();                   // Get underlying java.net.Socket
s.setSoTimeout(3000);                     // Time out after three seconds
// Now connect the socket channel to the desired host and port
sc.connect(new InetSocketAddress("www.davidflanagan.com", 80));

ByteBuffer buffer = ByteBuffer.allocate(8192);  // Create a buffer
try { sc.read(buffer); }                        // Try to read from socket
catch(SocketTimeoutException e) {               // Catch timeouts here
  System.out.println("The remote computer is not responding.");
  sc.close();
}
```

In addition to the `SocketChannel` class, the `java.nio.channels` package defines a `DatagramChannel` for networking with datagrams instead of sockets. `DatagramChannel` is not demonstrated here, but you can read about it in the reference section.

One of the most powerful features of the New I/O API is that channels such as `SocketChannel` and `DatagramChannel` can be used in nonblocking mode. We'll see examples of this in later sections.

5.11.6. Server-Side Networking

The `java.net` package defines a `Socket` class for communication between a client and a server and defines a `ServerSocket` class used by the server to listen for and accept connections from clients. The `java.nio.channels` package is analogous: it defines a `SocketChannel` class for data transfer and a `ServerSocketChannel` class for accepting connections. `ServerSocketChannel` is an unusual channel because it does not implement `ReadableByteChannel` or `WritableByteChannel`. Instead of `read()` and `write()` methods, it has an `accept()` method for accepting client connections and obtaining a `SocketChannel` through which it communicates with the client. Here is the code for a simple, single-threaded server that listens for connections on port 8000 and reports the current time to any client that connects:

```
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class DateServer {
    public static void main(String[] args) throws java.io.IOException {
        // Get a CharsetEncoder for encoding the text sent to the client
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
        CharsetEncoder encoder = Charset.forName("US-ASCII").newEncoder();

        // Create a new ServerSocketChannel and bind it to port 8000
        // Note that this must be done using the underlying ServerSocket
        ServerSocketChannel server = ServerSocketChannel.open();
        server.socket().bind(new java.net.InetSocketAddress(8000));

        for(;;) {  // This server runs forever
            // Wait for a client to connect
            SocketChannel client = server.accept();
            // Get the current date and time as a string
            String response = new java.util.Date().toString() + "\r\n";
            // Wrap, encode, and send the string to the client
            client.write(encoder.encode(CharBuffer.wrap(response)));
            // Disconnect from the client
            client.close();
        }
    }
}
```

5.11.7. Nonblocking I/O

The preceding `DateServer` class is a simple network server. Because it does not maintain a connection with any client, it never needs to communicate with more than one at a time, and there is never more than one `SocketChannel` in use. More realistic servers must be able to communicate with more than one client at a time. The `java.io` and `java.net` APIs allow only blocking I/O, so servers written using these APIs must use a separate thread for each client. For large-scale servers with many clients, this approach does not scale well. To solve this problem, the New I/O API allows most channels (but not `FileChannel`) to be used in nonblocking mode and allows a single thread to manage all pending connections. This is done with a `Selector` object, which keeps track of a set of registered channels and can block until one or more of those channels is ready for I/O, as the following code illustrates. This is a longer example than most in this chapter, but it is a complete working server class that manages a `ServerSocketChannel` and any number of `SocketChannel` connections to clients through a single `Selector` object.

```
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;          // For Set and Iterator

public class NonBlockingServer {
    public static void main(String[] args) throws IOException {

        // Get the character encoders and decoders you'll need
        Charset charset = Charset.forName("ISO-8859-1");
        CharsetEncoder encoder = charset.newEncoder();
        CharsetDecoder decoder = charset.newDecoder();

        // Allocate a buffer for communicating with clients
        ByteBuffer buffer = ByteBuffer.allocate(512);

        // All of the channels in this code will be in nonblocking mode.
        // So create a Selector object that will block while monitoring
        // all of the channels and stop blocking only when one or more
        // of the channels is ready for I/O of some sort.
        Selector selector = Selector.open();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Create a new ServerSocketChannel and bind it to port 8000
// Note that this must be done using the underlying ServerSocket
ServerSocketChannel server = ServerSocketChannel.open();
server.socket().bind(new java.net.InetSocketAddress(8000));
// Put the ServerSocketChannel into nonblocking mode
server.configureBlocking(false);
// Now register it with the Selector (note that register() is called
// on the channel, not on the selector object, however).
// The SelectionKey represents the registration of this channel with
// this Selector.
SelectionKey serverkey = server.register(selector,
                                         SelectionKey.OP_ACCEPT);

for(;;) {  // The main server loop. The server runs forever.
    // This call blocks until there is activity on one of the
    // registered channels. This is the key method in nonblocking
    // I/O.
    selector.select();

    // Get a java.util.Set containing the SelectionKey objects for
    // all channels that are ready for I/O.
    Set keys = selector.selectedKeys();

    // Use a java.util.Iterator to loop through the selected keys
    for(Iterator i = keys.iterator(); i.hasNext(); ) {
        // Get the next SelectionKey in the set and remove it
        // from the set. It must be removed explicitly, or it will
        // be returned again by the next call to select().
        SelectionKey key = (SelectionKey) i.next();
        i.remove();

        // Check whether this key is the SelectionKey obtained when
        // you registered the ServerSocketChannel.
        if (key == serverkey) {
            // Activity on the ServerSocketChannel means a client
            // is trying to connect to the server.
            if (key.isAcceptable()) {
                // Accept the client connection and obtain a
                // SocketChannel to communicate with the client.
                SocketChannel client = server.accept();
                // Put the client channel in nonblocking mode
                client.configureBlocking(false);
                // Now register it with the Selector object,
                // telling it that you'd like to know when
                // there is data to be read from this channel.
                SelectionKey clientkey =
                    client.register(selector, SelectionKey.OP_READ);
                // Attach some client state to the key. You'll
                // use this state when you talk to the client.
                clientkey.attach(new Integer(0));
            }
        }
        else {
            // If the key obtained from the Set of keys is not the
            // ServerSocketChannel key, then it must be a key
            // representing one of the client connections.
            // Get the channel from the key.
            SocketChannel client = (SocketChannel) key.channel();

            // If you are here, there should be data to read from
            // the channel, but double-check.
            if (!key.isReadable()) continue;

            // Now read bytes from the client. Assume that all the
            // client's bytes are in one read operation.
            int bytesread = client.read(buffer);

            // If read() returns -1, it indicates end-of-stream,
            // which means the client has disconnected, so
            // deregister the selection key and close the channel.
            if (bytesread == -1) {
                key.cancel();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
                        client.close();
                        continue;
                    }

                    // Otherwise, decode the bytes to a request string
                    buffer.flip();
                    String request = decoder.decode(buffer).toString();
                    buffer.clear();
                    // Now reply to the client based on the request string
                    if (request.trim().equals("quit")) {
                        // If the request was "quit", send a final message
                        // Close the channel and deregister the
                        // SelectionKey
                        client.write(encoder.encode(CharBuffer.wrap("Bye.")));
                        key.cancel();
                        client.close();
                    }
                    else {
                        // Otherwise, send a response string comprised of
                        // the sequence number of this request plus an
                        // uppercase version of the request string. Note
                        // that you keep track of the sequence number by
                        // "attaching" an Integer object to the
                        // SelectionKey and incrementing it each time.

                        // Get sequence number from SelectionKey
                        int num = ((Integer)key.attachment()).intValue();
                        // For response string
                        String response = num + ": " +
                            request.toUpperCase();
                        // Wrap, encode, and write the response string
                        client.write(encoder.encode(CharBuffer.wrap(response)));
                        // Attach an incremented sequence nubmer to the key
                        key.attach(new Integer(num+1));
                    }
                }
            }
        }
    }
}
```

Nonblocking I/O is most useful for writing network servers. It is also useful in clients that have more than one network connection pending at the same time. For example, consider a web browser downloading a web page and the images referenced by that page at the same time. One other interesting use of nonblocking I/O is to perform nonblocking socket connection operations. The idea is that you can ask a `SocketChannel` to establish a connection to a remote host and then go do other stuff (such as build a GUI, for example) while the underlying OS is setting up the connection across the network. Later, you do a `select()` call to block until the connection has been established, if it hasn't been already. The code for a nonblocking connect looks like this:

```
// Create a new, unconnected SocketChannel. Put it in nonblocking
// mode, register it with a new Selector, and then tell it to connect.
// The connect call will return instead of waiting for the network
// connect to be fully established.
Selector selector = Selector.open();
SocketChannel channel = SocketChannel.open();
channel.configureBlocking(false);
channel.register(selector, SelectionKey.OP_CONNECT);
channel.connect(new InetSocketAddress(hostname, port));

// Now go do other stuff while the connection is set up
// For example, you can create a GUI here
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Now block if necessary until the SocketChannel is ready to connect.
// Since you've registered only one channel with this selector, you
// don't need to examine the key set; you know which channel is ready.
while(selector.select() == 0) /* empty loop */;

// This call is necessary to finish the nonblocking connections
channel.finishConnect();

// Finally, close the selector, which deregisters the channel from it
selector.close();
```

5.12. XML

Java 1.4 and Java 5.0 have added powerful XML processing features to the Java platform:

`org.xml.sax`

This package and its two subpackages define the de facto standard SAX API (SAX stands for Simple API for XML). SAX is an event-driven, XML-parsing API: a SAX parser invokes methods of a specified `ContentHandler` object (as well as some other related handler objects) as it parses an XML document. The structure and content of the document are fully described by the method calls. This is a streaming API that does not build any permanent representation of the document. It is up to the `ContentHandler` implementation to store any state or perform any actions that are appropriate. This package includes classes for the SAX 2 API and deprecated classes for SAX 1.

`org.w3c.dom`

This package defines interfaces that represent an XML document in tree form. The Document Object Model (DOM) is a recommendation (essentially a standard) of the World Wide Web Consortium (W3C). A DOM parser reads an XML document and converts it into a tree of nodes that represent the full content of the document. Once the tree representation of the document is created, a program can examine and manipulate it however it wants. Java 1.4 includes the core module of the Level 2 DOM, and Java 5.0 includes the core, events, and load/save modules of the Level 3 DOM.

`javax.xml.parsers`

This package provides high-level interfaces for instantiating SAX and DOM parsers for parsing XML documents.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`javax.xml.transform`

This package and its subpackages define a Java API for transforming XML document content and representation using the XSLT standard.

`javax.xml.validation`

This Java 5.0 package provides support for validating an XML document against a schema. Implementations are required to support the W3C XML Schema standard and may also support other schema types as well.

`javax.xml.xpath`

This package, also new in Java 5.0, supports the evaluation of XPath for selecting nodes in an XML document.

Examples using each of these packages are presented in the following sections.

5.12.1. Parsing XML with SAX

The first step in parsing an XML document with SAX is to obtain a SAX parser. If you have a SAX parser implementation of your own, you can simply instantiate the appropriate parser class. It is usually simpler, however, to use the `javax.xml.parsers` package to instantiate whatever SAX parser is provided by the Java implementation. The code looks like this:

```
import javax.xml.parsers.*;

// Obtain a factory object for creating SAX parsers
SAXParserFactory parserFactory = SAXParserFactory.newInstance();

// Configure the factory object to specify attributes of the parsers it creates
parserFactory.setValidating(true);
parserFactory.setNamespaceAware(true);

// Now create a SAXParser object
SAXParser parser = parserFactory.newSAXParser();   // May throw exceptions
```

The `SAXParser` class is a simple wrapper around the `org.xml.sax.XMLReader` class. Once you have obtained one, as shown in the previous code, you can parse a document by simply calling one of the various `parse()` methods. Some of these methods use the deprecated SAX 1 `HandlerBase` class, and others use the current SAX 2 `org.xml.sax.helpers.DefaultHandler` class. The `DefaultHandler` class provides an empty implementation of all the methods of the `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver` interfaces. These are all the methods that the SAX parser can call while parsing an XML document. By subclassing `DefaultHandler` and defining the methods you care about, you can perform whatever

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

actions are necessary in response to the method calls generated by the parser. The following code shows a method that uses SAX to parse an XML file and determine the number of XML elements that appear in a document as well as the number of characters of plain text (possibly excluding "ignorable whitespace") that appear within those elements:

```
import java.io.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class SAXCount {
    public static void main(String[] args)
        throws SAXException,IOException, ParserConfigurationException
    {
        // Create a parser factory and use it to create a parser
        SAXParserFactory parserFactory = SAXParserFactory.newInstance();
        SAXParser parser = parserFactory.newSAXParser();
        // This is the name of the file you're parsing
        String filename = args[0];
        // Instantiate a DefaultHandler subclass to do your counting for you
        CountHandler handler = new CountHandler();
        // Start the parser. It reads the file and calls methods of the handler.
        parser.parse(new File(filename), handler);
        // When you're done, report the results stored by your handler object
        System.out.println(filename + " contains " + handler.numElements +
                           " elements and " + handler.numChars +
                           " other characters ");
    }

    // This inner class extends DefaultHandler to count elements and text in
    // the XML file and saves the results in public fields. There are many
    // other DefaultHandler methods you could override, but you need only
    // these.
    public static class CountHandler extends DefaultHandler {
        public int numElements = 0, numChars = 0;  // Save counts here
        // This method is invoked when the parser encounters the opening tag
        // of any XML element. Ignore the arguments but count the element.
        public void startElement(String uri, String localname, String qname,
                                 Attributes attributes) {
            numElements++;
        }

        // This method is called for any plain text within an element
        // Simply count the number of characters in that text
        public void characters(char[] text, int start, int length) {
            numChars += length;
        }
    }
}
```

5.12.2. Parsing XML with DOM

The DOM API is much different from the SAX API. While SAX is an efficient way to scan an XML document, it is not well-suited for programs that want to modify documents. Instead of converting an XML document into a series of method calls, a DOM parser converts the document into an `org.w3c.dom.Document` object, which is a tree of `org.w3c.dom.Node` objects. The conversion of the complete XML document to tree form allows random access to the entire document but can consume substantial amounts of memory.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

In the DOM API, each node in the document tree implements the `Node` interface and a type-specific subinterface. (The most common types of node in a DOM document are `Element` and `Text` nodes.) When the parser is done parsing the document, your program can examine and manipulate that tree using the various methods of `Node` and its subinterfaces. The following code uses JAXP to obtain a DOM parser (which, in JAXP parlance, is called a `DocumentBuilder`). It then parses an XML file and builds a document tree from it. Next, it examines the `Document` tree to search for `<sect1>` elements and prints the contents of the `<title>` of each.

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class GetSectionTitles {
    public static void main(String[] args)
        throws IOException, ParserConfigurationException,
               org.xml.sax.SAXException
    {
        // Create a factory object for creating DOM parsers and configure it
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setIgnoringComments(true); // We want to ignore comments
        factory.setCoalescing(true);       // Convert CDATA to Text nodes
        factory.setNamespaceAware(false);  // No namespaces: this is default
        factory.setValidating(false);      // Don't validate DTD: also default

        // Now use the factory to create a DOM parser, a.k.a. DocumentBuilder
        DocumentBuilder parser = factory.newDocumentBuilder();

        // Parse the file and build a Document tree to represent its content
        Document document = parser.parse(new File(args[0]));

        // Ask the document for a list of all <sect1> elements it contains
        NodeList sections = document.getElementsByTagName("sect1");
        // Loop through those <sect1> elements one at a time
        int numSections = sections.getLength();
        for(int i = 0; i < numSections; i++) {
            Element section = (Element)sections.item(i);  // A <sect1>
            // The first Element child of each <sect1> should be a <title>
            // element, but there may be some whitespace Text nodes first, so
            // loop through the children until you find the first element
            // child.
            Node title = section.getFirstChild();
            while(title != null && title.getNodeType() != Node.ELEMENT_NODE)
                title = title.getNextSibling();
            // Print the text contained in the Text node child of this element
            if (title != null)
                System.out.println(title.getFirstChild().getNodeValue());
        }
    }
}
```

5.12.3. Transforming XML Documents

The `javax.xml.transform` package defines a `TransformerFactory` class for creating `Transformer` objects. A `Transformer` can transform a document from its `Source` representation into a new `Result` representation and optionally apply an XSLT transformation to the document content in the process. Three subpackages define concrete implementations of the `Source` and `Result` interfaces, which allow documents to be transformed among three representations:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

`javax.xml.transform.stream`

Represents documents as streams of XML text.

`javax.xml.transform.dom`

Represents documents as DOM `Document` trees.

`javax.xml.transform.sax`

Represents documents as sequences of SAX method calls.

The following code shows one use of these packages to transform the representation of a document from a DOM `Document` tree into a stream of XML text. An interesting feature of this code is that it does not create the `Document` tree by parsing a file; instead, it builds it up from scratch.

```
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class DOMToStream {
    public static void main(String[] args)
        throws ParserConfigurationException,
               TransformerConfigurationException,
               TransformerException
    {
        // Create a DocumentBuilderFactory and a DocumentBuilder
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        // Instead of parsing an XML document, however, just create an empty
        // document that you can build up yourself.
        Document document = db.newDocument();

        // Now build a document tree using DOM methods
        Element book = document.createElement("book"); // Create new element
        book.setAttribute("id", "javanut4");           // Give it an attribute
        document.appendChild(book);                    // Add to the document
        for(int i = 1; i <= 3; i++) {                  // Add more elements
            Element chapter = document.createElement("chapter");
            Element title = document.createElement("title");
            title.appendChild(document.createTextNode("Chapter " + i));
            chapter.appendChild(title);
            chapter.appendChild(document.createElement("para"));
            book.appendChild(chapter);
        }

        // Now create a TransformerFactory and use it to create a Transformer
        // object to transform our DOM document into a stream of XML text.
        // No arguments to newTransformer() means no XSLT stylesheet
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();

        // Create the Source and Result objects for the transformation
        DOMSource source = new DOMSource(document);          // DOM document
        StreamResult result = new StreamResult(System.out);  // to XML text
```

```
        // Finally, do the transformation
        transformer.transform(source, result);
    }
}
```

The most interesting uses of `javax.xml.transform` involve XSLT stylesheets. XSLT is a complex but powerful XML grammar that describes how XML document content should be converted to another form (e.g., XML, HTML, or plain text). A tutorial on XSLT stylesheets is beyond the scope of this book, but the following code (which contains only six key lines) shows how you can apply such a stylesheet (which is an XML document itself) to another XML document and write the resulting document to a stream:

```
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class Transform {
    public static void main(String[] args)
        throws TransformerConfigurationException,
               TransformerException
    {
        // Get Source and Result objects for input, stylesheet, and output
        StreamSource input = new StreamSource(new File(args[0]));
        StreamSource stylesheet = new StreamSource(new File(args[1]));
        StreamResult output = new StreamResult(new File(args[2]));

        // Create a transformer and perform the transformation
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer(stylesheet);
        transformer.transform(input, output);
    }
}
```

5.12.4. Validating XML Documents

The `javax.xml.validation` package allows you to validate XML documents against a schema. SAX and DOM parsers obtained from the `javax.xml.parsers` package can perform validation against a DTD during the parsing process, but this package separates validation from parsing and also provides general support for arbitrary schema types. All implementations must support W3C XML Schema and are allowed to support other schema types, such as RELAX NG.

To use this package, begin with a `SchemaFactory` instance—a parser for a specific type of schema. Use this parser to parse a schema file into a `Schema` object. Obtain a `Validator` from the `Schema`, and then use the `Validator` to validate your XML document. The document is specified as a `SAXSource` or `DOMSource` object. You may recall these classes from the subpackages of `javax.xml.transform`.

If the document is valid, the `validate( )` method of the `Validator` object returns normally. If it is not valid, `validate( )` throws a `SAXException`. You can install an

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

org.xml.sax.ErrorHandler object for the Validator to provide some control over the kinds of validation errors that cause exceptions.

```
import javax.xml.XMLConstants;
import javax.xml.validation.*;
import javax.xml.transform.sax.SAXSource;
import org.xml.sax.*;
import java.io.*;

public class Validate {
    public static void main(String[] args) throws IOException {
        File documentFile = new File(args[0]);  // 1st arg is document
        File schemaFile = new File(args[1]);    // 2nd arg is schema

        // Get a parser to parse W3C schemas.  Note use of javax.xml package
        // This package contains just one class of constants.
        SchemaFactory factory =
            SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

        // Now parse the schema file to create a Schema object
        Schema schema = null;
        try { schema = factory.newSchema(schemaFile); }
        catch(SAXException e) { fail(e); }

        // Get a Validator object from the Schema.
        Validator validator = schema.newValidator();

        // Get a SAXSource object for the document
        // We could use a DOMSource here as well
        SAXSource source =
            new SAXSource(new InputSource(new FileReader(documentFile)));

        // Now validate the document
        try { validator.validate(source); }
        catch(SAXException e) { fail(e); }

        System.err.println("Document is valid");
    }

    static void fail(SAXException e) {
        if (e instanceof SAXParseException) {
            SAXParseException spe = (SAXParseException) e;
            System.err.printf("%s:%d:%d: %s%n",
                              spe.getSystemId(), spe.getLineNumber(),
                              spe.getColumnNumber(), spe.getMessage());
        }
        else {
            System.err.println(e.getMessage());
        }
        System.exit(1);
    }
}
```

5.12.5. Evaluating XPath Expressions

XPath is a language for referring to specific nodes in an XML document. For example, the XPath expression "//section/title/text( )" refers to the text inside of a <title> element inside a <section> element at any depth within the document. A full description of the XPath language is beyond the scope of this book. The javax.xml.xpath package, new in Java 5.0, provides a way to find all nodes in a document that match an XPath expression.

```
import javax.xml.xpath.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
public class XPathEvaluator {
    public static void main(String[] args)
        throws ParserConfigurationException, XPathExpressionException,
               org.xml.sax.SAXException, java.io.IOException
    {
        String documentName = args[0];
        String expression = args[1];

        // Parse the document to a DOM tree
        // XPath can also be used with a SAX InputSource
        DocumentBuilder parser =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = parser.parse(new java.io.File(documentName));

        // Get an XPath object to evaluate the expression
        XPath xpath = XPathFactory.newInstance().newXPath();

        System.out.println(xpath.evaluate(expression, doc));

        // Or evaluate the expression to obtain a DOM NodeList of all matching
        // nodes.  Then loop through each of the resulting nodes
        NodeList nodes = (NodeList)xpath.evaluate(expression, doc,
                                                  XPathConstants.NODESET);
        for(int i = 0, n = nodes.getLength(); i < n; i++) {
            Node node = nodes.item(i);
            System.out.println(node);
        }
    }
}
```

5.13. Types, Reflection, and Dynamic Loading

The `java.lang.Class` class represents data types in Java and, along with the classes in the `java.lang.reflect` package, gives Java programs the capability of introspection (or self-reflection); a Java class can look at itself, or any other class, and determine its superclass, what methods it defines, and so on.

5.13.1. Class Objects

You can obtain a `Class` object in Java in several ways:

```
// Obtain the Class of an arbitrary object o
Class c = o.getClass();

// Obtain a Class object for primitive types with various predefined constants
c = Void.TYPE;          // The special "no-return-value" type
c = Byte.TYPE;          // Class object that represents a byte
c = Integer.TYPE;       // Class object that represents an int
c = Double.TYPE;        // etc; see also Short, Character, Long, Float

// Express a class literal as a type name followed by ".class"
c = int.class;          // Same as Integer.TYPE
c = String.class;       // Same as "dummystring".getClass()
c = byte[].class;       // Type of byte arrays
c = Class[][].class;    // Type of array of arrays of Class objects
```

5.13.2. Reflecting on a Class

Once you have a `Class` object, you can perform some interesting reflective operations with it:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
import java.lang.reflect.*;

Object o;                   // Some unknown object to investigate
Class c = o.getClass();     // Get its type

// If it is an array, figure out its base type
while (c.isArray()) c = c.getComponentType();

// If c is not a primitive type, print its class hierarchy
if (!c.isPrimitive()) {
  for(Class s = c; s != null; s = s.getSuperclass())
    System.out.println(s.getName() + " extends");
}

// Try to create a new instance of c; this requires a no-arg constructor
Object newobj = null;
try { newobj = c.newInstance(); }
catch (Exception e) {
  // Handle InstantiationException, IllegalAccessException
}

// See if the class has a method named setText that takes a single String
// If so, call it with a string argument
try {
    Method m = c.getMethod("setText", new Class[] { String.class });
    m.invoke(newobj, new Object[] { "My Label" });
} catch(Exception e) { /* Handle exceptions here */ }

// These are varargs methods in Java 5.0 so the syntax is much cleaner.
// Look for and invoke a method named "put" that takes two Object arguments
try {
    Method m = c.getMethod("add", Object.class, Object.class);
    m.invoke(newobj, "key", "value");
} catch(Exception e) { System.out.println(e); }

// In Java 5.0 we can use reflection on enumerated types and constants
Class<Thread.State> ts = Thread.State.class;  // Thread.State type
if (ts.isEnum()) {  // If it is an enumerated type
    Thread.State[] constants = ts.getEnumConstants(); // get its constants
}
try {
    Field f = ts.getField("RUNNABLE");       // Get the field named "RUNNABLE"
    System.out.println(f.isEnumConstant());  // Is it an enumerated constant?
}
catch(Exception e) { System.out.println(e); }

// The VM discards generic type information at runtime, but it is stored
// in the class file for the compiler and is accessible through reflection
try {
    Class map = Class.forName("java.util.Map");

    TypeVariable<?>[] typevars = map.getTypeParameters();
    for(TypeVariable<?> typevar : typevars) {
        System.out.print(typevar.getName());
        Type[] bounds = typevar.getBounds();
        if (bounds.length > 0) System.out.print(" extends ");
        for(int i = 0; i < bounds.length; i++) {
            if (i > 0) System.out.print(" & ");
            System.out.print(bounds[i]);
        }
        System.out.println();
    }
}
catch(Exception e) { System.out.println(e); }

// In Java 5.0, reflection can also be used on annotation types and to
// determine the values of runtime visible annotations
Class<?> a = Override.class;  // an annotation class
if (a.isAnnotation()) {       // is this an annotation type?
    // Look for some meta-annotations
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    java.lang.annotation.Retention retention =
        a.getAnnotation(java.lang.annotation.Retention.class);
    if (retention != null)
        System.out.printf("Retention: %s%n", retention.value());
}
```

5.13.3. Dynamic Class Loading

`Class` also provides a simple mechanism for dynamic class loading in Java. For more complete control over dynamic class loading, however, you should use a `java.lang.ClassLoader` object, typically a `java.net.URLClassLoader`. This technique is useful, for example, when you want to load a class that is named in a configuration file instead of being hardcoded into your program:

```
// Dynamically load a class specified by name in a config file
String classname =                     // Look up the name of the class
    config.getProperty("filterclass",  // The property name
                       "com.davidflanagan.filters.Default"); // A default

try {
  Class c = Class.forName(classname);  // Dynamically load the class
  Object o = c.newInstance();          // Dynamically instantiate it
} catch (Exception e) { /* Handle exceptions */ }
```

The preceding code works only if the class to be loaded is in the class path. If this is not the case, you can create a custom `ClassLoader` object to load a class from a path (or URL) you specify yourself:

```
import java.net.*;
String classdir = config.getProperty("filterDirectory"); // Look up class path
try {
  ClassLoader loader = new URLClassLoader(new URL[] { new URL(classdir) });
  Class c = loader.loadClass(classname);
}
catch (Exception e) { /* Handle exceptions */ }
```

5.13.4. Dynamic Proxies

The `Proxy` class and `InvocationHandler` interface to the `java.lang.reflect` package were added to Java 1.3. `Proxy` is a powerful but infrequently used class that allows you to dynamically create a new class or instance that implements a specified interface or set of interfaces. It also dispatches invocations of the interface methods to an `InvocationHandler` object.

5.14. Object Persistence

The Java platform provides two mechanisms for object persistence: the ability to save object state so that the object can later be recreated. Both mechanisms involve serialization; the second is aimed particularly at JavaBeans.

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

5.14.1. Serialization

One of the most important features of the `java.io` package is the ability to *serialize* objects: to convert an object into a stream of bytes that can later be deserialized back into a copy of the original object. The following code shows how to use serialization to save an object to a file and later read it back:

```
Object o;  // The object we are serializing; it must implement Serializable
File f;    // The file we are saving it to

try {
  // Serialize the object
  ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(f));
  oos.writeObject(o);
  oos.close();

  // Read the object back in
  ObjectInputStream ois = new ObjectInputStream(new FileInputStream(f));
  Object copy = ois.readObject();
  ois.close();
}
catch (IOException e) { /* Handle input/output exceptions */ }
catch (ClassNotFoundException cnfe) { /* readObject() can throw this */ }
```

The previous example serializes to a file, but remember, you can write serialized objects to any type of stream. Thus, you can write an object to a byte array, then read it back from the byte array, creating a deep copy of the object. You can write the object's bytes to a compression stream or even write the bytes to a stream connected across a network to another program!

5.14.2. JavaBeans Persistence

Java 1.4 introduced a serialization mechanism intended for use with JavaBeans components. `java.io` serialization works by saving the state of the internal fields of an object. `java.beans` persistence, on the other hand, works by saving a bean's state as a sequence of calls to the public methods defined by the class. Since it is based on the public API rather than on the internal state, the JavaBeans persistence mechanism allows interoperability between different implementations of the same API, handles version skew more robustly, and is suitable for longer-term storage of serialized objects.

A bean and any descendant beans or other objects that are serialized with `java.beans.XMLEncoder` can be deserialized with `java.beans.XMLDecoder`. These classes write to and read from specified streams, but they are not stream classes themselves. Here is how you might encode a bean:

```
// Create a JavaBean, and set some properties on it
javax.swing.JFrame bean = new javax.swing.JFrame("PersistBean");
bean.setSize(300, 300);
// Now save its encoded form to the file bean.xml
BufferedOutputStream out =                // Create an output stream
    new BufferedOutputStream(new FileOutputStream("bean.xml"));
XMLEncoder encoder = new XMLEncoder(out);  // Create encoder for stream
encoder.writeObject(bean);                 // Encode the bean
encoder.close();                           // Close encoder and stream
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01
Prepared for Chris Kriel, Safari ID: chris@locale.co.za
User number: 907899 Copyright 2008, Safari Books Online, LLC.

Here is the corresponding code to decode the bean from its serialized form:

```
BufferedInputStream in =                    // Create input stream
    new BufferedInputStream(new FileInputStream("bean.xml"));
XMLDecoder decoder = new XMLDecoder(in);   // Create decoder for stream
Object b = decoder.readObject();           // Decode a bean
decoder.close();                           // Close decoder and stream
bean = (javax.swing.JFrame) b;             // Cast bean to proper type
bean.setVisible(true);                     // Start using it
```

5.15. Security

The `java.security` package defines quite a few classes related to the Java access-control architecture, which is discussed in more detail in Chapter 6. These classes allow Java programs to run untrusted code in a restricted environment from which it can do no harm. While these are important classes, you rarely need to use them. The more interesting classes are the ones used for message digests and digital signatures; they are demonstrated in the sections that follow.

5.15.1. Message Digests

A *message digest* is a value, also known as cryptographic checksum or secure hash, that is computed over a sequence of bytes. The length of the digest is typically much smaller than the length of the data for which it is computed, but any change, no matter how small, in the input bytes produces a change in the digest. When transmitting data (a message), you can transmit a message digest along with it. The recipient of the message can then recompute the message digest on the received data and, by comparing the computed digest to the received digest, determine whether the message or the digest was corrupted or tampered with during transmission. We saw a way to compute a message digest earlier in the chapter when we discussed streams. A similar technique can be used to compute a message digest for nonstreaming binary data:

```
import java.security.*;

// Obtain an object to compute message digests using the "Secure Hash
// Algorithm"; this method can throw a NoSuchAlgorithmException.
MessageDigest md = MessageDigest.getInstance("SHA");

byte[] data, data1, data2, secret;  // Some byte arrays initialized elsewhere

// Create a digest for a single array of bytes
byte[] digest = md.digest(data);

// Create a digest for several chunks of data
md.reset();            // Optional: automatically called by digest()
md.update(data1);      // Process the first chunk of data
md.update(data2);      // Process the second chunk of data
digest = md.digest();  // Compute the digest

// Create a keyed digest that can be verified if you know the secret bytes
md.update(data);             // The data to be transmitted with the digest
digest = md.digest(secret);  // Add the secret bytes and compute the digest

// Verify a digest like this
byte[] receivedData, receivedDigest;  // The data and the digest we received
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01
Prepared for Chris Kriel, Safari ID: chris@locale.co.za
User number: 907899 Copyright 2008, Safari Books Online, LLC.

```
byte[] verifyDigest = md.digest(receivedData);  // Digest the received data
// Compare computed digest to the received digest
boolean verified =  java.util.Arrays.equals(receivedDigest, verifyDigest);
```

5.15.2. Digital Signatures

A *digital signature* combines a message-digest algorithm with public-key cryptography. The sender of a message, Alice, can compute a digest for a message and then encrypt that digest with her private key. She then sends the message and the encrypted digest to a recipient, Bob. Bob knows Alice's public key (it is public, after all), so he can use it to decrypt the digest and verify that the message has not been tampered with. In performing this verification, Bob also learns that the digest was encrypted with Alice's private key since he was able to decrypt the digest successfully using Alice's public key. As Alice is the only one who knows her private key, the message must have come from Alice. A digital signature is called such because, like a pen-and-paper signature, it serves to authenticate the origin of a document or message. Unlike a pen-and-paper signature, however, a digital signature is very difficult, if not impossible, to forge, and it cannot simply be cut and pasted onto another document.

Java makes creating digital signatures easy. In order to create a digital signature, however, you need a `java.security.PrivateKey` object. Assuming that a keystore exists on your system (see the *keytool* documentation in Chapter 8), you can get one with code like the following:

```
// Here is some basic data we need
File homedir = new File(System.getProperty("user.home"));
File keyfile = new File(homedir, ".keystore"); // Or read from config file
String filepass = "KeyStore password"          // Password for entire file
String signer = "david";                       // Read from config file
String password = "No one can guess this!";    // Better to prompt for this
PrivateKey key;  // This is the key we want to look up from the keystore

try {
  // Obtain a KeyStore object and then load data into it
  KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
  keystore.load(new BufferedInputStream(new FileInputStream(keyfile)),
                filepass.toCharArray());
  // Now ask for the desired key
  key = (PrivateKey) keystore.getKey(signer, password.toCharArray());
}
catch (Exception e) { /* Handle various exception types here */ }
```

Once you have a `PrivateKey` object, you can create a digital signature with a `java.security.Signature` object:

```
PrivateKey key;             // Initialized as shown previously
byte[] data;                // The data to be signed
Signature s =               // Obtain object to create and verify signatures
  Signature.getInstance("SHA1withDSA");  // Can throw a
                                          // NoSuchAlgorithmException
s.initSign(key);             // Initialize it; can throw an InvalidKeyException
s.update(data);              // Data to sign; can throw a SignatureException
/* s.update(data2); */       // Call multiple times to specify all data
byte[] signature = s.sign(); // Compute signature
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

A `Signature` object can verify a digital signature:

```
byte[] data;        // The signed data; initialized elsewhere
byte[] signature;   // The signature to be verified; initialized elsewhere
String signername;  // Who created the signature; initialized elsewhere
KeyStore keystore;  // Where certificates stored; initialize as shown earlier

// Look for a public-key certificate for the signer
java.security.cert.Certificate cert = keystore.getCertificate(signername);
PublicKey publickey = cert.getPublicKey();  // Get the public key from it

Signature s = Signature.getInstance("SHA1withDSA"); // Or some other algorithm
s.initVerify(publickey);                            // Setup for verification
s.update(data);                                     // Specify signed data
boolean verified = s.verify(signature);             // Verify signature data
```

5.15.3. Signed Objects

The `java.security.SignedObject` class is a convenient utility for wrapping a digital signature around an object. The `SignedObject` can then be serialized and transmitted to a recipient, who can deserialize it and use the `verify( )` method to verify the signature:

```
Serializable o;  // The object to be signed; must be Serializable
PrivateKey k;    // The key to sign with; initialized elsewhere
Signature s = Signature.getInstance("SHA1withDSA"); // Signature "engine"
SignedObject so = new SignedObject(o, k, s);        // Create the SignedObject

// The SignedObject encapsulates the object o; it can now be serialized
// and transmitted to a recipient.

// Here's how the recipient verifies the SignedObject
SignedObject so;           // The deserialized SignedObject
Object o;                  // The original object to extract from it
PublicKey pk;              // The key to verify with
Signature s = Signature.getInstance("SHA1withDSA"); // Verification "engine"
if (so.verify(pk,s))       // If the signature is valid,
  o = so.getObject();      // retrieve the encapsulated object.
```

5.16. Cryptography

The `java.security` package includes cryptography-based classes, but it does not contain classes for actual encryption and decryption. That is the job of the `javax.crypto` package. This package supports symmetric-key cryptography, in which the same key is used for both encryption and decryption and must be known by both the sender and the receiver of encrypted data.

5.16.1. Secret Keys

The `SecretKey` interface represents an encryption key; the first step of any cryptographic operation is to obtain an appropriate `SecretKey`. Unfortunately, the *keytool* program supplied with the JDK cannot generate and store secret keys, so a program must handle these tasks itself. Here is some code that shows various ways to work with `SecretKey` objects:

```
import javax.crypto.*;
import javax.crypto.spec.*;
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// Generate encryption keys with a KeyGenerator object
KeyGenerator desGen = KeyGenerator.getInstance("DES");       // DES algorithm
SecretKey desKey = desGen.generateKey();                     // Generate a key
KeyGenerator desEdeGen = KeyGenerator.getInstance("DESede"); // Triple DES
SecretKey desEdeKey = desEdeGen.generateKey();               // Generate a key

// SecretKey is an opaque representation of a key. Use SecretKeyFactory to
// convert to a transparent representation that can be manipulated: saved
// to a file, securely transmitted to a receiving party, etc.
SecretKeyFactory desFactory = SecretKeyFactory.getInstance("DES");
DESKeySpec desSpec = (DESKeySpec)
  desFactory.getKeySpec(desKey, javax.crypto.spec.DESKeySpec.class);
byte[] rawDesKey = desSpec.getKey();
// Do the same for a DESede key
SecretKeyFactory desEdeFactory = SecretKeyFactory.getInstance("DESede");
DESedeKeySpec desEdeSpec = (DESedeKeySpec)
  desEdeFactory.getKeySpec(desEdeKey, javax.crypto.spec.DESedeKeySpec.class);
byte[] rawDesEdeKey = desEdeSpec.getKey();

// Convert the raw bytes of a key back to a SecretKey object
DESedeKeySpec keyspec = new DESedeKeySpec(rawDesEdeKey);
SecretKey k = desEdeFactory.generateSecret(keyspec);

// For DES and DESede keys, there is an even easier way to create keys
// SecretKeySpec implements SecretKey, so use it to represent these keys
byte[] desKeyData = new byte[8];        // Read 8 bytes of data from a file
byte[] tripleDesKeyData = new byte[24]; // Read 24 bytes of data from a file
SecretKey myDesKey = new SecretKeySpec(desKeyData, "DES");
SecretKey myTripleDesKey = new SecretKeySpec(tripleDesKeyData, "DESede");
```

5.16.2. Encryption and Decryption with Cipher

Once you have obtained an appropriate `SecretKey` object, the central class for encryption and decryption is `Cipher`. Use it like this:

```
SecretKey key;     // Obtain a SecretKey as shown earlier
byte[] plaintext;  // The data to encrypt; initialized elsewhere

// Obtain an object to perform encryption or decryption
Cipher cipher = Cipher.getInstance("DESede");  // Triple-DES encryption
// Initialize the cipher object for encryption
cipher.init(Cipher.ENCRYPT_MODE, key);
// Now encrypt data
byte[] ciphertext = cipher.doFinal(plaintext);

// If we had multiple chunks of data to encrypt, we can do this
cipher.update(message1);
cipher.update(message2);
byte[] ciphertext = cipher.doFinal();

// We simply reverse things to decrypt
cipher.init(Cipher.DECRYPT_MODE, key);
byte[] decryptedMessage = cipher.doFinal(ciphertext);

// To decrypt multiple chunks of data
byte[] decrypted1 = cipher.update(ciphertext1);
byte[] decrypted2 = cipher.update(ciphertext2);
byte[] decrypted3 = cipher.doFinal(ciphertext3);
```

5.16.3. Encrypting and Decrypting Streams

The `Cipher` class can also be used with `CipherInputStream` or `CipherOutputStream` to encrypt or decrypt while reading or writing streaming data:

```
byte[] data;                              // The data to encrypt
SecretKey key;                            // Initialize as shown earlier
Cipher c = Cipher.getInstance("DESede");  // The object to perform encryption
c.init(Cipher.ENCRYPT_MODE, key);         // Initialize it

// Create a stream to write bytes to a file
FileOutputStream fos = new FileOutputStream("encrypted.data");

// Create a stream that encrypts bytes before sending them to that stream
// See also CipherInputStream to encrypt or decrypt while reading bytes
CipherOutputStream cos = new CipherOutputStream(fos, c);

cos.write(data);                  // Encrypt and write the data to the file
cos.close();                      // Always remember to close streams
java.util.Arrays.fill(data, (byte)0); // Erase the unencrypted data
```

5.16.4. Encrypted Objects

Finally, the `javax.crypto.SealedObject` class provides an especially easy way to perform encryption. This class serializes a specified object and encrypts the resulting stream of bytes. The `SealedObject` can then be serialized itself and transmitted to a recipient. The recipient can retrieve the original object only if she knows the required `SecretKey`:

```
Serializable o;           // The object to be encrypted; must be Serializable
SecretKey key;                             // The key to encrypt it with
Cipher c = Cipher.getInstance("Blowfish");  // Object to perform encryption
c.init(Cipher.ENCRYPT_MODE, key);           // Initialize it with the key
SealedObject so = new SealedObject(o, c);   // Create the sealed object

// Object so is a wrapper around an encrypted form of the original object o;
// it can now be serialized and transmitted to another party.
// Here's how the recipient decrypts the original object
Object original = so.getObject(key);        // Must use the same SecretKey
```

5.17. Miscellaneous Platform Features

The following sections detail important but miscellaneous features of the Java platform, including properties, preferences, processes, and management and instrumentation.

5.17.1. Properties

`java.util.Properties` is a subclass of `java.util.Hashtable`, a legacy collections class that predates the Collections API introduced in Java 1.2. A `Properties` object maintains a mapping between string keys and string values and defines methods that allow the mappings to be written to and read from a simple text file or (in Java 5.0) an XML file. This makes the `Properties` class ideal for configuration and user preference files. The `Properties` class is also used for the system properties returned by `System.getProperty( )`:

```
import java.util.*;
import java.io.*;

// Note: many of these system properties calls throw a security exception if
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
// called from untrusted code such as applets.
String homedir = System.getProperty("user.home"); // Get a system property
Properties sysprops = System.getProperties();     // Get all system properties

// Print the names of all defined system properties
for(Enumeration e = sysprops.propertyNames(); e.hasMoreElements();)
  System.out.println(e.nextElement());

sysprops.list(System.out); // Here's an even easier way to list the properties

// Read properties from a configuration file
Properties options = new Properties();             // Empty properties list
File configfile = new File(homedir, ".config");    // The configuration file
try {
  options.load(new FileInputStream(configfile));   // Load props from the file
} catch (IOException e) { /* Handle exception here */ }

// Query a property ("color"), specifying a default ("gray") if undefined
String color = options.getProperty("color", "gray");

// Set a property named "color" to the value "green"
options.setProperty("color", "green");

// Store the contents of the Properties object back into a file
try {
  options.store(new FileOutputStream(configfile),  // Output stream
                "MyApp Config File");              // File header comment text
} catch (IOException e) { /* Handle exception */ }

// In Java 5.0 properties can be written to or read from XML files
try {
  options.storeToXML(new FileOutputStream(configfile),  // Output stream
                     "MyApp Config File");              // Comment text
  options.loadFromXML(new FileInputStream(configfile)); // Read it back in
}
catch(IOException e) { /* Handle exception */ }
catch(InvalidPropertiesFormatException e) { /* malformed input */ }
```

5.17.2. Preferences

Java 1.4 introduced the Preferences API, which is specifically tailored for working with user and systemwide preferences and is more useful than Properties for this purpose. The Preferences API is defined by the `java.util.prefs` package. The key class in that package is `Preferences`. You can obtain a `Preferences` object that contains user-specific preferences with the static method `Preferences.userNodeForPackage()` and obtain a `Preferences` object that contains systemwide preferences with `Preferences.systemNodeForPackage()`. Both methods take a `java.lang.Class` object as their sole argument and return a `Preferences` object shared by all classes in that package. (This means that the preference names you use must be unique within the package.) Once you have a `Preferences` object, use the `get( )` method to query the string value of a named preference, or use other type-specific methods such as `getInt( )`, `getBoolean()`, and `getByteArray()`. Note that to query preference values, a default value must be passed for all methods. This default value is returned if no preference with the specified name has been registered or if the file or database that holds the preference data cannot be accessed. A typical use of `Preferences` is the following:

Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
package com.davidflanagan.editor;
import java.util.prefs.Preferences;

public class TextEditor {
  // Fields to be initialized from preference values
  public int width;             // Screen width in columns
  public String dictionary;     // Dictionary name for spell checking

  public void initPrefs() {
    // Get Preferences objects for user and system preferences for this package
    Preferences userprefs = Preferences.userNodeForPackage(TextEditor.class);
    Preferences sysprefs = Preferences.systemNodeForPackage(TextEditor.class);

    // Look up preference values. Note that you always pass a default value.
    width = userprefs.getInt("width", 80);
    // Look up a user preference using a system preference as the default
    dictionary = userprefs.get("dictionary",
                               sysprefs.get("dictionary",
                                            "default_dictionary"));
  }
}
```

In addition to the `get( )` methods for querying preference values, there are corresponding `put()` methods for setting the values of named preferences:

```
// User has indicated a new preference, so store it
userprefs.putBoolean("autosave", false);
```

If your application wants to be notified of user or system preference changes while the application is in progress, it may register a `PreferenceChangeListener` with `addPreferenceChangeListener()`. A `Preferences` object can export the names and values of its preferences as an XML file and can read preferences from such an XML file. (See `importPreferences( )`, `exportNode()`, and `exportSubtree( )` in `java.util.pref.Preferences` in the reference section.) `Preferences` objects exist in a hierarchy that typically corresponds to the hierarchy of package names. Methods for navigating this hierarchy exist but are not typically used by ordinary applications.

5.17.3. Processes

Earlier in the chapter, we saw how easy it is to create and manipulate multiple threads of execution running within the same Java interpreter. Java also has a `java.lang.Process` class that represents an operating system process running externally to the interpreter. A Java program can communicate with an external process using streams in the same way that it might communicate with a server running on some other computer on the network. Using a `Process` is always platform-dependent and is rarely portable, but it is sometimes a useful thing to do:

```
// Maximize portability by looking up the name of the command to execute
// in a configuration file.
java.util.Properties config;
String cmd = config.getProperty("sysloadcmd");
if (cmd != null) {
  // Execute the command; Process p represents the running command
  Process p = Runtime.getRuntime().exec(cmd);         // Start the command
  InputStream pin = p.getInputStream();               // Read bytes from it
  InputStreamReader cin = new InputStreamReader(pin); // Convert them to chars
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
    BufferedReader in = new BufferedReader(cin);        // Read lines of chars
    String load = in.readLine();                        // Get the command output
    in.close();                                         // Close the stream
}
```

In Java 5.0 the `java.lang.ProcessBuilder` class provides a more flexible way to launch new processes than the `Runtime.exec()` method. `ProcessBuilder` allows control of environment variables through a `Map` and makes it simple to set the working directory. It also has an option to automatically redirect the standard error stream of the processes it launches to the standard output stream, which makes it much easier to read all output of a `Process`.

```
import java.util.Map;
import java.io.*

public class JavaShell {
    public static void main(String[] args) {
        // We use this to start commands
        ProcessBuilder launcher = new ProcessBuilder();
        // Our inherited environment vars.  We may modify these below
        Map<String,String> environment = launcher.environment();
        // Our processes will merge error stream with standard output stream
        launcher.redirectErrorStream(true);
        // Where we read the user's input from
        BufferedReader console =
            new BufferedReader(new InputStreamReader(System.in));

        while(true) {
            try {
                System.out.print("> ");               // display prompt
                System.out.flush();                   // force it to show
                String command = console.readLine();  // Read input

                if (command.equals("exit")) return;   // Exit command

                else if (command.startsWith("cd ")) { // change directory
                    launcher.directory(new File(command.substring(3)));
                }

                else if (command.startsWith("set ")) {// set environment var
                    command = command.substring(4);
                    int pos = command.indexOf('=');
                    String name = command.substring(0,pos).trim();
                    String var = command.substring(pos+1).trim();
                    environment.put(name, var);
                }

                else { // Otherwise it is a process to launch
                    // Break command into individual tokens
                    String[] words = command.split(" ");
                    launcher.command(words);      // Set the command
                    Process p = launcher.start(); // And launch a new process

                    // Now read and display output from the process
                    // until there is no more output to read
                    BufferedReader output = new BufferedReader(
                             new InputStreamReader(p.getInputStream()));
                    String line;
                    while((line = output.readLine()) != null)
                        System.out.println(line);

                    // The process should be done now, but wait to be sure.
                    p.waitFor();
                }
            }
            catch(Exception e) {
                System.out.println(e);
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
            }
        }
    }
}
```

5.17.4. Management and Instrumentation

Java 5.0 includes the powerful JMX API for remote monitoring and management of running applications. The full `javax.management` API is beyond the scope of this book. The reference section does cover the `java.lang.management` package, however: this package is an application of JMX for the monitoring and management of the Java virtual machine itself. `java.lang.instrument` is another Java 5.0 package: it allows the definition of "agents" that can be used to instrument the running JVM. In VMs that support it, `java.lang.instrument` can be used to redefine class files as they are loaded to add profiling or coverage testing code, for example. Class redefinition is beyond the scope of this chapter, but the following code uses the new instrumentation and management features of Java 5.0 to determine resource usages of a Java program. The example also demonstrates the `Runtime.addShutdownHook()` method, which registers code to be run when the VM starts shutting down.

```
import java.lang.instrument.*;
import java.lang.management.*;
import java.util.List;
import java.io.*;

public class ResourceUsageAgent {
    // A Java agent class defines a premain() method to run before main()
    public static void premain(final String args, final Instrumentation inst) {
       // This agent simply registers a shutdown hook to run when the VM exits
        Runtime.getRuntime().addShutdownHook(new Thread() {
                public void run() {
                    // This code runs when the VM exits
                    try {
                        // Decide where to send our output
                        PrintWriter out;
                        if (args != null && args.length() > 0)
                            out = new PrintWriter(new FileWriter(args));
                        else
                            out = new PrintWriter(System.err);

                        // Use java.lang.management to query peak thread usage
                        ThreadMXBean tb = ManagementFactory.getThreadMXBean();
                        out.printf("Current thread count: %d%n",
                                   tb.getThreadCount());
                        out.printf("Peak thread count: %d%n",
                                   tb.getPeakThreadCount());

                        // Use java.lang.management to query peak memory usage
                        List<MemoryPoolMXBean> pools =
                            ManagementFactory.getMemoryPoolMXBeans();
                        for(MemoryPoolMXBean pool: pools) {
                            MemoryUsage peak = pool.getPeakUsage();
                            out.printf("Peak %s memory used: %,d%n",
                                       pool.getName(), peak.getUsed());
                            out.printf("Peak %s memory reserved: %,d%n",
                                       pool.getName(), peak.getCommitted());
                        }

                        // Use the Instrumentation object passed to premain()
                        // to get a list of all classes that have been loaded
                        Class[] loaded = inst.getAllLoadedClasses();
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01

```
                    out.println("Loaded classes:");
                    for(Class c : loaded) out.println(c.getName());

                    out.close();  // close and flush the output stream
                }
                catch(Throwable t) {
                    // Exceptions in shutdown hooks are ignored so
                    // we've got to print this out explicitly
                    System.err.println("Exception in agent: " + t);
                }
            }
        });
    }
}
```

To monitor the resource usage of a Java program with this agent, you first must compile the class normally. You then store the generated class files in a JAR file with a manifest that specifies the class that contains the `premain()` method. Create a manifest file that contains this line:

```
Premain-Class: ResourceUsageAgent
```

Create the JAR file with a command like this:

```
% jar cmf manifest agent.jar ResourceUsageAgent*.class
```

Finally, to use the agent, specify the JAR file and the agent arguments with the -`javaagent` flag to the Java interpreter:

```
% java -javaagent:agent.jar=/tmp/usage.info my.java.Program
```


Java in a Nutshell, 5th Edition By David Flanagan ISBN: 0596007736 Publisher: O'Reilly
Print Publication Date: 2005/03/01