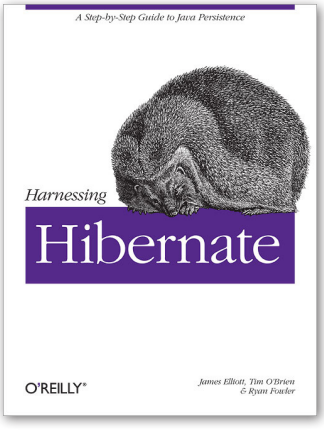
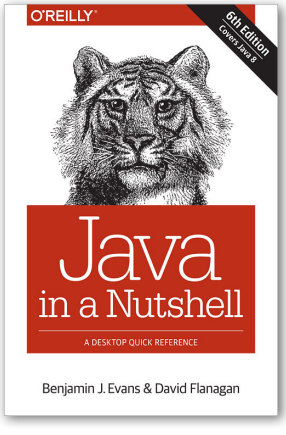
O'REILLY®

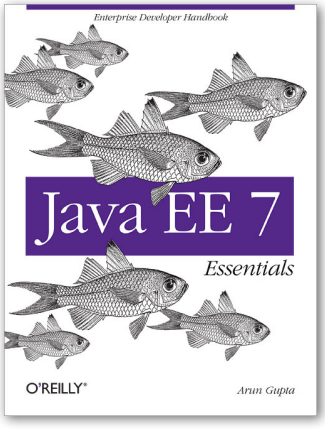
FREE DOWNLOAD

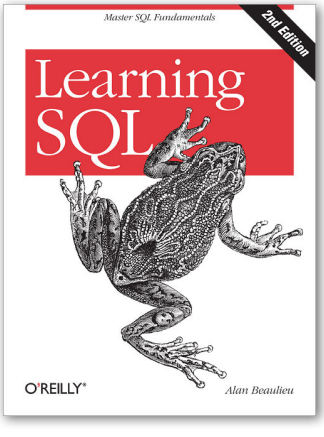
Back-End Java Development

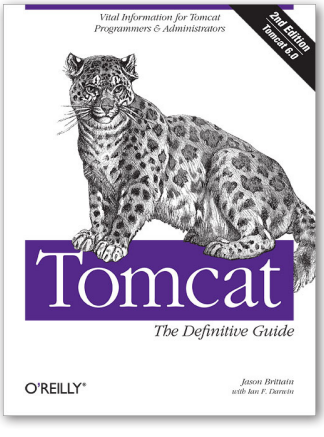
A Curated Collection of Chapters from the O'Reilly Programming and Data Libraries







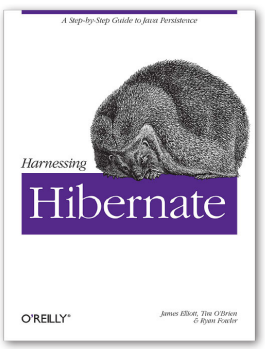


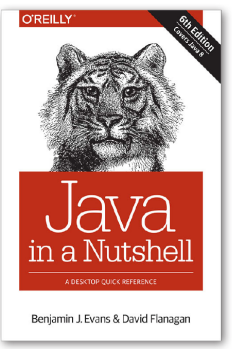


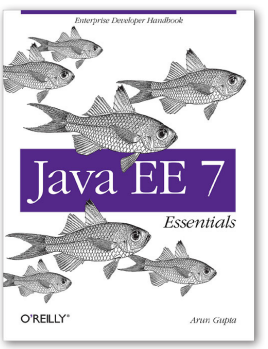
Back-End Java Development

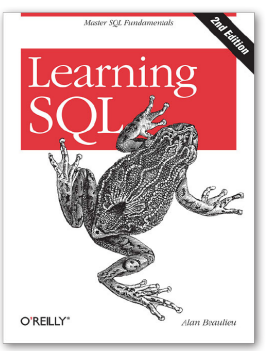
A Curated Collection of Chapters from the O'Reilly Programming and Data Libraries

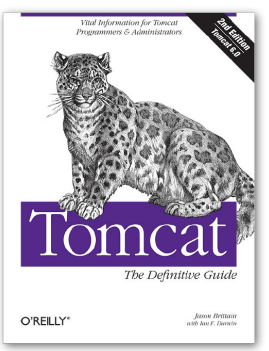
These curated chapters from bestselling O'Reilly books represent the most widely used technologies for developing server-side applications in Java. Get started by learning about Java's basic syntax, and from there scale up your knowledge base by taking a lightning-fast tour through the ecosystem's best tools, frameworks, application servers, and database management languages. All of the leading technologies are included here—Java EE, Spring, Tomcat, Hibernate, and SQL—so back-end developers can rest assured they'll have a trusty roadmap for deeper exploration and discovery.











Java in a Nutshell 6E
Available here
Chapter 2. Java Syntax from the Ground Up

Java EE 7 Essentials
Available here
Chapter 2. Servlets

Just Spring
Available here
Chapter 2. Fundamentals

Tomcat: The Definitive Guide
Available here
Chapter 1. Getting Started with Tomcat

Harnessing Hibernate
Available here
Chapter 3. Harnessing Hibernate

Learning SQL, 2nd Edition
Available here
Chapter 2. Creating and Populating a Database

O'REILLY®

6th Edition
Covers Java 8

Java in a Nutshell

A DESKTOP QUICK REFERENCE

Benjamin J. Evans & David Flanagan

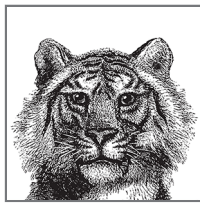
JAVA

IN A NUTSHELL

Sixth Edition

Benjamin J. Evans and David Flanagan

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®



2

Java Syntax from the Ground Up

This chapter is a terse but comprehensive introduction to Java syntax. It is written primarily for readers who are new to the language but have some previous programming experience. Determined novices with no prior programming experience may also find it useful. If you already know Java, you should find it a useful language reference. The chapter includes some comparisons of Java to C and C++ for the benefit of programmers coming from those languages.

This chapter documents the syntax of Java programs by starting at the very lowest level of Java syntax and building from there, covering increasingly higher orders of structure. It covers:

- The characters used to write Java programs and the encoding of those characters.
- Literal values, identifiers, and other tokens that comprise a Java program.
- The data types that Java can manipulate.
- The operators used in Java to group individual tokens into larger expressions.
- Statements, which group expressions and other statements to form logical chunks of Java code.
- Methods, which are named collections of Java statements that can be invoked by other Java code.
- Classes, which are collections of methods and fields. Classes are the central program element in Java and form the basis for object-oriented programming. Chapter 3 is devoted entirely to a discussion of classes and objects.
- Packages, which are collections of related classes.

- Java programs, which consist of one or more interacting classes that may be drawn from one or more packages.

The syntax of most programming languages is complex, and Java is no exception. In general, it is not possible to document all elements of a language without referring to other elements that have not yet been discussed. For example, it is not really possible to explain in a meaningful way the operators and statements supported by Java without referring to objects. But it is also not possible to document objects thoroughly without referring to the operators and statements of the language. The process of learning Java, or any language, is therefore an iterative one.

Java Programs from the Top Down

Before we begin our bottom-up exploration of Java syntax, let's take a moment for a top-down overview of a Java program. Java programs consist of one or more files, or *compilation units*, of Java source code. Near the end of the chapter, we describe the structure of a Java file and explain how to compile and run a Java program. Each compilation unit begins with an optional `package` declaration followed by zero or more `import` declarations. These declarations specify the namespace within which the compilation unit will define names, and the namespaces from which the compilation unit imports names. We'll see `package` and `import` again later in this chapter in "Packages and the Java Namespace" on page 88.

The optional `package` and `import` declarations are followed by zero or more reference type definitions. We will meet the full variety of possible reference types in Chapters 3 and 4, but for now, we should note that these are most often either `class` or `interface` definitions.

Within the definition of a reference type, we will encounter *members* such as *fields*, *methods*, and *constructors*. Methods are the most important kind of member. Methods are blocks of Java code comprised of *statements*.

With these basic terms defined, let's start by approaching a Java program from the bottom up by examining the basic units of syntax—often referred to as *lexical tokens*.

Lexical Structure

This section explains the lexical structure of a Java program. It starts with a discussion of the Unicode character set in which Java programs are written. It then covers the tokens that comprise a Java program, explaining comments, identifiers, reserved words, literals, and so on.

The Unicode Character Set

Java programs are written using Unicode. You can use Unicode characters anywhere in a Java program, including comments and identifiers such as variable names. Unlike the 7-bit ASCII character set, which is useful only for English, and

the 8-bit ISO Latin-1 character set, which is useful only for major Western European languages, the Unicode character set can represent virtually every written language in common use on the planet.

If you do not use a Unicode-enabled text editor, or if you do not want to force other programmers who view or edit your code to use a Unicode-enabled editor, you can embed Unicode characters into your Java programs using the special Unicode escape sequence `\u`*xxxx*, in other words, a backslash and a lowercase u, followed by four hexadecimal characters. For example, `\u0020` is the space character, and `\u03c0` is the character π.

Java has invested a large amount of time and engineering effort in ensuring that its Unicode support is first class. If your business application needs to deal with global users, especially in non-Western markets, then the Java platform is a great choice.

Case Sensitivity and Whitespace

Java is a case-sensitive language. Its keywords are written in lowercase and must always be used that way. That is, `While` and `WHILE` are not the same as the `while` keyword. Similarly, if you declare a variable named `i` in your program, you may not refer to it as `I`.

In general, relying on case sensitivity to distinguish identifiers is a terrible idea. Do not use it in your own code, and in particular never give an identifier the same name as a keyword but differently cased.

Java ignores spaces, tabs, newlines, and other whitespace, except when it appears within quoted characters and string literals. Programmers typically use whitespace to format and indent their code for easy readability, and you will see common indentation conventions in the code examples of this book.

Comments

Comments are natural-language text intended for human readers of a program. They are ignored by the Java compiler. Java supports three types of comments. The first type is a single-line comment, which begins with the characters `//` and continues until the end of the current line. For example:

```
int i = 0;   // Initialize the loop variable
```

The second kind of comment is a multiline comment. It begins with the characters `/*` and continues, over any number of lines, until the characters `*/`. Any text between the `/*` and the `*/` is ignored by `javac`. Although this style of comment is typically used for multiline comments, it can also be used for single-line comments.

This type of comment cannot be nested (i.e., one `/* */` comment cannot appear within another). When writing multiline comments, programmers often use extra `*` characters to make the comments stand out. Here is a typical multiline comment:

```
/*
 * First, establish a connection to the server.
 * If the connection attempt fails, quit right away.
 */
```

The third type of comment is a special case of the second. If a comment begins with `/**`, it is regarded as a special *doc comment*. Like regular multiline comments, doc comments end with `*/` and cannot be nested. When you write a Java class you expect other programmers to use, use doc comments to embed documentation about the class and each of its methods directly into the source code. A program named `javadoc` extracts these comments and processes them to create online documentation for your class. A doc comment can contain HTML tags and can use additional syntax understood by `javadoc`. For example:

```
/**
 * Upload a file to a web server.
 *
 * @param file The file to upload.
 * @return <tt>true</tt> on success,
 *         <tt>false</tt> on failure.
 * @author David Flanagan
 */
```

See Chapter 7 for more information on the doc comment syntax and Chapter 13 for more information on the `javadoc` program.

Comments may appear between any tokens of a Java program, but may not appear within a token. In particular, comments may not appear within double-quoted string literals. A comment within a string literal simply becomes a literal part of that string.

Reserved Words

The following words are reserved in Java (they are part of the syntax of the language and may not be used to name variables, classes, and so forth):

```
abstract   const      final        int          public       throw
assert     continue   finally      interface    return       throws
boolean    default    float        long         short        transient
break      do         for          native       static       true
byte       double     goto         new          strictfp     try
case       else       if           null         super        void
catch      enum       implements   package      switch       volatile
char       extends    import       private      synchronized while
class      false      instanceof   protected    this
```

We'll meet each of these reserved words again later in this book. Some of them are the names of primitive types and others are the names of Java statements, both of

which are discussed later in this chapter. Still others are used to define classes and their members (see Chapter 3).

Note that `const` and `goto` are reserved but aren't actually used in the language, and that `interface` has an additional variant form—`@interface`, which is used when defining types known as annotations. Some of the reserved words (notably `final` and `default`) have a variety of different meanings depending on context.

Identifiers

An *identifier* is simply a name given to some part of a Java program, such as a class, a method within a class, or a variable declared within a method. Identifiers may be of any length and may contain letters and digits drawn from the entire Unicode character set. An identifier may not begin with a digit. In general, identifiers may not contain punctuation characters. Exceptions include the ASCII underscore (_) and dollar sign ($) as well as other Unicode currency symbols such as £ and ¥.

Currency symbols are intended for use in automatically generated source code, such as code produced by `javac`. By avoiding the use of currency symbols in your own identifiers, you don't have to worry about collisions with automatically generated identifiers.

Formally, the characters allowed at the beginning of and within an identifier are defined by the methods `isJavaIdentifierStart()` and `isJavaIdentifierPart()` of the class `java.lang.Character`.

The following are examples of legal identifiers:

```
i    x1    theCurrentTime    the_current_time    獺
```

Note in particular the example of a UTF-8 identifier—獺. This is the Kanji character for "otter" and is perfectly legal as a Java identifier. The usage of non-ASCII identifiers is unusual in programs predominantly written by Westerners, but is sometimes seen.

Literals

Literals are values that appear directly in Java source code. They include integer and floating-point numbers, single characters within single quotes, strings of characters within double quotes, and the reserved words `true`, `false`, and `null`. For example, the following are all literals:

```
1    1.0    '1'    "one"    true    false    null
```

The syntax for expressing numeric, character, and string literals is detailed in "Primitive Data Types" on page 22.

Punctuation

Java also uses a number of punctuation characters as tokens. The Java Language Specification divides these characters (somewhat arbitrarily) into two categories, separators and operators. The twelve separators are:

```
(   )   {   }   [   ]

... @ ::

;   ,   .
```

The operators are:

```
+   -   *   /   %   &   |   ^   <<   >>   >>>
+=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=  >>>=
=   ==  !=  <   <=  >   >=
!   ~   &&  ||  ++  --  ?   :   ->
```

We'll see separators throughout the book, and will cover each operator individually in "Expressions and Operators" on page 30.

Primitive Data Types

Java supports eight basic data types known as *primitive types* as described in Table 2-1. The primitive types include a Boolean type, a character type, four integer types, and two floating-point types. The four integer types and the two floating-point types differ in the number of bits that represent them and therefore in the range of numbers they can represent.

Table 2-1. Java primitive data types

Type	Contains	Default	Size	Range
`boolean`	`true` or `false`	`false`	1 bit	NA
`char`	Unicode character	`\u0000`	16 bits	`\u0000` to `\uFFFF`
`byte`	Signed integer	0	8 bits	-128 to 127
`short`	Signed integer	0	16 bits	-32768 to 32767
`int`	Signed integer	0	32 bits	-2147483648 to 2147483647
`long`	Signed integer	0	64 bits	-9223372036854775808 to 9223372036854775807
`float`	IEEE 754 floating point	0.0	32 bits	1.4E-45 to 3.4028235E+38
`double`	IEEE 754 floating point	0.0	64 bits	4.9E-324 to 1.7976931348623157E+308

The next section summarizes these primitive data types. In addition to these primitive types, Java supports nonprimitive data types known as reference types, which are introduced in "Reference Types" on page 84.

The boolean Type

The `boolean` type represents truth values. This type has only two possible values, representing the two Boolean states: on or off, yes or no, true or false. Java reserves the words `true` and `false` to represent these two Boolean values.

Programmers coming to Java from other languages (especially JavaScript) should note that Java is much stricter about its Boolean values than other languages—in particular, a `boolean` is neither an integral nor an object type, and incompatible values cannot be used in place of a `boolean`. In other words, you cannot take shortcuts such as the following in Java:

```
Object o = new Object();
int i = 1;

if (o) {
  while(i) {
    //...
  }
}
```

Instead, Java forces you to write cleaner code by explicitly stating the comparisons you want:

```
if (o != null) {
  while(i != 0) {
    // ...
  }
}
```

The char Type

The `char` type represents Unicode characters. Java has a slightly unique approach to representing characters—`javac` accepts identifiers as UTF-8 (a variable-width encoding) in input, but represents chars internally as a fixed-width encoding that is 16 bits wide.

These distinctions do not normally need to concern the developer, however. In most cases, all that is required is to remember the rule that to include a character literal in a Java program, simply place it between single quotes (apostrophes):

```
char c = 'A';
```

You can, of course, use any Unicode character as a character literal, and you can use the `\u` Unicode escape sequence. In addition, Java supports a number of other escape sequences that make it easy both to represent commonly used nonprinting ASCII characters such as `newline` and to escape certain punctuation characters that have special meaning in Java. For example:

```
char tab = '\t', nul = '\000', aleph = '\u05D0', slash = '\\';
```

Table 2-2 lists the escape characters that can be used in `char` literals. These characters can also be used in string literals, which are covered in the next section.

Table 2-2. Java escape characters

Escape sequence	Character value
`\b`	Backspace
`\t`	Horizontal tab
`\n`	Newline
`\f`	Form feed
`\r`	Carriage return
`\"`	Double quote
`\'`	Single quote
`\\`	Backslash
`\` *xxx*	The Latin-1 character with the encoding *xxx*, where *xxx* is an octal (base 8) number between 000 and 377. The forms `\` *x* and `\` *xx* are also legal, as in `\0`, but are not recommended because they can cause difficulties in string constants where the escape sequence is followed by a regular digit. This form is generally discouraged in favor of the `\u`XXXX form.
`\u` *xxxx*	The Unicode character with encoding *xxxx*, where *xxxx* is four hexadecimal digits. Unicode escapes can appear anywhere in a Java program, not only in character and string literals.

`char` values can be converted to and from the various integral types, and the `char` data type is a 16-bit integral type. Unlike `byte`, `short`, `int`, and `long`, however, `char` is an unsigned type. The `Character` class defines a number of useful `static` methods for working with characters, including `isDigit()`, `isJavaLetter()`, `isLowerCase()`, and `toUpperCase()`.

The Java language and its `char` type were designed with Unicode in mind. The Unicode standard is evolving, however, and each new version of Java adopts a new version of Unicode. Java 7 uses Unicode 6.0 and Java 8 uses Unicode 6.2.

Recent releases of Unicode include characters whose encodings, or *codepoints*, do not fit in 16 bits. These supplementary characters, which are mostly infrequently

used Han (Chinese) ideographs, occupy 21 bits and cannot be represented in a single `char` value. Instead, you must use an `int` value to hold the codepoint of a supplementary character, or you must encode it into a so-called "surrogate pair" of two `char` values.

Unless you commonly write programs that use Asian languages, you are unlikely to encounter any supplementary characters. If you do anticipate having to process characters that do not fit into a `char`, methods have been added to the `Character`, `String`, and related classes for working with text using `int` codepoints.

String literals

In addition to the `char` type, Java also has a data type for working with strings of text (usually simply called *strings*). The `String` type is a class, however, and is not one of the primitive types of the language. Because strings are so commonly used, though, Java does have a syntax for including string values literally in a program. A `String` literal consists of arbitrary text within double quotes (as opposed to the single quotes for `char` literals). For example:

```
"Hello, world"
"'This' is a string!"
```

String literals can contain any of the escape sequences that can appear as `char` literals (see Table 2-2). Use the `\"` sequence to include a double quote within a `String` literal. Because `String` is a reference type, string literals are described in more detail later in this chapter in "Object Literals" on page 74. Chapter 9 contains more details on some of the ways you can work with `String` objects in Java.

Integer Types

The integer types in Java are `byte`, `short`, `int`, and `long`. As shown in Table 2-1, these four types differ only in the number of bits and, therefore, in the range of numbers each type can represent. All integral types represent signed numbers; there is no `unsigned` keyword as there is in C and C++.

Literals for each of these types are written exactly as you would expect: as a string of decimal digits, optionally preceded by a minus sign.[1] Here are some legal integer literals:

```
0
1
123
-42000
```

Integer literals can also be expressed in hexadecimal, binary, or octal notation. A literal that begins with `0x` or `0X` is taken as a hexadecimal number, using the letters `A` to `F` (or `a` to `f`) as the additional digits required for base-16 numbers.

1 Technically, the minus sign is an operator that operates on the literal, but is not part of the literal itself.

Integer binary literals start with `0b` and may, of course, only feature the digits 1 or 0. As binary literals can be very long, underscores are often used as part of a binary literal. The underscore character is ignored whenever it is encountered in any numerical literal—it's allowed purely to help with readability of literals.

Java also supports octal (base-8) integer literals. These literals begin with a leading `0` and cannot include the digits 8 or 9. They are not often used and should be avoided unless needed. Legal hexadecimal, binary, and octal literals include:

```
0xff              // Decimal 255, expressed in hexadecimal
0377              // The same number, expressed in octal (base 8)
0b0010_1111       // Decimal 47, expressed in binary
0xCAFEBABE        // A magic number used to identify Java class files
```

Integer literals are 32-bit `int` values unless they end with the character `L` or `l`, in which case they are 64-bit `long` values:

```
1234        // An int value
1234L       // A long value
0xffL       // Another long value
```

Integer arithmetic in Java never produces an overflow or an underflow when you exceed the range of a given integer type. Instead, numbers just wrap around. For example:

```
byte b1 = 127, b2 = 1;        // Largest byte is 127
byte sum = (byte)(b1 + b2);   // Sum wraps to -128, the smallest byte
```

Neither the Java compiler nor the Java interpreter warns you in any way when this occurs. When doing integer arithmetic, you simply must ensure that the type you are using has a sufficient range for the purposes you intend. Integer division by zero and modulo by zero are illegal and cause an `ArithmeticException` to be thrown.

Each integer type has a corresponding wrapper class: `Byte`, `Short`, `Integer`, and `Long`. Each of these classes defines `MIN_VALUE` and `MAX_VALUE` constants that describe the range of the type. The classes also define useful static methods, such as `Byte.parseByte()` and `Integer.parseInt()`, for converting strings to integer values.

Floating-Point Types

Real numbers in Java are represented by the `float` and `double` data types. As shown in Table 2-1, `float` is a 32-bit, single-precision floating-point value, and `double` is a 64-bit, double-precision floating-point value. Both types adhere to the IEEE 754-1985 standard, which specifies both the format of the numbers and the behavior of arithmetic for the numbers.

Floating-point values can be included literally in a Java program as an optional string of digits, followed by a decimal point and another string of digits. Here are some examples:

```
123.45
0.0
.01
```

Floating-point literals can also use exponential, or scientific, notation, in which a number is followed by the letter `e` or `E` (for exponent) and another number. This second number represents the power of 10 by which the first number is multiplied. For example:

```
1.2345E02     // 1.2345 * 10^2 or 123.45
1e-6          // 1 * 10^-6 or 0.000001
6.02e23       // Avogadro's Number: 6.02 * 10^23
```

Floating-point literals are `double` values by default. To include a `float` value literally in a program, follow the number with `f` or `F`:

```
double d = 6.02E23;
float f = 6.02e23f;
```

Floating-point literals cannot be expressed in hexadecimal, binary, or octal notation.

Floating-Point Representations

Most real numbers, by their very nature, cannot be represented exactly in any finite number of bits. Thus, it is important to remember that `float` and `double` values are only approximations of the numbers they are meant to represent. A `float` is a 32-bit approximation, which results in at least six significant decimal digits, and a `double` is a 64-bit approximation, which results in at least 15 significant digits. In Chapter 9, we will cover floating-point representations in more detail.

In addition to representing ordinary numbers, the `float` and `double` types can also represent four special values: positive and negative infinity, zero, and NaN. The infinity values result when a floating-point computation produces a value that overflows the representable range of a `float` or `double`. When a floating-point computation underflows the representable range of a `float` or a `double`, a zero value results.

The Java floating-point types make a distinction between positive zero and negative zero, depending on the direction from which the underflow occurred. In practice, positive and negative zero behave pretty much the same. Finally, the last special floating-point value is NaN, which stands for "Not-a-number." The NaN value results when an illegal floating-point operation, such as 0.0/0.0, is performed. Here are examples of statements that result in these special values:

```
double inf = 1.0/0.0;          // Infinity
double neginf = -1.0/0.0;      // Negative Infinity
double negzero = -1.0/inf;     // Negative zero
double NaN = 0.0/0.0;          // Not-a-number
```

Because the Java floating-point types can handle overflow to infinity and underflow to zero and have a special NaN value, floating-point arithmetic never throws exceptions, even when performing illegal operations, like dividing zero by zero or taking the square root of a negative number.

The `float` and `double` primitive types have corresponding classes, named `Float` and `Double`. Each of these classes defines the following useful constants: `MIN_VALUE`, `MAX_VALUE`, `NEGATIVE_INFINITY`, `POSITIVE_INFINITY`, and `NaN`.

The infinite floating-point values behave as you would expect. Adding or subtracting any finite value to or from infinity, for example, yields infinity. Negative zero behaves almost identically to positive zero, and, in fact, the `==` equality operator reports that negative zero is equal to positive zero. One way to distinguish negative zero from positive, or regular, zero is to divide by it: 1.0/0.0 yields positive infinity, but 1.0 divided by negative zero yields negative infinity. Finally, because NaN is Not-a-number, the `==` operator says that it is not equal to any other number, including itself! To check whether a `float` or `double` value is NaN, you must use the `Float.isNaN()` and `Double.isNaN()` methods.

Primitive Type Conversions

Java allows conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, `char` values can be converted to and from the integer and floating-point types. In fact, `boolean` is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two basic types of conversions. A *widening conversion* occurs when a value of one type is converted to a wider type—one that has a larger range of legal values. For example, Java performs widening conversions automatically when you assign an `int` literal to a `double` variable or a `char` literal to an `int` variable.

Narrowing conversions are another matter, however. A *narrowing conversion* occurs when a value is converted to a type that is not wider than it is. Narrowing conversions are not always safe: it is reasonable to convert the integer value 13 to a `byte`, for example, but it is not reasonable to convert 13,000 to a `byte`, because `byte` can hold only numbers between -128 and 127. Because you can lose data in a narrowing conversion, the Java compiler complains when you attempt any narrowing conversion, even if the value being converted would in fact fit in the narrower range of the specified type:

```
int i = 13;
byte b = i;    // The compiler does not allow this
```

The one exception to this rule is that you can assign an integer literal (an `int` value) to a `byte` or `short` variable if the literal falls within the range of the variable.

If you need to perform a narrowing conversion and are confident you can do so without losing data or precision, you can force Java to perform the conversion using

a language construct known as a *cast*. Perform a cast by placing the name of the desired type in parentheses before the value to be converted. For example:

```
int i = 13;
byte b = (byte) i;   // Force the int to be converted to a byte
i = (int) 13.456;    // Force this double literal to the int 13
```

Casts of primitive types are most often used to convert floating-point values to integers. When you do this, the fractional part of the floating-point value is simply truncated (i.e., the floating-point value is rounded toward zero, not toward the nearest integer). The static methods `Math.round()`, `Math.floor()`, and `Math.ceil()` perform other types of rounding.

The `char` type acts like an integer type in most ways, so a `char` value can be used anywhere an `int` or `long` value is required. Recall, however, that the `char` type is *unsigned*, so it behaves differently than the `short` type, even though both are 16 bits wide:

```
short s = (short) 0xffff; // These bits represent the number -1
char c = '\uffff';        // The same bits, as a Unicode character
int i1 = s;               // Converting the short to an int yields -1
int i2 = c;               // Converting the char to an int yields 65535
```

Table 2-3 shows which primitive types can be converted to which other types and how the conversion is performed. The letter N in the table means that the conversion cannot be performed. The letter Y means that the conversion is a widening conversion and is therefore performed automatically and implicitly by Java. The letter C means that the conversion is a narrowing conversion and requires an explicit cast.

Finally, the notation Y* means that the conversion is an automatic widening conversion, but that some of the least significant digits of the value may be lost in the conversion. This can happen when converting an `int` or `long` to a floating-point type—see the table for details. The floating-point types have a larger range than the integer types, so any `int` or `long` can be represented by a `float` or `double`. However, the floating-point types are approximations of numbers and cannot always hold as many significant digits as the integer types (see Chapter 9 for some more detail about floating-point numbers).

Table 2-3. Java primitive type conversions

	Convert to:							
Convert from:	boolean	byte	short	char	int	long	float	double
boolean	-	N	N	N	N	N	N	N
byte	N	-	Y	C	Y	Y	Y	Y
short	N	C	-	C	Y	Y	Y	Y

Convert from:	Convert to: boolean	byte	short	char	int	long	float	double
char	N	C	C	-	Y	Y	Y	Y
int	N	C	C	C	-	Y	Y*	Y
long	N	C	C	C	C	-	Y*	Y*
float	N	C	C	C	C	C	-	Y
double	N	C	C	C	C	C	C	-

Expressions and Operators

So far in this chapter, we've learned about the primitive types that Java programs can manipulate and seen how to include primitive values as *literals* in a Java program. We've also used *variables* as symbolic names that represent, or hold, values. These literals and variables are the tokens out of which Java programs are built.

An *expression* is the next higher level of structure in a Java program. The Java interpreter *evaluates* an expression to compute its value. The very simplest expressions are called *primary expressions* and consist of literals and variables. So, for example, the following are all expressions:

```
1.7         // A floating-point literal
true        // A Boolean literal
sum         // A variable
```

When the Java interpreter evaluates a literal expression, the resulting value is the literal itself. When the interpreter evaluates a variable expression, the resulting value is the value stored in the variable.

Primary expressions are not very interesting. More complex expressions are made by using *operators* to combine primary expressions. For example, the following expression uses the assignment operator to combine two primary expressions—a variable and a floating-point literal—into an assignment expression:

```
sum = 1.7
```

But operators are used not only with primary expressions; they can also be used with expressions at any level of complexity. The following are all legal expressions:

```
sum = 1 + 2 + 3 * 1.2 + (4 + 8)/3.0
sum/Math.sqrt(3.0 * 1.234)
(int)(sum + 33)
```

Operator Summary

The kinds of expressions you can write in a programming language depend entirely on the set of operators available to you. Java has a wealth of operators, but to work

effectively with them, there are two important concepts that need to be understood: *precedence* and *associativity*. These concepts—and the operators themselves—are explained in more detail in the following sections.

Precedence

The **P** column of Table 2-4 specifies the *precedence* of each operator. Precedence specifies the order in which operations are performed. Operations that have higher precedence are performed before those with lower precedence. For example, consider this expression:

```
a + b * c
```

The multiplication operator has higher precedence than the addition operator, so `a` is added to the product of `b` and `c`, just as we expect from elementary mathematics. Operator precedence can be thought of as a measure of how tightly operators bind to their operands. The higher the number, the more tightly they bind.

Default operator precedence can be overridden through the use of parentheses that explicitly specify the order of operations. The previous expression can be rewritten to specify that the addition should be performed before the multiplication:

```
(a + b) * c
```

The default operator precedence in Java was chosen for compatibility with C; the designers of C chose this precedence so that most expressions can be written naturally without parentheses. There are only a few common Java idioms for which parentheses are required. Examples include:

```
// Class cast combined with member access
((Integer) o).intValue();

// Assignment combined with comparison
while((line = in.readLine()) != null) { ... }

// Bitwise operators combined with comparison
if ((flags & (PUBLIC | PROTECTED)) != 0) { ... }
```

Associativity

Associativity is a property of operators that defines how to evaluate expressions that would otherwise be ambiguous. This is particularly important when an expression involves several operators that have the same precedence.

Most operators are left-to-right associative, which means that the operations are performed from left to right. The assignment and unary operators, however, have right-to-left associativity. The **A** column of Table 2-4 specifies the associativity of each operator or group of operators. The value L means left to right, and R means right to left.

The additive operators are all left-to-right associative, so the expression `a+b-c` is evaluated from left to right: `(a+b)-c`. Unary operators and assignment operators are evaluated from right to left. Consider this complex expression:

```
a = b += c = -~d
```

This is evaluated as follows:

```
a = (b += (c = -(~d)))
```

As with operator precedence, operator associativity establishes a default order of evaluation for an expression. This default order can be overridden through the use of parentheses. However, the default operator associativity in Java has been chosen to yield a natural expression syntax, and you should rarely need to alter it.

Operator summary table

Table 2-4 summarizes the operators available in Java. The P and A columns of the table specify the precedence and associativity of each group of related operators, respectively. You should use this table as a quick reference for operators (especially their precedence) when required.

Table 2-4. Java operators

P	A	Operator	Operand type(s)	Operation performed
16	L	.	object, member	Object member access
		[ ]	array, int	Array element access
		(*args*)	method, arglist	Method invocation
		++, --	variable	Post-increment, post-decrement
15	R	++, --	variable	Pre-increment, pre-decrement
		+, -	number	Unary plus, unary minus
		~	integer	Bitwise complement
		!	boolean	Boolean NOT
14	R	new	class, arglist	Object creation
		(*type*)	type, any	Cast (type conversion)
13	L	*, /, %	number, number	Multiplication, division, remainder
12	L	+, -	number, number	Addition, subtraction

P	A	Operator	Operand type(s)	Operation performed
		+	string, any	String concatenation
11	L	<<	integer, integer	Left shift
		>>	integer, integer	Right shift with sign extension
		>>>	integer, integer	Right shift with zero extension
10	L	<, <=	number, number	Less than, less than or equal
		>, >=	number, number	Greater than, greater than or equal
		instanceof	reference, type	Type comparison
9	L	==	primitive, primitive	Equal (have identical values)
		!=	primitive, primitive	Not equal (have different values)
		==	reference, reference	Equal (refer to same object)
		!=	reference, reference	Not equal (refer to different objects)
8	L	&	integer, integer	Bitwise AND
		&	boolean, boolean	Boolean AND
7	L	^	integer, integer	Bitwise XOR
		^	boolean, boolean	Boolean XOR
6	L	|	integer, integer	Bitwise OR
		|	boolean, boolean	Boolean OR
5	L	&&	boolean, boolean	Conditional AND
4	L	||	boolean, boolean	Conditional OR
3	R	? :	boolean, any	Conditional (ternary) operator
2	R	=	variable, any	Assignment
		*=, /=, %=,	variable, any	Assignment with operation

P	A	Operator	Operand type(s)	Operation performed
		+=, -=, <<=,		
		>>=, >>>=,		
		&=, ^=, |=		
1	R	->	arglist, method body	lambda expression

Operand number and type

The fourth column of Table 2-4 specifies the number and type of the operands expected by each operator. Some operators operate on only one operand; these are called unary operators. For example, the unary minus operator changes the sign of a single number:

```
-n              // The unary minus operator
```

Most operators, however, are binary operators that operate on two operand values. The - operator actually comes in both forms:

```
a - b           // The subtraction operator is a binary operator
```

Java also defines one ternary operator, often called the conditional operator. It is like an `if` statement inside an expression. Its three operands are separated by a question mark and a colon; the second and third operands must be convertible to the same type:

```
x > y ? x : y  // Ternary expression; evaluates to larger of x and y
```

In addition to expecting a certain number of operands, each operator also expects particular types of operands. The fourth column of the table lists the operand types. Some of the codes used in that column require further explanation:

Number
: An integer, floating-point value, or character (i.e., any primitive type except `boolean`). Autounboxing (see "Boxing and Unboxing Conversions" on page 87) means that the wrapper classes (such as `Character`, `Integer`, and `Double`) for these types can be used in this context as well.

Integer
: A `byte`, `short`, `int`, `long`, or `char` value (`long` values are not allowed for the array access operator `[ ]`). With autounboxing, `Byte`, `Short`, `Integer`, `Long`, and `Character` values are also allowed.

Reference
: An object or array.

Variable
: A variable or anything else, such as an array element, to which a value can be assigned.

Return type

Just as every operator expects its operands to be of specific types, each operator produces a value of a specific type. The arithmetic, increment and decrement, bitwise, and shift operators return a `double` if at least one of the operands is a `double`. They return a `float` if at least one of the operands is a `float`. They return a `long` if at least one of the operands is a `long`. Otherwise, they return an `int`, even if both operands are `byte`, `short`, or `char` types that are narrower than `int`.

The comparison, equality, and Boolean operators always return `boolean` values. Each assignment operator returns whatever value it assigned, which is of a type compatible with the variable on the left side of the expression. The conditional operator returns the value of its second or third argument (which must both be of the same type).

Side effects

Every operator computes a value based on one or more operand values. Some operators, however, have *side effects* in addition to their basic evaluation. If an expression contains side effects, evaluating it changes the state of a Java program in such a way that evaluating the expression again may yield a different result.

For example, the `++` increment operator has the side effect of incrementing a variable. The expression `++a` increments the variable `a` and returns the newly incremented value. If this expression is evaluated again, the value will be different. The various assignment operators also have side effects. For example, the expression `a*=2` can also be written as `a=a*2`. The value of the expression is the value of `a` multiplied by 2, but the expression has the side effect of storing that value back into `a`.

The method invocation operator `()` has side effects if the invoked method has side effects. Some methods, such as `Math.sqrt()`, simply compute and return a value without side effects of any kind. Typically, however, methods do have side effects. Finally, the `new` operator has the profound side effect of creating a new object.

Order of evaluation

When the Java interpreter evaluates an expression, it performs the various operations in an order specified by the parentheses in the expression, the precedence of the operators, and the associativity of the operators. Before any operation is performed, however, the interpreter first evaluates the operands of the operator. (The exceptions are the `&&`, `||`, and `?:` operators, which do not always evaluate all their operands.) The interpreter always evaluates operands in order from left to right. This matters if any of the operands are expressions that contain side effects. Consider this code, for example:

```
int a = 2;
int v = ++a + ++a * ++a;
```

Although the multiplication is performed before the addition, the operands of the + operator are evaluated first. As the operands of ++ are both a, these are evaluated to 3 and 4, and so the expression evaluates to `3 + 4 * 5`, or 23.

Arithmetic Operators

The arithmetic operators can be used with integers, floating-point numbers, and even characters (i.e., they can be used with any primitive type other than `boolean`). If either of the operands is a floating-point number, floating-point arithmetic is used; otherwise, integer arithmetic is used. This matters because integer arithmetic and floating-point arithmetic differ in the way division is performed and in the way underflows and overflows are handled, for example. The arithmetic operators are:

Addition (+)
: The + operator adds two numbers. As we'll see shortly, the + operator can also be used to concatenate strings. If either operand of + is a string, the other one is converted to a string as well. Be sure to use parentheses when you want to combine addition with concatenation. For example:

```
System.out.println("Total: " + 3 + 4);    // Prints "Total: 34", not 7!
```

Subtraction (-)
: When the - operator is used as a binary operator, it subtracts its second operand from its first. For example, 7-3 evaluates to 4. The - operator can also perform unary negation.

Multiplication (*)
: The * operator multiplies its two operands. For example, 7*3 evaluates to 21.

Division (/)
: The / operator divides its first operand by its second. If both operands are integers, the result is an integer, and any remainder is lost. If either operand is a floating-point value, however, the result is a floating-point value. When dividing two integers, division by zero throws an `ArithmeticException`. For floating-point calculations, however, division by zero simply yields an infinite result or NaN:

```
7/3          // Evaluates to 2
7/3.0f       // Evaluates to 2.333333f
7/0          // Throws an ArithmeticException
7/0.0        // Evaluates to positive infinity
0.0/0.0      // Evaluates to NaN
```

Modulo (%)
: The % operator computes the first operand modulo the second operand (i.e., it returns the remainder when the first operand is divided by the second operand an integral number of times). For example, 7%3 is 1. The sign of the result is

the same as the sign of the first operand. While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, 4.3%2.1 evaluates to 0.1. When operating with integers, trying to compute a value modulo zero causes an `ArithmeticException`. When working with floating-point values, anything modulo 0.0 evaluates to `NaN`, as does infinity modulo anything.

Unary minus (-)
: When the - operator is used as a unary operator—that is, before a single operand—it performs unary negation. In other words, it converts a positive value to an equivalently negative value, and vice versa.

String Concatenation Operator

In addition to adding numbers, the + operator (and the related += operator) also concatenates, or joins, strings. If either of the operands to + is a string, the operator converts the other operand to a string. For example:

```
// Prints "Quotient: 2.3333333"
System.out.println("Quotient: " + 7/3.0f);
```

As a result, you must be careful to put any addition expressions in parentheses when combining them with string concatenation. If you do not, the addition operator is interpreted as a concatenation operator.

The Java interpreter has built-in string conversions for all primitive types. An object is converted to a string by invoking its `toString()` method. Some classes define custom `toString()` methods so that objects of that class can easily be converted to strings in this way. An array is converted to a string by invoking the built-in `toString()` method, which, unfortunately, does not return a useful string representation of the array contents.

Increment and Decrement Operators

The ++ operator increments its single operand, which must be a variable, an element of an array, or a field of an object, by 1. The behavior of this operator depends on its position relative to the operand. When used before the operand, where it is known as the *pre-increment* operator, it increments the operand and evaluates to the incremented value of that operand. When used after the operand, where it is known as the *post-increment* operator, it increments its operand, but evaluates to the value of that operand before it was incremented.

For example, the following code sets both `i` and `j` to 2:

```
i = 1;
j = ++i;
```

But these lines set `i` to 2 and `j` to 1:

```
i = 1;
j = i++;
```

Similarly, the `--` operator decrements its single numeric operand, which must be a variable, an element of an array, or a field of an object, by one. Like the `++` operator, the behavior of `--` depends on its position relative to the operand. When used before the operand, it decrements the operand and returns the decremented value. When used after the operand, it decrements the operand, but returns the *undecremented* value.

The expressions `x++` and `x--` are equivalent to `x=x+1` and `x=x-1`, respectively, except that when using the increment and decrement operators, `x` is only evaluated once. If `x` is itself an expression with side effects, this makes a big difference. For example, these two expressions are not equivalent:

```
a[i++]++;              // Increments an element of an array
// Adds 1 to an array element and stores new value in another element
a[i++] = a[i++] + 1;
```

These operators, in both prefix and postfix forms, are most commonly used to increment or decrement the counter that controls a loop.

Comparison Operators

The comparison operators consist of the equality operators that test values for equality or inequality and the relational operators used with ordered types (numbers and characters) to test for greater than and less than relationships. Both types of operators yield a `boolean` result, so they are typically used with `if` statements and `while` and `for` loops to make branching and looping decisions. For example:

```
if (o != null) ...;           // The not equals operator
while(i < a.length) ...;      // The less than operator
```

Java provides the following equality operators:

Equals (==)

The `==` operator evaluates to `true` if its two operands are equal and `false` otherwise. With primitive operands, it tests whether the operand values themselves are identical. For operands of reference types, however, it tests whether the operands refer to the same object or array. In other words, it does not test the equality of two distinct objects or arrays. In particular, note that you cannot test two distinct strings for equality with this operator.

If `==` is used to compare two numeric or character operands that are not of the same type, the narrower operand is converted to the type of the wider operand before the comparison is done. For example, when comparing a `short` to a `float`, the `short` is first converted to a `float` before the comparison is performed. For floating-point numbers, the special negative zero value tests equal to the regular, positive zero value. Also, the special NaN (Not-a-number) value is not equal to any other number, including itself. To test whether a floating-point value is NaN, use the `Float.isNan()` or `Double.isNan()` method.

Not equals (`!=`)
: The `!=` operator is exactly the opposite of the `==` operator. It evaluates to `true` if its two primitive operands have different values or if its two reference operands refer to different objects or arrays. Otherwise, it evaluates to `false`.

  The relational operators can be used with numbers and characters, but not with `boolean` values, objects, or arrays because those types are not ordered. Java provides the following relational operators:

Less than (`<`)
: Evaluates to `true` if the first operand is less than the second.

Less than or equal (`<=`)
: Evaluates to `true` if the first operand is less than or equal to the second.

Greater than (`>`)
: Evaluates to `true` if the first operand is greater than the second.

Greater than or equal (`>=`)
: Evaluates to `true` if the first operand is greater than or equal to the second.

Boolean Operators

As we've just seen, the comparison operators compare their operands and yield a `boolean` result, which is often used in branching and looping statements. In order to make branching and looping decisions based on conditions more interesting than a single comparison, you can use the Boolean (or logical) operators to combine multiple comparison expressions into a single, more complex expression. The Boolean operators require their operands to be `boolean` values and they evaluate to `boolean` values. The operators are:

Conditional AND (`&&`)
: This operator performs a Boolean AND operation on its operands. It evaluates to `true` if and only if both its operands are `true`. If either or both operands are `false`, it evaluates to `false`. For example:

```
if (x < 10 && y > 3) ... // If both comparisons are true
```

  This operator (and all the Boolean operators except the unary `!` operator) have a lower precedence than the comparison operators. Thus, it is perfectly legal to write a line of code like the one just shown. However, some programmers prefer to use parentheses to make the order of evaluation explicit:

```
if ((x < 10) && (y > 3)) ...
```

  You should use whichever style you find easier to read.

  This operator is called a conditional AND because it conditionally evaluates its second operand. If the first operand evaluates to `false`, the value of the expression is `false`, regardless of the value of the second operand. Therefore, to

increase efficiency, the Java interpreter takes a shortcut and skips the second operand. The second operand is not guaranteed to be evaluated, so you must use caution when using this operator with expressions that have side effects. On the other hand, the conditional nature of this operator allows us to write Java expressions such as the following:

```
if (data != null && i < data.length && data[i] != -1)
    ...
```

The second and third comparisons in this expression would cause errors if the first or second comparisons evaluated to `false`. Fortunately, we don't have to worry about this because of the conditional behavior of the `&&` operator.

Conditional OR (`||`)

This operator performs a Boolean OR operation on its two `boolean` operands. It evaluates to `true` if either or both of its operands are `true`. If both operands are `false`, it evaluates to `false`. Like the `&&` operator, `||` does not always evaluate its second operand. If the first operand evaluates to `true`, the value of the expression is `true`, regardless of the value of the second operand. Thus, the operator simply skips the second operand in that case.

Boolean NOT (`!`)

This unary operator changes the `boolean` value of its operand. If applied to a `true` value, it evaluates to `false`, and if applied to a `false` value, it evaluates to `true`. It is useful in expressions like these:

```
if (!found) ...          // found is a boolean declared somewhere
while (!c.isEmpty()) ... // The isEmpty() method returns a boolean
```

Because `!` is a unary operator, it has a high precedence and often must be used with parentheses:

```
if (!(x > y && y > z))
```

Boolean AND (`&`)

When used with `boolean` operands, the `&` operator behaves like the `&&` operator, except that it always evaluates both operands, regardless of the value of the first operand. This operator is almost always used as a bitwise operator with integer operands, however, and many Java programmers would not even recognize its use with `boolean` operands as legal Java code.

Boolean OR (`|`)

This operator performs a Boolean OR operation on its two `boolean` operands. It is like the `||` operator, except that it always evaluates both operands, even if the first one is `true`. The `|` operator is almost always used as a bitwise operator on integer operands; its use with `boolean` operands is very rare.

Boolean XOR (`^`)

When used with `boolean` operands, this operator computes the exclusive OR (XOR) of its operands. It evaluates to `true` if exactly one of the two operands is

`true`. In other words, it evaluates to `false` if both operands are `false` or if both operands are `true`. Unlike the `&&` and `||` operators, this one must always evaluate both operands. The `^` operator is much more commonly used as a bitwise operator on integer operands. With `boolean` operands, this operator is equivalent to the `!=` operator.

Bitwise and Shift Operators

The bitwise and shift operators are low-level operators that manipulate the individual bits that make up an integer value. The bitwise operators are not commonly used in modern Java except for low-level work (e.g., network programming). They are used for testing and setting individual flag bits in a value. In order to understand their behavior, you must understand binary (base-2) numbers and the two's complement format used to represent negative integers.

You cannot use these operators with floating-point, `boolean`, array, or object operands. When used with `boolean` operands, the `&`, `|`, and `^` operators perform a different operation, as described in the previous section.

If either of the arguments to a bitwise operator is a `long`, the result is a `long`. Otherwise, the result is an `int`. If the left operand of a shift operator is a `long`, the result is a `long`; otherwise, the result is an `int`. The operators are:

Bitwise complement (~)
: The unary `~` operator is known as the bitwise complement, or bitwise NOT, operator. It inverts each bit of its single operand, converting 1s to 0s and 0s to 1s. For example:

```
byte b = ~12;           // ~00001100 =  => 11110011 or -13 decimal
flags = flags & ~f;     // Clear flag f in a set of flags
```

Bitwise AND (&)
: This operator combines its two integer operands by performing a Boolean AND operation on their individual bits. The result has a bit set only if the corresponding bit is set in both operands. For example:

```
10 & 7                  // 00001010 & 00000111 =  => 00000010 or 2
if ((flags & f) != 0)   // Test whether flag f is set
```

When used with `boolean` operands, `&` is the infrequently used Boolean AND operator described earlier.

Bitwise OR (|)
: This operator combines its two integer operands by performing a Boolean OR operation on their individual bits. The result has a bit set if the corresponding bit is set in either or both of the operands. It has a zero bit only where both corresponding operand bits are zero. For example:

```
10 | 7                  // 00001010 | 00000111 =  => 00001111 or 15
flags = flags | f;      // Set flag f
```

When used with `boolean` operands, `|` is the infrequently used Boolean OR operator described earlier.

Bitwise XOR (^)
: This operator combines its two integer operands by performing a Boolean XOR (exclusive OR) operation on their individual bits. The result has a bit set if the corresponding bits in the two operands are different. If the corresponding operand bits are both 1s or both 0s, the result bit is a 0. For example:

```
10 ^ 7              // 00001010 ^ 00000111 =  => 00001101 or 13
```

When used with `boolean` operands, `^` is the seldom used Boolean XOR operator.

Left shift (<<)
: The `<<` operator shifts the bits of the left operand left by the number of places specified by the right operand. High-order bits of the left operand are lost, and zero bits are shifted in from the right. Shifting an integer left by *n* places is equivalent to multiplying that number by 2^n. For example:

```
10 << 1    // 00001010 << 1 = 00010100 = 20 = 10*2
7 << 3     // 00000111 << 3 = 00111000 = 56 = 7*8
-1 << 2    // 0xFFFFFFFF << 2 = 0xFFFFFFFC = -4 = -1*4
```

If the left operand is a `long`, the right operand should be between 0 and 63. Otherwise, the left operand is taken to be an `int`, and the right operand should be between 0 and 31.

Signed right shift (>>)
: The `>>` operator shifts the bits of the left operand to the right by the number of places specified by the right operand. The low-order bits of the left operand are shifted away and are lost. The high-order bits shifted in are the same as the original high-order bit of the left operand. In other words, if the left operand is positive, 0s are shifted into the high-order bits. If the left operand is negative, 1s are shifted in instead. This technique is known as *sign extension*; it is used to preserve the sign of the left operand. For example:

```
10 >> 1      // 00001010 >> 1 = 00000101 = 5 = 10/2
27 >> 3      // 00011011 >> 3 = 00000011 = 3 = 27/8
-50 >> 2     // 11001110 >> 2 = 11110011 = -13 != -50/4
```

If the left operand is positive and the right operand is *n*, the `>>` operator is the same as integer division by 2^n.

Unsigned right shift (>>>)
: This operator is like the `>>` operator, except that it always shifts zeros into the high-order bits of the result, regardless of the sign of the left-hand operand. This technique is called *zero extension*; it is appropriate when the left operand is being treated as an unsigned value (despite the fact that Java integer types are all signed). These are examples:

```
0xff >>> 4     // 11111111 >>> 4 = 00001111 = 15  = 255/16
-50 >>> 2      // 0xFFFFFFCE >>> 2 = 0x3FFFFFF3 = 1073741811
```

Assignment Operators

The assignment operators store, or assign, a value into some kind of variable. The left operand must evaluate to an appropriate local variable, array element, or object field. The right side can be any value of a type compatible with the variable. An assignment expression evaluates to the value that is assigned to the variable. More importantly, however, the expression has the side effect of actually performing the assignment. Unlike all other binary operators, the assignment operators are right-associative, which means that the assignments in `a=b=c` are performed right to left, as follows: `a=(b=c)`.

The basic assignment operator is `=`. Do not confuse it with the equality operator, `==`. In order to keep these two operators distinct, we recommend that you read `=` as "is assigned the value."

In addition to this simple assignment operator, Java also defines 11 other operators that combine assignment with the 5 arithmetic operators and the 6 bitwise and shift operators. For example, the `+=` operator reads the value of the left variable, adds the value of the right operand to it, stores the sum back into the left variable as a side effect, and returns the sum as the value of the expression. Thus, the expression `x+=2` is almost the same as `x=x+2`. The difference between these two expressions is that when you use the `+=` operator, the left operand is evaluated only once. This makes a difference when that operand has a side effect. Consider the following two expressions, which are not equivalent:

```
a[i++] += 2;
a[i++] = a[i++] + 2;
```

The general form of these combination assignment operators is:

```
var op= value
```

This is equivalent (unless there are side effects in `var`) to:

```
var = var op value
```

The available operators are:

```
+=    -=    *=    /=    %=    // Arithmetic operators plus assignment

&=    |=    ^=                // Bitwise operators plus assignment

<<=   >>=   >>>=              // Shift operators plus assignment
```

The most commonly used operators are `+=` and `-=`, although `&=` and `|=` can also be useful when working with `boolean` flags. For example:

```
i += 2;          // Increment a loop counter by 2
c -= 5;          // Decrement a counter by 5
```

```
flags |= f;      // Set a flag f in an integer set of flags
flags &= ~f;     // Clear a flag f in an integer set of flags
```

The Conditional Operator

The conditional operator `?:` is a somewhat obscure ternary (three-operand) operator inherited from C. It allows you to embed a conditional within an expression. You can think of it as the operator version of the `if/else` statement. The first and second operands of the conditional operator are separated by a question mark (`?`) while the second and third operands are separated by a colon (:). The first operand must evaluate to a `boolean` value. The second and third operands can be of any type, but they must be convertible to the same type.

The conditional operator starts by evaluating its first operand. If it is `true`, the operator evaluates its second operand and uses that as the value of the expression. On the other hand, if the first operand is `false`, the conditional operator evaluates and returns its third operand. The conditional operator never evaluates both its second and third operand, so be careful when using expressions with side effects with this operator. Examples of this operator are:

```
int max = (x > y) ? x : y;
String name = (name != null) ? name : "unknown";
```

Note that the `?:` operator has lower precedence than all other operators except the assignment operators, so parentheses are not usually necessary around the operands of this operator. Many programmers find conditional expressions easier to read if the first operand is placed within parentheses, however. This is especially true because the conditional `if` statement always has its conditional expression written within parentheses.

The instanceof Operator

The `instanceof` operator is intimately bound up with objects and the operation of the Java type system. If this is your first look at Java, it may be preferable to skim this definition and return to this section after you have a decent grasp of Java's objects.

`instanceof` requires an object or array value as its left operand and the name of a reference type as its right operand. It evaluates to `true` if the object or array is an *instance* of the specified type; it returns `false` otherwise. If the left operand is `null`, `instanceof` always evaluates to `false`. If an `instanceof` expression evaluates to `true`, it means that you can safely cast and assign the left operand to a variable of the type of the right operand.

The `instanceof` operator can be used only with reference types and objects, not primitive types and values. Examples of `instanceof` are:

```
// True: all strings are instances of String
"string" instanceof String
// True: strings are also instances of Object
```

```
"" instanceof Object
// False: null is never an instance of anything
null instanceof String

Object o = new int[] {1,2,3};
o instanceof int[]   // True: the array value is an int array
o instanceof byte[]  // False: the array value is not a byte array
o instanceof Object  // True: all arrays are instances of Object

// Use instanceof to make sure that it is safe to cast an object
if (object instanceof Point) {
   Point p = (Point) object;
}
```


Special Operators

Java has six language constructs that are sometimes considered operators and sometimes considered simply part of the basic language syntax. These "operators" were included in Table 2-4 in order to show their precedence relative to the other true operators. The use of these language constructs is detailed elsewhere in this book, but is described briefly here so that you can recognize them in code examples:

Object member access (`.`)
: An *object* is a collection of data and methods that operate on that data; the data fields and methods of an object are called its members. The dot (.) operator accesses these members. If `o` is an expression that evaluates to an object reference, and `f` is the name of a field of the object, `o.f` evaluates to the value contained in that field. If `m` is the name of a method, `o.m` refers to that method and allows it to be invoked using the `()` operator shown later.

Array element access (`[]`)
: An *array* is a numbered list of values. Each element of an array can be referred to by its number, or *index*. The `[ ]` operator allows you to refer to the individual elements of an array. If `a` is an array, and `i` is an expression that evaluates to an `int`, `a[i]` refers to one of the elements of `a`. Unlike other operators that work with integer values, this operator restricts array index values to be of type `int` or narrower.

Method invocation (`()`)
: A *method* is a named collection of Java code that can be run, or *invoked*, by following the name of the method with zero or more comma-separated expressions contained within parentheses. The values of these expressions are the *arguments* to the method. The method processes the arguments and optionally returns a value that becomes the value of the method invocation expression. If `o.m` is a method that expects no arguments, the method can be invoked with `o.m()`. If the method expects three arguments, for example, it can be invoked with an expression such as `o.m(x,y,z)`. Before the Java interpreter invokes a method, it evaluates each of the arguments to be passed to the method. These

expressions are guaranteed to be evaluated in order from left to right (which matters if any of the arguments have side effects).

Lambda expression (->)
: A *lambda expression* is an anonymous collection of executable Java code, essentially a method body. It consists of a method argument list (zero or more comma-separated expressions contained within parentheses) followed by the lambda *arrow* operator followed by a block of Java code. If the block of code comprises just a single statement, then the usual curly braces to denote block boundaries can be omitted.

Object creation (`new`)
: In Java, objects (and arrays) are created with the `new` operator, which is followed by the type of the object to be created and a parenthesized list of arguments to be passed to the object *constructor*. A constructor is a special block of code that initializes a newly created object, so the object creation syntax is similar to the Java method invocation syntax. For example:

```
new ArrayList();
new Point(1,2)
```

Type conversion or casting (`()`)
: As we've already seen, parentheses can also be used as an operator to perform narrowing type conversions, or casts. The first operand of this operator is the type to be converted to; it is placed between the parentheses. The second operand is the value to be converted; it follows the parentheses. For example:

```
(byte) 28          // An integer literal cast to a byte type
(int) (x + 3.14f)  // A floating-point sum value cast to an integer
(String)h.get(k)   // A generic object cast to a string
```

Statements

A *statement* is a basic unit of execution in the Java language—it expresses a single piece of intent by the programmer. Unlike expressions, Java statements do not have a value. Statements also typically contain expressions and operators (especially assignment operators) and are frequently executed for the side effects that they cause.

Many of the statements defined by Java are flow-control statements, such as conditionals and loops, that can alter the default, linear order of execution in well-defined ways. Table 2-5 summarizes the statements defined by Java.

Table 2-5. Java statements

Statement	Purpose	Syntax
expression	side effects	*var* = *expr* ; *expr* ++; *method* (); new *Type* ( );
compound	group statements	{ *statements* }
empty	do nothing	;
labeled	name a statement	*label* : *statement*
variable	declare a variable	[final] *type name* [= *value*] [, *name* [= *value*]] …;
if	conditional	if (*expr*) *statement* [else *statement*]
switch	conditional	switch (*expr*) { [case *expr* : *statements*] … [default: *statements*] }
while	loop	while (*expr*) *statement*
do	loop	do *statement* while (*expr*);
for	simplified loop	for (*init* ; *test* ; *increment*) *statement*
foreach	collection iteration	for (*variable* : *iterable*) *statement*
break	exit block	break [*label*] ;
continue	restart loop	continue [*label*] ;
return	end method	return [*expr*] ;
synchronized	critical section	synchronized (*expr*) { *statements* }
throw	throw exception	throw *expr* ;
try	handle exception	try { *statements* } [catch (*type name*) { *statements* }] ...[finally { *statements* }]
assert	verify invariant	assert *invariant* [: *error*];

Expression Statements

As we saw earlier in the chapter, certain types of Java expressions have side effects. In other words, they do not simply evaluate to some value; they also change the

program state in some way. Any expression with side effects can be used as a statement simply by following it with a semicolon. The legal types of expression statements are assignments, increments and decrements, method calls, and object creation. For example:

```
a = 1;                              // Assignment
x *= 2;                             // Assignment with operation
i++;                                // Post-increment
--c;                                // Pre-decrement
System.out.println("statement");    // Method invocation
```

Compound Statements

A *compound statement* is any number and kind of statements grouped together within curly braces. You can use a compound statement anywhere a statement is required by Java syntax:

```
for(int i = 0; i < 10; i++) {
   a[i]++;           // Body of this loop is a compound statement.
   b[i]--;           // It consists of two expression statements
}                    // within curly braces.
```

The Empty Statement

An *empty statement* in Java is written as a single semicolon. The empty statement doesn't do anything, but the syntax is occasionally useful. For example, you can use it to indicate an empty loop body in a `for` loop:

```
for(int i = 0; i < 10; a[i++]++)  // Increment array elements
     /* empty */;                 // Loop body is empty statement
```

Labeled Statements

A *labeled statement* is simply a statement that has been given a name by prepending an identifier and a colon to it. Labels are used by the `break` and `continue` statements. For example:

```
rowLoop: for(int r = 0; r < rows.length; r++) {        // Labeled loop
   colLoop: for(int c = 0; c < columns.length; c++) {  // Another one
     break rowLoop;                                    // Use a label
   }
}
```

Local Variable Declaration Statements

A *local variable*, often simply called a variable, is a symbolic name for a location to store a value that is defined within a method or compound statement. All variables must be declared before they can be used; this is done with a variable declaration statement. Because Java is a statically typed language, a variable declaration specifies the type of the variable, and only values of that type can be stored in the variable.

In its simplest form, a variable declaration specifies a variable's type and name:

```
int counter;
String s;
```

A variable declaration can also include an *initializer*: an expression that specifies an initial value for the variable. For example:

```
int i = 0;
String s = readLine();
int[] data = {x+1, x+2, x+3}; // Array initializers are discussed later
```

The Java compiler does not allow you to use a local variable that has not been initialized, so it is usually convenient to combine variable declaration and initialization into a single statement. The initializer expression need not be a literal value or a constant expression that can be evaluated by the compiler; it can be an arbitrarily complex expression whose value is computed when the program is run.

A single variable declaration statement can declare and initialize more than one variable, but all variables must be of the same type. Variable names and optional initializers are separated from each other with commas:

```
int i, j, k;
float x = 1.0f, y = 1.0f;
String question = "Really Quit?", response;
```

Variable declaration statements can begin with the `final` keyword. This modifier specifies that once an initial value is specified for the variable, that value is never allowed to change:

```
final String greeting = getLocalLanguageGreeting();
```

We will have more to say about the `final` keyword later on, especially when talking about the immutable style of programming.

C programmers should note that Java variable declaration statements can appear anywhere in Java code; they are not restricted to the beginning of a method or block of code. Local variable declarations can also be integrated with the *initialize* portion of a `for` loop, as we'll discuss shortly.

Local variables can be used only within the method or block of code in which they are defined. This is called their *scope* or *lexical scope*:

```
void method() {            // A method definition
   int i = 0;              // Declare variable i
   while (i < 10) {        // i is in scope here
     int j = 0;            // Declare j; the scope of j begins here
     i++;                  // i is in scope here; increment it
   }                       // j is no longer in scope;
   System.out.println(i);  // i is still in scope here
}                          // The scope of i ends here
```

The if/else Statement

The `if` statement is a fundamental control statement that allows Java to make decisions or, more precisely, to execute statements conditionally. The `if` statement has an associated expression and statement. If the expression evaluates to `true`, the interpreter executes the statement. If the expression evaluates to `false`, the interpreter skips the statement.

Java allows the expression to be of the wrapper type `Boolean` instead of the primitive type `boolean`. In this case, the wrapper object is automatically unboxed.

Here is an example `if` statement:

```
if (username == null)          // If username is null,
   username = "John Doe";      // use a default value
```

Although they look extraneous, the parentheses around the expression are a required part of the syntax for the `if` statement. As we already saw, a block of statements enclosed in curly braces is itself a statement, so we can write `if` statements that look like this as well:

```
if ((address == null) || (address.equals(""))) {
   address = "[undefined]";
   System.out.println("WARNING: no address specified.");
}
```

An `if` statement can include an optional `else` keyword that is followed by a second statement. In this form of the statement, the expression is evaluated, and, if it is `true`, the first statement is executed. Otherwise, the second statement is executed. For example:

```
if (username != null)
   System.out.println("Hello " + username);
else {
   username = askQuestion("What is your name?");
   System.out.println("Hello " + username + ". Welcome!");
}
```

When you use nested `if/else` statements, some caution is required to ensure that the `else` clause goes with the appropriate `if` statement. Consider the following lines:

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
else
   System.out.println("i doesn't equal j");    // WRONG!!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear (except from the hint given by the indentation) which `if` the `else` goes with. And in this example, the indentation hint is wrong. The rule is that an `else` clause like this is associated with the nearest `if` statement. Properly indented, this code looks like this:

```
if (i == j)
   if (j == k)
     System.out.println("i equals k");
   else
     System.out.println("i doesn't equal j");    // WRONG!!
```

This is legal code, but it is clearly not what the programmer had in mind. When working with nested `if` statements, you should use curly braces to make your code easier to read. Here is a better way to write the code:

```
if (i == j) {
   if (j == k)
     System.out.println("i equals k");
}
else {
   System.out.println("i doesn't equal j");
}
```

The else if clause

The `if/else` statement is useful for testing a condition and choosing between two statements or blocks of code to execute. But what about when you need to choose between several blocks of code? This is typically done with an `else if` clause, which is not really new syntax, but a common idiomatic usage of the standard `if/else` statement. It looks like this:

```
if (n == 1) {
    // Execute code block #1
}
else if (n == 2) {
    // Execute code block #2
}
else if (n == 3) {
    // Execute code block #3
}
else {
    // If all else fails, execute block #4
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is part of the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing these statements out in their fully nested form:

```
if (n == 1) {
   // Execute code block #1
```

```
}
else {
   if (n == 2) {
     // Execute code block #2
   }
   else {
     if (n == 3) {
       // Execute code block #3
     }
     else {
       // If all else fails, execute block #4
     }
   }
}
```

The switch Statement

An `if` statement causes a branch in the flow of a program's execution. You can use multiple `if` statements, as shown in the previous section, to perform a multiway branch. This is not always the best solution, however, especially when all of the branches depend on the value of a single variable. In this case, it is inefficient to repeatedly check the value of the same variable in multiple `if` statements.

A better solution is to use a `switch` statement, which is inherited from the C programming language. Although the syntax of this statement is not nearly as elegant as other parts of Java, the brute practicality of the construct makes it worthwhile.

A `switch` statement starts with an expression whose type is an `int`, `short`, `char`, `byte` (or their wrapper type), `String`, or an enum (see Chapter 4 for more on enumerated types).

This expression is followed by a block of code in curly braces that contains various entry points that correspond to possible values for the expression. For example, the following `switch` statement is equivalent to the repeated `if` and `else/if` statements shown in the previous section:

```
switch(n) {
   case 1:                          // Start here if n == 1
     // Execute code block #1
     break;                         // Stop here
   case 2:                          // Start here if n == 2
     // Execute code block #2
     break;                         // Stop here
   case 3:                          // Start here if n == 3
     // Execute code block #3
     break;                         // Stop here
   default:                         // If all else fails...
     // Execute code block #4
```

```
        break;                          // Stop here
    }
```

As you can see from the example, the various entry points into a `switch` statement are labeled either with the keyword `case`, followed by an integer value and a colon, or with the special `default` keyword, followed by a colon. When a `switch` statement executes, the interpreter computes the value of the expression in parentheses and then looks for a `case` label that matches that value. If it finds one, the interpreter starts executing the block of code at the first statement following the `case` label. If it does not find a `case` label with a matching value, the interpreter starts execution at the first statement following a special-case `default`: label. Or, if there is no `default`: label, the interpreter skips the body of the `switch` statement altogether.

Note the use of the `break` keyword at the end of each `case` in the previous code. The `break` statement is described later in this chapter, but, in this case, it causes the interpreter to exit the body of the `switch` statement. The `case` clauses in a `switch` statement specify only the starting point of the desired code. The individual cases are not independent blocks of code, and they do not have any implicit ending point. Therefore, you must explicitly specify the end of each case with a `break` or related statement. In the absence of `break` statements, a `switch` statement begins executing code at the first statement after the matching `case` label and continues executing statements until it reaches the end of the block. On rare occasions, it is useful to write code like this that falls through from one `case` label to the next, but 99% of the time you should be careful to end every `case` and `default` section with a statement that causes the `switch` statement to stop executing. Normally you use a `break` statement, but `return` and `throw` also work.

A `switch` statement can have more than one `case` clause labeling the same statement. Consider the `switch` statement in the following method:

```
boolean parseYesOrNoResponse(char response) {
   switch(response) {
     case 'y':
     case 'Y': return true;
     case 'n':
     case 'N': return false;
     default:
       throw new IllegalArgumentException("Response must be Y or N");
   }
}
```

The `switch` statement and its `case` labels have some important restrictions. First, the expression associated with a `switch` statement must have an appropriate type—either `byte`, `char`, `short`, `int` (or their wrappers), or an enum type or a `String`. The floating-point and `boolean` types are not supported, and neither is `long`, even though `long` is an integer type. Second, the value associated with each `case` label must be a constant value or a constant expression the compiler can evaluate. A `case` label cannot contain a runtime expression involving variables or method calls, for example. Third, the `case` label values must be within the range of the data type used

for the `switch` expression. And finally, it is not legal to have two or more `case` labels with the same value or more than one `default` label.

The while Statement

The `while` statement is a basic statement that allows Java to perform repetitive actions—or, to put it another way, it is one of Java's primary *looping constructs*. It has the following syntax:

```
while (expression)
   statement
```

The `while` statement works by first evaluating the *`expression`*, which must result in a `boolean` or `Boolean` value. If the value is `false`, the interpreter skips the *`statement`* associated with the loop and moves to the next statement in the program. If it is `true`, however, the *`statement`* that forms the body of the loop is executed, and the *`expression`* is reevaluated. Again, if the value of *`expression`* is `false`, the interpreter moves on to the next statement in the program; otherwise, it executes the *`statement`* again. This cycle continues while the *`expression`* remains `true` (i.e., until it evaluates to `false`), at which point the `while` statement ends, and the interpreter moves on to the next statement. You can create an infinite loop with the syntax `while(true)`.

Here is an example `while` loop that prints the numbers 0 to 9:

```
int count = 0;
while (count < 10) {
   System.out.println(count);
   count++;
}
```

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has executed 10 times, the expression becomes `false` (i.e., `count` is no longer less than 10), the `while` statement finishes, and the Java interpreter can move to the next statement in the program. Most loops have a counter variable like `count`. The variable names `i`, `j`, and `k` are commonly used as loop counters, although you should use more descriptive names if it makes your code easier to understand.

The do Statement

A `do` loop is much like a `while` loop, except that the loop expression is tested at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
   statement
while (expression);
```

Notice a couple of differences between the `do` loop and the more ordinary `while` loop. First, the `do` loop requires both the `do` keyword to mark the beginning of the

loop and the `while` keyword to mark the end and introduce the loop condition. Also, unlike the `while` loop, the `do` loop is terminated with a semicolon. This is because the `do` loop ends with the loop condition rather than simply ending with a curly brace that marks the end of the loop body. The following `do` loop prints the same output as the `while` loop just discussed:

```
int count = 0;
do {
   System.out.println(count);
   count++;
} while(count < 10);
```

The `do` loop is much less commonly used than its `while` cousin because, in practice, it is unusual to encounter a situation where you are sure you always want a loop to execute at least once.

The for Statement

The `for` statement provides a looping construct that is often more convenient than the `while` and `do` loops. The `for` statement takes advantage of a common looping pattern. Most loops have a counter, or state variable of some kind, that is initialized before the loop starts, tested to determine whether to execute the loop body, and then incremented or updated somehow at the end of the loop body before the test expression is evaluated again. The initialization, test, and update steps are the three crucial manipulations of a loop variable, and the `for` statement makes these three steps an explicit part of the loop syntax:

```
for(initialize; test; update) {
    statement
}
```

This `for` loop is basically equivalent to the following `while` loop:

```
initialize;
while (test) {
   statement;
   update;
}
```

Placing the *`initialize`*, *`test`*, and *`update`* expressions at the top of a `for` loop makes it especially easy to understand what the loop is doing, and it prevents mistakes such as forgetting to initialize or update the loop variable. The interpreter discards the values of the *`initialize`* and *`update`* expressions, so to be useful, these expressions must have side effects. *`initialize`* is typically an assignment expression while *`update`* is usually an increment, decrement, or some other assignment.

The following `for` loop prints the numbers 0 to 9, just as the previous `while` and `do` loops have done:

```
int count;
for(count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

Notice how this syntax places all the important information about the loop variable on a single line, making it very clear how the loop executes. Placing the update expression in the `for` statement itself also simplifies the body of the loop to a single statement; we don't even need to use curly braces to produce a statement block.

The `for` loop supports some additional syntax that makes it even more convenient to use. Because many loops use their loop variables only within the loop, the `for` loop allows the *`initialize`* expression to be a full variable declaration, so that the variable is scoped to the body of the loop and is not visible outside of it. For example:

```
for(int count = 0 ; count < 10 ; count++)
   System.out.println(count);
```

Furthermore, the `for` loop syntax does not restrict you to writing loops that use only a single variable. Both the *`initialize`* and *`update`* expressions of a `for` loop can use a comma to separate multiple initializations and update expressions. For example:

```
for(int i = 0, j = 10 ; i < 10 ; i++, j--)
     sum += i * j;
```

Even though all the examples so far have counted numbers, `for` loops are not restricted to loops that count numbers. For example, you might use a `for` loop to iterate through the elements of a linked list:

```
for(Node n = listHead; n != null; n = n.nextNode())
   process(n);
```

The *`initialize`*, *`test`*, and *`update`* expressions of a `for` loop are all optional; only the semicolons that separate the expressions are required. If the *`test`* expression is omitted, it is assumed to be `true`. Thus, you can write an infinite loop as `for(;;)`.

The foreach Statement

Java's `for` loop works well for primitive types, but it is needlessly clunky for handling collections of objects. Instead, an alternative syntax known as a *foreach* loop is used for handling collections of objects that need to be looped over.

The foreach loop uses the keyword `for` followed by an opening parenthesis, a variable declaration (without initializer), a colon, an expression, a closing parenthesis, and finally the statement (or block) that forms the body of the loop:

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

Despite its name, the foreach loop does not have a keyword `foreach`—instead, it is common to read the colon as "in"—as in "foreach name in studentNames."

For the `while`, `do`, and `for` loops, we've shown an example that prints 10 numbers. The foreach loop can do this too, but it needs a collection to iterate over. In order to loop 10 times (to print out 10 numbers), we need an array or other collection with 10 elements. Here's code we can use:

```
// These are the numbers we want to print
int[] primes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
// This is the loop that prints them
for(int n : primes)
     System.out.println(n);
```


What foreach cannot do

Foreach is different from the `while`, `for`, or `do` loops, because it hides the loop counter or `Iterator` from you. This is a very powerful idea, as we'll see when we discuss lambda expressions, but there are some algorithms that cannot be expressed very naturally with a foreach loop.

For example, suppose you want to print the elements of an array as a comma-separated list. To do this, you need to print a comma after every element of the array except the last, or equivalently, before every element of the array except the first. With a traditional `for` loop, the code might look like this:

```
for(int i = 0; i < words.length; i++) {
     if (i > 0) System.out.print(", ");
     System.out.print(words[i]);
}
```

This is a very straightforward task, but you simply cannot do it with foreach. The problem is that the foreach loop doesn't give you a loop counter or any other way to tell if you're on the first iteration, the last iteration, or somewhere in between.

A similar issue exists when using foreach to iterate through the elements of a collection. Just as a foreach loop over an array has no way to obtain the array index of the current element, a foreach loop over a collection has no way to obtain the `Iterator` object that is being used to itemize the elements of the collection.

Here are some other things you cannot do with a foreach style loop:

- Iterate backward through the elements of an array or `List`.
- Use a single loop counter to access the same-numbered elements of two distinct arrays.
- Iterate through the elements of a `List` using calls to its `get()` method rather than calls to its iterator.

The break Statement

A `break` statement causes the Java interpreter to skip immediately to the end of a containing statement. We have already seen the `break` statement used with the `switch` statement. The `break` statement is most often written as simply the keyword `break` followed by a semicolon:

```
break;
```

When used in this form, it causes the Java interpreter to immediately exit the innermost containing `while`, `do`, `for`, or `switch` statement. For example:

```
for(int i = 0; i < data.length; i++) {
    if (data[i] == target) {  // When we find what we're looking for,
        index = i;              // remember where we found it
        break;                  // and stop looking!
    }
}   // The Java interpreter goes here after executing break
```

The `break` statement can also be followed by the name of a containing labeled statement. When used in this form, `break` causes the Java interpreter to immediately exit the named block, which can be any kind of statement, not just a loop or `switch`. For example:

```
TESTFORNULL: if (data != null) {
   for(int row = 0; row < numrows; row++) {
     for(int col = 0; col < numcols; col++) {
       if (data[row][col] == null)
         break TESTFORNULL;           // treat the array as undefined.
     }
   }
}  // Java interpreter goes here after executing break TESTFORNULL
```

The continue Statement

While a `break` statement exits a loop, a `continue` statement quits the current iteration of a loop and starts the next one. `continue`, in both its unlabeled and labeled forms, can be used only within a `while`, `do`, or `for` loop. When used without a label, `continue` causes the innermost loop to start a new iteration. When used with a label that is the name of a containing loop, it causes the named loop to start a new iteration. For example:

```
for(int i = 0; i < data.length; i++) {  // Loop through data.
   if (data[i] == -1)                   // If a data value is missing,
     continue;                          // skip to the next iteration.
   process(data[i]);                    // Process the data value.
}
```

`while`, `do`, and `for` loops differ slightly in the way that `continue` starts a new iteration:

- With a `while` loop, the Java interpreter simply returns to the top of the loop, tests the loop condition again, and, if it evaluates to `true`, executes the body of the loop again.
- With a `do` loop, the interpreter jumps to the bottom of the loop, where it tests the loop condition to decide whether to perform another iteration of the loop.
- With a `for` loop, the interpreter jumps to the top of the loop, where it first evaluates the *`update`* expression and then evaluates the *`test`* expression to decide whether to loop again. As you can see from the examples, the behavior of a `for` loop with a `continue` statement is different from the behavior of the "basically equivalent" `while` loop presented earlier; *`update`* gets evaluated in the `for` loop but not in the equivalent `while` loop.

The return Statement

A `return` statement tells the Java interpreter to stop executing the current method. If the method is declared to return a value, the `return` statement must be followed by an expression. The value of the expression becomes the return value of the method. For example, the following method computes and returns the square of a number:

```
double square(double x) {       // A method to compute x squared
   return x * x;                // Compute and return a value
}
```

Some methods are declared `void` to indicate that they do not return any value. The Java interpreter runs methods like this by executing their statements one by one until it reaches the end of the method. After executing the last statement, the interpreter returns implicitly. Sometimes, however, a `void` method has to return explicitly before reaching the last statement. In this case, it can use the `return` statement by itself, without any expression. For example, the following method prints, but does not return, the square root of its argument. If the argument is a negative number, it returns without printing anything:

```
// A method to print square root of x
void printSquareRoot(double x) {
   if (x < 0) return;                // If x is negative, return
   System.out.println(Math.sqrt(x)); // Print the square root of x
}                                    // Method end: return implicitly
```

The synchronized Statement

Java has always provided support for multithreaded programming. We cover this in some detail later on (especially in "Java's Support for Concurrency" on page 208)—but the reader should be aware that concurrency is difficult to get right, and has a number of subtleties.

In particular, when working with multiple threads, you must often take care to prevent multiple threads from modifying an object simultaneously in a way that might corrupt the object's state. Java provides the `synchronized` statement to help the programmer prevent corruption. The syntax is:

```
synchronized ( expression ) {
   statements
}
```

`expression` is an expression that must evaluate to an object or an array. *`statements`* constitute the code of the section that could cause damage and must be enclosed in curly braces.

Before executing the statement block, the Java interpreter first obtains an exclusive lock on the object or array specified by *`expression`*. It holds the lock until it is finished running the block, then releases it. While a thread holds the lock on an object, no other thread can obtain that lock.

The `synchronized` keyword is also available as a method modifier in Java, and when applied to a method, the `synchronized` keyword indicates that the entire method is locked. For a `synchronized` class method (a static method), Java obtains an exclusive lock on the class before executing the method. For a `synchronized` instance method, Java obtains an exclusive lock on the class instance. (Class and instance methods are discussed in Chapter 3.)

The throw Statement

An *exception* is a signal that indicates some sort of exceptional condition or error has occurred. To *throw* an exception is to signal an exceptional condition. To *catch* an exception is to handle it—to take whatever actions are necessary to recover from it. In Java, the `throw` statement is used to throw an exception:

```
throw expression;
```

The *`expression`* must evaluate to an exception object that describes the exception or error that has occurred. We'll talk more about types of exceptions shortly; for now, all you need to know is that an exception is represented by an object, which has a slightly specialized role. Here is some example code that throws an exception:

```
public static double factorial(int x) {
   if (x < 0)
     throw new IllegalArgumentException("x must be >= 0");
   double fact;
   for(fact=1.0; x > 1; fact *= x, x--)
     /* empty */ ;          // Note use of the empty statement
   return fact;
}
```

When the Java interpreter executes a `throw` statement, it immediately stops normal program execution and starts looking for an exception handler that can catch, or handle, the exception. Exception handlers are written with the `try/catch/finally`

statement, which is described in the next section. The Java interpreter first looks at the enclosing block of code to see if it has an associated exception handler. If so, it exits that block of code and starts running the exception-handling code associated with the block. After running the exception handler, the interpreter continues execution at the statement immediately following the handler code.

If the enclosing block of code does not have an appropriate exception handler, the interpreter checks the next higher enclosing block of code in the method. This continues until a handler is found. If the method does not contain an exception handler that can handle the exception thrown by the `throw` statement, the interpreter stops running the current method and returns to the caller. Now the interpreter starts looking for an exception handler in the blocks of code of the calling method. In this way, exceptions propagate up through the lexical structure of Java methods, up the call stack of the Java interpreter. If the exception is never caught, it propagates all the way up to the `main()` method of the program. If it is not handled in that method, the Java interpreter prints an error message, prints a stack trace to indicate where the exception occurred, and then exits.

The try/catch/finally Statement

Java has two slightly different exception-handling mechanisms. The classic form is the `try/catch/finally` statement. The `try` clause of this statement establishes a block of code for exception handling. This `try` block is followed by zero or more `catch` clauses, each of which is a block of statements designed to handle specific exceptions. Each catch block can handle more than one different exception—to indicate that a catch block should handle multiple exceptions, we use the `|` symbol to separate the different exceptions a catch block should handle. The `catch` clauses are followed by an optional `finally` block that contains cleanup code guaranteed to be executed regardless of what happens in the `try` block.

try Block Syntax

Both the `catch` and `finally` clauses are optional, but every `try` block must be accompanied by at least one or the other. The `try`, `catch`, and `finally` blocks all begin and end with curly braces. These are a required part of the syntax and cannot be omitted, even if the clause contains only a single statement.

The following code illustrates the syntax and purpose of the `try/catch/finally` statement:

```
try {
   // Normally this code runs from the top of the block to the bottom
   // without problems. But it can sometimes throw an exception,
   // either directly with a throw statement or indirectly by calling
   // a method that throws an exception.
}
catch (SomeException e1) {
```

```
    // This block contains statements that handle an exception object
    // of type SomeException or a subclass of that type. Statements in
    // this block can refer to that exception object by the name e1.
}
catch (AnotherException | YetAnotherException e2) {
    // This block contains statements that handle an exception of
    // type AnotherException or YetAnotherException, or a subclass of
    // either of those types. Statements in this block refer to the
    // exception object they receive by the name e2.
}
finally {
    // This block contains statements that are always executed
    // after we leave the try clause, regardless of whether we leave it:
    //   1) normally, after reaching the bottom of the block;
    //   2) because of a break, continue, or return statement;
    //   3) with an exception that is handled by a catch clause above;
    //   4) with an uncaught exception that has not been handled.
    // If the try clause calls System.exit(), however, the interpreter
    // exits before the finally clause can be run.
}
```

try

The `try` clause simply establishes a block of code that either has its exceptions handled or needs special cleanup code to be run when it terminates for any reason. The `try` clause by itself doesn't do anything interesting; it is the `catch` and `finally` clauses that do the exception-handling and cleanup operations.

catch

A `try` block can be followed by zero or more `catch` clauses that specify code to handle various types of exceptions. Each `catch` clause is declared with a single argument that specifies the types of exceptions the clause can handle (possibly using the special `|` syntax to indicate that the `catch` block can handle more than one type of exception) and also provides a name the clause can use to refer to the exception object it is currently handling. Any type that a `catch` block wishes to handle must be some subclass of `Throwable`.

When an exception is thrown, the Java interpreter looks for a `catch` clause with an argument that matches the same type as the exception object or a superclass of that type. The interpreter invokes the first such `catch` clause it finds. The code within a `catch` block should take whatever action is necessary to cope with the exceptional condition. If the exception is a `java.io.FileNotFoundException` exception, for example, you might handle it by asking the user to check his spelling and try again.

It is not required to have a `catch` clause for every possible exception; in some cases, the correct response is to allow the exception to propagate up and be caught by the invoking method. In other cases, such as a programming error signaled by `NullPointerException`, the correct response is probably not to catch the exception at

all, but allow it to propagate and have the Java interpreter exit with a stack trace and an error message.

finally

The `finally` clause is generally used to clean up after the code in the `try` clause (e.g., close files and shut down network connections). The `finally` clause is useful because it is guaranteed to be executed if any portion of the `try` block is executed, regardless of how the code in the `try` block completes. In fact, the only way a `try` clause can exit without allowing the `finally` clause to be executed is by invoking the `System.exit()` method, which causes the Java interpreter to stop running.

In the normal case, control reaches the end of the `try` block and then proceeds to the `finally` block, which performs any necessary cleanup. If control leaves the `try` block because of a `return`, `continue`, or `break` statement, the `finally` block is executed before control transfers to its new destination.

If an exception occurs in the `try` block and there is an associated `catch` block to handle the exception, control transfers first to the `catch` block and then to the `finally` block. If there is no local `catch` block to handle the exception, control transfers first to the `finally` block, and then propagates up to the nearest containing `catch` clause that can handle the exception.

If a `finally` block itself transfers control with a `return`, `continue`, `break`, or `throw` statement or by calling a method that throws an exception, the pending control transfer is abandoned, and this new transfer is processed. For example, if a `finally` clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a `finally` clause issues a `return` statement, the method returns normally, even if an exception has been thrown and has not yet been handled.

`try` and `finally` can be used together without exceptions or any `catch` clauses. In this case, the `finally` block is simply cleanup code that is guaranteed to be executed, regardless of any `break`, `continue`, or `return` statements within the `try` clause.

The try-with-resources Statement

The standard form of a `try` block is very general, but there is a common set of circumstances that require developers to be very careful when writing `catch` and `finally` blocks. These circumstances are when operating with resources that need to be cleaned up or closed when no longer needed.

Java (since version 7) provides a very useful mechanism for automatically closing resources that require cleanup. This is known as `try`-with-resources, or TWR. We discuss TWR in detail in "Classic Java I/O" on page 289—but for completeness, let's

introduce the syntax now. The following example shows how to open a file using the `FileInputStream` class (which results in an object which will require cleanup):

```
try (InputStream is = new FileInputStream("/Users/ben/details.txt")) {
  // ... process the file
}
```

This new form of `try` takes parameters that are all objects that require cleanup.[2] These objects are scoped to this `try` block, and are then cleaned up automatically no matter how this block is exited. The developer does not need to write any `catch` or `finally` blocks—the Java compiler automatically inserts correct cleanup code.

All new code that deals with resources should be written in the TWR style—it is considerably less error prone than manually writing `catch` blocks, and does not suffer from the problems that plague techniques such as finalization (see "Finalization" on page 206 for details).

The assert Statement

An `assert` statement is an attempt to provide a capability to verify design assumptions in Java code. An *assertion* consists of the `assert` keyword followed by a boolean expression that the programmer believes should always evaluate to `true`. By default, assertions are not enabled, and the `assert` statement does not actually do anything.

It is possible to enable assertions as a debugging tool, however; when this is done, the `assert` statement evaluates the expression. If it is indeed `true`, `assert` does nothing. On the other hand, if the expression evaluates to `false`, the assertion fails, and the `assert` statement throws a `java.lang.AssertionError`.

Outside of the core JDK libraries, the `assert` statement is *extremely* rarely used. It turns out to be too inflexible for testing most applications and is not often used by ordinary developers, except sometimes for field-debugging complex multithreaded applications.

The `assert` statement may include an optional second expression, separated from the first by a colon. When assertions are enabled and the first expression evaluates to `false`, the value of the second expression is taken as an error code or error message and is passed to the `AssertionError()` constructor. The full syntax of the statement is:

```
assert assertion;
```

or:

```
assert assertion : errorcode;
```

2 Technically, they must all implement the `AutoCloseable` interface.

To use assertions effectively, you must also be aware of a couple of fine points. First, remember that your programs will normally run with assertions disabled and only sometimes with assertions enabled. This means that you should be careful not to write assertion expressions that contain side effects.

You should never throw `AssertionError` from your own code, as it may have unexpected results in future versions of the platform.

If an `AssertionError` is thrown, it indicates that one of the programmer's assumptions has not held up. This means that the code is being used outside of the parameters for which it was designed, and it cannot be expected to work correctly. In short, there is no plausible way to recover from an `AssertionError`, and you should not attempt to catch it (unless you catch it at the top level simply so that you can display the error in a more user-friendly fashion).

Enabling assertions

For efficiency, it does not make sense to test assertions each time code is executed—`assert` statements encode assumptions that should always be true. Thus, by default, assertions are disabled, and `assert` statements have no effect. The assertion code remains compiled in the class files, however, so it can always be enabled for diagnostic or debugging purposes. You can enable assertions, either across the board or selectively, with command-line arguments to the Java interpreter.

To enable assertions in all classes except for system classes, use the `-ea` argument. To enable assertions in system classes, use `-esa`. To enable assertions within a specific class, use `-ea` followed by a colon and the classname:

```
java -ea:com.example.sorters.MergeSort com.example.sorters.Test
```

To enable assertions for all classes in a package and in all of its subpackages, follow the `-ea` argument with a colon, the package name, and three dots:

```
java -ea:com.example.sorters... com.example.sorters.Test
```

You can disable assertions in the same way, using the `-da` argument. For example, to enable assertions throughout a package and then disable them in a specific class or subpackage, use:

```
java -ea:com.example.sorters... -da:com.example.sorters.QuickSort
java -ea:com.example.sorters... -da:com.example.sorters.plugins..
```

Finally, it is possible to control whether or not assertions are enabled or disabled at classloading time. If you use a custom classloader (see Chapter 11 for details on custom classloading) in your program and want to turn on assertions, you may be interested in these methods.

Methods

A *method* is a named sequence of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as *arguments*. The method performs some computations and, optionally, returns a value. As described earlier in "Expressions and Operators" on page 30, a method invocation is an expression that is evaluated by the Java interpreter. Because method invocations can have side effects, however, they can also be used as expression statements. This section does not discuss method invocation, but instead describes how to define methods.

Defining Methods

You already know how to define the body of a method; it is simply an arbitrary sequence of statements enclosed within curly braces. What is more interesting about a method is its *signature*.[3] The signature specifies the following:

- The name of the method
- The number, order, type, and name of the parameters used by the method
- The type of the value returned by the method
- The checked exceptions that the method can throw (the signature may also list unchecked exceptions, but these are not required)
- Various method modifiers that provide additional information about the method

A method signature defines everything you need to know about a method before calling it. It is the method *specification* and defines the API for the method. In order to use the Java platform's online API reference, you need to know how to read a method signature. And, in order to write Java programs, you need to know how to define your own methods, each of which begins with a method signature.

A method signature looks like this:

```
modifiers type name ( paramlist ) [ throws exceptions ]
```

The signature (the method specification) is followed by the method body (the method implementation), which is simply a sequence of Java statements enclosed in curly braces. If the method is *abstract* (see Chapter 3), the implementation is omitted, and the method body is replaced with a single semicolon.

The signature of a method may also include type variable declarations—such methods are known as *generic methods*. Generic methods and type variables are discussed in Chapter 4.

3 In the Java Language Specification, the term "signature" has a technical meaning that is slightly different than that used here. This book uses a less formal definition of method signature.

Here are some example method definitions, which begin with the signature and are followed by the method body:

```
// This method is passed an array of strings and has no return value.
// All Java programs have an entry point with this name and signature.
public static void main(String[] args) {
     if (args.length > 0) System.out.println("Hello " + args[0]);
     else System.out.println("Hello world");
}

// This method is passed two double arguments and returns a double.
static double distanceFromOrigin(double x, double y) {
     return Math.sqrt(x*x + y*y);
}

// This method is abstract which means it has no body.
// Note that it may throw exceptions when invoked.
protected abstract String readText(File f, String encoding)
    throws FileNotFoundException, UnsupportedEncodingException;
```

`modifiers` is zero or more special modifier keywords, separated from each other by spaces. A method might be declared with the `public` and `static` modifiers, for example. The allowed modifiers and their meanings are described in the next section.

The *`type`* in a method signature specifies the return type of the method. If the method does not return a value, *`type`* must be `void`. If a method is declared with a non-`void` return type, it must include a `return` statement that returns a value of (or convertible to) the declared type.

A *constructor* is a block of code, similar to a method, that is used to initialize newly created objects. As we'll see in Chapter 3, constructors are defined in a very similar way to methods, except that their signatures do not include this *`type`* specification.

The *`name`* of a method follows the specification of its modifiers and type. Method names, like variable names, are Java identifiers and, like all Java identifiers, may contain letters in any language represented by the Unicode character set. It is legal, and often quite useful, to define more than one method with the same name, as long as each version of the method has a different parameter list. Defining multiple methods with the same name is called *method overloading*.

Unlike some other languages, Java does not have anonymous methods. Instead, Java 8 introduces lambda expressions, which are similar to anonymous methods, but which the Java runtime automatically converts to a suitable named method—see "Lambda Expressions" on page 76 for more details.

For example, the `System.out.println()` method we've seen already is an overloaded method. One method by this name prints a string and other methods by the same name print the values of the various primitive types. The Java compiler

decides which method to call based on the type of the argument passed to the method.

When you are defining a method, the name of the method is always followed by the method's parameter list, which must be enclosed in parentheses. The parameter list defines zero or more arguments that are passed to the method. The parameter specifications, if there are any, each consist of a type and a name and are separated from each other by commas (if there are multiple parameters). When a method is invoked, the argument values it is passed must match the number, type, and order of the parameters specified in this method signature line. The values passed need not have exactly the same type as specified in the signature, but they must be convertible to those types without casting.

When a Java method expects no arguments, its parameter list is simply `()`, not `(void)`. Java does not regard `void` as a type—C and C++ programmers in particular should pay heed.

Java allows the programmer to define and invoke methods that accept a variable number of arguments, using a syntax known colloquially as *varargs*. Varargs are covered in detail later in this chapter.

The final part of a method signature is the `throws` clause, which is used to list the *checked exceptions* that a method can throw. Checked exceptions are a category of exception classes that must be listed in the `throws` clauses of methods that can throw them. If a method uses the `throw` statement to throw a checked exception, or if it calls some other method that throws a checked exception and does not catch or handle that exception, the method must declare that it can throw that exception. If a method can throw one or more checked exceptions, it specifies this by placing the `throws` keyword after the argument list and following it by the name of the exception class or classes it can throw. If a method does not throw any exceptions, it does not use the `throws` keyword. If a method throws more than one type of exception, separate the names of the exception classes from each other with commas. More on this in a bit.

Method Modifiers

The modifiers of a method consist of zero or more modifier keywords such as `public`, `static`, or `abstract`. Here is a list of allowed modifiers and their meanings:

`abstract`

An `abstract` method is a specification without an implementation. The curly braces and Java statements that would normally comprise the body of the method are replaced with a single semicolon. A class that includes an `abstract` method must itself be declared `abstract`. Such a class is incomplete and cannot be instantiated (see Chapter 3).

`final`
: A `final` method may not be overridden or hidden by a subclass, which makes it amenable to compiler optimizations that are not possible for regular methods. All `private` methods are implicitly `final`, as are all methods of any class that is declared `final`.

`native`
: The `native` modifier specifies that the method implementation is written in some "native" language such as C and is provided externally to the Java program. Like `abstract` methods, `native` methods have no body: the curly braces are replaced with a semicolon.

Implementing native Methods

When Java was first released, `native` methods were sometimes used for efficiency reasons. That is almost never necessary today. Instead, `native` methods are used to interface Java code to existing libraries written in C or C++. `native` methods are implicitly platform-dependent, and the procedure for linking the implementation with the Java class that declares the method is dependent on the implementation of the Java virtual machine. `native` methods are not covered in this book.

`public, protected, private`
: These access modifiers specify whether and where a method can be used outside of the class that defines it. These very important modifiers are explained in Chapter 3.

`static`
: A method declared `static` is a *class method* associated with the class itself rather than with an instance of the class (we cover this in more detail in Chapter 3).

`strictfp`
: The `fp` in this awkwardly named, rarely used modifier stands for "floating point." Java normally takes advantage of any extended precision available to the runtime platform's floating-point hardware. The use of this keyword forces Java to strictly obey the standard while running the `strictfp` method and only perform floating-point arithmetic using 32- or 64-bit floating-point formats, even if this makes the results less accurate.

`synchronized`
: The `synchronized` modifier makes a method threadsafe. Before a thread can invoke a `synchronized` method, it must obtain a lock on the method's class (for `static` methods) or on the relevant instance of the class (for non-`static` methods). This prevents two threads from executing the method at the same time.

The `synchronized` modifier is an implementation detail (because methods can make themselves threadsafe in other ways) and is not formally part of the method specification or API. Good documentation specifies explicitly whether a method is threadsafe; you should not rely on the presence or absence of the `synchronized` keyword when working with multithreaded programs.

Annotations are an interesting special case (see Chapter 4 for more on annotations)—they can be thought of as a halfway house between a method modifier and additional supplementary type information.

Checked and Unchecked Exceptions

The Java exception-handling scheme distinguishes between two types of exceptions, known as *checked* and *unchecked* exceptions.

The distinction between checked and unchecked exceptions has to do with the circumstances under which the exceptions could be thrown. Checked exceptions arise in specific, well-defined circumstances, and very often are conditions from which the application may be able to partially or fully recover.

For example, consider some code that might find its configuration file in one of several possible directories. If we attempt to open the file from a directory it isn't present in, then a `FileNotFoundException` will be thrown. In our example, we want to catch this exception and move on to try the next possible location for the file. In other words, although the file not being present is an exceptional condition, it is one from which we can recover, and it is an understood and anticipated failure.

On the other hand, in the Java environment there are a set of failures that cannot easily be predicted or anticipated, due to such things as runtime conditions or abuse of library code. There is no good way to predict an `OutOfMemoryError`, for example, and any method that uses objects or arrays can throw a `NullPointerException` if it is passed an invalid `null` argument.

These are the unchecked exceptions—and practically any method can throw an unchecked exception at essentially any time. They are the Java environment's version of Murphy's law: "Anything that can go wrong, will go wrong." Recovery from an unchecked exception is usually very difficult, if not impossible—simply due to their sheer unpredictability.

To figure out whether an exception is checked or unchecked, remember that exceptions are `Throwable` objects and that exceptions fall into two main categories, specified by the `Error` and `Exception` subclasses. Any exception object that is an `Error` is unchecked. There is also a subclass of `Exception` called `RuntimeException`—and any subclass of `RuntimeException` is also an unchecked exception. All other exceptions are checked exceptions.

Working with checked exceptions

Java has different rules for working with checked and unchecked exceptions. If you write a method that throws a checked exception, you must use a `throws` clause to declare the exception in the method signature. The Java compiler checks to make sure you have declared them in method signatures and produces a compilation error if you have not (that's why they're called "checked exceptions").

Even if you never throw a checked exception yourself, sometimes you must use a `throws` clause to declare a checked exception. If your method calls a method that can throw a checked exception, you must either include exception-handling code to handle that exception or use `throws` to declare that your method can also throw that exception.

For example, the following method tries to estimate the size of a web page—it uses the standard `java.net` libraries, and the class `URL` (we'll meet these in Chapter 10) to contact the web page. It uses methods and constructors that can throw various types of `java.io.IOException` objects, so it declares this fact with a `throws` clause:

```
public static estimateHomepageSize(String host) throws IOException {
    URL url = new URL("http://"+ host +"/");
    try (InputStream in = url.openStream()) {
        return in.available();
    }
}
```

In fact, the preceding code has a bug: we've misspelled the protocol specifier—there's no such protocol as *http://*. So, the `estimateHomepageSize()` method will always fail with a `MalformedURLException`.

How do you know if the method you are calling can throw a checked exception? You can look at its method signature to find out. Or, failing that, the Java compiler will tell you (by reporting a compilation error) if you've called a method whose exceptions you must handle or declare.

Variable-Length Argument Lists

Methods may be declared to accept, and may be invoked with, variable numbers of arguments. Such methods are commonly known as *varargs* methods. The "print formatted" method `System.out.printf()` as well as the related `format()` methods of `String` use varargs, as do a number of important methods from the Reflection API of `java.lang.reflect`.

A variable-length argument list is declared by following the type of the last argument to the method with an ellipsis (…), indicating that this last argument can be repeated zero or more times. For example:

```
public static int max(int first, int... rest) {
    /* body omitted for now */
}
```

Varargs methods are handled purely by the compiler. They operate by converting the variable number of arguments into an array. To the Java runtime, the `max()` method is indistinguishable from this one:

```
public static int max(int first, int[] rest) {
    /* body omitted for now */
}
```

To convert a varargs signature to the "real" signature, simply replace `...` with `[ ]`. Remember that only one ellipsis can appear in a parameter list, and it may only appear on the last parameter in the list.

Let's flesh out the `max()` example a little:

```
public static int max(int first, int... rest) {
    int max = first;
    for(int i : rest) { // legal because rest is actually an array
        if (i > max) max = i;
    }
    return max;
}
```

This `max()` method is declared with two arguments. The first is just a regular `int` value. The second, however, may be repeated zero or more times. All of the following are legal invocations of `max()`:

```
max(0)
max(1, 2)
max(16, 8, 4, 2, 1)
```

Because varargs methods are compiled into methods that expect an array of arguments, invocations of those methods are compiled to include code that creates and initializes such an array. So the call `max(1,2,3)` is compiled to this:

```
max(1, new int[] { 2, 3 })
```

In fact, if you already have method arguments stored in an array, it is perfectly legal for you to pass them to the method that way, instead of writing them out individually. You can treat any `...` argument as if it were declared as an array. The converse is not true, however: you can only use varargs method invocation syntax when the method is actually declared as a varargs method using an ellipsis.

Introduction to Classes and Objects

Now that we have introduced operators, expressions, statements, and methods, we can finally talk about classes. A *class* is a named collection of fields that hold data values and methods that operate on those values. Classes are just one of five reference types supported by Java, but they are the most important type. Classes are thoroughly documented in a chapter of their own (Chapter 3). We introduce them here, however, because they are the next higher level of syntax after methods, and because the rest of this chapter requires a basic familiarity with the concept of a

class and the basic syntax for defining a class, instantiating it, and using the resulting *object*.

The most important thing about classes is that they define new data types. For example, you might define a class named `Point` to represent a data point in the two-dimensional Cartesian coordinate system. This class would define fields (each of type `double`) to hold the *x* and *y* coordinates of a point and methods to manipulate and operate on the point. The `Point` class is a new data type.

When discussing data types, it is important to distinguish between the data type itself and the values the data type represents. `char` is a data type: it represents Unicode characters. But a `char` value represents a single specific character. A class is a data type; a class value is called an *object*. We use the name class because each class defines a type (or kind, or species, or class) of objects. The `Point` class is a data type that represents *x,y* points, while a `Point` object represents a single specific *x,y* point. As you might imagine, classes and their objects are closely linked. In the sections that follow, we will discuss both.

Defining a Class

Here is a possible definition of the `Point` class we have been discussing:

```
/** Represents a Cartesian (x,y) point */
public class Point {
     // The coordinates of the point
     public double x, y;
     public Point(double x, double y) {     // A constructor that
         this.x = x; this.y = y;            // initializes the fields
     }

     public double distanceFromOrigin() {   // A method that operates
         return Math.sqrt(x*x + y*y);       // on the x and y fields
     }
}
```

This class definition is stored in a file named *Point.java* and compiled to a file named *Point.class*, where it is available for use by Java programs and other classes. This class definition is provided here for completeness and to provide context, but don't expect to understand all the details just yet; most of Chapter 3 is devoted to the topic of defining classes.

Keep in mind that you don't have to define every class you want to use in a Java program. The Java platform includes thousands of predefined classes that are guaranteed to be available on every computer that runs Java.

Creating an Object

Now that we have defined the `Point` class as a new data type, we can use the following line to declare a variable that holds a `Point` object:

```
Point p;
```

Declaring a variable to hold a `Point` object does not create the object itself, however. To actually create an object, you must use the `new` operator. This keyword is followed by the object's class (i.e., its type) and an optional argument list in parentheses. These arguments are passed to the constructor for the class, which initializes internal fields in the new object:

```
// Create a Point object representing (2,-3.5).
// Declare a variable p and store a reference to the new Point object
Point p = new Point(2.0, -3.5);

// Create some other objects as well
// A Date object that represents the current time
Date d = new Date();
// A HashSet object to hold a set of object
Set words = new HashSet();
```

The `new` keyword is by far the most common way to create objects in Java. A few other ways are also worth mentioning. First, classes that meet certain criteria are so important that Java defines special literal syntax for creating objects of those types (as we discuss later in this section). Second, Java supports a dynamic loading mechanism that allows programs to load classes and create instances of those classes dynamically. See Chapter 11 for more details. Finally, objects can also be created by deserializing them. An object that has had its state saved, or serialized, usually to a file, can be re-created using the `java.io.ObjectInputStream` class.

Using an Object

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of those fields and has access to those methods. We use the dot character (.) to access the named fields and methods of an object. For example:

```
Point p = new Point(2, 3);          // Create an object
double x = p.x;                     // Read a field of the object
p.y = p.x * p.x;                    // Set the value of a field
double d = p.distanceFromOrigin(); // Access a method of the object
```

This syntax is very common when programming in object-oriented languages, and Java is no exception, so you'll see it a lot. Note, in particular, `p.distanceFromOrigin()`. This expression tells the Java compiler to look up a method named `distanceFromOrigin()` (which is defined by the class `Point`) and use that method to perform a computation on the fields of the object `p`. We'll cover the details of this operation in Chapter 3.

Object Literals

In our discussion of primitive types, we saw that each primitive type has a literal syntax for including values of the type literally into the text of a program. Java also defines a literal syntax for a few special reference types, as described next.

String literals

The `String` class represents text as a string of characters. Because programs usually communicate with their users through the written word, the ability to manipulate strings of text is quite important in any programming language. In Java, strings are objects; the data type used to represent text is the `String` class. Modern Java programs usually use more string data than anything else.

Accordingly, because strings are such a fundamental data type, Java allows you to include text literally in programs by placing it between double-quote (") characters. For example:

```
String name = "David";
System.out.println("Hello, " + name);
```

Don't confuse the double-quote characters that surround string literals with the single-quote (or apostrophe) characters that surround `char` literals. String literals can contain any of the escape sequences `char` literals can (see Table 2-2). Escape sequences are particularly useful for embedding double-quote characters within double-quoted string literals. For example:

```
String story = "\t\"How can you stand it?\" he asked sarcastically.\n";
```

String literals cannot contain comments and may consist of only a single line. Java does not support any kind of continuation-character syntax that allows two separate lines to be treated as a single line. If you need to represent a long string of text that does not fit on a single line, break it into independent string literals and use the `+` operator to concatenate the literals. For example:

```
// This is illegal; string  literals cannot be broken across lines.
String x = "This is a test of the
            emergency broadcast system";

String s = "This is a test of the " +     // Do this instead
           "emergency broadcast system";
```

This concatenation of literals is done when your program is compiled, not when it is run, so you do not need to worry about any kind of performance penalty.

Type literals

The second type that supports its own special object literal syntax is the class named `Class`. Instances of the `Class` class represent a Java data type, and contain metadata about the type that is referred to. To include a `Class` object literally in a Java program, follow the name of any data type with `.class`. For example:

```
Class<?> typeInt = int.class;
Class<?> typeIntArray = int[].class;
Class<?> typePoint = Point.class;
```

The null reference

The `null` keyword is a special literal value that is a reference to nothing, or an absence of a reference. The `null` value is unique because it is a member of every reference type. You can assign `null` to variables of any reference type. For example:

```
String s = null;
Point p = null;
```

Lambda Expressions

In Java 8, a major new feature was introduced—*lambda expressions*. These are a very common programming language construct, and in particular are extremely widely used in the family of languages known as *functional programming languages* (e.g., Lisp, Haskell, and OCaml). The power and flexibility of lambdas goes far beyond just functional languages, and they can be found in almost all modern programming languages.

> ### Definition of a Lambda Expression
>
> A lambda expression is essentially a function that does not have a name, and can be treated as a value in the language. As Java does not allow code to run around on its own outside of classes, in Java, this means that a lambda is an anonymous method that is defined on some class (that is possibly unknown to the developer).

The syntax for a lambda expression looks like this:

```
( paramlist ) -> { statements }
```

One simple, very traditional example:

```
Runnable r = () -> System.out.println("Hello World");
```

When a lambda expression is used as a value it is automatically converted to a new object of the correct type for the variable that it is being placed into. This autoconversion and *type inference* is essential to Java's approach to lambda expressions. Unfortunately, it relies on a proper understanding of Java's type system as a whole. "Lambda Expressions" on page 171 provides a more detailed explanation of lambda expressions—so for now, it suffices to simply recognize the syntax for lambdas.

A slightly more complex example:

```
ActionListener listener = (e) -> {
  System.out.println("Event fired at: "+ e.getWhen());
  System.out.println("Event command: "+ e.getActionCommand());
};
```

Arrays

An *array* is a special kind of object that holds zero or more primitive values or references. These values are held in the *elements* of the array, which are unnamed variables referred to by their position or *index*. The type of an array is characterized by its *element type*, and all elements of the array must be of that type.

Array elements are numbered starting with zero, and valid indexes range from zero to the number of elements minus one. The array element with index 1, for example, is the *second* element in the array. The number of elements in an array is its *length*. The length of an array is specified when the array is created, and it never changes.

The element type of an array may be any valid Java type, including array types. This means that Java supports arrays of arrays, which provide a kind of multidimensional array capability. Java does not support the matrix-style multidimensional arrays found in some languages.

Array Types

Array types are reference types, just as classes are. Instances of arrays are objects, just as the instances of a class are.[4] Unlike classes, array types do not have to be defined. Simply place square brackets after the element type. For example, the following code declares three variables of array type:

```
byte b;                         // byte is a primitive type
byte[] arrayOfBytes;            // byte[] is an array of byte values
byte[][] arrayOfArrayOfBytes;   // byte[][] is an array of byte[]
String[] points;                // String[] is an array of strings
```

The length of an array is not part of the array type. It is not possible, for example, to declare a method that expects an array of exactly four `int` values, for example. If a method parameter is of type `int[]`, a caller can pass an array with any number (including zero) of elements.

Array types are not classes, but array instances are objects. This means that arrays inherit the methods of `java.lang.Object`. Arrays implement the `Cloneable` interface and override the `clone()` method to guarantee that an array can always be cloned and that `clone()` never throws a `CloneNotSupportedException`. Arrays also implement `Serializable` so that any array can be serialized if its element type can be serialized. Finally, all arrays have a `public final int` field named `length` that specifies the number of elements in the array.

Array type widening conversions

Because arrays extend `Object` and implement the `Cloneable` and `Serializable` interfaces, any array type can be widened to any of these three types. But certain

4 There is a terminology difficulty when discussing arrays. Unlike with classes and their instances, we use the term "array" for both the array type and the array instance. In practice, it is usually clear from context whether a type or a value is being discussed.

array types can also be widened to other array types. If the element type of an array is a reference type `T`, and `T` is assignable to a type `S`, the array type `T[]` is assignable to the array type `S[]`. Note that there are no widening conversions of this sort for arrays of a given primitive type. As examples, the following lines of code show legal array widening conversions:

```
String[] arrayOfStrings;      // Created elsewhere
int[][] arrayOfArraysOfInt;   // Created elsewhere
// String is assignable to Object,
// so String[] is assignable to Object[]
Object[] oa = arrayOfStrings;
// String implements Comparable, so a String[] can
// be considered a Comparable[]
Comparable[] ca = arrayOfStrings;
// An int[] is an Object, so int[][] is assignable to Object[]
Object[] oa2 = arrayOfArraysOfInt;
// All arrays are cloneable, serializable Objects
Object o = arrayOfStrings;
Cloneable c = arrayOfArraysOfInt;
Serializable s = arrayOfArraysOfInt[0];
```

This ability to widen an array type to another array type means that the compile-time type of an array is not always the same as its runtime type.

This widening is known as *array covariance*—and as we shall see in "Wildcards" on page 146 it is regarded by modern standards as a historical artifact and a misfeature, because of the mismatch between compile and runtime typing that it exposes.

The compiler must usually insert runtime checks before any operation that stores a reference value into an array element to ensure that the runtime type of the value matches the runtime type of the array element. If the runtime check fails, an `ArrayStoreException` is thrown.

C compatibility syntax

As we've seen, an array type is written simply by placing brackets after the element type. For compatibility with C and C++, however, Java supports an alternative syntax in variable declarations: brackets may be placed after the name of the variable instead of, or in addition to, the element type. This applies to local variables, fields, and method parameters. For example:

```
// This line declares local variables of type int, int[] and int[][]
int justOne, arrayOfThem[], arrayOfArrays[][];

// These three lines declare fields of the same array type:
public String[][] aas1;   // Preferred Java syntax
public String aas2[][];   // C syntax
public String[] aas3[];   // Confusing hybrid syntax
```

```
// This method signature includes two parameters with the same type
public static double dotProduct(double[] x, double y[]) { ... }
```

This compatibility syntax is extremely uncommon, and you should not use it.

Creating and Initializing Arrays

To create an array value in Java, you use the `new` keyword, just as you do to create an object. Array types don't have constructors, but you are required to specify a length whenever you create an array. Specify the desired size of your array as a nonnegative integer between square brackets:

```
// Create a new array to hold 1024 bytes
byte[] buffer = new byte[1024];
// Create an array of 50 references to strings
String[] lines = new String[50];
```

When you create an array with this syntax, each of the array elements is automatically initialized to the same default value that is used for the fields of a class: `false` for `boolean` elements, *\u0000* for `char` elements, 0 for integer elements, 0.0 for floating-point elements, and `null` for elements of reference type.

Array creation expressions can also be used to create and initialize a multidimensional array of arrays. This syntax is somewhat more complicated and is explained later in this section.

Array initializers

To create an array and initialize its elements in a single expression, omit the array length and follow the square brackets with a comma-separated list of expressions within curly braces. The type of each expression must be assignable to the element type of the array, of course. The length of the array that is created is equal to the number of expressions. It is legal, but not necessary, to include a trailing comma following the last expression in the list. For example:

```
String[] greetings = new String[] { "Hello", "Hi", "Howdy" };
int[] smallPrimes = new int[] { 2, 3, 5, 7, 11, 13, 17, 19, };
```

Note that this syntax allows arrays to be created, initialized, and used without ever being assigned to a variable. In a sense, these array creation expressions are anonymous array literals. Here are examples:

```
// Call a method, passing an anonymous array literal that
// contains two strings
String response = askQuestion("Do you want to quit?",
                              new String[] {"Yes", "No"});
```

```
// Call another method with an anonymous array (of anonymous objects)
double d = computeAreaOfTriangle(new Point[] { new Point(1,2),
                                               new Point(3,4),
                                               new Point(3,2) });
```

When an array initializer is part of a variable declaration, you may omit the `new` keyword and element type and list the desired array elements within curly braces:

```
String[] greetings = { "Hello", "Hi", "Howdy" };
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128};
```

Array literals are created and initialized when the program is run, not when the program is compiled. Consider the following array literal:

```
int[] perfectNumbers = {6, 28};
```

This is compiled into Java byte codes that are equivalent to:

```
int[] perfectNumbers = new int[2];
perfectNumbers[0] = 6;
perfectNumbers[1] = 28;
```

The fact that Java does all array initialization at runtime has an important corollary. It means that the expressions in an array initializer may be computed at runtime and need not be compile-time constants. For example:

```
Point[] points = { circle1.getCenterPoint(), circle2.getCenterPoint() };
```

Using Arrays

Once an array has been created, you are ready to start using it. The following sections explain basic access to the elements of an array and cover common idioms of array usage such as iterating through the elements of an array and copying an array or part of an array.

Accessing array elements

The elements of an array are variables. When an array element appears in an expression, it evaluates to the value held in the element. And when an array element appears on the left-hand side of an assignment operator, a new value is stored into that element. Unlike a normal variable, however, an array element has no name, only a number. Array elements are accessed using a square bracket notation. If `a` is an expression that evaluates to an array reference, you index that array and refer to a specific element with `a[i]`, where `i` is an integer literal or an expression that evaluates to an `int`. For example:

```
// Create an array of two strings
String[] responses = new String[2];
responses[0] = "Yes";  // Set the first element of the array
responses[1] = "No";   // Set the second element of the array

// Now read these array elements
```

```
System.out.println(question + " (" + responses[0] + "/" +
                   responses[1] + " ): ");

// Both the array reference and the array index may be more complex
double datum = data.getMatrix()[data.row() * data.numColumns() +
                   data.column()];
```

The array index expression must be of type `int`, or a type that can be widened to an `int`: `byte`, `short`, or even `char`. It is obviously not legal to index an array with a `boolean`, `float`, or `double` value. Remember that the `length` field of an array is an `int` and that arrays may not have more than `Integer.MAX_VALUE` elements. Indexing an array with an expression of type `long` generates a compile-time error, even if the value of that expression at runtime would be within the range of an `int`.

Array bounds

Remember that the first element of an array `a` is `a[0]` , the second element is `a[1]`, and the last is `a[a.length-1]`.

A common bug involving arrays is use of an index that is too small (a negative index) or too large (greater than or equal to the array `length`). In languages like C or C++, accessing elements before the beginning or after the end of an array yields unpredictable behavior that can vary from invocation to invocation and platform to platform. Such bugs may not always be caught, and if a failure occurs, it may be at some later time. While it is just as easy to write faulty array indexing code in Java, Java guarantees predictable results by checking every array access at runtime. If an array index is too small or too large, Java immediately throws an `ArrayIndexOutOfBoundsException`.

Iterating arrays

It is common to write loops that iterate through each of the elements of an array in order to perform some operation on it. This is typically done with a `for` loop. The following code, for example, computes the sum of an array of integers:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
int sumOfPrimes = 0;
for(int i = 0; i < primes.length; i++)
    sumOfPrimes += primes[i];
```

The structure of this `for` loop is idiomatic, and you'll see it frequently. Java also has the foreach syntax that we've already met. The summing code could be rewritten succinctly as follows:

```
for(int p : primes) sumOfPrimes += p;
```

Copying arrays

All array types implement the `Cloneable` interface, and any array can be copied by invoking its `clone()` method. Note that a cast is required to convert the return value

to the appropriate array type, but that the `clone()` method of arrays is guaranteed not to throw `CloneNotSupportedException`:

```
int[] data = { 1, 2, 3 };
int[] copy = (int[]) data.clone();
```

The `clone()` method makes a shallow copy. If the element type of the array is a reference type, only the references are copied, not the referenced objects themselves. Because the copy is shallow, any array can be cloned, even if the element type is not itself `Cloneable`.

Sometimes you simply want to copy elements from one existing array to another existing array. The `System.arraycopy()` method is designed to do this efficiently, and you can assume that Java VM implementations perform this method using high-speed block copy operations on the underlying hardware.

`arraycopy()` is a straightforward function that is difficult to use only because it has five arguments to remember. First pass the source array from which elements are to be copied. Second, pass the index of the start element in that array. Pass the destination array and the destination index as the third and fourth arguments. Finally, as the fifth argument, specify the number of elements to be copied.

`arraycopy()` works correctly even for overlapping copies within the same array. For example, if you've "deleted" the element at index `0` from array `a` and want to shift the elements between indexes `1` and `n` down one so that they occupy indexes `0` through `n-1` you could do this:

```
System.arraycopy(a, 1, a, 0, n);
```

Array utilities

The `java.util.Arrays` class contains a number of static utility methods for working with arrays. Most of these methods are heavily overloaded, with versions for arrays of each primitive type and another version for arrays of objects. The `sort()` and `binarySearch()` methods are particularly useful for sorting and searching arrays. The `equals()` method allows you to compare the content of two arrays. The `Arrays.toString()` method is useful when you want to convert array content to a string, such as for debugging or logging output.

The `Arrays` class also includes `deepEquals()`, `deepHashCode()`, and `deepToString()` methods that work correctly for multidimensional arrays.

Multidimensional Arrays

As we've seen, an array type is written as the element type followed by a pair of square brackets. An array of `char` is `char[]`, and an array of arrays of `char` is `char[][]`. When the elements of an array are themselves arrays, we say that the array is *multidimensional*. In order to work with multidimensional arrays, you need to understand a few additional details.

Imagine that you want to use a multidimensional array to represent a multiplication table:

```
int[][] products;      // A multiplication table
```

Each of the pairs of square brackets represents one dimension, so this is a two-dimensional array. To access a single `int` element of this two-dimensional array, you must specify two index values, one for each dimension. Assuming that this array was actually initialized as a multiplication table, the `int` value stored at any given element would be the product of the two indexes. That is, `products[2][4]` would be 8, and `products[3][7]` would be 21.

To create a new multidimensional array, use the `new` keyword and specify the size of both dimensions of the array. For example:

```
int[][] products = new int[10][10];
```

In some languages, an array like this would be created as a single block of 100 `int` values. Java does not work this way. This line of code does three things:

- Declares a variable named `products` to hold an array of arrays of `int`.
- Creates a 10-element array to hold 10 arrays of `int`.
- Creates 10 more arrays, each of which is a 10-element array of `int`. It assigns each of these 10 new arrays to the elements of the initial array. The default value of every `int` element of each of these 10 new arrays is 0.

To put this another way, the previous single line of code is equivalent to the following code:

```
int[][] products = new int[10][]; // An array to hold 10 int[] values
for(int i = 0; i < 10; i++)       // Loop 10 times...
    products[i] = new int[10];    // ...and create 10 arrays
```

The `new` keyword performs this additional initialization automatically for you. It works with arrays with more than two dimensions as well:

```
float[][][] globalTemperatureData = new float[360][180][100];
```

When using `new` with multidimensional arrays, you do not have to specify a size for all dimensions of the array, only the leftmost dimension or dimensions. For example, the following two lines are legal:

```
float[][][] globalTemperatureData = new float[360][][];
float[][][] globalTemperatureData = new float[360][180][];
```

The first line creates a single-dimensional array, where each element of the array can hold a `float[][]`. The second line creates a two-dimensional array, where each element of the array is a `float[]`. If you specify a size for only some of the dimensions of an array, however, those dimensions must be the leftmost ones. The following lines are not legal:

```
float[][][] globalTemperatureData = new float[360][][100];  // Error!
float[][][] globalTemperatureData = new float[][180][100];  // Error!
```

Like a one-dimensional array, a multidimensional array can be initialized using an array initializer. Simply use nested sets of curly braces to nest arrays within arrays. For example, we can declare, create, and initialize a 5 × 5 multiplication table like this:

```
int[][] products = { {0, 0, 0, 0, 0},
                     {0, 1, 2, 3, 4},
                     {0, 2, 4, 6, 8},
                     {0, 3, 6, 9, 12},
                     {0, 4, 8, 12, 16} };
```

Or, if you want to use a multidimensional array without declaring a variable, you can use the anonymous initializer syntax:

```
boolean response = bilingualQuestion(question, new String[][] {
                                                  { "Yes", "No" },
                                                  { "Oui", "Non" }});
```

When you create a multidimensional array using the `new` keyword, it is usually good practice to only use *rectangular* arrays: one in which all the array values for a given dimension have the same size.

Reference Types

Now that we've covered arrays and introduced classes and objects, we can turn to a more general description of *reference types*. Classes and arrays are two of Java's five kinds of reference types. Classes were introduced earlier and are covered in complete detail, along with *interfaces*, in Chapter 3. Enumerated types and annotation types are reference types introduced in Chapter 4.

This section does not cover specific syntax for any particular reference type, but instead explains the general behavior of reference types and illustrates how they differ from Java's primitive types. In this section, the term *object* refers to a value or instance of any reference type, including arrays.

Reference Versus Primitive Types

Reference types and objects differ substantially from primitive types and their primitive values:

- Eight primitive types are defined by the Java language, and the programmer cannot define new primitive types. Reference types are user-defined, so there is an unlimited number of them. For example, a program might define a class named `Point` and use objects of this newly defined type to store and manipulate *x,y* points in a Cartesian coordinate system.
- Primitive types represent single values. Reference types are aggregate types that hold zero or more primitive values or objects. Our hypothetical `Point` class, for

example, might hold two `double` values to represent the *x* and *y* coordinates of the points. The `char[]` and `Point[]` array types are aggregate types because they hold a sequence of primitive `char` values or `Point` objects.

- Primitive types require between one and eight bytes of memory. When a primitive value is stored in a variable or passed to a method, the computer makes a copy of the bytes that hold the value. Objects, on the other hand, may require substantially more memory. Memory to store an object is dynamically allocated on the heap when the object is created and this memory is automatically "garbage collected" when the object is no longer needed.

When an object is assigned to a variable or passed to a method, the memory that represents the object is not copied. Instead, only a reference to that memory is stored in the variable or passed to the method.

References are completely opaque in Java and the representation of a reference is an implementation detail of the Java runtime. If you are a C programmer, however, you can safely imagine a reference as a pointer or a memory address. Remember, though, that Java programs cannot manipulate references in any way.

Unlike pointers in C and C++, references cannot be converted to or from integers, and they cannot be incremented or decremented. C and C++ programmers should also note that Java does not support the `&` address-of operator or the `*` and `->` dereference operators.

Manipulating Objects and Reference Copies

The following code manipulates a primitive `int` value:

```
int x = 42;
int y = x;
```

After these lines execute, the variable `y` contains a copy of the value held in the variable `x`. Inside the Java VM, there are two independent copies of the 32-bit integer 42.

Now think about what happens if we run the same basic code but use a reference type instead of a primitive type:

```
Point p = new Point(1.0, 2.0);
Point q = p;
```

After this code runs, the variable `q` holds a copy of the reference held in the variable `p`. There is still only one copy of the `Point` object in the VM, but there are now two copies of the reference to that object. This has some important implications. Suppose the two previous lines of code are followed by this code:

```
System.out.println(p.x);  // Print out the x coordinate of p: 1.0
q.x = 13.0;               // Now change the X coordinate of q
System.out.println(p.x);  // Print out p.x again; this time it is 13.0
```

Because the variables `p` and `q` hold references to the same object, either variable can be used to make changes to the object, and those changes are visible through the other variable as well. As arrays are a kind of object then the same thing happens with arrays, as illustrated by the following code:

```
// greet holds an array reference
char[] greet = { 'h','e','l','l','o' };
char[] cuss = greet;             // cuss holds the same reference
cuss[4] = '!';                   // Use reference to change an element
System.out.println(greet);       // Prints "hell!"
```

A similar difference in behavior between primitive types and reference types occurs when arguments are passed to methods. Consider the following method:

```
void changePrimitive(int x) {
    while(x > 0) {
        System.out.println(x--);
    }
}
```

When this method is invoked, the method is given a copy of the argument used to invoke the method in the parameter `x`. The code in the method uses `x` as a loop counter and decrements it to zero. Because `x` is a primitive type, the method has its own private copy of this value, so this is a perfectly reasonable thing to do.

On the other hand, consider what happens if we modify the method so that the parameter is a reference type:

```
void changeReference(Point p) {
    while(p.x > 0) {
        System.out.println(p.x--);
    }
}
```

When this method is invoked, it is passed a private copy of a reference to a `Point` object and can use this reference to change the `Point` object. For example, consider the following:

```
Point q = new Point(3.0, 4.5);  // A point with an x coordinate of 3
changeReference(q);             // Prints 3,2,1 and modifies the Point
System.out.println(q.x);        // The x coordinate of q is now 0!
```

When the `changeReference()` method is invoked, it is passed a copy of the reference held in variable `q`. Now both the variable `q` and the method parameter `p` hold references to the same object. The method can use its reference to change the contents of the object. Note, however, that it cannot change the contents of the variable `q`. In other words, the method can change the `Point` object beyond recognition, but it cannot change the fact that the variable `q` refers to that object.

Comparing Objects

We've seen that primitive types and reference types differ significantly in the way they are assigned to variables, passed to methods, and copied. The types also differ in the way they are compared for equality. When used with primitive values, the equality operator (==) simply tests whether two values are identical (i.e., whether they have exactly the same bits). With reference types, however, == compares references, not actual objects. In other words, == tests whether two references refer to the same object; it does not test whether two objects have the same content. Here's an example:

```
String letter = "o";
String s = "hello";              // These two String objects
String t = "hell" + letter;      // contain exactly the same text.
if (s == t) System.out.println("equal"); // But they are not equal!

byte[] a = { 1, 2, 3 };
// A copy with identical content.
byte[] b = (byte[]) a.clone();
if (a == b) System.out.println("equal"); // But they are not equal!
```

When working with reference types, there are two kinds of equality: equality of reference and equality of object. It is important to distinguish between these two kinds of equality. One way to do this is to use the word "identical" when talking about equality of references and the word "equal" when talking about two distinct objects that have the same content. To test two nonidentical objects for equality, pass one of them to the `equals()` method of the other:

```
String letter = "o";
String s = "hello";              // These two String objects
String t = "hell" + letter;      // contain exactly the same text.
if (s.equals(t)) {               // And the equals() method
    System.out.println("equal"); // tells us so.
}
```

All objects inherit an `equals()` method (from `Object`), but the default implementation simply uses == to test for identity of references, not equality of content. A class that wants to allow objects to be compared for equality can define its own version of the `equals()` method. Our `Point` class does not do this, but the `String` class does, as indicated in the code example. You can call the `equals()` method on an array, but it is the same as using the == operator, because arrays always inherit the default `equals()` method that compares references rather than array content. You can compare arrays for equality with the convenience method `java.util.Arrays.equals()`.

Boxing and Unboxing Conversions

Primitive types and reference types behave quite differently. It is sometimes useful to treat primitive values as objects, and for this reason, the Java platform includes *wrapper classes* for each of the primitive types. `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double` are immutable, final classes whose instances each

hold a single primitive value. These wrapper classes are usually used when you want to store primitive values in collections such as `java.util.List`:

```
// Create a List collection
List numbers = new ArrayList();
// Store a wrapped primitive
numbers.add(new Integer(-1));
// Extract the primitive value
int i = ((Integer)numbers.get(0)).intValue();
```

Java allows types of conversions known as boxing and unboxing conversions. Boxing conversions convert a primitive value to its corresponding wrapper object and unboxing conversions do the opposite. You may explicitly specify a boxing or unboxing conversion with a cast, but this is unnecessary, as these conversions are automatically performed when you assign a value to a variable or pass a value to a method. Furthermore, unboxing conversions are also automatic if you use a wrapper object when a Java operator or statement expects a primitive value. Because Java performs boxing and unboxing automatically, this language feature is often known as *autoboxing*.

Here are some examples of automatic boxing and unboxing conversions:

```
Integer i = 0;   // int literal 0 boxed to an Integer object
Number n = 0.0f; // float literal boxed to Float and widened to Number
Integer i = 1;   // this is a boxing conversion
int j = i;       // i is unboxed here
i++;             // i is unboxed, incremented, and then boxed up again
Integer k = i+2; // i is unboxed and the sum is boxed up again
i = null;
j = i;           // unboxing here throws a NullPointerException
```

Autoboxing makes dealing with collections much easier as well. Let's look at an example that uses Java's *generics* (a language feature we'll meet properly in "Java Generics" on page 142) that allows us to restrict what types can be put into lists and other collections:

```
List<Integer> numbers = new ArrayList<>(); // Create a List of Integer
numbers.add(-1);                           // Box int to Integer
int i = numbers.get(0);                    // Unbox Integer to int
```

Packages and the Java Namespace

A *package* is a named collection of classes, interfaces, and other reference types. Packages serve to group related classes and define a namespace for the classes they contain.

The core classes of the Java platform are in packages whose names begin with `java`. For example, the most fundamental classes of the language are in the package `java.lang`. Various utility classes are in `java.util`. Classes for input and output are in `java.io`, and classes for networking are in `java.net`. Some of these packages contain subpackages, such as `java.lang.reflect` and `java.util.regex`.

Extensions to the Java platform that have been standardized by Oracle (or originally Sun) typically have package names that begin with `javax`. Some of these extensions, such as `javax.swing` and its myriad subpackages, were later adopted into the core platform itself. Finally, the Java platform also includes several "endorsed standards," which have packages named after the standards body that created them, such as `org.w3c` and `org.omg`.

Every class has both a simple name, which is the name given to it in its definition, and a fully qualified name, which includes the name of the package of which it is a part. The `String` class, for example, is part of the `java.lang` package, so its fully qualified name is `java.lang.String`.

This section explains how to place your own classes and interfaces into a package and how to choose a package name that won't conflict with anyone else's package name. Next, it explains how to selectively import type names or static members into the namespace so that you don't have to type the package name of every class or interface you use.

Package Declaration

To specify the package a class is to be part of, you use a `package` declaration. The `package` keyword, if it appears, must be the first token of Java code (i.e., the first thing other than comments and space) in the Java file. The keyword should be followed by the name of the desired package and a semicolon. Consider a Java file that begins with this directive:

```
package org.apache.commons.net;
```

All classes defined by this file are part of the package `org.apache.commons.net`.

If no `package` directive appears in a Java file, all classes defined in that file are part of an unnamed default package. In this case, the qualified and unqualified names of a class are the same.

The possibility of naming conflicts means that you should not use the default package. As your project grows more complicated, conflicts become almost inevitable—much better to create packages right from the start.

Globally Unique Package Names

One of the important functions of packages is to partition the Java namespace and prevent name collisions between classes. It is only their package names that keep the `java.util.List` and `java.awt.List` classes distinct, for example. In order for this to work, however, package names must themselves be distinct. As the developer of Java, Oracle controls all package names that begin with `java`, `javax`, and `sun`.

One scheme in common use is to use your domain name, with its elements reversed, as the prefix for all your package names. For example, the Apache Project produces a networking library as part of the Apache Commons project. The Commons project can be found at *http://commons.apache.org/* and accordingly, the package name used for the networking library is `org.apache.commons.net`.

Note that these package-naming rules apply primarily to API developers. If other programmers will be using classes that you develop along with unknown other classes, it is important that your package name be globally unique. On the other hand, if you are developing a Java application and will not be releasing any of the classes for reuse by others, you know the complete set of classes that your application will be deployed with and do not have to worry about unforeseen naming conflicts. In this case, you can choose a package naming scheme for your own convenience rather than for global uniqueness. One common approach is to use the application name as the main package name (it may have subpackages beneath it).

Importing Types

When referring to a class or interface in your Java code, you must, by default, use the fully qualified name of the type, including the package name. If you're writing code to manipulate a file and need to use the `File` class of the `java.io` package, you must type `java.io.File`. This rule has three exceptions:

- Types from the package `java.lang` are so important and so commonly used that they can always be referred to by their simple names.
- The code in a type `p.T` may refer to other types defined in the package `p` by their simple names.
- Types that have been *imported* into the namespace with an `import` declaration may be referred to by their simple names.

The first two exceptions are known as "automatic imports." The types from `java.lang` and the current package are "imported" into the namespace so that they can be used without their package name. Typing the package name of commonly used types that are not in `java.lang` or the current package quickly becomes tedious, and so it is also possible to explicitly import types from other packages into the namespace. This is done with the `import` declaration.

`import` declarations must appear at the start of a Java file, immediately after the `package` declaration, if there is one, and before any type definitions. You may use any number of `import` declarations in a file. An `import` declaration applies to all type definitions in the file (but not to any `import` declarations that follow it).

The `import` declaration has two forms. To import a single type into the namespace, follow the `import` keyword with the name of the type and a semicolon:

```
import java.io.File;    // Now we can type File instead of java.io.File
```

This is known as the "single type import" declaration.

The other form of `import` is the "on-demand type import." In this form, you specify the name of a package followed by the characters `.*` to indicate that any type from that package may be used without its package name. Thus, if you want to use several other classes from the `java.io` package in addition to the `File` class, you can simply import the entire package:

```
import java.io.*;   // Use simple names for all classes in java.io
```

This on-demand `import` syntax does not apply to subpackages. If I import the `java.util` package, I must still refer to the `java.util.zip.ZipInputStream` class by its fully qualified name.

Using an on-demand type import declaration is not the same as explicitly writing out a single type import declaration for every type in the package. It is more like an explicit single type import for every type in the package *that you actually use* in your code. This is the reason it's called "on demand"; types are imported as you use them.

Naming conflicts and shadowing

`import` declarations are invaluable to Java programming. They do expose us to the possibility of naming conflicts, however. Consider the packages `java.util` and `java.awt`. Both contain types named `List`.

`java.util.List` is an important and commonly used interface. The `java.awt` package contains a number of important types that are commonly used in client-side applications, but `java.awt.List` has been superseded and is not one of these important types. It is illegal to import both `java.util.List` and `java.awt.List` in the same Java file. The following single type import declarations produce a compilation error:

```
import java.util.List;
import java.awt.List;
```

Using on-demand type imports for the two packages is legal:

```
import java.util.*;  // For collections and other utilities.
import java.awt.*;   // For fonts, colors, and graphics.
```

Difficulty arises, however, if you actually try to use the type `List`. This type can be imported "on demand" from either package, and any attempt to use `List` as an unqualified type name produces a compilation error. The workaround, in this case, is to explicitly specify the package name you want.

Because `java.util.List` is much more commonly used than `java.awt.List`, it is useful to combine the two on-demand type import declarations with a single-type import declaration that serves to disambiguate what we mean when we say `List`:

```
import java.util.*;    // For collections and other utilities.
import java.awt.*;     // For fonts, colors, and graphics.
import java.util.List; // To disambiguate from java.awt.List
```

With these `import` declarations in place, we can use `List` to mean the `java.util.List` interface. If we actually need to use the `java.awt.List` class, we can still do so as long as we include its package name. There are no other naming conflicts between `java.util` and `java.awt`, and their types will be imported "on demand" when we use them without a package name.

Importing Static Members

As well as types, you can import the static members of types using the keywords `import static`. (Static members are explained in Chapter 3. If you are not already familiar with them, you may want to come back to this section later.) Like type import declarations, these static import declarations come in two forms: single static member import and on-demand static member import. Suppose, for example, that you are writing a text-based program that sends a lot of output to `System.out`. In this case, you might use this single static member import to save yourself typing:

```
import static java.lang.System.out;
```

With this import in place, you can then use `out.println()` instead of `System.out.println()`. Or suppose you are writing a program that uses many of the trigonometric and other functions of the `Math` class. In a program that is clearly focused on numerical methods like this, having to repeatedly type the class name "Math" does not add clarity to your code; it just gets in the way. In this case, an on-demand static member import may be appropriate:

```
import static java.lang.Math.*
```

With this import declaration, you are free to write concise expressions like `sqrt(abs(sin(x)))` without having to prefix the name of each static method with the class name `Math`.

Another important use of `import static` declarations is to import the names of constants into your code. This works particularly well with enumerated types (see Chapter 4). Suppose, for example, that you want to use the values of this enumerated type in code you are writing:

```
package climate.temperate;
enum Seasons { WINTER, SPRING, SUMMER, AUTUMN };
```

You could import the type `climate.temperate.Seasons` and then prefix the constants with the type name: `Seasons.SPRING`. For more concise code, you could import the enumerated values themselves:

```
import static climate.temperate.Seasons.*;
```

Using static member import declarations for constants is generally a better technique than implementing an interface that defines the constants.

Static member imports and overloaded methods

A static import declaration imports a *name*, not any one specific member with that name. Because Java allows method overloading and allows a type to have fields and methods with the same name, a single static member import declaration may actually import more than one member. Consider this code:

```
import static java.util.Arrays.sort;
```

This declaration imports the name "sort" into the namespace, not any one of the 19 `sort()` methods defined by `java.util.Arrays`. If you use the imported name `sort` to invoke a method, the compiler will look at the types of the method arguments to determine which method you mean.

It is even legal to import static methods with the same name from two or more different types as long as the methods all have different signatures. Here is one natural example:

```
import static java.util.Arrays.sort;
import static java.util.Collections.sort;
```

You might expect that this code would cause a syntax error. In fact, it does not because the `sort()` methods defined by the `Collections` class have different signatures than all of the `sort()` methods defined by the `Arrays` class. When you use the name "sort" in your code, the compiler looks at the types of the arguments to determine which of the 21 possible imported methods you mean.

Java File Structure

This chapter has taken us from the smallest to the largest elements of Java syntax, from individual characters and tokens to operators, expressions, statements, and methods, and on up to classes and packages. From a practical standpoint, the unit of Java program structure you will be dealing with most often is the Java file. A Java file is the smallest unit of Java code that can be compiled by the Java compiler. A Java file consists of:

- An optional `package` directive
- Zero or more `import` or `import static` directives
- One or more type definitions

These elements can be interspersed with comments, of course, but they must appear in this order. This is all there is to a Java file. All Java statements (except the `package` and `import` directives, which are not true statements) must appear within methods, and all methods must appear within a type definition.

Java files have a couple of other important restrictions. First, each file can contain at most one top-level class that is declared `public`. A `public` class is one that is designed for use by other classes in other packages. A class can contain any number

of nested or inner classes that are `public`. We'll see more about the `public` modifier and nested classes in Chapter 3.

The second restriction concerns the filename of a Java file. If a Java file contains a `public` class, the name of the file must be the same as the name of the class, with the extension *.java* appended. Therefore, if `Point` is defined as a `public` class, its source code must appear in a file named *Point.java*. Regardless of whether your classes are `public` or not, it is good programming practice to define only one per file and to give the file the same name as the class.

When a Java file is compiled, each of the classes it defines is compiled into a separate *class* file that contains Java byte codes to be interpreted by the Java Virtual Machine. A class file has the same name as the class it defines, with the extension *.class* appended. Thus, if the file *Point.java* defines a class named `Point`, a Java compiler compiles it to a file named *Point.class*. On most systems, class files are stored in directories that correspond to their package names. The class `com.davidflanagan.examples.Point` is thus defined by the class file *com/davidflanagan/examples/Point.class*.

The Java interpreter knows where the class files for the standard system classes are located and can load them as needed. When the interpreter runs a program that wants to use a class named `com.davidflanagan.examples.Point`, it knows that the code for that class is located in a directory named *com/davidflanagan/examples/* and, by default, it "looks" in the current directory for a subdirectory of that name. In order to tell the interpreter to look in locations other than the current directory, you must use the `-classpath` option when invoking the interpreter or set the `CLASSPATH` environment variable. For details, see the documentation for the Java interpreter, *java*, in Chapter 8.

Defining and Running Java Programs

A Java program consists of a set of interacting class definitions. But not every Java class or Java file defines a program. To create a program, you must define a class that has a special method with the following signature:

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

This `main()` method is the main entry point for your program. It is where the Java interpreter starts running. This method is passed an array of strings and returns no value. When `main()` returns, the Java interpreter exits (unless `main()` has created separate threads, in which case the interpreter waits for all those threads to exit).

To run a Java program, you run the Java interpreter, *java*, specifying the fully qualified name of the class that contains the `main()` method. Note that you specify the name of the class, *not* the name of the class file that contains the class. Any additional arguments you specify on the command line are passed to the `main()` method as its `String[]` parameter. You may also need to specify the `-classpath` option (or

`-cp`) to tell the interpreter where to look for the classes needed by the program. Consider the following command:

```
java -classpath /opt/Jude com.davidflanagan.jude.Jude datafile.jude
```

`java` is the command to run the Java interpreter. `-classpath /usr/local/Jude` tells the interpreter where to look for *.class* files. `com.davidflanagan.jude.Jude` is the name of the program to run (i.e., the name of the class that defines the `main()` method). Finally, *datafile.jude* is a string that is passed to that `main()` method as the single element of an array of `String` objects.

There is an easier way to run programs. If a program and all its auxiliary classes (except those that are part of the Java platform) have been properly bundled in a Java archive (JAR) file, you can run the program simply by specifying the name of the JAR file. In the next example, we show how to start up the Censum garbage collection log analyzer:

```
java -jar /usr/local/Censum/censum.jar
```

Some operating systems make JAR files automatically executable. On those systems, you can simply say:

```
% /usr/local/Censum/censum.jar
```

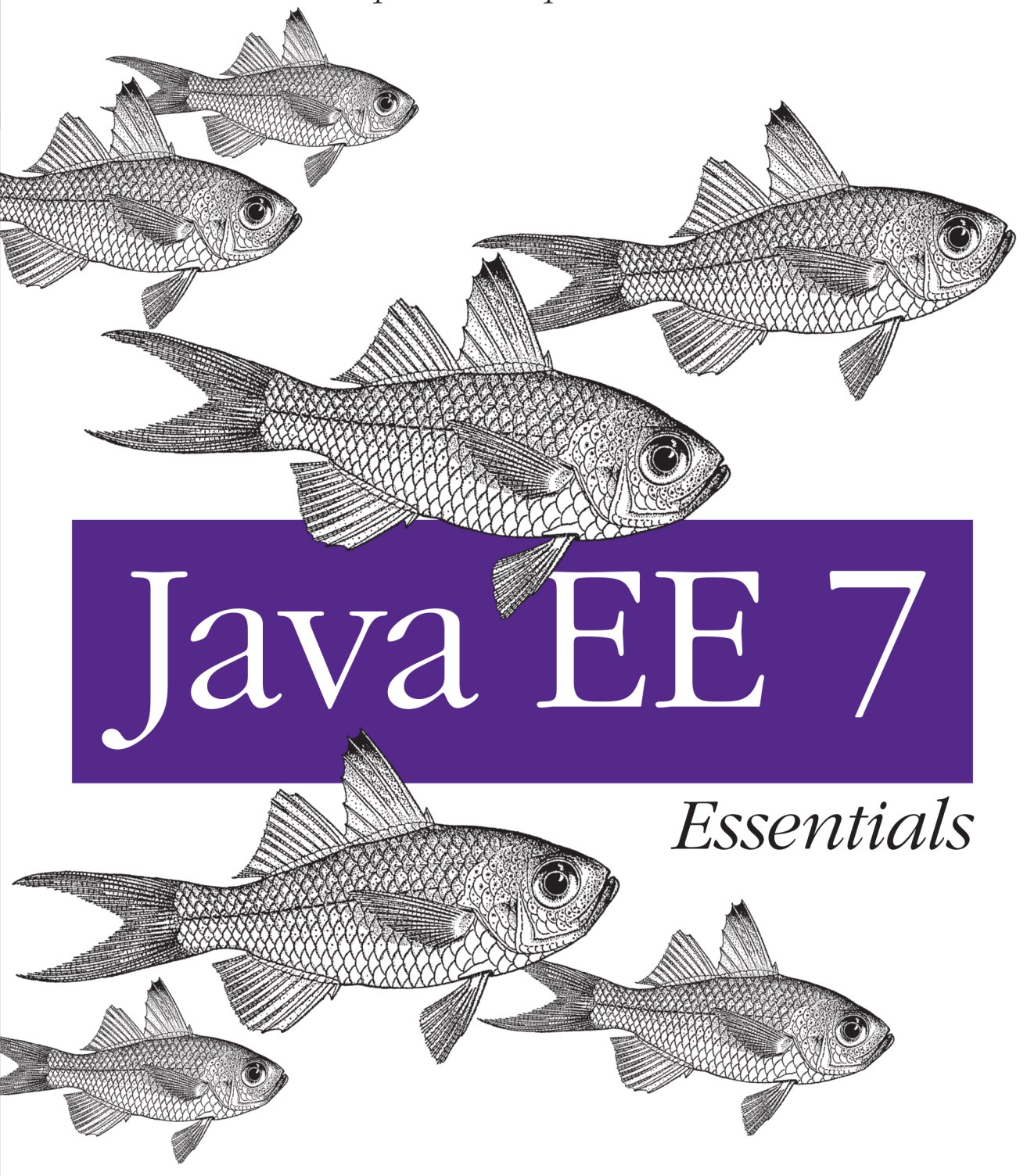
See Chapter 13 for more details on how to execute Java programs.

Summary

In this chapter, we've introduced the basic syntax of the Java language. Due to the interlocking nature of the syntax of programming languages, it is perfectly fine if you don't feel at this point that you have completely grasped all of the syntax of the language. It is by practice that we acquire proficiency in any language, human or computer.

It is also worth observing that some parts of syntax are far more regularly used than others. For example, the `strictfp` and `assert` keywords are almost never used. Rather than trying to grasp every aspect of Java's syntax, it is far better to begin to acquire facility in the core aspects of Java and then return to any details of syntax that may still be troubling you. With this in mind, let's move to the next chapter and begin to discuss the classes and objects that are so central to Java and the basics of Java's approach to object-oriented programming.

Enterprise Developer Handbook



Java EE 7

Essentials

O'REILLY®

Arun Gupta

Java EE 7 Essentials

Arun Gupta

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

CHAPTER 2
Servlets

Servlets are defined as JSR 340, and the complete specification can be downloaded.

A servlet is a web component hosted in a servlet container and generates dynamic content. The web clients interact with a servlet using a request/response pattern. The servlet container is responsible for the life cycle of the servlet, receives requests and sends responses, and performs any other encoding/decoding required as part of that.

WebServlet

A servlet is defined using the `@WebServlet` annotation on a POJO, and must extend the `javax.servlet.http.HttpServlet` class.

Here is a sample servlet definition:

```
@WebServlet("/account")
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The fully qualified class name is the default servlet name, and may be overridden using the `name` attribute of the annotation. The servlet may be deployed at multiple URLs:

```
@WebServlet(urlPatterns={"/account", "/accountServlet"})
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The `@WebInitParam` can be used to specify an initialization parameter:

```
@WebServlet(urlPatterns="/account",
            initParams={
               @WebInitParam(name="type", value="checking")
                       }
           )
```

```
public class AccountServlet extends javax.servlet.http.HttpServlet {
  //. . .
}
```

The `HttpServlet` interface has one `do`*`XXX`* method to handle each of HTTP `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, and `TRACE` requests. Typically the developer is concerned with overriding the `doGet` and `doPost` methods. The following code shows a servlet handling the `GET` request:

```
@WebServlet("/account")
public class AccountServlet
  extends javax.servlet.http.HttpServlet {
  @Override
  protected void doGet(
        HttpServletRequest request,
        HttpServletResponse response) {
    //. . .
  }
}
```

In this code:

- The `HttpServletRequest` and `HttpServletResponse` capture the request/response with the web client.
- The request parameters; HTTP headers; different parts of the path such as host, port, and context; and much more information is available from `HttpServletRequest`.

The HTTP cookies can be sent and retrieved as well. The developer is responsible for populating the `HttpServletResponse`, and the container then transmits the captured HTTP headers and/or the message body to the client.

This code shows how an HTTP `GET` request received by a servlet displays a simple response to the client:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
    try (PrintWriter out = response.getWriter()) {
        out.println("<html><head>");
        out.println("<title>MyServlet</title>");
        out.println("</head><body>");
        out.println("<h1>My First Servlet</h1>");
        //. . .
        out.println("</body></html>");
    } finally {
        //. . .
    }
}
```

Request parameters may be passed in `GET` and `POST` requests. In a `GET` request, these parameters are passed in the query string as name/value pairs. Here is a sample URL to invoke the servlet explained earlier with request parameters:

```
. . ./account?tx=10
```

In a `POST` request, the request parameters can also be passed in the posted data that is encoded in the body of the request. In both `GET` and `POST` requests, these parameters can be obtained from `HttpServletRequest`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  String txValue = request.getParameter("tx");
  //. . .
}
```

Request parameters can differ for each request.

Initialization parameters, also known as *init params*, may be defined on a servlet to store startup and configuration information. As explained earlier, `@WebInitParam` is used to specify init params for a servlet:

```
String type = null;

@Override
public void init(ServletConfig config) throws ServletException {
  type = config.getInitParameter("type");
  //. . .
}
```

You can manipulate the default behavior of the servlet's life-cycle call methods by overriding the `init`, `service`, and `destroy` methods of the `javax.servlet.Servlet` interface. Typically, database connections are initialized in `init` and released in `destroy`.

You can also define a servlet using the `servlet` and `servlet-mapping` elements in the deployment descriptor of the web application, *web.xml*. You can define the `AccountServlet` using *web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>AccountServlet</servlet-name>
    <servlet-class>org.sample.AccountServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>AccountServlet</servlet-name>
    <url-pattern>/account</url-pattern>
```

```
  </servlet-mapping>
</web-app>
```

The annotations cover most of the common cases, so *web.xml* is not required in those cases. But some cases, such as ordering of servlets, can only be done using *web.xml*.

If the `metadata-complete` element in *web.xml* is true, then the annotations in the class are not processed:

```
<web-app version="3.1"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
  metadata-complete="true">
  //. . .
</web-app>
```

The values defined in the deployment descriptor override the values defined using annotations.

A servlet is packaged in a web application in a *.war* file. Multiple servlets may be packaged together, and they all share a *servlet context*. The `ServletContext` provides detail about the execution environment of the servlets and is used to communicate with the container—for example, by reading a resource packaged in the web application, writing to a logfile, or dispatching a request.

The `ServletContext` can be obtained from `HttpServletRequest`:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  ServletContext context = request.getServletContext();
  //. . .
}
```

A servlet can send an HTTP cookie, named `JSESSIONID`, to the client for session tracking. This cookie may be marked as `HttpOnly`, which ensures that the cookie is not exposed to client-side scripting code, and thus helps mitigate certains kinds of cross-site scripting attacks:

```
SessionCookieConfig config = request.getServletContext().
                             getSessionCookieConfig();
config.setHttpOnly(true);
```

Alternatively, URL rewriting may be used by the servlet as a basis for session tracking. The `ServletContext.getSessionCookieConfig` method returns `SessionCookieConfig`, which can be used to configure different properties of the cookie.

The `HttpSession` interface can be used to view and manipulate information about a session such as the session identifier and creation time, and to bind objects to the session. A new session object may be created:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  HttpSession session = request.getSession(true);
  //. . .
}
```

The `session.setAttribute` and `session.getAttribute` methods are used to bind objects to the session.

A servlet may forward a request to another servlet if further processing is required. You can achieve this by dispatching the request to a different resource using `RequestDispatcher`, which can be obtained from `HttpServletRequest.getRequestDispatcher` or `ServletContext.getRequestDispatcher`. The former can accept a relative path, whereas the latter can accept a path relative to the current context only:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  request.getRequestDispatcher("bank").forward(request, response);
  //. . .
}
```

In this code, `bank` is another servlet deployed in the same context.

The `ServletContext.getContext` method can be used to obtain `ServletContext` for foreign contexts. It can then be used to obtain a `RequestDispatcher`, which can dispatch requests in that context.

You can redirect a servlet response to another resource by calling the `HttpServletResponse.sendRedirect` method. This sends a temporary redirect response to the client, and the client issues a new request to the specified URL. Note that in this case, the original request object is not available to the redirected URL. The redirect may also be marginally slower because it entails two requests from the client, whereas `forward` is performed within the container:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  //. . .
  response.sendRedirect("http://example.com/SomeOtherServlet");
}
```

Here the response is redirected to the *http://example.com/SomeOtherServlet* URL. Note that this URL could be on a different host/port and may be relative or absolute to the container.

In addition to declaring servlets using `@WebServlet` and *web.xml*, you can define them programmatically using `ServletContext.addServlet` methods. You can do this from the `ServletContainerInitializer.onStartup` or `ServletContextListener.contextInitialized` method. You can read more about this in "Event Listeners" on page 17.

The `ServletContainerInitializer.onStartup` method is invoked when the application is starting up for the given `ServletContext`. The `addServlet` method returns `ServletRegistration.Dynamic`, which can then be used to create URL mappings, set security roles, set initialization parameters, and manage other configuration items:

```
public class MyInitializer implements ServletContainerInitializer {
  @Override
  public void onStartup (Set<Class<?>> clazz, ServletContext context) {
    ServletRegistration.Dynamic reg =
        context.addServlet("MyServlet", "org.example.MyServlet");
    reg.addMapping("/myServlet");
  }
}
```

Servlet Filters

A servlet filter may be used to update the request and response payload and header information from and to the servlet. It is important to realize that filters do not create the response—they only modify or adapt the requests and responses. Authentication, logging, data compression, and encryption are some typical use cases for filters. The filters are packaged along with a servlet and act upon the dynamic or static content.

You can associate filters with a servlet or with a group of servlets and static content by specifying a URL pattern. You define a filter using the `@WebFilter` annotation:

```
@WebFilter("/*")
public class LoggingFilter implements javax.servlet.Filter {
  public void doFilter(HttpServletRequest request,
                       HttpServletResponse response) {
    //. . .
  }
}
```

In the code shown, the `LoggingFilter` is applied to all the servlets and static content pages in the web application.

The `@WebInitParam` may be used to specify initialization parameters here as well.

A filter and the target servlet always execute in the same invocation thread. Multiple filters may be arranged in a filter chain.

You can also define a filter using `<filter>` and `<filter-mapping>` elements in the deployment descriptor:

```
<filter>
  <filter-name>LoggingFilter</filter-name>
  <filter-class>org.sample.LoggingFilter</filter-class>
</filter>
    . . .
<filter-mapping>
  <filter-name>LoggingFilter</filter-name>
```

```
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

In addition to declaring filters using `@WebFilter` and *web.xml*, you can define them programmatically using `ServletContext.addFilter` methods. You can do this from the `ServletContainerInitializer.onStartup` method or the `ServletContextListener.contextInitialized` method. The `addFilter` method returns `ServletRegistration.Dynamic`, which can then be used to add mapping for URL patterns, set initialization parameters, and handle other configuration items:

```
public class MyInitializer implements ServletContainerInitializer {
  public void onStartup (Set<Class<?>> clazz, ServletContext context) {
    FilterRegistration.Dynamic reg =
        context.addFilter("LoggingFilter",
                  "org.example.LoggingFilter");
    reg.addMappingForUrlPatterns(null, false, "/");
  }
}
```

Event Listeners

Event listeners provide life-cycle callback events for `ServletContext`, `HttpSession`, and `ServletRequest` objects. These listeners are classes that implement an interface that supports event notifications for state changes in these objects. Each class is annotated with `@WebListener`, declared in *web.xml*, or registered via one of the `ServletContext.addListener` methods. A typical example of these listeners is where an additional servlet is registered programmatically without an explicit need for the programmer to do so, or a database connection is initialized and restored back at the application level.

There may be multiple listener classes listening to each event type, and they may be specified in the order in which the container invokes the listener beans for each event type. The listeners are notified in the reverse order during application shutdown.

Servlet context listeners listen to the events from resources in that context:

```
@WebListener
public class MyContextListener implements ServletContextListener {

  @Override
  public void contextInitialized(ServletContextEvent sce) {
    ServletContext context = sce.getServletContext();
    //. . .
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
    //. . .
  }
}
```

The `ServletContextAttributeListener` is used to listen for attribute changes in the context:

```
public class MyServletContextAttributeListener
             implements ServletContextAttributeListener {

  @Override
  public void attributeAdded(ServletContextAttributeEvent event) {
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void attributeRemoved(ServletContextAttributeEvent event) {
    //. . .
  }

  @Override
  public void attributeReplaced(ServletContextAttributeEvent event) {
    //. . .
  }

}
```

The `HttpSessionListener` listens to events from resources in that session:

```
@WebListener
public class MySessionListener implements HttpSessionListener {

  @Override
  public void sessionCreated(HttpSessionEvent hse) {
    HttpSession session = hse.getSession();
    //. . .
  }

  @Override
  public void sessionDestroyed(HttpSessionEvent hse) {
    //. . .
  }
}
```

The `HttpSessionActivationListener` is used to listen for events when the session is passivated or activated:

```
public class MyHttpSessionActivationListener
             implements HttpSessionActivationListener {

  @Override
  public void sessionWillPassivate(HttpSessionEvent hse) {
    // ... hse.getSession();
  }

  @Override
```

```
  public void sessionDidActivate(HttpSessionEvent hse) {
    // ...
  }
}
```

The `HttpSessionAttributeListener` is used to listen for attribute changes in the session:

```
public class MyHttpSessionAttributeListener
             implements HttpSessionAttributeListener {

  @Override
  public void attributeAdded(HttpSessionBindingEvent event) {
    HttpSession session = event.getSession();
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void attributeRemoved(HttpSessionBindingEvent event) {
    //. . .
  }

  @Override
  public void attributeReplaced(HttpSessionBindingEvent event) {
    //. . .
  }
}
```

The `HttpSessionBindingListener` is used to listen to events when an object is bound to or unbound from a session:

```
public class MyHttpSessionBindingListener
             implements HttpSessionBindingListener {

  @Override
  public void valueBound(HttpSessionBindingEvent event) {
    HttpSession session = event.getSession();
    //. . . event.getName();
    //. . . event.getValue();
  }

  @Override
  public void valueUnbound(HttpSessionBindingEvent event) {
    //. . .
  }
}
```

The `ServletRequestListener` listens to the events from resources in that request:

```
@WebListener
public class MyRequestListener implements ServletRequestListener {
  @Override
  public void requestDestroyed(ServletRequestEvent sre) {
```

```
    ServletRequest request = sre.getServletRequest();
    //. . .
  }

  @Override
  public void requestInitialized(ServletRequestEvent sre) {
    //. . .
  }
}
```

The `ServletRequestAttributeListener` is used to listen for attribute changes in the request.

There is also `AsyncListener`, which is used to manage async events such as completed, timed out, or an error.

In addition to declaring listeners using `@WebListener` and *web.xml*, you can define them programmatically using `ServletContext.addListener` methods. You can do this from the `ServletContainerInitializer.onStartup` or `ServletContextListener.contextInitialized` method.

The `ServletContainerInitializer.onStartup` method is invoked when the application is starting up for the given `ServletContext`:

```
public class MyInitializer implements ServletContainerInitializer {
  public void onStartup(Set<Class<?>> clazz, ServletContext context) {
    context.addListener("org.example.MyContextListener");
  }
}
```

Asynchronous Support

Server resources are valuable and should be used conservatively. Consider a servlet that has to wait for a JDBC connection to be available from the pool, receiving a JMS message or reading a resource from the filesystem. Waiting for a "long-running" process to return completely blocks the thread—waiting, sitting, and doing nothing—which is not an optimal usage of your server resources. This is where the server can be asynchronously processed such that the control (or thread) is returned to the container to perform other tasks while waiting for the long-running process to complete. The request processing continues in the same thread after the response from the long-running process is returned, or may be dispatched to a new resource from within the long-running process. A typical use case for a long-running process is a chat application.

The asynchronous behavior needs to be explicitly enabled on a servlet. You achieve this by adding the `asyncSupported` attribute on `@WebServlet`:

```
@WebServlet(urlPatterns="/async", asyncSupported=true)
public class MyAsyncServlet extends HttpServlet {
```

```
    //. . .
  }
```

You can also enable the asynchronous behavior by setting the `<async-supported>` element to true in *web.xml* or calling `ServletRegistration.setAsyncSupported(true)` during programmatic registration.

You can then start the asynchronous processing in a separate thread using the `startAsync` method on the request. This method returns `AsyncContext`, which represents the execution context of the asynchronous request. Then you can complete the asynchronous request by calling `AsyncContext.complete` (explicit) or dispatching to another resource (implicit). The container completes the invocation of the asynchronous request in the latter case.

Let's say the long-running process is implemented:

```
class MyAsyncService implements Runnable {
  AsyncContext ac;

  public MyAsyncService(AsyncContext ac) {
    this.ac = ac;
  }

  @Override
  public void run() {
    //. . .
    ac.complete();
  }
}
```

This service may be invoked from the `doGet` method:

```
@Override
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  AsyncContext ac = request.startAsync();
  ac.addListener(new AsyncListener() {
    public void onComplete(AsyncEvent event)
        throws IOException {
      //. . .
    }

    public void onTimeout(AsyncEvent event)
        throws IOException {
      //. . .
    }
    //. . .
  });

  ScheduledThreadPoolExecutor executor = new ScheduledThreadPoolExecutor(10);
  executor.execute(new MyAsyncService(ac));
}
```

In this code, the request is put into asynchronous mode. `AsyncListener` is registered to listen for events when the request processing is complete, has timed out, or resulted in an error. The long-running service is invoked in a separate thread and calls `AsyncContext.complete`, signalling the completion of request processing.

A request may be dispatched from an asynchronous servlet to synchronous, but the other way around is illegal.

The asynchronous behavior is available in the servlet filter as well.

Nonblocking I/O

Servlet 3.0 allowed asynchronous request processing but only permitted traditional I/O, which restricted the scalability of your applications. In a typical application, `ServletInputStream` is read in a `while` loop:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
  ServletInputStream input = request.getInputStream();
  byte[] b = new byte[1024];
  int len = -1;
  while ((len = input.read(b)) != -1) {
    //. . .
  }
}
```

If the incoming data is blocking or streamed slower than the server can read, then the server thread is waiting for that data. The same can happen if the data is written to `ServletOutputStream`. This restricts the scalability of the Web Container.

Nonblocking I/O allows developers to read data as it becomes available or write data when it's possible to do so. This increases not only the scalability of the Web Container but also the number of connections that can be handled simultaneously. Nonblocking I/O only works with async request processing in Servlets, Filters, and Upgrade Processing.

Servlet 3.1 achieves nonblocking I/O by introducing two new interfaces: `ReadListener` and `WriteListener`. These listeners have callback methods that are invoked when the content is available to be read or can be written without blocking.

The `doGet` method needs to be rewritten in this case:

```
AsyncContext context = request.startAsync();
ServletInputStream input = request.getInputStream();
input.setReadListener(new MyReadListener(input, context));
```

Invoking `set`*`XXX`*`Listener` methods indicates that nonblocking I/O is used instead of traditional.

`ReadListener` has three callback methods:

- The `onDataAvailable` callback method is called whenever data can be read without blocking.
- The `onAllDataRead` callback method is invoked whenever data for the current request is completely read.
- The `onError` callback is invoked if there is an error processing the request:

```
@Override
public void onDataAvailable() {
  try {
    StringBuilder sb = new StringBuilder();
    int len = -1;
    byte b[] = new byte[1024];
    while (input.isReady() && (len = input.read(b)) != -1) {
      String data = new String(b, 0, len);
    }
  } catch (IOException ex) {
    //. . .
  }
}

@Override
public void onAllDataRead() {
  context.complete();
}

@Override
public void onError(Throwable t) {
  t.printStackTrace();
  context.complete();
}
```

In this code, the `onDataAvailable` callback is invoked whenever data can be read without blocking. The `ServletInputStream.isReady` method is used to check if data can be read without blocking and then the data is read. `context.complete` is called in `onAllDataRead` and `onError` methods to signal the completion of data read. `ServletInputStream.isFinished` may be used to check the status of a nonblocking I/O read.

At most, one `ReadListener` can be registered on `ServletIntputStream`.

`WriteListener` has two callback methods:

- The `onWritePossible` callback method is called whenever data can be written without blocking.
- The `onError` callback is invoked if there is an error processing the response.

At most, one `WriteListener` can be registered on `ServletOutputStream`. `ServletOutputStream.canWrite` is a new method to check if data can be written without blocking.

Web Fragments

A web fragment is part or all of the *web.xml* file included in a library or framework JAR's *META-INF* directory. If this framework is bundled in the *WEB-INF/lib* directory, the container will pick up and configure the framework without requiring the developer to do it explicitly.

It can include almost all of the elements that can be specified in *web.xml*. However, the top-level element must be `web-fragment` and the corresponding *file* must be called *web-fragment.xml*. This allows logical partitioning of the web application:

```
<web-fragment>
  <filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>org.example.MyFilter</filter-class>
    <init-param>
      <param-name>myInitParam</param-name>
      <param-value>...</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-fragment>
```

The developer can specify the order in which the resources specified in *web.xml* and *web-fragment.xml* need to be loaded. The `<absolute-ordering>` element in *web.xml* is used to specify the exact order in which the resources should be loaded, and the `<ordering>` element within *web-fragment.xml* is used to specify relative ordering. The two orders are mutually exclusive, and absolute ordering overrides relative.

The absolute ordering contains one or more `<name>` elements specifying the name of the resources and the order in which they need to be loaded. Specifying `<others/>` allows for the other resources not named in the ordering to be loaded:

```
<web-app>
  <name>MyApp</name>
  <absolute-ordering>
    <name>MyServlet</name>
    <name>MyFilter</name>
  </absolute-ordering>
</web-app>
```

In this code, the resources specified in *web.xml* are loaded first and followed by `MyServlet` and `MyFilter`.

Zero or one `<before>` and `<after>` elements in `<ordering>` are used to specify the resources that need to be loaded before and after the resource named in the `web-fragment` is loaded:

```
<web-fragment>
  <name>MyFilter</name>
  <ordering>
    <after>MyServlet</after>
  </ordering>
</web-fragment>
```

This code will require the container to load the resource `MyFilter` after the resource `MyServlet` (defined elsewhere) is loaded.

If *web.xml* has `metadata-complete` set to true, then the *web-fragment.xml* file is not processed. The *web.xml* file has the highest precedence when resolving conflicts between *web.xml* and *web-fragment.xml*.

If a *web-fragment.xml* file does not have an `<ordering>` element and *web.xml* does not have an `<absolute-ordering>` element, the resources are assumed to not have any ordering dependency.

Security

Servlets are typically accessed over the Internet, and thus having a security requirement is common. You can specify the servlet security model, including roles, access control, and authentication requirements, using annotations or in *web.xml*.

`@ServletSecurity` is used to specify security constraints on the servlet implementation class for all methods or a specific `do`*XXX* method. The container will enforce that the corresponding `do`*XXX* messages can be invoked by users in the specified roles:

```
@WebServlet("/account")
@ServletSecurity(
  value=@HttpConstraint(rolesAllowed = {"R1"}),
  httpMethodConstraints={
    @HttpMethodConstraint(value="GET",
                          rolesAllowed="R2"),
    @HttpMethodConstraint(value="POST",
                          rolesAllowed={"R3", "R4"})
  }
)
public class AccountServlet
             extends javax.servlet.http.HttpServlet {
  //. . .
}
```

In this code, `@HttpMethodConstraint` is used to specify that the `doGet` method can be invoked by users in the `R2` role, and the `doPost` method can be invoked by users in the `R3` and `R4` roles. The `@HttpConstraint` specifies that all other methods can be invoked by users in the role `R1`. The roles are mapped to security principals or groups in the container.

The security constraints can also be specified using the `<security-constraint>` element in *web.xml*. Within it, a `<web-resource-collection>` element is used to specify constraints on HTTP operations and web resources, `<auth-constraint>` is used to specify the roles permitted to access the resource, and `<user-data-constraint>` indicates how data between the client and server should be protected by the subelement `<transport-guarantee>`:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>

  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>

  <user-data-constraint>
    <transport-guarantee>INTEGRITY</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

This deployment descriptor requires that only the `GET` method at the `/account/*` URL is protected. This method can only be accessed by a user in the `manager` role with a requirement for content integrity. All HTTP methods other than `GET` are unprotected.

If HTTP methods are not enumerated within a `security-constraint`, the protections defined by the constraint apply to the complete set of HTTP (extension) methods:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
  </web-resource-collection>

  . . .
</security-constraint>
```

In this code, all HTTP methods at the `/account/*` URL are protected.

Servlet 3.1 defines *uncovered* HTTP protocol methods as the methods that are not listed in the `<security-constraint>` and if at least one `<http-method>` is listed in `<security-constraint>`:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  . . .
</security-constraint>
```

In this code fragment, only the HTTP `GET` method is protected and all other HTTP protocols methods such as `POST` and `PUT` are uncovered.

The `<http-method-omission>` element can be used to specify the list of HTTP methods not protected by the constraint:

```
<security-constraint>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method-omission>GET</http-method-omission>
  </web-resource-collection>

  . . .
</security-constraint>
```

In this code, only the HTTP `GET` method is not protected and all other HTTP protocol methods are protected.

The `<deny-uncovered-http-methods>` element, a new element in Servlet 3.1, can be used to deny an HTTP method request for an uncovered HTTP method. The denied request is returned with a 403 (`SC_FORBIDDEN`) status code:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
           http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
  <deny-uncovered-http-methods/>
  <web-resource-collection>
    <url-pattern>/account/*</url-pattern>
    <http-method>GET</http-method>
  </web-resource-collection>
  . . .
</web-app>
```

In this code, the `<deny-uncovered-http-methods>` element ensures that HTTP `GET` is called with the required security credentials, and all other HTTP methods are denied with a 403 status code.

`@RolesAllowed`, `@DenyAll`, `@PermitAll`, and `@TransportProtected` provide an alternative set of annotations to specify security roles on a particular resource or a method of the resource:

```
@RolesAllowed("R2")
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
  //. . .
}
```

If an annotation is specified on both the class and the method level, the one specified on the method overrides the one specified on the class.

Servlet 3.1 introduces two new predefined roles:

- * maps to any defined role.
- ** maps to any authenticated user independent of the role.

This allows you to specify security constraints at a higher level than a particular role.

At most, one of `@RolesAllowed`, `@DenyAll`, or `@PermitAll` may be specified on a target. The `@TransportProtected` annotation may occur in combination with either the `@RolesAllowed` or `@PermitAll` annotations.

The servlets can be configured for HTTP Basic, HTTP Digest, HTTPS Client, and form-based authentication:

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password" autocomplete="off">
  <input type="button" value="submit">
</form>
```

This code shows how form-based authentication can be achieved. The login form must contain fields for entering a username and a password. These fields must be named `j_username` and `j_password`, respectively. The action of the form is always `j_security_check`.

Servlet 3.1 requires `autocomplete="off"` on the password form field, further strengthening the security of servlet-based forms.

The `HttpServletRequest` also provides programmatic security with the `login`, `logout`, and `authenticate` methods.

The `login` method validates the provided username and password in the password validation realm (specific to a container) configured for the `ServletContext`. This ensures that the `getUserPrincipal`, `getRemoteUser`, and `getAuthType` methods return valid values. The `login` method can be used as a replacement for form-based login.

The `authenticate` method uses the container login mechanism configured for the `ServletContext` to authenticate the user making this request.

Resource Packaging

You can access resources bundled in the *.war* file using the `ServletContext.getResource` and `.getResourceAsStream` methods. The resource path is specified as a string with a leading "/." This path is resolved relative to the root of the context or relative to the *META-INF/resources* directory of the JAR files bundled in the *WEB-INF/lib* directory:

```
myApplication.war
  WEB-INF
```

```
    lib
      library.jar
```

library.jar has the following structure:

```
library.jar
  MyClass1.class
  MyClass2.class
  stylesheets
    common.css
  images
    header.png
    footer.png
```

Normally, if *stylesheets* and *image* directories need to be accessed in the servlet, you need to manually extract them in the root of the web application. Servlet 3.0 allows the library to package the resources in the *META-INF/resources* directory:

```
library.jar
  MyClass1.class
  MyClass2.class
  META-INF
    resources
      stylesheets
        common.css
      images
        header.png
        footer.png
```

In this case, the resources need not be extracted in the root of the application and can be accessed directly instead. This allows resources from third-party JARs bundled in *META-INF/resources* to be accessed directly instead of manually extracted.

The application always looks for resources in the root before scanning through the JARs bundled in the *WEB-INF/lib* directory. The order in which it scans JAR files in the *WEB-INF/lib* directory is undefined.

Error Mapping

An HTTP error code or an exception thrown by a serlvet can be mapped to a resource bundled with the application to customize the appearance of content when a servlet generates an error. This allows fine-grained mapping of errors from your web application to custom pages. These pages are defined via `<error-page>`:

```
<error-page>
  <error-code>404</error-code>
  <location>/error-404.jsp</location>
</error-page>
```

Adding the preceding fragment to *web.xml* will display the */error-404.jsp* page to a client attempting to access a nonexistent resource. You can easily implement this mapping for other HTTP status codes as well by adding other `<error-page>` elements.

The `<exception-type>` element is used to map an exception thrown by a servlet to a resource in the web application:

```
<error-page>
  <exception-type>org.example.MyException</exception-type>
  <location>/error.jsp</location>
</error-page>
```

Adding the preceding fragment to *web.xml* will display the */error.jsp* page to the client if the servlet throws the `org.example.MyException` exception. You can easily implement this mapping for other exceptions as well by adding other `<error-page>` elements.

The `<error-page>` declaration must be unique for each class name and HTTP status code.

Handling Multipart Requests

`@MultipartConfig` may be specified on a servlet, indicating that it expects a request of type `multipart/form-data`. The `HttpServletRequest.getParts` and `.getPart` methods then make the various parts of the multipart request available:

```
@WebServlet(urlPatterns = {"/FileUploadServlet"})
@MultipartConfig(location="/tmp")
public class FileUploadServlet extends HttpServlet {

  @Override
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
    for (Part part : request.getParts()) {
      part.write("myFile");
    }
  }
}
```

In this code:

- `@MultipartConfig` is specified on the class, indicating that the `doPost` method will receive a request of type `multipart/form-data`.
- The `location` attribute is used to specify the directory location where the files are stored.
- The `getParts` method provides a `Collection` of parts for this multipart request.
- `part.write` is used to write this uploaded part to disk.

Servlet 3.1 adds a new method, `Part.getSubmittedFileName`, to get the filename specified by the client.

This servlet can be invoked from a JSP page:

```
<form action="FileUploadServlet"
      enctype="multipart/form-data"
      method="POST">
    <input type="file" name="myFile"><br>
    <input type="Submit" value="Upload File"><br>
</form>
```

In this code, the form is `POST`ed to `FileUploadServlet` with encoding `multipart/form-data`.

Upgrade Processing

Section 14.42 of HTTP 1.1 (RFC 2616) defines an upgrade mechanism that allows you to transition from HTTP 1.1 to some other, incompatible protocol. The capabilities and nature of the application-layer communication after the protocol change are entirely dependent upon the new protocol chosen. After an upgrade is negotiated between the client and the server, the subsequent requests use the newly chosen protocol for message exchanges. A typical example is how the WebSocket protocol is upgraded from HTTP, as described in the Opening Handshake section of RFC 6455.

The servlet container provides an HTTP upgrade mechanism. However, the servlet container itself does not have any knowledge about the upgraded protocol. The protocol processing is encapsulated in the `HttpUpgradeHandler`. Data reading or writing between the servlet container and the `HttpUpgradeHandler` is in byte streams.

The decision to upgrade is made in the `Servlet.service` method. Upgrading is achieved by adding a new method, `HttpServletRequest.upgrade`, and two new interfaces, `javax.servlet.http.HttpUpgradeHandler` and `javax.servlet.http.WebConnection`:

```
if (request.getHeader("Upgrade").equals("echo")) {
    response.setStatus(HttpServletResponse.SC_SWITCHING_PROTOCOLS);
    response.setHeader("Connection", "Upgrade");
    response.setHeader("Upgrade", "echo");
    request.upgrade(MyProtocolHandler.class);
    System.out.println("Request upgraded to MyProtocolHandler");
}
```

The request looks for the `Upgrade` header and makes a decision based upon its value. In this case, the connection is upgraded if the `Upgrade` header is equal to `echo`. The correct response status and headers are set. The `upgrade` method is called on `HttpServletRequest` by passing an instance of `HttpUpgradeHandler`.

After exiting the `service` method of the servlet, the servlet container completes the processing of all filters and marks the connection to be handled by the instance of `HttpUpgradeHandler`:

```
public class MyProtocolHandler implements HttpUpgradeHandler {
  @Override
  public void init(WebConnection wc) {
    //. . .
  }

  @Override
  public void destroy() {
    //. . .
  }
}
```

This code shows an implementation of `HttpUpgradeHandler`. The servlet container calls the `HttpUpgradeHandler`'s `init` method, passing a `WebConnection` to allow the protocol handler access to the data streams. When the upgrade processing is done, `HttpUpgradeHandler.destroy` is invoked.

The servlet filters only process the initial HTTP request and response. They are not involved in subsequent communications.

A Lightweight Introduction to the Spring Framework

Just Spring

O'REILLY®

Madhusudhan Konda

Just Spring

Madhusudhan Konda

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

CHAPTER 2

Fundamentals

We saw the bare-minimum basics of Spring Framework in the last chapter. We worked with new things such as beans, bean factories, and containers. This chapter explains them in detail. It discusses writing beans, naming conventions, how they are wired into containers, and so on.

Configuring Beans

For Spring, all objects are beans! The fundamental step in the Spring Framework is to define your objects as beans. Beans are nothing but object instances that would be created and manage by the Spring Framework by looking at their class definitions. These definitions basically form the configuration metadata. The framework then creates a plan for which objects need to be instantiated, which dependencies need to be set and injected, and the scope of the newly created instance, etc., is based on this configuration metadata.

The metadata can be supplied in a simple XML file, as we saw in Chapter 1. Alternatively, one could provide the metadata as annotation or Java Configuration.

We first discover the definitions of the Spring beans by using a config file which is discussed in the next section.

Using XML

We saw earlier that the Framework reads the Java classes defined in the XML config and initializes and loads them as Spring beans into a runtime container. The container is a runtime bucket of all the fully prepared instances of Java classes. Let's take a look at an example of how this process is executed.

We will define a bean with a name `person` that corresponds to a class `Person.` The `Person` has three properties, two of which (`firstName` and `lastName`) were set via the

constructor, while the third one (age) is set by using a setter. There is also another property called address. However, this property is not an simple Java type, but instead points (references) to another class Address.

The following snippet shows the Person bean class's definition:

```
public class Person {
  private int age = 0;
  private String firstName = null;
  private String lastName = null;
  private Address address = null;

  public Person(String fName, String lName){
    firstName = fName;
    lastName = lName;
  }
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }
  public Address getAddress() {
    return address;
  }
  public void setAddress(Address address) {
    this.address = address;
  }
  public String getDetails(){
    return firstName +" "+lastName
      +" is "+getAge()+" old and lives at "+getAddress();
  }
}
```

As you can see, only age and address variables have setters—meaning they are set in a different way than the variables set via constructor. For completeness, the Address class definition is:

```
public class Address {
  private int doorNumber = 0;
  private String firstLine = null;
  private String secondLine = null;
  private String zipCode = null;
  // getters and setters for these variables go here
  ....
}
```

The ultimate goal is to create the Person and Address beans via our configuration files. The classes are declared in a Spring configuration XML file as shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="person" class="com.madhusudhan.jscore.beans.Person">
      <constructor-arg value="Madhusudhan" />
      <constructor-arg value="Konda" />
      <property name="age" value="99"/>
      <property name="address" ref="address"/>
    </bean>

    <bean name="address" class="com.madhusudhan.jscore.beans.Address">
      <property name="doorNumber" value="99"/>
      <property name="firstLine" value="Rainbow Vistas"/>
      <property name="secondLine" value="Kukatpally, Hyderabad"/>
      <property name="zipCode" value="101010"/>
    </bean>
</beans>
```

There are few things that we should take a note of here.

The topmost node declares `<beans>` as your root element. All bean definitions would then follow using a `<bean>` tag. Usually, the XML file consists of at least one bean. Each bean definition may contain sets of information, most importantly the name and the class tags. It may also have other information, such as the id, the scope of the bean, the dependencies, and others.

Basically, when the config file is loaded at runtime, the framework will pick up these definitions and create the instance of `Person`. It then gives a name as `person`. This name is then used to query for the same instance from the container by using a Framework's API.

For example, the following code snippet illustrates how a `PersonClient` loads up its container and queries the beans:

```
public class PersonClient {
  private static ApplicationContext context = null;
  public PersonClient() {
    context = new ClassPathXmlApplicationContext("ch2-spring-beans.xml");
  }
  public String getPersonDetails() {
    Person person =
     (Person) context.getBean("person");
     return person.getDetails();
  }
}
```

From the above snippet, two steps are significant.

- The `ApplicationContext` is instantiated by an appropriate beans config file. This context is nothing but our bucket of beans—a container in Spring's terminology.

- Querying the container for our newly created `Person` bean. We use the framework's Context API to search the Java instance, using the `getBean()` method. The string value that you pass to this `getBean()` query API method is the name that you've given the bean in your XML file—`person` in this case.

You can split the bean definitions across multiple files.

For example, you create all the beans that deliver the business functions in a file called *business-beans.xml*, the utility beans in *util-beans.xml*, data access beans in *dao-beans.xml*, and so on. We will see how to instantiate the Spring container by using multiple files later in the chapter.

Generally, I follow the convention of creating the files by using two parts separated by a hyphen. The first part usually represents the business function, while the second part simply indicates that these are spring beans. There is no restriction on the naming convention, so feel free to name your beans whatever you like.

Each bean should either have a name or id field attached to it. You can create the beans with neither of these things, making them anonymous beans (which are not available to query in your client code). The name and id fields both serve the same purpose, except that the id field corresponds to XML specification's id notation. This means that checks are imposed on the id (for example, no special characters in the id value). The name field does not attract any of these restrictions.

The class field declares the fully qualified name of the class. If the instantiation of the class requires any initialization data, it is set via properties or a constructor argument.

For example, in the above XML file, the `Person` object is instantiated with both: a constructor argument and property setters. The firstName and lastName were set using the `<constructor-arg value="..."/>` tag, while the rest of the properties were set using a simple property tag: `<property name="age" value=".."/>`.

The value fields can be simple values or references to other beans. A `ref` tag is used if a the bean needs another bean, as is seen for address: `<property name="address" ref="address"/>` . Note the use of `ref` keyword rather than `value` keyword when another bean is referenced.

You can name the bean as you wish. However, I would suggest sticking to a camelCase class name, with the first letter lowercase.

Using Annotations

One possible way of Spring wiring is using annotations. When you choose to go along the path of annotations to define your beans and wire them, you are effectively reducing your XML meta data configurations.

Let's see how we can define beans using annotations. The ReservationManager has one dependency—a ReservationService. The sevice property must be injected into our manager to do its flight reservation work.

The dummy service definition is shown here:

```
public class ReservationService {
    public void doReserve(ReservationMessage msg){
        //
    }
}
```

The ResevationManager has a dependency—it needs the reservation service so it process the flight reservations. In order to fulfil this dependency, we need to inject the service into the manager. Unlike the config route, we will inject this dependency by using an annotation called `@Autowired` annotation. See how we decorated the variable reservationService, using this annotation shown in the following snippet:

```
public class ReservationManager {
  @Autowired
  private ReservationService reservationService = null;

  public void process(Reservation r) {
    reservationService.reserve(r);
  }
}
```

When a variable, method or constructor is annotated with @Autowired, framework will find the relevant dependency and inject that dependency automatically. In the preceding case, the `ReservationManager` is looking for a `ReservationService` instance. Behind the scenes, the framework will wire the bean by finding it `byType` (check the autowiring section for more details).

One last thing we need to do is let framework know we are going to use annotations. The way we do this is to declare a special tag in the XML file:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config/>
  <!-- our reservation service -->
  <bean name="resSvcABC"
    class="com.madhusudhan.jscore.fundamentals.annotations.ReservationService"/>
  <bean name="reservationManager"
    class="com.madhusudhan.jscore.fundamentals.annotations.ReservationManager"/>
</beans>
```

The `annotation-config` tag lets the framework know we are following the annotations route. This way, it starts searching for the classes that may have the appropriate annotations (in our case, we have `ReservationManager` with `@Autowired` annotation) and do fulfil any obligations.

Note that we need to import appropriate `context` schemas as shown in bold in the above snippet.

When ReservationManger gets created, the framework looks for a ReservationService type bean (we have created one in the bean config) and injects it. I have deliberately set the name of the service bean to `resSvsABC` so you would know that framework uses `byType` rather than `byName` when picking up the dependencies.

When the `reservationManager` gets created, it will always be injected with the service bean!

There are couple of things that we should look at.

We did not declare any properties on the `ReservationManager`, for example, `<property name="reservationService" ref="resSvs"/>`, as we would do normally. There are no setters and getters on the `ReservationManager` too. This is all redundant because the job is cleanly done by the annotation configuration!

Before we close this section, there's another attribute that we should see—the `component-scan` attribute on the `context` namespace.

In the previous snippet, although we have eliminated much of XML, we still had to define the service bean in the XML. Is there a way we could avoid this and condense our XML further?

Yes, we should use the `component-scan` attribute that gobbles away much of our XML. As the tag may indicate, `component-scan` basicaly scans a particular drectory or directories to find out special annotated classes.

Revisiting the `ReservationService`, annotate the class by using the `@Component` annotation:

```
@Component
public class ReservationService {
  ..
}
```

Any class annotated with this `@Component` annotation will be picked up by the framework (as long as the class is the part of the `component-scan`'s base-package value) and gets instantiated. The next thing we need to do is to declare the `component-scan` as shown here:

```
<context:component-scan
  base-package="com.madhusudhan.jscore.fundamentals.annotations"/>
```

By declaring this tag, what we are telling the framework is to search all the annotated (with `@Component`) classes in the package indicated by `base-package` attribute. Hence `ReservationService` will be searched for and instantiated by the framework and gets injected into the `ReservationManager` bean because of `@Autowired` annotation.

If you do not wish to tie up to Spring's `@Autowired` annotation usage, perhaps think about using JSR-330's `@Inject` annotation. Spring supports this annotation too. Although we can't discuss `@Inject` annotation here, it is almost replica of `@Autowired` annotation.

There's a lot of debate on using annotations against XML configurations. Although annotations do not clutter our XML files, they are tied to the source. There are various schools of thought, some encouraging while others are discouraging the use of annotations. Personally, I like annotations and their clean and neat approach to solving configuration issues. However, I prefer to use meta-data for the simple reason that I can change the configuration without having to recompile/rebuild the application.

XML Namespaces

Sometimes the configuration files seems to have lot of bean information. This is an eyesore to readers at times. However, the good news is that they can be condensed by using schema-based XML configuration. We have been using XML Schemas all along in as the following meta data snippet illustrates:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <!-- declare your beans here -->
  <bean>
                ...
  </bean>
</beans>
```

As you can see, the `beans` schema has been used in the above configuration. Spring defines various schemas such as `jms`, `aop`, `jee`, `tx`, `lang`, `util`, and others.

Adding the appropriate schemas is quite straightforward. For example, if you are going to use a jms schema, add the following lines (in bold) to the configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
```

```
        http://www.springframework.org/schema/jms
        http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">
        ....
</beans>
```

A xmlns tag defines a jms namespace while the last two line indicate the location of the schema definitions. So, replace the xxx shown in the following pattern with a specific schema you wish to import:

```
<!-- Replace xxx with one of the many schemas such as jms, aop, tx etc -->
http://www.springframework.org/schema/xxx
http://www.springframework.org/schema/xxx/spring-xxx-3.0.xsd
```

Once you have the appropriate schema and the associated namespace (using the `xmlns` tag), using the schema-based configuration is a breeze.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/jms
       http://www.springframework.org/schema/jms/spring-jms-3.0.xsd">

       <jms:listener-container>
         ...
       </jms:listener-contaner>
</beans>
```

See how the `listener-container` bean is added to the container by using the jms namespace.

The good old DTD-style configuration is still valid too:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  ....
</beans>
```

However, I strongly recommend using the schema-based configuration unless you have a valid reason to stick to older DTD style.

Creating Beans

The beans are instances wired together to achieve an application's goal.

Usually in a standard Java application, we follow a specific life cycle of the components, including their dependencies and associations.

For example, when you start a main class, it automatically creates all the dependencies, sets the properties, and instantiates the dependent instances for your application to progress.

The responsibility of creating the dependency and associating this dependency to the appropriate instance is given to your main class in our standalone application.

However, in Spring, this responsibility is taken away from us and given to the Spring Framework. The instances (a.k.a. beans) are created, the associations are established, and dependencies are injected by the Spring Framework entirely. You and I have no say except in defining and loading them.

So, creating the whole object graph is the responsibility of the framework!

These beans that exists in a container are then queried by the application and act upon them. Of course, you would have to declare these associations and other configuration metadata either in an XML file or provide them as annotations for the Framework to understand what it should do.

One important characteristic of the Framework while creating beans is to follow a *fail-fast* approach.

When the Framework encounters any issues in loading a bean, it just quits—no container with half-baked beans ever gets created. This is a really good characteristic of a framework as it would be forced to catch all errors during compile time rathen than at runtime.

Life Cycle

The Spring Framework does quite a few things behind the scenes.

The life cycle of a bean is easy to understand, yet different from the life cycle exposed in a standard Java application. In a normal Java process, a bean is usually instantiated using a *new* operator. The Framework executes more actions in addition to simply creating the beans. Once they are created, they are loaded into the appropriate container (we will learn about containers in Chapter 3) for a client to access them.

The usual life-cycle steps are listed here:

- The framework factory loads a bean definitions and creates it.
- The bean is then populated with the properties as declared in the bean definitions. If the property is a reference to another bean, the other bean will be created and populated, and the reference is injected into this bean.

- If our bean implements any of the Spring's interfaces, such as `BeanNameAware` or `BeanFactoryAware`, appropriate callback methods will be invoked.
- The framework also invokes any `BeanPostProcessor`'s associated with your bean for pre-initialzation.
- The `init-method`, if specified, is invoked on the bean.
- The post-initialization will be performed if specified on the bean.

Do not get stressed if the things mentioned here don't get digested yet—we will discuss these points in the coming sections in detail.

When comes to creating the beans, beans which have no dependencies will be created normally. Whereas, the beans which has dependencies (that is, some of its properties refer to other beans) will be created only after satisfying the dependencies they have.

We will discuss this process in the next section, using some examples.

Note that we can also create beans by using static methods or Factories. We will look at them in the next chapter in detail.

Instantiating Beans Without Dependencies

Do you remember the `FileReader` class we defined in our earlier chapter? Here's the snippet of the class if you can't recall:

```
public class FileReader implements IReader{
  private StringBuilder builder = null;
  private Scanner scanner = null;

  // constructor
  public FileReader(String fileName) { ... }

  // read method implementation
  public String read() { ... }
}
```

The `FileReader`'s constructor takes in a `fileName` as its argument in order to get ready for the action. This bean is defined by passing an argument to the constructor, using meta-data. Look at the meta data of the bean:

```
<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FileReader">
  <constructor-arg value="src/main/resources/basics/basics-trades-data.txt" />
</bean>
```

The required `fileName` variable is set with a value via the `constructor-arg` tag, providing a value of `src/main/resources/basics/basics-trades-data.txt` as an argument.

When the Framework reads the definition of this class, it creates an instance by using the *new* operator (in reality, the bean is instantiated by using Java Reflection). As this bean has no dependencies, it will now be instantiated and ready to use.

Instantiating Beans With Dependencies

There is a second case where beans depend on other beans. For example, we have seen a `Person` having an `Address`. Unless the `Address` object is created, creating the Person can't be succesful. So, the Person bean is *dependent* on the Address bean. However, if the `FileReader` bean has a dependency on another bean, the other bean will be created and instantiated. See the following snippet.

See the definition of `Person` having an `address` property which refers to another bean named `address`? The Person object will be injected with an Address object to satisfy the dependency. The bean dependencies can be one or more beans.

```
<bean name="person"
  class="com.madhusudhan.jscore.beans.Person">
  ....
  <property name="address" ref="address"/>
</bean>

<bean name="address"
  class="com.madhusudhan.jscore.beans.Address">
  ....
</bean>
```

The order of creation is important to Spring. After digesting the configuration metadata, Spring creates a plan (it allocates certain priorities to each bean) with the order of beans that needs to be created to satisfy dependencies. Hence, the `Address` object is created first, before the `Person`. If Spring encounters any exception while creating the `Address` object, it will fail fast and quit the program. It does not create any further beans and lets the developer know why it won't progress further.

Aliasing Beans

Sometimes, we may need to give the same bean a different name—usually as alias. For example, an Address bean can be called as shipping address or a billing address. Aliasing is a way of naming the same bean with different names. We use the `alias` tag to give a name to a predefined bean.

```
<bean name="address"
  class="com.madhusudhan.jscore.fundamentals.Address">
  ...
</bean>
```

```
<alias name="address" alias="billingAddress"/>
<alias name="address" alias="shippingAddress"/>
```

In the above snippet, we have declared an `Address` bean with `address` as it's name. Two aliases, `billingAddress` and `shippingAddress` were declared, both pointing to the same bean. We can use either of the aliases in our application as if they were original beans.

Anonymous Beans

We can also create beans whose existence is associated to the referencing bean only. These types of beans are called Anonymous or Inner beans. They are *nameless* and hence not available for our programs to query them.

```
<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FileReader">
  <constructor-arg value="src/main/resources/basics/basics-trades-data.txt" />
  <property name="platformLineEnder"
    <bean class="com.madhusudhan.jscore.basics.readers.WindowsLineEnder"/>
  </property>
</bean>
```

The `platformLineEnder` property refers to a `WindowsLineEnder` bean. Because the `platformLineEnder` bean has been defined as a property (no `ref` tag is defined), it is not available to any other beans in the context except the `FileReader` bean.

Injection Types

Spring allows us to inject the properties via constructors or setters. While both types are equally valid and simple, it's a matter of personal preference in choosing one over the other.

One advantage to using constructor types over setters is that we do not have to write additional setter code. Having said that, it is not ideal to create constructors with lots of properties as arguments. I detest writing a class with a constructor that has more than a couple of arguments!

Constructor Type Injection

In the previous examples, we have seen how to inject the properties via constructors by using the `constructor-arg` attribute. Those snippets illustrate the constructor injection method. The basic idea is that the class will have a constructor that takes the arguments, and these arguments are wired via the config file.

Here is an `FtpReader` code snippet that has a constructor taking two arguments:

```
public class FtpReader implements IReader {
  private String ftpHost = null;
```

```
  private int ftpPort = 0;

  public FtpReader(String ftpHost, int ftpPort) {
    this.ftpHost = ftpHost;
    this.ftpPort = ftpPort;
  }

  @Override
  public String read() {
    // your impl goes here
    return null;
  }
}
```

The `ftpHost` and `ftpPort` arguments are then wired using `constructor-arg` attributes defined in the XML config file:

```
<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg value="madhusudhan.com" />
  <constructor-arg value="10009" />
</bean>
```

You can set references to other beans, too via the constructor arguments. For example, the following snippet injects a reference `FileReader` into the `ReaderService` constructor:

```
<bean name="readerService"
    class="com.madhusudhan.jscore.basics.service.ReaderService">
  <constructor-arg ref="reader" />
</bean>

<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FileReader">
  <constructor-arg value="src/main/resources/basics/basics-trades-data.txt" />
</bean>
```

This is how the `ReaderService` will look with a constructor accepting an `IReader` type:

```
public class ReaderService {
  private IReader reader = null;

  public ReaderService(IReader reader) {
    this.reader = reader;
  }
  ...
}
```

Argument type resolution

One quick note about constructor type injection—there are couple of rules that framework will follow when resolving the types of the arguments. In the preceding `FtpReader` example, the first argument was `ftpHost` followed by `ftpPort`. The case is straight-

forward—the constructor expects a string and an integer, so the framework picks the first argument as String type and the second one as Integer type.

Although you declare them as string values in your config file (such as `value=".."`), the `java.beans.PropertyEditor`'s are used by the framework to convert the string value to the appropriate property type.

Ideally, the declaration should define the types too as shown in the following snippet:

```
<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg type="String" value="madhusudhan.com" />
  <constructor-arg type="int" value="10009" />
</bean>
```

The types are normal Java types—such as `int`, `boolean`, `double`, `String`, and so on.

You could also set index's on the values, starting the index from zero as shown here:

```
<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FtpReader">
  <constructor-arg index="0" type="String" value="madhusudhan.com" />
  <constructor-arg index="1" type="int" value="10009" />
</bean>
```

Setter Type Injection

In addition to injecting the dependent beans and properties via constructors, Spring also allows them to be injected via setters, too. In order to use the setter injection, we have to provide setters on the respective variables. If the property exhibits read and write characteristics, provide both a setter and a getter on the variable.

So, in our `ReaderService` class, create a variable of `IReader` type and a matching setter/getter for that property. The constructor is left empty as the properties are now populated using the setters. You should follow the normal bean conventions when creating setters and getters.

Modified `ReaderService` is given here:

```
public class ReaderService {
  private IReader reader = null;

  // Setter and getter
  public void setReader(IReader reader) {
    this.reader = reader;
  }

  public IReader getReader() {
    return reader;
  }
  ...
}
```

The significant points are the the setter and getter methods on the `IReader` variable and the omission of the constructor altogether. The configuration of the class in our XML file looks like this:

```
<bean name="readerService"
    class="com.madhusudhan.jscore.basics.service.ReaderService">
  <!-- Setter type injection -->
  <property name="reader" ref="reader"/>
</bean>

<bean name="reader" class="com.madhusudhan.jscore.basics.readers.FileReader">
  ...
</bean>
```

The notable change is to create a property called `reader` and set it with a reference to `the FileReader` class. The framework will check the `ReaderService` for a `reader` property and invoke `setReader` by passing the `FileReader` instance.

Mixing Constructor and Setter

You can mix and match the injection types as you like, too. The revised `FileReader` class listed here has a constructor as well as few other properties. The `componentName` is set using setter, while the `fileName` is injected via constuctor.

```
<bean name="reader"
    class="com.madhusudhan.jscore.fundamentals.injection.FileReader">
  <constructor-arg value="src/main/resources/basics/basics-trades-data.txt" />
  <property name="componentName" value="TradeFileReader"/>
</bean>
```

Although mixing and matching the injection types is absolutely possible, I recommend sticking with one or the other of them, rather than both, to avoid complicating matters.

Bean Callbacks

Spring Framework provides a couple of hooks to our beans in the form of callback methods. These methods provide opportunity for the bean to initialize properties or clean up resources. There are two such method hooks: `init-method` and `destroy-method`.

init-method

When a bean is being created, we can let Spring invoke a specific method on our bean to initialize. This method provides a chance for the bean to do housekeeping stuff and any initialization, such as creating data structures, creating thread pools, and so on.

Say we have a requirement of creating a class for fetching Foreign Exchange (FX) rates. The FxRateProvider is a class that provides us with these rates when queried (mind you, it's a dummy implementation of FX Rates!).

See the code snippet here:

```
public class FxRateProvider {
  private double rate = 0.0;
  private String baseCurrency = "USD";
  private Map<String, Double> currencies = null;

  /* Invoked via Spring's init-method callback */
  public void initMe(){
    currencies = new HashMap<String, Double>();
    currencies.put("GBP",1.5);
    currencies.put("USD",1.0);
    currencies.put("JPY", 1000.0);
    currencies.put("EUR",1.4);
    currencies.put("INR",50.00);
  }

  public double getRate(String currency){
   if(!currencies.containsKey(currency))
     return 0;
   return currencies.get(currency);
  }
}
```

A noticeable point is the `initMe` method. It is a normal method invoked by the Framework during the process of its creation.

The associated configuration of the bean is provided in the following XML:

```
<bean name="fxRateProvider"
  class="com.madhusudhan.jscore.fundamentals.callbacks.FxRateProvider"
  init-method="initMe">
  <property name="baseCurrency" value="USD"/>
</bean>
```

destroy-method

Similar to the initialization, framework provides a destroy method to clean up before destroying the bean—named as `destroy-method`. The FxRateProvider shown here has a `destroy` method:

```
public class FxRateProvider {

  public void destroyMe() {
    // do your cleanup operations here
    currencies = null;
  }
```

```
...
}
```

We should refer the `destroyMe` method in the XML declaration like this:

```
<bean name="fxRateProvider"
  class="com.madhusudhan.jscore.fundamentals.callbacks.FxRateProvider"
  init-method="initMe"
  destroy-method="destroyMe">

  <property name="baseCurrency" value="USD"/>
</bean>
```

When the program quits, the framework destroys the beans. During the process of destruction, when the config metadata declares `destroyMe` as the destroy method, the `destroyMe` method is invoked. This gives the bean a chance to do some housekeeping activities if we wish.

Ideally, this method should be coded to free up resources, nullify objects, and other cleanup operations.

Common Callbacks

Let's say we religiously code a `init` and `destroy` methods on all our beans. Does it mean we have to declare the `init-method` and `destroy-method` explicitly on each and every bean? Well, not exactly.

As long as we define the same method names across all the beans, we can use Framework's facility of declaring default callbacks—`default-init-method` and `default-destroy-method`.

Instead of declaring the individual methods on each of the bean, we need to declare these default callbacks at the topmost `beans` element. See how they have been declared in the following XML snippet:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-init-method="initMe"
       default-destroy-method="destroyMe" >
  <bean>
  ...
  </bean>
</beans>
```

Note that these default methods are associated to the `beans` element rather than a `bean` element.

From the above example, the `initMe` and `destroyMe` methods will be invoked automaticaly on all the beans. If any of the beans don't have these methods, Framework ignores them and no action is performed.

Summary

This chapter discussed the Spring Framework in detail. It explained the concept of beans and bean factories. We have also learned about the life cycle of the bean scopes and touched upon the property editors used in injecting Java Collections and other types of objects.

We discuss the containers and application contexts in the next chapter, which forms a crucial concept in putting the framework to work.

Vital Information for Tomcat Programmers & Administrators

2nd Edition
Tomcat 6.0

Tomcat

The Definitive Guide

Jason Brittain
with Ian F. Darwin

O'REILLY®

SECOND EDITION

Tomcat

The Definitive Guide

Jason Brittain with Ian F. Darwin

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1

Getting Started with Tomcat

Because Tomcat is written in Java, some people assume that you have to be a Java guru to use it. That is not so! Although you need to know Java to modify the internals of Tomcat or to write your own servlet programs, you do not need to know any Java to use Tomcat or to write or maintain many JavaServer Pages (JSPs). You can have JSPs that use "JavaBeans" or "JSP Custom Tags"; in both cases, you are simply using Java components that a developer has set up for you.

In this chapter, we explain how to install Tomcat, get it running, and test it to make sure that it's functioning properly.

As of this writing, there are several production-ready versions of Tomcat available, but we strongly suggest you use the latest *stable* version of the 6.0 branch or whichever is the latest stable version of Tomcat by the time you read this. See the Apache Tomcat home page (*http://tomcat.apache.org*) to find the latest stable version. For Tomcat versions 5.5 and 6.0, this book provides an abundance of answers and explanations about the general concepts of how Tomcat works, in addition to showing rich detail about how to use these popular versions of Tomcat.

Installing Tomcat

There are several paths to getting Tomcat up and running. The quickest one is to download and run the compiled binary. Tomcat is written in Java, which means you need to have a modern and complete Java runtime installed before you can build or test it. Read Appendix A to make sure you have Java installed properly. *Do not skip this step; it is more important than it sounds!*

One of the benefits of open source projects is that programmers find and fix bugs and make improvements to the software. If you're not a programmer, there is little or nothing to be gained from recompiling Tomcat from its source code, as you are not interested in this level of interaction. Also, if you're not an experienced Tomcat

developer, attempting to build and use your own Tomcat binaries may actually cause problems because it is relatively easy to build Tomcat in ways that quietly disable important features. To get started quickly, you should download an official release binary package for your system.

If you want some hints on compiling from source, see Chapter 9.

There are two levels of packaging. The Apache Software Foundation publishes binaries in the form of releases and nightly builds. Other organizations rebundle these into RPM packages and other kinds of installers for Linux, "packages" for BSD, and so forth. The best way to install Tomcat depends on your system. We explain the process on several systems: Linux, Solaris, Windows, Mac OS X, and FreeBSD.

Tomcat 6 requires any Java runtime version 1.5 or higher (which Sun's marketing group calls "Java 5"). We suggest that you run Tomcat 6 on Java 1.6 or higher, however, due to the additional features, fixes, and performance improvements that Java 1.6 (or higher) JVMs offer.

Installing Tomcat on Linux

Tomcat is available in at least two different binary release forms for Linux users to choose from:

Multiplatform binary releases
: You can download, install, and run any of the binary releases of Tomcat from Apache's web site regardless of the Linux distribution you run. This format comes in the form of gzipped tar archives (*tar.gz* files) and zip archive files. This allows you to install Tomcat into any directory you choose, and you can install it as any user ID in the system. However, this kind of installation is not tracked by any package manager and will be more difficult to upgrade or uninstall later. Also, it does not come with an `init` script for integration into the system's startup and shutdown.

Distribution native package
: If you run Fedora or Red Hat Linux (or another Linux that uses the Red Hat package manager, such as SUSE or Mandriva), you can download a binary RPM package of Tomcat. This allows for easy uninstalls and upgrades via the Red Hat Package Manager, plus it installs a Tomcat `init` script for stopping, starting, and restarting Tomcat from the command line and on reboots. The downside to this method of installation is that you must install the Tomcat RPM package as the root user. As of this writing there are at least two RPM package implementations for you to choose from, each with different features.

Keep in mind, though, that Linux is just the operating system kernel, and the complete operating system is a "distribution." Today, there are many different Linux distributions. Some examples include Fedora, Red Hat, Ubuntu, Mandriva, Gentoo, and Debian. Although any two Linux distributions tend to be similar, there are also usually enough differences that make it difficult for developers to write one script that runs successfully on two. Also, each Linux distribution may primarily use a different native package manager, so each version of a distribution can change any number of things in the operating system, including Java* and Tomcat. It is not uncommon for Linux distributions to bundle software written in Java that does not work only because the distribution's own package of it is broken in a subtle way. Distributions also tend to include old versions of Tomcat that are either unstable or less than ideal to run your web site compared to the latest stable version available. For these reasons, it's almost always best to install your own recent stable version of Tomcat.

Because there are so many Linux distributions, and because they are significantly different from each other, giving specific instructions on how best to install Tomcat on each version of each Linux distribution is beyond the scope of this book. Luckily, there is enough similarity between the popular Linux distributions for you to follow more generic Linux installation instructions for installing Tomcat from an Apache binary release archive.

If you run a Fedora or Red Hat Linux distribution, more than one implementation of Tomcat RPM packages exists for you to choose from:

The Tomcat RPM package that comes with this book
: This is a fully relocateable RPM package that can be easily rebuilt via a custom *ant build* file. It does not build Tomcat itself but instead bundles the official multiplatform Apache release class binaries of the Tomcat 6 version of your choice. This RPM package depends on no other RPM packages, so it can be installed as a single package, but needs to be configured to use an installed Java runtime (JDK or JRE). See Appendix E for the full source listing of the RPM package's scripts.

The Tomcat RPM package that is available from JPackage.org
: This is a nonrelocateable RPM package that installs Tomcat into the */var* directory. It rebuilds Tomcat from source code and then packages up the resulting multiplatform class binaries. This RPM package depends on many other RPM packages (each potentially requiring yet more packages) from JPackage.org and must be installed as a graph of RPM packages. As of this writing, JPackage.org does not have a Tomcat 6.0 RPM, only a Tomcat 5.5 RPM.

* See Appendix A for more information about how to work around a distribution's incompatible Java implementation.

Each of these RPM packages includes detailed scripts for installing, uninstalling, and upgrading Tomcat, as well as scripts for runtime integration with the operating system. We suggest you try ours first.

If you run Gentoo Linux, there is an ebuild of Tomcat 6 that you can install and use. See the guide for it by William L. Thomson Jr. at *http://www.gentoo.org/proj/en/java/tomcat6-guide.xml*. Also, see the Tomcat Gentoo ebuild page on the Gentoo Wiki at *http://gentoo-wiki.com/Tomcat_Gentoo_ebuild*. In addition to the ebuild, the RPM package from this book is written to install and run on Gentoo; just install the *rpm* command first.

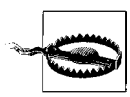
Installing Tomcat from an Apache multiplatform binary release

For security reasons, you should probably create a `tomcat` user with low privileges and run Tomcat as that user. We suggest setting that user's login shell to */sbin/nologin* and locking the user's password so that it can't be guessed. Also, it's probably a good idea to make the `tomcat` user's primary group the `nobody` group or another group with similarly low permissions. You will need to do this as the root user:

```
# useradd -g 46 -s /sbin/nologin -d /opt/tomcat/temp tomcat
```

If you do not have root access, you could run Tomcat as your login user, but beware that any security vulnerabilities (which are extremely rare) in Tomcat could be exploited remotely as your user account.

Now download a release archive from the Apache binary release page at *http://tomcat.apache.org/download-60.cgi*. You should download the latest stable version as listed on the Tomcat home page at *http://tomcat.apache.org*.



Even if you intend to install only a subset of the archive files of the Tomcat version you chose, you should download *all* of the archive files for that version in case you need them later. The Apache Software Foundation does archive releases of Tomcat, but you should store your own copies as well. If you are a heavy user of Tomcat, you should probably also download archives of the source code for your release and store your own copies of them as well so that you may investigate any potential bugs you may encounter in the version you've chosen.

Uncompress the main Tomcat binary release archive. If you downloaded the *apache-tomcat-6.0.14.tar.gz* archive, for example, uncompress it wherever you want Tomcat's files to reside:

```
$ cd $HOME
$ tar zxvf apache-tomcat-6.0.14.tar.gz
```

Before you go any further, you should briefly look at the *RELEASE-NOTES* text file that resides in the root of your new Tomcat installation. It contains important information for everyone installing Tomcat and can give you details specific to the version you downloaded. Something else that is very important for you to do before proceeding with the installation is to read the online Tomcat changelog for your branch of Tomcat. For example, Tomcat 6.0's online changelog is at *http://tomcat.apache.org/tomcat-6.0-doc/changelog.html*. Regardless of the version of Tomcat you install and use, you should look at the bugs listed in the changelog because bugs that exist in your version are fixed in *newer* versions of Tomcat and will show up in the changelog listed under newer versions.

Although Java 1.5.x runtimes work fine with Tomcat 6, it is suggested that you use Java 1.6.x.

If you'll be running Tomcat as user `tomcat` (or any user other than the one you log in as), you must install the files so that this user may read and write those files. After you have unpacked the archives, you must set the file permissions on the Tomcat files so that the `tomcat` user has read/write permissions. To do that for a different user account, you'll need root (superuser) access again. Here's one way to do that from the shell:

```
# chown -R tomcat apache-tomcat-6.0.14
```

Tomcat should now be ready to run, although it will not restart on reboots. To learn how to make it run when your server computer boots up, see "Automatic Startup," later in this chapter.

Installing Tomcat from this book's Linux RPM packages

This book contains a production quality example of a Tomcat RPM package for Linux (see Appendix E for the source). It serves as both an elegant way to get Tomcat installed and running on Linux and as an example of how you may build your own custom Tomcat RPM package.

Before you begin, you must install Apache Ant (*http://ant.apache.org*) version 1.6.2 or higher (but not version 1.6.4—that release was broken), preferably 1.7.x or higher. It must be usable from the shell, like this:

```
# ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
```

You must also have the *rpmbuild* binary available in your shell. In Fedora and Red Hat distributions, this is part of the RPM package named *rpm-build*. You must use version 4.2.1 or higher (the 4.2.0 version that is included with Red Hat 9 has a bug that prevents *rpmbuild* from working properly—but that is becoming antiquated!). Just make sure it's installed and you can run the *rpmbuild* command successfully:

```
# rpmbuild --version
RPM version 4.3.2
```

Download this book's examples archive from *http://catalog.oreilly.com/examples/9780596101060*.

Unpack it like this:

```
$ unzip tomcatbook-examples-2.0.zip
```

Change directory into the *tomcat-package* directory:

```
$ cd tomcatbook-examples/tomcat-package
```

Now, download the binary release archives from the Apache binary releases page at *http://tomcat.apache.org/download-60.cgi*. You should download the latest stable version as listed on the Tomcat home page at *http://tomcat.apache.org*. Download all the *tar.gz* archive files for the version of Tomcat that you've chosen.

Move all the Tomcat binary release archive files into the *tomcatbook-examples/tomcat-package/* directory so they can be included in the RPM package set you're about to build:

```
# cp apache-tomcat-6.0.14*.tar.gz tomcatbook-examples/tomcat-package/
```

Edit the *conf/tomcat-env.sh* file to match the setup of the machines where you'll deploy your Tomcat RPM packages. At the minimum, you should make sure that *JAVA_HOME* is an absolute filesystem path to a Java 1.5 or 1.6 compliant virtual machine (either a JDK or a JRE).

Then, invoke `ant` to build your Tomcat 6 RPM package set:

```
$ ant
```

This should build the Tomcat RPM packages, and when the build is complete, you will find them in the *dist/* directory:

```
# ls dist/
tomcat-6.0.14-0.noarch.rpm  tomcat-6.0.14-0-src.tar.gz
tomcat-6.0.14-0.src.rpm     tomcat-6.0.14-0.tar.gz
```

The Tomcat RPM package builder also builds a Tomcat source RPM package,* plus a *tar.gz* archive of the RPM package as a convenience.

Copy the RPM package to the machine(s) you wish to install it on.

When you're ready to install it, you have two choices:

- Install it into its default path of */opt/tomcat*.
- Install it, relocating it to a path of your choice.

Here's how to install it to the default path:

* Think of this source RPM package as the content necessary to build the binary RPM package, not necessarily the Java source code to Tomcat itself. This book's Tomcat RPM package was built using the officially compiled Tomcat class files, so the Java source isn't included in the source RPM package, nor is it necessary to build the multiplatform "binary" RPM package.

```
# rpm -ivh tomcat-6.0.14-0.noarch.rpm
Preparing...                ########################################### [100%]
   1:tomcat                 ########################################### [100%]
```

The following error:

```
error: Failed dependencies:
        /bin/sh is needed by tomcat-6.0.14-0.noarch
```

usually occurs on operating systems that do not primarily use the RPM package manager, and you are installing this Tomcat RPM package when the RPM package manager's database is empty (no package in the database provides the */bin/sh* interpreter). This may happen, for example, if you are installing the Tomcat RPM package on a Debian Linux OS after installing the *rpm* command.

Try to install it again like this:

```
# rpm -ivh --nodeps tomcat-6.0.14-0.noarch.rpm
```

If you get warnings such as these about users and groups:

```
warning: user tomcat does not exist - using root
warning: group nobody does not exist - using root
```

you need to add a `tomcat` user and `nobody` group by hand using `adduser` and `addgroup`. Just make sure that the tomcat user's primary group is `nobody`. Also, make sure that you set user `tomcat`'s home directory to "*/opt/tomcat/temp*," and set tomcat's login shell to something that doesn't actually work, such as */sbin/nologin* if you have that:

```
# groupadd nobody
# useradd -s /sbin/nologin -d /opt/tomcat/temp -c 'Tomcat User' \
  -g nobody tomcat
```

Once you are done with this, try again to install the `tomcat` package:

```
# rpm -e tomcat
# rpm -ivh --nodeps tomcat-6.0.14-0.noarch.rpm
```

Once it's installed, just verify that the *JAVA_HOME* path set in */opt/tomcat/conf/tomcat-env.sh* points to the 1.5 or 1.6 JVM that you want it to. That's it! Tomcat should be ready to run.

With these same RPM packages, you can install Tomcat and relocate it to a different filesystem path, like this:

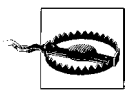
```
# rpm -ivh --prefix /usr/local tomcat-6.0.14-0.noarch.rpm
```

This would install Tomcat, relocating it so that *CATALINA_HOME* becomes */usr/local/tomcat*. You may install the *admin* and *compat* packages this way as well.

As of this writing, JPackage.org does not offer a Tomcat 6 RPM package, but instead offers a Tomcat 5.5 RPM package.

Installing Tomcat from the JPackage.org Linux RPM packages

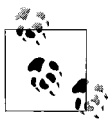
To download and install the JPackage.org Tomcat RPM packages, visit *http://JPackage.org/repos.php*. This page discusses how to configure meta package managers, such as *yum*, *apt-rpm*, *urpmi*, and *up2date*. This is the only reasonable way to install the JPackage.org Tomcat RPM package set due to its large number of installation dependencies. Also, because the details about how to set up the repository configuration for the meta package manager can change at any time, we are not able to show an example of how to do it in this book. See JPackage.org's web site for the details.



The JPackage.org Tomcat 5.5 RPM creates a user and group both named `tomcat5` and runs Tomcat with that user and group. The default shell of the `tomcat5` user is */bin/sh*. Don't try to change this or Tomcat will stop running correctly.

Installing Tomcat on Solaris

Before you install a new Tomcat package on Solaris, you should probably inspect your system to find out if there is already one present and decide if you should remove it. By default, no Tomcat package should be installed, at least on Sun's Solaris 10.



Solaris 9 ships with an older version of Tomcat. Check to see if it's installed:

```
jasonb$ pkginfo | grep -i tomcat
```

If this command outputs one or more packages, a version of Tomcat is installed. To get more information about the package, use *pkginfo* with the `-l` switch. For example, if the preinstalled Tomcat package name was `SUNWtomcat`:

```
jasonb$ pkginfo -l SUNWtomcat
```

Even if Tomcat is installed, it should not cause problems. To be safe, we suggest that you uninstall an existing Tomcat package only if you're prepared to deal with any breakage that removal may cause. If you're sure the package is causing you problems, as the root user, you can remove it:

```
# pkgrm SUNWtomcat
```

To install a Tomcat Solaris package, you need to set your user identity to the root user or else you will not have sufficient permissions to write the files. Usually, this is done either with the `sudo` or `su` commands. For example:

```
# su -
Password:
Sun Microsystems Inc.   SunOS 5.10      Generic January 2005
# id
uid=0(root) gid=0(root)
```

Then, you can proceed with the installation.

Solaris already comes standard with Java 1.5.0, but you should make sure to upgrade it to a newer, more robust version. See Appendix A for details on what to get and where to get it.

As of this writing, the only Solaris package of Tomcat that we could find is a Tomcat 5.5 package included in the Blastwave Solaris Community Software (CSW) package set. This package set is a community supported set of open source packages, analogous to a Linux distribution's package set. See the Blastwave CSW page about it at *http://www.blastwave.org*. The CSW package is best installed via the `pkg-get` command. This command does not come with Solaris, but it is easy to install.

Install `pkg-get` from the URL *http://www.blastwave.org/pkg-get.php*. we were able to use `wget` to download it like this:

```
# PATH=/opt/csw/bin:/usr/sfw/bin:/usr/sfw/sbin:$PATH
# export PATH
# wget http://www.blastwave.org/pkg_get.pkg
```

If that doesn't work (for example, you don't have `wget` installed), just use a web browser to download the *pkg_get.pkg* file to your Solaris machine.

Install the `pkg_get` package like this:

```
# pkgadd -d pkg_get.pkg
```

And hit enter or answer **`y`** at the prompts.

Now, add the path setting to the system's */etc/default/login* file.

First, make it writable by root:

```
# chmod u+w /etc/default/login
```

Then, edit */etc/default/login* and add this:

```
PATH=/opt/csw/bin:/usr/sfw/bin:/usr/sfw/sbin:$PATH
export PATH
```

Then, save the file and put the permissions back:

```
# chmod u-w /etc/default/login
```

Do the same with */etc/profile*, except you shouldn't need to modify its file permissions. Edit */etc/profile* and insert the same lines at the end of the file, and then save it.

Before using `pkg-get`, update `pkg-get`'s catalog, like this:

```
# pkg-get -U
```

Once that's done, you can install packages using `pkg-get`.

Once you have the `pkg-get` command installed and working, you can install Tomcat 5.5. Make sure to switch to the root user; you can install packages from there. Install Tomcat's package like this:

```
# pkg-get install tomcat5
```

There is no CSW package for Tomcat 5.0, so the Tomcat 5.5 package is called *CSWtomcat5*.

If it tells you that some of the scripts must run as the superuser and asks you if you are sure you want to install the packages, just type **y** and hit enter.

Installing the *CSWtomcat5* package also starts it. When the installation is complete, you're already running Tomcat! Test it at the URL *http://localhost:8080*.

Once it is installed, the base install directory is */opt/csw/share/tomcat5*, and the `init` script is installed as */etc/init.d/cswtomcat5*. When you first get this Tomcat package installed, you should read the comments at the top of the `init` script to learn details about your Solaris Tomcat package. The details can change with each revision of the package.

Installing Tomcat on Windows

For Windows systems, Tomcat is available as a Windows-style graphical installer that is available directly from the Apache Software Foundation's Tomcat downloads page. Although you can also install Tomcat from a zipped binary release, the Windows graphical installer does a lot of setup and operating system integration for you as well, and we recommend it. Start by downloading an installer release, such as *apache-tomcat-6.0.14.exe* (or later; unless there is a good reason not to, use the latest available stable version), from the release page at *http://tomcat.apache.org/download-60.cgi*.

When you download and run this installer program, it will first verify that it can find a JDK and JRE, and then prompt you with a license agreement. This license is the Apache Software License, which allows you to do pretty much anything with the software as long as you give credit where it's due. Accept the license as shown in Figure 1-1.

Next, the installer will allow you to select which Tomcat components to install. At the top of the installer window, there is a handy drop-down list from which you can select a different typical packaged set of components (see Figure 1-2). To hand select which components to install, choose `Custom` in the drop-down list, and you may select and deselect any component or subcomponent.

If you want to have Tomcat started automatically and be able to control it from the Services Control Panel, check the box to install the `Service` software.

Then, specify where to install Tomcat. The default is in *C:\Program Files\Apache Software Foundation\Tomcat 6.0*. Change it if you want, as shown in Figure 1-3.

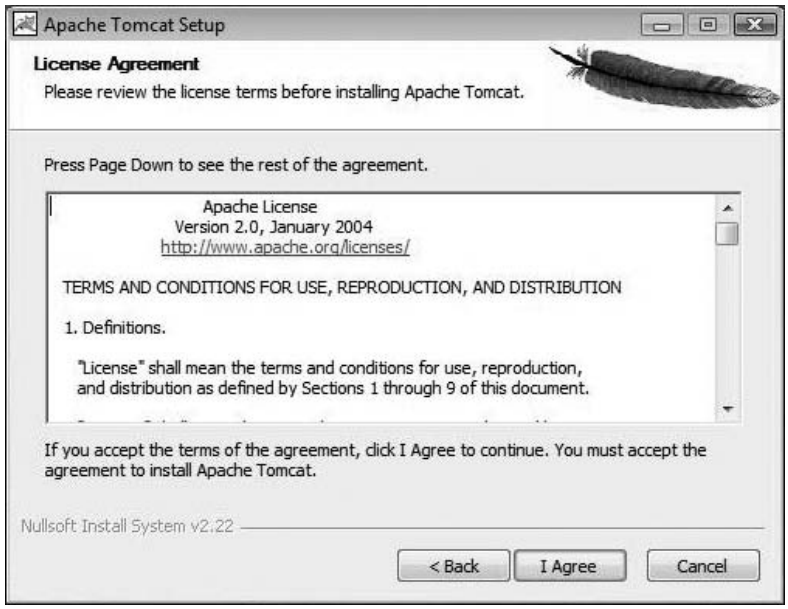


Figure 1-1. The Tomcat installer for Windows: accepting Tomcat's Apache license

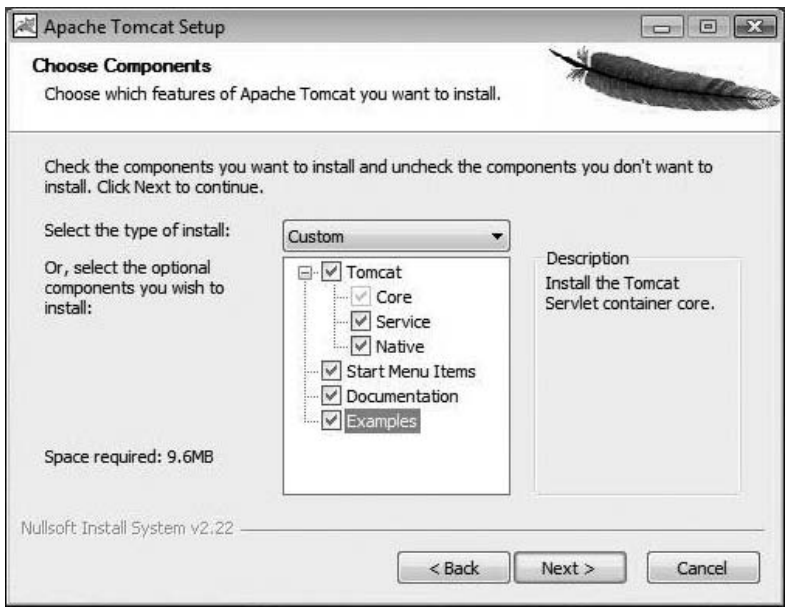


Figure 1-2. Windows choosing Tomcat components to install

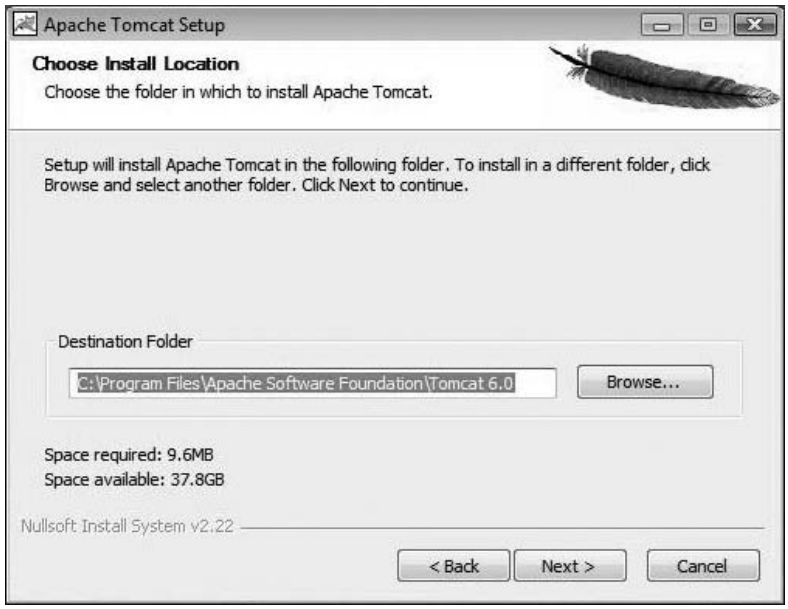


Figure 1-3. Windows installation directory

Next, the installer will prompt you for the HTTP/1.1 connector port—this is Tomcat's web server port. By default it is set to port `8080`, but on Windows feel free to change it to `80` if you want Tomcat to be your first contact web server (Tomcat does a wonderful job in that role). The installer also asks for the administrator login username and password to set for Tomcat. Set the password to something that will not be easily guessed, but don't forget what it is! That will be your username and password to log into Tomcat's Manager webapp.

The installer then allows you to choose a Java runtime for Tomcat from the runtimes you have installed at that time. We suggest Java 1.6.x or higher for this. Once you have configured it with a Java runtime, the `Install` button becomes clickable. Click it and the installer will begin installing Tomcat.

Once the installation completes normally, you should see the message "Completing the Apache Tomcat Setup Wizard" at the end, as shown in Figure 1-4.

From the installer, you can select to start Tomcat and click Finish. Then, in your web browser, type in the URL to your Tomcat, such as *http://localhost:8080*, and you should see the Tomcat start page as shown in Figure 1-5.

Congratulations! Your new Tomcat is installed and ready to use. You now need to start the server for initial testing, as described in the upcoming section "Starting, Stopping, and Restarting Tomcat."

Figure 1-4. Windows installation of Tomcat is complete

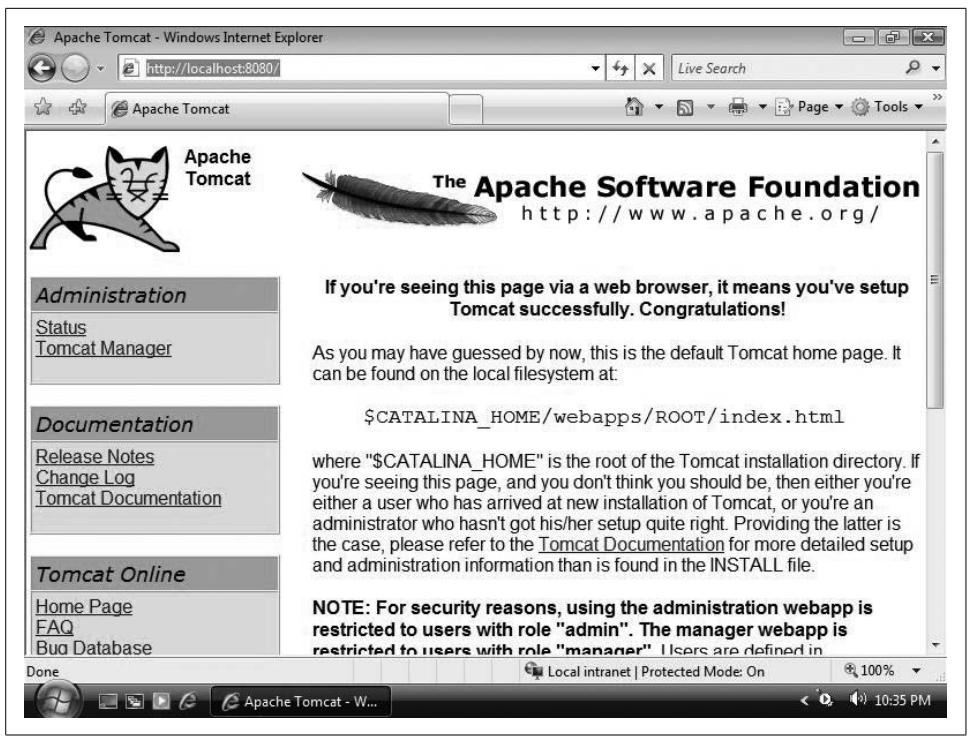


Figure 1-5. Testing Apache Tomcat

Installing Tomcat on Mac OS X

Thanks to the wonderful BSD underpinnings of Mac OS X, installing Tomcat on Mac OS X is similar to the non-RPM Linux installation you have seen. When installing on Mac OS X, you should download the *.tar.gz* file rather than the *.zip* file from the Tomcat site as Unix file permissions are not properly preserved in zip files. In particular, execute permission is lost on the scripts included with Tomcat, making it impossible to start or stop Tomcat until the permissions are restored. Before choosing which version of Tomcat to install, you need to check your Java version as shown below:

```
$ java -version
java version "1.5.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-164)
Java HotSpot(TM) Client VM (build 1.5.0_07-87, mixed mode, sharing)
```

If your Java version is at least 1.5.0, you can install Tomcat 5.5 or Tomcat 6.0. If you do not have a Java runtime of at least version 1.5.0, you cannot install Tomcat 6.0 or higher without first updating Java. If you are running a fully updated version of OS X 10.4 (Tiger) or higher, you have a minimum Java version of 1.5.0. You can download it from the Apple Developer Connection at *http://connect.apple.com*. Register if you have not (it's free!), and then navigate to the Java downloads section; this can also be done via Apple's Software Update. Please ensure that you are installing the latest version of the JDK from Apple. By the time you read this, Apple's 1.6 JDK will almost certainly be available, and we encourage you to install and use it.

These instructions rely on the use of the `sudo` command. On OS X, you must be logged in as a user with administrative privileges to use this command. `sudo` executes a single command as a different user. These instructions use `sudo` to execute commands as the users `root` and `nobody`. You should note that when `sudo` asks for a password, you should enter your login password, not the password for the user you are executing the command as, like you would with the `su` command.

These instructions will install Tomcat to */usr/local/*. There is some debate as to the more appropriate place to install Linux or BSD style programs on OS X with some preferring to use */Library/* rather than */user/local/*.

As on Linux, it is advisable to install Tomcat to run as a nonprivileged user. Some people like to create a special user just for Tomcat, but that is not necessary. It is simpler to use the built-in `nobody` user instead. These instructions use this preexisting user rather than create a new user.

Once you have downloaded the *.tar.gz* file for your chosen version from the Tomcat site, you need to extract it. You can do this from the Finder or from the Terminal as follows (replacing the filename as appropriate):

```
$ tar -xzf apache-tomcat-6.0.14.tar.gz
```

Once you have downloaded and extracted Tomcat, you need to move the files to the folder you are installing to; again you can do this from the Finder, but because we'll need to use the Terminal for the remainder of these instructions, you may as well use it here for this step too. Once you have changed into the folder containing the files you extracted from the *.tar.gz* file, you need to run the following (replacing the filename as needed):

```
$ sudo mv apache-tomcat-6.0.14 /usr/local/
```

Enter your login password, and the directory will be relocated to */usr/local*.

To simplify future upgrades, you should create a symbolic link from */usr/local/tomcat* to the folder you have just moved to */usr/local/*, as follows (again replacing the filename as appropriate):

```
$ sudo ln -s /usr/local/apache-tomcat-6.0.14/ /usr/local/tomcat
```

Tomcat requires two environment variables to run: *JAVA_HOME* and *CATALINA_HOME*. *JAVA_HOME* specifies the Java Virtual Machine to be used by Tomcat, and *CATALINA_HOME* specifies the root directory of the unpacked Tomcat binary (runtime) distribution. They should be set by adding the following lines to the end of */etc/profile* with your favorite text editor (e.g., **`sudo vi /etc/profile`**):

```
export JAVA_HOME=/Library/Java/Home
export CATALINA_HOME=/usr/local/tomcat
```

The above assumes that you are using the default JVM for your version of OS X. If you wish to use a different JVM, you'll have to change the value for *JAVA_HOME*.

Because */etc/profile* is only read when a Terminal is opened, you should close your Terminal and open a new one at this point. You can check that the variables have been set properly as follows:

```
$ echo $JAVA_HOME
/Library/Java/Home
$ echo $CATALINA_HOME
/usr/local/tomcat
```

Later, if you decide to use `launchd` for starting and stopping Tomcat, as we show you below, you do not need the environment variable definitions in */etc/profile*.

You're almost done now; you just need to change the ownership of your Tomcat install to the user `nobody`:

```
$ sudo chown -R nobody:nobody /usr/local/tomcat
$ cd /usr/local
$ ls -l
total 0
drwxr-x---    13 nobody    nobody     442 Sep 27 15:36 apache-tomcat-6.0.14
```

Notice that Tomcat is now by owned `nobody` and has very restrictive permissions for execution.

Tomcat should now be ready to run, although it will not restart on reboots. To see how to make Tomcat run when your server computer boots up, see the upcoming section "Automatic Startup."

Installing Tomcat on FreeBSD

The FreeBSD ports system includes a port of Tomcat 6. See *http://www.freshports.org/www/tomcat6/* for more up-to-date details about it.

First, *make sure* you update your Tomcat 6 port tree. Here's how:

```
# cd /root
# cp /usr/share/examples/cvsup/ports-supfile tc6-supfile
```

Edit the *tc6-supfile*.

Change the lines that say:	To say:
*default host=CHANGE_THIS.FreeBSD.org	*default host=cvsup.FreeBSD.org
ports-all	#ports-all
#ports-www	ports-www

See the end of the page *http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/cvsup.html* to find the best *default host* name to use for your geographical location.

Now, use the modified supfile to update the tree:

```
# pkg_add -r cvsup
# cvsup -g -L 2 tc6-supfile
```

Once the tree is up to date, install it like this:

```
# cd /usr/ports/www/tomcat-6
# make install
```

This does not build Tomcat from its source code. Instead, it "builds" the FreeBSD ports package files by extracting the official Apache binary release archives, and adds FreeBSD-specific packaging files, and then installs them all where they should be installed on FreeBSD. When that is done, edit your */etc/rc.conf* file and add these lines to the end:

```
tomcat60_enable="YES"
tomcat60_java_opts="-Djava.net.preferIPv4Stack=true"
```

The first line enables the `RCng` `init` script—this `init` script has code that will not try to start Tomcat unless the `tomcat60_enable` variable is enabled this way, to prevent Tomcat from accidentally starting at boot time. Adding the second line will avoid a problem that prevents Tomcat from opening its TCP server ports.

Starting, Stopping, and Restarting Tomcat

Once you have the installation completed, you will probably be eager to start Tomcat and see if it works. This section details how to start up and shut down Tomcat, including specific information on each supported operating system. It also details common errors that you may encounter, enabling you to quickly identify and resolve any problems you run into.

Starting Up and Shutting Down

The correct way to start and stop Tomcat depends on how you installed it. For example, if you installed Tomcat from a Linux RPM package, you should use the `init` script that came with that package to start and stop Tomcat. Or, if you installed Tomcat on Windows via the graphical installer from *tomcat.apache.org*, you should start and stop Tomcat as you would any Windows service. Details about each of these package-specific cases are given in the next several sections. If you installed Tomcat by downloading the binary release archive (*.zip* or *.tar.gz*) from the Tomcat downloads page—what we'll call the generic installation case—you should use the command-line scripts that reside in the *CATALINA_HOME/bin* directory.

There are several scripts in the *bin* directory that you will use for starting and stopping Tomcat. All the scripts you will need to invoke directly are provided both as shell script files for Unix (*.sh*) and batch files for Windows (*.bat*). Table 1-1 lists these scripts and describes each. When referring to these, we have omitted the filename extension because *catalina.bat* has the same meaning for Microsoft Windows users that *catalina.sh*[*] has for Unix users. Therefore, the name in the table appears simply as *catalina*. You can infer the appropriate file extension for your system.

Table 1-1. Tomcat invocation scripts

Script	Purpose
`catalina`	The main Tomcat script. This runs the `java` command to invoke the Tomcat startup and shutdown classes.
`cpappend`	This is used internally, and then only on Windows systems, to append items to Tomcat classpath environment variables.
`digest`	This makes a `crypto` digest of Tomcat passwords. Use it to generate encrypted passwords.
`service`	This script installs and uninstalls Tomcat as a Windows service.
`setclasspath`	This is also only used internally and sets the Tomcat classpath and several other environment variables.
`shutdown`	This runs *catalina stop* and shuts down Tomcat.
`startup`	This runs *catalina start* and starts up Tomcat.

[*] Linux, BSD, and Unix users may object to the *.sh* extension for all of the scripts. However, renaming these to your preferred conventions is only temporary, as the *.sh* versions will reappear on your next upgrade. You are better off getting used to typing *catalina.sh*.

Table 1-1. Tomcat invocation scripts (continued)

Script	Purpose
`tool-wrapper`	This is a generic Tomcat command-line tool wrapper script that can be used to set environment variables and then call the main method of any fully qualified class that is in the classpath that is set. This is used internally by the `digest` script.
`version`	This runs the *catalina version*, which outputs Tomcat's version information.

The main script, `catalina`, is invoked with one of several arguments. The most common arguments are `start`, `run`, or `stop`. When invoked with `start` (as it is when called from *startup*), it starts up Tomcat with the standard output and standard error streams directed into the file *CATALINA_HOME/logs/catalina.out*. The `run` argument causes Tomcat to leave the standard output and error streams where they currently are (such as to the console window) useful for running from a terminal when you want to see the startup output. This output should look similar to Example 1-1.

Example 1-1. Output from catalina run

```
ian:389$ bin/catalina.sh start
Using CATALINA_BASE:   /home/ian/apache-tomcat-6.0.14
Using CATALINA_HOME:   /home/ian/apache-tomcat-6.0.14
Using CATALINA_TMPDIR: /home/ian/apache-tomcat-6.0.14/temp
Using JRE_HOME:       /usr/java/jdk1.6.0_02
Sep 27, 2007 10:42:16 PM org.apache.catalina.core.AprLifecycleListener lifecycleEvent
INFO: The Apache Tomcat Native library which allows optimal performance in produ ction
environments was not found on the java.library.path: /usr/java/jdk1.5.0_06/bin/../jre/bin:
/usr/lib
Sep 27, 2007 10:42:17 PM org.apache.coyote.http11.Http11BaseProtocol init
INFO: Initializing Coyote HTTP/1.1 on http-8080
Sep 27, 2007 10:42:17 PM org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 948 ms
Sep 27, 2007 10:42:17 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Sep 27, 2007 10:42:17 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.14
Sep 27, 2007 10:42:17 PM org.apache.catalina.core.StandardHost start
INFO: XML validation disabled
Sep 27, 2007 10:42:27 PM org.apache.coyote.http11.Http11BaseProtocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Sep 27, 2007 10:42:28 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Sep 27, 2007 10:42:29 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/106  config=null
INFO: Find registry server-registry.xml at classpath resource
Sep 27, 2007 10:42:30 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1109 ms
```

If you use *catalina* with the `start` option or invoke the `startup` script instead of using the `run` argument, you see only the first few `Using...` lines on your console; all the rest of the output is redirected into the *catalina.out* logfile. The `shutdown` script invokes *catalina* with the argument `stop`, which causes Tomcat to connect to the default port specified in your `Server` element (discussed in Chapter 7) and send it a shutdown message. A complete list of startup options is listed in Table 1-2.

Table 1-2. Startup options

Option	Purpose
`-config [`*`server.xml`* `file]`	This specifies an alternate *server.xml* configuration file to use. The default is to use the *server.xml* file that resides in the *$CATALINA_BASE/conf* directory. See the "server.xml" section in Chapter 7 for more information about *server.xml*'s contents.
`-help`	This prints out a summary of the command-line options.
`-nonaming`	This disables the use of JNDI within Tomcat.
`-security`	This enables the use of the *catalina.policy* file.
`debug`	This starts Tomcat in debugging mode.
`embedded`	This allows Tomcat to be tested in an embedded mode, and is usually used by application server developers.
`jpda start`	This starts Tomcat as a Java Platform Debugger Architecture-compliant debugger. See Sun's JPDA documentation at *http://java.sun.com/products/jpda*.
`run`	This starts up Tomcat without redirecting the standard output and errors.
`start`	This starts up Tomcat, with standard output and errors going to the Tomcat logfiles.
`stop`	This stops Tomcat.
`version`	This outputs Tomcat's version information.

Environment variables

To prevent runaway programs from overwhelming the operating system, Java runtime environments feature limits such as "maximum heap size." These limits were established when memory was more expensive than at present; for JDK 1.3, for example, the default limit was only 32 MB. However, there are options supplied to the `java` command that let you control the limits. The exact form depends upon the Java runtime, but if you are using the Sun runtime, you can enter:

```
java -Xmx=256M MyProg
```

This will run a class file called `MyProg` with a maximum memory size of 256 MB for the entire Java runtime process.

These options become important when using Tomcat, as running servlets can begin to take up a lot of memory in your Java environment. To pass this or any other option into the `java` command that is used to start Tomcat, you can set the option in the environment variable `JAVA_OPTS` before running one of the Tomcat startup scripts.

Windows users should set this environment variable from the Control Panel, and Unix users should set it directly in a shell prompt or login script:

```
$ export JAVA_OPTS="-Xmx256M"  # Korn and Bourne shell
C:\> set JAVA_OPTS="-Xmx256M"  # MS-DOS
$ setenv JAVA_OPTS "-Xmx256M"  # C-shell
```

Other Tomcat environment variables you can set are listed in Table 1-3.

Table 1-3. Tomcat environment variables

Option	Purpose	Default
CATALINA_BASE	This sets the base directory for writable or customized portions of a Tomcat installation tree, such as logging files, work directories, Tomcat's *conf* directory, and the *webapps* directory. It is an alias for CATALINA_HOME.	Tomcat installation directory
CATALINA_HOME	This sets the base directory for static (read-only) portions of Tomcat, such as Tomcat's *lib* directories and command-line scripts.	Tomcat installation directory
CATALINA_OPTS	This passes through Tomcat-specific command-line options to the *java* command.	None
CATALINA_TMPDIR	This sets the directory for Tomcat temporary files.	*CATALINA_HOME/temp*
JAVA_HOME	This sets the location of the Java runtime or JDK that Tomcat will use.	None
JRE_HOME	This is an alias to *JAVA_HOME*.	None
JAVA_OPTS	This is where you may set any Java command-line options.	None
JPDA_TRANSPORT	This variable may set the transport protocol used for JPDA debugging.	dt_socket
JPDA_ADDRESS	This sets the address for the JPDA used with the catalina jpda start command.	8000
JSSE_HOME	This sets the location of the Java Secure Sockets Extension used with HTTPS.	None
CATALINA_PID	This variable may optionally hold the path to the process ID file that Tomcat should use when starting up and shutting down.	None

Starting and stopping: The general case

If you have installed Tomcat via an Apache binary release archive (either a *.zip* file or a *.tar.gz* file), change directory into the directory where you installed Tomcat:

```
$ cd apache-tomcat-6.0.14
```

Echo your $JAVA_HOME environment variable. Make sure it's set to the absolute path of the directory where the Java installation you want Tomcat to use resides. If it's not, set it and export it now. It's OK if the java interpreter is not on your $PATH because Tomcat's scripts are smart enough to find and use Java based on your setting of $JAVA_HOME.

Make sure you're not running a TCP server on port 8080 (the default Tomcat HTTP server socket port), nor on TCP port 8005 (the default Tomcat shutdown server socket port). Try running *telnet localhost 8080* and *telnet localhost 8005* to see if any existing server accepts a connection, just to be sure.

Start up Tomcat with its `startup.sh` script like this:

```
$ bin/startup.sh
Using CATALINA_BASE:   /home/jasonb/apache-tomcat-6.0.14
Using CATALINA_HOME:   /home/jasonb/apache-tomcat-6.0.14
Using CATALINA_TMPDIR: /home/jasonb/apache-tomcat-6.0.14/temp
Using JAVA_HOME:       /usr/java/jdk1.6.0_02
```

You should see output similar to this when Tomcat starts up. Once started, it should be able to serve web pages on port 8080 (if the server is *localhost*, try *http://localhost:8080* in your web browser).

Invoke the `shutdown.sh` script to shut Tomcat down:

```
$ bin/shutdown.sh
Using CATALINA_BASE:   /home/jasonb/apache-tomcat-6.0.14
Using CATALINA_HOME:   /home/jasonb/apache-tomcat-6.0.14
Using CATALINA_TMPDIR: /home/jasonb/apache-tomcat-6.0.14/temp
Using JAVA_HOME:       /usr/java/jdk1.6.0_02
```

Starting and stopping on Linux

If you've installed Tomcat via the RPM package on Linux, you can test it out by issuing a start command via Tomcat's `init` script, like this:

```
# /etc/rc.d/init.d/tomcat start
Starting tomcat:                                           [  OK  ]
```

Or, on some Linux distributions, such as Fedora and Red Hat, to do the same thing, you may instead type the shorter command:

```
# service tomcat start
```

If you installed the JPackage.org Tomcat RPM package, the name of the `init` script is *tomcat55*, so the command would be:

```
# /etc/rc.d/init.d/tomcat55 start
```

Then, check to see if it's running:

```
# ps auwwx | grep catalina.startup.Bootstrap
```

You should see several Java processes scroll by. Another way to see whether Tomcat is running is to request a web page from the server over TCP port 8080.

If Tomcat fails to startup correctly, go back and make sure that the */opt/tomcat/conf/tomcat-env.sh* file has all the right settings for your server computer (in the JPackage.org RPM installation case, it's the */etc/tomcat55/tomcat55.conf* file). Also check out the "Common Errors" section, later in this chapter.

To stop Tomcat, issue a stop command like this:

```
# /etc/rc.d/init.d/tomcat stop
```

Or (shorter):

```
# service tomcat stop
```

Starting and stopping on Solaris

To use Tomcat's `init` script on Solaris, you must be the `root` user. Switch to `root` first. Then, you can start Tomcat like this:

```
# /etc/init.d/cswtomcat5 start
```

And, you can stop it like this:

```
# /etc/init.d/cswtomcat5 stop
```

Watch your *catalina.out* logfile in */opt/csw/share/tomcat5/logs* so that you'll know if there are any errors.

Starting and stopping on Windows

On Microsoft Windows, Tomcat can be started and stopped either as a windows service or by running a batch file. If you arrange for automatic startup (detailed later in this chapter), you may manually start Tomcat in the control panel. If not, you can start Tomcat from the desktop icon.

If you have Tomcat running in a console window, you can interrupt it (usually with Ctrl-C) and it will catch the signal and shut down:

```
Apache Tomcat/6.0.14
^C
Stopping service Tomcat-Standalone
C:\>
```

If the graceful shutdown does not work, you need to find the running process and terminate it. The JVM running Tomcat will usually be identified as a Java process; be sure you get the correct Java if other people or systems may be using Java. Use Ctrl-Alt-Delete to get to the task manager, select the correct Java process, and click on End Task.

Starting and stopping on Mac OS X

The Mac OS X installation of Tomcat is simply the binary distribution, which means you can use the packaged shell scripts that come with the Apache binary release. This provides a quick and easy set of scripts to start and stop Tomcat as required. First, we will show you the general case for starting and stopping Tomcat on Mac OS X.

Mac OS X sets all your paths for you so all you need to do is ensure that there are no TCP services already running on port 8080 (the default Tomcat HTTP server socket

port), nor on port 8005 (the default Tomcat shutdown port). This can be done easily by running the `netstat` command:

```
$ netstat -an | grep 8080
```

You should see no output. If you do, it means another program is listening on port 8080, and you should shut it down first, or you must change the port numbers in your *CATALINA_HOME/conf/server.xml* configuration file. Do the same for port 8005.

Tomcat can be started on OS X with the following command:

```
$ cd /; sudo -u nobody /usr/local/tomcat/bin/startup.sh; cd -
```

Tomcat can be stopped with the following command:

```
$ cd /; sudo -u nobody /usr/local/tomcat/bin/shutdown.sh; cd -
```

Because the user `nobody` is an unprivileged user, a lot of folders on your disk are not accessible to it. This is of course a good thing, but because the scripts for starting and stopping Tomcat attempt to determine the current directory, you will get errors if the scripts are not being called from a folder to which the user `nobody` has read access. To avoid this, the above commands consist of three subcommands. First, they change to the root folder (`/`), next they call script to start or stop Tomcat as the user `nobody`, and finally they return to the folder they started in. If you are running the commands from a folder to which the user `nobody` has read access (e.g., `/`), you can shorten the commands by leaving out the first and last parts as follows:

```
$ sudo -u nobody /usr/local/tomcat/bin/startup.sh
$ sudo -u nobody /usr/local/tomcat/bin/shutdown.sh
```

Later in the "Automatic Startup on Mac OS X" section, we show you how to create and install `init` scripts via Apple's `launchd`, as you see in the Linux RPM installations and the BSD port installs, to allow you to not only start and stop Tomcat, but also to automatically start Tomcat on boot—the Mac OS X way!

Starting and stopping on FreeBSD

This port installs Tomcat into the root path */usr/local/tomcat6.0/*. The behavior of Tomcat may be configured through variables in your */etc/rc.conf* file, which override settings that are contained in the */etc/defaults/rc.conf* file. This port includes an `RCng` script named *${PREFIX}/etc/rc.d/tomcat60.sh*. By default, this ends up being */usr/local/etc/rc.d/tomcat60.sh*. Read the top of this file to see what Tomcat variable settings you may apply in your */etc/rc.conf* file.

Try starting Tomcat like this:

```
# /usr/local/etc/rc.d/tomcat60.sh start
Starting tomcat60.
```

This will only work if you have added this line to your */etc/rc.conf* file:

```
tomcat60_enable="YES"
```

You may use the `tomcat60.sh` script to start, stop, and restart Tomcat 6.

By default, this FreeBSD port of Tomcat 6.0 sets Tomcat's default HTTP port to be 8180, which is different than the default that is originally set (for all operating systems) in the Apache Software Foundation's distribution of Tomcat (which is 8080). Try accessing your FreeBSD Tomcat port via the URL *http://localhost:8180/*.

Common Errors

Several common problems can result when you try to start up Tomcat. While there are many more errors that you can run into, these are the ones we most often encounter.

Another server is running on port 80 or 8080
: Ensure that you don't have Tomcat already started. If you don't, check to see if other programs, such as another Java application server or Apache Web Server, are running on these ports.

Another instance of Tomcat is running
: Remember that not only must the HTTP port of different Tomcat instances (JVMs) be different, every port number in the `Server` and `Connector` elements in the *server.xml* files must be different between instances. For more information on these elements, consult Chapter 7.

Restarting Tomcat

At the time of this writing, there is no restart script that is part of the Tomcat 6.0 distribution because it is tough to write a script that can make sure that when Tomcat stops, it shuts down properly before being started up again. The reasons outlined below for Tomcat shutdowns being unreliable are almost exclusively *edge conditions*. That means they don't usually happen, but that they can occur in unusual situations. Here are some reasons why shutdowns may be unreliable:

- The Java Servlet Specification does not mandate any time limit for how long a Java servlet may take to perform its work. Writing a servlet that takes forever to perform its work does not break compliance with the Java Servlet Specification, but it can prevent Tomcat from shutting down.
- The Java Servlet Specification also dictates that on shutdowns, servlet containers must wait for each servlet to finish serving all requests that are in progress before taking the servlet out of service, or wait a container-specific timeout duration before taking servlets out of service. For Tomcat 6, that timeout duration is a maximum of a half-second per servlet. When a servlet misbehaves and takes too long to finish serving requests, it's up to Tomcat to figure out that the servlet has taken too long and forcibly take it out of service so that Tomcat can shut down. This processing takes time, though, and slows Tomcat's own shutdown processing.

- Multithreading in Java virtual machines is specified in a way that means that Java code will not always be able to tell exactly how much real time is going by (Java SE is not a real-time programming environment). Also, due to the way Java threads are scheduled on the CPU, threads can become blocked and stay blocked. Because of these limitations, the Java code that is called on invocations of *shutdown.sh* will not always know how long to wait for Tomcat to shut down, nor can Tomcat always know it's taking too long to shut down. That means that shutdowns are not completely reliable when written in pure Java. An external program would need to be written in some other programming language to reliably shut down Tomcat.
- Because Tomcat is an embeddable servlet container, it tries not to call `System.exit(0)` when shutting down the server because Tomcat does not know what else may need to stay running in the same Java virtual machine. Instead, Tomcat shuts down all of its own threads so that the VM can exit gracefully if nothing else needs to run. Because of that, a servlet could spawn a thread that would keep the VM from exiting even when Tomcat's threads are all shut down.
- The Java Servlet Specification allows servlets to create additional Java threads that perform work as long as any security manager allows it.* Once another thread is spawned from a servlet, it can raise its own priority higher than Tomcat's threads' priorities (if the security manager allows) and could keep Tomcat from shutting down or from running at all. Usually if this happens, it's not malicious code but buggy code. Try not to do this!
- If your Tomcat instance has run completely out of memory (as evidenced by the dreaded "Permgen memory" error in the logs), it will usually be unable to accept new connections on its web port *or* on its shutdown port.

To fix some of the problems, you may want to configure and use a security manager. See Chapter 6 for more information on how to place limits on webapps to guard against some of these problems.

The general case

If you installed Tomcat "by hand" by downloading and unpacking an official binary release archive (*tar.gz* or *.zip*) from *tomcat.apache.org*, regardless of the operating system you're using, here is the standard way to restart Tomcat:

1. Change directory into the root of the Tomcat installation directory (commonly known as the *CATALINA_HOME* directory):

```
$ cd apache-tomcat-6.0.14
```

* An urban legend about developing Java webapps says that according to the Java Servlet Specification, servlets in webapps are not allowed to spawn any Java threads. That is false. The servlet specification does not preclude doing this, so it is OK to spawn one or more threads as long as the thread(s) are well behaved. This is often the rub, since webapp developers often report bugs against Tomcat that turn out to be caused by their own code running in a separate thread.

2. Issue a shutdown via the `shutdown.sh` script:

```
$ bin/shutdown.sh
```

3. Decide how long you want to wait for Tomcat to shut down gracefully, and wait that period of time. Reasonable maximum shutdown durations depend on your web application, your server computer's hardware, and how busy your server computer is, but in practice, Tomcat often takes several seconds to completely shut down.
4. Query your operating system for *java* processes to make sure it shut down. On Windows, hit Ctrl-Alt-Delete to get to the task manager, and scroll through the list to look for it. On all other operating systems, use the `jps` command to look for any remaining Tomcat processes that are your Tomcat's Java virtual machine. The `jps` command comes with Java. Try this:

```
$ jps | grep Bootstrap
```

   If that fails, use an OS-dependent Process Status (ps) command, such as this:

```
$ ps auwwx | grep catalina.startup.Bootstrap \
    # On Linux or *BSD

$ /usr/ucb/ps auwwx | grep catalina.startup.Bootstrap \
    # On Solaris
```

5. If no Tomcat *java* processes are running, skip to step 6. Otherwise, because the Tomcat JVM is not shutting down in the time you've allowed, you may want to force it to exit. Send a `TERM` signal to the processes you find, asking the JVM to perform a shutdown (ensuring you have the correct user permissions):

```
$ kill -TERM <process-ID-list>
```

6. Do another `ps` like you did in step 4. If the Tomcat JVM processes remain, repeat step 5 until they're gone. If they persist, have your operating system kill the *java* process. On Windows, use the task manager to end the task(s). On all other operating systems, use the kill command to tell the kernel to kill the process(es) like this:

```
$ kill -KILL <process-ID-list>
```

7. Once you're sure that Tomcat's JVM is no longer running, start a new Tomcat process:

```
$ bin/startup.sh
```

Usually, the shutdown process goes smoothly and Tomcat JVMs shut down quickly. But, because there are situations when they don't, the above procedure should always suffice. We realize this is not a very convenient way to restart Tomcat; however, if you try to cut corners here, you will likely not always shut down Tomcat and get errors due to the new Tomcat JVM bumping into the existing Tomcat JVM when you go to start it again. Luckily, for most operating systems, there are scripts that automate this entire procedure, implemented as a "restart" command. You'll find these integrated into most OS-specific Tomcat installation packages (Linux RPM packages, the FreeBSD port, etc.).

Restarting Tomcat on Linux

The following outlines how to reliably restart Tomcat on Linux. If you have installed Tomcat via an RPM package, either the one from this book or the one from JPackage.org, restarting Tomcat is easy. If you installed the RPM package from this book, do:

```
# service tomcat restart
```

And, if you installed the JPackage.org RPM package, do:

```
# service tomcat55 restart
```

which should reliably restart Tomcat. Be sure to check your logfiles for any startup problems.

Restarting Tomcat on Solaris

The following outlines how to reliably restart Tomcat on Solaris. If you have installed Tomcat via a Blastwave Solaris CSW package, restarting Tomcat is easy:

```
# /etc/init.d/cswtomcat5 restart
```

That should restart Tomcat. Be sure to check your logfiles for any startup problems.

As of this writing, the Blastwave package's `init` script does not contain any code to reliably restart Tomcat—it does not watch the processes to make sure that they came down all the way, nor does it try to force the processes down if they do not come down on their own. Read the `init` script source and you'll see what we mean. So, it is up to the Solaris administrator to ensure (by hand) that the restart actually occurred.

Restarting the Tomcat Windows Service

If you have Tomcat running as a Windows Service, you can restart it from the control panel. Either right-click on the service and select Restart from the pop-up menu or, if it exists on your version of Windows, use the Restart button near the upper-right corner of the dialog box (see Figure 1-6).

Be sure to check your logfiles for any startup problems.

Restarting Tomcat on Mac OS X

The standard way to restart Tomcat on OS X is to stop and then start Tomcat.

If you have chosen to use the generic way to start Tomcat, there is no easy way to restart Tomcat in Mac OS X and the best solution is to call `shutdown.sh`. Then, just as described in the Linux section of this chapter, you would decide how long you will wait for Tomcat to shut down and take the appropriate steps, as outlined above.

A simple way to see if Tomcat is running is to check if there is a service listening on TCP port 8080 with the `netstat` command. This will, of course, only work if you are running Tomcat on the port you specify (its default port of 8080, for example) and not running any other service on that port.

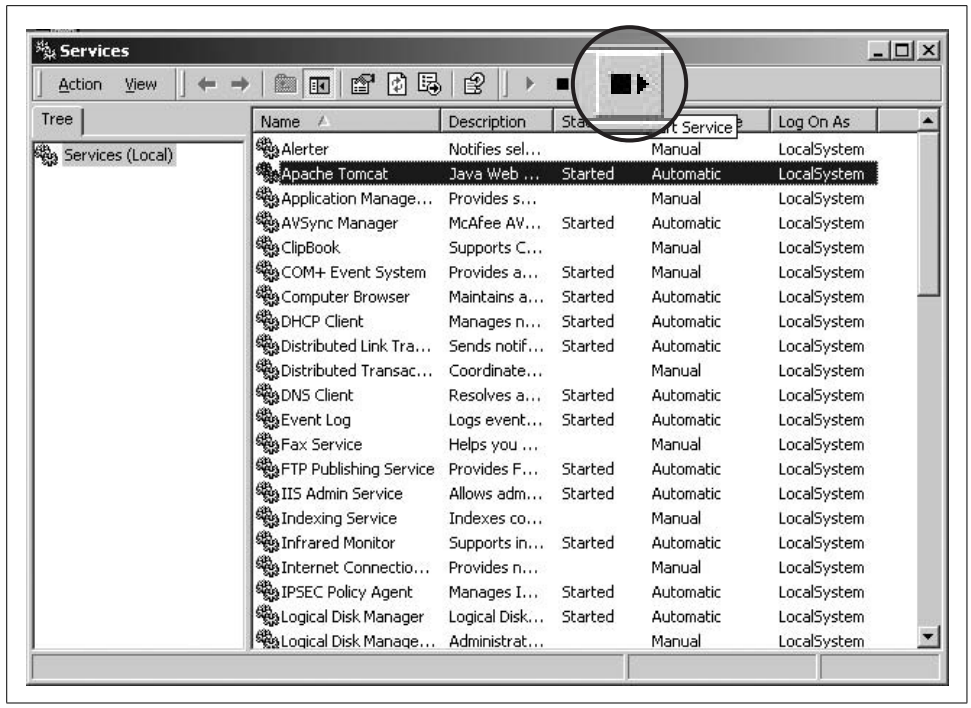


Figure 1-6. Restart button in Control Panel

First, shut down the currently running Tomcat instance:

```
$ netstat -an | grep 8080
tcp46      0      0  *.8080                 *.*                    LISTEN
$ cd /; sudo -u nobody /usr/local/tomcat/bin/shutdown.sh; cd -
Using CATALINA_BASE:   /usr/local/tomcat
Using CATALINA_HOME:   /usr/local/tomcat
Using CATALINA_TMPDIR: /usr/local/tomcat/temp
Using JRE_HOME:       /Library/Java/Home/Users/bart
```

Then, check to make sure Tomcat is no longer running:

```
$ netstat -an | grep 8080
```

You should see no output, meaning that Tomcat has shut down. Then, you may start it back up again, like this:

```
$ cd /; sudo -u nobody /usr/local/tomcat/bin/startup.sh; cd -
Using CATALINA_BASE:   /usr/local/tomcat
Using CATALINA_HOME:   /usr/local/tomcat
Using CATALINA_TMPDIR: /usr/local/tomcat/temp
Using JRE_HOME:       /Library/Java/Home/Users/bart
```

After waiting some seconds, check to make sure that Tomcat is running and listening on port 8080 again:

```
$ netstat -an | grep 8080
tcp46      0      0  *.8080                 *.*                    LISTEN
```

If you have chosen to use the automatic startup and shutdown scripts for Tomcat via Apple's `launchd` (see the section "Automatic Startup on Mac OS X," later in this chapter, for details about how to set that up), it's very easy to restart Tomcat simply by unloading the service and reloading it into `launchd`:

```
$ sudo launchctl unload /Library/LaunchDaemons/tomcat.plist
$ sudo launchctl load /Library/LaunchDaemons/tomcat.plist
```

Restarting Tomcat on FreeBSD

The following outlines how to reliably restart Tomcat on FreeBSD. You can restart the Tomcat 6 port by running:

```
# /usr/local/etc/rc.d/tomcat60.sh restart
```

That should reliably restart Tomcat. Be sure to check your logfiles for any startup problems.

Automatic Startup

Once you have Tomcat installed and running, you can set it to start automatically when your system reboots. This will ensure that every time your system comes up, Tomcat will be running and handling requests. Unix users will make changes to their `init` scripts, and Windows users will need to set Tomcat up as a service. Both approaches are outlined in this section.

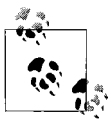
Automatic Startup on Linux

If you've installed Tomcat via an RPM package, getting it to run on a reboot is just a matter of telling your system to run the *tomcat* or *tomcat55* service (depending on which RPM package you installed) when it enters a multiuser run level.

If you know how to use `chkconfig`, as the root user you can simply `chkconfig tomcat on` for the run level(s) of your choice.

Use the `chkconfig` command to make the `tomcat` service start in the run level(s) of your choice. Here's an example of how to make it start in run levels 2, 3, 4, and 5:

```
# chkconfig --level 2345 tomcat on
```



If `chkconfig` does not see the `tomcat` service, try `tomcat55` instead (the *JPackage.org* RPM package's `init` script has this name). Otherwise, you probably did not install Tomcat as an RPM package. Below, we show how to add a simple init script to make it work anyway.

Now, query your configuration to make sure that startup is actually set:

```
# chkconfig --list tomcat
tomcat    0:off   1:off   2:on   3:on    4:on    5:on    6:off
```

Now, reboot and see if Tomcat starts up when the system comes back up.

If you didn't use the RPM package to install Tomcat, you can still set up Tomcat to start on reboots. Tomcat does not come with a Linux `init` script, but it is simple to create one that would just start Tomcat at boot time and stop it on shutdown.- Example 1-2 is a very simple Tomcat `init` script for Linux.

Example 1-2. A Tomcat init script for Linux

```
#!/bin/sh
# Tomcat init script for Linux.
#
# chkconfig: 2345 96 14
# description: The Apache Tomcat servlet/JSP container.

JAVA_HOME=/usr/java/jdk1.6.0_02
CATALINA_HOME=/opt/apache-tomcat-6.0.14
export JAVA_HOME CATALINA_HOME

exec $CATALINA_HOME/bin/catalina.sh $*
```

Save this script in a file named *tomcat* and change the file ownership and group to `root`, and then `chmod` it to 755:

```
# chown root.root tomcat
# chmod 755 tomcat
```

Copy the script to the */etc/rc.d/init.d* directory after modifying the `JAVA_HOME` and `CATALINA_HOME` environment variables to fit your system. Then, set the new `tomcat` service to start and stop automatically by using `chkconfig`, as shown earlier in this section.

Automatic Startup on Solaris

If you have installed Tomcat via a Blastwave Solaris CSW package, your Tomcat has been preconfigured to start at boot time. You do not have to do anything extra to make it work.

If not, you'll need to create yourself a simple `init` script, as shown for Linux in the previous section; it should work fine. Save it to */etc/init.d/tomcat* and set the permissions like this:

```
# chmod 755 /etc/init.d/tomcat
# chown root /etc/init.d/tomcat
# chgrp sys /etc/init.d/tomcat
```

Set the new `tomcat` service to start and stop automatically by symbolically linking it into the */etc/rc3.d* directory (as the `root` user):

```
# ln -s /etc/init.d/tomcat /etc/rc3.d/S63tomcat
# ln -s /etc/init.d/tomcat /etc/rc3.d/K37tomcat
```

The numbers `S63` and `K37` may be varied according to what other startup scripts you have; the `S` number controls the startup sequence and the `K` number controls the shutdown (kill) sequence. The system startup program init invokes all files matching */etc/rc3.d/S** with the parameter `start` as part of normal system startup, and start is just the right parameter for `catalina.sh`. The `init` program also invokes each script file named *rc3.d/K** with the parameter `stop` when the system is being shut down.

Automatic Startup on Windows

Under Windows, Tomcat can be run as a Windows service. Although you can use this to start and stop the server, the most common reason for creating a Tomcat service is to ensure that it is started each time your machine boots up.

Your first task is to find the Services control panel. On a standard Windows install, this requires accessing several menus: Start Menu → Programs → Administrative Tools → Services. Alternately, you can go Start Menu → Settings → Control Panel, and then double-click on Administrative Tools, and again on Services. Once you have the Services control panel, locate the entry for Apache Tomcat (the entries are normally in alphabetical order), and double-click on it, as shown in Figure 1-7.

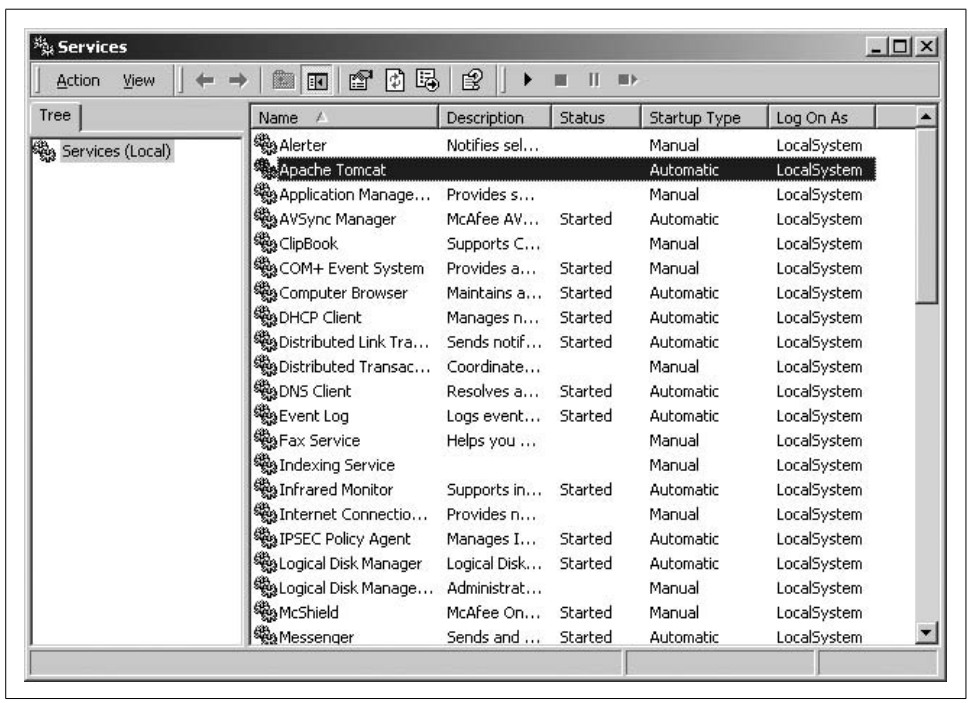


Figure 1-7. Automatic startup under Windows

In the Apache Tomcat Properties dialog box, you should ensure that the startup type is set to Automatic rather than Manual, which will cause Tomcat to start up whenever your machine reboots.

Automatic Startup on Mac OS X

Mac OS X, like most other operating systems, uses system `init` scripts to allow you to start, stop, and restart services automatically just as you would on a Linux system via */etc/rc.d/init.d* or via BSD's */etc/init.d*. In Mac OS X Tiger (10.4), Apple has introduced a new central system-wide controller called `launchd`.* `launchd` gives you more flexibility over how services are controlled and who can access these services. It provides a very simple property list (*plist*) configuration file that allows you to set up what daemon runs and how the daemon is accessed. Due to the differences† in behavior between how `launchd` expects the daemon it has launched to react and how the Tomcat scripts operate, we have to create a shell script that won't fork or have the parent process exit to overcome this problem.

Let's create the script for usage in the *tomcat.plist* and put it in the Tomcat installation binary directory (both the following shell script and the *.plist* file are included in the book's examples; you may download them from *http://www.oreilly.com/catalog/9780596101060*):

```
$ vi /usr/local/tomcat/bin/tomcat-launchd.sh
#!/bin/bash
# Shell script to launch a process that doesn't quit after launching the JVM
# This is required to interact with launchd correctly.

function shutdown( )
{
        $CATALINA_HOME/bin/catalina.sh stop
}

export CATALINA_HOME=/usr/local/tomcat
export TOMCAT_JVM_PID=/tmp/$$

. $CATALINA_HOME/bin/catalina.sh start

# Wait here until we receive a signal that tells Tomcat to stop..
trap shutdown HUP INT QUIT ABRT KILL ALRM TERM TSTP

wait `cat $TOMCAT_JVM_PID`
```

Next, we need to create the `launchd` property list file for Tomcat. Load up your favorite text editor and edit *tomcat.plist*:

* You can find a detailed overview on Apple's support page related to this great new service: *http://developer.apple.com/macosx/launchd.html*.

† `launchd` expects the service to be started and run until signaled, whereas the scripts for Tomcat (`catalina.sh`) launch the Tomcat JVM and then quit. This is a mismatch that the `tomcat-launchd.sh` script fixes.

```
$ vi tomcat.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Disabled</key>
        <false/>
        <key>EnvironmentVariables</key>
        <dict>
                <key>CATALINA_HOME</key><string>/usr/local/tomcat</string>
                <key>JAVA_HOME</key><string>/System/Library/Frameworks/JavaVM.
framework/Home</string>
        </dict>
        <key>Label</key><string>org.apache.tomcat</string>
        <key>OnDemand</key><false/>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/local/tomcat/bin/tomcat-launchd.sh</string>
        </array>
        <key>RunAtLoad</key><true/>
        <key>ServiceDescription</key><string>Apache Tomcat</string>
        <key>StandardErrorPath</key><string>usr/local/tomcat/logs/launchd.stderr</
string>
        <key>StandardOutPath</key><string>usr/local/tomcat/logs/launchd.stdout</
string>
        <key>UserName</key><string>nobody</string>
</dict>
</plist>
```

Now that we have the configuration file, we need to place it in the correct location so `launchd` can access it. To ensure the script is executed even if no users are logged in, the script should be placed in */Library/LaunchDaemons*:

```
$ sudo cp tomcat.plist /Library/LaunchDaemons
```

Another requirement of `launchd` is that both the daemon and property list file need to be owned by the `root` user and the daemon needs to be executable. Let's ensure that the correct ownership and executable flag is set on these files:

```
$ chown root:wheel tomcat-launchd.sh
$ chmod +x tomcat-launchd.sh
$ chown root:wheel /Library/LaunchDaemons/tomcat.plist
```

Our final step in this process is to load the script into `launchd`:

```
$ sudo launchctl load /Library/LaunchDaemons/tomcat.plist
```

You can ensure your *plist* has been loaded by running the following command:

```
$ sudo launchctl list
com.apple.dashboard.advisory.fetch
com.apple.dnbobserverd
com.apple.KernelEventAgent
com.apple.mDNSResponder
com.apple.nibindd
```

```
com.apple.periodic-daily
com.apple.periodic-monthly
com.apple.periodic-weekly
com.apple.portmap
com.apple.syslogd
com.vix.cron
org.samba.nmbd
org.postfix.master
org.xinetd.xinetd
org.samba.smbd
org.apache.tomcat
```

Notice that Tomcat is now running via `launchd` (*org.apache.tomcat*) this is the `Label` you specified in the property list file above.

If for some reason it hasn't loaded, ensure that all your paths are correct, the files have the correct permissions and are otherwise accessible.

Automatic Startup on FreeBSD

If you installed the FreeBSD port of Tomcat 6, this section shows the standard way of configuring Tomcat to start at boot time and stop on a shutdown.

To enable Tomcat to start on a reboot and be shut down gracefully as part of the shutdown sequence, you need to put a controller script in the */usr/local/etc/rc.d/* directory. The controller script's filename must end in `.sh`, and it must be executable (see "man rc").

The FreeBSD port of Tomcat 6 comes with an RCng script that you can use for starting, stopping, and restarting the server. This script is */usr/local/etc/rc.d/tomcat60.sh*.

Make sure you have added this line to your */etc/rc.conf* file:

```
tomcat60_enable="YES"
```

This is what enables Tomcat 6 to start at boot time. Once you have done that and you reboot, Tomcat should start. It should also be able to shut down gracefully when you shut down your computer.

Testing Your Tomcat Installation

Once you have Tomcat installed and started, you should confirm that it has successfully started up. Open the URL *http://localhost:8080* (it's port 8180 if you're running FreeBSD and installed the FreeBSD port) in a browser to verify that you see output like that shown in Figure 1-8.

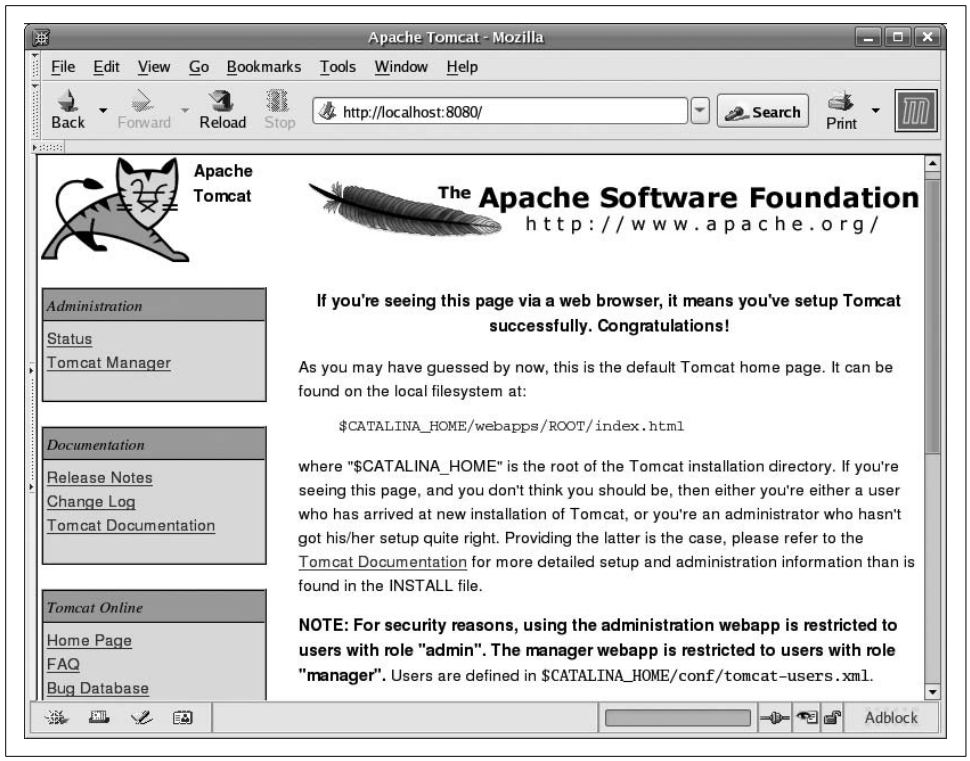


Figure 1-8. Success!

If you have changed the port number in *server.xml*, you will need to use that same port here.

Now that Tomcat is up and running, you can begin to customize its behavior, which is discussed in Chapter 2.

Where Did Tomcat Come From?

The first Java servlet container was Sun Microsystems's Java Web Server (JWS). Sun's Java Web Server was a product that Sun offered for sale. It was more affordable than most commercial server offerings, but it did not enjoy widespread commercial success—largely due to Java still being new, and servlets being only recently introduced. One of Java Web Server's main outgrowths, however, was the Java Servlet Specification as a de facto standard that Sun documented and made available separately. One big success of JWS was that it put Java servlets in the limelight.

In 1996, a plethora of free Java servlet containers became popular. Apache's JServ and CERN/W3C's Jigsaw were two of the earliest open source Java servlet containers. They were quickly followed by several more, including Jetty (*http://jetty.mortbay.org*), the Locomotive Application Server (see the web archives at *http://web.archive.org/web/*/http://www.locomotive.org*), Enhydra (*http://www.enhydra.org*), and many others. At the same time, commercial servlet containers were starting to become available as the industry embraced the Java Servlet standard; some of these were WebLogic's Tengah, ATG's Dynamo, and LiveSoftware's JRun.

In 1997, Sun released its first version of the Java Servlet Development Kit (JSDK). The JSDK was a very small servlet container that supported JSP and had a built-in HTTP 1.0 web server. In an effort to provide a reference implementation for developing servlets, Sun made it available as a free download to anyone wanting to experiment with the new Java server-side standard. It also had success as a testing and development platform in preparation for deployment to a commercial server.

In the first half of 1998, Sun announced its new JSP specification, which built upon the Java Servlet API and allowed more rapid development of dynamic web application content. After the 2.1 release of the JSDK (now called the JSWDK to add "Web" to the name), James Duncan Davidson at Sun rewrote the core of the older JSDK server. At the heart of this new Java servlet engine reference implementation was a brand new servlet container named Tomcat, and its version number started at 3.0 because it was a follow-on to version 2.1 that it replaced.

Why the Name Tomcat?

Tomcat was created when James Duncan Davidson (then an employee at Sun) wrote a new server based on the Servlet and JSP idea but without using any code from JWS.

As James put it when we asked him about this, "O'Reilly books have animals on the covers. So what animal would I want on the cover of the O'Reilly book covering the technology?

"Furthermore, I wanted the animal to be something that was self-sufficient. Able to take care of itself, even if neglected, etc. Tomcat came out of that thought."

He code-named it Tomcat, and the name was effectively obscured from view because it was the internal engine of the JSWDK, and not a product name. Until ". . . at the 4th JavaOne, somebody asked about it in the audience as they had decompiled the sources and wanted to know what com.sun.tomcat was."

As the servlet and JSP specifications' reference implementation, Tomcat evolved rapidly. As the specifications became rich with features, so did Tomcat and with it the JSWDK. For various reasons, James and Sun wanted to open the code to the JSWDK. This was largely so developers everywhere could examine how servlets and JSPs operated. Here's what Jason Hunter of the Apache Software Foundation says about what happened next:

> Sun wanted to spread the adoption of the technology, especially JSP, and Apache was a good venue to enable that. From what James said at the time and since, they wouldn't have open sourced it on their own except if Apache (with majority web server marketshare) would take the code, well then! What's funny is Sun gave it for JSP, Apache took it for servlets.

Nevertheless, the open source Tomcat project has enjoyed rapid development in areas including both servlets and JSP functionality from the developer community since its donation to the Apache Software Foundation.

Being freely distributable, backed by both Sun and the Apache Software Foundation, being the reference implementation for the Java Servlet Specification, and being all-around "cool," Tomcat went on to redefine the very meaning of a Java server, let alone a servlet container. Today, Tomcat is one of the most widely used open source software packages and is a collaborative project bustling with activity every day of the year.

While Tomcat's popularity steadily increased, Sun Microsystems moved on to develop a new reference implementation—this time for all of Java EE. The Glassfish Java EE server is the new reference implementation, and the web container component of Glassfish is based heavily on Tomcat. Meanwhile, Tomcat remains the most popular, most widely used open source servlet container implementation. All open source Java EE application server implementations include Tomcat, in whole or in part. Tomcat remains 100 percent compliant with Sun's latest specifications for servlets, JSP, and other Java EE web container specifications.

A Step-by-Step Guide to Java Persistence

Harnessing

Hibernate

James Elliott, Tim O'Brien
& Ryan Fowler

O'REILLY®

Harnessing Hibernate

CHAPTER 3

Harnessing Hibernate

All right, we've set up a whole bunch of infrastructure, defined an object/relational mapping, and used it to create a matching Java class and database table. But what does that buy us? It's time to see how easy it is to work with persistent data from your Java code.

Configuring Hibernate

Before we can continue working with Hibernate, we need to get some busy work out of the way. In the previous chapter, we configured Hibernate's JDBC connection using a *hibernate.properties* file in the *src* directory. In this chapter we introduce a way to configure the JDBC connection, SQL dialect, and much more, using a Hibernate XML Configuration file. Just like the *hibernate.properties* file, we'll place this file in the *src* directory. Enter the content shown in Example 3-1 into a file called *hibernate.cfg.xml* within *src*, and delete the *hibernate.properties* file.

Example 3-1. Configuring Hibernate using XML: hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property> ❶

    <!-- Database connection settings --> ❷
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:data/music</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">true</property>

    <!-- JDBC connection pool (use the built-in one) -->
```

```
    <property name="connection.pool_size">1</property> ❸

    <!-- Enable Hibernate's automatic session context management -->
    <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache  --> ❹
    <property
     name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <!-- disable batching so HSQLDB will propagate errors correctly. -->
    <property name="jdbc.batch_size">0</property> ❺

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property> ❻

    <!-- List all the mapping documents we're using --> ❼
    <mapping resource="com/oreilly/hh/data/Track.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

As you can see, *hibernate.cfg.xml* configures the dialect, the JDBC parameters, the connection pool, and the cache provider. This XML document also references the mapping document we wrote in the previous chapter, eliminating the need to reference the mapping documents from within our Java source. Let's go through the details of this configuration file one section at a time:

❶ Just like the *hibernate.properties* file from the previous chapter, we're defining the dialect needed to work with HSQLDB. You might notice that the `property` element's `name` attribute is `dialect`, which is similar to the name of the property in the properties file, `hibernate.dialect`. When you configure Hibernate using the XML configuration file, you are passing the same properties to Hibernate. In the XML you can omit the `hibernate.` prefix from the name of the property. This section (dialect) and the next section (connection) are the same set of properties that were configured by the *hibernate.properties* file we used in Chapter 2.

❷ The properties used to configure the JDBC connection from Hibernate—`connection.driver_class`, `connection.url`, `connection.username`, `connection.password`, and `connection.shutdown`—match the properties set in *hibernate.properties* in Example 2-4.

❸ The `connection.pool_size` property is set to a value of `1`. This means that Hibernate will create a JDBC Connection pool with only one connection. Connection pooling is important in larger applications that need to scale, but for the purposes of this book, we can safely configure Hibernate to use the built-in connection pool with a single JDBC connection. Hibernate allows for a great deal of flexibility when it comes to connection pool implementations; it is very easy to configure Hibernate to use other connection pool implementations such as Apache Commons DBCP and C3P0.

❹ As things stand, when we start telling Hibernate to perform actual persistence operations for us, it is going to warn us that we haven't properly configured its second-

level caching systems. For a simple application like this, we don't need any at all; this line turns off the second-level caching and sends every operation to the database.

❺ Here we are turning off Hibernate's JDBC batching feature. This reduces efficiency slightly—although far less so for an in-memory database like HSQLDB—but is necessary for usable error reporting in current HSQLDB releases. With batching turned on, if any statement in the batch has a problem, the only exception you get from HSQLDB is a `BatchUpdateException` telling you that the batch failed. This makes it almost impossible to debug your program. The author of HSQLDB reports that this problem will be fixed in the next major release; until then, we have to live without batching when using HSQLDB, for our sanity's sake.

❻ The `show_sql` property is a useful property to set when developing and debugging a program that uses Hibernate. Setting `show_sql` to `true` tells Hibernate to print out every statement it executes against a database. Set this property to `false` if you do not want to see the SQL statements printed out to the console.

❼ This final section lists all of the mapping documents in our project. Note that the path contains forward slashes and is relative to the *src* directory. This path addresses the *.hbm.xml* files on the class path as resources. By listing these files here, we no longer need to explicitly find them within *build.xml* targets that manipulate mapped classes (as you'll see later), nor load them from within each `main( )` method in our example source code, as we did in the previous version of the book. We think you'll agree it's nicer to keep this all in one place.

In addition to the *hibernate.cfg.xml* file in the *src* directory, you will also need to alter your *build.xml* to reference the XML configuration file. Change the `configuration` elements within the `codegen` and `schema` targets to look like the boldfaced lines in Example 3-2.

Example 3-2. Changes to build.xml to use XML configuration for Hibernate

```
...

  <!-- Generate the java code for all mapping files in our source tree -->
  <target name="codegen" depends="prepare"
          description="Generate Java source from the O/R mapping files">
    <hibernatetool destdir="${source.root}">
      <configuration configurationfile="${source.root}/hibernate.cfg.xml"/>
      <hbm2java/>
    </hibernatetool>
  </target>

...

  <!-- Generate the schemas for all mapping files in our class tree -->
  <target name="schema" depends="prepare"
          description="Generate DB schema from the O/R mapping files">

    <hibernatetool destdir="${source.root}">
      <configuration configurationfile="${source.root}/hibernate.cfg.xml"/>
```

```
      <hbm2ddl drop="yes" />
    </hibernatetool>
  </target>

...
```

These two lines tell the Hibernate Tools Ant tasks where to look for the Hibernate XML configuration, and they find all the information they need within that configuration (including the mapping documents we're working with; recall that in Chapter 2 we had to explicitly build Ant `fileset` elements inside our `configuration` elements to match all mapping files found within the project source tree). We will be using the Hibernate XML configuration throughout the remainder of this book, which makes it easier to pass all kinds of information to all the Hibernate-related tools.

Now that we've successfully configured Hibernate, let's return to our main task for the chapter: making some persistent objects.

Creating Persistent Objects

Let's start by creating some new `Track` instances and persisting them to the database, so we can see how they turn into rows and columns for us. Because of the way we've organized our mapping document and configuration file, it's extremely easy to configure the Hibernate session factory and get things rolling.

How do I do that?

This discussion assumes you've created the schema and generated Java code by following the preceding examples. If you haven't, you can start by downloading the examples archive from this book's web site[*], jumping into the *ch03* directory, and using the commands *ant prepare* and *ant codegen*[†] followed by *ant schema* to automatically fetch the Hibernate and HSQLDB libraries and set up the generated Java code and database schema on which this example is based. (As with the other examples, these commands should be issued in a shell whose current working directory is the top of your project tree, containing Ant's *build.xml* file.)

We'll start with a simple example class, `CreateTest`, containing the necessary imports and housekeeping code to bring up the Hibernate environment and create some `Track` instances that can be persisted using the XML mapping document with which we started. Type the source of Example 3-3 in the directory *src/com/oreilly/hh*.

[*] *http://www.oreilly.com/catalog/9780596517724/*

[†] Even though the `codegen` target depends on the `prepare` target, the very first time you're working with one of the example directories you need to create the proper classpath structure for Ant to be happy by running `prepare` explicitly first, as discussed in "Cooking Up a Schema" back in Chapter 2.

Example 3-3. Data creation test, CreateTest.java

```
package com.oreilly.hh;

import org.hibernate.*; ❶
import org.hibernate.cfg.Configuration;

import com.oreilly.hh.data.*;

import java.sql.Time;
import java.util.Date;

/**
 * Create sample data, letting Hibernate persist it for us.
 */
public class CreateTest {

    public static void main(String args[]) throws Exception {
        // Create a configuration based on the XML file we've put
        // in the standard place.
        Configuration config = new Configuration(); ❷
        config.configure();

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory(); ❸

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession(); ❹
        Transaction tx = null;
        try {
            // Create some data and persist it
            tx = session.beginTransaction(); ❺

            Track track = new Track("Russian Trance",
                                    "vol2/album610/track02.mp3",
                                    Time.valueOf("00:03:30"), new Date(),
                                    (short)0);
            session.save(track);

            track = new Track("Video Killed the Radio Star",
                              "vol2/album611/track12.mp3",
                              Time.valueOf("00:03:49"), new Date(),
                              (short)0);
            session.save(track);


             track = new Track("Gravity's Angel",
                              "vol2/album175/track03.mp3",
                               Time.valueOf("00:06:06"), new Date(),
                               (short)0);
            session.save(track);

            // We're done; make our changes permanent
            tx.commit(); ❻

        } catch (Exception e) {
```

```
            if (tx != null) {
                // Something went wrong; discard all partial changes
                tx.rollback();
            }
            throw new Exception("Transaction failed", e);
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close(); ❼
    }
}
```

The first part of *CreateTest.java* needs a little explanation:

❶ We import some useful Hibernate classes, including `Configuration`, which is used to set up the Hibernate environment. We also want to import any and all data classes that Hibernate has generated for us based on our mapping documents; these will all be found in our `data` package. The `Time` and `Date` classes are used in our data objects to represent track playing times and creation timestamps. The only method we implement in `CreateTest` is the `main( )` method that supports invocation from the command line.

❷ When this class is run, it starts by creating a Hibernate `Configuration` object. Since we don't tell it otherwise, Hibernate looks for a file named *hibernate.cfg.xml* at the root level in the classpath. It finds the one we created earlier, which tells it we're using HSQLDB and how to find the database. This XML configuration file also references the Hibernate Mapping XML document for the `Track` object. Calling `config.configure( )` automatically adds the mapping for the `Track` class.

❸ That's all the configuration we need in order to create and persist track data, so we're ready to create the `SessionFactory`. Its purpose is to provide us with `Session` objects, the main avenue for interaction with Hibernate. The `SessionFactory` is thread-safe, and you need only one for your entire application. (To be more precise, you need one for each database environment for which you want persistence services; most applications therefore need only one.) Creating the session factory is a pretty expensive and slow operation, so you'll definitely want to share it throughout your application. It's trivial in a one-class application like this one, but the reference documentation provides some good examples of ways to do it in more realistic scenarios.

❹ When it comes time to actually perform persistence, we ask the `SessionFactory` to open a `Session` for us, which establishes a JDBC connection to the database and provides us with a context in which we can create, obtain, manipulate, and delete persistent objects. As long as the session is open, a connection to the database is maintained, and changes to the persistent objects associated with the session are tracked so they can be applied to the database when the session is closed. Conceptually, you can think of a session as a "large-scale transaction" between the persistent objects and the database, which may encompass several database-level transactions. As with a database transaction, though, you should not think about keeping your Hibernate session open over long periods of application existence (such as while you're waiting for user input). A single session is used for a specific and bounded operation in the application, something like populating the user interface or making a change that has been committed by the user. The next operation will use a new session. Also note that `Session` objects themselves are not thread-safe, so they cannot be shared between threads. Each thread needs to obtain its own session from the factory.

It's worth getting a solid understanding of the purposes and lifecycles of these objects. This book gives you just enough information to get started; you'll want to spend some time with the reference documentation and understand the examples in depth.

It's worth going into a bit more depth about the lifecycle of mapped objects in Hibernate and how this relates to sessions because the terminology is rather specific and the concepts are quite important. A mapped object, such as an instance of our `Track` class, moves back and forth between two states with respect to Hibernate: *transient* and *persistent*. An object that is transient is not associated with any session. When you first create a `Track` instance using `new( )`, it is transient; unless you tell Hibernate to persist it, the object will vanish when your application terminates.

Passing a transient mapped object to a `Session`'s `save( )` method causes it to become persistent. It will survive past its scope in the Java VM, until it is explicitly deleted later. If you've got a persistent object and you call `Session`'s `delete( )` method on it, the object transitions back to transient state. The object still exists as an instance in your application, but it is no longer going to stay around unless you change your mind and save it again. On the other hand, if you haven't deleted it (so it's still persistent) and you make changes to the object, there's no need to save it again in order for those changes to be reflected. Hibernate automatically tracks changes to any persistent objects and flushes those changes to the database at appropriate times. When you close the session, any pending changes are flushed.

Hang in there, we'll be back to the example soon!

An important but subtle point concerns the status of persistent objects with which you worked in a session that has been closed, such as after you run a query to find all entities matching some criteria (you'll see how to do this in "Finding Persistent Objects" later in this chapter). As noted earlier, you don't want to keep this session around longer than necessary to perform the database operation, so you close it once your queries are finished. What's the deal with the mapped objects you've loaded at this point? Well, they were persistent while the session was around, but once they are no longer associated with an active session (in this case, because the session has been closed), they are not persistent any longer. Now, this doesn't mean that they no longer exist in the database; indeed, if you run the query again (assuming nobody has changed the data in the meantime), you'll get back the same set of objects. It simply means that there is not currently an active correspondence being maintained between the state of the objects in your virtual machine and the database; they are *detached*. It is perfectly reasonable to carry on working with the objects. If you later need to make changes to the objects and you want the changes to stick, you will open a new session and use it to save the changed objects. Because each entity has a unique ID, Hibernate has no problem figuring out how to link the transient objects back to the appropriate persistent state in the new session.

Of course, as with any environment in which you're making changes to an offline copy of information backed by a database, you need to think about application-level data-integrity constraints. You may need to devise some higher-level locking or versioning protocol to support them. Hibernate can offer help with this task, too, but the design and detailed implementation is up to you. The reference manual does strongly recommend the use of a version field, and there are several approaches available.

❺ Armed with these concepts and terms, the remainder of the example is easy enough to understand. We set up a database transaction using our open session. Within that, we create a few `Track` instances containing sample data and save them in the session, turning them from transient instances into persistent entities.

❻ Finally, we commit our transaction, atomically (as a single, indivisible unit) making all the database changes permanent. The `try`/`catch`/`finally` block wrapped around all this shows an important and useful idiom for working with transactions. If anything goes wrong, the `catch` block will roll back the transaction and then bubble out the exception, leaving the database the way we found it. The session is closed in the `finally` portion, ensuring that this takes place whether we exit through the "happy path" of a successful commit or via an exception that caused rollback. Either way, it gets closed as it should.

❼ At the end of our method we also close the session factory itself. This is something you'd do in the "graceful shutdown" section of your application. In a web applica-

tion environment, it would be in the appropriate lifecycle event handler. In this simple example, when the `main( )` method returns, the application is ending.

With all we've got in place, by now it's quite easy to tell Ant how to compile and run this test. Add the targets shown in Example 3-4 right before the closing `</project>` tag at the end of *build.xml*.

Example 3-4. Ant targets to compile all Java source and invoke data creation test

```
<!-- Compile the java source of the project -->
<target name="compile" depends="prepare" ❶
        description="Compiles all Java classes">
  <javac srcdir="${source.root}"
         destdir="${class.root}"
         debug="on"
         optimize="off"
         deprecation="on">
    <classpath refid="project.class.path"/>
  </javac>
</target>

<target name="ctest" description="Creates and persists some sample data"
        depends="compile"> ❷
  <java classname="com.oreilly.hh.CreateTest" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

❶ The aptly named `compile` target uses the built-in `javac` task to compile all the Java source files found in the *src* tree to the *classes* tree. Happily, this task also supports the project class path we've set up, so the compiler can find all the libraries we're using. The `depends=prepare` attribute in the target definition tells Ant that before running the `compile` target, `prepare` must be run. Ant manages dependencies so that when you're building multiple targets with related dependencies, they are executed in the right order, and each dependency gets executed only once, even if it is mentioned by multiple targets.

If you're accustomed to using shell scripts to compile a lot of Java source, you'll be surprised by how quickly the compilation happens. Ant invokes the Java compiler within the same virtual machine that it is using, so there is no process startup delay for each compilation.

❷ The `ctest` target uses `compile` to make sure the class files are built and then creates a new Java virtual machine to run our `CreateTest` class.

All right, we're ready to create some data! Example 3-5 shows the results of invoking the new `ctest` target. Its dependency on the `compile` target ensures the `CreateTest` class gets compiled before we try to use it. The output for `ctest` itself shows the logging emitted by Hibernate as the environment and mappings are set up and the connection is shut back down.

Example 3-5. Invoking the CreateTest class

```
% ant ctest
prepare:

compile:
    [javac] Compiling 2 source files to /Users/jim/svn/oreilly/hib_dev_2e/curren
t/examples/ch03/classes

ctest:
     [java] 00:21:45,833  INFO Environment:514 - Hibernate 3.2.5
     [java] 00:21:45,852  INFO Environment:547 - hibernate.properties not found
     [java] 00:21:45,864  INFO Environment:681 - Bytecode provider name : cglib
     [java] 00:21:45,875  INFO Environment:598 - using JDK 1.4 java.sql.Timestam
p handling
     [java] 00:21:46,032  INFO Configuration:1426 - configuring from resource: /
hibernate.cfg.xml
     [java] 00:21:46,034  INFO Configuration:1403 - Configuration resource: /hib
ernate.cfg.xml
     [java] 00:21:46,302  INFO Configuration:553 - Reading mappings from resourc
e : com/oreilly/hh/data/Track.hbm.xml
     [java] 00:21:46,605  INFO HbmBinder:300 - Mapping class: com.oreilly.hh.dat
a.Track -> TRACK
     [java] 00:21:46,678  INFO Configuration:1541 - Configured SessionFactory: n
ull
     [java] 00:21:46,860  INFO DriverManagerConnectionProvider:41 - Using Hibern
ate built-in connection pool (not for production use!)
     [java] 00:21:46,862  INFO DriverManagerConnectionProvider:42 - Hibernate co
nnection pool size: 1
     [java] 00:21:46,864  INFO DriverManagerConnectionProvider:45 - autocommit m
ode: false
     [java] 00:21:46,879  INFO DriverManagerConnectionProvider:80 - using driver
: org.hsqldb.jdbcDriver at URL: jdbc:hsqldb:data/music
     [java] 00:21:46,891  INFO DriverManagerConnectionProvider:86 - connection p
roperties: {user=sa, password=****, shutdown=true}
     [java] 00:21:47,533  INFO SettingsFactory:89 - RDBMS: HSQL Database Engine,
 version: 1.8.0
     [java] 00:21:47,538  INFO SettingsFactory:90 - JDBC driver: HSQL Database E
ngine Driver, version: 1.8.0
     [java] 00:21:47,613  INFO Dialect:152 - Using dialect: org.hibernate.dialec
t.HSQLDialect
     [java] 00:21:47,638  INFO TransactionFactoryFactory:31 - Using default tran
saction strategy (direct JDBC transactions)
     [java] 00:21:47,646  INFO TransactionManagerLookupFactory:33 - No Transacti
onManagerLookup configured (in JTA environment, use of read-write or transaction
al second-level cache is not recommended)
     [java] 00:21:47,649  INFO SettingsFactory:143 - Automatic flush during befo
reCompletion(): disabled
     [java] 00:21:47,650  INFO SettingsFactory:147 - Automatic session close at
end of transaction: disabled
     [java] 00:21:47,657  INFO SettingsFactory:154 - JDBC batch size: 15
     [java] 00:21:47,659  INFO SettingsFactory:157 - JDBC batch updates for vers
ioned data: disabled
     [java] 00:21:47,664  INFO SettingsFactory:162 - Scrollable result sets: ena
bled
     [java] 00:21:47,666  INFO SettingsFactory:170 - JDBC3 getGeneratedKeys(): d
```

```
isabled
     [java] 00:21:47,668  INFO SettingsFactory:178 - Connection release mode: au
to
     [java] 00:21:47,671  INFO SettingsFactory:205 - Default batch fetch size: 1
     [java] 00:21:47,678  INFO SettingsFactory:209 - Generate SQL with comments:
 disabled
     [java] 00:21:47,680  INFO SettingsFactory:213 - Order SQL updates by primar
y key: disabled
     [java] 00:21:47,681  INFO SettingsFactory:217 - Order SQL inserts for batch
ing: disabled
     [java] 00:21:47,684  INFO SettingsFactory:386 - Query translator: org.hiber
nate.hql.ast.ASTQueryTranslatorFactory
     [java] 00:21:47,690  INFO ASTQueryTranslatorFactory:24 - Using ASTQueryTran
slatorFactory
     [java] 00:21:47,694  INFO SettingsFactory:225 - Query language substitution
s: {}
     [java] 00:21:47,695  INFO SettingsFactory:230 - JPA-QL strict compliance: d
isabled
     [java] 00:21:47,702  INFO SettingsFactory:235 - Second-level cache: enabled
     [java] 00:21:47,704  INFO SettingsFactory:239 - Query cache: disabled
     [java] 00:21:47,706  INFO SettingsFactory:373 - Cache provider: org.hiberna
te.cache.NoCacheProvider
     [java] 00:21:47,707  INFO SettingsFactory:254 - Optimize cache for minimal
puts: disabled
     [java] 00:21:47,709  INFO SettingsFactory:263 - Structured second-level cac
he entries: disabled
     [java] 00:21:47,724  INFO SettingsFactory:283 - Echoing all SQL to stdout
     [java] 00:21:47,731  INFO SettingsFactory:290 - Statistics: disabled
     [java] 00:21:47,732  INFO SettingsFactory:294 - Deleted entity synthetic id
entifier rollback: disabled
     [java] 00:21:47,734  INFO SettingsFactory:309 - Default entity-mode: pojo
     [java] 00:21:47,735  INFO SettingsFactory:313 - Named query checking : enab
led
     [java] 00:21:47,838  INFO SessionFactoryImpl:161 - building session factory
     [java] 00:21:48,464  INFO SessionFactoryObjectFactory:82 - Not binding fact
ory to JNDI, no JNDI name configured
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
     [java] 00:21:49,365  INFO SessionFactoryImpl:769 - closing
     [java] 00:21:49,369  INFO DriverManagerConnectionProvider:147 - cleaning up
 connection pool: jdbc:hsqldb:data/music

BUILD SUCCESSFUL
Total time: 2 seconds
```

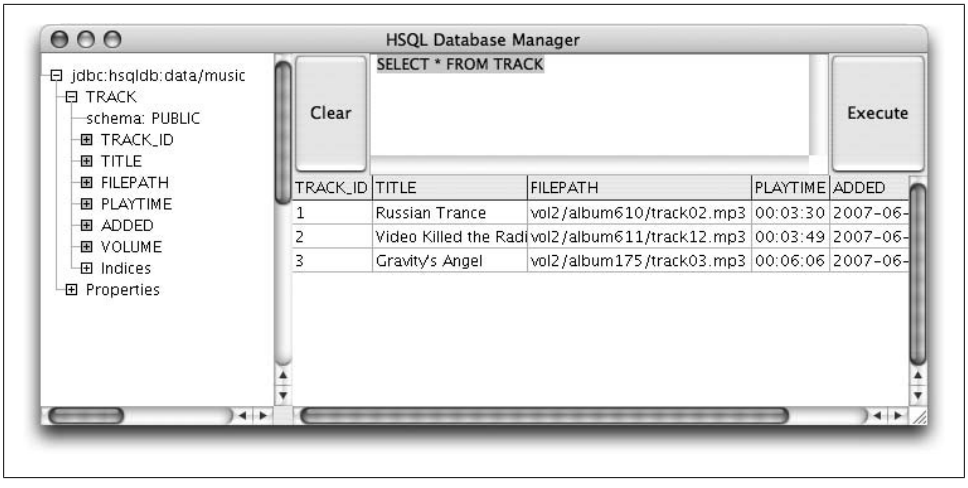


Figure 3-1. Test data persisted into the TRACK table

What just happened?

If you scan through all the messages Hibernate prints out because we've turned on "info" logging, you can see that our test class fired up Hibernate, loaded the mapping information for the `Track` class, opened a persistence session to the associated HSQLDB database, and used that to create some instances and persist them in the `TRACK` table. Then it shut down the session and closed the database connection, ensuring the data was saved.

After running this test, you can use *ant db* to take a look at the contents of the database. You should find three rows in the `TRACK` table now, as shown in Figure 3-1. (Type your query in the text box at the top of the window and click the Execute button. You can get a command skeleton and syntax documentation by choosing Command→Select in the menu bar.)

At this point, it's worth pausing a moment to reflect on the fact that we wrote no code to connect to the database or issue SQL commands. Looking back to the preceding sections, we didn't even have to create the table ourselves, nor the `Track` object that encapsulates our data. Yet the query output in Figure 3-1 shows nicely readable data representing the Java objects created and persisted by our short, simple test program. Hopefully you'll agree that this reflects very well on the power and convenience of Hibernate as a persistence service. For being free and lightweight, Hibernate can certainly do a lot for you, quickly and easily.

If you have been using JDBC directly, especially if you're pretty new to it, you may be used to relying on the "auto-commit" mode in the database driver rather than always using database transactions. Hibernate is rightly opinionated that this is the wrong way to structure application code; the only place it makes sense is in database console experimentation by humans. So, you'll always need to use transactions around your persistence operations in Hibernate.

As noted in Chapter 1, you can also look directly at the SQL statements creating your data in the *music.script* file in the *data* directory as shown in Example 3-6.

Example 3-6. Looking at the raw database script file

```
% cat data/music.script
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE MEMORY TABLE TRACK(TRACK_ID INTEGER GENERATED BY DEFAULT AS IDENTITY(STAR
T WITH 1) NOT NULL PRIMARY KEY,TITLE VARCHAR(255) NOT NULL,FILEPATH VARCHAR(255)
 NOT NULL,PLAYTIME TIME,ADDED DATE,VOLUME SMALLINT NOT NULL)
ALTER TABLE TRACK ALTER COLUMN TRACK_ID RESTART WITH 4
CREATE USER SA PASSWORD ""
GRANT DBA TO SA
SET WRITE_DELAY 10
SET SCHEMA PUBLIC
INSERT INTO TRACK VALUES(1,'Russian Trance','vol2/album610/track02.mp3','00:03:3
0','2007-06-17',0)
INSERT INTO TRACK VALUES(2,'Video Killed the Radio Star','vol2/album611/track12.
mp3','00:03:49','2007-06-17',0)
INSERT INTO TRACK VALUES(3,'Gravity''s Angel','vol2/album175/track03.mp3','00:06
:06','2007-06-17',0)
```

The final three statements show our `TRACK` table rows. The top contains the schema and the user that gets provided by default when creating a new database. (Of course, in a real application environment, you'd want to change these credentials, unless you were only enabling in-memory access.)

Tempted to learn more about HSQLDB? We won't try to stop you!

What about…

...objects with relationships to other objects? Collections of objects? You're right—these are cases where persistence gets more challenging (and, if done right, valuable). Hibernate can handle associations like this just fine. In fact, there isn't any special effort involved on our part. We'll discuss this in Chapter 4. For now, let's look at how to retrieve objects that were persisted in earlier sessions.

Finding Persistent Objects

It's time to throw the giant lever into reverse and look at how you load data from a database into Java objects.

Let's see how it works, using Hibernate Query Language to get an object-oriented view of the contents of mapped database tables. These might have started out as objects persisted in a previous session or might be data that came from completely outside your application code.

How do I do that?

Example 3-7 shows a program that runs a simple query using the test data we just created. The overall structure will look very familiar because all the Hibernate setup is the same as in the previous program.

Example 3-7. Data retrieval test, QueryTest.java

```
package com.oreilly.hh;

import org.hibernate.*;
import org.hibernate.cfg.Configuration;

import com.oreilly.hh.data.*;

import java.sql.Time;
import java.util.*;

/**
 * Retrieve data as objects
 */
public class QueryTest {

    /**
     * Retrieve any tracks that fit in the specified amount of time.
     *
     * @param length the maximum playing time for tracks to be returned.
     * @param session the Hibernate session that can retrieve data.
     * @return a list of {@link Track}s meeting the length restriction.
     */
    public static List tracksNoLongerThan(Time length, Session session) { ❶
        Query query = session.createQuery("from Track as track " +
                                          "where track.playTime <= ?");
        query.setParameter(0, length, Hibernate.TIME);
        return query.list();
    }

    /**
     * Look up and print some tracks when invoked from the command line.
     */
    public static void main(String args[]) throws Exception {
        // Create a configuration based on the properties file we've put
```

```
        // in the standard place.
        Configuration config = new Configuration();
        config.configure();

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory();

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession();
        try { ❷
            // Print the tracks that will fit in five minutes
            List tracks = tracksNoLongerThan(Time.valueOf("00:05:00"),
                                             session);
            for (ListIterator iter = tracks.listIterator() ;
                 iter.hasNext() ; ) {
                Track aTrack = (Track)iter.next();
                System.out.println("Track: \"" + aTrack.getTitle() +
                                   "\", " + aTrack.getPlayTime());
            }
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close();
    }
}
```

Once again, we add a target, shown in Example 3-8, at the end of *build.xml* (right before the closing `project` tag) to run this test.

Example 3-8. Ant target to invoke our query test

```
  <target name="qtest" description="Run a simple Hibernate query"
          depends="compile">
    <java classname="com.oreilly.hh.QueryTest" fork="true">
      <classpath refid="project.class.path"/>
    </java>
  </target>
```

With this in place, we can simply type *ant qtest* to retrieve and display some data, with the results shown in Example 3-9. To save space in the output, we've edited *log4j.properties* to turn off all the "info" messages, since they're no different than in the previous example. You can do this yourself by changing the line:

```
log4j.logger.org.hibernate=info
```

to replace the word `info` with `warn`:

```
log4j.logger.org.hibernate=warn
```

Example 3-9. Running the query test

```
% ant qtest
Buildfile: build.xml
```

```
prepare:
     [copy] Copying 1 file to /Users/jim/svn/oreilly/hib_dev_2e/current/examples
/ch03/classes

compile:
    [javac] Compiling 1 source file to /Users/jim/svn/oreilly/hib_dev_2e/current
/examples/ch03/classes

qtest:
     [java] Hibernate: select track0_.TRACK_ID as TRACK1_0_, track0_.title as ti
tle0_, track0_.filePath as filePath0_, track0_.playTime as playTime0_, track0_.a
dded as added0_, track0_.volume as volume0_ from TRACK track0_ where track0_.pla
yTime<=?
     [java] Track: "Russian Trance", 00:03:30
     [java] Track: "Video Killed the Radio Star", 00:03:49

BUILD SUCCESSFUL
Total time: 2 seconds
```

What just happened?

❶ We started out by defining a utility method, `tracksNoLongerThan( )`, which performs the actual Hibernate query. It retrieves any tracks whose playing time is less than or equal to the amount specified as a parameter. Notice that HQL, Hibernate's SQL-inspired query language, supports parameter placeholders, much like `PreparedStatement` in JDBC. And, just like in that environment, using them is preferable to putting together queries through string manipulation (especially since this protects you from SQL injection attacks). As you'll see, however, Hibernate offers even better ways of working with queries in Java.

The query itself looks a little strange. It starts with `from` rather than `select` *`something`*, as you might expect. While you can certainly use the more familiar format, and will do so when you want to pull out individual properties from an object in your query, if you want to retrieve entire objects, you can use this more abbreviated syntax.

Also note that the query is expressed in terms of the mapped Java *objects* and *properties*, rather than the tables and columns. It's not obvious in this case, since the object and table have the same name, as do the property and column, but it is true. Keeping the names consistent is a fairly natural choice and will always be the case when you're using Hibernate to generate the schema and the data objects, unless you tell it explicitly to use different column names.

When you're working with preexisting databases and objects, it's important to realize that HQL queries refer to object properties rather than to database table columns.

Also, just as in SQL, you can alias a column or table to another name. In HQL, you alias classes in order to select or constrain their properties. This won't come up in this simple introduction, but if you dig into the resources mentioned in Appendix E, you'll encounter it.

❷ The rest of the program should look mighty familiar from the previous example. Our `try` block is simplified because we don't need explicit access to our transaction, as we're not changing any data. We still use `try` so that we can have a `finally` clause to close our session cleanly. The body is quite simple, calling our query method to request any tracks whose playing time is five minutes or less, and then iterating over the resulting `Track` objects, printing their titles and playing times.

Also, now that we've turned off "info" level logging from within Hibernate, the SQL debugging output we configured in *hibernate.cfg.xml* is much easier to spot—the first line in the `"qtest:"` section of the output is not something we wrote ourselves in *QueryTest.java*; it's Hibernate showing us the SQL it generated to implement the HQL query we requested. Interesting... and if you ever get sick of seeing it, remember that you can set the `show_sql` property value to `false`.

What about…

...deleting objects? If you've made changes to your data-creation script and want to start with a "clean slate" in the form of an empty database so you can test them, all you need to do is run *ant schema* again. This will drop and recreate the `Track` table in a pristine and empty state. Don't do it unless you mean it!

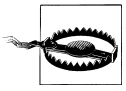
If you want to be more selective about what you delete, you can either do it through SQL commands in the HSQLDB UI (*ant db*), or you can make a variant of the query-test example that retrieves the objects you want to get rid of. Once you've got a reference to a persistent object, passing it to the `Session`'s `delete( )` method will remove it from the database:

```
session.delete(aTrack);
```

You've still got at least one reference to it in your program, until `aTrack` goes out of scope or gets reassigned, so conceptually the easiest way to understand what `delete( )` does is to think of it as turning a persistent object back into a transient one.

Another way to delete is to write an HQL deletion query that matches multiple objects. This lets you delete many persistent objects at once, whether or not you have them as objects in memory, without writing your own loop. A Java-based alternative to *ant schema*, and a slightly less violent way of clearing out all the tracks, would therefore be something like this:

```
Query query = session.createQuery("delete from Track");
query.executeUpdate();
```



Don't forget that regardless of which of these approaches you use, you'll need to wrap the data-manipulation code inside a Hibernate transaction and commit the transaction if you want your changes to stick.

Better Ways to Build Queries

As mentioned earlier in this chapter, HQL lets you go beyond the use of JDBC-style query placeholders to get parameters conveniently into your queries. You can use *named parameters* and *named queries* to make your programs much easier to read and maintain.

Why do I care?

Named parameters make code easier to understand because the purpose of the parameter is clear both within the query itself and within the Java code that is setting it up. This self-documenting nature is valuable in itself, but it also reduces the potential for error by freeing you from counting commas and question marks, and it can modestly improve efficiency by letting you use the same parameter more than once in a single query.

Named queries let you move the queries completely out of the Java code. Keeping queries out of your Java source makes them much easier to read and edit because they aren't giant concatenated series of Java strings spread across multiple lines and interwoven with extraneous quotation marks, backslashes, and other Java punctuation. Typing them the first time is bad enough, but if you've ever had to perform significant surgery on a query embedded in a program in this way, you will have had your fill of moving quotation marks and plus signs around to try to get the lines to break in nice places again.

How do I do that?

The key to both of these capabilities in Hibernate is the `Query` interface. We already began using this interface in Example 3-7 because, starting with Hibernate 3, it is the only nondeprecated way of performing queries. So today it's even less of an effort than it used to be to use the nice features described in this section.

We'll start by changing our query to use a named parameter, as shown in Example 3-10. (This isn't nearly as big a deal for a query with a single parameter like this one, but it's worth getting into the habit right away. You'll be very thankful when you start working with the light-dimming queries that power your real projects!)

Example 3-10. Revising our query to use a named parameter

```
public static List tracksNoLongerThan(Time length, Session session) {
    Query query = session.createQuery("from Track as track " +
                                      "where track.playTime <= :length");
```

```
    query.setTime("length", length);
    return query.list();
}
```

Named parameters are identified within the query body by prefixing them with a colon. Here, we've changed the `?` to `:length`. The `Session` object provides a `createQuery( )` method that gives us back an implementation of the `Query` interface with which we can work. `Query` has a full complement of type-safe methods for setting the values of named parameters. Here we are passing in a `Time` value, so we use `setTime( )`. Even in a simple case like this, the syntax is more natural and readable than the original version of our query. If we had been passing in anonymous arrays of values and types (as would have been necessary with more than one parameter), the improvement would be even more significant. And we've added a layer of compile-time type checking, always a welcome change.

Running this version produces exactly the same output as our original program.

So, how do we get the query text out of the Java source? Again, this query is short enough that the need to do so isn't as pressing as usual in real projects, but it's the best way to do things, so let's start practicing! As you may have predicted, the place we can store queries is inside the mapping document. Example 3-11 shows what this looks like. We have to use the somewhat clunky `CDATA` construct, since our query contains characters (like `<`) that could otherwise confuse the XML parser.

Example 3-11. Our query in the mapping document

```
  <query name="com.oreilly.hh.tracksNoLongerThan">
    <![CDATA[
        from Track as track
        where track.playTime <= :length
      ]]>
  </query>
```

Put this just after the closing tag of the class definition in *Track.hbm.xml* (right before the `</hibernate-mapping>` line). Then we can revise *QueryTest.java* one last time, as shown in Example 3-12. Once again, the program produces exactly the same output as the initial version. It's just better organized now, and we're in great shape if we ever want to make the query more complex.

Example 3-12. Final version of our query method

```
public static List tracksNoLongerThan(Time length, Session session) {
    Query query = session.getNamedQuery(
                      "com.oreilly.hh.tracksNoLongerThan");
    query.setTime("length", length);
    return query.list();
}
```

The `Query` interface has other useful capabilities beyond what we've examined here. You can use it to control how many rows (and which specific rows) you retrieve. If your

JDBC driver supports scrollable `ResultSet`s, you can access this capability as well. Check the JavaDoc or the Hibernate reference manual for more details.

What about…

...avoiding a SQL-like language altogether? Or diving into HQL and exploring more complex queries? These are both options that are covered later in this book.

Chapter 8 discusses criteria queries, an interesting mechanism that lets you express the constraints on the entities you want, using a natural Java API. This lets you build Java objects to represent the data you want to find, which is easier for people who aren't database experts to understand; lets you leverage your IDE's code completion as a memory aid; and even gives you compile-time checking of your syntax. It also supports a form of "query by example," where you can supply objects that are similar to the ones you're searching for, which is particularly handy for implementing search interfaces in an application.

SQL veterans who'd like to see more tricks with HQL can jump to Chapter 9, which explores more of its capabilities and unique features.

For now, we'll continue our exploration of mapping by looking at how to cope with objects that are linked to each other, which you will need in any nontrivial program.

Master SQL Fundamentals

2nd Edition

Learning SQL

O'REILLY®

Alan Beaulieu

SECOND EDITION

Learning SQL

Alan Beaulieu

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

CHAPTER 2

Creating and Populating a Database

This chapter provides you with the information you need to create your first database and to create the tables and associated data used for the examples in this book. You will also learn about various data types and see how to create tables using them. Because the examples in this book are executed against a MySQL database, this chapter is somewhat skewed toward MySQL's features and syntax, but most concepts are applicable to any server.

Creating a MySQL Database

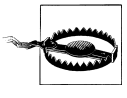
If you already have a MySQL database server available for your use, you can skip the installation instructions and start with the instructions in Table 2-1. Keep in mind, however, that this book assumes that you are using MySQL version 6.0 or later, so you may want to consider upgrading your server or installing another server if you are using an earlier release.

The following instructions show you the minimum steps required to install a MySQL 6.0 server on a Windows computer:

1. Go to the download page for the MySQL Database Server at *http://dev.mysql.com/downloads*. If you are loading version 6.0, the full URL is *http://dev.mysql.com/downloads/mysql/6.0.html*.
2. Download the Windows Essentials (x86) package, which includes only the commonly used tools.
3. When asked "Do you want to run or save this file?" click Run.
4. The MySQL Server 6.0—Setup Wizard window appears. Click Next.
5. Activate the Typical Install radio button, and click Next.
6. Click Install.
7. A MySQL Enterprise window appears. Click Next twice.

8. When the installation is complete, make sure the box is checked next to "Configure the MySQL Server now," and then click Finish. This launches the Configuration Wizard.
9. When the Configuration Wizard launches, activate the Standard Configuration radio button, and then select both the "Install as Windows Service" and "Include Bin Directory in Windows Path" checkboxes. Click Next.
10. Select the Modify Security Settings checkbox and enter a password for the `root` user (make sure you write down the password, because you will need it shortly!), and click Next.
11. Click Execute.

At this point, if all went well, the MySQL server is installed and running. If not, I suggest you uninstall the server and read the "Troubleshooting a MySQL Installation Under Windows" guide (which you can find at *http://dev.mysql.com/doc/refman/6.0/en/windows-troubleshooting.html*).



If you uninstalled an older version of MySQL before loading version 6.0, you may have some further cleanup to do (I had to clean out some old Registry entries) before you can get the Configuration Wizard to run successfully.

Next, you will need to open a Windows command window, launch the `mysql` tool, and create your database and database user. Table 2-1 describes the necessary steps. In step 5, feel free to choose your own password for the `lrngsql` user rather than "xyz" (but don't forget to write it down!).

Table 2-1. Creating the sample database

Step	Description	Action
1	Open the Run dialog box from the Start menu	Choose Start and then Run
2	Launch a command window	Type **`cmd`** and click OK
3	Log in to MySQL as `root`	`mysql -u root -p`
4	Create a database for the sample data	`create database bank;`
5	Create the `lrngsql` database user with full privileges on the `bank` database	`grant all privileges on bank.* to 'lrngsql'@'localhost' identified by 'xyz';`
6	Exit the `mysql` tool	`quit;`
7	Log in to MySQL as `lrngsql`	`mysql -u lrngsql -p;`
8	Attach to the `bank` database	`use bank;`

You now have a MySQL server, a database, and a database user; the only thing left to do is create the database tables and populate them with sample data. To do so, download the script at *http://examples.oreilly.com/learningsql/* and run it from the `mysql` utility. If you saved the file as *c:\temp\LearningSQLExample.sql*, you would need to do the following:

1. If you have logged out of the `mysql` tool, repeat steps 7 and 8 from Table 2-1.
2. Type **`source c:\temp\LearningSQLExample.sql;`** and press Enter.

You should now have a working database populated with all the data needed for the examples in this book.

Using the mysql Command-Line Tool

Whenever you invoke the `mysql` command-line tool, you can specify the username and database to use, as in the following:

```
mysql -u lrngsql -p bank
```

This will save you from having to type `use bank;` every time you start up the tool. You will be asked for your password, and then the `mysql>` prompt will appear, via which you will be able to issue SQL statements and view the results. For example, if you want to know the current date and time, you could issue the following query:

```
mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2008-02-19 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

The `now()` function is a built-in MySQL function that returns the current date and time. As you can see, the `mysql` command-line tool formats the results of your queries within a rectangle bounded by +, -, and | characters. After the results have been exhausted (in this case, there is only a single row of results), the `mysql` command-line tool shows how many rows were returned and how long the SQL statement took to execute.

About Missing from Clauses

With some database servers, you won't be able to issue a query without a `from` clause that names at least one table. Oracle Database is a commonly used server for which this is true. For cases when you only need to call a function, Oracle provides a table called `dual`, which consists of a single column called `dummy` that contains a single row of data. In order to be compatible with Oracle Database, MySQL also provides a `dual` table. The previous query to determine the current date and time could therefore be written as:

```
mysql> SELECT now()
       FROM dual;
+---------------------+
| now()               |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

If you are not using Oracle and have no need to be compatible with Oracle, you can ignore the `dual` table altogether and use just a `select` clause without a `from` clause.

When you are done with the `mysql` command-line tool, simply type **`quit;`** or **`exit;`** to return to the Windows command shell.

MySQL Data Types

In general, all the popular database servers have the capacity to store the same types of data, such as strings, dates, and numbers. Where they typically differ is in the specialty data types, such as XML documents or very large text or binary documents. Since this is an introductory book on SQL, and since 98% of the columns you encounter will be simple data types, this book covers only the character, date, and numeric data types.

Character Data

Character data can be stored as either fixed-length or variable-length strings; the difference is that fixed-length strings are right-padded with spaces and always consume the same number of bytes, and variable-length strings are not right-padded with spaces and don't always consume the same number of bytes. When defining a character column, you must specify the maximum size of any string to be stored in the column. For example, if you want to store strings up to 20 characters in length, you could use either of the following definitions:

```
char(20)    /* fixed-length */
varchar(20) /* variable-length */
```

The maximum length for `char` columns is currently 255 bytes, whereas `varchar` columns can be up to 65,535 bytes. If you need to store longer strings (such as emails, XML

documents, etc.), then you will want to use one of the text types (`mediumtext` and `longtext`), which I cover later in this section. In general, you should use the `char` type when all strings to be stored in the column are of the same length, such as state abbreviations, and the `varchar` type when strings to be stored in the column are of varying lengths. Both `char` and `varchar` are used in a similar fashion in all the major database servers.

Oracle Database is an exception when it comes to the use of `varchar`. Oracle users should use the `varchar2` type when defining variable-length character columns.

Character sets

For languages that use the Latin alphabet, such as English, there is a sufficiently small number of characters such that only a single byte is needed to store each character. Other languages, such as Japanese and Korean, contain large numbers of characters, thus requiring multiple bytes of storage for each character. Such character sets are therefore called *multibyte character sets*.

MySQL can store data using various character sets, both single- and multibyte. To view the supported character sets in your server, you can use the `show` command, as in:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
```

```
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
| cp932    | SJIS for Windows Japanese   | cp932_japanese_ci   |      2 |
| eucjpms  | UJIS for Windows Japanese   | eucjpms_japanese_ci |      3 |
+----------+-----------------------------+---------------------+--------+
36 rows in set (0.11 sec)
```

If the value in the fourth column, `maxlen`, is greater than 1, then the character set is a multibyte character set.

When I installed the MySQL server, the `latin1` character set was automatically chosen as the default character set. However, you may choose to use a different character set for each character column in your database, and you can even store different character sets within the same table. To choose a character set other than the default when defining a column, simply name one of the supported character sets after the type definition, as in:

```
varchar(20) character set utf8
```

With MySQL, you may also set the default character set for your entire database:

```
create database foreign_sales character set utf8;
```

While this is as much information regarding character sets as I'm willing to discuss in an introductory book, there is a great deal more to the topic of internationalization than what is shown here. If you plan to deal with multiple or unfamiliar character sets, you may want to pick up a book such as Andy Deitsch and David Czarnecki's *Java Internationalization (http://oreilly.com/catalog/9780596000196/)* (O'Reilly) or Richard Gillam's *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard* (Addison-Wesley).

Text data

If you need to store data that might exceed the 64 KB limit for `varchar` columns, you will need to use one of the text types.

Table 2-2 shows the available text types and their maximum sizes.

Table 2-2. MySQL text types

Text type	Maximum number of bytes
`Tinytext`	255
`Text`	65,535
`Mediumtext`	16,777,215
`Longtext`	4,294,967,295

When choosing to use one of the text types, you should be aware of the following:

- If the data being loaded into a text column exceeds the maximum size for that type, the data will be truncated.
- Trailing spaces will not be removed when data is loaded into the column.
- When using `text` columns for sorting or grouping, only the first 1,024 bytes are used, although this limit may be increased if necessary.
- The different text types are unique to MySQL. SQL Server has a single `text` type for large character data, whereas DB2 and Oracle use a data type called `clob`, for Character Large Object.
- Now that MySQL allows up to 65,535 bytes for `varchar` columns (it was limited to 255 bytes in version 4), there isn't any particular need to use the `tinytext` or `text` type.

If you are creating a column for free-form data entry, such as a `notes` column to hold data about customer interactions with your company's customer service department, then `varchar` will probably be adequate. If you are storing documents, however, you should choose either the `mediumtext` or `longtext` type.

Oracle Database allows up to 2,000 bytes for `char` columns and 4,000 bytes for `varchar2` columns. SQL Server can handle up to 8,000 bytes for both `char` and `varchar` data.

Numeric Data

Although it might seem reasonable to have a single numeric data type called "numeric," there are actually several different numeric data types that reflect the various ways in which numbers are used, as illustrated here:

A column indicating whether a customer order has been shipped
: This type of column, referred to as a *Boolean*, would contain a `0` to indicate `false` and a `1` to indicate `true`.

A system-generated primary key for a transaction table
: This data would generally start at `1` and increase in increments of one up to a potentially very large number.

An item number for a customer's electronic shopping basket
: The values for this type of column would be positive whole numbers between 1 and, at most, 200 (for shopaholics).

Positional data for a circuit board drill machine
: High-precision scientific or manufacturing data often requires accuracy to eight decimal points.

To handle these types of data (and more), MySQL has several different numeric data types. The most commonly used numeric types are those used to store whole numbers. When specifying one of these types, you may also specify that the data is *unsigned*, which tells the server that all data stored in the column will be greater than or equal to zero. Table 2-3 shows the five different data types used to store whole-number integers.

Table 2-3. MySQL integer types

Type	Signed range	Unsigned range
`Tinyint`	−128 to 127	0 to 255
`Smallint`	−32,768 to 32,767	0 to 65,535
`Mediumint`	−8,388,608 to 8,388,607	0 to 16,777,215
`Int`	−2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`Bigint`	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

When you create a column using one of the integer types, MySQL will allocate an appropriate amount of space to store the data, which ranges from one byte for a `tinyint` to eight bytes for a `bigint`. Therefore, you should try to choose a type that will be large enough to hold the biggest number you can envision being stored in the column without needlessly wasting storage space.

For floating-point numbers (such as 3.1415927), you may choose from the numeric types shown in Table 2-4.

Table 2-4. MySQL floating-point types

Type	Numeric range
`Float(`*`p`*`,`*`s`*`)`	−3.402823466E+38 to −1.175494351E-38
	and 1.175494351E-38 to 3.402823466E+38
`Double(`*`p`*`,`*`s`*`)`	−1.7976931348623157E+308 to −2.2250738585072014E-308
	and 2.2250738585072014E-308 to 1.7976931348623157E+308

When using a floating-point type, you can specify a *precision* (the total number of allowable digits both to the left and to the right of the decimal point) and a *scale* (the number of allowable digits to the right of the decimal point), but they are not required. These values are represented in Table 2-4 as *`p`* and *`s`*. If you specify a precision and scale for your floating-point column, remember that the data stored in the column will be

rounded if the number of digits exceeds the scale and/or precision of the column. For example, a column defined as `float(4,2)` will store a total of four digits, two to the left of the decimal and two to the right of the decimal. Therefore, such a column would handle the numbers 27.44 and 8.19 just fine, but the number 17.8675 would be rounded to 17.87, and attempting to store the number 178.375 in your `float(4,2)` column would generate an error.

Like the integer types, floating-point columns can be defined as `unsigned`, but this designation only prevents negative numbers from being stored in the column rather than altering the range of data that may be stored in the column.

Temporal Data

Along with strings and numbers, you will almost certainly be working with information about dates and/or times. This type of data is referred to as *temporal*, and some examples of temporal data in a database include:

- The future date that a particular event is expected to happen, such as shipping a customer's order
- The date that a customer's order was shipped
- The date and time that a user modified a particular row in a table
- An employee's birth date
- The year corresponding to a row in a `yearly_sales` fact table in a data warehouse
- The elapsed time needed to complete a wiring harness on an automobile assembly line

MySQL includes data types to handle all of these situations. Table 2-5 shows the temporal data types supported by MySQL.

Table 2-5. MySQL temporal types

Type	Default format	Allowable values
`Date`	YYYY-MM-DD	`1000-01-01` to `9999-12-31`
`Datetime`	YYYY-MM-DD HH:MI:SS	`1000-01-01 00:00:00` to `9999-12-31 23:59:59`
`Timestamp`	YYYY-MM-DD HH:MI:SS	`1970-01-01 00:00:00` to `2037-12-31 23:59:59`
`Year`	YYYY	`1901` to `2155`
`Time`	HHH:MI:SS	`-838:59:59` to `838:59:59`

While database servers store temporal data in various ways, the purpose of a format string (second column of Table 2-5) is to show how the data will be represented when retrieved, along with how a date string should be constructed when inserting or updating a temporal column. Thus, if you wanted to insert the date March 23, 2005 into a `date` column using the default format `YYYY-MM-DD`, you would use the string

'2005-03-23'. Chapter 7 fully explores how temporal data is constructed and displayed.

Each database server allows a different range of dates for temporal columns. Oracle Database accepts dates ranging from 4712 BC to 9999 AD, while SQL Server only handles dates ranging from 1753 AD to 9999 AD (unless you are using SQL Server 2008's new `datetime2` data type, which allows for dates ranging from 1 AD to 9999 AD). MySQL falls in between Oracle and SQL Server and can store dates from 1000 AD to 9999 AD. Although this might not make any difference for most systems that track current and future events, it is important to keep in mind if you are storing historical dates.

Table 2-6 describes the various components of the date formats shown in Table 2-5.

Table 2-6. Date format components

Component	Definition	Range
YYYY	Year, including century	1000 to 9999
MM	Month	01 (January) to 12 (December)
DD	Day	01 to 31
HH	Hour	00 to 23
HHH	Hours (elapsed)	-838 to 838
MI	Minute	00 to 59
SS	Second	00 to 59

Here's how the various temporal types would be used to implement the examples shown earlier:

- Columns to hold the expected future shipping date of a customer order and an employee's birth date would use the `date` type, since it is unnecessary to know at what time a person was born and unrealistic to schedule a future shipment down to the second.
- A column to hold information about when a customer order was actually shipped would use the `datetime` type, since it is important to track not only the date that the shipment occurred but the time as well.
- A column that tracks when a user last modified a particular row in a table would use the `timestamp` type. The `timestamp` type holds the same information as the `datetime` type (year, month, day, hour, minute, second), but a `timestamp` column will automatically be populated with the current date/time by the MySQL server when a row is added to a table or when a row is later modified.
- A column holding just year data would use the `year` type.

- Columns that hold data regarding the length of time needed to complete a task would use the `time` type. For this type of data, it would be unnecessary and confusing to store a date component, since you are interested only in the number of hours/minutes/seconds needed to complete the task. This information could be derived using two `datetime` columns (one for the task start date/time and the other for the task completion date/time) and subtracting one from the other, but it is simpler to use a single `time` column.

Chapter 7 explores how to work with each of these temporal data types.

Table Creation

Now that you have a firm grasp on what data types may be stored in a MySQL database, it's time to see how to use these types in table definitions. Let's start by defining a table to hold information about a person.

Step 1: Design

A good way to start designing a table is to do a bit of brainstorming to see what kind of information would be helpful to include. Here's what I came up with after thinking for a short time about the types of information that describe a person:

- Name
- Gender
- Birth date
- Address
- Favorite foods

This is certainly not an exhaustive list, but it's good enough for now. The next step is to assign column names and data types. Table 2-7 shows my initial attempt.

Table 2-7. Person table, first pass

Column	Type	Allowable values
`Name`	`Varchar(40)`	
`Gender`	`Char(1)`	`M`, `F`
`Birth_date`	`Date`	
`Address`	`Varchar(100)`	
`Favorite_foods`	`Varchar(200)`	

The `name`, `address`, and `favorite_foods` columns are of type `varchar` and allow for free-form data entry. The `gender` column allows a single character which should equal only `M` or `F`. The `birth_date` column is of type `date`, since a time component is not needed.

Step 2: Refinement

In Chapter 1, you were introduced to the concept of *normalization*, which is the process of ensuring that there are no duplicate (other than foreign keys) or compound columns in your database design. In looking at the columns in the `person` table a second time, the following issues arise:

- The `name` column is actually a compound object consisting of a first name and a last name.
- Since multiple people can have the same name, gender, birth date, and so forth, there are no columns in the `person` table that guarantee uniqueness.
- The `address` column is also a compound object consisting of street, city, state/province, country, and postal code.
- The `favorite_foods` column is a list containing `0`, `1`, or more independent items. It would be best to create a separate table for this data that includes a foreign key to the `person` table so that you know to which person a particular food may be attributed.

After taking these issues into consideration, Table 2-8 gives a normalized version of the `person` table.

Table 2-8. Person table, second pass

Column	Type	Allowable values
`Person_id`	`Smallint (unsigned)`	
`First_name`	`Varchar(20)`	
`Last_name`	`Varchar(20)`	
`Gender`	`Char(1)`	`M, F`
`Birth_date`	`Date`	
`Street`	`Varchar(30)`	
`City`	`Varchar(20)`	
`State`	`Varchar(20)`	
`Country`	`Varchar(20)`	
`Postal_code`	`Varchar(20)`	

Now that the `person` table has a primary key (`person_id`) to guarantee uniqueness, the next step is to build a `favorite_food` table that includes a foreign key to the `person` table. Table 2-9 shows the result.

Table 2-9. Favorite_food table

Column	Type
Person_id	Smallint (unsigned)
Food	Varchar(20)

The person_id and food columns comprise the primary key of the favorite_food table, and the person_id column is also a foreign key to the person table.

How Much Is Enough?

Moving the favorite_foods column out of the person table was definitely a good idea, but are we done yet? What happens, for example, if one person lists "pasta" as a favorite food while another person lists "spaghetti"? Are they the same thing? In order to prevent this problem, you might decide that you want people to choose their favorite foods from a list of options, in which case you should create a food table with food_id and food_name columns, and then change the favorite_food table to contain a foreign key to the food table. While this design would be fully normalized, you might decide that you simply want to store the values that the user has entered, in which case you may leave the table as is.

Step 3: Building SQL Schema Statements

Now that the design is complete for the two tables holding information about people and their favorite foods, the next step is to generate SQL statements to create the tables in the database. Here is the statement to create the person table:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender CHAR(1),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Everything in this statement should be fairly self-explanatory except for the last item; when you define your table, you need to tell the database server what column or columns will serve as the primary key for the table. You do this by creating a *constraint* on the table. You can add several types of constraints to a table definition. This constraint is a *primary key constraint*. It is created on the person_id column and given the name pk_person.

While on the topic of constraints, there is another type of constraint that would be useful for the person table. In Table 2-7, I added a third column to show the allowable values for certain columns (such as 'M' and 'F' for the gender column). Another type of constraint called a *check constraint* constrains the allowable values for a particular column. MySQL allows a check constraint to be attached to a column definition, as in the following:

```
gender CHAR(1) CHECK (gender IN ('M','F')),
```

While check constraints operate as expected on most database servers, the MySQL server allows check constraints to be defined but does not enforce them. However, MySQL does provide another character data type called enum that merges the check constraint into the data type definition. Here's what it would look like for the gender column definition:

```
gender ENUM('M','F'),
```

Here's how the person table definition looks with an enum data type for the gender column:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender ENUM('M','F'),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Later in this chapter, you will see what happens if you try to add data to a column that violates its check constraint (or, in the case of MySQL, its enumeration values).

You are now ready to run the create table statement using the mysql command-line tool. Here's what it looks like:

```
mysql> CREATE TABLE person
    ->  (person_id SMALLINT UNSIGNED,
    ->   fname VARCHAR(20),
    ->   lname VARCHAR(20),
    ->   gender ENUM('M','F'),
    ->   birth_date DATE,
    ->   street VARCHAR(30),
    ->   city VARCHAR(20),
    ->   state VARCHAR(20),
    ->   country VARCHAR(20),
    ->   postal_code VARCHAR(20),
    ->   CONSTRAINT pk_person PRIMARY KEY (person_id)
    ->  );
 Query OK, 0 rows affected (0.27 sec)
```

After processing the `create table` statement, the MySQL server returns the message "Query OK, 0 rows affected," which tells me that the statement had no syntax errors. If you want to make sure that the `person` table does, in fact, exist, you can use the `describe` command (or `desc` for short) to look at the table definition:

```
mysql> DESC person;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| fname       | varchar(20)          | YES  |     | NULL    |       |
| lname       | varchar(20)          | YES  |     | NULL    |       |
| gender      | enum('M','F')        | YES  |     | NULL    |       |
| birth_date  | date                 | YES  |     | NULL    |       |
| street      | varchar(30)          | YES  |     | NULL    |       |
| city        | varchar(20)          | YES  |     | NULL    |       |
| state       | varchar(20)          | YES  |     | NULL    |       |
| country     | varchar(20)          | YES  |     | NULL    |       |
| postal_code | varchar(20)          | YES  |     | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
10 rows in set (0.06 sec)
```

Columns 1 and 2 of the `describe` output are self-explanatory. Column 3 shows whether a particular column can be omitted when data is inserted into the table. I purposefully left this topic out of the discussion for now (see the sidebar "What Is Null?" on page 29 for a short discourse), but we explore it fully in Chapter 4. The fourth column shows whether a column takes part in any keys (primary or foreign); in this case, the `person_id` column is marked as the primary key. Column 5 shows whether a particular column will be populated with a default value if you omit the column when inserting data into the table. The `person_id` column shows a default value of `0`, although this would work only once, since each row in the `person` table must contain a unique value for this column (since it is the primary key). The sixth column (called "Extra") shows any other pertinent information that might apply to a column.

What Is Null?

In some cases, it is not possible or applicable to provide a value for a particular column in your table. For example, when adding data about a new customer order, the `ship_date` column cannot yet be determined. In this case, the column is said to be *null* (note that I do not say that it *equals* null), which indicates the absence of a value. Null is used for various cases where a value cannot be supplied, such as:

- Not applicable
- Unknown
- Empty set

When designing a table, you may specify which columns are allowed to be null (the default), and which columns are not allowed to be null (designated by adding the keywords `not null` after the type definition).

Now that you've created the person table, your next step is to create the favorite_food table:

```
mysql> CREATE TABLE favorite_food
    ->  (person_id SMALLINT UNSIGNED,
    ->   food VARCHAR(20),
    ->   CONSTRAINT pk_favorite_food PRIMARY KEY (person_id, food),
    ->   CONSTRAINT fk_fav_food_person_id FOREIGN KEY (person_id)
    ->     REFERENCES person (person_id)
    ->  );
Query OK, 0 rows affected (0.10 sec)
```

This should look very similar to the create table statement for the person table, with the following exceptions:

- Since a person can have more than one favorite food (which is the reason this table was created in the first place), it takes more than just the person_id column to guarantee uniqueness in the table. This table, therefore, has a two-column primary key: person_id and food.
- The favorite_food table contains another type of constraint called a *foreign key constraint*. This constrains the values of the person_id column in the favorite_food table to include *only* values found in the person table. With this constraint in place, I will not be able to add a row to the favorite_food table indicating that person_id 27 likes pizza if there isn't already a row in the person table having a person_id of 27.

If you forget to create the foreign key constraint when you first create the table, you can add it later via the alter table statement.

Describe shows the following after executing the create table statement:

```
mysql> DESC favorite_food;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| food        | varchar(20)          |      | PRI |         |       |
+-------------+----------------------+------+-----+---------+-------+
```

Now that the tables are in place, the next logical step is to add some data.

Populating and Modifying Tables

With the person and favorite_food tables in place, you can now begin to explore the four SQL data statements: insert, update, delete, and select.

Inserting Data

Since there is not yet any data in the `person` and `favorite_food` tables, the first of the four SQL data statements to be explored will be the `insert` statement. There are three main components to an `insert` statement:

- The name of the table into which to add the data
- The names of the columns in the table to be populated
- The values with which to populate the columns

You are not required to provide data for every column in the table (unless all the columns in the table have been defined as `not null`). In some cases, those columns that are not included in the initial `insert` statement will be given a value later via an `update` statement. In other cases, a column may never receive a value for a particular row of data (such as a customer order that is canceled before being shipped, thus rendering the `ship_date` column inapplicable).

Generating numeric key data

Before inserting data into the `person` table, it would be useful to discuss how values are generated for numeric primary keys. Other than picking a number out of thin air, you have a couple of options:

- Look at the largest value currently in the table and add one.
- Let the database server provide the value for you.

Although the first option may seem valid, it proves problematic in a multiuser environment, since two users might look at the table at the same time and generate the same value for the primary key. Instead, all database servers on the market today provide a safe, robust method for generating numeric keys. In some servers, such as the Oracle Database, a separate schema object is used (called a *sequence*); in the case of MySQL, however, you simply need to turn on the *auto-increment* feature for your primary key column. Normally, you would do this at table creation, but doing it now provides the opportunity to learn another SQL schema statement, `alter table`, which is used to modify the definition of an existing table:

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED AUTO_INCREMENT;
```

This statement essentially redefines the `person_id` column in the `person` table. If you describe the table, you will now see the auto-increment feature listed under the "Extra" column for `person_id`:

```
mysql> DESC person;
+-------------+--------------------------+------+-----+---------+----------------+
| Field       | Type                     | Null | Key | Default | Extra          |
+-------------+--------------------------+------+-----+---------+----------------+
| person_id   | smallint(5) unsigned     |      | PRI | NULL    | auto_increment |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
```

When you insert data into the `person` table, simply provide a `null` value for the `person_id` column, and MySQL will populate the column with the next available number (by default, MySQL starts at 1 for auto-increment columns).

The insert statement

Now that all the pieces are in place, it's time to add some data. The following statement creates a row in the `person` table for William Turner:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (null, 'William','Turner', 'M', '1972-05-27');
Query OK, 1 row affected (0.01 sec)
```

The feedback ("Query OK, 1 row affected") tells you that your statement syntax was proper, and that one row was added to the database (since it was an `insert` statement). You can look at the data just added to the table by issuing a `select` statement:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

As you can see, the MySQL server generated a value of `1` for the primary key. Since there is only a single row in the `person` table, I neglected to specify which row I am interested in and simply retrieved all the rows in the table. If there were more than one row in the table, however, I could add a `where` clause to specify that I want to retrieve data only for the row having a value of `1` for the `person_id` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

While this query specifies a particular primary key value, you can use any column in the table to search for rows, as shown by the following query, which finds all rows with a value of `'Turner'` for the `lname` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

Before moving on, a couple of things about the earlier `insert` statement are worth mentioning:

- Values were not provided for any of the address columns. This is fine, since `null`s are allowed for those columns.
- The value provided for the `birth_date` column was a string. As long as you match the required format shown in Table 2-5, MySQL will convert the string to a date for you.
- The column names and the values provided must correspond in number and type. If you name seven columns and provide only six values, or if you provide values that cannot be converted to the appropriate data type for the corresponding column, you will receive an error.

William has also provided information about his favorite three foods, so here are three `insert` statements to store his food preferences:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

Here's a query that retrieves William's favorite foods in alphabetical order using an `order by` clause:

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
```

```
+---------+
3 rows in set (0.02 sec)
```

The `order by` clause tells the server how to sort the data returned by the query. Without the `order by` clause, there is no guarantee that the data in the table will be retrieved in any particular order.

So that William doesn't get lonely, you can execute another `insert` statement to add Susan Smith to the `person` table:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date,
    ->   street, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'F', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
 Query OK, 1 row affected (0.01 sec)
```

Since Susan was kind enough to provide her address, we included five more columns than when William's data was inserted. If you query the table again, you will see that Susan's row has been assigned the value `2` for its primary key value:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
|         2 | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

Can I Get That in XML?

If you will be working with XML data, you will be happy to know that most database servers provide a simple way to generate XML output from a query. With MySQL, for example, you can use the `--xml` option when invoking the `mysql` tool, and all your output will automatically be formatted using XML. Here's what the favorite-food data looks like as an XML document:

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...

Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="person_id">1</field>
        <field name="food">cookies</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">nachos</field>
  </row>
```

```
  <row>
        <field name="person_id">1</field>
        <field name="food">pizza</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

With SQL Server, you don't need to configure your command-line tool; you just need to add the `for xml` clause to the end of your query, as in:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

Updating Data

When the data for William Turner was initially added to the table, data for the various address columns was omitted from the `insert` statement. The next statement shows how these columns can be populated via an `update` statement:

```
mysql> UPDATE person
    -> SET street = '1225 Tremont St.',
    ->   city = 'Boston',
    ->   state = 'MA',
    ->   country = 'USA',
    ->   postal_code = '02138'
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The server responded with a two-line message: the "Rows matched: 1" item tells you that the condition in the `where` clause matched a single row in the table, and the "Changed: 1" item tells you that a single row in the table has been modified. Since the `where` clause specifies the primary key of William's row, this is exactly what you would expect to have happen.

Depending on the conditions in your `where` clause, it is also possible to modify more than one row using a single statement. Consider, for example, what would happen if your `where` clause looked as follows:

```
WHERE person_id < 10
```

Since both William and Susan have a `person_id` value less than 10, both of their rows would be modified. If you leave off the `where` clause altogether, your `update` statement will modify every row in the table.

Deleting Data

It seems that William and Susan aren't getting along very well together, so one of them has got to go. Since William was there first, Susan will get the boot courtesy of the `delete` statement:

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

Again, the primary key is being used to isolate the row of interest, so a single row is deleted from the table. Similar to the `update` statement, more than one row can be deleted depending on the conditions in your `where` clause, and all rows will be deleted if the `where` clause is omitted.

When Good Statements Go Bad

So far, all of the SQL data statements shown in this chapter have been well formed and have played by the rules. Based on the table definitions for the `person` and `favorite_food` tables, however, there are lots of ways that you can run afoul when inserting or modifying data. This section shows you some of the common mistakes that you might come across and how the MySQL server will respond.

Nonunique Primary Key

Because the table definitions include the creation of primary key constraints, MySQL will make sure that duplicate key values are not inserted into the tables. The next statement attempts to bypass the auto-increment feature of the `person_id` column and create another row in the `person` table with a `person_id` of `1`:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

There is nothing stopping you (with the current schema objects, at least) from creating two rows with identical names, addresses, birth dates, and so on, as long as they have different values for the `person_id` column.

Nonexistent Foreign Key

The table definition for the `favorite_food` table includes the creation of a foreign key constraint on the `person_id` column. This constraint ensures that all values of `person_id` entered into the `favorite_food` table exist in the `person` table. Here's what would happen if you tried to create a row that violates this constraint:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails ('bank'.'favorite_food', CONSTRAINT 'fk_fav_food_person_id' FOREIGN KEY
('person_id') REFERENCES 'person' ('person_id'))
```

In this case, the `favorite_food` table is considered the *child* and the `person` table is considered the *parent*, since the `favorite_food` table is dependent on the `person` table

for some of its data. If you plan to enter data into both tables, you will need to create a row in `parent` before you can enter data into `favorite_food`.

Foreign key constraints are enforced only if your tables are created using the InnoDB storage engine. We discuss MySQL's storage engines in Chapter 12.

Column Value Violations

The `gender` column in the `person` table is restricted to the values `'M'` for male and `'F'` for female. If you mistakenly attempt to set the value of the column to any other value, you will receive the following response:

```
mysql> UPDATE person
    -> SET gender = 'Z'
    -> WHERE person_id = 1;
ERROR 1265 (01000): Data truncated for column 'gender' at row 1
```

The error message is a bit confusing, but it gives you the general idea that the server is unhappy about the value provided for the `gender` column.

Invalid Date Conversions

If you construct a string with which to populate a `date` column, and that string does not match the expected format, you will receive another error. Here's an example that uses a date format that does not match the default date format of "YYYY-MM-DD":

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980' for column 'birth_date'
at row 1
```

In general, it is always a good idea to explicitly specify the format string rather than relying on the default format. Here's another version of the statement that uses the `str_to_date` function to specify which format string to use:

```
mysql> UPDATE person
    -> SET birth_date = str_to_date('DEC-21-1980' , '%b-%d-%Y')
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Not only is the database server happy, but William is happy as well (we just made him eight years younger, without the need for expensive cosmetic surgery!).

Earlier in the chapter, when I discussed the various temporal data types, I showed date-formatting strings such as "YYYY-MM-DD". While many database servers use this style of formatting, MySQL uses `%Y` to indicate a four-character year. Here are a few more formatters that you might need when converting strings to `datetime`s in MySQL:

```
%a The short weekday name, such as Sun, Mon, ...
%b The short month name, such as Jan, Feb, ...
%c The numeric month (0..12)
%d The numeric day of the month (00..31)
%f The number of microseconds (000000..999999)
%H The hour of the day, in 24-hour format (00..23)
%h The hour of the day, in 12-hour format (01..12)
%i The minutes within the hour (00..59)
%j The day of year (001..366)
%M The full month name (January..December)
%m The numeric month
%p AM or PM
%s The number of seconds (00..59)
%W The full weekday name (Sunday..Saturday)
%w The numeric day of the week (0=Sunday..6=Saturday)
%Y The four-digit year
```

The Bank Schema

For the remainder of the book, you use a group of tables that model a community bank. Some of the tables include `Employee`, `Branch`, `Account`, `Customer`, `Product`, and `Transaction`. The entire schema and example data should have been created when you followed the final steps at the beginning of the chapter for loading the MySQL server and generating the sample data. To see a diagram of the tables and their columns and relationships, see Appendix A.

Table 2-10 shows all the tables used in the bank schema along with short definitions.

Table 2-10. Bank schema definitions

Table name	Definition
`Account`	A particular product opened for a particular customer
`Branch`	A location at which banking transactions are conducted
`Business`	A corporate customer (subtype of the `Customer` table)
`Customer`	A person or corporation known to the bank
`Department`	A group of bank employees implementing a particular banking function
`Employee`	A person working for the bank
`Individual`	A noncorporate customer (subtype of the `Customer` table)
`Officer`	A person allowed to transact business for a corporate customer
`Product`	A banking service offered to customers
`Product_type`	A group of products having a similar function
`Transaction`	A change made to an account balance

Feel free to experiment with the tables as much as you want, including adding your own tables to expand the bank's business functions. You can always drop the database and re-create it from the downloaded file if you want to make sure your sample data is intact.

If you want to see the tables available in your database, you can use the `show tables` command, as in:

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_bank |
+----------------+
| account        |
| branch         |
| business       |
| customer       |
| department     |
| employee       |
| favorite_food  |
| individual     |
| officer        |
| person         |
| product        |
| product_type   |
| transaction    |
+----------------+
13 rows in set (0.10 sec)
```

Along with the 11 tables in the bank schema, the table listing also includes the two tables created in this chapter: `person` and `favorite_food`. These tables will not be used in later chapters, so feel free to drop them by issuing the following commands:

```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)
mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

If you want to look at the columns in a table, you can use the `describe` command. Here's an example of the `describe` output for the `customer` table:

```
mysql> DESC customer;
+-------------+------------------+------+-----+---------+----------------+
| Field       | Type             | Null | Key | Default | Extra          |
+-------------+------------------+------+-----+---------+----------------+
| cust_id     | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| fed_id      | varchar(12)      | NO   |     | NULL    |                |
| cust_type_cd | enum('I','B')   | NO   |     | NULL    |                |
| address     | varchar(30)      | YES  |     | NULL    |                |
| city        | varchar(20)      | YES  |     | NULL    |                |
| state       | varchar(20)      | YES  |     | NULL    |                |
| postal_code | varchar(10)      | YES  |     | NULL    |                |
+-------------+------------------+------+-----+---------+----------------+
7 rows in set (0.03 sec)
```

The more comfortable you are with the example database, the better you will understand the examples and, consequently, the concepts in the following chapters.