Chapter 1

A Crash Course in Java

CHAPTER TOPICS

- "Hello, World!" in Java
- Documentation Comments
- Primitive Types
- Control Flow Statements
- Object References
- Parameter Passing
- Packages
- Basic Exception Handling
- Strings
- Reading Input
- Array Lists and Linked Lists
- Arrays
- Static Fields and Methods
- Programming Style

The purpose of this chapter is to teach you the elements of the Java programming language—or to give you an opportunity to review them—assuming that you know an object-oriented programming language. In

particular, you should be familiar with the concepts of classes and objects. If you know C++ and understand classes, member functions, and constructors, then you will find that it is easy to make the switch to Java.

1.1 "Hello, World!" in Java

Classes are the building blocks of Java programs. Let's start our crash course by looking at a simple but typical class:

```
public class Greeter
{
   public Greeter(String aName)
   {
      name = aName;
   }

   public String sayHello()
   {
      return "Hello, " + name + "!";
   }

   private String name;
}
```

This class has three features:

- A *constructor* `Greeter(String aName)` that is used to construct new objects of this class.
- A *method* `sayHello()` that you can apply to objects of this class. (Java uses the term "method" for a function defined in a class.)
- A *field* `name`. Every object of this class has an instance of this field.

A class definition contains the implementation of constructors, methods, and fields.

Each feature is tagged as `public` or `private`. Implementation details (such as the `name` field) are private. Features that are intended for the class user (such as the constructor and `sayHello` method) are public. The class itself is declared as public as well. You will see the reason in the section on packages.

To construct an object, you use the `new` operator, which invokes the constructor.

```
new Greeter("World")
```

The `new` operator returns the constructed object, or, more precisely, a reference to that object—we will discuss this distinction in detail in the section on object references.

The `new` operator constructs new instances of a class.

The object that the `new` operator returns belongs to the `Greeter` class. In object-oriented parlance, we say that it is an *instance* of the `Greeter` class. The process of constructing an object of a class is often called "instantiating the class".

After you obtain an instance of a class, you can call (or *invoke*) methods on it. The call

```
(new Greeter("World")).sayHello()
```

creates a new object and causes the `sayHello` method to be executed. The result is the string `"Hello, World!"`, the concatenation of the strings `"Hello, "`, `name`, and `"!"`.

Object-oriented programming follows the "client-server" model. The client code requests a service by invoking a method on an object.

The code that invokes a method is often called the *client code*. We think of the object as providing a service for the client.

You often need *variables* to store object references that are the result of the `new` operator or a method call:

```
Greeter worldGreeter = new Greeter("World");
String greeting = worldGreeter.sayHello();
```

Now that you have seen how to define a class, you're ready to build your first Java program, the traditional program that displays the words "Hello, World!" on the screen.

You will define a second class, `GreeterTester`, to produce the output.

Ch1/helloworld/GreeterTester.java

```
1 public class GreeterTester
2 {
3    public static void main(String[] args)
4    {
5       Greeter worldGreeter = new Greeter("World");
6       String greeting = worldGreeter.sayHello();
7       System.out.println(greeting);
8    }
9 }
```

Execution of a Java program starts with the `main` method of a class.

This class has a `main` method, which is required to start a Java application. The `main` method is *static*, which means that it doesn't operate on an object. (We will discuss static methods—also called *class methods*—in greater detail later in this chapter.) When the application is launched, there aren't any objects yet. It is the job of the `main` method to construct the objects that are needed to start the program.

The `args` parameter of the `main` method holds the *command-line arguments*, which are not used in this example. We will discuss command-line arguments in the section on arrays.

You have already seen the first two statements inside the `main` method. They construct a `Greeter` object, store it in an object variable, invoke the `sayHello` method, and capture the result in a string variable. The last statement invokes the `println` method on the `System.out` object. The result is to print the message and a line terminator to the standard output stream.

To build and execute the program, put the `Greeter` class inside a file `Greeter.java` and the `GreeterTester` class inside a separate file `GreeterTester.java`. The directions for compiling and running the program depend on your development environment.

The Java Software Development Kit (SDK) from Sun Microsystems is a set of command-line programs for compiling, running, and documenting Java programs.

Versions for several platforms are available at `http://java.sun.com/j2se`. If you use the Java SDK, then follow these instructions:

1. Create a new directory of your choice to hold the program files.
2. Use a text editor of your choice to prepare the files `Greeter.java` and `GreeterTester.java`. Place them inside the directory you just created.
3. Open a shell window.
4. Use the `cd` command to change to the directory you just created.
5. Run the compiler with the command

   ```
   javac GreeterTester.java
   ```

   If the Java compiler is not on the search path, then you need to use the full path (such as `/usr/local/jdk1.5.0/bin/javac` or `c:\jdk1.5.0\bin\javac`) instead of just `javac`. Note that the `Greeter.java` file is automatically compiled as well since the `GreeterTester` class requires the `Greeter` class. If any compilation errors are reported, then make a note of the file and line numbers and fix them.
6. Have a look at the files in the current directory. Verify that the compiler has generated two *class files*, `Greeter.class` and `GreeterTester.class`.
7. Start the Java interpreter with the command

   ```
   java GreeterTester
   ```

Now you will see a message "Hello, World!" in the shell window (see Figure 1).

The structure of this program is typical for a Java application. The program consists of a collection of classes. One class has a `main` method. You run the program by launching the Java interpreter with the name of the class whose `main` method contains the instructions for starting the program activities.

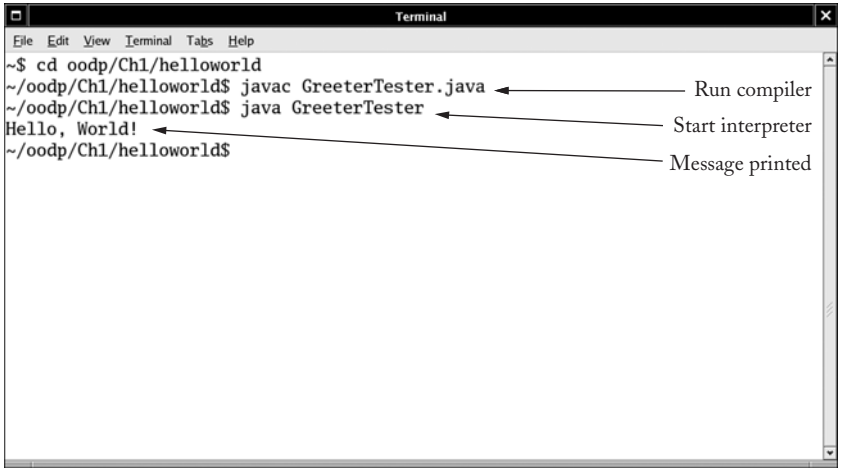


Figure 1

Running the "Hello, World!" Program in a Shell Window

Some programming environments allow you to execute Java code without requiring a `main` method.

The BlueJ development environment, developed at Monash University, lets you test classes without having to write a new program for every test. BlueJ supplies an interactive environment for constructing objects and invoking methods on the objects. You can download BlueJ from `http://www.bluej.org`.

With BlueJ, you don't need a `GreeterTester` class to test the `Greeter` class. Instead, just follow these steps.

1. Select "Project → New…" from the menu; point the file dialog box to a directory of your choice and type in the name of the subdirectory that should hold your classes—this must be the name of a new directory. BlueJ will create it.
2. Click on the "New Class…" button and type in the class name `Greeter`. Right-click on the class rectangle and type in the code of the `Greeter` class.
3. Click on the "Compile" button to compile the class. Click on the "Close" button.
4. The class is symbolized as a rectangle. Right-click on the class rectangle and select "new Greeter(aName)" to construct a new object. Call the object `worldGreeter` and supply the constructor parameter `"World"` (including the quotation marks).
5. The object appears in the object workbench. Right-click on the object rectangle and select "String sayHello( )" to execute the `sayHello` method.
6. A dialog box appears to display the result (see Figure 2).

As you can see, BlueJ lets you think about objects and classes without fussing with `public static void main`.

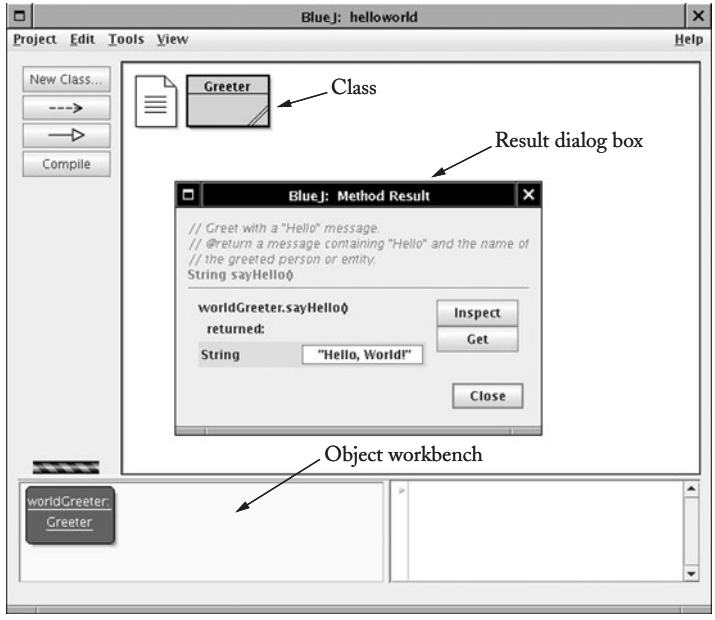


Figure 2

Testing a Class with BlueJ

1.2 Documentation Comments

Java has a standard form for comments that describe classes and their features. The Java development kit contains a tool, called `javadoc`, that automatically generates a convenient set of HTML pages that document your classes.

Documentation comments are delimited by `/**` and `*/`. Both class and method comments start with freeform text. The `javadoc` utility copies the *first sentence* of each comment to a summary table. Therefore, it is best to write that first sentence with some care. It should start with an uppercase letter and end with a period. It does not have to be a grammatically complete sentence, but it should be meaningful when it is pulled out of the comment and displayed in a summary.

Method and constructor comments contain additional information. For each parameter, supply a line that starts with `@param`, followed by the parameter name and a short explanation. Supply a line that starts with `@return` to describe the return value. Omit the `@param` tag for methods that have no parameters, and omit the `@return` tag for methods whose return type is `void`.

Here is the `Greeter` class with documentation comments for the class and its public interface.

ch1/helloworld/Greeter.java

```
1  /**
2     A class for producing simple greetings.
3  */
4  public class Greeter
5  {
6     /**
7        Constructs a Greeter object that can greet a person or entity.
8        @param aName  the name of the person or entity who should
9        be addressed in the greetings.
10    */
11    public Greeter(String aName)
12    {
13       name = aName;
14    }
15
16    /**
17       Greet with a "Hello" message.
18       @return  a message containing "Hello" and the name of
19       the greeted person or entity.
20    */
21    public String sayHello()
22    {
23       return "Hello, " + name + "!";
24    }
25
26    private String name;
27 }
```

Your first reaction may well be "Whoa! I am supposed to write all this stuff?" These comments do seem pretty repetitive. But you should still take the time to write them, even if it feels silly at times. There are three reasons.

First, the `javadoc` utility will format your comments into a nicely formatted set of HTML documents. It makes good use of the seemingly repetitive phrases. The first sentence of each method comment is used for a *summary table* of all methods of your class (see Figure 3). The `@param` and `@return` comments are neatly formatted in the detail descriptions of each method (see Figure 4). If you omit any of the comments, then `javadoc` generates documents that look strangely empty.

Supply comments for all methods and public fields of a class.

Next, it is possible to spend more time pondering whether a comment is too trivial to write than it takes just to write it. In practical programming, very simple methods are rare. It is harmless to have a trivial method overcommented, whereas a complicated method without any comment can cause real grief to future maintenance programmers. According to the standard Java documentation style, *every* class, *every* method, *every* parameter, and *every* return value should have a comment.

Finally, it is always a good idea to write the method comment *first*, before writing the method code. This is an excellent test to see that you firmly understand what you need to program. If you can't explain what a class or method does, you aren't ready to implement it.

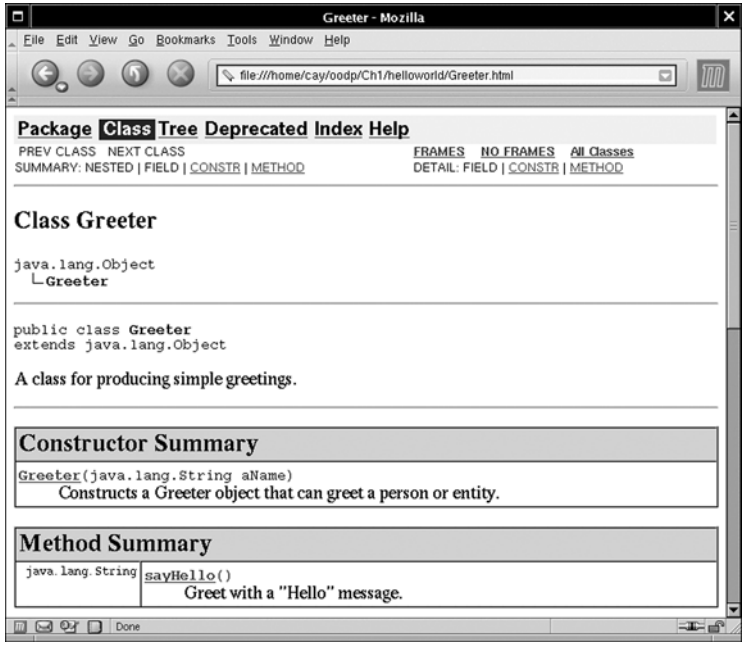


Figure 3

A `javadoc` Class Summary

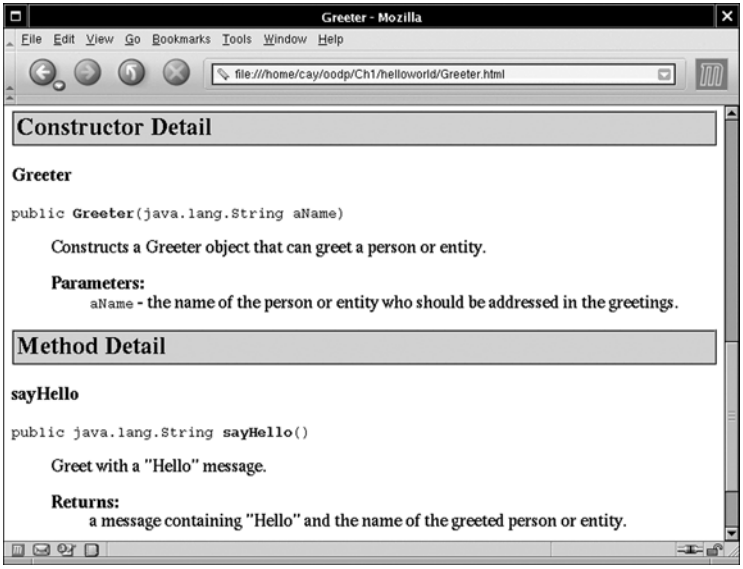


Figure 4

Parameter and Return Value Documentation in `javadoc`

After you have written the documentation comments, invoke the `javadoc` utility.

1. Open a shell window.
2. Use the `cd` command to change to the directory you just created.
3. Run the `javadoc` utility

   ```
   javadoc *.java
   ```

   If the Java development tools are not on the search path, then you need to use the full path (such as `/usr/local/jdk1.5.0/bin/javadoc` or `c:\jdk1.5.0\bin\javadoc`) instead of just `javadoc`.

The `javadoc` utility extracts documentation comments and produces a set of cross-linked HTML files.

The `javadoc` utility then produces one HTML file for each class (such as `Greeter.html` and `GreeterTester.html`) as well as a file `index.html` and a number of other summary files. The `index.html` file contains links to all classes.

The `javadoc` tool is wonderful because it does one thing right: It allows you to put the documentation *together with your code*. That way, when you update your programs, you can see immediately which documentation needs to be updated. Hopefully, you will then update it right then and there. Afterwards, run `javadoc` again and get a set of nicely formatted HTML pages with the updated comments.

INTERNET The `DocCheck` program reports any missing `javadoc` comments. Download it from `http://java.sun.com/j2se/javadoc/doccheck/`.

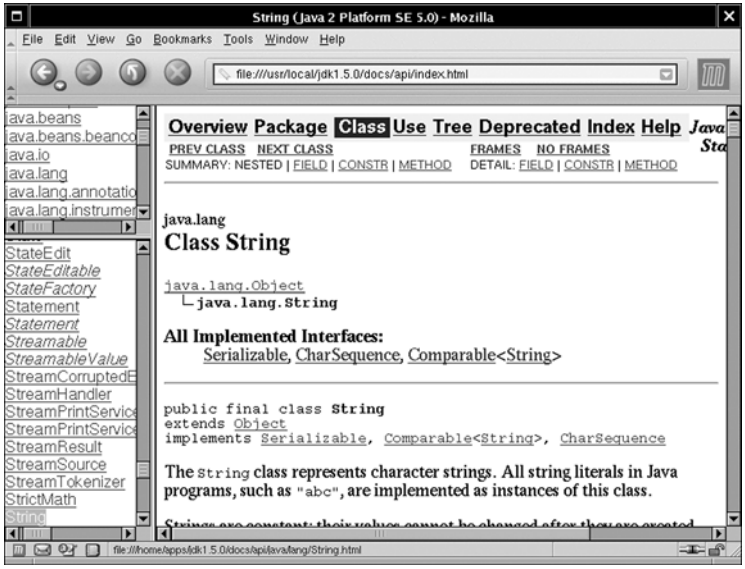


Figure 5

The Java API Documentation

The Java development kit contains the documentation for *all* classes in the Java library, also called the application programming interface or API. Figure 5 shows the documentation of the `String` class. This documentation is directly extracted from the library source code. The programmers who wrote the Java library documented every class and method and then simply ran `javadoc` to extract the HTML documentation.

TIP Download the SDK documentation from `http://java.sun.com/j2se`. Install the documentation into the same location as the Java development kit. Point your browser to the `docs/api/index.html` file inside your Java development kit directory, and make a bookmark. Do it now! You will need to access this information frequently.

1.3 Primitive Types

Java has eight primitive types for integers, floating-point numbers, bytes, characters, and `boolean` values.

In Java, numbers, characters, and Boolean values are not objects but values of a primitive type. Table 1 shows the eight primitive types of the Java language.

To indicate `long` constants, use a suffix `L`, such as `10000000000L`. Similarly, `float` constants have a suffix `F`, such as `3.1415927F`.

Characters are encoded in *Unicode*, a uniform encoding scheme for characters in many languages around the world. Character constants are enclosed in single quotes, such as `'a'`. Several characters, such as a newline `'\n'`, are represented as two-character escape

Type	Size	Range
`int`	4 bytes	–2,147,483,648 . . . 2,147,483,647
`long`	8 bytes	–9,223,372,036,854,775,808L . . . 9,223,372,036,854,775,807L
`short`	2 bytes	–32768 . . . 32767
`byte`	1 byte	–128 . . . 127
`char`	2 bytes	`'\u0000' ... '\uFFFF'`
`boolean`		`false, true`
`double`	8 bytes	approximately ±1.79769313486231570E+308
`float`	4 bytes	approximately ±3.40282347E+38F

Table 1

The Primitive Types of the Java Language

Escape Sequence	Meaning
`\b`	backspace (`\u0008`)
`\f`	form feed (`\u000C`)
`\n`	newline (`\u000A`)
`\r`	return (`\u000D`)
`\t`	tab (`\u0009`)
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\u`$n_1n_2n_3n_4$	Unicode encoding

Table 2

Character Escape Sequences

sequences. Table 2 shows the most common permitted escape sequences. Arbitrary Unicode characters are denoted by a `\u`, followed by four hexadecimal digits enclosed in single quotes. For example, `'\u2122'` is the trademark symbol (™).

INTERNET You can find the encodings of tens of thousands of letters in many alphabets at `http://www.unicode.org`.

Conversions that don't incur information loss (such as `short` to `int` or `float` to `double`) are always legal. Values of type `char` can be converted to `int`. All integer types can be converted to `float` or `double`, even though some of the conversions (such as `long` to `double`) lose precision. All other conversions require a *cast:*

```
double x = 10.0 / 3.0; // sets x to 3.3333333333333335
int n = (int) x; // sets n to 3
float f = (float) x; // sets f to 3.3333333
```

It is not possible to convert between the `boolean` type and number types.

The `Math` class implements useful mathematical methods. Table 3 contains some of the most useful ones. The methods of the `Math` class do not operate on objects. Instead, numbers are supplied as parameters. (Recall that numbers are not objects in Java.) For example, here is how to call the `sqrt` method:

```
double y = Math.sqrt(x);
```

Since the method doesn't operate on an object, the class name must be supplied to tell the compiler that the `sqrt` method is in the `Math` class. In Java, every method must belong to some class.

Method	Description
`Math.sqrt(x)`	Square root of x, $\sqrt{x}$
`Math.pow(x, y)`	x^y ($x > 0$, or $x = 0$ and $y > 0$, or $x < 0$ and y is an integer)
`Math.toRadians(x)`	Converts x degrees to radians (i.e., returns $x \cdot \pi/180$)
`Math.toDegrees(x)`	Converts x radians to degrees (i.e., returns $x \cdot 180/\pi$)
`Math.round(x)`	Closest integer to x (as a `long`)
`Math.abs(x)`	Absolute value $\lvert x \rvert$

Table 3

Mathematical Methods

1.4 Control Flow Statements

The `if` statement is used for conditional execution. The `else` branch is optional.

```
if (x >= 0) y = Math.sqrt(x); else y = 0;
```

The `while` and `do` statements are used for loops. The body of a `do` loop is executed at least once.

```
while (x < target)
{
   x = x * a;
   n++;
}

do
{
   x = x * a;
   n++;
}
while (x < target);
```

The `for` statement is used for loops that are controlled by a loop counter.

```
for (i = 1; i <= n; i++)
{
   x = x * a;
   sum = sum + x;
}
```

A variable can be defined in a `for` loop. Its scope extends to the end of the loop.

```
for (int i = 1; i <= n; i++)
{
   x = x * a;
   sum = sum + x;
}
// i no longer defined here
```

Java 5.0 introduces an enhanced form of the `for` loop. We will discuss that construct later in this chapter.

1.5 Object References

In Java, an object value is always a *reference* to an object, or, in other words, a value that describes the location of the object. For example, consider the statement

```
Greeter worldGreeter = new Greeter("World");
```

An object reference describes the location of an object. In Java, you manipulate object references, not objects.

The value of the `new` expression is the location of the newly constructed object. The variable `worldGreeter` can hold the location of any `Greeter` object, and it is being filled with the location of the new object (see Figure 6.)

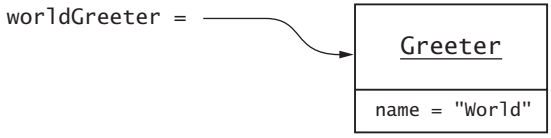


Figure 6

An Object Reference

There can be multiple variables that store references to the same object. For example, after the assignment

```
Greeter anotherGreeter = worldGreeter;
```

the two object variables refer to the same object (see Figure 7).

When you copy object references, the copy accesses the same object as the original.

If the `Greeter` class has a `setName` method that allows modification of the object, and if that method is invoked on the object reference, then both variables access the modified object.

```
anotherGreeter.setName("Dave");
// now worldGreeter also refers to the changed object
```

To make a copy of the actual object, instead of just copying the object reference, use the `clone` method. Implementing the `clone` method correctly is a subtle process that is discussed in greater detail in Chapter 7. However, many library classes have a `clone` method. It is then a simple matter to make a copy of an object of such a class. For example, here is how you clone a `Date` object:

```
Date aDate = . . .;
Date anotherDate = (Date) aDate.clone();
```

The cast `(Date)` is necessary because `clone` is a generic method with return type `Object`. In Java, all classes extend the class `Object`.

The special reference `null` refers to no object. You can set an object variable to `null`:

```
worldGreeter = null;
```

You can test if an object reference is currently `null`:

```
if (worldGreeter == null) . . .
```

The `null` reference refers to no object.

If you invoke a method on a `null` reference, a `NullPointerException` is thrown. Unless you supply a handler for the exception, the program terminates. (Exception handling is discussed later in this chapter.)

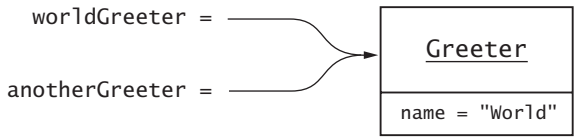


Figure 7

A Shared Object

It can happen that an object has no references pointing to it, namely when all object variables that previously referred to it are filled with other values or have been recycled. In that case, the memory that was used for storing the object will be automatically reclaimed by the garbage collector. In Java, you never need to manually recycle memory.

NOTE If you are familiar with the C++ programming language, you will recognize that object references in Java behave just like *pointers* in C++. In C++, you can have multiple pointers to the same value, and a `NULL` pointer points to no value at all. Of course, in C++, pointers strike fear in the hearts of many programmers because it is so easy to create havoc with invalid pointers. It is sometimes said that Java is easier than C++ because it has no pointers. That statement is not true. Java *always* uses pointers (and calls them references), so you don't have to worry about the distinction between pointers and values. More importantly, the pointers in Java are *safe*. It is not possible to create invalid pointers, and the garbage collector automatically reclaims unused objects.

1.6 Parameter Passing

The object reference on which you invoke a method is called the *implicit parameter*. In addition, a method may have any number of *explicit parameters* that are supplied between parentheses. For example, in the call

```
myGreeter.setName("Mars");
```

the reference stored in `myGreeter` is the implicit parameter, and the string `"Mars"` is the explicit parameter. The explicit parameters are so named because they are explicitly defined in a method, whereas the implicit parameter is implied in the method definition.

Occasionally, you need to refer to the implicit parameter of a method by its special name, `this`. For example, consider the following implementation of the `setName` method:

```
public class Greeter
{
   . . .
   /**
      Sets this greeter's name to the given name.
      @param name  the new name for this greeter
   */
   public void setName(String name)
   {
      this.name = name;
   }

   . . .
}
```

The `this` reference refers to the object on which a method was invoked.

The `this` reference refers to the object on which the method was invoked (such as `myGreeter` in the call `myGreeter.setName("Mars")`). The `name` field is set to the value of the explicit parameter that is also called `name`. In the example, the use of the `this` reference was necessary to resolve the ambiguity between the `name` field and the `name` parameter.

Occasionally, the `this` reference is used for greater clarity, as in the next example.

A method can change the state of an object whose reference it receives.

In Java, a method can modify the state of an object because the corresponding parameter variable is set to a copy of the passed object reference. Consider this contrived method of the `Greeter` class:

```
public class Greeter
{
   . . .
   /**
      Sets another greeter's name to this greeter's name.
      @param other  a reference to the other Greeter
   */
   public void copyNameTo(Greeter other)
   {
      other.name = this.name;
   }
   . . .
}
```

Now consider this call:

```
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
worldGreeter.copyNameTo(daveGreeter);
```

Figure 8 shows how the `other` parameter variable is initialized with the `daveGreeter` reference. The `copyNameTo` method changes `other.name`, and after the method returns, `daveGreeter.name` has been changed.

However, in Java, a method can never update the *contents of a variable* that is passed as a parameter. For example, after the call

```
worldGreeter.copyNameTo(daveGreeter);
```

the contents of `daveGreeter` is the same object reference before and after the call. It is not possible to write a method that would change the contents of `daveGreeter` to another object reference. In this regard, Java differs from languages such as C++ and C# that have a "call by reference" mechanism.

To see that Java does not support call by reference, consider yet another set of contrived methods. These methods try to modify a parameter, but they have no effect at all.

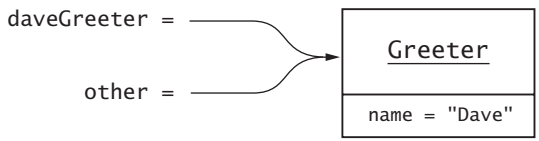


Figure 8

Accessing an Object through a Parameter Variable

```
public class Greeter
{
   . . .
   /**
      Tries to copy the length of this greeter's name into an integer variable.
      @param n the variable into which the method tries to copy the length
   */
   public void copyLengthTo(int n)
   {
      // this assignment has no effect outside the method
      n = name.length();
   }

   /**
      Tries to set another Greeter object to a copy of this object.
      @param other the Greeter object to initialize
   */
   public void copyGreeterTo(Greeter other)
   {
      // this assignment has no effect outside the method
      other = new Greeter(name);
   }
   . . .
}
```

Let's call these two methods:

```
int length = 0;
Greeter worldGreeter = new Greeter("World");
Greeter daveGreeter = new Greeter("Dave");
worldGreeter.copyLengthTo(length);
   // has no effect on the contents of length
worldGreeter.copyGreeterTo(daveGreeter);
   // has no effect on the contents of daveGreeter
```

Java uses "call by value" when passing parameters.

Neither method call has any effect. Changing the value of the parameter variable does not affect the variable supplied in the method call. Thus, Java has no call by reference. Java uses the "call by value" mechanism for both primitive types and object references.

1.7 Packages

Java classes can be grouped into *packages*. Package names are dot-separated sequences of identifiers, such as

```
java.util
javax.swing
com.sun.misc
edu.sjsu.cs.cs151.alice
```

Java uses packages to group related classes and to ensure unique class names.

To guarantee the uniqueness of package names, the inventors of Java recommend that you start a package name with a domain name in reverse (such as `com.sun` or `edu.sjsu.cs`), because domain names are guaranteed to be unique. Then use some other mechanism within your organization to ensure that the remainder of the package name is unique as well.

You place a class inside a package by adding a package statement as the first statement of the source file:

```
package edu.sjsu.cs.cs151.alice;
public class Greeter
{
   . . .
}
```

Any class without a package statement is in the "default package" with no package name.

The *full* name of a class consists of the package name followed by the class name, such as `edu.sjsu.cs.cs151.alice.Greeter`. Some full class name examples from the Java library are `java.util.ArrayList` and `javax.swing.JOptionPane`.

The `import` directive allows programmers to omit package names when referring to classes.

It is tedious to use these full names in your code. Use the `import` statement to use the shorter class names instead. For example, after you place a statement

```
import java.util.Scanner;
```

into your source file, then you can refer to the class simply as `Scanner`. If you simultaneously use two classes with the same short name (such as `java.util.Date` and `java.sql.Date`), then you are stuck—you must use the full name for one of them.

You can also import all classes from a package:

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

However, you never need to import the classes in the `java.lang` package, such as `String` or `Math`.

Organize your class files in directories that match the package names.

Large programs consist of many classes in multiple packages. The class files must be located in subdirectories that match the package names. For example, the class file `Greeter.class` for the class

```
edu.sjsu.cs.cs151.alice.Greeter
```

must be in a subdirectory

```
edu/sjsu/cs/cs151/alice
```

or

```
edu\sjsu\cs\cs151\alice
```

of the project's *base directory* (see Figure 9). The base directory is the directory that contains all package directories as well as any classes that are contained in the default package (that is, the package without a name).

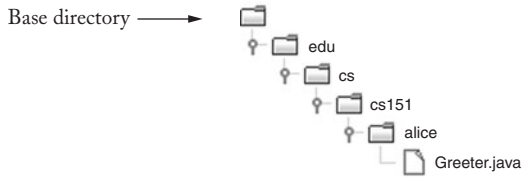


Figure 9

Package Name Must Match the Directory Path

Always compile from the base directory, for example

```
javac edu/sjsu/cs/cs151/alice/Greeter.java
```

or

```
javac edu\sjsu\cs\cs151\alice\Greeter.java
```

Then the class file is automatically placed in the correct location.

To run a program, you must start the Java virtual machine in the base directory and specify the full class name of the class that contains the main method:

```
java edu.sjsu.cs.cs151.alice.Greeter
```

1.8 Basic Exception Handling

When a program carries out an illegal action, an *exception* is generated. Here is a common example. Suppose you initialize a variable with the `null` reference, intending to assign an actual object reference later, but then you accidentally use the variable when it is still `null`.

```
String name = null;
int n = name.length(); // Illegal
```

When an exception occurs and there is no handler for it, the program terminates.

Applying a method call to `null` is illegal. The virtual machine now throws a `NullPointerException`. Unless your program handles this exception, it will terminate after displaying a *stack trace* (the sequence of pending method calls) such as this one:

```
Exception in thread "main" java.lang.NullPointerException
      at Greeter.sayHello(Greeter.java:25)
      at GreeterTester.main(GreeterTester.java:6)
```

Different programming errors lead to different exceptions. For example, trying to open a file with an illegal file name causes a `FileNotFoundException`.

Throw an exception to indicate an error condition that the current method cannot handle.

You can also throw your own exceptions if you find that a programmer makes an error when using one of your classes. For example, if you require that the parameter of one of your methods should be positive, and the caller supplies a negative value, you can throw an `IllegalArgumentException`:

```
if (n <= 0) throw new IllegalArgumentException("n should be > 0");
```

There are two categories of exceptions: checked and unchecked. If you call a method that might throw a checked exception, you must either declare it or catch it.

We will discuss the hierarchy of exception types in greater detail in Chapter 6. At this point you need to be aware of an important distinction between two kinds of exceptions, called *checked exceptions* and *unchecked exceptions*. The `NullPointerException` is an example of an unchecked exception. That is, the compiler does not check that your code handles the exception. If the exception occurs, it is detected at runtime and may terminate your program. The `IOException`, on the other hand, is a checked exception. If you call a method that might throw this exception, you must also specify how you want the program to deal with this failure.

In general, a checked exception is caused by an external condition beyond the programmer's control. Exceptions that occur during input and output are checked because the file system or network may spontaneously cause problems that the programmer cannot control. Therefore, the compiler insists that the programmer provide code to handle these situations.

On the other hand, unchecked exceptions are generally the programmer's fault. You should never get a `NullPointerException`. Therefore, the compiler doesn't tell you to provide a handler for a `NullPointerException`. Instead, you should spend your energy on making sure that the error doesn't occur in the first place. Either initialize your variables properly, or test that they aren't `null` before making a method call.

Whenever you write code that might cause a checked exception, you must take one of two actions:

1. Declare the exception in the method header.
2. Handle (or *catch*) the exception.

Consider this example. You want to read data from a file.

```
public void read(String filename)
{
   FileReader reader = new FileReader(filename);
   . . .
}
```

If there is no file with the given name, the `FileReader` constructor throws a `FileNotFoundException`. Because it is a checked exception, the compiler insists that you handle it. However, the implementor of the `read` method probably has no idea how to correct this situation. Therefore, the optimal remedy is to let the exception *propagate to its caller*. That means that the `read` method terminates, and that the exception is thrown to the method that called it.

Whenever a method propagates a *checked* exception, you must declare the exception in the method header, like this:

```
public void read(String filename) throws FileNotFoundException
{
   FileReader reader = new FileReader(filename);
   . . .
}
```

TIP There is no shame associated with acknowledging that your method might throw a checked exception—it is just "truth in advertising".

If a method can throw multiple exceptions, you list them all, separated by commas. Here is a typical example. As you will see in Chapter 7, reading objects from an object stream can cause both an `IOException` (if there is a problem with reading the input) and a `ClassNotFoundException` (if the input contains an object from an unknown class). A `read` method can declare that it throws both exceptions:

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

When you tag a method with a `throws` clause, the callers of this method are now put on notice that there is the possibility that a checked exception may occur. Of course, those calling methods also need to deal with these exceptions. Generally, the calling methods also add `throws` declarations. When you carry this process out for the entire program, the `main` method ends up being tagged as well:

```
public static void main(String[] args)
      throws IOException, ClassNotFoundException
{
   . . .
}
```

If an exception actually occurs, the `main` method is terminated, a stack trace is displayed, and the program exits.

However, if you write a professional program, you do not want the program to terminate whenever a user supplies an invalid file name. In that case, you want to *catch* the exception. Use the following syntax:

```
try
{
   . . .
   code that might throw an IOException
   . . .
}
catch (IOException exception)
{
   take corrective action
}
```

When an exception is thrown, the program jumps to the closest matching `catch` clause.

An appropriate corrective action might be to display an error message and to inform the user that the attempt to read the file has failed.

In most programs, the lower-level methods simply propagate exceptions to their callers. Some higher-level method, such as `main` or a part of the user interface, catches exceptions and informs the user.

For debugging purposes, you sometimes want to see the stack trace. Call the `printStackTrace` method like this:

```
try
{
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
   take corrective action
}
```

Occasionally, a method must carry out an action even if a prior statement caused an exception. A typical example is closing a file. A program can only open a limited number of files at one time, and it should close all files that it opened. To ensure that a file is closed even if an exception occurred during file processing, use the `finally` clause:

```
FileReader reader = null;
reader = new FileReader(name);
try
{
   . . .
}
finally
{
   reader.close();
}
```

Code in a `finally` clause is executed during normal processing and when an exception is thrown.

The `finally` clause is executed when the `try` block exits without an exception, and also when an exception is thrown inside the `try` block. In either case, the `close` method is called. Note that the `FileReader` constructor is *not* contained inside the `try` block. If the constructor throws an exception, then `reader` has not yet been assigned a value, and the `close` method should not be called.

1.9 Strings

Java strings are sequences of Unicode characters. The `charAt` method yields the individual characters of a string. String positions start at 0.

```
String greeting = "Hello";
char ch = greeting.charAt(1); // sets ch to 'e'
```

A Java string is an immutable sequence of Unicode characters.

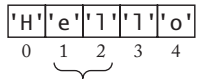
Java strings are *immutable*. Once created, a string cannot be changed. Thus, there is no `setCharAt` method. This may sound like a severe restriction, but in practice it isn't. For example, suppose you initialized `greeting` to `"Hello"`. You can still change your mind:

```
greeting = "Goodbye";
```

The string object `"Hello"` hasn't changed, but `greeting` now refers to a different string object.

The `length` method yields the length of a string. For example, `"Hello".length()` is 5.

Figure 10
Extracting a Substring



Note that the empty string "" of length 0 is different from `null`—a reference to no string at all.

The `substring` method computes substrings of a string. You need to specify the positions of the first character that you want to include in the substring and the first character that you no longer want to include. For example, `"Hello".substring(1, 3)` is the string `"el"` (see Figure 10). Note that the difference between the two positions equals the length of the substring.

Since strings are objects, you need to use the `equals` method to compare whether two strings have the same contents.

```
if (greeting.equals("Hello")) . . . // OK
```

If you use the == operator, you only test whether two string references have the identical *location*. For example, the following test fails:

```
if ("Hello".substring(1, 3) == "el") . . . // NO
```

The substring is not at the same location as the constant string `"el"`, even though it has the same contents.

You have already seen the string concatenation operator: `"Hello, " + name` is the concatenation of the string `"Hello, "` and the string object to which `name` refers.

If either argument of the + operator is a string, then the other is *converted to a string*. For example,

```
int n = 7;
String greeting = "Hello, " + n;
```

constructs the string `"Hello, 7"`.

If a string and an object are concatenated, then the object is converted to a string by invoking its `toString` method. For example, the `toString` method of the `Date` class in the `java.util` package returns a string containing the date and time that is encapsulated in the `Date` object. Here is what happens when you concatenate a string and a `Date` object:

```
// default Date constructor sets current date/time
Date now = new Date();
String greeting = "Hello, " + now;
// greeting is a string such as "Hello, Wed Jan 18 16:57:18 PST 2006"
```

Sometimes, you have a string that contains a number, for example the string `"7"`. To convert the string to its number value, use the `Integer.parseInt` and `Double.parseDouble` methods. For example,

```
String input = "7";
n = Integer.parseInt(input); // sets n to 7
```

If the string doesn't contain a number, or contains additional characters besides a number, the unchecked `NumberFormatException` is thrown.

1.10 Reading Input

The Scanner class can be used to read input from the console or a file.

Starting with Java 5.0, the simplest way to read input in a Java program is to use the `Scanner` class. To read console input, construct a `Scanner` from `System.in`. Call the `nextInt` or `nextDouble` method to read an integer or a floating-point number. For example,

```
Scanner in = new Scanner(System.in);
System.out.print("How old are you? ");
int age = in.nextInt();
```

If the user types input that is not a number, an (unchecked) `InputMismatchException` is thrown. You can protect yourself against that exception by calling the `hasNextInt` or `hasNextDouble` method before calling `nextInt` or `nextDouble`.

The `next` method reads the next whitespace-delimited token, and `nextLine` reads the next input line.

You can read input from a file by constructing a `Scanner` from a `FileReader`. For example, the following loop reads all lines from the file `input.txt`:

```
Scanner in = new Scanner(new FileReader("input.txt"));
while (in.hasNextLine())
{
   String line = in.nextLine();
   . . .
}
```

1.11 Array Lists and Linked Lists

The `ArrayList` class of the `java.util` package lets you collect a sequence of objects of any type. The `add` method adds an object to the end of the array list.

```
ArrayList<String> countries = new ArrayList<String>();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

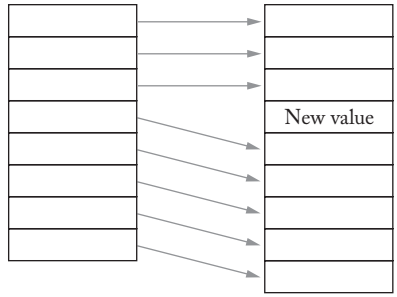
Starting with Java 5.0, the `ArrayList` class is a *generic* class with a *type parameter*. The type parameter (`String` in our example) denotes the type of the list elements. You can form array lists of any type, except for primitive types. For example, you can use an `ArrayList<Date>` but not an `ArrayList<int>`.

The `size` method returns the number of elements in the array list. The `get` method returns the element at a given position; legal positions range from 0 to `size() - 1`. For example, the following loop prints all elements of the `countries` list:

```
for (int i = 0; i < countries.size(); i++)
{
   String country = countries.get(i);
   System.out.println(country);
}
```

Figure 11

Inserting into an Array List



This loop is so common that Java 5.0 introduces a convenient shortcut: the *enhanced* `for` loop or "for each" loop:

```
for (String country : countries)
   System.out.println(country);
```

In each loop iteration, the variable before the `:` is set to the next element of the `countries` list.

The `set` method lets you overwrite an existing element with another:

```
countries.set(1, "France");
```

If you access a nonexistent position (`< 0` or `>= size()`), then an `IndexOutOfBoundsException` is thrown.

Finally, you can insert and remove elements in the middle of the array list.

```
countries.add(1, "Germany");
countries.remove(0);
```

An array list is a collection of objects that supports efficient access to all storage locations.

A linked list is a collection of objects that supports efficient insertion and removal of elements. You use an iterator to traverse a linked list.

These operations move the remaining elements up or down. The name "array list" signifies that the public interface allows both array operations (`get/set`) and list operations (`add/remove`).

Inserting and removing elements in the middle of an array list is not efficient. All elements beyond the location of insertion or removal must be moved (see Figure 11). A *linked list* is a data structure that supports efficient insertion and removal at any location. When inserting or removing an element, all elements stay in place, and only the neighboring links are rearranged (see Figure 12). The standard Java library supplies a class `LinkedList` implementing this data structure.

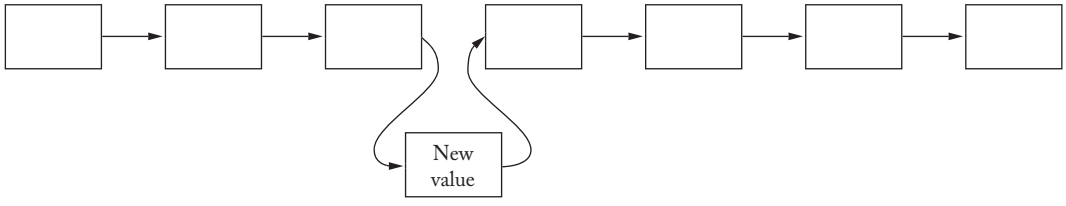


Figure 12

Inserting into a Linked List

As with array lists, you use the `add` method to add elements to the end of a linked list.

```
LinkedList<String> countries = new LinkedList<String>();
countries.add("Belgium");
countries.add("Italy");
countries.add("Thailand");
```

However, accessing elements in the middle of the linked list is not as simple. You don't want to access a position by an integer index. To find an element with a given index, it is necessary to follow a sequence of links, starting with the head of the list. That process is not very efficient. Instead, you need an *iterator*, an object that can access a position anywhere in the list:

```
ListIterator<String> iterator = countries.listIterator();
```

The `next` method advances the iterator to the next position of the list and returns the element that the iterator just passed (see Figure 13). The `hasNext` method tests whether the iterator is already past the last element in the list. Thus, the following loop prints all elements in the list:

```
while (iterator.hasNext())
{
   String country = iterator.next();
   System.out.println(country);
}
```

To add an element in the middle of the list, advance an iterator past the insert location and call `add`:

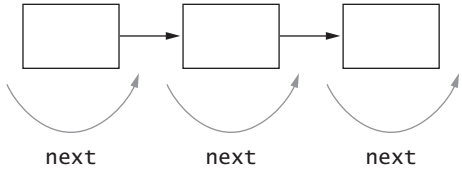
```
iterator = countries.listIterator();
iterator.next();
iterator.add("France");
```

To remove an element from the list, call `next` until you jump over the element that you want to remove, then call `remove`. For example, this code removes the second element of the `countries` list.

```
iterator = countries.listIterator();
iterator.next();
iterator.next();
iterator.remove();
```

Figure 13

Iterator Movement



1.12 Arrays

Array lists and linked lists have one drawback—they can only hold objects, not values of primitive types. *Arrays*, on the other hand, can hold sequences of arbitrary values. You construct an array as

```
new T[n]
```

where `T` is any type and `n` any integer-valued expression. The array has type `T[]`.

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

An array stores a fixed number of values of any given type.

Now `numbers` is a reference to an array of 10 integers—see Figure 14.

When an array is constructed, its elements are set to zero, `false`, or `null`.

The length of an array is stored in the `length` field.

```
int length = numbers.length;
```

Note that an empty array of length 0

```
new int[0]
```

is different from `null`—a reference to no array at all.

You access an array element by enclosing the index in brackets, such as

```
numbers[i] = i * i;
```

If you access a nonexistent position (`< 0` or `>= length`), then an `ArrayIndexOutOfBoundsException` is thrown.

As with array lists, you can use the "for each" loop to traverse the elements of an array. For example, the loop

```
for (int n : numbers)
   System.out.println(n);
```

is a shorthand for

```
for (int i = 0; i < numbers.length; i++)
   System.out.println(numbers[i]);
```

There is a convenient shorthand for constructing and initializing an array. Enclose the array elements in braces, like this:

```
int[] numbers = { 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 };
```

Occasionally, it is convenient to construct an *anonymous array*, without storing the array reference in a variable. For example, the `Polygon` class has a constructor

```
Polygon(int[] xvalues, int[] yvalues, int n);
```

You can construct a triangle by calling

```
Polygon triangle = new Polygon(
   new int[] { 0, 10, 5 }, // anonymous array of integers
   new int[] { 10, 0, 5 }, // another anonymous array
   3);
```

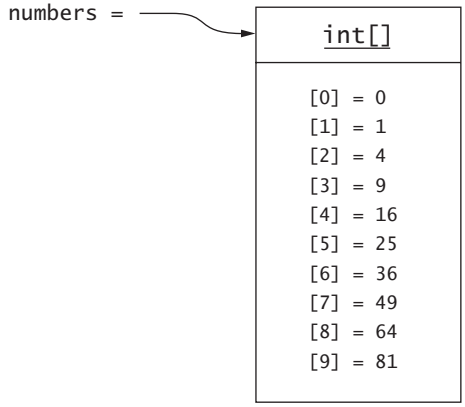


Figure 14
An Array Reference

After an array has been constructed, you cannot change its length. If you want a larger array, you have to construct a new array and move the elements from the old array to the new array.

You can obtain a two-dimensional array like this:

```
int[][] table = new int[10][20];
```

You access the elements as `table[row][column]`.

When you launch a program by typing its name into a command shell, then you can supply additional information to the program by typing it after the program name. The entire input line is called the command line, and the strings following the program name are the command-line arguments. The `args` parameter of the `main` method is an array of strings, the strings specified in the command line. The first string after the class name is `args[0]`. For example, if you invoke a program as

```
java GreeterTester Mars
```

then `args.length` is 1 and `args[0]` is `"Mars"` and not `"java"` or `"GreeterTester"`.

Java 5.0 introduces methods with a variable number of parameters. When you call such a method, you supply as many parameter values as you like. The parameter values are automatically placed in an array. Here is an example:

```
public double sum(double... values)
{
   double sum = 0;
   for (double v : values) sum += v;
   return sum;
}
```

The `...` symbol indicates that the method takes a variable number of parameters of type `double`. The parameter variable `values` is actually an array of type `double[]`. If you call the method, for example as

```
double result = sum(0.25, -1, 10);
```

then the `values` parameter is initialized with `new double[] { 0.25, -1, 10 }`.

1.13 Static Fields and Methods

Occasionally, you would like to share a variable among all objects of a class. Here is a typical example. The `Random` class in the `java.util` package implements a random number generator. It has methods such as `nextInt`, `nextDouble`, and `nextBoolean` that return random integers, floating-point numbers, and Boolean values. For example, here is how you print 10 random integers:

```
Random generator = new Random();
for (int i = 1; i <= 10; i++)
   System.out.println(generator.nextInt());
```

Let's use a random number generator in the `Greeter` class:

```
public String saySomething()
{
   if (generator.nextBoolean())
      return "Hello, " + name + "!";
   else
      return "Goodbye, " + name + "!";
}
```

It would be wasteful to supply each `Greeter` object with its own random number generator. To share one generator among all `Greeter` objects, declare the field as `static`:

```
public class Greeter
{
   . . .
   private static Random generator = new Random();
}
```

A static field belongs to the class, not to individual objects.

The term "static" is an unfortunate and meaningless holdover from C++. A static field is more accurately called a *class variable:* there is only a single variable for the entire class.

Class variables are relatively rare. A more common use for the `static` keyword is to define constants. For example, the `Math` class contains the following definitions:

```
public class Math
{
   . . .
   public static final double E = 2.7182818284590452354;
   public static final double PI = 3.14159265358979323846;
}
```

The keyword `final` denotes a constant value. After a `final` variable has been initialized, you cannot change its value.

These constants are public. You refer to them as `Math.PI` and `Math.E`.

A static method (or class method) is a method that does not operate on an object. You have already encountered static methods such as `Math.sqrt` and `JOptionPane.showInputDialog`. Another use for static methods is *factory methods*, methods that return an object, similar to a constructor. Here is a factory method for the `Greeter` class that returns a greeter object with a random name:

```
public class Greeter
{
   public static Greeter getRandomInstance()
   {
      if (generator.nextBoolean())
         return new Greeter("Venus");
      else
         return new Greeter("Mars");
   }
   . . .
}
```

A static method is not invoked on an object.

You invoke this method as `Greeter.getRandomInstance()`. Note that static methods can access static fields but not instance fields—they don't operate on an object.

Static fields and methods have their place, but they are quite rare in object-oriented programs. If your programs contain many static fields and methods, then this may mean that you have missed an opportunity to discover sufficient classes to implement your program in an object-oriented manner. Here is a bad example that shows how you can write very poor non-object-oriented programs with static fields and methods:

```
public class BadGreeter
{
   public static void main(String[] args)
   {
      name = "World";
      printHello();
   }

   public static void printHello() // Bad style
   {
      System.out.println("Hello, " + name + "!");
   }

   private static String name; // Bad style
}
```

1.14 Programming Style

Class names should always start with an uppercase letter and use mixed case, such as `String`, `StringTokenizer`, and so on. Package names should always be lowercase, such as `edu.sjsu.cs.cs151.alice`. Field and method names should always start with a lowercase letter and use mixed case, such as `name` and `sayHello`. Underscores are not commonly used in class or method names. Constants should be in all uppercase with an occasional underscore, such as `PI` or `MAX_VALUE`.

Follow the standard naming conventions for classes, methods, fields, and constants.

These are not requirements of the Java language but a convention that is followed by essentially all Java programmers. Your programs would look very strange to other Java programmers if you used classes that started with a lowercase letter or methods that started with an uppercase letter. It is not considered good style by most Java programmers to use prefixes for fields (such as `_name` or `m_name`).

It is very common to use `get` and `set` prefixes for methods that get or set a property of an object, such as

```
public String getName()
public void setName(String aName)
```

However, a Boolean property has prefixes `is` and `set`, such as

```
public boolean isPolite()
public void setPolite(boolean b)
```

Use a consistent style for braces. We suggest that you line up { and } in the same row or column.

There are two common brace styles: The "Allmann" style in which braces line up, and the compact but less clear "Kernighan and Ritchie" style. Here is the `Greeter` class, formatted in the Kernighan and Ritchie style.

```
public class Greeter {
   public Greeter(String aName) {
      name = aName;
   }

   public String sayHello() {
      return "Hello, " + name + "!";
   }

   private String name;
}
```

We use the Allmann style in this book.

Some programmers list fields before methods in a class:

```
public class Greeter
{
   private String name;
      // Listing private features first is not a good idea

   public Greeter(String aName)
   {
      . . .
   }
   . . .
}
```

However, from an object-oriented programming point of view, it makes more sense to list the public interface first. That is the approach we use in this book.

Make sure that you declare all instance fields `private`.

Except for `public static final` fields, all fields should be declared `private`. If you omit the access specifier, the field has *package visibility*—all methods of classes in the same package can access it—an unsafe practice that you should avoid. Anyone can add classes to a package at any time. Therefore, there is an open-ended and uncontrollable set of methods that can potentially access fields with package visibility.

It is technically legal—as a sop to C++ programmers—to declare array variables as

```
int numbers[]
```

You should avoid that style and use

```
int[] numbers
```

That style clearly shows the type `int[]` of the variable.

All classes, methods, parameters, and return values should have documentation comments.

You should put spaces *around* binary operators and after keywords, but not after method names.

Good	Bad
`x > y`	`x>y`
`if (x > y)`	`if(x > y)`
`Math.sqrt(x)`	`Math.sqrt (x)`

You should not use *magic numbers*. Use named constants (`final` variables) instead. For example, don't use

```
h = 31 * h + val[off]; // Bad—what's 31?
```

What is 31? The number of days in January? The position of the highest bit in an integer? No, it's the hash multiplier.

Instead, declare a local constant in the method

```
final int HASH_MULTIPLIER = 31
```

or a static constant in the class (if it is used by more than one method)

```
private static final int HASH_MULTIPLIER = 31
```

Then use the named constant:

```
h = HASH_MULTIPLIER * h + val[off]; // Much better
```

INTERNET The CheckStyle program (`http://checkstyle.sourceforge.net`) can automatically check the quality of your code. It reports misaligned braces, missing documentation comments, and many other style errors.

Exercises

Exercise 1.1. Add a `sayGoodbye` method to the `Greeter` class and add a call to test the method in the `GreeterTester` class (or test it in BlueJ).

Exercise 1.2. What happens when you run the Java interpreter on the `Greeter` class instead of the `GreeterTester` class? Try it out and explain.

Exercise 1.3. Add comments to the `GreeterTester` class and the `main` method. Document `args` as "unused". Use `javadoc` to generate a file `GreeterTester.html`. Inspect the file in your browser.

Exercise 1.4. Bookmark `docs/api/index.html` in your browser. Find the documentation of the `String` class. How many methods does the `String` class have?

Exercise 1.5. Write a program that prints "Hello, San José". Use a `\u` escape sequence to denote the letter é.

Exercise 1.6. What is the Unicode character for the Greek letter "pi" (π)? For the Chinese character "bu" (不)?

Exercise 1.7. Run the `javadoc` utility on the `Greeter` class. What output do you get? How does the output change when you remove some of the documentation comments?

Exercise 1.8. Download and install the `DocCheck` utility. What output do you get when you remove some of the documentation comments of the `Greeter` class?

Exercise 1.9. Write a program that computes and prints the square root of 1000, rounded to the nearest integer.

Exercise 1.10. Write a program that computes and prints the sum of integers from 1 to 100 and the sum of integers from 100 to 1000. Create an appropriate class `Summer` that has no `main` method for this purpose. If you don't use BlueJ, create a second class with a `main` method to construct two objects of the `Summer` class.

Exercise 1.11. Add a `setName` method to the `Greeter` class. Write a program with two `Greeter` variables that refer to the same `Greeter` object. Invoke `setName` on one of the references and `sayHello` on the other. Print the return value. Explain.

Exercise 1.12. Write a program that sets a `Greeter` variable to `null` and then calls `sayHello` on that variable. Explain the resulting output. What does the number behind the file name mean?

Exercise 1.13. Write a test program that tests the `setName`, `copyNameTo`, `copyLengthTo`, and `copyGreeterTo` methods of the examples in Section 1.6, printing out the parameter variables before and after the method call.

Exercise 1.14. Write a method `void swapNames(Greeter other)` of the `Greeter` class that swaps the names of this greeter and another.

Exercise 1.15. Write a program in which `Greeter` is in the package `edu.sjsu.cs.`*yourcourse*`.`*yourname* and `GreeterTester` is in the default package. Into which directories do you put the source files and the class files?

Exercise 1.16. What is wrong with the following code snippet?

```
ArrayList<String> strings;
strings.add("France");
```

Exercise 1.17. Write a `GreeterTester` program that constructs `Greeter` objects for all command-line arguments and prints out the results of calling `sayHello`. For example, if your program is invoked as

```
java GreeterTester Mars Venus
```

then the program should print

```
Hello, Mars!
Hello, Venus!
```

Exercise 1.18. What are the values of the following?

(a) `2 + 2 + "2"`

(b) `"" + countries`, where `countries` is an `ArrayList` filled with several strings

(c) `"Hello" + new Greeter("World")`

Write a small sample program to find out, then explain your answers.

Exercise 1.19. Write a program that prints the sum of its command-line arguments (assuming they are numbers). For example,

```
java Adder 3 2.5 -4.1
```

should print `The sum is 1.4`

Exercise 1.20. Write a program that reads input data from a file and prints the minimum, maximum, and average value of the input data. The file name should be specified on the command line. Use a class `DataAnalyzer` and a separate class `DataAnalyzerTester`.

Exercise 1.21. Write a `GreeterTester` program that asks the user "What is your name?" and then prints out `"Hello, `*username*`"`.

Exercise 1.22. Write a class that can generate random strings with characters in a given set. For example,

```
RandomStringGenerator generator = new RandomStringGenerator();
generator.addRange('a', 'z');
generator.addRange('A', 'Z');
String s = generator.nextString(10);
   // A random string consisting of ten lowercase
   // or uppercase English characters
```

Your class should keep an `ArrayList<Range>` of `Range` objects.

Exercise 1.23. Write a program that plays TicTacToe with a human user. Use a class `TicTacToeBoard` that stores a 3×3 array of `char` values (filled with `'x'`, `'o'`, or space characters). Your program should use a random number generator to choose who begins. When it's the computer's turn, randomly generate a legal move. When it's the human's turn, read the move and check that it is legal.

Exercise 1.24. Improve the performance of the `getRandomInstance` factory method by returning one of two fixed `Greeter` objects (stored in static fields) rather than constructing a new object with every call.

Exercise 1.25. Use any ZIP utility or the `jar` program from the Java SDK to uncompress the `src.zip` file that is part of the Java SDK. Then look at the source code of the `String` class in `java/lang/String.java`. How many style rules do the programmers violate? Look at the `hashCode` method. How can you rewrite it in a less muddleheaded way?

Exercise 1.26. Look inside the source code of the class `java.awt.Window`. List the instance fields of the class. Which of them are private, and which of them have package visibility? Are there any other classes in the `java.awt` package that access those fields? If not, why do you think that they are not private?