Getting Started with Java™

JBuilder® X

Borland®
Excellence Endures™

Borland Software Corporation
100 Enterprise Way
Scotts Valley, California 95066-3249
www.borland.com

Refer to the file `deploy.html` located in the `redist` directory of your JBuilder product for a complete list of files that you can distribute in accordance with the JBuilder License Statement and Limited Warranty.

Borland Software Corporation may have patents and/or pending patent applications covering subject matter in this document. Please refer to the product CD or the About dialog box for the list of applicable patents. The furnishing of this document does not give you any license to these patents.

For third-party conditions and disclaimers, see the Release Notes on your JBuilder product CD.

Printed in the U.S.A.

JXE0010WW21000gsjava 5E5R1103
0304050607-9 8 7 6 5 4 3 2 1
PDF

Contents

Chapter

1

Introduction

Java is an *object-oriented* programming language. Switching to object-oriented programming (OOP) from other programming paradigms can be difficult. Java focuses on creating objects (data structures or behaviors) that can be assessed and manipulated by the program.

Like other programming languages, Java provides support for reading and writing data to and from different input and output devices. Java uses processes that increase the efficiency of input/output, facilitate internationalization, and provide better support for non-UNIX platforms. Java looks over your program as it runs and automatically deallocates memory that is no longer required. This means you don't have to keep track of memory pointers or manually deallocate memory. This feature means a program is less likely to crash and that memory can't be intentionally misused.

This book is intended to serve programmers who use other languages as a general introduction to the Java programming language. The Borland Community site provides an annotated list of books on Java programming and related subjects at `http://community.borland.com/books/java/0,1427,c|3,00.html`. Examples of applications, APIs, and code snippets are at `http://codecentral.borland.com/codecentral/ccweb.exe/home`.

This book includes the following chapters:

- Java syntax: Chapter 2, "Java language elements," Chapter 3, "Java language structure," and Chapter 4, "Java language control."

  These three chapters define basic Java syntax and introduce you to object-oriented programming concepts. Each section is divided into two main parts: "Terms" and "Applying concepts." "Terms" builds vocabulary, adding to concepts you already understand. "Applying concepts"

demonstrates the use of concepts presented up to that point. Some concepts are revisited several times, at increasing levels of complexity.

- Chapter 5, “The Java class libraries”

  This chapter presents an overview of the Java 2 class libraries and the Java 2 Platform editions.

- Chapter 6, “Object-oriented programming in Java”

  This chapter introduces the object-oriented features of Java. You will create Java classes, instantiate objects, and access member variables in a short tutorial. You will learn to use inheritance to create new classes, use interfaces to add new capabilities to your classes, use polymorphism to make related classes respond in different ways to the same message, and use packages to group related classes together.

- Chapter 7, “Threading techniques”

  A thread is a single sequential flow of control within a program. One of the powerful aspects of the Java language is you can easily program multiple threads of execution to run concurrently within the same program. This chapter explains how to create multithreaded programs, and provides links to other resources with more in-depth information.

- Chapter 8, “Serialization”

  Serialization saves and restores a Java object’s state. This chapter describes how to serialize objects using Java. It describes the `Serializable` interface, how to write an object to disk, and how to read the object back into memory again.

- Chapter 9, “An introduction to the Java Virtual Machine”

  The JVM is the native software that allows a Java program to run on a particular machine. This chapter explains the JVM’s general structure and purpose. It discusses the major roles of the JVM, particularly in Java security. It goes into more detail about three specific security features: the Java verifier, the Security Manager, and the Class Loader.

- Chapter 10, “Working with the Java Native Interface (JNI)”

  This chapter explains how to invoke native methods in Java applications using the Java Native Method Interface (JNI). It begins by explaining how the JNI works, then discusses the `native` keyword and how any Java method can become a `native` method. Finally, it examines the JDK’s `javah` tool, which is used to generate C header files for Java classes.

- Appendix A, “Java language quick reference”

  This appendix contains a partial list of class libraries and their main functions, a list of the Java2 platform editions, a complete list of Java keywords as of JDK 1.3, extensive tables of data type conversions between

primitive and reference types, Java escape sequences, and extensive tables of operators and their actions.

- Appendix B, “Learning more about Java”

  Resources on the Java language abound. This chapter has links to, and ideas for finding, good sources of information about the Java language.

Documentation conventions

The Borland documentation for JBuilder uses the typefaces and symbols described in the following table to indicate special text.

Table 1.1 Typeface and symbol conventions

Typeface	Meaning
Bold	Bold is used for java tools, bmj (Borland Make for Java), bcj (Borland Compiler for Java), and compiler options. For example: **javac**, **bmj**, **-classpath**.
Italics	Italicized words are used for new terms being defined, for book titles, and occasionally for emphasis.
Keycaps	This typeface indicates a key on your keyboard, such as “Press *Esc* to exit a menu.”
`Monospaced type`	Monospaced type represents the following: ■ text as it appears onscreen ■ anything you must type, such as “Type `Hello World` in the Title field of the Application wizard.” ■ file names ■ path names ■ directory and folder names ■ commands, such as `SET PATH` ■ Java code ■ Java data types, such as `boolean`, `int`, and `long`. ■ Java identifiers, such as names of variables, classes, package names, interfaces, components, properties, methods, and events ■ argument names ■ field names ■ Java keywords, such as `void` and `static`
[ ]	Square brackets in text or syntax listings enclose optional items. Do not type the brackets.

Table 1.1 Typeface and symbol conventions (continued)

Typeface	Meaning
< >	Angle brackets are used to indicate variables in directory paths, command options, and code samples. For example, `<filename>` may be used to indicate where you need to supply a file name (including file extension), and <username> typically indicates that you must provide your user name. When replacing variables in directory paths, command options, and code samples, replace the entire variable, including the angle brackets (< >). For example, you would replace `<filename>` with the name of a file, such as `employee.jds`, and omit the angle brackets. **Note:** Angle brackets are used in HTML, XML, JSP, and other tag-based files to demarcate document elements, such as <font color=red> and <ejb-jar>. The following convention describes how variable strings are specified within code samples that are already using angle brackets for delimiters.
Italics, serif	This formatting is used to indicate variable strings within code samples that are already using angle brackets as delimiters. For example, `<url="jdbc:borland:`*jbuilder*`\\samples\\guestbook.jds">`
...	In code examples, an ellipsis (...) indicates code that has been omitted from the example to save space and improve clarity. On a button, an ellipsis indicates that the button links to a selection dialog box.

JBuilder is available on multiple platforms. See the following table for a description of platform conventions used in the documentation.

Table 1.2 Platform conventions

Item	Meaning
Paths	Directory paths in the documentation are indicated with a forward slash (/). For Windows platforms, use a backslash (\).
Home directory	The location of the standard home directory varies by platform and is indicated with a variable, `<home>`. ■ For UNIX and Linux, the home directory can vary. For example, it could be `/user/<username>` or `/home/<username>` ■ For Windows NT, the home directory is `C:\Winnt\Profiles\<username>` ■ For Windows 2000 and XP, the home directory is `C:\Documents and Settings\<username>`
Screen shots	Screen shots reflect the Metal Look & Feel on various platforms.

Developer support and resources

Borland provides a variety of support options and information resources to help developers get the most out of their Borland products. These options include a range of Borland Technical Support programs, as well as free services on the Internet, where you can search our extensive information base and connect with other users of Borland products.

Contacting Borland Developer Support

Borland offers several support programs for customers and prospective customers. You can choose from several categories of support, ranging from free support on installation of the Borland product to fee-based consultant-level support and extensive assistance.

For more information about Borland's developer support services, see our web site at `http://www.borland.com/devsupport/`, call Borland Assist at (800) 523-7070, or contact our Sales Department at (831) 431-1064.

When contacting support, be prepared to provide complete information about your environment, the version of the product you are using, and a detailed description of the problem.

For support on third-party tools or documentation, contact the vendor of the tool.

Online resources

You can get information from any of these online sources:

World Wide Web	`http://www.borland.com/` `http://www.borland.com/techpubs/jbuilder/`
Electronic newsletters	To subscribe to electronic newsletters, use the online form at: `http://www.borland.com/products/newsletters/index.html`

World Wide Web

Check `www.borland.com/jbuilder` regularly. This is where the Java Products Development Team posts white papers, competitive analyses, answers to frequently asked questions, sample applications, updated software, updated documentation, and information about new and existing products.

You may want to check these URLs in particular:

- `http://www.borland.com/jbuilder/` (updated software and other files)
- `http://www.borland.com/techpubs/jbuilder/` (updated documentation and other files)
- `http://community.borland.com/` (contains our web-based news magazine for developers)

Borland newsgroups

When you register JBuilder you can participate in many threaded discussion groups devoted to JBuilder. The Borland newsgroups provide a means for the global community of Borland customers to exchange tips and techniques about Borland products and related tools and technologies.

You can find user-supported newsgroups for JBuilder and other Borland products at `http://www.borland.com/newsgroups/`.

Usenet newsgroups

The following Usenet groups are devoted to Java and related programming issues:

- news:comp.lang.java.advocacy
- news:comp.lang.java.announce
- news:comp.lang.java.beans
- news:comp.lang.java.databases
- news:comp.lang.java.gui
- news:comp.lang.java.help
- news:comp.lang.java.machine
- news:comp.lang.java.programmer
- news:comp.lang.java.security
- news:comp.lang.java.softwaretools

Note These newsgroups are maintained by users and are not official Borland sites.

Reporting bugs

If you find what you think may be a bug in the software, please report it to Borland at one of the following sites:

- Support Programs page at `http://www.borland.com/devsupport/namerica/`. Click the “Reporting Defects” link to bring up the Entry Form.
- Quality Central at `http://qc.borland.com`. Follow the instructions on the Quality Central page in the “Bugs Reports” section.

When you report a bug, please include all the steps needed to reproduce the bug, including any special environmental settings you used and other programs you were using with JBuilder. Please be specific about the expected behavior versus what actually happened.

If you have comments (compliments, suggestions, or issues) for the JBuilder documentation team, you may email `jpgpubs@borland.com`. This is for documentation issues only. Please note that you must address support issues to developer support.

JBuilder is made by developers for developers. We really value your input.

Chapter

2

Java language elements

This section provides you with foundational concepts about the elements of the Java programming language that will be used throughout this chapter. It assumes you understand general programming concepts, but have little or no experience with Java.

Terms

The following terms and concepts are discussed in this chapter:

- Identifier
- Data type
- Strings
- Arrays
- Variable
- Literal

Identifier

The identifier is the name you choose to call an element (such as a variable or a method). Java will accept any valid identifier, but for reasons of usability, it's best to use a plain-language term that's modified to meet the following requirements:

- It should start with a letter. Strictly speaking, it can begin with a Unicode currency symbol or an underscore (_), but some of these symbols may be used in imported files or internal processing. They are best avoided.

- After that, it may contain any alphanumeric characters (letters or numbers), underscores, or Unicode currency symbols (such as £ or $), but no other special characters.
- It must be all one word (no spaces or hyphens).

Capitalization of an identifier depends on the kind of identifier it is. Java is case-sensitive, so be careful of capitalization. Correct capitalization styles are mentioned in context.

Data type

Data types classify the kind of information that certain Java programming elements can contain. Data types fall into two main categories:

- Primitive or basic
- Composite or reference

Naturally, different kinds of data types can hold different kinds and amounts of information. You can convert the data type of a variable to a different type, within limits: you cannot cast to or from the `boolean` type, and you cannot cast an object to an object of an unrelated class.

Java will prevent you from risking your data. This means it will easily let you convert a variable or object to a larger type, but will try to prevent you from converting it to a smaller type. When you change a data type with a larger capacity to one with a smaller capacity, you must use a type of statement called a *type cast*.

Primitive data types

Primitive, or basic, data types are classified as Boolean (specifying an on/off state), character (for single characters and Unicode characters), integer (for whole numbers), or floating-point (for decimal numbers). In code, primitive data types are all lower case.

The Boolean data type is called `boolean`, and takes one of two values: `true` or `false`. Java doesn't store these values numerically, but uses the `boolean` data type to store these values.

The character data type is called `char` and takes single Unicode characters with values up to 16 bits long. In Java, Unicode characters (letters, special characters, and punctuation marks) are put between single quotation marks: `'b'`. Java's Unicode default value is `\u0000`, ranging from `\u0000` to `\uFFFF`.

Briefly, the Unicode numbering system takes numbers from 0 to 65535, but the numbers must be specified in hexadecimal notation, preceded by the escape sequence `\u`.

Not all special characters can be represented in this way. Java provides its own set of escape sequences, many of which can be found in the "Escape sequences" table on page 138.

In Java, the size of primitive data types is absolute, rather than platform-dependent. This improves portability.

Different numeric data types take different kinds and sizes of numbers. Their names and capacities are listed below:

Type	Attributes	Range
`double`	Java's default. A floating-point type that takes an 8-byte number to about fifteen decimal places.	+/– 9.00×10^{18}
`int`	Most common option. An integer type that takes a 4-byte whole number.	+/– 2×10^{9}
`long`	An integer type that takes an 8-byte whole number.	+/– 9×10^{18}
`float`	A floating-point type that takes a 4-byte number to about seven decimal places.	+/– 2.0×10^{9}
`short`	An integer type that takes a 2-byte whole number.	+/– 32768
`byte`	An integer type that takes a 1-byte whole number.	+/– 128

Composite data types

Each of the data types above accepts one number, one character, or one state. Composite, or reference, data types consist of more than a single element. Composite data types are of two kinds: classes and arrays. Class and array names start with an upper case letter and the first letter of each natural word within the name is capitalized also, for instance, `NameOfClass`.

A class is a complete and coherent piece of code that defines a logically unified set of objects and their behavior. For more information on classes, see Chapter 6, "Object-oriented programming in Java."

Any class can be used as a data type once it has been created and imported into the program. Because the `String` class is the class most often used as a data type, we will focus on it in this chapter.

Strings

The `String` data type is actually the `String` class. The `String` class stores any sequence of alphanumeric characters, spaces, and normal punctuation (termed *strings*), enclosed in double quotes. Strings can contain any of the Unicode escape sequences and require `\"` to put double quotes inside of a string, but, generally, the `String` class itself tells the program how to interpret the characters correctly.

Arrays

An array is a data structure containing a group of values of the same data type. For instance, an array can accept a group of `String` values, a group of `int` values, or a group of `boolean` values. As long as all of the values are of the same data type, they can go into the same array.

Arrays are characterized by a pair of square brackets. When you declare an array in Java, you can put the brackets either after the identifier or after the data type:

```
int studentID[];
char[] grades;
```

Note that the array size is not specified. Declaring an array does not allocate memory for that array. In most other languages the array's size must be included in its declaration, but in Java you don't specify its size until you use it. Then the appropriate memory is allocated.

Variable

A variable is a value that a programmer names and defines. Variables need an identifier and a value.

Literal

A literal is the actual representation of a number, a character, a state, or a string. A literal represents the value of an identifier.

Alphanumeric literals include strings in double quotes, single `char` characters in single quotes, and `boolean` true/false values.

Integer literals may be stored as decimals, octals, or hexadecimals, but think about your syntax: any integer with a leading 0 (as in a date) will be interpreted as an octal. Floating point literals can only be expressed as decimals. They will be treated as `double` unless you specify the type.

For a more detailed explanation of literals and their capacities, see *The Java Handbook* by Patrick Naughton.

Applying concepts

The following sections demonstrate how to apply the terms and concepts introduced earlier in this chapter.

Declaring variables

The act of declaring a variable sets aside memory for the variable you declare. Declaring a variable requires only two things: a data type and an identifier, in that order. The data type tells the program how much memory to allocate. The identifier labels the allocated memory.

Declare the variable only once. Once you have declared the variable appropriately, just refer to its identifier in order to access that block of memory.

Variable declarations look like this:

`boolean isOn;`	The data type `boolean` can be set to `true` or `false`. The identifier `isOn` is the name that the programmer has given to the memory allocated for this variable. The name `isOn` has meaning for the human reader as something that would logically accept true/false values.
`int studentsEnrolled;`	The data type `int` tells you that you will be dealing with a whole number of less than ten digits. The identifier `studentsEnrolled` suggests what the number will signify. Since students are whole people, the appropriate data type calls for whole numbers.
`float creditCardSales;`	The data type `float` is appropriate because money is generally represented in decimals. You know that money is involved because the programmer has usefully named this variable `creditCardSales`.

Methods

Methods in Java are equivalent to functions or subroutines in other languages. The method defines an action to be performed on an object.

Methods consist of a name and a pair of parentheses:

```
getData()
```

Here, `getData` is the name and the parentheses tell the program that it is a method.

If the method needs particular information in order to get its job done, what it needs goes inside the parentheses. What's inside the parentheses is called the *argument*, or *arg* for short. In a method declaration, the arg must include a data type and an identifier:

```
drawString(String remark)
```

Here, `drawString` is the name of the method, and `String remark` is the data type and variable name for the string that the method must draw.

You must tell the program what type of data the method will return, or if it will return anything at all. This is called the *return type*. You can make a method return data of any primitive type. If the method doesn't need to return anything (as in most action-performing methods), the return type must be `void`.

Return type, name, and parentheses with any needed args give a very basic method declaration:

```
String drawString(String remark);
```

Your method is probably more complex than that. Once you have typed and named it and told it what args it will need (if any), you must define it completely. You do this below the method name, nesting the body of the

definition in a pair of curly braces. This gives a more complex method declaration:

```
String drawString(String remark) {         //Declares the method.
  remark = "My, what big teeth you have!"; //Defines what's in the method.
}                                          //Closes the method body.
```

Once you have defined the method, you only need to refer to it by its name and pass it any args it needs to do its job right then:

```
drawString(remark);.
```

Chapter

3

Java language structure

This section provides you with foundational concepts about the structure of the Java programming language that will be used throughout this chapter. It assumes you understand general programming concepts, but have little or no experience with Java.

Terms

The following terms and concepts are discussed in this chapter:

- Keywords
- Operators
- Comments
- Statements
- Code blocks
- Understanding scope

Keywords

Keywords are reserved Java terms that modify other syntax elements. Keywords can define an object's accessibility, a method's flow, or a variable's data type. Keywords can never be used as identifiers.

Many of Java's keywords are borrowed from C/C++. Also, as in C/C++, keywords are always written in lowercase. Generally speaking, Java's

keywords can be categorized according to their functions (examples are in parentheses):

- Data and return types and terms (`int, void, return`)
- Package, class, member, and interface (`package, class, static`)
- Access modifiers (`public, private, protected`)
- Loops and loop controls (`if, switch, break`)
- Exception handling (`throw, try, finally`)
- Reserved words—not used yet, but unavailable (`goto, const`)

Some keywords are discussed in context in these chapters. For a complete list of keywords and what they mean, see the "Keywords" table on page 123.

Operators

Operators allow you to access, manipulate, relate, or refer to Java language elements, from variables to classes. Operators have properties of *precedence* and *associativity*. When several operators act on the same element (or *operand*), the operators' precedence determines which operator will act first. When more than one operator has the same precedence, the rules of associativity apply. These rules are generally mathematical; for instance, operators will usually be used from left to right, and operator expressions inside parentheses will be evaluated before operator expressions outside parentheses.

Operators generally fall into six categories: assignment, arithmetic, logical, comparison, bitwise, and ternary.

Assignment means storing the value to the *right* of the `=` inside the variable to the *left* of it. You can either assign a value to a variable *when* you declare it or *after* you have declared it. The machine doesn't care; you decide which way makes sense in your program and your practice:

```
double bankBalance;           //Declaration
bankBalance = 100.35;         //Assignment
double bankBalance = 100.35;  //Declaration with assignment
```

In both cases, the value of `100.35` is stored inside the memory reserved by the declaration of the `bankBalance` variable.

Assignment operators allow you to assign values to variables. They also allow you to perform an operation on an expression and then assign the new value to the right-hand operand, using a single combined expression.

Arithmetic operators perform mathematical calculations on both integer and floating-point values. The usual mathematical signs apply: `+` adds, `-` subtracts, `*` multiplies, and `/` divides two numbers.

Logical, or Boolean, operators allow the programmer to group `boolean` expressions in a useful way, telling the program exactly how to determine a specific condition.

Comparison operators evaluate single expressions against other parts of the code. More complex comparisons (like string comparisons) are done programmatically.

Bitwise operators act on the individual 0s and 1s of binary digits. Java's bitwise operators can preserve the sign of the original number; not all languages do.

The ternary operator, `?:`, provides a shorthand way of writing a very simple `if-then-else` statement. It consists of a Boolean condition statement followed by two expressions:

```
condition1 ? expression2 : expression3 ;
```

The conditional expression is evaluated first; if it's true, the second expression is evaluated; if the second expression is false, then the third expression is used.

Below is a partial list of other operators and their attributes:

Operator	Operand	Behavior
.	object member	Accesses a member of the object.
(<type>)	data type	Casts a data type.[1]
+	String	Joins up strings (concatenator).
	number	Adds.
-	number	This is the unary[2] minus (reverses number sign).
	number	Subtracts.
!	boolean	This is the `boolean NOT` operator.
&	integer, boolean	This is both the bitwise (integer) and `boolean AND` operator. When doubled (`&&`), it is the `boolean` conditional `AND`.
=	most elements with variables	Assigns an element to another element (for instance, a value to a variable, or a class to an instance). This can be combined with other operators to perform the other operation and assign the resulting value. For instance, `+=` adds the left-hand value to the right, then assigns the new value to the right-hand side of the expression.

1. It's important to distinguish between operation and punctuation. Parentheses are used around args as punctuation. They are used around a data type in an operation that changes a variable's data type to the one inside the parentheses.

2. A *unary operator* takes a single operand, a *binary operator* takes two operands, and a *ternary operator* takes three operands.

Comments

Commenting code is excellent programming practice. Good comments can help you scan your code more quickly, keep track of what you've done as you build a complex program, and remind you of things you want to add or tune. You can use comments to hide parts of code that you want to save for special situations or keep out of the way while you work on something that might conflict. Comments can help you remember what you were doing when you

return to one project after working on another, or when you come back from vacation. In a team development environment or whenever code is passed between programmers, comments can help others understand the purpose and associations of everything you comment on, without having to parse out every bit of it to be sure they understand.

Java uses three kinds of comments: single-line comments, multi-line comments, and Javadoc comments.

Comment	Tag	Purpose
Single-line	`// ...`	Suitable for brief remarks on the function or structure of a statement or expression. They require only an opening tag: as soon as you start a new line, you're back into code.
Multi-line	`/* ... */`	Good for any comment that will cover more than one line, as when you want to go into some detail about what's happening in the code or when you need to embed legal notices in the code. It requires both opening and closing tags.
Javadoc	`/** ... */`	This is a multi-line comment that the JDK's Javadoc utility can read and turn into HTML documentation. Javadoc has tags you can use to extend its functionality. It's used to provide help for APIs, generate to do lists, and embed flags in code. It requires both opening and closing tags. To learn more about the Javadoc tool, go to Sun's Javadoc page at `http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/`.

Here are some examples:

```
/* You can put as many lines of
   discussion or as many pages of
   boilerplate as you like between
   these two tags.
*/

/* Note that, if you really get carried away,
   you can nest single-line comments
   //inside of the multi-line comments
   and the compiler will have no trouble
   with it at all.
*/

/* Just don't try nesting
   /* multi-line types of comments
   */
   /** of any sort
   */
   because that will generate a
   compiler error.
*/
```

```
/**Useful information about what the code
   does goes in Javadoc tags. Special tags
   such as @todo can be used here to take
   advantage of Javadoc's helpful features.
*/
```

Statements

A statement is a single command. One command can cover many lines of code, but the compiler reads the whole thing as one command. Individual (usually single-line) statements end in a semicolon (`;`), and group (multi-line) statements end in a closing curly brace (`}`). Multi-line statements are generally called *code blocks*.

By default, Java runs statements in the order in which they're written, but Java allows forward references to terms that haven't been defined yet.

Code blocks

A code block is everything between the curly braces, and includes the expression that introduces the curly brace part:

```
class GettingRounder {
  ...
}
```

Understanding scope

Scope rules determine where in a program a variable is recognized. Variables fall into two main scope categorizes:

- Global variables: Variables that are recognized across an entire class.
- Local variables: Variables that are recognized only in the code block where they were declared.

Scope rules are tightly related to code blocks. The one general scope rule is: a variable declared in a code block is visible only in that block and any blocks nested inside it. The following code illustrates this:

```
class Scoping {
  int x = 0;
  void method1() {
    int y;
    y = x;  // This works. method1 can access y.
  }
  void method2() {
    int z = 1;
    z = y;  // This does not work:
            // y is defined outside method2's scope.
  }
}
```

This code declares a class called `scoping`, which has two methods: `method1()` and `method2()`. The class itself is considered the main code block, and the two methods are its nested blocks.

The `x` variable is declared in the main block, so it is *visible* (recognized by the compiler) in both `method1()` and `method2()`. Variables `y` and `z`, on the other hand, were declared in two independent, nested blocks; therefore, attempting to use `y` in `method2()` is illegal since `y` is not visible in that block.

Note A program that relies on global variables can be error-prone for two reasons:

1. Global variables are difficult to keep track of.
2. A change to a global variable in one part of the program can have an unexpected side effect in another part of the program.

Local variables are safer to use since they have a limited life span. For example, a variable declared inside a method can be accessed only from that method, so there is no danger of it being misused somewhere else in the program.

End every simple statement with a semicolon. Be sure every curly brace has a mate. Organize your curly braces in some consistent way (as in the examples above) so you can keep track of the pairs. Many Java IDEs (such as JBuilder) automatically nest the curly braces according to your settings.

Applying concepts

The following sections demonstrate how to apply the terms and concepts introduced earlier in this chapter.

Using operators

Review There are six basic kinds of operators (arithmetic, logical, assignment, comparison, bitwise, and ternary), and operators affect one, two, or three operands, making them unary, binary, or ternary operators. They have properties of precedence and associativity, which determine the order they're processed in.

Operators are assigned numbers that establish their precedence. These numbers vary by circumstances and are determined by a complex set of rules. The higher the number, the higher the order of precedence (that is, the more likely it is to be evaluated sooner than others). An operator of precedence 1 (the lowest) will be evaluated last, and an operator with a precedence of 15 (the highest) will be evaluated first.

Operators with the same precedence are normally evaluated from left to right.

Precedence is evaluated before associativity. For instance, the expression `a + b - c * d` will not be evaluated from left to right; multiplication has precedence over addition, so `c * d` is evaluated first. Addition and subtraction

have the same order of precedence, so associativity applies: `a` and `b` are added next, then the product of `c * d` is subtracted from that sum.

It's good practice to use parentheses around mathematical expressions you want evaluated first, regardless of their precedence, for instance: `(a + b) - (c * d)`. The program will evaluate this operation the same way, but for the human reader, this format is clearer.

See also

- "Expressions, Statements, and Blocks" in the Java Tutorial, at `http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html`

Arithmetic operators

Java provides a full set of operators for mathematical calculations. Java, unlike some languages, can perform mathematical functions on both integer and floating-point values. You will probably find these operators familiar.

Here are the arithmetic operators:

Operator	Definition	Prec.	Assoc.
`++/--`	Auto-increment/decrement: Adds one to, or subtracts one from, its single operand. If the value of `i` is 4, `++i` is 5. See below for more information.	1	Right
`+/-`	Unary plus/minus: sets or changes the positive/ negative value of a single number.	2	Right
`*`	Multiplication.	4	Left
`/`	Division.	4	Left
`%`	Modulus: Divides the first operand by the second operand and returns the remainder. See below for a brief mathematical review.	4	Left
`+/-`	Addition/subtraction	5	Left

Whether you use pre-increment/decrement or post-increment/decrement matters only when you return the initial value of the incremented/decremented variable. Use pre- or post-increment/decrement depending on when you want the new value to be assigned:

```
int y = 3;    //1. variable declaration
int b = 9;    //2.
int x;        //3.
int a;        //4.
x = ++y;      //5. pre-increment: y and x return 4.
a = b--;      //6. post-decrement: b returns 9, a returns 8.
```

In statement 4, `pre-increment`, the `y` variable's value is incremented by 1, and then its new value (4) is assigned to `x`. Both `x` and `y` originally had a value of 3; now they both have the value of 4.

In statement 5, `post-decrement`, `b`'s current value (9) is assigned to `a` and *then* the value of `b` is decremented (to 8). `b` originally had a value of 9 and `a` had no value assigned; now `a` is 9 and `b` is 8.

The modulus operator requires an explanation to those who last studied math a long time ago. Remember that when you divide two numbers, they rarely divide evenly. What is left over after you have divided the numbers (without adding any new decimal places) is the *remainder*. For instance, 3 goes into 5 once, with 2 left over. The remainder (in this case, 2) is what the modulus operator evaluates for. Since remainders recur in a division cycle on a predictable basis (for instance, an hour is modulus 60), the modulus operator is particularly useful when you want to tell a program to repeat a process at specific intervals.

Logical operators

Logical (or Boolean) operators allow the programmer to group boolean expressions to determine certain conditions. These operators perform the standard Boolean operations AND, OR, NOT, and XOR.

The following table lists the logical operators:

Operator	Definition	Prec.	Assoc.
!	Boolean NOT (unary) Changes true to false or false to true. Because of its low precedence, you may need to use parentheses around this statement.	2	Right
&	Evaluation AND (binary) Yields true only if both operands are true. Always evaluates both operands. Rarely used as a logical operator.	9	Left
^	Evaluation XOR (binary) Yields true if only one operand is true. Evaluates both operands.	10	Left
|	Evaluation OR (binary) Yields true if one or both of the operands is true. Evaluates both operands.	11	Left
&&	Conditional AND (binary) Yields true only if both operands are true. Called “conditional” because it only evaluates the second operand if the first operand is true.	12	Left
||	Conditional OR (binary) Yields true if either one or both operands is true; returns false if both are false. Doesn’t evaluate second operand if first operand is true.	13	Left

The evaluation operators always evaluate both operands. The conditional operators, on the other hand, always evaluate the first operand, and if that

determines the value of the whole expression, they don't evaluate the second operand. For example:

```
if ( !isHighPressure && (temperature1 > temperature2)) {
  ...
}                                   //Statement 1: conditional

boolean1 = (x < y) || ( a > b);     //Statement 2: conditional

boolean2 = (10 > 5) & (5 > 1);      //Statement 3: evaluation
```

The first statement evaluates `!isHighPressure` first. If `!isHighPressure` is `false` (that is, if the pressure *is* high; note the logical double-negative of `!` and `false`), the second operand, `temperature1 > temperature2`, doesn't need to be evaluated. `&&` only needs one `false` value in order to know what value to return.

In the second statement, the value of `boolean1` will be `true` if `x` is less than `y`. If `x` is more than `y`, the second expression will be evaluated; if `a` is less than `b`, the value of `boolean1` will still be `true`.

In the third statement, however, the compiler will compute the values of both operands before assigning `true` or `false` to `boolean2`, because `&` is an evaluation operator, not a conditional one.

Assignment operators

You know that the basic assignment operator (`=`) lets you assign a value to a variable. With Java's set of assignment operators, you can perform an operation on either operand and assign the new value to a variable in one step.

The following table lists assignment operators:

Operator	Definition	Prec.	Assoc.
`=`	Assign the value on the right to the variable on the left.	15	Right
`+=`	Add the value on the right to the value of the variable on the left; assign the new value to the original variable.	15	Right
`-=`	Subtract the value on the right from the value of the variable on the left; assign the new value to the original variable.	15	Right
`*=`	Multiply the value on the right with the value of the variable on the left; assign the new value to the original variable.	15	Right
`/=`	Divide the value on the right from the value of the variable on the left; assign the new value to the original variable.	15	Right

The first operator is familiar by now. The rest of the assignment operators perform an operation first, and then store the result of the operation in the operand on the left side of the expression. Here are some examples:

```
int y = 2;
y *= 2;  //same as (y = y * 2)

boolean b1 = true, b2 = false;
b1 &= b2;  //same as (b1 = b1 & b2)
```

Comparison operators

Comparison operators allow you to compare one value to another.

The following table lists the comparison operators:

Operator	Definition	Assoc.
<	Less than	Left
>	Greater than	Left
<=	Less than or equal to	Left
>=	Greater than or equal to	Left
==	Equal to	Left
!=	Not equal to	Left

The equality operator can be used to compare two object variables of the same type. In this case, the result of the comparison is true only if both variables refer to the same object. Here is a demonstration:

```
m1 = new Mammal();
m2 = new Mammal();
boolean b1 = m1 == m2;  //b1 is false

m1 = m2;
boolean b2 = m1 == m2;  //b2 is true
```

The result of the first equality test is false because `m1` and `m2` refer to different objects (even though they are of the same type). The second comparison is true because both variables now represent the same object.

Note Most of the time, however, the `equals()` method in the `Object` class is used instead of the comparison operator. The comparing class must be subclassed from `Object` before objects of the comparing class can be compared using `equals()`.

Bitwise operators

Bitwise operators are of two types: shift operators and Boolean operators. The shift operators are used to shift the binary digits of an integer to the right or the left. Consider the following example (the `short` integer type is used instead of `int` for conciseness):

```
short i = 13; //i is 0000000000001101
i = i << 2;   //i is 0000000000110100
```

In the second line, the bitwise left shift operator shifted all the bits of `i` two positions to the left.

Both of these are positive numbers. They can be represented from the first 1, like this: 1101, 110100. Unless otherwise specified, the signed bit is assumed to be positive: 0...1101, 0...110100. To reverse the sign of a binary number,

1 Reverse all the bits:

0000000000001101 becomes
1111111111110010

2 Then add 1:

1111111111110011

Note The shifting operation is different in Java than in C/C++ in how it is used with *signed* integers. A signed integer is one whose left-most bit is used to indicate the integer's positive or negative sign: the bit is `1` if the integer is negative, `0` if positive. In Java, integers are always signed, whereas in C/C++ they are signed only by default. In most implementations of C/C++, a bitwise shift operation does not preserve the integer's sign; the sign bit would be shifted out. In Java, however, the shift operators preserve the sign bit (unless you use the `>>>` to perform an *unsigned shift*). This means that the sign bit is duplicated, then shifted.

The following table lists Java's bitwise operators:

Operator	Definition	Assoc.
`~`	Bitwise `NOT` Inverts each bit of the operand, so each 0 becomes 1 and vice versa.	Right
`<<`	Signed left shift Shifts the bits of the left operand to the left, by the number of digits specified in the right operand, with 0's shifted in from the right. High-order bits are lost.	Left
`>>`	Signed right shift Shifts the bits of the left operand to the right, by the number of digits specified on the right. If the left operand is negative, 0's are shifted in from the left; if it is positive, 1's are shifted in. This preserves the original sign.	Left
`>>>`	Zero-fill right shift Shifts right, but always fills in with 0's.	Left
`&`	Bitwise `AND` Can be used with `=` to assign the value.	Left
`|`	Bitwise `OR` Can be used with `=` to assign the value.	Left
`^`	Bitwise `XOR` Can be used with `=` to assign the value.	Left
`<<=`	Left-shift with assignment	Left
`>>=`	Right-shift with assignment	Left
`>>>=`	Zero-fill right shift with assignment	Left

?:, the ternary operator

`?:` is a ternary operator that Java borrowed from C. It provides a handy shortcut to create a very simple if-then-else kind of statement.

This is the syntax:

```
<expression 1, a Boolean condition> ? <expression 2> : <expression 3>;
```

The Boolean condition, `expression 1`, is evaluated first. If it resolves `true` or if its resolution depends on the rest of the ternary statement, then `expression 2` is evaluated. If `expression 2` is false, `expression 3` is used. For example:

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Here, `max` is assigned the value of `x` or `y`, depending on which is greater. The value of `x` is not greater than the value of `y`, so the value of `y` is assigned to `max`.

Using methods

You know that methods are what get things done. Methods cannot contain other methods, but they can contain variables and class references.

Here is a brief example to review. This method helps a music store with its inventory:

```
//Declare the method: return type, name, args:
public int getTotalCDs(int numRockCDs, int numJazzCDs, int numPopCDs) {
  //Declare the variable totalCDs. The other three variables are declared
        elsewhere:
  int totalCDs = numRockCDs + numJazzCDs + numPopCDs;
  //Make it do something useful. In this case, print this line on
    the screen:
  System.out.println("Total CDs in stock = " + totalCDs);
}
```

In Java, you can define more than one method with the same name, as long as the different methods require different arguments. For instance, both `public int getTotalCDs(int numRockCDs, int numJazzCDs, in numPopCDs)` and `public int getTotalCDs(int salesRetailCD, int salesWholesaleCD)` are legal in the same class. Java will recognize the different patterns of arguments (the *method signatures*) and apply the correct method when you make a call. Assigning the same name to different methods is called *method overloading*.

To access a method from other parts of a program, you must first create an instance of the class the method resides in, and then use that object to call the method:

```
//Create an instance totalCD of the class Inventory:
Inventory totalCD = new Inventory();

//Access the getTotalCDs() method inside of Inventory, storing the value
      in total:
int total = totalCD.getTotalCDs(myNumRockCDs, myNumJazzCDs, myNumPopCDs);
```

Using arrays

Note that the size of an array is not part of its declaration. The memory an array requires is not actually allocated until you initialize the array.

To initialize the array (and allocate the needed memory), you must use the `new` operator as follows:

```
int studentID[] = new int[20];            //Creates array of 20 int
elements.
char[] grades = new char[20];             //Creates array of 20 char
elements.
float[][] coordinates = new float[10][5]; //2-dimensional, 10x5 array
                                            of float elements.
```

Note In creating two-dimensional arrays, the first array number defines number of rows and the second array number defines number of columns.

Java counts positions starting with 0. This means the elements of a 20-element array will be numbered from 0 to 19: the first element will be 0, the second will be 1, and so on. Be careful how you count when you're working with arrays.

When an array is created, the value of all its elements is `null` or 0; values are assigned later.

Note The use of the `new` operator in Java is similar to that of the `malloc` command in C and the `new` operator in C++.

To initialize an array, specify the values of the array elements inside a set of curly braces. For multi-dimensional arrays, use nested curly braces. For example:

```
char[] grades = {'A', 'B', 'C', 'D', 'F');
float[][] coordinates = {{0.0, 0.1}, {0.2, 0.3}};
```

The first statement creates a `char` array called `grades`. It initializes the array's elements with the values `'A'` through `'F'`. Notice that we did not have to use the `new` operator to create this array; by initializing the array, enough memory is automatically allocated for the array to hold all the initialized values. Therefore, the first statement creates a `char` array of 5 elements.

The second statement creates a two-dimensional `float` array called `coordinates`, whose size is 2 by 2. Basically, `coordinates` is an array consisting of two array elements: the array's first row is initialized to `0.0` and `0.1`, and the second row to `0.2` and `0.3`.

Using constructors

A *class* is a full piece of code, enclosed in a pair of curly braces, that defines a logically coherent set of variables, attributes, and actions. A *package* is a logically associated set of classes.

Note that a class is just a set of instructions. It doesn't do anything itself. It's analogous to a recipe: you can make a cake from the right recipe, but the

recipe is not the cake, it's only the instructions for it. The cake is an *object* you have created from the instructions in the recipe. In Java, we would say that we have created an *instance* of cake from the recipe Cake.

The act of creating an instance of a class is called *instantiating* that object. You *instantiate* an *object* of a *class*.

To instantiate an object, use the assignment operator (`=`), the keyword `new`, and a special kind of method called a *constructor*. A call to a constructor is the name of the class being instantiated followed by a pair of parentheses. Although it looks like a method, it takes a class's name; that's why it's capitalized:

```
<ClassName> <instanceName> = new <Constructor()>;
```

For example, to instantiate a new object of the `Geek` class and name the instance `thisProgrammer`:

```
Geek thisProgrammer = new Geek();
```

A constructor sets up a new instance of a class: it initializes all the variables in that class, making them immediately available. It can also perform any start-up routines required by the object.

For example, when you need to drive your car, the first thing you do is open the door, climb in, put the clutch in, and start the engine. (After that, you can do all the things normally involved in driving, like getting into gear and using the accelerator.) The constructor handles the programmatic equivalents of the actions and objects involved in getting in and starting the car.

Once you have created an instance, you can use the instance name to access members of that class.

For more information on constructors, see "Case study: A simple OOP example" on page 74.

Member access

The access operator (**.**) is used to access members inside of an instantiated object. The basic syntax is:

```
<instanceName>.<memberName>
```

Precise syntax of the member name depends on the kind of member. These can include variables (`<memberName>`), methods (`<memberName>()`), or subclasses (`<MemberName>`).

You can use this operation inside of other syntax elements wherever you need to access a member. For example:

```
setColor(Color.pink);
```

This method needs a color to do its job. The programmer used an access operation as an arg to access the variable `pink` within the class `Color`.

Arrays

Array elements are accessed by subscripting, or *indexing*, the array variable. To index an array variable, follow the array variable's name with the element's number (index) surrounded by square brackets. *Arrays are always indexed starting from 0.* If you have an array with 9 elements, the first element is the 0 index and the last element is the 8 index. (Coding as if elements were numbered from 1 is a common mistake.)

In the case of multi-dimensional arrays, you must use an index for each dimension to access an element. The first index is the row and the second index is the column.

For example:

```
firstElement = grades[0];   //firstElement = 'A'
fifthElement = grades[4];   //fifthElement = 'F'
row2Col1 = coordinates[1][0]; //row2Col1 = 0.2
```

The following snippet of code demonstrates one use of arrays. It creates an array of 5 `int` elements called `intArray`, then uses a `for` loop to store the integers 0 through 4 in the elements of the array:

```
int[] intArray = new int [5];
int index;
for (index = 0; index < 5; index++) intArray [index] = index;
```

This code increments the `index` variable from `0` to `4`, and at every pass, it stores its value in the element of `intArray` indexed by the variable `index`.

Chapter

4

Java language control

This section provides you with foundational concepts about control of the Java programming language that will be used throughout this chapter. It assumes you understand general programming concepts, but have little or no experience with Java.

Terms

The following terms and concepts are discussed in this chapter:

- String handling
- Type casting and conversion
- Return types and statements
- Flow control statements

String handling

The `String` class provides methods that allow you to get substrings or to *index* characters within a string. However, the value of a declared `String` can't be changed. If you need to change the `String` value associated with that variable, you must point the variable to a new value:

```
String text1 = new String("Good evening."); // Declares text1 and assigns
                                                a value.
text1 = "Hi, honey, I'm home!"              // Assigns a new value to
                                                text1.
```

Indexing allows you to point to a particular character in a string. Java counts each position in a string starting from 0, so that the first position is 0, the second position is 1, and so on. This gives the eighth position in a string an index of 7.

The `StringBuffer` class provides a workaround. It also offers several other ways to manipulate a string's contents. The `StringBuffer` class stores your string in a buffer (a special area of memory) whose size you can explicitly control; this allows you to change the string as much as necessary before you have to declare a `String` and make the string permanent.

Generally, the `String` class is for string storage and the `StringBuffer` class is for string manipulation.

Type casting and conversion

Values of data types can be converted from one type to another. Class values can be converted from one class to another in the same class hierarchy. Note that conversion does not change the original type of that value, it only changes the compiler's perception of it for that one operation.

Obvious logical restrictions apply. A widening conversion—from a smaller type to a larger type—is easy, but a narrowing conversion—converting from a larger type (for instance, `double` or `Mammal`) to a smaller type (for instance, `float` or `Bear`)—risks your data, unless you're certain that your data will fit into the parameters of the new type. A narrowing conversion requires a special operation called a *cast*.

The following table shows widening conversions of primitive values. These won't risk your data:

Original Type	Converts to Type
`byte`	`short, char, int, long, float, double`
`short`	`int, long, float, double`
`char`	`int, long, float, double`
`int`	`long, float, double`
`long`	`float, double`
`float`	`double`

To cast a data type, put the type you want to cast *to* in parentheses immediately before the variable you want to cast: `(int)x`. This is what it looks like in context, where `x` is the variable being cast, `float` is the original data type, `int` is the target data type, and `y` is the variable storing the new value:

```
float x = 1.00;    //declaring x as a float
int y = (int)x; //casting x to an int named y
```

This assumes that the value of `x` would fit inside of `int`. Note that `x`'s decimal values are lost in the conversion. Java rounds decimals down to the nearest whole number.

Return types and statements

You know that a method declaration requires a return type, just as a variable declaration requires a data type. The return types are the same as the data types (`int`, `boolean`, `String`, and so on), with the exception of `void`.

`void` is a special return type. It signifies that the method doesn't need to give anything back when it's finished. It is most commonly used in action methods that are only required to do something, not to pass any information on.

All other return types require a `return` statement at the end of the method. You can use the `return` statement in a `void` method to leave the method at a certain point, but otherwise it's needless.

A `return` statement consists of the word `return` and the string, data, variable name, or concatenation required:

```
return numCD;
```

It's common to use parentheses for concatenations:

```
return ("Number of files: " + numFiles);
```

Flow control statements

Flow control statements tell the program how to order and use the information that you give it. With flow control, you can reiterate statements, conditionalize statements, create recursive loops, and control loop behavior.

Flow control statements can be grouped into three kinds of statements: *iteration* statements such as `for`, `while`, and `do-while`, which create loops; *selection* statements such as `switch`, `if`, `if-else`, `if-then-else`, and `if-else-if` ladders, which conditionalize the use of statements; and the *jump* statements `break`, `continue`, and `return`, which shift control to another part of your program.

A special form of flow control is *exception handling*. Exception handling provides a structured means of catching runtime errors in your program and making them return meaningful information about themselves. You can also set the exception handler to perform certain actions before allowing the program to terminate.

Applying concepts

The following sections demonstrate how to apply the terms and concepts introduced earlier in this chapter.

Escape sequences

A special type of character literal is called an *escape sequence*. Like C/C++, Java uses escape sequences to represent special control characters and characters that cannot be printed. An escape sequence is represented by a

backslash (\) followed by a character code. The following table summarizes these escape sequences:

Character	Escape Sequence
Backslash	\\
Backspace	\b
Carriage return	\r
Double quote	\"
Form feed	\f
Horizontal tab	\t
New line	\n
Octal character	\DDD
Single quote	\'
Unicode character	\uHHHH

Non-decimal numeric characters are escape sequences. An octal character is represented by three octal digits, and a Unicode character is represented by lowercase `u` followed by four hexadecimal digits. For example, the decimal number 57 is represented by the octal code `\071` and the Unicode sequence `\u0039`.

The sample string in the following statement prints out the words `Name` and `"Hildegaard von Bingen"` separated by two tabs on one line, and prints out `ID` and `"1098"`, also separated by two tabs, on the second line:

```
String escapeDemo = new
  String("Name\t\t\"Hildegaard von Bingen\"\nID\t\t\"1098\"");
```

Strings

The string of characters you specify in a `String` is a literal; the program will use exactly what you specify, without changing it in any way. However, the `String` class provides the means to chain strings together (called *string concatenation*), see and use what's inside of strings (compare strings, search strings, or extract a substring from a string), and convert other kinds of data to strings. Some examples follow:

- Declare variables of the `String` type and assign values:

```
String firstNames = "Joseph, Elvira and Hans";
String modifier = " really ";
String tastes = "like chocolate.";
```

- Get a substring from a string, selecting from the ninth column to the end of the string:

```
String sub = firstNames.substring(8); // "Elvira and Hans"
```

- Compare part of the substring to another string, convert a string to capital letters, then concatenate it with other strings to get a return value:

```
boolean bFirst = firstNames.startsWith("Emine"); // Returns false in
                                                    this case.
String caps = modifier.toUpperCase();          // Yields " REALLY "
return firstNames + caps + tastes;             // Returns the line:
                                               // Elvira and Hans
                                                  REALLY like chocolate.
```

For more information on how to use the `String` class, see Sun's API documentation at `http://java.sun.com/j2se/1.4/docs/api/java/lang/String.html`.

StringBuffer

If you want more control over your strings, use the `StringBuffer` class. This class is part of the `java.lang` package.

`StringBuffer` stores your strings in a buffer so that you don't have to declare a permanent `String` until you need it. Some of the advantages to this are that you don't have to redeclare a `String` if its content changes. You can reserve a size for the buffer larger than what is already in there.

`StringBuffer` provides methods in addition to those in `String` that allow you to modify the contents of strings in new ways. For instance, `StringBuffer`'s `setCharAt()` method changes the character at the index specified in the first parameter, to the new value specified in the second parameter:

```
StringBuffer word = new StringBuffer ("yellow");
word.setCharAt (0, 'b');  //word is now "bellow"
```

Determining access

By default, classes are available to all of the members inside them, and the members within the class are available to each other. However, this access can be widely modified.

Access modifiers determine how visible a class's or member's information is to other members and classes. Access modifiers include:

- `public`: A public member is visible to members outside the public member's scope, as long as the parent class is visible. A public class is visible to all other classes in all other packages.
- `private`: A private member's access is limited to the member's own class.
- `protected`: A protected member can be accessed by other members of its class and by members of classes in the same package (as long as the member's parent class is accessible), but not from other packages. A protected class is available to other classes in the same package, but not to other packages.
- If no access modifier is declared, the member is available to all classes inside the parent package, but not outside the package.

Let's look at this in context:

```
class Waistline {
  private boolean invitationGiven = false;  // This is private.
  private int weight = 170;                 // So is this.

  public void acceptInvitation() {          // This is public.
    invitationGiven = true;
  }

  //Class JunkFood is declared and object junkFood is
    instantiated elsewhere:
  public void eat(JunkFood junkFood) {

    /*This object only accepts more junkFood if it has an invitation
    * and if it is able to accept. Notice that isAcceptingFood()
    * checks to see if the object is too big to accept more food:
    */
    if (invitationGiven && isAcceptingFood()) {

      /*This object's new weight will be whatever its current weight
      * is, plus the weight added by junkFood. Weight increments
      * as more junkFood is added:
      */
      weight += junkFood.getWeight();
    }
  }

  //Only the object knows if it's accepting food:
  private boolean isAcceptingFood() {
    // This object will only accept food if there's room:
    return (isTooBig() ? false : true);
  }

  //Objects in the same package can see if this object is too big:
  protected boolean isTooBig() {
    //It can accept food if its weight is less than 185:
    return (weight > 185) ? true : false;
  }
}
```

Notice that `isAcceptingFood()` and `invitationGiven` are `private`. Only members inside this class know if this object is capable of accepting food or if it has an invitation.

`isTooBig()` is `protected`. Only classes inside this package can see if this object's weight exceeds its limit or not.

The only methods that are exposed to the outside are `acceptInvitation()` and `eat()`. Any class can perceive these methods.

Handling methods

The `main()` method deserves special attention. It is the point of entry into a program (except an applet). It's written like this:

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

There are specific variations allowed inside the parentheses, but the general form is consistent.

The keyword `static` is important. A `static` method is always associated with its entire class, rather than with any particular instance of that class. (The keyword `static` can also be applied to classes. All of the members of a `static` class are associated with the class's entire parent class.) `static` methods are also called *class methods*.

Since the `main()` method is the starting-point within the program, it must be `static` in order to remain independent of the many objects the program may generate from its parent class.

`static`'s class-wide association affects how you call a `static` method and how you call other methods from within a `static` method. `static` members can be called from other types of members by simply using the name of the method, and `static` members can call each other the same way. You don't need to create an instance of the class in order to access a `static` method within it.

To access non`static` members of a non`static` class from within a `static` method, you must instantiate the class of the member you want to reach and use that instance with the access operator, just as you would for any other method call.

Notice that the arg for the `main()` method is a `String` array, with other args allowed. Remember that this method is where the compiler starts working. When you pass an arg from the command line, it's passed as a string to the `String` array in the declaration of the `main()` method, and uses that arg to start running the program. When you pass a data type other than a `String`, it will still be received as a string. You must code into the body of the `main()` method the required conversion from `String` to the data type needed.

Using type conversions

Review Type conversion is the process of converting the data type of a variable for the duration of a specific operation. The standard form for a narrowing conversion is called a cast; it may risk your data.

Implicit casting

There are times when a cast is performed implicitly by the compiler. The following is an example:

```
if (3 > 'a') {
  ...
}
```

In this case, the value of `'a'` is converted to an integer value (the ASCII value of the letter `a`) before it is compared with the number `3`.

Explicit conversion

Syntax for a widening cast is simple:

```
<nameOfOldValue> = (<new type>) <nameOfNewValue>
```

Java doesn't want you to make a narrowing conversion, so you must be more explicit when doing so:

```
floatValue = (float)doubValue; // To float "floatValue"
                               // from double "doubValue".

longValue = (long)floatValue;  // To long "longValue"
                               // from float "floatValue".
                               // This is one of four possible
                                  constructions.
```

(Note that decimals are rounded down by default.) Be sure you thoroughly understand the syntax for the types you want to cast; this process can get messy.

For more information, see "Converting and casting data types" on page 125.

Flow control

Review

There are three types of loop statements: iteration statements (`for`, `while`, and `do-while`) create loops, selection statements (`switch` and all the `if` statements) tell the program under what circumstances the program will use statements, and jump statements (`break`, `continue`, and `return`) shift control out to another part of the program.

Loops

Each statement in a program is executed once. However, it is sometimes necessary to execute statements several times until a condition is met. Java provides three ways to loop statements: `while`, `do` and `for` loops.

- **The while loop**

  The `while` loop is used to create a block of code that will execute as long as a particular condition is met. This is the general syntax of the `while` loop:

  ```
  while ( <boolean condition statement> ) {
    <code to execute as long as that condition is true>
  }
  ```

  The loop first checks the condition. If the condition's value is `true`, it executes the entire block. It then reevaluates the condition, and repeats this process until the condition becomes `false`. At that point, the loop stops executing. For instance, to print "Looping" 10 times:

  ```
  int x = 0;                          //Initiates x at 0.
  while (x < 10){                     //Boolean condition statement.
    System.out.println("Looping");  //Prints "Looping" once.
    x++;                              //Increments x for the next iteration.
  }
  ```

  When the loop first starts executing, it checks whether the value of `x` is less than `10`. Since it is, the body of the loop is executed. In this case, the word "`Looping`" is printed on the screen, and then the value of `x` is incremented. This loop continues until the value of `x` equals 10, when the loop stops executing.

  Unless you intend to write an infinite loop, make sure there is some point in the loop where the condition's value becomes `false` and the loop terminates. You can also terminate a loop by using the `return`, `continue`, or `break` statements.

- **The do-while loop**

  The `do-while` loop is similar to the `while` loop, except that it evaluates the condition *after* the statements instead of before. The following code shows the previous `while` loop converted to a `do` loop:

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

  The main difference between the two loop constructs is that the `do-while` loop is always going to execute at least once, but the `while` loop won't execute at all if the initial condition is not met.

- **The for loop**

  The `for` loop is the most powerful loop construct. Here is the general syntax of a `for` loop:

```
for ( <initialization> ; <boolean condition> ; <iteration> ) {
  <execution code>
}
```

  The `for` loop consists of three parts: an initialization expression, a Boolean condition expression, and an iteration expression. The third expression usually updates the loop variable initialized in the first expression. Here is the `for` loop equivalent of the previous `while` loop:

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

  This `for` loop and its equivalent `while` loop are very similar. For almost every `for` loop, there is an equivalent `while` loop.

  The `for` loop is the most versatile loop construct, but still very efficient. For example, a `while` loop and a `for` loop can both add the numbers one through twenty, but a `for` loop can do it in one line less.

  While:

```
int x = 1, z = 0;
while (x <= 20) {
  z += x;
  x++;
}
```

  For:

```
int z = 0;
for (int x=1; x <= 20; x++) {
  z+= x;
}
```

  We can tweak the `for` loop to make the loop execute half as many times:

```
for (int x=1,y=20, z=0; x<=10 && y>10; x++, y--) {
  z+= x+y;
}
```

  Let's break this loop up into its four main sections:

  a The initialization expression: `int x =1, y=20, z=0`

  b The Boolean condition: `x<=10 && y>10`

  c The iteration expression: `x++, y--`

  d The main body of executable code: `z+= x + y`

Loop control statements

These statements add control to the loop statements.

- **The break statement**

  The `break` statement will allow you to exit a loop structure before the test condition is met. Once a `break` statement is encountered, the loop immediately terminates, skipping any remaining code. For instance:

```
int x = 0;
while (x < 10){
  System.out.println("Looping");
  x++;
  if (x == 5)
    break;
  else
    ... //do something else
}
```

  In this example, the loop will stop executing when `x` equals `5`.

- **The continue statement**

  The `continue` statement is used to skip the rest of the loop and resume execution at the next loop iteration.

```
for ( int x = 0 ; x < 10 ; x++){
  if(x == 5)
    continue; //go back to beginning of loop with x=6
  System.out.println("Looping");
}
```

  This example will not print "`Looping`" if `x` is 5, but will continue to print for 6, 7, 8, and 9.

Conditional statements

Conditional statements are used to provide your code with decision-making capabilities. There are two conditional structures in Java: the `if-else` statement, and the `switch` statement.

- **The if-else statement**

  The syntax of an `if-else` statement is as follows:

```
if (<condition1>) {
  ... //code block 1
}
else if (<condition2>) {
  ... //code block 2
}
else {
  ... //code block 3
}
```

The `if-else` statement is typically made up of multiple blocks. Only one of the blocks will execute when the `if-else` statement executes, based on which of the conditions is true.

The `else-if` and `else` blocks are optional. Also, the `if-else` statement is not restricted to three blocks: it can contain as many `else-if` blocks as needed.

The following examples demonstrate the use of the `if-else` statement:

```
if ( x % 2 == 0)
  System.out.println("x is even");
else
  System.out.println("x is odd");
if (x == y)
  System.out.println("x equals y");
else if (x < y)
  System.out.println("x is less than y");
else
  System.out.println("x is greater than y");
```

- **The switch statement**

  The `switch` statement is similar to the `if-else` statement. Here is the general syntax of the `switch` statement:

```
switch (<expression>){
  case <value1>: <codeBlock1>;
      break;
  case <value2>: <codeBlock2>;
      break;
  default    : <codeBlock3>;
}
```

  Note the following:

  - If there is only one statement in a code block, the block does not need to be enclosed in braces.
  - The `default` code block corresponds to the `else` block in an `if-else` statement.
  - The code blocks are executed based on the value of a variable or expression, not on a condition.
  - The value of `<expression>` must be of an integer type, or a type that can be safely converted to `int`, such as `char`.
  - The `case` values must be constant expressions that are of the same data type as the original `expression`.
  - The `break` keyword is optional. It is used to end the execution of the `switch` statement once a code block executes. If it's not used after `codeBlock1`, then `codeBlock2` executes right after `codeBlock1` finishes executing.
  - If a code block should execute when `expression` is one of a number of values, each of the values must be specified like this: `case <value>:`.

Here is an example, where `c` is of type `char`:

```
switch (c){
  case '1': case '3': case '5': case '7': case '9':
    System.out.println("c is an odd number");
    break;
  case '0': case '2': case '4': case '6': case '8':
    System.out.println("c is an even number");
    break;
  case ' ':
    System.out.println("c is a space");
    break;
  default :
    System.out.println("c is not a number or a space");
  }
```

The `switch` will evaluate `c` and jump to the `case` statement whose value is equal to `c`. If none of the `case` values equal `c`, the `default` section will be executed. Notice how multiple values can be used for each block.

Handling exceptions

Exception handling provides a structured means of catching run-time errors in your program and making them return meaningful information about themselves. You can also set the exception handler to perform certain actions before allowing the program to terminate. Exception handling uses the keywords `try`, `catch`, and `finally`. A method can declare an exception by using the `throws` and `throw` keywords.

In Java, an exception can be a subclass of the class `java.lang.Exception` or `java.lang.Error`. When a method declares that an exception has occurred, we say that it *throws* an exception. To *catch* an exception means to handle an exception.

Exceptions that are explicitly declared in the method declaration *must* be caught, or the code will not compile. Exceptions that are not explicitly declared in the method declaration could still halt your program when it runs, but it will compile. Note that good exception handling makes your code more robust.

To catch an exception, you enclose the code which might cause the exception in a `try` block, then enclose the code you want to use to handle the exception in a `catch` block. If there is important code (such as clean-up code) that you want to make sure will run even if an exception is thrown and the program gets shut down, enclose that code in a `finally` block at the end. Here is an example of how this works:

```
try {
  ... // Some code that might throw an exception goes here.
}
catch( Exception e ) {
  ... // Exception handling code goes here.
  // This next line outputs a stack trace of the exception:
  e.printStackTrace();
}
```

```
finally {
  ... // Code in here is guaranteed to be executed,
      // whether or not an exception is thrown in the try block.
}
```

The `try` block should be used to enclose any code that might throw an exception that needs to be handled. If no exception is thrown, all of the code in the `try` block will execute. If, however, an exception is thrown, then the code in the `try` block stops executing at the point where the exception is thrown and the control flows to the `catch` block, where the exception is handled.

You can do whatever you need to do to handle the exception in one or more `catch` blocks. The simplest way to handle exceptions is to handle all of them in one `catch` block. To do this, the argument in parentheses after `catch` should indicate the class `Exception`, followed by a variable name to assign to this exception. This indicates that any exception which is an instance of `java.lang.Exception` or any of its subclasses will be caught; in other words, any exception.

If you need to write different exception handling code depending on the type of exception, you can use more than one `catch` block. In that case, instead of passing `Exception` as the type of exception in the `catch` argument, you indicate the class name of the specific type of exception you want to catch. This may be any subclass of `Exception`. Keep in mind that the catch block will always catch the indicated type of exception and any of its subclasses.

Code in the `finally` block is guaranteed to be executed, even if the `try` block code does not complete for some reason. For instance, the code in the `try` block might not complete if it throws an exception, but the code in the `finally` block will still execute. This makes the `finally` block a good place to put clean-up code.

If you know that a method you're writing is going to be called by other code, you might leave it up to the calling code to handle the exception that your method might throw. In that case, you would simply declare that the method can throw an exception. Code that might throw an exception can use the `throws` keyword to declare an exception. This can be an alternative to catching the exception, since if a method declares that it throws an exception, it does not have to handle that exception.

Here is an example of using `throws`:

```
public void myMethod() throws SomeException {
  ... // Code here might throw SomeException, or one of its subclasses.
      // SomeException is assumed to be a subclass of Exception.
}
```

You can also use the `throw` keyword to indicate that something has gone wrong. For instance, you might use this to throw an exception of your own when a user has entered invalid information and you want to show them an error message. To do this, you would use a statement like:

```
throw new SomeException("invalid input");
```

Chapter

5

The Java class libraries

Most programming languages rely on pre-built libraries of classes to support certain functionality. In the Java language, these groups of related classes called packages vary by Java edition. Each edition is used for specific purposes, such as applications, enterprise applications, and consumer products.

Java 2 Platform editions

The Java 2 Platform is available in several editions used for various purposes. Because Java is a language that can run anywhere and on any platform, it is used in a variety of environments: Internet, intranets, consumer electronic products, and computer applications. Due to Java's varied applications, it has been packaged in several editions: Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE), and Java 2 Micro Edition (J2ME). In some cases, as in the development of enterprise applications, a larger set of packages is used. In other cases, as in consumer electronic products, only a small portion of the language is used. Each edition contains a Java 2 Software

Development Kit (SDK) used to develop applications and a Java 2 Runtime Environment (JRE) used to run applications.

Table 5.1 Java 2 Platform editions

Java 2 Platform	Abbreviation	Description
Standard Edition	J2SE	Contains classes that are the core of the Java language.
Enterprise Edition	J2EE	Contains J2SE classes and additional classes for developing enterprise applications.
Micro Edition	J2ME	Contains a subset of J2SE classes and is used in consumer electronic products.

Standard Edition

The Java 2 Platform, Standard Edition (J2SE) provides developers with a feature-rich, stable, secure, cross-platform development environment. This Java edition supports such core features as database connectivity, user interface design, input/output, and network programming and includes the fundamental packages of the Java language.

See also

- Java 2 Platform Standard Edition Overview at `http://java.sun.com/j2se/1.4/`
- "Introducing the Java Platform" at `http://developer.java.sun.com/developer/onlineTraining/new2java/programming/intro/`
- "Java 2 Standard Edition packages" on page 45

Enterprise Edition

The Java 2, Enterprise Edition (J2EE) provides the developer with tools to build and deploy multitier enterprise applications. J2EE includes the J2SE packages as well as additional packages which support Enterprise JavaBeans development, Java servlets, JavaServer Pages, XML, and flexible transaction control.

See also

- "Java 2 Platform Enterprise Edition Overview" at `http://java.sun.com/j2ee/overview.html`
- Java 2 Enterprise Edition technical articles at `http://developer.java.sun.com/developer/technicalArticles/J2EE/index.html`

Micro Edition

The Java 2, Micro Edition (J2ME) is used in a variety of consumer electronic products, such as pagers, smart cards, cell phones, hand-held PDAs, and set-top boxes. While J2ME provides the same Java language advantages of code portability across platforms, the ability to run anywhere, and safe network delivery as J2SE and J2EE, it uses a smaller set of packages. J2ME includes a subset of the J2SE packages with an additional package specific to the Micro Edition, `javax.microedition.io`. In addition, J2ME applications are upwardly scalable to work with J2SE and J2EE.

See also

- "Java 2 Platform Micro Edition Overview" at `http://java.sun.com/j2me/`
- Consumer & Embedded Products technical articles at `http://developer.java.sun.com/developer/technicalArticles/ConsumerProducts/index.html`

Java 2 Standard Edition packages

The Java 2 Platform, Standard Edition (J2SE) comes with a very impressive library that includes support for database connectivity, user interface design, input and output (I/O), and network programming. These libraries are organized into groups of related classes called packages. The following table briefly describes some of these packages.

Table 5.2 J2SE packages

Package	Package Name	Description
Language	`java.lang`	Classes that contain the main core of the Java language.
Utilities	`java.util`	Support for utility data structures.
I/O	`java.io`	Support for various types of input/output.
Text	`java.text`	Localization support for handling text, dates, numbers, and messages.
Math	`java.math`	Classes for performing arbitrary-precision integer and floating-point arithmetic.
AWT	`java.awt`	User interface design and event-handling.
Swing	`javax.swing`	Classes for creating all-Java, lightweight components that behave similarly on all platforms.
Javax	`javax`	Extensions to the Java language.
Applet	`java.applet`	Classes for creating applets.
Beans	`java.beans`	Classes for developing JavaBeans.
Reflection	`java.lang.reflect`	Classes used to obtain runtime class information.

Table 5.2 J2SE packages (continued)

Package	Package Name	Description

XML processing	`org.w3c.dom`
	`org.xml.sax`
	`javax.xml.transform`
	`javax.xml.parsers`
		Java API for XML processing (JAXP) includes the basic facilities for working with XML documents: Document Object Model (DOM), Simple API for XML Parsing (SAX), XSL Transformations (XSLT), and a pluggability layer for parsers.

SQL	`java.sql` `javax.sql`	Support for accessing and processing data in databases using the JDBC API.
RMI	`java.rmi`	Support for distributed programming.
Networking	`java.net`	Classes that support development of networking applications.
Security	`java.security`	Support for cryptographic security.

Note Java packages vary by Java 2 Platform edition. The Java 2 Software Development Kit (SDK) is available in several editions used for various purposes: Standard Edition (J2SE), Enterprise Edition (J2EE), and Micro Edition (J2ME).

See also

- “Java 2 Platform editions” on page 43
- “Java 2 Platform, Standard Edition, API Specification” in the JDK API Documentation
- Sun’s tutorial, “Creating and using packages” at `http://www.java.sun.com/docs/books/tutorial/java/interpack/packages.html`
- “Packages” in “Managing paths” in *Building Applications with JBuilder*

The Language package: java.lang

One of the most important packages in the Java class library is the `java.lang` package. This package, which is automatically imported into every Java program, contains the language’s main support classes which are fundamental to the design of the Java programming language.

See also

- `java.lang` in the JDK API Documentation
- “Key java.lang classes” on page 53

The Utility package: java.util

The `java.util` package contains various utility classes and interfaces that are crucial for Java development. Classes in this package support the collections framework and date and time facilities.

See also

- `java.util` in the JDK API Documentation
- "Key java.util classes" on page 59

The I/O package: java.io

The `java.io` package provides support for reading and writing data to and from different devices. Java also supports input and output of character streams. In addition, the `File` class in the `java.io` package uses an abstract, system-independent representation of file and directory pathnames for better support of non-UNIX platforms. The classes in this package are divided into the following groups: input stream classes, output stream classes, file classes, and the `StreamTokenizer` class.

See also

- `java.io` in the JDK API Documentation
- "Key java.io classes" on page 62

The Text package: java.text

The `java.text` package provides classes and interfaces that provide localization support for handling text, dates, numbers, and messages. Classes in this package, such as `NumberFormat`, `DateFormat`, and `Collator`, can format numbers, date and time, and strings in a locale-specific way. Other classes support parsing, searching, and sorting of strings.

See also

- `java.text` in the JDK API Documentation
- "Internationalizing programs with JBuilder" in *Building Applications with JBuilder*

The Math package: java.math

The `java.math` package, not to be confused with the `java.lang.Math` class, provides classes for performing arbitrary-precision integer arithmetic (`BigInteger`) and arbitrary-precision floating point arithmetic (`BigDecimal`).

The `BigInteger` class provides support for representing arbitrarily large integers.

The `BigDecimal` class is used for calculations requiring decimal support, such as monetary calculations, and also provides operations for basic arithmetic, scale manipulation, comparison, format conversion, and hashing.

See also

- `java.math` in the JDK API Documentation
- `java.lang.Math` class in the JDK API Documentation
- "Arbitrary-Precision Math" in the JDK Guide to Features

The AWT package: java.awt

The Abstract Window Toolkit (AWT) package, part of the Java Foundation Classes (JFC), provides support for Graphical User Interface (GUI) programming and includes such features as user interface components, event-handling models, layout managers, graphics and imaging tools, and data transfer classes for cut and paste.

See also

- `java.awt` in the JDK API Documentation
- "Abstract Window Toolkit (AWT)" in the JDK Guide to Features
- "AWT Fundamentals" at `http://developer.java.sun.com/developer/onlineTraining/awt/`
- "Tutorial: Building an applet" in *Introducing JBuilder*
- "Visual Design with JBuilder" in *Designing Applications with JBuilder*

The Swing package: javax.swing

The `javax.swing` package provides a set of "lightweight" (all-Java language) components that automatically have the look and feel of any OS platform. Swing components are 100% Pure Java versions of the existing AWT component set, such as button, scrollbar, and label, with an additional set of components, such as tree view, table, and tabbed pane.

Note `javax` packages are extensions to the core Java language.

See also

- `javax.swing` in the JDK API Documentation
- "Java Foundation Classes (JFC)" at `http://java.sun.com/docs/books/tutorial/post1.0/preview/jfc.html`
- Sun's Swing tutorial, "Trail: Creating a GUI with JFC/Swing" at `http://www.java.sun.com/docs/books/tutorial/uiswing/index.html`

- Related chapters in *Designing Applications with JBuilder*:
  - "Introduction"
  - "Managing the component palette"
  - "Using layout managers"
  - "Using nested panels and layouts"
  - "Tutorial: Building a Java text editor"

The Javax packages: javax

The many `javax` packages are extensions to the core Java language. These include such packages as `javax.swing`, `javax.sound`, `javax.rmi`, `javax.transactions`, and `javax.naming`. Developers can also author their own custom `javax` packages.

See also

- `javax.accessibility` in the JDK API Documentation
- `javax.naming` in the JDK API Documentation
- `javax.rmi` in the JDK API Documentation
- `javax.sound.midi` in the JDK API Documentation
- `javax.sound.sampled` in the JDK API Documentation
- `javax.swing` in the JDK API Documentation
- `javax.transaction` in the JDK API Documentation

The Applet package: java.applet

The `java.applet` package provides the classes for creating applets, as well as classes that applets use to communicate with its applet context, usually a web browser. Applets are Java programs that are not intended to run on their own, but rather to be embedded inside another application. Commonly, applets are stored on an Internet/intranet server, called by an HTML page, and downloaded to multiple client platforms where they are run in a Java Virtual Machine (JVM) provided by the browser on the client machine. This delivery and execution is done under the supervision of a Security Manager, which can prevent applets from performing such tasks as formatting the hard drive or opening connections to "untrusted" machines.

Due to security issues and browser JDK compatibility, it is important to fully understand applets before developing them. Applets do not have the full functionality of Java programs for security reasons. Applets also rely on the browser's JDK version which may not be current. Many of the browsers at the time of this writing do not fully support the most recent JDK. For example, most browsers include an older JDK version that does not support Swing. Therefore, applets using Swing components do not run in these browsers.

See also

- `java.applet` in the JDK API Documentation
- Sun's tutorial, "Trail: Writing applets" at `http://www.java.sun.com/docs/books/tutorial/applet/index.html`
- Chapter 9, "An introduction to the Java Virtual Machine"
- "Working with applets" in the *Web Application Developer's Guide*
- "Tutorial: Building an applet" in *Introducing JBuilder*

The Beans package: java.beans

The `java.beans` package contains classes related to JavaBeans development. JavaBeans, Java classes that serve as self-contained, reusable components, extend the Java platform's "write once, run anywhere" capability to reusable component development. These reusable pieces of code can be manipulated and updated with minimal impact on the testing of the program.

See also

- `java.beans` in the JDK API Documentation
- JavaBeans Technology technical articles at `http://developer.java.sun.com/developer/technicalArticles/jbeans/index.html`
- "Creating JavaBeans with BeansExpress" in *Building Applications with JBuilder*

The Reflection package: java.lang.reflect

The `java.lang.reflect` package provides classes and interfaces for examining and manipulating classes at runtime. Reflection allows access to information about fields and methods and constructors of loaded classes. The Java code can use this reflected information to operate on counterparts of objects.

Classes in this package provide applications such as debuggers, interpreters, object inspectors, class browsers, and services such as Object Serialization and JavaBeans access to public members or members declared by a class.

See also

- `java.lang.reflect` in the JDK API Documentation
- "Reflection" in the JDK Guide to Features

XML processing

Java, in conjunction with XML (Extensible Markup Language), provides a portable, flexible framework for creating, exchanging, and manipulating information between applications and over the Internet, as well as

transforming XML documents into other document types. The Java API for XML processing (JAXP) includes the basic facilities for working with XML documents: Document Object Model (DOM), Simple API for XML Parsing (SAX), XSL Transformations (XSLT), and a pluggability layer for parsers.

See also

- `org.w3c.dom` in the JDK API Documentation
- `org.xml.sax` in the JDK API Documentation
- `javax.xml.transform` in the JDK API Documentation
- `javax.xml.parsers` in the JDK API Documentation
- "Java Technology & XML Home Page" at `http://java.sun.com/xml/`
- "XML in the Java 2 Platform" in the JDK Guide to Features
- Sun's XML Tutorial at `http://java.sun.com/xml/tutorial_intro.html`

The SQL package: java.sql

The `java.sql` package contains classes that provide the API for accessing and processing data in a data source. The `java.sql` package is also referred to as the JDBC 2.0 (Java Database Connectivity) API. This API includes a framework for dynamically installing different drivers to access different types of data sources. JDBC is an industry standard that allows the Java platform to connect with almost any database, even those written in other languages such as Structured Query Language (SQL).

The `java.sql` package includes classes, interfaces, and methods for making database connections, sending SQL statements to a database, retrieving and updating query results, mapping SQL values, providing information about a database, throwing exceptions, and providing security.

See also

- `java.sql` in the JDK API Documentation
- Sun's tutorial, "Trail: JDBC Database Access" at `http://java.sun.com/docs/books/tutorial/jdbc/index.html`
- JBuilder's "SQL reference" in the *JDataStore Developer's Guide*

The RMI package: java.rmi

The `java.rmi` package provides classes for Java Remote Method Invocation (RMI). Remote Method Invocation (RMI) enables you to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap naming service provided by RMI or by receiving the reference as an argument

or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

A sample RMI application, `SimpleRMI.jpr`, is installed in the `samples/Rmi` directory of your JBuilder installation. See the HTML project file, `SimpleRMI.html`, for a description of the sample application. (This sample is a feature of JBuilder Developer and Enterprise.)

An example of writing a distributed database application using RMI and `DataSetData` is located in the `samples/DataExpress/StreamableDataSets` directory of your JBuilder installation. This example includes a server application that will take data from the employee sample table and send the data via RMI in the form of `DataSetData`. A client application communicates with the server through a custom `Provider` and a custom `Resolver`, and displays the data in a grid. (This sample is a feature of JBuilder Enterprise.)

See also

- `java.rmi` in the JDK API Documentation
- "Java Remote Method Invocation (RMI)" in the JDK Guide to Features
- "The Java Remote Method Invocation - Distributed Computing for Java (a White Paper)" at `http://java.sun.com/marketing/collateral/javarmi.html`

The Networking package: java.net

The `java.net` package contains classes for developing networking applications. Using the socket classes, you can communicate with any server on the Internet or implement your own Internet server. Classes are also provided for data retrieval from the Internet.

See also

- `java.net` in the JDK API Documentation
- "Networking Features" in the JDK Guide to Features

The Security package: java.security

The Security package, `java.security`, defines classes and interfaces for the security framework. There are two category of classes:

- Classes that implement access control and prevent untrusted code from performing sensitive operations.
- Authentication classes that implement message digests and digital signatures and authenticate classes and other objects.

Using these classes, developers can protect access to applets and Java code, including applications, beans, and servlets, by creating permissions and security policies. When code is loaded, it is assigned permissions based on

the security policy. Permissions specify which resources can be accessed, such as read/write or connection access. The policy, which controls which permissions are available, is usually initialized from an external configurable policy file which defines the code's security policy. The use of permissions and policy allow for flexible, configurable, and extensible access control to the code.

See also

- `java.security` in the JDK API Documentation
- "Security" in the JDK Guide to Features

Key java.lang classes

The Object class: java.lang.Object

The `Object` class in the `java.lang` package is the parent class or superclass of all Java classes. This simply means that the `Object` class is the root of the class hierarchy and that all Java classes are derived from it. The `Object` class itself contains a constructor and several methods of importance, including `clone()`, `equals()`, and `toString()`.

Method	Argument	Description
`clone`	`()`	Creates and returns a copy of an object.
`equals`	`(Object obj)`	Indicates whether another object is "equal to" the specified object.
`toString`	`()`	Returns a string representation of an object

An object that uses the `clone()` method simply makes a copy of itself. When a copy is made, the new memory is allocated for the clone first, then contents of the original object is copied into the clone object. In the following example where the `Document` class implements the `Cloneable` interface, a copy of the `Document` class containing a `text` and `author` property is created using the `clone()` method. An object is not seen as cloneable unless it implements the `Cloneable` interface.

```
Document document1 = new Document("docText.txt", "Joe Smith");
Document document2 = document1.clone();
```

The `equals()` method compares two objects of the same type for equality by comparing the properties of both objects. It simply returns a boolean value depending on the results of the object that calls it and the object that is passed to it. For instance, if `equals()` is called by an object that passes it an object that is identical, the `equals()` method returns a true value.

The `toString()` method returns a string representing the object. For this method to return proper information about different types of objects, the object's class must override it.

See also

- `java.lang.Object` in the JDK API Documentation

Type wrapper classes

Due to performance reasons, primitive data types are not used as objects in Java. These primitive data types include numbers, booleans, and characters.

However, some Java classes and methods require primitive data types to be objects. Java uses wrapper classes to wrap or encapsulate the primitive type as an object as shown in the following table.

Primitive type	Description	Wrapper
`boolean`	True or False (1 Bit)	`java.lang.Boolean`
`byte`	-128 to 127 (8-bit signed integer)	`java.lang.Byte`
`char`	Unicode character (16-bit)	`java.lang.Character`
`double`	+1.79769313486231579E+308 to +4.9406545841246544E-324 (64-bit)	`java.lang.Double`
`float`	+3.40282347E+28 to +1.40239846E-45 (32-bit)	`java.lang.Float`
`int`	-2147483648 to 2147483647 (32-bit signed integer)	`java.lang.Integer`
`long`	-9223372036854775808 to 9223372036854775807 (64-bit signed integer)	`java.lang.Long`
`short`	-32768 to 32767 (16-bit signed integer)	`java.lang.Short`
`void`	An uninstantiable placeholder class to hold a reference to the Class object representing the primitive Java type void.	`java.lang.Void`

The constructor for the wrapper classes, such as `Character(char value)`, simply takes an argument of the class type it is wrapping. For example, the following code demonstrates how a `Character` wrapper class is constructed.

```
Character charWrapper = new Character('T');
```

Although each of these classes contains its own methods, several methods are standard throughout each object. These methods include methods that return a primitive type, `toString()`, and `equals()`.

Each wrapper class has a method, such as `charValue()`, that returns the primitive type of the wrapper class. The following code demonstrates this using the `charWrapper` object. Notice that `charPrimitive` is a primitive data type (declared as a `char`). Using this method, primitive data types can be assigned to type wrappers.

```
char charPrimitive = charWrapper.charValue();
```

The `toString()` and `equals()` methods are used similarly as in the `Object` class.

The Math class: java.lang.Math

The `Math` class in the `java.lang` package, not to be confused with the `java.math` package, provides useful methods that implement common math functions. This class is not instantiated and is declared `final`, so it cannot be subclassed. Some of the methods included in this class are: `sin()`, `cos()`, `exp()`, `log()`, `max()`, `min()`, `random()`, `sqrt()`, and `tan()`. The following are several examples of these methods.

```
double d1 = Math.sin(45);
double d2 = 23.4;
double d3 = Math.exp(d2);
double d4 = Math.log(d3);
double d5 = Math.max(d2, Math.pow(d1, 10));
```

Some of these methods are overloaded to accept and return different data types.

The `Math` class also declares the constants `PI` and `E`.

Note The `java.math` package, unlike `java.lang.Math`, provides support classes for working with arbitrarily large numbers.

See also

- `java.lang.Math` in the JDK API Documentation
- `java.math` in the JDK API Documentation

The String class: java.lang.String

The `String` class in the `java.lang` package is used to represent character strings. Unlike C/C++, Java does not use character arrays to represent strings. Strings are constant, meaning their values cannot change once they are created. The `String` class is typically constructed when the Java compiler encounters a string in quotes. However, strings can be constructed several ways. The following table lists several of the `String`'s constructors and the arguments they accept.

Constructor	Argument	Description
`String`	`()`	Initializes a new `String` object.
`String`	`(String value)`	Initializes a new `String` object with the contents of the `String` argument.
`String`	`(char[] value)`	Creates a new `String` that contains the array in the same sequence.
`String`	`(char[] value, int offset, int count)`	Creates a new `String` that contains a subarray of the argument.
`String`	`(StringBuffer buffer)`	Initializes a new `String` object with the contents of the `StringBuffer` argument.

The `String` class contains several important methods that are essential when dealing with strings. These methods are used to edit, compare, and analyze strings. Because strings are immutable and cannot be changed, none of these methods can change the character sequence. The `StringBuffer` class, discussed in the next section, provides methods for changing strings.

The following table lists some of the more crucial methods and declares what they accept and return.

Method	Argument	Returns	Description
`length`	`()`	`int`	Returns the number of characters in the string.
`charAt`	`(int index)`	`char`	Returns the character at the specified index of the string.
`compareTo`	`(String value)`	`int`	Compares a string to the argument string.
`indexOf`	`(int ch)`	`int`	Returns the index location of the first occurrence of the specified character.
`substring`	`(int beginIndex, int endIndex)`	`String`	Returns a new string that is a substring of the string.
`concat`	`(String str)`	`String`	Concatenates the specified `String` to the end of this string.
`toLowerCase`	`()`	`String`	Returns the string in lowercase.
`toUpperCase`	`()`	`String`	Returns the string in uppercase.
`valueOf`	`(Object obj)`	`String`	Returns the string representation of the `Object` argument.

A very efficient feature associated with many of these methods is that they are overloaded for more flexibility. The following demonstrates how the `String` class and some of its methods can be used.

Important Remember that array and `String` indexes begin at zero.

```
String s1 = new String("Hello World.");

char cArray[] = {'J', 'B', 'u', 'i', 'l', 'd', 'e', 'r'};
String s2 = new String(cArray);        //s2 = "JBuilder"

int i = s1.length();                   //i = 12
char c = s1.charAt(6);                 //c = 'W'
i = s1.indexOf('e');                   //i = 1 (index of 'e' in "Hello
                                         World.")

String s3 = "abcdef".substring(2, 5); //s3 = "cde"
String s4 = s3.concat("f");           //s4 = "cdef"
String s5 = String.valueOf(i);        //s5 = "1" (valueOf() is static)
```

See also

- `java.lang.String` in the JDK API Documentation

The StringBuffer class: java.lang.StringBuffer

The `StringBuffer` class in the `java.lang` package, like the `String` class, represents a sequence of characters. Unlike a string, the contents of a `StringBuffer` can be modified. Using various `StringBuffer` methods, the length and content of the string buffer can be changed. Additionally, the `StringBuffer` object can grow in length when necessary. Finally, after modifying the `StringBuffer`, you can create a new string representing the contents in the `StringBuffer`.

The `StringBuffer` class has several constructors shown in the following table.

Constructor	Argument	Description
`StringBuffer`	`()`	Creates an empty string buffer which can hold up to 16 characters.
`StringBuffer`	`(int length)`	Creates an empty string buffer which can hold the number of characters specified by `length`.
`StringBuffer`	`(String str)`	Creates a string buffer which contains a copy of the `String str`.

There are several important methods that separate the `StringBuffer` class from the `String` class: `capacity()`, `setLength()`, `setCharAt()`, `append()`, `insert()` and `toString()`. The `append()` and `insert()` methods are overloaded to accept various data types.

Method	Argument	Description
`setLength`	`(int newLength)`	Sets the length of the `Stringbuffer`.
`capacity`	`()`	Returns the amount of memory allocated to the `StringBuffer`.
`setCharAt`	`(int index, char ch)`	Sets the character at the specified index of the `StringBuffer` to `ch`.
`append`	`(char c)`	Adds the string representation of the data type in the argument to the `StringBuffer`. This method is overloaded to accept various data types.
`insert`	`(int offset, char c)`	Inserts the string representation of the data type in the argument into this `StringBuffer`. This method is overloaded to accept various data types.
`toString`	`()`	Converts the `StringBuffer` to a `String`.

The `capacity()` method, which returns the amount of memory allocated for the `StringBuffer`, can return a larger value than the `length()` method. Memory allocated for a `StringBuffer` can be set with the `StringBuffer(int length)` constructor.

The following code demonstrates some of the methods associated with the `StringBuffer` class.

```
StringBuffer s1 = new StringBuffer(10);

int c = s1.capacity();        //c = 10
int len = s1.length();        //len = 0

s1.append("Bor");             //s1 = "Bor"
s1.append("land");            //s1 = "Borland"

c = s1.capacity();            //c = 10
len = s1.length();            //len = 7

s1.setLength(2);              //s1 = "Bo"

StringBuffer s2 = new StringBuffer("Helo World");
s2.insert(3, "l");            //s2 = "Hello World"
```

See also

- `java.lang.StringBuffer` in the JDK API Documentation

The System class: java.lang.System

The `System` class in the `java.lang` package contains several useful class fields and methods for accessing platform-independent system resources and information, copying arrays, loading files and libraries, and getting and setting properties. For example, the `currentTimeMillis()` method provides access to the current system time. It's also possible to retrieve and change system resources using the `getProperty` and `setProperty` methods. Another convenient feature that the `System` class provides is the `gc()` method, which requests that the garbage collector perform a thorough garbage collection; and finally, the `System` class allows developers to load dynamic link libraries with the `loadLibrary()` method.

The `System` class is declared as a `final` class and can't be subclassed. It also declares its methods and variables as `static`. This simply allows them to be available without the class being instantiated.

The `System` class also declares several variables which are used to interact with the system. These variables include `in`, `out`, and `err`. The `in` variable represents the system's standard input stream, whereas the `out` variable

represents the standard output stream. The `err` variable is the standard error stream. Streams are discussed in more detail in the I/O package section.

Method	Argument	Description
`arrayCopy`	`arraycopy(Object src, int src_position, Object dst, int dst_position, int length)`	Copies the specified source array, beginning at the specified position, to the specified position of the destination array.
`currentTimeMillis`	`()`	Returns the current time in milliseconds.
`loadLibrary`	`(String libname)`	Loads the system library specified by the argument.
`getProperty`	`(String key)`	Gets the system property indicated by the key.
`gc`	`()`	Runs the garbage collector which deletes objects that are no longer in use.
`load`	`(String filename)`	Loads a code file from the local file system as a dynamic library.
`exit`	`(int status)`	Exits the current program.
`setProperty`	`(String key, String value)`	Sets the system property specified by the key.

See also

- `java.lang.System` in the JDK API Documentation

Key java.util classes

The Enumeration interface: java.util.Enumeration

The `Enumeration` interface in the `java.util` package is used to implement a class capable of enumerating values. A class that implements the `Enumeration` interface can facilitate the traversal of data structures.

The methods defined in the `Enumeration` interface allow the `Enumeration` object to continuously retrieve all the elements from a set of values, one by one. There are only two methods declared in the `Enumeration` interface, `hasMoreElements()` and `nextElement()`.

The `hasMoreElements()` method returns true if more elements remain in the data structure. The `nextElement()` method is used to return the next value in the structure being enumerated.

The following example creates a class called `CanEnumerate`, which implements the `Enumeration` interface. An instance of that class is used to print all the elements of the `Vector` object, `v`.

```
Enumeration enum = CanEnumerate.v.elements();

while (enum.hasMoreElements()) {
  System.out.println(enum.nextElement());
}
```

There is one limitation on an `Enumeration` object; it can only be used once. There is no method defined in the interface that allows the `Enumeration` object to backtrack to previous elements. So, once it enumerates the entire set of values, it's consumed.

See also

- `java.util.Enumeration` in the JDK API Documentation

The Vector class: java.util.Vector

Java does not include support for all dynamic data structures; it only defines a `Stack` class. However, the `Vector` class in the `java.util` package provides an easy way to implement dynamic data structures.

The `Vector` class is efficient, because it allocates more memory than needed when adding new elements. Therefore, a `Vector's` capacity is usually greater than its actual size. The `capacityIncrement` argument in the fourth constructor, shown in the following table, defines a `Vector's` capacity increase whenever an element is added to it.

Constructor	Argument	Description
`Vector`	`()`	Constructs an empty vector with an array size of 10 and a capacity increment of zero.
`Vector`	`(Collection c)`	Constructs a vector containing the elements of the collection, in the order they are returned by the collection's iterator.
`Vector`	`(int initialCapacity)`	Constructs an empty vector with the specified initial capacity and a capacity increment of zero.
`Vector`	`(int initialCapacity, int capacityIncrement)`	Constructs an empty vector with the specified initial capacity and capacity increment.

The following table lists some of the more important methods of the `Vector` class and the arguments they accept.

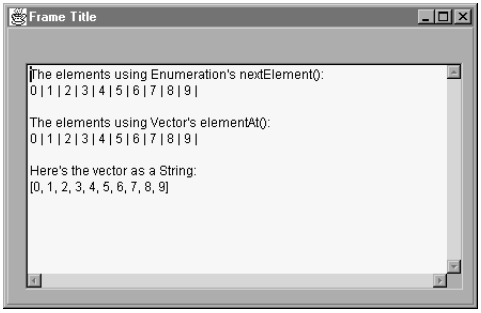
Method	Argument	Description
`setSize`	`(int newSize)`	Sets the size of a vector.
`capacity`	`()`	Returns the capacity of a vector.
`size`	`()`	Returns the number of elements stored in a vector.
`elements`	`()`	Returns an enumeration of elements of a vector.
`elementAt`	`(int)`	Returns the element at the specified index.
`firstElement`	`()`	Returns the first element of a vector (at index 0).
`lastElement`	`()`	Returns the last element of a vector.
`removeElementAt`	`(int index)`	Removes the element at the specified index.
`addElement`	`(Object obj)`	Adds the specified object to the end of a vector, increasing its size by one.
`toString`	`()`	Returns a string representation of each element in a vector.

The following code demonstrates the use of the `Vector` class. A `Vector` object called `vector1` is created and enumerates its elements in three ways: using `Enumeration's nextElement()` method, using `Vector's elementAt()` method, and using `Vector's toString()` method. An AWT component, `textArea`, is created to display the output. The text property is set using the `setText()` method.

```
Vector vector1 = new Vector();

for (int i = 0; i < 10;i++) {
  vector1.addElement(new Integer(i)); //addElement accepts object or
                                      //composite types
}                                     //but not primitive types

//enumerate vector1 using nextElement()
Enumeration e = vector1.elements();
textArea1.setText("The elements using Enumeration's nextElement():\n");
while (e.hasMoreElements()) {
  textArea1.append(e.nextElement()+ " | ");
}
textArea1.append("\n\n");

//enumerate using the elementAt() method
textArea1.append("The elements using Vector's elementAt():\n");
for (int i = 0; i < vector1.size();i++) {
  textArea1.append(vector1.elementAt(i) + " | ");
}
textArea1.append("\n\n");

//enumerate using the toString() method
textArea1.append("Here's the vector as a String:\n");
textArea1.append(vector1.toString());
```

The following figure demonstrates what this code would accomplish if it was used within an application.

Figure 5.1 Vector and Enumeration example



See also

- `java.util.Vector` in the JDK API Documentation

Key java.io classes

Input stream classes

An input stream is used to read data from an input source, such as a file, a string, or memory. Examples of input stream classes in the `java.io` package include `InputStream`, `BufferedInputStream`, `DataInputStream`, and `FileInputStream`.

The basic method of reading data using an input stream class is always the same:

1. Create an instance of an input stream class.
2. Tell it where to read the data from.

Note Input stream classes read data as a continuous stream of bytes. If no data is currently available, the input stream class blocks or waits until data becomes available.

In addition to the input stream classes, the `java.io` package provides reader classes (except for `DataInputStream`). Examples of reader classes include `Reader`, `BufferedReader`, `FileReader`, and `StringReader`. Reader classes are identical to input stream classes, except that they read Unicode characters instead of bytes.

InputStream class: java.io.InputStream

The `InputStream` class in the `java.io` package is an abstract class and the superclass of all other input stream classes. It provides the basic interface for reading a stream of bytes. The following table lists some of the methods

defined in the `InputStream` class and the arguments these methods accept. Each of these methods returns an `int` value, except the `close()` method.

Method	Argument	Description
`read`	`()`	Reads the next byte from the input stream and returns it as an integer. When it reaches the end of the stream, it returns -1.
`read`	`(byte b[])`	Reads multiple bytes and stores them in array `b`. It returns the number of bytes read or -1 when the end of the stream is reached.
`read`	`(byte b[], int off, int len)`	Reads up to `len` bytes of data starting from offset `off` from the input stream into an array.
`available`	`()`	Returns the number of bytes that can be read from an input stream without blocking by the next caller of a method for the input stream.
`skip`	`(long n)`	Skips over and discards `n` bytes of data from an input stream.
`close`	`()`	Closes an input stream and releases system resources used by the stream.

See also

- `java.io.InputStream` in the JDK API Documentation

FileInputStream class: java.io.FileInputStream

The `FileInputStream` class in the `java.io` package is very similar to the `InputStream` class, only it's designed specifically for reading files. It contains three constructors: `FileInputStream(String filename)`, `FileInputStream(File fileobject)`, and `FileInputStream(FileDescriptor fdObj)`. The first constructor takes the file's name as an argument, while the second simply takes a file object. The third constructor takes a file descriptor object. File classes are discussed later.

Constructor	Argument	Description
`FileInputStream`	`(String filename)`	Creates a `FileInputStream` by opening a connection to the file named by the path name `filename` in the file system.
`FileInputStream`	`(File fileobject)`	Creates a `FileInputStream` by opening a connection to the file named by the `fileobject` file in the file system.
`FileInputStream`	`(FileDescriptor fdObj)`	Creates a `FileInputStream` by using the file descriptor `fdObj`, which represents an existing connection to an actual file in the file system.

The following example demonstrates the use of the `FileInputStream` class.

```
import java.io.*;

class FileReader {
 public static void main(String args[]) {
  byte buff[] = new byte[80];
  try {
   InputStream fileIn = new FileInputStream("Readme.txt");
   int i = fileIn.read(buff);
   String s = new String(buff);
   System.out.println(s);
  }
  catch(FileNotFoundException e) {
  }
  catch(IOException e) {
  }
 }
}
```

In this example, a character array that stores the input data is created. Then, a `FileInputStream` object is instantiated and the input file's name is passed to its constructor. Next, the `FileInputStream read()` method is used to read a stream of characters and store them in the `buff` array. The first 80 bytes are read from the `Readme.txt` file and stored in the `buff` array.

Note The `FileReader` class could also be used in place of the `FileInputStream()` method. The only changes needed would be a `char` array used in place of the `byte` array, and the `reader` object would be instantiated as follows:

```
Reader fileIn = new FileReader("Readme.txt");
```

Finally, to see the result of the `read` call, a `String` object is created using the `buff` array and then passed to the `System.out.println()` method.

As mentioned earlier, the `System` class defined in `java.lang` provides access to system resources. `System.out`, a `static` member of `System`, represents the standard output device. The `println()` method is called to send the output to the standard output device. The `System.out` object is of type `PrintStream`, which is discussed below.

The `System.in` object, another `static` member of the `System` class, is of type `InputStream` and represents the standard input device.

See also

- `java.io.FileInputStream` in the JDK API Documentation

Output stream classes

The output stream classes are the counterparts to the input stream classes. They are used to output streams of data to various output sources. The main output stream classes in Java, located in the `java.io` package, are `OutputStream`, `PrintStream`, `BufferedOutputStream`, `DataOutputStream`, and `FileOutputStream`.

OutputStream class: java.io.OutputStream

To output a stream of data, an `OutputStream` object is created and directed to output the data to a particular output source. As expected, there are also corresponding writer classes for each class, except the `DataOutputStream` class. Some of the methods defined in the `OutputStream` class include:

Method	Argument	Description
`write`	`(int b)`	Writes `b` to an output stream.
`write`	`(byte b[])`	Writes array `b` to an output stream.
`write`	`(byte b[], int off, int len)`	Writes `len` bytes from the byte array starting at offset `off` to the output stream.
`flush`	`()`	Flushes the output stream and forces the output of any buffered data.
`close`	`()`	Closes the output stream and releases any system resources associated with it.

See also

- `java.io.OutputStream` in the JDK API Documentation

PrintStream class: java.io.PrintStream

The `PrintStream` class in the `java.io` package, primarily designed to output data as text, has two constructors. The first constructor flushes the buffered data based on specified conditions, while the second flushes the data when it encounters a new line character (if `autoflush` is set to `true`).

Constructor	Argument	Description
`PrintStream`	`(OutputStream out)`	Creates a new print stream.
`PrintStream`	`(OutputStream out, boolean autoflush)`	Creates a new print stream.

Several of the methods defined in the `PrintStream` class are shown in the following table.

Method	Argument	Description
`checkError`	`()`	Flushes the stream and returns a false value if an error is detected.
`print`	`(Object obj)`	Prints an object.
`print`	`(String s)`	Prints a string.
`println`	`()`	Prints and terminates the line using the line separator string which is defined by the system property `line.separator` and is not necessarily a single newline character ('\n').
`println`	`(Object obj)`	Prints an object and terminates the line. This method behaves as though it invokes `print(Object)` and then `println()`.

The `print()` and `println()` methods are overloaded to receive different data types.

See also

- `java.io.PrintStream` in the JDK API Documentation

BufferedOutputStream class: java.io.BufferedOutputStream

The `BufferedOutputStream` class in the `java.io` package implements a buffered output stream which increases output efficiency by storing values in a buffer and writing them only when the buffer is full or the `flush()` method is called.

Constructor	Argument	Description
`BufferedOutputStream`	`(OutputStream out)`	Creates a new 512-byte buffered output stream to write data to the output stream.
`BufferedOutputStream`	`(OutputStream out, int size)`	Creates a new buffered output stream to write data to the output stream with the specified buffer size.

`BufferedOutputStream` has three methods to flush and write to the output stream.

Method	Argument	Description
`flush`	`()`	Flushes the buffered output stream.
`write`	`(byte[] b, int off, int len)`	Writes `len` bytes from the byte array starting at offset `off` to the buffered output stream.
`write`	`(int b)`	Writes the byte to the buffered output stream.

See also

- `java.io.BufferedOutputStream` in the JDK API Documentation

DataOutputStream class: java.io.DataOutputStream

A data output stream allows an application to write primitive Java data types to an output stream in a portable binary format. An application can then use a data input stream to read the data back in.

The `DataOutputStream` class in the `java.io` package has a single constructor, `DataOutputStream(OutputStream out)`, that creates a new data output stream used to write data to an output stream.

The `DataOutputStream` class uses a variety of `write()` methods to output primitive data types, as well as a `flush()` and `size()` method.

Method	Argument	Description
`flush`	`()`	Flushes the data output stream.
`size`	`()`	Returns the number of bytes written to the data output stream.
`write`	`(int b)`	Writes the byte to the output stream.
`writeType`	`(type v)`	Writes the specified primitive type to the output stream as bytes.

See also

- java.io.DataOutputStream in the JDK API Documentation

FileOutputStream class: java.io.FileOutputStream

A file output stream is an output stream for writing data to a file or to a FileDescriptor. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms allow a file to be opened for writing by only one FileOutputStream at a time. In such situations the constructors in this class fail if the file involved is already open. FileOutputStream in the java.io package, a subclass of OutputStream, has several constructors.

Constructor	Argument	Description
FileOutputStream	(File file)	Creates a file output stream to write to the file specified.
FileOutputStream	(FileDescriptor fdObj)	Creates an output file stream to write to the file descriptor, which represents an existing connection to an actual file in the file system.
FileOutputStream	(String name)	Creates an output file stream to write to the file with the specified name.
FileOutputStream	(String name, boolean append)	Creates an output file stream to write to the file with the specified name.

FileOutputStream has several methods, including close(), finalize(), and several write() methods.

Method	Argument	Description
close	()	Closes the file output stream and releases any associated system resources.
finalize	()	Cleans up the connection to the file and calls the close() method when there are no more references to the stream.
getFD	()	Returns the file descriptor associated with the stream.
write	(byte[] b, int off, int len)	Writes len bytes from the byte array starting at offset off to the file output stream.

See also

- java.io.FileOutputStream in the JDK API Documentation

File classes

The FileInputStream and FileOutputStream classes in the java.io package only provide basic functions for handling file input and output. The java.io package provides the File class and RandomAccessFile class for more advanced file

support. The `File` class provides easy access to file attributes and functions, while the `RandomAccessFile` class provides various methods for reading and writing to and from a file.

File class: java.io.File

Java's `File` class uses an abstract, platform-independent representation of file and directory pathnames. The `File` class has three constructors listed in the following table.

Constructor	Argument	Description
`File`	`(String path)`	Creates a new `File` instance by converting the given pathname string into an abstract pathname.
`File`	`(String parent, String child)`	Creates a new `File` instance from a parent pathname string and a child pathname string.
`File`	`(File parent, String child)`	Creates a new `File` instance from a parent abstract pathname and a child pathname string.

The `File` class also implements many important methods which check for the existence, readability, writeability, type, size, and modification time of files and directories, as well as making new directories and renaming and deleting files and directories.

Method	Argument	Returns	Description
`delete`	`()`	`boolean`	Deletes the file or directories.
`canRead`	`()`	`boolean`	Tests whether the application can read the file denoted by the abstract pathname.
`canWrite`	`()`	`boolean`	Tests whether the application can write to the file.
`renameTo`	`(File dest)`	`boolean`	Renames the file.
`getName`	`()`	`String`	Returns the name string of the file or directory.
`getParent`	`()`	`String`	Returns the pathname string of the parent directory of the file or directory.
`getPath`	`()`	`String`	Converts the abstract pathname into a pathname string.

See also

- `java.io.File` in the JDK API Documentation

RandomAccessFile class: java.io.RandomAccessFile

The `RandomAccessFile` class in the `java.io` package is more powerful than the `FileInputStream` and `FileOutputStream` classes, which only provide sequential access to a file. The `RandomAccessFile` class allows you to read and write arbitrary bytes, text, and Java data types to or from any specified location in a

file. It has two constructors: `RandomAccessFile(String name, String mode)` and `RandomAccessFile(File file, String mode)`. The `mode` argument indicates whether the `RandomAccessFile` object is used for reading ("r") or reading/writing ("rw").

Constructor	Argument	Description
`RandomAccessFile`	`(String name, String mode)`	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.
`RandomAccessFile`	`(File file, String mode)`	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

There are many powerful methods implemented by the `RandomAccessFile` class. Some of these methods include:

Method	Argument	Description
`seek`	`(long pos)`	Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occur.
`read`	`()`	Reads the next byte of data from the input stream.
`read`	`(byte b[], int off, int len)`	Reads up to `len` bytes of data starting from offset `off` from the input stream into an array.
`readType`	`()`	Reads the specified data type from a file, such as `readChar`, `readByte`, `readLong`.
`write`	`(int b)`	Writes the specified byte to the file.
`write`	`(byte b[], int off, int len)`	Writes `len` bytes from the byte array starting at offset `off` to the output stream.
`length`	`()`	Returns the length of the file.
`close`	`()`	Closes the file and releases any associated system resources.

See also

- `java.io.RandomAccessFile` in the JDK API Documentation

The StreamTokenizer class: java.io.StreamTokenizer

The `StreamTokenizer` class in the `java.io` package is used to read an input stream and break it up or parse it into individual tokens that can then be processed one by one. Tokens are groups of characters that represent a number or word. The stream tokenizer can recognize strings, numbers, identifiers, and comments. This technique of processing streams into tokens is perhaps most commonly used in writing parsers, compilers, or programs that process character input.

This class has one constructor, `StreamTokenizer(Reader r)`, and defines the four following constants.

Constant	Description
`TT_EOF`	Indicates that the end of the file has been read.
`TT_EOL`	Indicates that the end of the line has been read.
`TT_NUMBER`	Indicates that a number token has been read.
`TT_WORD`	Indicates that a word token has been read.

The `StreamTokenizer` class uses instance variables `nval`, `sval`, and `ttype` to hold the number value, string value, and type of the token respectively.

The `StreamTokenizer` class implements several methods used to define the lexical syntax of tokens.

Method	Argument	Description
`nextToken`	`()`	Parses the next token from the input stream. Returns `TT_NUMBER` if the next token is a number, `TT_WORD` if the next token is a word or a character.
`parseNumbers`	`()`	Parses the numbers.
`lineno`	`()`	Returns the current line number.
`pushBack`	`()`	Returns the current value in the `ttype` field on the next call of the `nextToken()` method.
`toString`	`()`	Returns the string equivalent of the current token.

Follow these steps when using a stream tokenizer:

1. Create a `StreamTokenizer` object for a `Reader`.
2. Define how to process the characters.
3. Use the `nextToken()` method to get the next token.
4. Read the `ttype` instance variable to find the token type.
5. Read the value of the token from the instance variable.
6. Process the token.
7. Repeat steps 3 to 6 above until `nextToken()` returns `StreamTokenizer.TT_EOF`.

See also

- `java.io.StreamTokenizer` in the JDK API Documentation

Chapter

6

Object-oriented programming in Java

Object-oriented programming has been around since the introduction of the language Simula ‘67 in 1967. It really came to the forefront of programming paradigms in the mid-1980s, however.

Unlike traditional structured programming, object-oriented programming places the data and the operations that pertain to the data within a single data structure. In structured programming, the data and the operations on the data are separate and data structures are sent to procedures and functions to be operated on. Object-oriented programming solves many of the problems inherent in this design because the attributes and operations are part of the same entity. This more closely models the real world, in which all objects have both attributes and activities associated with them.

Java is a *pure* object-oriented language, meaning that the outermost level of data structure in Java is the *object*. There are no stand-alone constants, variables, or functions in Java. Everything is accessed through classes and objects. This is one of the nicest features of Java. Other hybrid object-oriented languages have aspects of structured languages in addition to object extensions. For example, C++ and Object Pascal are object-oriented languages, but you can still write structured programming constructs, which dilutes the effectiveness of the object-oriented extensions. You just can’t do that in Java!

This chapter assumes you have some knowledge about programming in other object-oriented languages. If you don’t, you should refer to other sources to

find a more in-depth explanation of object-oriented programming. This chapter attempts to highlight and summarize the object-oriented features of Java.

Classes

Classes and objects are not the same thing. A class is a type definition, whereas an object is a declaration of an instance of a class type. Once you create a class, you can create as many objects based on that class as you want. The same relationship exists between classes and objects as between cherry pie recipes and cherry pies; you can make as many cherry pies as you want from a single recipe.

The process of creating an object from a class is referred to as *instantiating* an object or creating an *instance* of a class.

Declaring and instantiating classes

A class in Java can be very simple. Here is a class definition for an empty class:

```
class MyClass {
}
```

While this class is not yet useful, it is legal in Java. A more useful class would contain some data members and methods, which you'll add soon. First, examine the syntax for instantiating a class. To create an instance of this class, use the `new` operator in conjunction with the class name. You must declare an instance variable for the object:

```
MyClass myObject;
```

Just declaring an instance variable doesn't allocate memory and other resources for the object, however. Doing so creates a reference called `myObject`, but it doesn't instantiate the object. The `new` operator performs this task.

```
myObject = new MyClass();
```

Notice that the name of the class is used as if it were a method. This is not coincidental, as you will see in an upcoming section. Once this line of code has executed, the member variables and methods of the class, which don't yet exist, can be accessed using the "." operator

Once you have created the object, you never have to worry about destroying it. Objects in Java are automatically garbage collected, which means that when the object reference is no longer used, the virtual machine automatically deallocates any resources allocated by the `new` operator.

Data members

As stated above, a class in Java can contain both data members and methods. A data member or member variable is a variable declared within the class. A method is a function or routine that performs some task. Here is a class that contains just data members:

```
public class DogClass {
  String name, eyeColor;
  int age;
  boolean hasTail;
}
```

This example creates a class called `DogClass` that contains data members: `name`, `eyeColor`, `age`, and a flag called `hasTail`. You can include any data type as a member variable of a class. To access a data member, you must first create an instance of the class, then access the data using the "." operator.

Class methods

You can also include methods in classes. In fact, there are no standalone functions or procedures in Java. All subroutines are defined as methods of classes. Here is an example of `DogClass` with a `speak()` method added:

```
public class DogClass {
  String name,eyeColor;
  int age;
  boolean hasTail;

  public void speak() {
    JOptionPane.showMessageDialog(null, "Woof! Woof!");
  }
}
```

Notice that when you define methods, the implementation for the method appears directly below the declaration. This is unlike some other object-oriented languages where the class is defined in one location and the implementation code appears somewhere else. A method must specify a return type and any parameters received by the method. The `speak()` method takes no parameters. It also doesn't return a value, so its return type is `void`.

To call the method, you would access it just like you would access the member variables; that is, using the ." operator. For example,

```
DogClass dog = new DogClass();
dog.age = 4;
dog.speak();
```

Constructors and finalizers

Every Java class has a special purpose method called a *constructor*. The constructor always has the same name as the class and it can't specify a return value. The constructor allocates all the resources needed by the object and returns an instance of the object. When you use the `new` operator, you are actually calling the constructor. You don't need to specify a return type for the constructor because the instance of the object is always the return type.

Most object-oriented languages have a corresponding method called a *destructor* that is called to deallocate all the resources that the constructor allocated. But because Java deallocates all the resources for you automatically, there is no destructor mechanism in Java.

There are situations, however, that require you to perform some special cleanup that the garbage collector can't handle as the class goes away. For example, you might have opened some files in the life of the object and you want to make sure the files are closed properly when the object is destroyed. There is another special purpose method that can be defined for a class called a *finalizer*. This method (if present) is called by the garbage collector immediately before the object is destroyed. Therefore, if there is any special cleanup that needs to be performed, the finalizer can handle it for you. The garbage collector runs as a low priority thread in the virtual machine, however, so you can never predict when it will actually destroy your object. So, you shouldn't put any time-sensitive code in the finalizer because you can't predict when it will be called.

Case study: A simple OOP example

In this section, you'll see a simple example of defining classes and instantiating objects. You'll develop an application that creates two objects (a dog and a man) and show their attributes on a form.

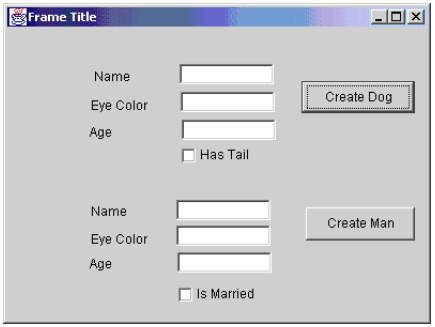
If you are completely new to JBuilder, you should put this chapter aside and learn about JBuilder's integrated development environment before you begin this sample. Begin with the *Introducing JBuilder* book, especially the "Building an application" tutorial and the following chapters that introduce the JBuilder integrated development environment. Also study the early chapters of *Designing Applications with JBuilder* to learn about using the UI designer.

You are ready to resume this chapter once you are comfortable performing these tasks:

- Beginning an application using JBuilder's Application wizard.
- Selecting components from the component palette and placing them on the UI designer.
- Setting component properties using the Inspector.
- Switching between the editor and the UI designer in JBuilder's content pane.
- Using the editor.

This is what the running sample application you will build looks like:

Figure 6.1 Sample application showing two instantiated objects



Follow these steps listed in this section to create a simple UI for this sample application.

1. Start creating the application and designing its UI:
   - **a** Start a new project. Choose File|New Project to start the Project wizard.
   - **b** Enter `oop1` in the Project Name field and click Finish. A new project opens.
   - **c** Choose File|New, click the General tab, and click the Application icon to start the Application wizard.
   - **d** Accept the default class name. The package name will be `oop1`.
   - **e** Click Next and then Finish to create a `Frame1.java` and an `Application1.java` file.
   - **f** Click the Design tab in the content pane to display the UI designer for `Frame1.java`.
   - **g** Select `contentPane` in the structure pane. In the Inspector set the `layout` property of `contentPane` to `XYLayout` (if you are a Foundation user, set `layout` to `null`). `XYLayout` and `null` are seldom ideal for an application, but until you learn about using layouts, you can use them to create a quick-and-dirty UI.
2. Place the needed components on the UI designer, using the above screen shot as a reference:
   - **a** Click the Swing tab on the component palette and click the `JTextField` component. (When you position your cursor over a component, a tooltip appears labeling the component. Click the component labeled `javax.swing.JTextField`.) Click in the UI designer and hold down the mouse button as you draw the component onscreen. Repeat this step five more times until you have two groups of three `JTextField` components on your form.
   - **b** Hold down the *Shift* key as you click each `JTextField` on the UI designer so that all of them are selected. Select the `text` property in the Inspector

and delete the text that appears there. This will delete all the text in each `JTextField` component.

c Change the `name` property value of each `JTextField`. Name the first one `txtfldDogName`, the second `txtfldDogEyeColor`, the third `txtfldDogAge`, the fourth `txtfldManName`, the fifth `txtfldManEyeColor`, and the sixth `txtfldManAge`.

d Draw six `JLabel` components on the form, each one adjacent to a `JTextField` component.

e Change the `text` property values for these components to label each `JTextField` component appropriately. For example, the `text` property of the `JLabel` at the top of the form should be `Name`, the second `Eye Color`, and so on.

f Place two `JCheckBox` components on the form. Place the first one below the first group of three `JTextField` components, and the second check box beneath the second group of `JTextField` components.

g Select each `JCheckBox` component on the form in turn and change the first one's `text` property to `Has Tail`, and second one's `text` property to `Is Married`.

h Change the value of the `name` property for the first check box to `chkboxDog`, and change the name of the second check box to `chkboxMan`.

i Place two `JButton` components on the form, one to the right of the top group of components, and the second to the right of the bottom group of components.

j Change the `text` property of the first button to `Create Dog`, and change the `text` property of the second button to `Create Man`.

The final step is to save the project by choosing File|Save All.

You're now ready to begin programming. First create a new class:

1 Choose File|New Class to start the Class wizard.

2 Keep the name of the package as `oop1`, specify the Class Name as `DogClass`, and don't change the Base Class.

3 Check only the Public and Generate Default Constructor options, unchecking all other options.

4 Click OK.

The Class wizard creates the `DogClass.java` file for you. Modify the code it created so that your code looks like this:

```
package oop1;

public class DogClass {
  String name, eyeColor;
  int age;
  boolean hasTail;
```

```
  public DogClass() {
    name = "Snoopy";
    eyeColor = "Brown";
    age = 2;
    hasTail = true;
  }
}
```

You defined `DogClass` with some member variables. There is also a constructor to instantiate `DogClass` objects.

Using the Class wizard, create a `ManClass.java` file by following the same steps except specifying the Class Name as `ManClass`. Modify the resulting code so that it looks like this:

```
package oop1;

public class ManClass {
  String name, eyeColor;
  int age;
  boolean isMarried;

  public ManClass() {
    name = "Steven";
    eyeColor = "Blue";
    age = 35;
    isMarried = true;
  }
}
```

The two classes are very similar. You'll take advantage of this similarity in an upcoming section.

Click the Frame1 tab at the top of content pane to return to the `Frame1` class. Click the Source tab at the bottom to return to open the editor. Declare two instance variables as references to the objects. Here is the source listing of the `Frame1` variable declarations shown in bold; add the lines in bold to your class:

```
public class Frame1 extends JFrame {
  // Create a reference for the dog and man objects
  DogClass dog;
  ManClass man;

  JPanel contentPane;
  JPanel jPanel1 = new JPanel();
  . . .
```

Click the Design tab at the bottom of content pane to return to the UI you designed. Double-click the Create Dog button. JBuilder creates the beginning of an event handler for that button and places your cursor within the event

handler code. Fill in the event handler code so that you instantiate a dog object and fill in the dog text fields. Your code should look like this:

```
void jButton1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.name);
  txtfldDogEyeColor.setText(dog.eyeColor);
  txtfldDogAge.setText(Integer.toString(dog.age));
  chkboxDog.setSelected(true);
}
```

As the code shows, you are calling the constructor for the dog object and then accessing its member variables.

Click the Design tab to return to the UI designer. Double-click the Create Man button. JBuilder creates an event handler for the Create Man button. Fill in the event handler so that it looks like this:

```
void jButton2_actionPerformed(ActionEvent e) {
  man = new ManClass();
  txtfldManName.setText(man.name);
  txtfldManEyeColor.setText(man.eyeColor);
  txtfldManAge.setText(Integer.toString(man.age));
  chkboxMan.setSelected(true);
}
```

You can now compile and run your application. Choose Project|Make Project “oop1.jpx” to compile it. If you have no errors, choose Run|Run Project.

If all goes well, the form appears on your screen. When you click the Create Dog button, a `dog` object is created and the dog values appear in the dog fields. When you click the Create Man button, a `man` object is created and the man values appear in the appropriate fields.

Class inheritance

The dog and man objects you created have many similarities. One of the benefits of object-oriented programming is the ability to handle similarities like this within a hierarchy. This ability is referred to as *inheritance*. When a class inherits from another class, the child class automatically inherits all the characteristics (member variables) and behavior (methods) from the parent class. Inheritance is always additive; there is no way to inherit from a class and get less than what the parent class has.

Inheritance in Java is handled through the keyword `extends`. When one class inherits from another class, the child class extends the parent class. For example,

```
public class DogClass extends MammalClass {
. . .
}
```

The items that men and dogs have in common could be said to be common to all mammals; therefore, you can create a `MammalClass` to handle these similarities. You can then remove the declarations of the common items from

DogClass and ManClass, declare them in MammalClass instead, and then subclass DogClass and ManClass from MammalClass.

Using the Class wizard, create a MammalClass. Fill in the resulting code so that it looks like this:

```
package oop1;

public class MammalClass {
  String name, eyeColor;
  int age;

  public MammalClass() {
    name = "The Name";
    eyeColor = "Brown";
    age = 0;
  }
}
```

Notice that the MammalClass has common characteristics from both the DogClass and the ManClass. Now, rewrite DogClass and ManClass to take advantage of inheritance.

Modify the code of DogClass so that it look like this:

```
package oop1;

public class DogClass extends MammalClass {

  boolean hasTail;

  public DogClass() {
    // implied super()
    name = "Snoopy";
    age = 2;
    hasTail = true;
  }
}
```

Modify the code of ManClass so that it looks like this:

```
package oop1;

public class ManClass extends MammalClass{

  boolean isMarried;

  public ManClass() {
    name = "Steven";
    eyeColor = "Blue";
    age = 35;
    isMarried = true;
  }
}
```

Notice that `DogClass` doesn't specifically assign an `eyeColor` value, but `ManClass` does. `DogClass` doesn't need to assign a value to `eyeColor` because the dog Snoopy has brown eyes and `DogClass` inherits brown eyes from the `MammalClass`, which declares an `eyeColor` variable and assigns it the value of "Brown". The man Steven, however, has blue eyes, so its necessary to assign the value "Blue" to the `eyeColor` variable inherited from `MammalClass`.

Try compiling and running your project again. (Choosing Run|Run Project will compile and then run your application.) You'll see that the UI of your program looks just as it did before, but now the `dog` and `man` objects inherit all common member variables from `MammalClass`.

As soon as `DogClass` extends `MammalClass`, `DogClass` has all the member variables and methods that the `MammalClass` has. In fact, even `MammalClass` is inherited from another class. All classes in Java ultimately extend the `Object` class; so if a class is declared that doesn't extend another class, it implicitly extends the `Object` class.

Classes in Java can inherit from only one class at a time (*single inheritance*). Unlike Java, some languages (such as C++) allow a class to inherit from several classes at once (*multiple inheritance*). A class can extend just one class at a time. Although there is no restriction on how many times you can use inheritance to extend the hierarchy, you must do so one extension at a time. Multiple inheritance is a nice feature, but it leads to very complex object hierarchies. Java has a mechanism that provides many of the same benefits without so much complexity, as you'll see later.

The `MammalClass` has a constructor that sets very practical and convenient default values. It would be nice if the subclasses could access this constructor.

In fact, they can. You can do this in Java two different ways. If you don't call the parent's constructor explicitly, **Java automatically calls the parent's default constructor for you as the first line of the child constructor**. The only way to prevent this behavior is to call one of the parent's constructors yourself as the first line of the child class constructor. Constructor calls are always chained like this, and you can't defeat this mechanism. This is a very nice feature of the Java language, because in other object-oriented languages, failing to call the parent's constructor is a common bug. Java will always do this for you if you don't. That is the meaning of the comment in the first line of the DogClass constructor, `// implied super()`. The `MammalClass` constructor is called at that point automatically. This mechanism relies on the existence of a superclass (parent class) constructor that takes no parameters. If the constructor doesn't exist and you don't call one of the other constructors as the first line of the child constructor, the class won't compile.

Calling the parent's constructor

Because you frequently want to call the superclass constructor explicitly, there is a keyword in Java that makes this easy. `super()` will call the parent's constructor that has the appropriate supplied parameters.

It's also possible to have more than one constructor in a class. When more than one method with the same name exists within a single class, the methods

are referred to as `overloaded`. It's common for a class to have multiple constructors.

For the sample application, the change in the hierarchy is the only difference between the first two versions of the sample. The instantiation of the objects and the main form haven't changed at all. However, the design of the application is more efficient, because now if you must modify any of the mammal characteristics, you can do so in the `MammalClass` and just recompile the child classes. The changes you make flow to the child classes.

Access modifiers

It's important to understand when members (both variables and methods) in the class are accessible. There are several options in Java to allow you to closely tailor how accessible you want these members to be.

Usually you want to limit the scope of program elements, including class members, as much as possible. The fewer places something is accessible, the fewer places it can be accessed incorrectly.

There are four different access modifiers for class members in Java: `private`, `protected`, `public`, and `default` (or the absence of any modifier). This is slightly complicated by the fact that classes within the same package have different access than classes outside the package. Therefore, here are two tables that show both the accessibility and inheritability of classes and member variables from within the same package and from outside the package (packages are discussed in a later section).

Access from within class's package

Access Modifier	Inherited	Accessible
default (no modifier)	Yes	Yes
Public	Yes	Yes
Protected	Yes	Yes
Private	No	No

This table shows how class members are accessed and inherited from with respect to other members in the same package. For example, a member that is declared to be private cannot be accessed by, or inherited from, other members of the same package. On the other hand, members declared using the other modifiers could be accessed by and inherited from all other members of that package. All parts of the sample application are part of the `oop1` package, so you don't have to worry about accessing classes in another package.

Access outside of a package

The rules change if you access code outside of your class's package:

Access Modifier	Inherited	Accessible
default (no modifier)	No	No
Public	Yes	Yes
Protected	Yes	No
Private	No	No

For example, this table shows that a protected member could be inherited from, but not accessed, by classes outside its package.

Note that in both access tables public members are available to anyone who wants to access them (notice that constructors are always public), whereas private members are never accessible nor inheritable outside the class. So, you should declare any member variable or method you want to keep internal to the class private.

A recommended practice in object-oriented programming is to hide information within the class by making all of the member variables of the class private and accessing them through methods that are in a specific format called accessor methods.

Accessor methods

Accessor methods (sometimes called getters and setters) are methods that provide the outward public interface to the class while keeping the actual data storage private to the class. This is a good idea because you can, at any time in the future, change the internal representation of the data in the class without touching the methods that actually set those internal values. As long as you don't change the public interface to the class, you don't break any code that relies on that class and its public methods.

Accessor methods in Java usually come in pairs: one to *get* the internal value, and another to *set* the internal value. By convention, the Get method uses the internal private variable name with "get" as a prefix. The Set method does the same with "set". A read-only property would only have a Get method. Usually, Boolean Get methods use "is" or "has" as the prefix instead of "get". Accessor methods also make it easy to validate the data that is assigned to a particular member variable.

Here is an example. In your `DogClass`, make all of the internal member variables private and add accessor methods to access the internal values. `DogClass` creates just one new member variable, `tail`.

```
package oop1;

public class DogClass extends MammalClass{

  // accessor methods for properties
  // Tail
  public boolean hasTail() {
    return tail;
  }

  public void setTail( boolean value ) {
    tail = value;
  }

  public DogClass() {
    setName("Snoopy");
    setAge(2);
    setTail(true);
  }

  private boolean tail;
}
```

The variable `tail` has been moved to the bottom of the class and now is declared as private. The location of the definition is not important, but it's common in Java to place the private members of the class at the bottom of the class definition (after all, you can't get to them outside the class; therefore, if you are reading the code, you are interested in the public aspects first). `DogClass` now has public methods to retrieve and set the value of `tail`. The getter is `hasTail()` and setter is `setTail()`.

Follow the same patterns and revise `ManClass` so that it looks like this:

```
package oop1;

 public class ManClass extends MammalClass {

  public boolean isMarried() {
    return married;
  }

  public void setMarried(boolean value) {
    married = value;
  }

  public ManClass() {
    setName("Steven");
    setAge(35);
    setEyeColor("Blue");
    setMarried(true);
  }

  private boolean married;
}
```

Note that the constructors for these two classes now use accessor methods to set the values of the variables of `MammalClass`. But the `MammalClass` doesn't have accessor methods for setting those values yet, so you must add them to `MammalClass`.

Change `MammalClass` so that its code looks like this:

```
public class MammalClass {

  // accessor methods for properties
  // name
  public String getName() {
    return name;
  }

  public void setName(String value) {
    name = value;
  }

  // eyecolor
  public String getEyeColor() {
    return eyeColor;
  }

  public void setEyeColor(String value) {
    eyeColor = value;
  }

  // sound
  public String getSound() {
    return sound;
  }

  public void setSound(String value) {
    sound = value;
  }

  // age
  public int getAge() {
    return age;
  }

  public void setAge(int value)  {
    if (value > 0) {
      age = value;
    }
    else
      age = 0;
  }
```

```
  public MammalClass() {
    setName("The Name");
    setEyeColor("Brown");
    setAge(0);
  }

  private String name, eyeColor, sound;
  private int age;
}
```

Also note that a new `sound` member variable has been added to `MammalClass`. It too has accessor methods. Because `DogClass` and `ManClass` extend `MammalClass`, they also have a `sound` property.

The event handlers in `Frame1.java` should also use the accessors. Modify the event handlers so they look like this:

```
void jButton1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.getName());
  txtfldDogEyeColor.setText(dog.getEyeColor());
  txtfldDogAge.setText(Integer.toString(dog.getAge()));
  chkboxDog.setSelected(true);
}

void jButton2_actionPerformed(ActionEvent e) {
  man = new ManClass();
  txtfldManName.setText(man.getName());
  txtfldManEyeColor.setText(man.getEyeColor());
  txtfldManAge.setText(Integer.toString(man.getAge()));
  chkboxMan.setSelected(true);
}
```

Abstract classes

It's possible to declare a method in a class as *abstract*, meaning that there will be no implementation for the method within this class, but all classes that extend this class must provide an implementation.

For example, suppose you want all mammals to have the ability to report their top running speed, but you want each mammal to report a different speed. In the mammal class you should create an abstract method called *speed()*. Add a `speed()` method to `MammalClass` just above the private member variable declarations at the bottom of the source code:

```
abstract public void speed();
```

Once you have an abstract method in a class, the entire class must also be declared as abstract. This indicates that a class that includes at least one abstract method (and is therefore an abstract class) cannot be instantiated. So

add the `abstract` keyword to the beginning of the `MammalClass` declaration so that it looks like this:

```
abstract public class MammalClass {

  public String getName() {
  ...
```

Now each class that extends `MammalClass` must implement a `speed()` method. So add this method to the `DogClass` code below the `DogClass()` constructor:

```
public void speed() {
  JOptionPane.showMessageDialog(null, "30 mph", "Dog Speed", 1);
}
```

Add this `speed()` method to the `ManClass` code:

```
public void speed() {
  JOptionPane.showMessageDialog(null, "17 mph", "Man Speed", 1);
}
```

Because each `speed()` method creates a `JOptionPane` component, which is a Swing component, add this statement just after the package statement to the top of both `DogClass` and `ManClass`:

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

This statement makes the entire Swing library available to these classes. You'll read more about import statements soon.

Polymorphism

Polymorphism is the ability for two separate yet related classes to receive the same message but to act on it in their own way. In other words, two different (but related) classes can have the same method name, but they implement the method in different ways.

Therefore, you can have a class method that is also implemented in a child class, and you can access the code from the parent's class (similar to the automatic constructor chaining discussed earlier). Just as in the constructor example, you can use the keyword `super` to access any methods or member variables of the superclass.

Here is a simple example. We have two classes, `Parent` and `Child`.

```
class Parent {
  int aValue = 1;
  int someMethod(){
    return aValue;
  }
}

class Child extends Parent {
  int aValue;                          // this aValue is part of this class
  int someMethod() {                   // this overrides Parent's method
```

```
    aValue = super.aValue + 1;        // access Parent's aValue with super
    return super.someMethod() + aValue;
  }
}
```

The `someMethod()` of `Child` overrides the `someMethod()` of `Parent`. A method of a child class with the same name as a method in the parent class, but that is implemented differently and therefore has different behavior is an `overridden` method.

Can you see how the `someMethod()` of the `Child` class would return the value of 3? The method accesses the `aValue` variable of `Parent` using the `super` keyword, adds the value of 1 to it, and assigns the resulting value of 2 to its own `aValue` variable. The last line of the method calls the `someMethod()` of `Parent`, which simply returns `Parent.aValue` with a value of 1. To that, it adds the value of `Child.aValue`, which was assigned the value of 2 in the previous line. So 1 + 2 = 3.

Using interfaces

An interface is much like an abstract class but with one important difference: an interface cannot include any code. The interface mechanism in Java is meant to replace multiple inheritance.

An interface is a specialized class declaration that can declare constants and method declarations, but not method implementations. You can never put code in an interface.

Here is an interface declaration for our sample application:

You can use the JBuilder Interface wizard to start an interface:

1. Choose File|New to open the object gallery and click the General tab. Double-click the Interface icon to display the Interface wizard.
2. Specify the name of the interface as `SoundInterface`, keeping all other values unchanged. (You can uncheck the Generate Header Comments to omit headers.)
3. Choose OK to generate the new interface.

Within the new `SoundInterface`, add a `speak()` method declaration so that the interface looks like this:

```
package oop1;

public interface SoundInterface {

  public void speak();
}
```

Note that the `interface` keyword is used instead of `class`. All methods declared in an interface are public by default, so there is no need to specify accessibility. A class can implement an interface by using the `implements` keyword. Also, a class can extend only one other class, but a class can implement as many interfaces as necessary. This is how situations that are

usually handled by multiple inheritance in other languages are handled by interfaces in Java. In many cases, you can treat the interface as if it were a class. In other words, you can treat objects that implement an interface as subclasses of the interface for convenience. Note, however, that you can only access the methods defined by that interface if you are casting an object that implements the interface.

The following is an example of both polymorphism and interfaces. We want the `MammalClass` definition to implement the new `SoundInterface`. You do that by adding the words `implements SoundInterface` to the class definition. Then, you must define and implement a `speak()` method for `MammalClass`. Modify your `MammalClass` so that it implements `SoundInterface` and a `speak()` method. Here is the code for `MammalClass` in its entirety:

```
package oop1;

import javax.swing.*;

abstract public class MammalClass implements SoundInterface {

  // accessor methods for properties
  // name
  public String getName() {
    return name;
  }

  public void setName( String value ) {
    name = value;
  }

  // eyecolor
  public String getEyeColor() {
    return eyeColor;
  }

  public void setEyeColor( String value ) {
    eyeColor = value;
  }

  // sound
  public String getSound() {
    return sound;
  }

  public void setSound( String value ) {
    sound = value;
  }

  // age
  public int getAge() {
    return age;
  }
```

```
  public void setAge( int value ) {
    if ( value > 0 )
    {
      age = value;
    }
    else
      age = 0;
  }

  public MammalClass() {
    setName("The Name");
    setEyeColor("Brown");
    setAge(0);
  }

public void speak() {
    JOptionPane.showMessageDialog(null, this.getSound(),
    this.getName() + " Says", 1);
  }


  abstract public void speed();

  private String name, eyeColor, sound;
  private int age;
}
```

The `MammalClass` definition now implements the `SoundInterface` fully. Because the `speak()` method implementation uses the `JOptionPane` component, which is part of the Swing library, you must add an import statement near the top of the file:

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

This import statement makes the entire Swing library available to `MammalClass`. You'll read more about import statements in "The import statement" on page 93.

Because `DogClass` and `ManClass` extend `MammalClass`, they now automatically have access to the `speak()` method defined in `MammalClass`. They don't have to specifically implement `speak()` themselves. The value of the `sound` variable passed to the `speak()` method is set in the constructors of `DogClass` and `ManClass`. Here is how the `DogClass` class should look:

```
package oop1;
import javax.swing.*;

public class DogClass extends MammalClass{

  public boolean hasTail() {
    return tail;
  }
```

```
  public void setTail(boolean value) {
    tail = value;
  }

  public DogClass() {
    setName("Snoopy");
    setSound("Woof, Woof!");
    setAge(2);
    setTail(true);
  }

  public void speed() {
    JOptionPane.showMessageDialog(null, "30 mph", "Dog Speed", 1);
  }

  private boolean tail;
}
```

This is how ManClass should look:

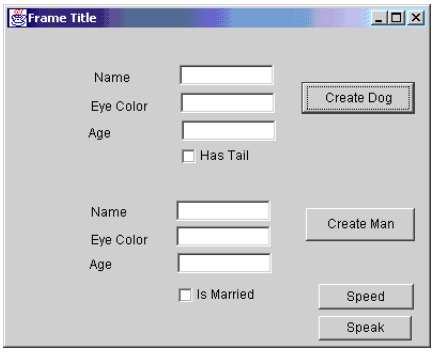
```
package oop1;
import javax.swing.*;

 public class ManClass extends MammalClass {

  public boolean isMarried() {
    return married;
  }

  public void setMarried(boolean value) {
    married = value;
  }

  public ManClass() {
    setName("Steven");
    setEyeColor("Blue");
    setSound("Hello there! I'm " + this.getName() + ".");
    setAge(35);
    setMarried(true);
  }

  public void speed() {
    JOptionPane.showMessageDialog(null, "17 mph", "Man Speed", 1);
  }

  private boolean married;
}
```

Adding two new buttons

Although you've added `speak()` and `speed()` methods to the sample application, so far the application never calls them. To change this, add two more buttons to the `Frame1.java` class:

1. Click the Frame1 tab in the content pane.
2. Click the Design tab to display the UI designer.
3. Place two additional buttons on the form.
4. Change the value of `text` property of the first button to Speed in the Inspector, and change the value of the `text` property of the second button to Speak.

Figure 6.2 New version of the sample application with Speed and Speak buttons added



Click the Source tab to return to the `Frame1.java` code and add code shown here in bold to the class definition:

```
// Create a reference for the objects
DogClass dog;
ManClass man;

//Create an Array of SoundInterface
SoundInterface soundList[] = new SoundInterface[2];

//Create an Array of Mammal
MammalClass mammalList[] = new MammalClass[2];
```

You have added code that creates two arrays: one for Mammals and one for SoundInterfaces.

Also add code to the Create Dog and Create Man event handlers that add references to the dog and man objects to the arrays:

```
void button1_actionPerformed(ActionEvent e) {
  dog = new DogClass();
  txtfldDogName.setText(dog.getName());
  txtfldDogEyeColor.setText(dog.getEyeColor());
  txtfldDogAge.setText(Integer.toString(dog.getAge()));
  chkboxDog.setSelected(true);
  mammalList[0] = dog;
```

```
    soundList[0]  = dog;
  }

  void button2_actionPerformed(ActionEvent e) {
    man = new ManClass();
    txtfldManName.setText(man.getName());
    txtfldManEyeColor.setText(man.getEyeColor());
    txtfldManAge.setText(Integer.toString(man.getAge()));
    chkboxMan.setSelected(true);
    mammalList[1] = man;
    soundList[1]  = man;
    }
```

Return to the UI designer and double-click the Speed button, and fill in the event handler JBuilder starts for you so that the code looks like this:

```
  void button3_actionPerformed(ActionEvent e) {
  for (int i = 0; i <= 1; i++) {
      mammalList[i].speed();
    }
  }
```

The code loops through the list of mammals held in the array (all two of them!) and tells each object to display its speed. The first time through the list, the dog displays its speed, the second time through the list, the man displays its speed. This is polymorphism in action—two separate but related objects receiving the same message and reacting to it in their own way.

The code for the Speak button is very similar.

```
  void button4_actionPerformed(ActionEvent e) {
  for (int i = 0; i <= 1; i++) {
      soundList[i].speak();
    }
  }
```

Choose File|Save All to save all your changes.

You can see that you can treat the `SoundInterface` as a class when it is convenient. Note that the interface gives you many of the benefits of multiple inheritance without the added complexity.

Running your application

You're ready to run your modified application. Choose Run|Run Project to recompile your project and then run it.

When your application begins running, be sure that you click the Create Dog and Create Man buttons to create the `dog` and `man` objects before you try the Speed and Speak buttons or you will get a `NullPointerException`.

Once your objects exist and you click the Speed button, a message box appears reporting the speed of the first mammal in the `mammalList` array, the dog. When you click OK to remove the message box, the second message box appears. It reports the speed of the second mammal, the man. Clicking the Speak button results in similar behavior, but the messages displayed are sounds each mammal might make.

Java packages

To facilitate code reuse, Java allows you to group several class definitions together in a logical grouping called a *package*. If, for instance, you create a group of business rules that model the work processes of your organization, you might want to place them together in a package. This makes it easier to reuse code that you have previously created.

The import statement

The Java language comes with many predefined packages. For instance, the `java.applet` package contains classes for working with Java applets:

```
public class Hello extends java.applet.Applet {
```

This code refers to the class called `Applet` in the Java package `java.applet`. You can imagine that it might get quite tedious to have to repeat the entire full class name `java.applet.Applet` every time you refer to this class. Instead, Java offers an alternative. You can choose to import a package you will use frequently:

```
import java.applet.*;
```

This tells the compiler "if you see a class name you do not recognize, look in the `java.applet` package for it." Now, when you declare a new class, you can say,

```
public class Hello extends Applet {
```

This is more concise. You have a problem, however, if you have two classes by the same name defined in two different imported packages. In this case, you must use the fully qualified name.

Declaring packages

Creating your own packages is almost as easy as using them. For instance, if you want to create a package called `mypackage`, you would simply use a `package` statement at the beginning of your file:

```
package mypackage;

public class Hello extends java.applet.Applet {

  public void init() {
    add(new java.awt.Label("Hello World Wide Web!"));
  }

} // end class
```

Now, any other program can access the classes declared in `mypackage` with the statement:

```
import mypackage.*;
```

Remember, this file should be in a subdirectory called `mypackage`. This allows your Java compiler to easily locate your package. JBuilder's Project wizard will automatically set the directory to match the project name. Also, keep in mind that the base directory of any package you import must be listed in the Source Path of the JBuilder IDE or the Source Path of your project. This is good to remember if you decide to relocate a package to a different base directory.

For more information about working with packages in JBuilder, see "Packages" in *Building Applications with JBuilder*.

Chapter

7

Threading techniques

Threads are a part of every Java program. A thread is a single sequential flow of control within a program. It has a beginning, a sequence, and an end. A thread cannot run on its own; it runs within a program. If your program is a single sequence of execution, you don't need to set up a thread explicitly, the Java Virtual Machine (VM) will take care of this for you.

One of the powerful aspects of the Java language is that you can easily program multiple threads of execution to run concurrently within the same program. For example, a web browser can download a file from one site, and access another site at the same time. If the browser can't do two simultaneous tasks, you'd need to wait until the file had finished downloading before you could browse to another site.

The Java VM always has multiple threads, called *daemon threads*, running. For example, a continually running daemon thread performs garbage collection tasks. Another daemon thread handles mouse and keyboard events. It is possible for your program to lock up one of the Java VM threads. If your program appears to be dead, with no events being sent to your program, try using threads.

The lifecycle of a thread

Every thread has a definite lifecycle—it starts and stops, it can pause and wait for an event, and it can notify another thread while it is running. This section will introduce some of the more common aspects of the thread lifecycle.

Customizing the run() method

Use the `run()` method to implement the thread's running behavior. This behavior can be anything a Java statement can accomplish—calculations, sorting, animations, etc.

You can use one of two techniques to implement the `run()` method for a thread:

- Subclass the `java.lang.Thread` class
- Implement the `java.lang.Runnable` interface

Subclassing the Thread class

If you are creating a new class whose objects you want to execute in separate threads, you need to subclass the `java.lang.Thread` class. The `Thread` class's default `run()` method does not do anything, so your class will need to override the `run()` method. The `run()` method is the first thing that executes when a thread is started.

As an example, the following class, `CountThread`, subclasses `Thread` and overrides its `run()` method. In this example, the `run()` method identifies a thread and prints its name to the screen. The `for` loop counts integers from the `start` value to the `finish` value and prints each count to the screen. Then, before the loop finishes execution, the method prints a string that indicates the thread has finished executing.

```
public class CountThread extends Thread {
   private int start;
   private int finish;

   public CountThread(int from, int to) {
     this.start = from;
     this.finish = to;
   }

   public void run() {
      System.out.println(this.getName()+ " started executing...");
      for (int i = start; i <= finish; i++) {
         System.out.print (i + " ");
      }
   System.out.println(this.getName() + " finished executing.");
   }
}
```

To test the `CountThread` class, you can create a test class:

```
public class ThreadTester {
   static public void main(String[] args) {
      CountThread thread1 = new CountThread(1, 10);
      CountThread thread2 = new CountThread(20, 30);
      thread1.start();
      thread2.start();
   }
}
```

The `main()` method in the test application creates two `CountThread` objects: `thread1` that counts from 1 to 10, and `thread2` that counts from 20 to 30. Both threads are then started by calling their `start()` methods. The output from this test application could look like this:

```
Thread-0 started executing...
1 2 3 4 5 6 7 8 9 10 Thread-0 finished executing.
Thread-1 started executing...
20 21 22 23 24 25 26 27 28 29 30 Thread-1 finished executing.
```

Notice that the output does not show the thread names as `thread1` and `thread2`. Unless you specifically assign a name to a thread, Java will automatically give it a name of the form `Thread-n`, where `n` is a unique number, starting with 0. You can assign a name to a thread in the class constructor or with the `setName(String)` method.

In this example, `Thread-0` started executing first and finished first. However, it could have started first and finished last, or partially started and been interrupted by `Thread-1`. This is because threads in Java are not guaranteed to execute in any particular sequence. In fact, each time you execute `ThreadTester`, you might get a different output. Basically, the process of scheduling threads is controlled by the Java thread scheduler, and not the programmer. For more information, see the topic called "Thread priority" on page 101.

Implementing the Runnable interface

If you want objects of an existing class to execute in their own threads, you can implement the `java.lang.Runnable` interface. This interface adds threading support to classes that do not inherit from the `Thread` class. It provides only one method, the `run()` method, which you have to implement for your class.

Note If your class subclasses a class other than `Thread`, for example, `Applet`, you should use the `Runnable` interface to create threads.

To create a new `CountThread` class that implements the `Runnable` interface, you need to change the class definition of the `CountThread` class. The class definition code, with the changes highlighted in bold-faced type, would look like this:

```
public class CountThread implements Runnable {
```

You would also have to change the way the name of the thread is obtained. Because you are not instantiating the class `Thread`, you cannot call the `getName()` method of `CountThread's` superclass, in this case, `java.lang.Object`. This method is not available. Instead, you need to specifically use the `Thread.currentThread()` method, which returns the thread's name in a format that is slightly different from the `getName()` method.

The entire class, with changes highlighted in bold-faced type, would then look like this:

```
public class CountThread implements Runnable {
   private int start;
   private int finish;
```

```
    public CountThread(int from, int to) {
       this.start = from;
       this.finish = to;
    }

    public void run() {
       System.out.println(Thread.currentThread() + " started executing...");
       for (int i=start; i <= finish; i++) {
          System.out.print (i + " ");
       }
    System.out.println(Thread.currentThread() + " finished executing.");
    }
}
```

The test application would need to change the way its objects are created. Instead of instantiating `CountThread`, the application needs to create a `Runnable` object from the new class and pass it to one of the thread's constructors. The code, with the changes highlighted in bold-faced type, would look like this:

```
public class ThreadTester {
   static public void main(String[] args) {
      CountThreadRun thread1 = new CountThreadRun(1, 10);
      new Thread(thread1).start();
      CountThreadRun thread2 = new CountThreadRun(20, 30);
      new Thread(thread2).start();
  }
}
```

The output from this program would look like this:

```
Thread[Thread-0,5,main] started executing...
1 2 3 4 5 6 7 8 9 10 Thread[Thread-0,5,main] finished executing.
Thread[Thread-1,5,main] started executing...
20 21 22 23 24 25 26 27 28 29 30 Thread[Thread-1,5,main] finished
    executing.
```

`Thread-0` is the name of the thread, `5` is the priority the thread was given when it was created, and `main` is the default `ThreadGroup` to which the thread was assigned. (The priority and the group are assigned by the Java VM if none are specified.)

See also

- "Thread priority" on page 101
- "Thread groups" on page 102

Defining a thread

The `Thread` class provides seven constructors. These constructors combine the following three parameters in various ways:

- A `Runnable` object whose `run()` method will execute inside the thread.
- A `String` object to identify the thread.
- A `ThreadGroup` object to assign the thread to. The `ThreadGroup` class organizes groups of related threads.

Constructor	Description
`Thread()`	Allocates a new `Thread` object.
`Thread(Runnable target)`	Allocates a new `Thread` object so that it has `target` as its run object.
`Thread(Runnable target, String name)`	Allocates a new `Thread` object so that it has `target` as its run object and the specified `name` as its name.
`Thread(String name)`	Allocates a new `Thread` object so that it has the specified `name` as its name.
`Thread(ThreadGroup group, Runnable target)`	Allocates a new `Thread` object so that it belongs to the thread group referred to by `group` and has `target` as its run object.
`Thread(ThreadGroup group, Runnable target, String name)`	Allocates a new `Thread` object so that it has `target` as its run object, the specified `name` as its name, and belongs to the thread group referred to by `group`.
`Thread(ThreadGroup group, String name)`	Allocates a new `Thread` object so that it belongs to the thread group referred to by `group` and has the specified `name` as its name.

If you want to associate state with a thread, use a `ThreadLocal` object when you create the thread. This class allows each thread to have its own independently initialized copy of a private static variable, for example, a user or transaction ID.

Starting a thread

To start a thread call the `start()` method. This method creates the system resources necessary to run the thread, schedules the thread, and calls the thread's `run()` method.

After the `start()` method returns, the thread is running and is in a runnable state. Because most computers have only a single CPU, the Java VM must schedule threads. For more information see the topic called "Thread priority" on page 101.

Making a thread not runnable

To put a thread into a not runnable state, use one of following techniques:

- A `sleep()` method: these methods allow you to specify a specific number of seconds and nanoseconds to not run.
- The `wait()` method: this method causes the current thread to wait for a specified condition to be met.
- Block the thread on input or output.

When the thread is not runnable, the thread will not run, even if the processor becomes available. To exit the not runnable state, the condition for the entrance to the not runnable state must be met. For example, if you used the `sleep()` method, the specified number of seconds must have passed. If you used the `wait()` method, another object must tell the waiting thread (with `notify()` or `notifyAll()`) of a change in condition. If a thread is blocked by input or output, the input or output must finish.

You can also use the `join()` method to have a thread wait for an executing thread to finish. You call this method for the thread being waited on. You can specify a timeout for a thread by passing a parameter to the method in milliseconds. The `join()` method waits on the thread until either the timeout has expired or the thread has terminated. This method works in conjunction with the `isAlive()` method—`isAlive()` returns `true` if the thread has been started and not stopped.

Note that the `suspend()` and `resume()` methods have been deprecated. The `suspend()` method is deadlock-prone. If the target thread is locking a monitor that protects a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. A monitor is a Java object used to verify that only one thread at a time is executing the synchronized methods for the object. For more information, see the topic called “Synchronizing threads” on page 101.

Stopping a thread

You can no longer stop a thread with the `stop()` method. This method has been deprecated, as it is unsafe. Stopping a thread will cause it to unlock all of the monitors it has locked. If an object previously protected by one of these monitors is in an inconsistent state, other threads will see that object as inconsistent. This can cause your program to be corrupted.

To stop a thread, terminate the `run()` method with a finite loop.

For more information, see the topic in the *Java 2 SDK, Standard Edition Documentation* called “Why are Thread.stop, Thread.suspend, Thread.resume and runtime.runFinalizersOnExit Deprecated?”.

Thread priority

When a Java thread is created, it inherits its priority from the thread that created it. You can set a thread's priority using the `setPriority()` method. Thread priorities are represented as integer values ranging from `MIN_PRIORITY` to `MAX_PRIORITY` (constants in the `Thread` class). The thread with the highest priority is executed.

When that thread stops, yields, or becomes not runnable, a lower priority thread will be executed. If two threads of the same priority are waiting, the Java scheduler will choose one of them to run in a round-robin fashion. The thread will run until:

- A higher priority thread becomes runnable.
- The thread yields, by use of the `yield()` method, or its `run()` method exits.
- Its time allotment has expired. This only applies to systems that support time slicing.

This type of scheduling is based on a scheduling algorithm called *fixed priority scheduling.* Threads are run based on their priority when compared to other threads. The thread with the highest priority will always be running.

Time slicing

Some operating systems use a scheduling mechanism knows as *time-slicing.* Time-slicing divides the CPU into time slots. The system gives the highest priority threads that are of equal priority time to run, until one or more of them finishes, or until a higher priority thread is in a runnable state. Because time-slicing is not supported on all operating systems, your program should not depend on a time-slicing scheduling mechanism.

Synchronizing threads

One of the central problems of multithreaded computing is handling situations where more than one thread has access to the same data structure. For example, if one thread was trying to update the elements in a list, while another thread was simultaneously trying to sort them, your program could deadlock or produce incorrect results. To prevent this problem, you need to use *thread synchronization.*

The simplest way to prevent two objects from accessing the same method at the same time is to require a thread to obtain a lock. While a thread holds the lock, another thread that needs a lock has to wait until the first thread releases the lock. To keep a method *thread-safe,* use the `synchronized` keyword when declaring methods that can only be executed by one thread at a time. Note than you can also synchronize on an object.

For example, if you create a `swap()` method that swaps values using a local variable and you create two different threads to execute the method, your

program could produce incorrect results. The first thread, due to the Java scheduler, might only be able to execute the first half of the method. Then, the second thread might be able to execute the entire method, but using incorrect values (since the first thread did not complete the operation). The first thread would then return to finish the method. In this case, it would appear as if the swapping of values never took place. To prevent this from happening, use the `synchronized` keyword in your method declaration.

As a basic rule, any method that modifies an object's property should be declared `synchronized`.

Thread groups

Every Java thread is a member of a *thread group.* A thread group collects multiple threads into a single object and manipulates all those threads at once. Thread groups are implemented by the `java.lang.ThreadGroup` class.

The runtime system puts a thread into a thread group during thread construction. The thread is either put into a default group or into a thread group you specify when the thread is created. You cannot move a thread into a new group once the thread has been created.

If you create a thread without specifying a group name in its constructor, the runtime system places the new thread in the same group as the thread that created it. Usually, unspecified threads are put into the `main` thread group. However, if you create a thread in an applet, the new thread might be put into a thread group other than `main`, depending on the browser or viewer the applet is running in.

If you construct a thread with a `ThreadGroup`, the group can be:

- A name of your own creation
- A group created by the Java runtime
- A group created by the application in which your applet is running

To obtain the name of the group your thread is part of, use the `getThreadGroup()` method. Once you know a thread's group, you can determine what other threads are in the group and manipulate them all at once.

Chapter

8

Serialization

Object serialization is the process of storing a complete object to disk or other storage system, ready to be restored at any time. The process of restoring the object is known as *deserialization*. In this section, you'll learn why serialization is useful and how Java implements serialization and deserialization.

An object that has been serialized is said to be *persistent*. Most objects in memory are *transient*, meaning that they go away when their references drop out of scope or the computer loses power. Persistent objects exist as long as there is a copy of them stored somewhere on a disk, tape, or in ROM.

Why serialize?

Traditionally, saving data to a disk or other storage device required that you define a special data format, write a set of functions to write and read that format, and create a mapping between the file format and the format of your data. The functions to read and write data were either simple and lacked extensibility, or they were complex and difficult to create and maintain.

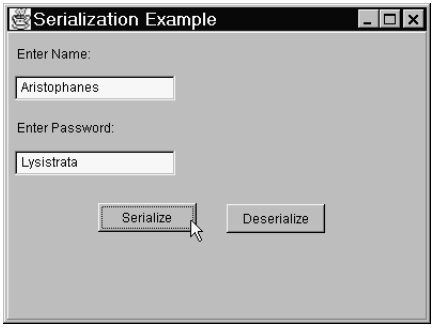
Java is completely based around objects and object-oriented programming and provides a storage mechanism for objects in the form of serialization. Using the Java way of doing things, you no longer have to worry about details of file formats and input/output (I/O). Instead, you can concentrate on solving your real-world tasks by designing and implementing objects. If, for instance, you make a class persistent and later add new fields to it, you don't have to worry about modifying routines that read and write the data for you. All fields in a serialized object will automatically be written and restored.

Java serialization

Serialization first appeared as a feature of JDK 1.1. Java's support for serialization consists of the `Serializable` interface, the `ObjectOutputStream` class and the `ObjectInputStream` class, as well as a few supporting classes and interfaces. We'll examine all three of these items as we demonstrate an application that can save user information to a disk and read it back.

Suppose, for instance, you wanted to save information about a particular user as shown here.

Figure 8.1 Saving a user name and password



After the user types in his or her name and password into the appropriate fields, the application should save information about this user to disk. Of course, this is a very simple example, but you can easily imagine saving data about user application preferences, the last document opened, and so on.

Using JBuilder, you can design a user interface like the one shown above. See the *Designing Applications with JBuilder* book if you need help with this task. Name the Name text field `textFieldName`, and the password field `passwordFieldName`. Besides the two labels you can see, add a third one near the bottom of the frame and name it `labelOutput`.

Using the Serializable interface

Create a new class that represents the current user. It must have properties that represent the user's name and the user's password.

To create the new class,

1. Choose File|New Class to display the Class wizard.
2. In the Class Information section, specify the new class name as UserInfo. Leave the other fields unchanged.
3. In the Options section, check just the Public and Generate Default Constructor options, unchecking all others.
4. Choose OK.

The Class wizard creates the new class file for you and adds it to the project. Modify the generated code so that it looks like this:

```
package serialize;

public class UserInfo implements java.io.Serializable {
    private String userName = "";
    private String userPassword = "";

  public UserInfo() {
  }

  public String getUserName() {
    return userName;
  }

  public void setUserName(String s) {
    userName = s;
  }

  public String getUserPassword() {
    return userPassword;
  }

  public void setUserPassword(String s) {
    userPassword = s;
  }
}
```

You've added a variable that holds the user's name and another for the user's password. You've also added accessor methods to both fields.

You'll note that the `UserInfo` class implements the `java.io.Serializable` interface. `Serializable` is known as a *tagging interface* because it specifies no methods to be implemented, but merely "tags" its objects as being of a particular type.

Any object that you expect to serialize must implement this the `Serializable` interface. This is critical because the techniques used later in this chapter won't work otherwise. If, for instance, you try to serialize an object that does not implement this interface, a `NotSerializableException` will be raised.

At this point, you should import the `java.io` package so that your application has access to the input and output classes and interfaces to needs to write and read objects. Add this import statement to those at the top of your frame class:

```
import java.io.*
```

Using output streams

Before you serialize the `UserInfo` object, you must instantiate it and set it up with the values that the user enters into the text fields. When the user enters

information in the fields and clicks the Serialize button, the values the user entered are stored in the `UserInfo` object instance:

```
void jButton1_actionPerformed(ActionEvent e) {
    UserInfo user = new UserInfo();            // instantiate a user object
    user.setUserName(textFieldName.getText());
    user.setUserPassword(textFieldPassword.getText());
}
```

If you are using JBuilder's UI designer, double-click the Serialize button and JBuilder starts the `jButton1_actionPerformed()` event code for you. Instantiate a user object, then add the `user.setUserName()` and `user.setUserPassword()` method calls to the event handler.

Next, open a `FileOutputStream` to the file that will contain the serialized data. In this example, the file will be called `C:\userInfo.ser`. Add this code to the Serialize button event handler:

```
try {
  FileOutputStream file = new FileOutputStream("c:\userInfo.ser");
```

Create an `ObjectOutputStream` that will serialize the object and send it to the `FileOutputStream` by adding this code to the event handler:

```
ObjectOutputStream out = new ObjectOutputStream(file);
```

Now you're ready to send the `UserInfo` object to the file. Do this by calling the `ObjectOutputStream's writeObject()` method. Call the `flush()` method to flush the output buffer to ensure that the object is actually written to the file.

```
out.writeObject(u);
out.flush();
```

Close the output stream to free up any resources, such as file descriptors, used by the stream.

```
  out.close();
}
```

Add code to the handler that catches an `IOException` if there were any problems writing to the file or if the object does not support the *Serializable* interface.

```
catch (java.io.IOException IOE) {
      labelOutput.setText("IOException");
}
```

This is how the event handler for the Serialize button should look in its entirety:

```
void jButton1_actionPerformed(ActionEvent e) {
    UserInfo user = new UserInfo();
    user.setUserName(textFieldName.getText());
    user.setUserPassword(textFieldPassword.getText());
    try {
       FileOutputStream file = new FileOutputStream("c:\userInfo.ser");
       ObjectOutputStream out = new ObjectOutputStream(file);
       out.writeObject(user);
       out.flush();
    }
```

```
        catch (java.io.IOException IOE) {
           labelOutput.setText("IOException");
        }
        finally {
           out.close();
        }
     }
```

Now compile your project and run it. Enter values in the Name and Password fields and click the Serialize button. You can verify that the object has been written by opening it in a text editor. (Don't try to edit it, or the file will probably be corrupted!) Notice that a serialized object contains a mixture of ASCII text and binary data:

Figure 8.2 The serialized object

ObjectOutputStream methods

The `ObjectOutputStream` class contains several useful methods for writing data to a stream. You aren't restricted to writing objects. Calling `writeInt()`, `writeFloat()`, `writeDouble()`, and so on, will write any of the fundamental types to a stream. If you want to write more than one object or fundamental type to the same stream, you can do so by repeatedly calling these methods against the same `ObjectOutputStream` object. When you do this, however, you must read the objects back *in the same order*.

Using input streams

You have now written the object to the disk, but how do you get if back? Once the user clicks the Deserialize button, you want to read the data back from the disk into a new object.

You can begin the process by creating a new `FileInputStream` object to read from the file you just wrote. If you are using JBuilder, double-click the Deserialize button in the UI Designer, and in the event handler that JBuilder creates for you, add the highlighted code:

```
void jButton2_actionPerformed(ActionEvent e) {
try {
  FileInputStream file = new FileInputStream("c:\userInfo.ser");
```

Next, create an `ObjectInputStream`, which gives you the capability to read objects from that file.

```
ObjectInputStream input = new ObjectInputStream(file);
```

After this, call the `ObjectInputStream.readObject()` method to read the first object from the file. `readObject()` returns type `Object`, so you'll want to cast it to the appropriate type `(UserInfo)`.

```
UserInfo user = (UserInfo)input.readObject();
```

When you're done reading, remember to close the `ObjectInputStream`, so you free up any resources associated with it, such as file descriptors.

```
input.close();
```

Finally, you can use the `user` object as you would any other object of the `UserInfo` class. In this case, you display the name and password in the third label field you added to the dialog box:

```
labelOutput.setText("Name is " + user.getUserName() +
         ", password is: " +
        user.getUserPassword());
```

Reading from a file could cause an `IOException`, so you should handle this exception. You might also get a `StreamCorruptedException` (a subclass of `IOException`) if the file has been corrupted in any way:

```
catch (java.io.IOException IOE) {
  labelOutput.setText("IOException");
}
```

There's another exception you must deal with. The `readObject()` method can throw a `ClassNotFoundException`. This exception can occur if you attempt to read an object for which you have no implementation. For instance, if this object was written by another program, or you have renamed the `UserInfo` class since the file was written, you'll get a `ClassNotFoundException`.

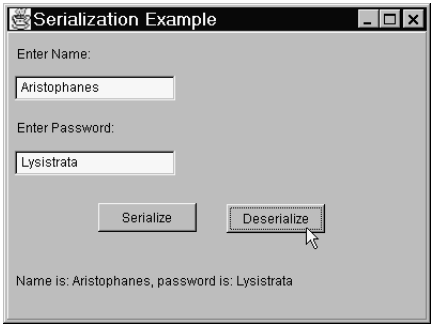
```
  catch (ClassNotFoundException cnfe) {
    labelOutput.setText("ClassNotFoundException");
  }
}
```

Here is the Deserialize button event handler in its entirety:

```
void jButton2_actionPerformed(ActionEvent e) {
  try {
    FileInputStream file = new FileInputStream("c:\userInfo.ser");
    ObjectInputStream input = new ObjectInputStream(file);
    UserInfo user = (UserInfo)input.readObject();
    input.close();
    labelOutput.setText("Name is " + user.getUserName() +
    ", password is: " +
    user.getUserPassword());
    }
  catch (java.io.IOException IOE) {
    labelOutput.setText("IOException");
  }
  catch (ClassNotFoundException cnfe) {
    labelOutput.setText("ClassNotFoundException");
  }
}
```

Now when you compile and run your project, enter Name and Password values and click the Serialize button to store the information on your disk. Then click the Deserialize button to read the serialized `UserInfo` object back again.

Figure 8.3 The object restored



ObjectInputStream methods

`ObjectInputStream` also has methods such as `readDouble()`, `readFloat()`, and so on, which are the counterparts to the `writeDouble()`, `writeFloat()`, and such methods. You must call each method in sequence, the same way the objects were written to the stream.

Writing and reading object streams

You might wonder what happens when an object you are serializing contains a field that refers to another object, rather than a primitive type. In this case, both the base object and the secondary object will be written to the stream. You should realize, however, that both objects written to the stream need to implement the `Serializable` interface. If they don't, a `NotSerializableException` will be thrown when the `writeObject()` method is called.

Recall that object serialization can create potential security problems. In the example above, we wrote a password to a serialized object. While this technique might be acceptable in some circumstances, keep security issues in mind when you choose to serialize an object.

Finally, if you want to create a persistent object, but don't want to use the default serialization mechanism, the `Serializable` interface documents two methods, `writeObject()` and `readObject()`, which you can implement to perform custom serialization. The `Externalizable` interface also provides a similar mechanism. Consult the JDK documentation for information about these techniques.

Chapter

9

An introduction to the Java Virtual Machine

This chapter provides an introduction to the Java Virtual Machine (JVM). While it is important for you to be familiar with basic information concerning the JVM, unless you get into very advanced Java programming, the JVM is typically something you don't need to worry about. This chapter is for your information only.

Before exploring the Java Virtual Machine, we will explain some of the terminology used in this chapter. First, the Java Virtual Machine (JVM) is the environment in which Java programs execute. The Java Virtual Machine specification essentially defines an abstract computer, and specifies the instructions that this computer can execute. These instructions are called *bytecodes*. Generally speaking, Java bytecodes are to the JVM what an instruction set is to a CPU. A bytecode is a byte-long instruction that the Java compiler generates, and the Java interpreter executes. When the compiler compiles a `.java` file, it produces a series of bytecodes and stores them in a `.class` file. The Java interpreter can then execute the bytecodes stored in the `.class` file.

Other terminology used in this chapter involves Java applications and applets. It is sometimes appropriate to distinguish between a Java application and a Java applet. In some sections of this chapter, however, that distinction is inappropriate. In such cases, we will use the word *app* to refer to both Java applications and Java applets.

It is important here to clarify what Java really is. Java is more than just a computer language; it is a computer *environment*. This is because Java is

composed of two separate main elements, each of which is an essential part of Java: the design-time Java (the Java language itself) and the runtime Java (the JVM). This interpretation of the word *Java* is a more technical one.

Interestingly enough, the practical interpretation of the word *Java* is that it stands for the runtime environment — not the language. When you say something like “this machine can run Java,” what you really mean is that the machine supports the Java Runtime Environment (JRE); more precisely, it implements a Java Virtual Machine.

A distinction should be made between the *Java Virtual Machine Specification* and an *implementation* of the Java Virtual Machine. The JVM specification is a document (available from Sun’s website) which defines how to implement a JVM. When an implementation of the JVM correctly follows this specification, it essentially ensures that Java apps can run on this implementation of the JVM with the same results those same Java apps produce when running on any other implementation of the JVM. The JVM specification ensures that Java programs will be able to run on any platform.

The JVM specification is platform independent, because it can be implemented on any platform. Note that a specific implementation of the JVM is platform dependent. This is because the JVM implementation is the only portion of Java that directly interacts with the operating system (OS) of your computer. Because each OS is different, any specific JVM implementation must know how to interact with the specific OS for which it is intended.

Having Java programs run under an implementation of the JVM guarantees a predictable runtime environment, because all implementations of the JVM conform to the JVM specification. Even though there are different implementations of the JVM, they all must meet certain requirements to guarantee portability. In other words, whatever differs among the various implementations does not affect portability.

The JVM is responsible for performing the following functions:

- Allocating memory for created objects
- Performing garbage collection
- Handling register and stack operations
- Calling on the host system for certain functions, such as device access
- Monitoring the security of Java apps

Throughout the remaining chapter, we will focus on the last function: security.

Java VM security

One of the JVM’s most important roles is monitoring the security of Java apps. The JVM uses a specific mechanism to force certain security restrictions on Java apps. This mechanism (or security model) has the following roles:

- Determines to what extent the code being run is “trusted” and assigns it the appropriate level of access

- Assures that bytecodes do not perform illegal operations
- Verifies that every bytecode is generated correctly

In the following sections, we will see how these security roles are taken care of in Java.

The security model

In this section, we will look at some of the different elements in Java's security model. In particular, we will examine the roles of the Java Verifier, the Security Manager and `java.security` package, and the Class Loader. These are some of the components that make Java apps secure.

The Java verifier

Every time a class is loaded, it must first go through a verification process. The main role of this verification process is to ensure that each bytecode in the class does not violate the specifications of the Java VM. Examples of bytecode violations are type errors and overflowed or underflowed arithmetic operations. The verification process is handled by the Java verifier, and it consists of the following four stages:

1 Verifying the structure of class files.

2 Performing system-level verifications.

3 Validating bytecodes.

4 Performing runtime type and access checks.

The first stage of the verifier is concerned with verifying the structure of the class file. All class files share a common structure; for example, they must always begin with what is called the *magic number*, whose value is `0xCAFEBABE`. At this stage, the verifier also checks that the constant pool is not corrupted (the constant pool is where the class file's strings and numbers are stored). In addition, the verifier makes sure that there are no added bytes at the end of the class file.

The second stage performs system-level verifications. This involves verifying the validity of all references to the constant pool, and ensuring that classes are subclassed properly.

The third stage involves validating the bytecodes. This is the most significant and complex stage in the entire verification process. Validating a bytecode means checking that its type is valid and that its arguments have the appropriate number and type. The verifier also checks that method calls are passed the correct type and number of arguments, and that each external function returns the proper type.

The final stage is where runtime checks take place. At this stage, externally referenced classes are loaded, and their methods are checked. The method check involves checking that the method calls match the signature of the methods in the external classes. The verifier also monitors access attempts by the currently loaded class to make sure that the class does not violate access

restrictions. Another access check is done on variables to ensure that `private` and `protected` variables are not accessed illegally.

From this exhaustive verification process, we can see how important the Java verifier is to the security model. It is also important to note that the verification process must be done at the verifier level, and not at the compiler's, since any compiler can be programmed to generate Java bytecodes. Clearly then, relying on the compiler to perform the verification process is dangerous, since the compiler can be programmed to bypass it. This point illustrates why the JVM is necessary.

If you need more information on the Java verifier, please see the *Java Virtual Machine Specification* at `http://java.sun.com/docs/books/vmspec/index.html`.

The Security Manager and the java.security Package

One of the classes defined in the `java.lang` package is the `SecurityManager` class. This class checks the security policy on Java apps to determine if the running app has permission to perform certain dangerous operations. The security policy's main role is to determine access rights. In Java 1.1, the `SecurityManager` class was solely responsible for setting the security policy, but in Java 2 and above, a much more detailed and robust security model is achieved using the new `java.security` package. The `SecurityManager` class has several methods that begin with "check". In Java 1.1, the default implementation of those "check" methods was to throw a `SecurityException`. Since Java 2, the default implementation of most of the "check" methods calls `SecurityManager.checkPermission()`, and that method's default implementation in turn calls `java.security.AccessController.checkPermission()`. It is `AccessController` which is responsible for the actual algorithm for checking permissions.

The `SecurityManager` class contains many methods used to check whether a particular operation is permitted. The `checkRead()` and `checkWrite()` methods, for example, check whether the method caller has the right to perform a read or write operation, respectively, to a specified file. They do this by calling `checkPermission()`, which in turn calls `AccessController.checkPermission()`. Many of the methods in the JDK use the `SecurityManager` before performing dangerous operations. The JDK does this for legacy reasons; `SecurityManager` existed in earlier versions of the JDK when there was a much more limited security model. In your apps, you may want to call `AccessController.checkPermission()` directly, instead of using the `SecurityManager` class (which calls the same method indirectly anyway).

The static `System.setSecurityManager()` method can be used to load the default security manager into the environment. Now, whenever a Java app needs to perform a dangerous operation, it can consult with the `SecurityManager` object that is loaded into the environment.

The way Java apps use the `SecurityManager` class is generally the same. An instance of `SecurityManager` is first created, either by using a special command

line argument when the app is started ("`-Djava.security.manager`"), or in code similar to the following:

```
SecurityManager security = System.getSecurityManager();
```

The `System.getSecurityManager()` method returns an instance of the currently loaded `SecurityManager`. If no `SecurityManager` has been set using the `System.setSecurityManager()` method, `System.getSecurityManager()` returns **null**; otherwise, it returns an instance of the `SecurityManager` that was loaded into the environment. Now, let's assume that the app wants to check whether it can read a file. It does so as follows:

```
if (security != null) {
  security.checkRead (fileName);
}
```

The `if` statement first checks whether a `SecurityManager` object exists, then it makes the call to the `checkRead()` method. If `checkRead()` does not permit the operation, a `SecurityException` is thrown and the operation never takes place; otherwise, all goes well.

There is typically a security manager loaded when an applet is running, because most Java-enabled browsers automatically use one. An application, on the other hand, does not automatically use a security manager, unless one is loaded into the environment using the `System.setSecurityManager()` method, or from the command line when starting the application. To use the same security policy for an application as for an applet, you must make sure the security manager is loaded.

In order to specify your own security policy, you will need to work with the classes in the `java.security` package. Important classes in this package include `Policy`, `Permission`, and `AccessController`. You should not subclass `SecurityManager` except as a last resort, and then with extreme caution. An in-depth discussion of the `security` package is outside the scope of this book. The default security policy should suffice for most beginning Java developers. When you do find you are concerned with more advanced security topics, or just for more information on the `java.security` package, please see the "Security Architecture" document in the JDK documentation.

The class loader

The class loader works alongside the security manager to monitor the security of Java apps. The main roles of the class loader are summarized below:

- Determines whether the class it is attempting to load has already been loaded
- Loads class files into the Virtual Machine
- Determines the permissions assigned to the loaded class in accordance with the security policy
- Provides certain information about loaded classes to the security manager
- Determines the path from which the class should be loaded (System classes are always loaded from the `BOOTCLASSPATH`)

Each instance of a class is associated with a class loader object, which is an instance of a subclass of the abstract class `java.lang.ClassLoader`. Class loading happens automatically when a class is instantiated. It is possible to create a custom class loader by subclassing `ClassLoader` or one of its existing subclasses, but in most cases this is not necessary. If you need more information about the class loader mechanism, see the documentation for `java.lang.ClassLoader` and the "Security Architecture" document in the JDK documentation.

So far, we've seen how the Java verifier, the `SecurityManager`, and the class loader work to ensure the security of Java apps. In addition to these, there are other mechanisms not described in this chapter, such as those in the `java.security` package, which add to the security of Java apps. There is also a measure of security built into the Java language itself, but that is outside the scope of this chapter.

What about Just-In-Time compilers?

It is appropriate to include a brief discussion of Just-In-Time (JIT) compilers in this chapter. JIT compilers translate Java bytecodes into native machine instructions to be directly executed by the CPU. This obviously boosts the performance of Java apps. But if native instructions are executed instead of bytecodes, what happens to the verification process mentioned earlier? Actually, the verification process does not change because the Java verifier still verifies the bytecodes before they are translated.

Chapter

10

Working with the Java Native Interface (JNI)

This chapter explains how to invoke native methods in Java applications using the Java Native Method Interface (JNI). It begins by explaining how the JNI works, then discusses the `native` keyword and how any Java method can become a native method. Finally, it examines the JDK's `javah` tool, which is used to generate C header files for Java classes.

Even though Java code is designed to run on multiple platforms, there are certain situations where it may not be enough by itself. For example,

- The standard Java class library doesn't support platform-dependent features needed by your application.
- You want to access an existing library from another language and make it accessible to your Java code.
- You have code you want to implement in a lower-level program like assembly, then have your Java application call it.

The Java Native Interface is a standard cross-platform programming interface included in the JDK. It enables you to write Java programs that can operate with applications and libraries written in other programming languages, such as C, C++, and assembly.

Using JNI, you can write *Java native methods* to create, inspect, and update Java objects (including arrays and strings), call Java methods, catch and throw exceptions, load classes and obtain class information, and perform runtime type checking.

In addition, you can use the Invocation API to embed the Java Virtual Machine into your native applications, then use the JNI interface pointer to access VM features. This allows you to make existing applications Java-enabled without having to link with the VM source code.

How JNI works

In order to achieve Java's main goal of platform independence, Sun did not standardize its implementation of the Java Virtual Machine; in other words, Sun did not want to rigidly specify the internal architecture of the JVM, but allowed vendors to have their own implementations of the JVM. This does not preclude Java from being platform-independent, because every JVM implementation must still comply with certain standards needed to achieve platform independence (such as the standard structure of a `.class` file).

The only problem with this scenario is that accessing native libraries from Java apps becomes difficult, since the runtime system differs across the various JVM implementations. For that reason, Sun came up with the JNI as a standard way for accessing native libraries from Java applications.

The way native methods are accessed from Java applications changed in the JDK 1.1. The old way allowed a Java class to directly access methods in a native library. The new implementation uses the JNI as an intermediate layer between a Java class and a native library. Instead of having the JVM make direct calls to native methods, the JVM uses a pointer to the JNI to make the actual calls. This way, even if the JVM implementations are different, the layer they use to access the native methods (the JNI) is always the same.

Using the native keyword

Making Java methods native is very easy. Below is a summary of the required steps:

1. Delete the body of the method.
2. Add a semicolon at the end of the method's signature.
3. Add the `native` keyword at the beginning of the method's signature.
4. Include the method's body in a native library to be loaded at runtime.

For example, assume the following method exists in a Java class:

```
public void nativeMethod () {
  //the method's body
}
```

This is how the method becomes native:

```
public native void nativeMethod ();
```

Now that you've declared the method to be `native`, its actual implementation will be included in a native library. It is the duty of the class, of which this

method is a member, to invoke the library so its implementation becomes globally available. The easiest way to have the class invoke the library is to add the following to the class:

```
static
{
  System.loadLibrary (nameOfLibrary);
}
```

A `static` code block is always executed once when the class is first loaded. You can include virtually anything in a `static` code block. However, loading libraries is the most common use for it. If, for some reason, the library fails to load, an `UnsatisfiedLineError` exception will be thrown once a method from that library is called. The JVM will add the correct extension to its name (`.dll` in Windows, and `.so` in UNIX)—you don't have to specify it in the library name.

Using the javah tool

The JDK supplies a tool called `javah`, which is used to generate C header files for Java classes. The following is the general syntax for using `javah`:

```
javah [options] className
```

`className` represents the name of the class (without the `.class` extension) for which you want to generate a C header file. You can specify more than one class at the command line. For each class, `javah` adds a `.h` file to the class's directory by default. To put the `.h` files in a different directory, use the `-o` option. If a class is in a package, you must specify the package along with the class name.

For example, to generate a header file for the class `myClass` in the package `myPackage`, do the following:

```
javah myPackage.myClass
```

The generated header file will include the package name, (`myPackage_myClass.h`).

Below is a list of some of the `javah` options:

Option	Description
`-jni`	Creates a JNI header file
`-verbose`	Displays progress information
`-version`	Displays the version of `javah`
`-o directoryName`	Outputs the `.h` file in specified directory
`-classpath path`	Overrides the default class path

The contents of the `.h` file generated by `javah` include all the function prototypes for the native methods in the class. The prototypes are modified to allow the Java runtime system to find and invoke the native methods. This modification basically involves changing the name of the method according to a naming convention established for native method invocation. The modified

name includes the prefix `Java_` to the class and method names. So, if you have a native method called `nativeMethod` in a class called `myClass`, the name that appears in the `myClass.h` file is `Java_myClass_nativeMethod`.

For more information on JNI, see the following:

- Java Native Interface at `http://java.sun.com/j2se/1.3/docs/guide/jni/`
- Java Native Interface Specification at `http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html`
- *The Java Tutorial*, "Trail: Java Native Interface" at `http://java.sun.com/docs/books/tutorial/native1.1/index.html`

The following books on Java Native Interface are also available:

- "The Java Native Interface: Programmer's Guide and Specification (Java Series)" by Sheng Liang

  amazon.com `http://www.amazon.com/exec/obidos/ASIN/0201325772/inprisecorporati/103-7218087-8544625`

  fatbrain.com `http://btob.barnesandnoble.com/home.asp?userid=2UT912FFK4&btob=Y`

- "Essential Jni: Java Native Interface (Essential Java)", by Rob Gordon

  amazon.com `http://www.amazon.com/exec/obidos/ASIN/0136798950/inprisecorporati/103-7218087-8544625`

  fatbrain.com `http://btob.barnesandnoble.com/home.asp?userid=2UT912FFK4&btob=Y`

Appendix

A

Java language quick reference

Java 2 platform editions

The Java 2 Platform is available in several editions used for various purposes. Because Java is a language that can run anywhere and on any platform, it is used in a variety of environments and has been packaged in several editions: Java 2 Standard Edition (J2SE), Java 2 Enterprise Edition (J2EE), and Java 2 Micro Edition (J2ME). In some cases, as in the development of enterprise applications, a larger set of packages is used. In other cases, as in consumer electronic products, only a small portion of the language is used. Each edition contains a Java 2 Software Development Kit (SDK) used to develop applications and a Java 2 Runtime Environment (JRE) used to run applications.

Table A.1 Java 2 Platform editions

Java 2 Platform	Abbreviation	Description
Standard Edition	J2SE	Contains classes that are the core of the Java language.
Enterprise Edition	J2EE	Contains J2SE classes and additional classes for developing enterprise applications.
Micro Edition	J2ME	Contains a subset of J2SE classes and is used in consumer electronic products.

Java class libraries

Java, like most programming languages, relies heavily on pre-built libraries to support certain functionality. In the Java language, these groups of related classes called packages vary by Java edition. Each edition is used for specific purposes, such as applications, enterprise applications, and consumer products.

The Java 2 Platform, Standard Edition (J2SE) provides developers with a feature-rich, stable, secure, cross-platform development environment. This Java edition supports such core features as database connectivity, user interface design, input/output, and network programming and includes the fundamental packages of the Java language. Some of these J2SE packages are listed in the following table.

Table A.2 J2SE packages

Package	Package Name	Description
Language	`java.lang`	Classes that contain the main core of the Java language.
Utilities	`java.util`	Support for utility data structures.
I/O	`java.io`	Support for various types of input/output.
Text	`java.text`	Localization support for handling text, dates, numbers, and messages.
Math	`java.math`	Classes for performing arbitrary-precision integer and floating-point arithmetic.
AWT	`java.awt`	User interface design and event-handling.
Swing	`javax.swing`	Classes for creating all-Java, lightweight components that behave similarly on all platforms.
Javax	`javax`	Extensions to the Java language.
Applet	`java.applet`	Classes for creating applets.
Beans	`java.beans`	Classes for developing JavaBeans.
Reflection	`java.lang.reflect`	Classes used to obtain runtime class information.
SQL	`java.sql`	Support for accessing and processing data in databases.
RMI	`java.rmi`	Support for distributed programming.
Networking	`java.net`	Classes that support development of networking applications.
Security	`java.security`	Support for cryptographic security.

Java keywords

These tables cover the following types of keywords:

Data and return types and terms

Keyword	Use	Keyword	Use
`boolean`	Boolean values.	`char`	16 bits, one character.
`byte`	one byte, integer.	`float`	4 bytes, single-precision.
`short`	2 bytes, integer.	`double`	8 bytes, double-precision.
`long`	8 bytes, integer.	`int`	4 bytes, integer.
`strictfp`	(proposed) Method or class to use standard precision in floating-point intermediate calculations.		
`void`	Return type where no return value is required.	`return`	Exit the current code block with any resulting values.

Packages, classes, members, and interfaces

Keyword	Use	Keyword	Use
`package`	Declares a package name for all classes defined in source files with the same package declaration.	`import`	Makes all classes in the imported class or package visible to the current program.
`class`	Declares a Java class.	`new`	Instantiates a class.
`super`	Inside a subclass, refers to the superclass.	`instanceof`	Checks an object's inheritance.
`final`	This class can't be extended.	`abstract`	This method or class must be extended to be used.

Keyword	Use	Keyword	Use
`extends`	Creates a subclass. Gives a class access to public and protected members of another class. Allows one interface to inherit another.	`implements`	In a class definition, implements a defined interface.
`interface`	Abstracts a class's interface from its implementation (tells what to do, not how to do it).	`synchronized`	Makes a code block thread-safe.
`native`	The body of this method is provided by a link to a native library.	`this`	Refers to the current object.
`static`	Member is available to the whole class, not just one object.	`transient`	This variable's value won't persist when the object is stored.
`volatile`	This variable's value can change unexpectedly.		

Access modifiers

Keyword	Use	Keyword	Use
`public`	Class: accessible from anywhere. Subclass: accessible as long as its class is accessible.	`protected`	Access limited to member's class's package.
`private`	Access limited to member's own class.	`package`	Default access level; don't use it explicitly. Cannot be subclassed by another package.

Loops and flow controls

Keyword	Use	Keyword	Use
`if`	Selection statement.	`else`	Selection statement.
`switch`	Selection statement.	`case`	Selection statement.
`break`	Breakout statement.	`default`	Fallback statement.
`for`	Iteration statement.	`do`	Iteration statement.
`while`	Iteration statement.	`continue`	Iteration statement.
`assert`	Checks a condition before allowing a statement to be executed.		

Exception handling

Keyword	Use	Keyword	Use
`throws`	Lists the exceptions a method could throw.	`throw`	Transfers control of the method to the exception handler.
`try`	Opening exception-handling statement.	`catch`	Captures the exception.
`finally`	Runs its code before terminating the program.		

Reserved

Keyword	Use	Keyword	Use
`goto`	Reserved for future use.	`const`	Reserved for future use.

Converting and casting data types

An object or variable's data type can be altered for a single operation when a different type is required. Widening conversions (from a smaller class or data type to a larger) can be implicit, but it's good practice to convert explicitly. Narrowing conversions must be explicitly converted, or *cast*. Novice programmers should avoid casts; they can be a rich source of errors and confusion.

For narrowing casts, put the type you want to cast *to* in parentheses immediately before the variable you want to cast: `(int)x`. This is what it looks like in context, where `x` is the variable being cast, `float` is the original data type, `int` is the target data type, and `y` is the variable storing the new value:

```
float x = 1.00; //declaring x as a float
int y = (int)x; //casting x to an int named y
```

This assumes that the value of `x` would fit inside of `int`. Note that `x`'s decimal values are lost in the conversion. Java rounds decimals down to the nearest whole number.

Note that Unicode sequences can represent numbers, letters, symbols, or nonprinting characters such as line breaks or tabs. For more information on Unicode, see `http://www.unicode.org/`

This section contains tables of the following conversions:

- Primitive to primitive
- Primitive to String
- Primitive to reference
- String to primitive
- Reference to primitive
- Reference to reference

Primitive to primitive

Java doesn't support casting to or from `boolean` values. In order to work around Java's strict logical typing, you must assign an appropriate equivalent value to the variable and then convert that. 0 and 1 are often used to represent `false` and `true` values.

Syntax	Comments
From other primitive type `p` **To** `boolean t`: `t = p != 0;`	Other primitive types include `byte`, `short`, `char`, `int`, `long`, `double`, `float`.
From `boolean t` **To** `byte b`: `b = (byte)(t ? 1 : 0);`	
From `boolean t` **To** `int, long, double`, or `float m`: `m = t ? 1 : 0;`	
From `boolean t` **To** `short s`: `s = (short) (t ? 1 : 0);`	
From `boolean t` **To** `byte b`: `b = (byte) (t?1:0);`	
From `boolean t` **To** `char c`: `c = (char) (t?'1':'0');`	
From `short, char, int, long, double`, or `float n` **To** `byte b`: `b = (byte)n;`	
From `byte b` **To** `short, int, long, double`, or `float n`: `n = b;`	
From `byte b` **To** `char c`: `c = (char)b;`	

Primitive to String

Primitive data types are mutable; reference types are immutable objects. Casting to or from a reference type is risky.

Java doesn't support casting to or from `boolean` values. In order to work around Java's strict logical typing, you must assign an appropriate equivalent value to the variable and then convert that. 0 and 1 are often used to represent `false` and `true` values.

Syntax	Comments
From `boolean t` **To** `String gg`: `gg = t ? "true" : "false";`	

From `byte b`
To `String gg`:
`gg = Integer.toString(b);`
or
`gg = String.valueOf(b);`
	The following may be substituted for `toString`, where appropriate:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Where you are using a base other than 10 or 2 (such as 8):
	`gg = Integer.toString(b, 7);`

From `short` or `int n`
To `String gg`:
`gg = Integer.toString(n);`
or
`gg = String.valueOf(n);`
	The following may be substituted for `toString`, where appropriate:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Where you are using a base other than 10 (such as 8):
	`gg = Integer.toString(n, 7);`

From `char c` **To** `String gg`: `gg = String.valueOf(c);`	

From `long n`
To `String gg`:
`gg = Long.toString(n);`
or
`gg = String.valueOf(n);`
	The following may be substituted for `toString`, where appropriate:
	`toBinaryString`
	`toOctalString`
	`toHexString`
	Where you are using a base other than 10 or 2 (such as 8):
	`gg = Integer.toString(n, 7);`

Syntax	Comments

From `float f`
To `String gg`:
`gg = Float.toString(f);`
or
`gg = String.valueOf(f);`
For decimal protection or scientific notation, see next column.
	These casts protect more data.
	Double precision:
	`java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00");`
	`gg = df2.format(f);`
	Scientific notation (protects exponents) (JDK 1.2.x and up):
	`java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00");`
	`gg = de.format(f);`

From `double d`
To `String gg`:
`gg = Double.toString(d);`
or
`gg = String.valueOf(d);`
For decimal protection or scientific notation, see next column.
	These casts protect more data.
	Double precision:
	`java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00");`
	`gg = df2.format(d);`
	Scientific notation (JDK 1.2.x and up):
	`java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00");`
	`gg = de.format(d);`

Primitive to reference

Java provides classes that correspond to primitive data types and provide methods that facilitate conversions.

Note that primitive data types are mutable; reference types are immutable objects. Casting to or from a reference type is risky.

Java doesn't support casting to or from `boolean` values. In order to work around Java's strict logical typing, you must assign an appropriate equivalent value to the variable and then convert that. 0 and 1 are often used to represent `false` and `true` values.

Syntax	Comments
From `boolean t` **To** `Boolean tt`: `tt = new Boolean(t);`	
From Primitive type `p` (other than `boolean`) **To** `Boolean tt` `tt = new Boolean(p != 0);` For `char`, see next column.	For `char c`, put single quotes around the zero: `tt = new Boolean(c != '0');`
From `boolean t` **To** `Character cc`: `cc = new Character(t ? '1' : '0');`	

Syntax	Comments
From byte b **To** Character cc: `cc = new Character((char) b);`	
From char c **To** Character cc: `cc = new Character(c);`	
From short, int, long, float, or double n **To** Character cc: `cc = new Character((char)n);`	
From boolean t **To** Integer ii: `ii = new Integer(t ? 1 : 0);`	
From byte b **To** Integer ii: `ii = new Integer(b);`	
From short, char, or int n **To** Integer ii: `ii = new Integer(n);`	
From long, float, or double f **To** Integer ii: `ii = new Integer((int) f);`	
From boolean t **To** Long nn: `nn = new Long(t ? 1 : 0);`	
From byte b **To** Long nn: `nn = new Long(b);`	
From short, char, int, or long s **To** Long nn: `nn = new Long(s);`	
From float, double f **To** Long nn: `nn = new Long((long)f);`	
From boolean t **To** Float ff: `ff = new Float(t ? 1 : 0);`	
From byte b **To** Float ff: `ff = new Float(b);`	

Syntax	Comments
From `short, char, int, long, float, or double n` **To** `Float ff`: `ff = new Float(n);`	
From `boolean t` **To** `Double dd`: `dd = new Double(t ? 1 : 0);`	
From `byte b` **To** `Double dd`: `dd = new Double(b);`	
From `short, char, int, long, float, or double n` **To** `Double dd`: `dd = new Double(n);`	

String to primitive

Note that primitive data types are mutable; reference types are immutable objects. Casting to or from a reference type is risky.

Java doesn't support casting to or from `boolean` values. In order to work around Java's strict logical typing, you must assign an appropriate equivalent value to the variable and then convert that. The numbers 0 and 1, the strings "true" and "false", or equally intuitive values are used here to represent `true` and `false` values.

Syntax	Comments
From `String gg` **To** `boolean t`: `t = new Boolean(gg.trim()).booleanValue();`	**Caution:** `t` will only be `true` when the value of `gg` is "`true`" (case insensitive); if the string is "1", "yes", or any other affirmative, this conversion will return a `false` value.
From `String gg` **To** `byte b`: `try { b = (byte)Integer.parseInt(gg.trim()); } catch (NumberFormatException e) { ... }`	**Note:** If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space. For bases other than 10, such as 8: `try { b = (byte)Integer.parseInt(gg.trim(), 7); } catch (NumberFormatException e) { ... }`

Syntax	Comments

From `String gg`
To `short s`:
`try {`
`  s = (short)Integer.parseInt(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	For bases other than 10, such as 8:
	`try {`
	`  s = (short)Integer.parseInt(gg.trim(),`
	`  7);`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `char c`:
`try {`
`  c = (char)Integer.parseInt(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	For bases other than 10, such as 8:
	`try {`
	`  c = (char)Integer.parseInt(gg.trim(),`
	`  7);`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `int i`
`try {`
`  i = Integer.parseInt(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	For bases other than 10, such as 8:
	`try {`
	`  i = Integer.parseInt(gg.trim(), 7);`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `long n`:
`try {`
`  n = Long.parseLong(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.

Syntax	Comments

From `String gg`
To `float f`:
`try {`
`f = Float.valueOf(gg.trim()).floatValue;`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	For JDK 1.2.x or better:
	`try {`
	`  f = Float.parseFloat(gg.trim());`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `double d`:
`try {`
`d = Double.valueOf(gg.trim()).doubleValue;`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	For JDK 1.2.x or better:
	`try {`
	`  d = Double.parseDouble(gg.trim());`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

Reference to primitive

Java provides classes that correspond to the primitive data types. This table shows how to convert a variable from one of these classes to a primitive data type for a single operation.

To convert from a reference type to a primitive, you must first get the value of the reference as a primitive, then cast the primitive.

Primitive data types are mutable; reference types are immutable objects. Converting to or from a reference type is risky.

Java doesn't support casting to or from `boolean` values. In order to work around Java's strict logical typing, you must assign an appropriate equivalent value to the variable and then convert that. 0 and 1 are often used to represent `false` and `true` values.

Syntax	Comments
From `Boolean tt` **To** `boolean t`: `t = tt.booleanValue();`	
From `Boolean tt` **To** `byte b`: `b = (byte)(tt.booleanValue() ? 1 : 0);`	

Syntax	Comments
From Boolean tt **To** short s: `s = (short)(tt.booleanValue() ? 1 : 0);`	
From Boolean tt **To** char c: `c = (char)(tt.booleanValue() ? '1' : '0');`	
From Boolean tt **To** int, long, float, or double n: `n = tt.booleanValue() ? 1 : 0);`	
From Character cc **To** boolean t: `t = cc.charValue() != 0;`	
From Character cc **To** byte b: `b = (byte)cc.charValue();`	
From Character cc **To** short s: `s = (short)cc.charValue();`	
From Character cc **To** char, int, long, float, or double n: `n = cc.charValue();`	
From Integer ii **To** boolean t: `t = ii.intValue() != 0;`	
From Integer ii **To** byte b: `b = ii.byteValue();`	
From Integer, Long, Float, or Double nn **To** short s: `s = nn.shortValue();`	
From Integer, Long, Float, or Double nn **To** char c: `c = (char)nn.intValue();`	
From Integer, Long, Float, or Double nn **To** int i: `i = nn.intValue();`	
From Integer ii **To** long n: `n = ii.longValue();`	

Syntax	Comments
From `Long, Float,` or `Double dd` **To** `long n`: `n = dd.longValue();`	
From `Integer, Long, Float,` or `Double nn` **To** `float f`: `f = nn.floatValue();`	
From `Integer, Long, Float,` or `Double nn` **To** `double d`: `d = nn.doubleValue();`	

Reference to reference

Java provides classes that correspond to the primitive data types. This table shows how to convert a variable from one of these classes to another for a single operation.

Note For legal class to class conversions apart from what's shown here, widening conversions are implicit. Narrowing casts use this syntax:

```
castToObjectName = (CastToObjectClass)castFromObjectName;
```

You must cast between classes that are in the same inheritance hierarchy. If you cast an object to an incompatible class, it will throw a `ClassCastException`.

Reference types are immutable objects. Converting between reference types is risky.

Syntax	Comments
From `String gg` **To** `Boolean tt`: `tt = new Boolean(gg.trim());`	**Note:** If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space. Alternative: `tt = Boolean.valueOf(gg.trim());`
From `String gg` **To** `Character cc`: `cc = new Character(gg.charAt(<index>);`	

Syntax	Comments

From `String gg`
To `Integer ii`:
`try {`
`  ii = new Integer(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	Alternative:
	`try {`
	`  ii = Integer.valueOf(gg.trim());`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `Long nn`:
`try {`
`  nn = new Long(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	Alternative:
	`try {`
	`  nn = Long.valueOf(gg.trim());`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

From `String gg`
To `Float ff`:
`try {`
`  ff = new Float(gg.trim());`
`}`
`catch (NumberFormatException e) {`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	Alternative:
	`try {`
	`  ff = Float.valueOf(gg.trim());`
	`}`
	`catch`
	`  ...`
	`}`

From `String gg`
To `Double dd`:
`try {`
`  dd = new Double(gg.trim());`
`}`
`catch`
`  ...`
`}`
	Note: If the value of `gg` is null, `trim()` will throw a `NullPointerException`. If you don't use `trim()`, make sure there's no trailing white space.
	Alternative:
	`try {`
	`  dd = Double.valueOf(gg.trim());`
	`}`
	`catch (NumberFormatException e) {`
	`  ...`
	`}`

Syntax	Comments
From Boolean tt **To** Character cc: `cc = new Character(tt.booleanValue() ?'1':'0');`	
From Boolean tt **To** Integer ii: `ii = new Integer(tt.booleanValue() ? 1 : 0);`	
From Boolean tt **To** Long nn: `nn = new Long(tt.booleanValue() ? 1 : 0);`	
From Boolean tt **To** Float ff: `ff = new Float(tt.booleanValue() ? 1 : 0);`	
From Boolean tt **To** Double dd: `dd = new Double(tt.booleanValue() ? 1 : 0);`	
From Character cc **To** Boolean tt: `tt = new Boolean(cc.charValue() != '0');`	
From Character cc **To** Integer ii: `ii = new Integer(cc.charValue());`	
From Character cc **To** Long nn: `nn = new Long(cc.charValue());`	
From any class rr **To** String gg: `gg = rr.toString();`	

From Float ff
To String gg:
`gg = ff.toString();`
	These variations protect more data.
	Double precision:
	`java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00"); gg = df2.format(ff.floatValue());`
	Scientific notation (JDK 1.2.x on up):
	`java.text.DecimalFormat de = new java.text.DecimalFormat("0.000000E00"); gg = de.format(ff.floatValue());`

Syntax	Comments

From Double dd
To String gg:
`gg = dd.toString();`
	These variations protect more data.
	Double precision:
	`java.text.DecimalFormat df2 = new java.text.DecimalFormat("###,##0.00");`
	`gg = df2.format(dd.doubleValue());`
	Scientific notation (JDK 1.2.x on up):
	`java.text.DecimalFormat de = new java.text.DecimalFormat("0.0000000000E00");`
	`gg = de.format(dd.doubleValue());`

From Integer ii **To** Boolean tt: `tt = new Boolean(ii.intValue() != 0);`	
From Integer ii **To** Character cc: `cc = new Character((char)ii.intValue());`	
From Integer ii **To** Long nn: `nn = new Long(ii.intValue());`	
From Integer ii **To** Float ff: `ff = new Float(ii.intValue());`	
From Integer ii **To** Double dd: `dd = new Double(ii.intValue());`	
From Long nn **To** Boolean tt: `tt = new Boolean(nn.longValue() != 0);`	
From Long nn **To** Character cc: `cc = new Character((char)nn.intValue());`	**Note:** Some Unicode values may be rendered as nonprintable characters. Consult http://www.unicode.org/
From Long nn **To** Integer ii: `ii = new Integer(nn.intValue());`	
From Long nn **To** Float ff: `ff = new Float(nn.longValue());`	
From Long nn **To** Double dd: `dd = new Double(nn.longValue());`	

Syntax	Comments
From Float ff **To** Boolean tt: `tt = new Boolean(ff.floatValue() != 0);`	
From Float ff **To** Character cc: `cc = new Character((char)ff.intValue());`	**Note:** Some Unicode values may be rendered as nonprintable characters. Consult http://www.unicode.org/
From Float ff **To** Integer ii: `ii = new Integer(ff.intValue());`	
From Float ff **To** Long nn: `nn = new Long(ff.longValue());`	
From Float ff **To** Double dd: `dd = new Double(ff.floatValue());`	
From Double dd **To** Boolean tt: `tt = new Boolean(dd.doubleValue() != 0);`	
From Double dd **To** Character cc: `cc = new Character((char)dd.intValue());`	**Note:** Some Unicode values may be rendered as nonprintable characters. Consult http://www.unicode.org/
From Double dd **To** Integer ii: `ii = new Integer(dd.intValue());`	
From Double dd **To** Long nn: `nn = new Long(dd.longValue());`	
From Double dd **To** Float ff: `ff = new Float(dd.floatValue());`	

Escape sequences

An octal character is represented by a sequence of three octal digits, and a Unicode character is represented by a sequence of four hexadecimal digits. Octal characters are preceded by the standard escape mark, `\`, and Unicode characters are preceded by `\u`. For example, the decimal number 57 is represented by the octal code `\071` and the Unicode sequence `\u0039`. Octal code accepts 0, 1, 2, or 3 in the left-most position, and any number from 0–7 in the other two positions. Unicode sequences can represent numbers, letters,

symbols, or nonprinting characters such as line breaks or tabs. For more information on Unicode, see http://www.unicode.org/

Character	Escape Sequence
Backslash	\\
Backspace	\b
Carriage return	\r
Double quote	\"
Form feed	\f
Horizontal tab	\t
New line	\n
Octal character	\DDD
Single quote	\'
Unicode character	\uHHHH

Operators

This section lists the following:

- Basic operators
- Arithmetic operators
- Logical operators
- Assignment operators
- Comparison operators
- Bitwise operators
- Ternary operator

The order in which operations in a compound statement are evaluated depends on associativity (left/right, parentheses) and precedence (hierarchy.) The rules of precedence are complicated. For a quick summary, see http://java.sun.com/docs/books/tutorial/java/nutsandbolts/expressions.html.

Basic operators

Operator	Operand	Behavior
.	object member	Accesses a member of the object.
(<type>)	data type	Casts a variable to a different data type.[1]
+	String	Joins up strings (concatenator).
	number	Adds.
-	number	This is the unary[2] minus (reverses number sign).
	number	Subtracts.
!	boolean	This is the boolean NOT operator.

Operator	Operand	Behavior
`&`	integer, boolean	This is both the bitwise (integer) and `boolean` AND operator. When doubled (`&&`), it is the `boolean` conditional AND.
`=`	most elements with variables	Assigns an element to another element (for instance, a value to a variable, or a class to an instance). This can be combined with other operators to perform the other operation and assign the resulting value. For instance, `+=` adds the left-hand value to the right, then assigns the new value to the right-hand side of the expression.

1. It's important to distinguish between an operator and a delimiter. Parentheses are used around args (for instance) as delimiters that mark the args in the statement. They are used around a data type as operators that change a variable's data type to the one inside the parentheses.

2. A *unary operator* affects a single operand, a *binary operator* affects two operands, and a *ternary operator* affects three operands.

Arithmetic operators

Operator	Assoc.	Definition
`++/--`	Right	Auto-increment/decrement: Adds one to, or subtracts one from, its single operand. Can be used before or after operand, depending on when you want the original value changed.
`+/-`	Right	Unary plus/minus: sets or changes the positive/ negative value of a single number.
`*`	Left	Multiplication.
`/`	Left	Division.
`%`	Left	Modulus: Divides the first operand by the second operand and returns the remainder (not the result).
`+/-`	Left	Addition/subtraction

Logical operators

Operator	Assoc.	Definition
`!`	Right	Boolean `NOT` (unary) Changes `true` to `false` or `false` to `true`. Because of its low precedence, you may need to use parentheses around this statement.
`&`	Left	Evaluation `AND` (binary) Yields `true` only if both operands are `true`. Always evaluates both operands.

Operator	Assoc.	Definition
^	Left	Evaluation XOR (binary) Yields true if only one operand is true. Evaluates both operands.
|	Left	Evaluation OR (binary) Yields true if one or both of the operands is true. Evaluates both operands.
&&	Left	Conditional AND (binary) Yields true only if both operands are true. Called "conditional" because it only evaluates the second operand if the first operand is true.
||	Left	Conditional OR (binary) Yields true if either one or both operands is true; returns false if both are false. Doesn't evaluate second operand if first operand is true.

Assignment operators

Operator	Assoc.	Definition
=	Right	Assign the value on the right to the variable on the left.
+=	Right	Add the value on the right to the value of the variable on the left; assign the new value to the original variable.
-=	Right	Subtract the value on the right from the value of the variable on the left; assign the new value to the original variable.
*=	Right	Multiply the value on the right with the value of the variable on the left; assign the new value to the original variable.
/=	Right	Divide the value on the right from the value of the variable on the left; assign the new value to the original variable.

Comparison operators

Operator	Assoc.	Definition
<	Left	Less than
>	Left	Greater than
<=	Left	Less than or equal to
>=	Left	Greater than or equal to
==	Left	Equal to
!=	Left	Not equal to

Bitwise operators

Note A signed integer is one whose left-most bit is used to indicate the integer's positive or negative sign: the bit is 1 if the integer is negative, 0 if positive. In

Java, integers are always signed, whereas in C/C++ they are signed only by default. In Java, the shift operators preserve the sign bit, so that the sign bit is duplicated, then shifted. For example, right shifting 10010011 by 1 is 11001001.

Operator	Assoc.	Definition
~	Right	Bitwise NOT Inverts each bit of the operand, so each 0 becomes 1 and vice versa.
<<	Left	Signed left shift Shifts the bits of the left operand to the left, by the number of digits specified in the right operand, with 0's shifted in from the right. High-order bits are lost.
>>	Left	Signed right shift Shifts the bits of the left operand to the right, by the number of digits specified in the right operand. If the left operand is negative, 0's are shifted in from the left; if it is positive, 1's are shifted in. This preserves the original sign.
>>>	Left	Zero-fill right shift Shifts right, but always fills in with 0's.
&	Left	Bitwise AND Can be used with = to assign the value.
|	Left	Bitwise OR Can be used with = to assign the value.
^	Left	Bitwise XOR Can be used with = to assign the value.
<<=	Left	Left-shift with assignment
>>=	Left	Right-shift with assignment
>>>=	Left	Zero-fill right shift with assignment

Ternary operator

The ternary operator ?: performs a very simple if-then-else operation inside of a single statement. For example:

```
<expression 1, a Boolean condition> ? <expression 2> : <expression 3>;
```

The Boolean condition, expression 1, is evaluated first. If it resolves true or if its resolution depends on the rest of the ternary statement, then expression 2 is evaluated. If expression 2 is false, expression 3 is used. For example:

```
int x = 3, y = 4, max;
max = (x > y) ? x : y;
```

Here, max is assigned the value of x or y, depending on which is greater. The frst expression evaluates which is greater. The value of x (expression 2) is not greater than the value of y (expression 3), so the value of y is assigned to max.

Appendix

B

Learning more about Java

Resources on the Java language abound. The http://java.sun.com web site of Sun Microsystems is a good place to search for more information; you'll find many interesting links. If you're new to Java, you'll especially want to see Sun's online Java tutorial at http://java.sun.com/docs/books/tutorial/.

Online glossaries

To quickly find definitions of Java terms, see one of Sun's online glossaries:

- Sun Microsystem's *Java Glossary* in HTML: http://java.sun.com/docs/glossary.nonjava.html#top
- Sun Microsystem's *Java Glossary* in Java: http://java.sun.com/docs/glossary.html

Books

There are many excellent books about programming with Java. To see a list of Java titles on the Borland Developer Network, go to http://bdn.borland.com/books/java/0,1427,c|3,00.html.

Sun also publishes a set of books called the Java Series. See the list of titles at http://java.sun.com/docs/books/. You'll find books for all levels of Java programming.

Besides browsing through Java books at your favorite book store, you might also simply type `Java books` into your favorite web search engine to find lists of books that Java programmers recommend or that favorite publishers are offering.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X