Chapter 7. Object-Oriented Programming in Java

Topics in This Chapter

- Instance variables: Creating classes with named fields
- Methods: Adding functions to classes
- Constructors: Defining functions that help build classes
- Destructors: Understanding why Java doesn't need functions to destroy objects
- Overloading: Creating more than one method with the same name
- Javadoc: Making hypertext documentation for your classes
- Inheritance: Reusing and augmenting capabilities from other classes
- Interfaces: Describing the behavior of multiple different classes
- Packages: Organizing classes
- Classpath: Telling Java where to look for classes
- Modifiers: Specifying which parts of your class are exposed

Objects are central to the Java programming language. Understanding how they are created and used is the first task for a beginning Java programmer. Understanding objects in Java is even more fundamental than the basic syntax summarized in Chapter 8. If you've never seen object-oriented programming, you'll want to take your time with this chapter. As usual, trying things out is more important than reading; be sure to write several of your own classes along the way. The time spent will more than pay for itself in increased productivity later. If you have worked with objects in other languages, you can skim most of these sections. But pay close attention to Sections 7.7 (Javadoc), 7.9 (Interfaces and Abstract Classes), and 7.10 (Packages, Classpath, and JAR Archives).

7.1 Instance Variables

In the simplest case, a class is like a structure or a record. An object (an instance of a class) is normally created by `new` before a call to a class constructor. The constructor looks like a method with the same name as the class being created. For example:

```
Point p1 = new Point(2, 4);
Color red = new Color(255, 0, 0);
```

Sometimes, however, the call to `new` can be hidden. For instance, method calls often return an object; the methods might call `new` internally or might return an object that already exists. For instance:

```
OutputStream out = someSocket.getOutputStream();
Point p1 = someWindow.location();
```

In a very few cases, such as for strings and arrays, Java has shorthand syntax for creating an object. For instance, the following are equivalent ways of creating a `String` object:

```
String string1 = new String("A String");
String string2 = "Another String";
```

The "fields" or "data members" of a class are often called "instance variables" in Java nomenclature. They are accessed through

```
objectReference.variableName
```

that is, by supplying the name of the instance variable separated by a dot from the reference to the actual object.

To illustrate, Listing 7.1 shows a class named `Ship1` used to represent a boat, perhaps for a simple simulation system. The test routine creates two `Ship1` instances with `new Ship1()`, then initializes the various instance variables. These fields are then updated to represent one "move" of the ship, and the new values for the speed and location of the ships are printed out.

Listing 7.1 `Test1.java`

```
// Create a class with five instance variables (fields):
// x, y, speed, direction, and name. Note that Ship1 is
// not declared "public", so it can be in the same file as
// Test1. A Java file can only contain one "public" class
// definition.

class Ship1 {
  public double x, y, speed, direction;
  public String name;
}

// The "driver" class containing "main".

public class Test1 {
  public static void main(String[] args) {
    Ship1 s1 = new Ship1();
    s1.x = 0.0;
    s1.y = 0.0;
    s1.speed = 1.0;
    s1.direction = 0.0;   // East
  s1.name = "Ship1";
    Ship1 s2 = new Ship1();
    s2.x = 0.0;
    s2.y = 0.0;
    s2.speed = 2.0;
    s2.direction = 135.0; // Northwest
    s2.name = "Ship2";
    s1.x = s1.x + s1.speed
           * Math.cos(s1.direction * Math.PI / 180.0);
    s1.y = s1.y + s1.speed
```

```
           * Math.sin(s1.direction * Math.PI / 180.0);
    s2.x = s2.x + s2.speed
           * Math.cos(s2.direction * Math.PI / 180.0);
    s2.y = s2.y + s2.speed
           * Math.sin(s2.direction * Math.PI / 180.0);
    System.out.println(s1.name + " is at ("
                       + s1.x + "," + s1.y + ").");
    System.out.println(s2.name + " is at ("
                       + s2.x + "," + s2.y + ").");
  }
}
```

Compiling and Running:

```
javac Test1.java
java Test1
```

Output:

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

You may have noticed a pattern in the variable names and classnames used in this example. A standard convention in Java is to name local variables and instance variables with the initial letter in lowercase (e.g., `someString`, `window`, `outputStream1`) and to name classes with a leading uppercase letter (e.g., `String`, `Window`, `OutputStream`). Subsequent "words" in the variable or class name typically have leading uppercase letters (e.g., `someInstanceVariable`, `SomeClass`), although that notation is not quite as universal a convention, since some people prefer underscores (`some_instance_variable`, `Some_Class`). Constants typically are all uppercase (`PI`). These conventions help people reading your code, and we suggest that you adopt them.

Core Approach

Name variables with an initial lowercase letter (`myVar`).For class names, use an initial uppercase letter (`MyClass`).

Now, the astute reader may observe that the example of Listing 7.1 appears to violate this naming convention. For instance, if `Math` is some global variable containing an object with a `PI` constant, why is it named `Math` instead of `math`? It turns out that this is not a global variable (Java has no such thing, in fact!) but is indeed the name of the Java `Math` class. Furthermore, in addition to instance variables, Java allows class variables: variables that are shared by all members of the class. These variables are indicated by the `static` keyword in their declaration and can be accessed either through an object reference or through the class name. So, the naming convention actually makes things clearer here; a reader who has never seen the `Math` class can tell that `PI` is a `static final` (constant) variable in the `Math` class simply by seeing the reference to `Math.PI`.

7.2 Methods

In the previous example, virtually identical code was repeated several times to update the `x` and `y` instance variables of the two ships. This approach represents poor coding style, not only because the initial repetition takes time, but, more importantly, because updates require changing code in multiple places. To solve this problem, classes can have functions associated with them, not just data, as in Listing 7.2. Java calls them "methods" as in Lisp/CLOS rather than "member functions" as in C++. Notice that, unlike the case with C++, instance variables can be directly initialized in the declaration, as with

```
public double x=0.0;
```

Note also the use of `public` and `private`. These modifiers are discussed in more detail in Section 7.11, but the basic point is that you use `public` for functionality that you are deliberately making available to users of your class. You use `private` for functionality that you use internally to implement your class but do not want exposed to users of your class.

Listing 7.2 `Test2.java`

```
// Give the ship public move and printLocation methods.

class Ship2 {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name = "UnnamedShip";

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }

  public void move() {
    double angle = degreesToRadians(direction);
    x = x + speed * Math.cos(angle);
    y = y + speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at " +
                       "(" + x + "," + y + ").");
  }
}
public class Test2 {

  public static void main(String[] args) {
    Ship2 s1 = new Ship2();
    s1.name = "Ship1";
    Ship2 s2 = new Ship2();
    s2.direction = 135.0; // Northwest
    s2.speed = 2.0;
    s2.name = "Ship2";
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Compiling and Running:

```
javac Test2.java
java Test2
```

Output:

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

7.3 Constructors and the "this" Reference

A class constructor is a special routine used to build an object. A constructor is called when you use `new ClassName(...)` to build an instance of a class. Constructors are defined similarly to an ordinary public method, except that the name must match the class name and no return type is given. Note that if you include a return type (e.g., `public` **`void`** `Ship2(...) {...}`), the class will compile without warning on most Java systems, but your constructor will not be called when you try to invoke it.

Core Warning

Be sure your constructor does not specify a return type.

You can define constructors with any number of arguments to let the user supply parameters at the time the object is instantiated. If you fail to define a constructor in the class, the Java compiler automatically provides the class an empty, zero-argument constructor, as in

```
public SomeClass() { }
```

However, if you define *any* constructors for the class, then the default zero-argument constructor is not added by the compiler. Therefore, if you provide any constructors in your class that contain arguments and you would still like a zero-argument constructor, then you will need to explicitly type in a zero-argument constructor.

Core Note

If you do not define constructors in your class, then the Java compiler will provide a default, zero-argument constructor. However, if you define any constructors in your class, then the compiler does not automatically provide the zero-argument constructor if one is missing.

A drawback to `Ship2` is that changing multiple fields takes multiple steps. A more convenient approach is to specify all of the fields when the ship is created. Furthermore, some people feel that relying on default values makes the code more difficult to read, since someone looking only at the code that creates a ship would not know what default values the various fields are assigned. So, we could make an improved `Ship3` with a constructor like the following:

```
public Ship3(double x, double y, ...) {
  // Initialize fields
}
```

However, this presents a problem: the local variable named `x` "shadows" (hides) the instance variable of the same name. So,

```
public Ship3(double x, double y, ...) {
  x = x;
  y = y;
  ...
}
```

is perfectly legal, but not too useful. All you'd be doing is reassigning the local variables of the method back to their current values. One alternative is to simply use different names, as follows:

```
public Ship3(double inputX, double inputY, ...) {
  x = inputX;
  y = inputY;
  ...
}
```

A second alternative is to use the `this` reference, as in Listing 7.3. Inside any class you can always use `this` to refer to the current instance of the class. In addition, `this` is often used to pass to external routines a reference to themselves. You can legally use `this` to refer to internal fields or methods, so that the `move` method could be implemented as:

```
public void move() {
  double angle = this.degreesToRadians(this.direction);
  this.x = this.x + this.speed * Math.cos(this.angle);
  this.y = this.y + this.speed * Math.sin(this.angle);
}
```

instead of the much simpler

```
public void move() {
  double angle = degreesToRadians(direction);
  x = x + speed * Math.cos(angle);
  y = y + speed * Math.sin(angle);
}
```

However, the former is quite cumbersome and tedious, so we recommend that you save `this` for situations that require it: namely, passing references to the current object to external routines and differentiating local variables from fields with the same names.

Listing 7.3 `Test3.java`

```
// Give Ship3 a constructor to let the instance variables
// be specified when the object is created.

class Ship3 {
  public double x, y, speed, direction;
  public String name;

  public Ship3(double x, double y, double speed,
               double direction, String name) {
    this.x = x; // "this" differentiates instance vars
    this.y = y; //  from local vars.
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }

  public void move() {
    double angle = degreesToRadians(direction);
    x = x + speed * Math.cos(angle);
    y = y + speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at " +
                       "(" + x + "," + y + ").");
  }
}
```

```
public class Test3 {
  public static void main(String[] args) {
    Ship3 s1 = new Ship3(0.0, 0.0, 1.0,   0.0, "Ship1");
    Ship3 s2 = new Ship3(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move();
    s1.printLocation();
    s2.printLocation();
  }
}
```

Compiling and Running:

```
javac Test3.java

java Test3
```

Output:

```
Ship1 is at (1,0).
Ship2 is at (-1.41421,1.41421).
```

Static Initialization Blocks

If you need something a little more complex than default variable values but a little less complicated than constructors, you can use a `static` initialization block, which is executed when the *class* is loaded (a class is loaded the first time an instance of the class is instantiated or when a `static` variable or method of the class is accessed). Here's an example:

```
public class SomeClass {
  int[] values = new int[12];

  static {
    for(int i=0; i<values.length; i++) {
      values[i] = 2 * i + 5;
    }
  }

  int lastValue = values[11];

  ...
}
```

In most cases, such behavior is placed in the class constructor. You probably won't want `static` initializers very often either.

7.4 Destructors

This section is intentionally left blank.

Just kidding, but destructors (functions to destroy objects) aren't needed in Java. If no reference to an object exists, then the garbage collector frees up the memory for you automatically. If the only reference to an object is a local variable, the object is available for collection when the method exits, or earlier if the variable is reassigned. If an instance variable has the only reference to an object, the object can be collected whenever the variable is reassigned. Not having destructors seems amazing to C++ programmers, but it really works. No dangling pointers: Java will *not* collect an object if a forgotten reference to it is still hanging around somewhere. No memory leaks: Java *will* collect any object that can't be reached from another live object, even if it has nonzero references (as with objects in a circularly linked structure disconnected from everything else). You still have to worry about "leaklets" (stashing a reference in an array or variable and forgetting to reassign it to `null` or some other value), but they are a relatively minor problem.

Although Java will collect all unused objects automatically, you sometimes want to do some bookkeeping when an object is destroyed. For instance, you might want to decrement a count, write a log to disk, or some such. For this kind of situation, you can use the `finalize` method of an object:

```
protected void finalize() throws Throwable {
  doSomeBookkeeping();
  super.finalize(); // Use parent's finalizer
}
```

Don't worry about the `throws` business or how Java knows which methods have `finalize` methods; the details will become clear later. For now, just declare the method exactly as written but do whatever you want for the `doSomeBookkeeping` part.

7.5 Overloading

As in C++ and other object-oriented languages, Java allows more than one method with the same name but with different behaviors, depending on the type or number of its arguments. For instance, you could define two `isBig` methods: one that determines if a `String` is "big" (by some arbitrary measure) and another that determines if an `int` is "big," as follows:

```
public boolean isBig(String s) {
  return(s.length() > 10);
}

public boolean isBig(int n) {
  return(n > 1000);
}
```

Note that

```
return(n > 1000);
```

is a more compact way of accomplishing the same thing as

```
if (n > 1000) {
  return(true);
} else {
  return(false);
}
```

In Listing 7.4, the `Ship4` constructor and the `move` method are overloaded. One constructor can call another constructor in the class by using `this(args)`, but the call has to be the first line of the constructor. Also, don't confuse the `this` constructor call with the `this` reference. For example:

```
public class SomeClass {
  public SomeClass() {
    this(12); // Invoke other constructor
    doSomething();
  }

  public SomeClass(int num) {
    doSomethingWith(num);
    doSomeOtherStuff();
  }

  ...
}
```

We also want to define a new version of `move` that lets you specify the number of "steps" the ship should move. If you assume that you will create a new method but leave the original one unchanged, the question is whether the new `move` should use the old version, as follows:

```
public void move() {
  double angle = degreesToRadians(direction);
  x = x + speed * Math.cos(angle);
  y = y + speed * Math.sin(angle);
}

public void move(int steps) {
  for(int i=0; i<steps; i++) {
    move();
  }
}
```

or if it should repeat the code, as in the following version:

```
public void move() {
  double angle = degreesToRadians(direction);
  x = x + speed * Math.cos(angle);
  y = y + speed * Math.sin(angle);
}

public void move2(int steps) {
  double angle = degreesToRadians(direction);
  x = x + (double)steps * speed * Math.cos(angle);
  y = y + (double)steps * speed * Math.sin(angle);
}
```

The first approach has the advantage that changes to the way in which movement is calculated only have to be implemented in one location but has the disadvantage that significant extra calculations are performed. This example illustrates a common dilemma: the tension between reusability and efficiency. In some instances, reuse can be achieved with no performance reduction. In others, performance has to be traded off against extensibility and reuse, and the appropriate balance depends on the situation. In this particular case, it is possible to get the best of both worlds by modifying the original `move` method, as follows:

```
public void move() {
  move(1);
}

public void move(int steps) {
  double angle = degreesToRadians(direction);
  x = x + (double)steps * speed * Math.cos(angle);
  y = y + (double)steps * speed * Math.sin(angle);
}
```

You can find this type of solution more often than you might think, so you should look for such an approach whenever you are faced with a similar problem. However, this approach is not possible if the original `move` was located in a class that we could not modify.

Listing 7.4 gives the full class definition.

Listing 7.4 `Test4.java`

```
class Ship4 {
  public double x=0.0, y=0.0, speed=1.0, direction=0.0;
  public String name;

  // This constructor takes the parameters explicitly.

  public Ship4(double x, double y, double speed,
               double direction, String name) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.direction = direction;
    this.name = name;
  }

  // This constructor requires a name but lets you accept
  // the default values for x, y, speed, and direction.
  public Ship4(String name) {
    this.name = name;
  }

  private double degreesToRadians(double degrees) {
    return(degrees * Math.PI / 180.0);
  }

  // Move one step.

  public void move() {
    move(1);
  }

  // Move N steps.

  public void move(int steps) {
    double angle = degreesToRadians(direction);
    x = x + (double)steps * speed * Math.cos(angle);
    y = y + (double)steps * speed * Math.sin(angle);
  }

  public void printLocation() {
    System.out.println(name + " is at (" + x + "," + y + ").");
  }
}

public class Test4 {
  public static void main(String[] args) {
    Ship4 s1 = new Ship4("Ship1");
    Ship4 s2 = new Ship4(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move(3);
    s1.printLocation();
    s2.printLocation();
  }
}
```

Compiling and Running:

```
javac Test4.java
java Test4
```

Output:

```
Ship1 is at (1,0).
Ship2 is at (-4.24264,4.24264).
```

7.6 Public Version in Separate File

Classes used in a single place are often combined in the same file as in the previous examples. Often, however, classes are designed to be reused and are placed in separate files where multiple other classes can have public access to them. For instance, Listing 7.5 defines the `Ship` class, and Listing 7.6 defines a driver routine that tests the class. When a developer is building reusable classes, there is a much greater burden to be sure the code is documented and extensible. Two particular strategies help in this regard.

Replace public instance variables with accessor methods.

Instead of direct access to the variables, a common practice is to create a pair of helping methods to set and retrieve the values. For instance, in the ship example, instead of the `x` and `y` fields being `public`, they are made `private`, and `getX`, `getY`, `setX`, and `setY` methods (accessor methods) are created to enable users to look up and modify the fields. Although this strategy appears to be considerable extra work, the time is well invested for classes that are widely used.

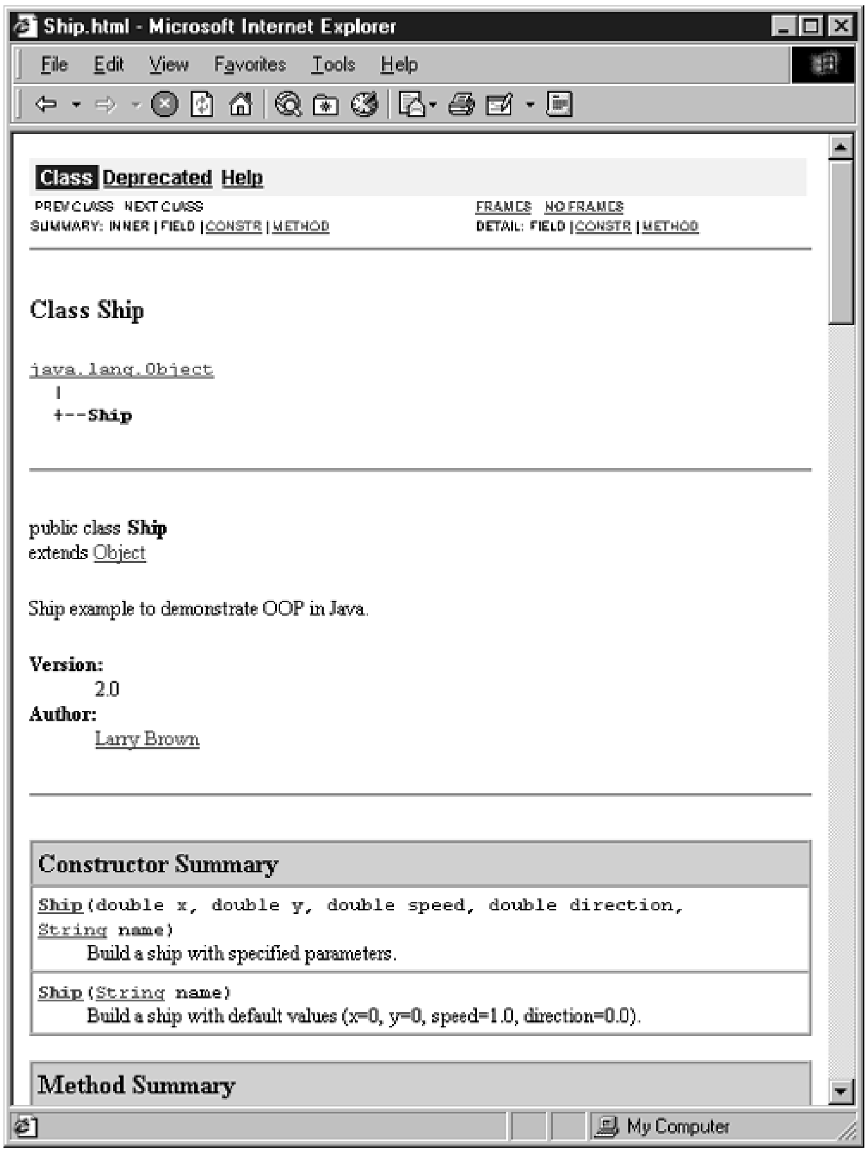
First of all, accessor methods provide a placeholder for later functionality. For instance, suppose that the developer decides to provide error checking to ensure that directions are nonnegative or that the ship's maximum speed wasn't exceeded. If users explicitly manipulated the variables directly, then there would be no mechanism for performing this check without having all the users change their code. But if `setX` and `setY` methods were already in place, checking legal values could be performed without any changes in user code. Similarly, suppose that the ship becomes part of a simulation and a graphical representation needs to be updated every time the `x` and `y` locations change. The `setX` and `setY` methods provide a perfect place for changing the values.

Second, using accessor methods shields users of the class from implementation changes. Suppose that the developer decides to use a `Point` data structure to store `x` and `y` instead of storing them individually. If the `x` and `y` variables are referenced directly, users of the class would have to change their code. But if the variables can only be modified through accessor methods, the definitions of `getX` and `getY` could be updated with no required changes by users of the class.

Use javadoc to create on-line documentation.

Documentation enclosed between `/**` and `*/` can be used by the `javadoc` program to create hypertext documentation for all the nonprivate methods and variables (if any). The `javadoc` command-line program is described in Section 7.7.Again, this approach is likely to require considerable extra effort but is well worth that effort for classes that are frequently used by multiple developers. For instance, Figure 7-1 was generated directly from the documentation of `Ship.java` (Listing 7.5).

Figure 7-1. Javadoc will generate hypertext documentation from Java source code.


Ship.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Class Deprecated Help
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
My Computer

Class Ship

```
java.lang.Object
  |
  +--Ship
```

public class **Ship**
extends Object

Ship example to demonstrate OOP in Java.

Version:
2.0
Author:
Larry Brown

Constructor Summary

`Ship(double x, double y, double speed, double direction, String name)`
Build a ship with specified parameters.

`Ship(String name)`
Build a ship with default values (x=0, y=0, speed=1.0, direction=0.0).

Method Summary

Listing 7.5 `Ship.java`

```
/** Ship example to demonstrate OOP in Java.
 *
 * @author <A HREF="mailto:larry@corewebprogramming.com">
 *         Larry Brown</A>
 * @version 2.0
 */

public class Ship {
  // Instance variables

  private double x=0.0, y=0.0, speed=1.0, direction=0.0;
  private String name;

  // Constructors

  /** Build a ship with specified parameters. */
```

```
public Ship(double x, double y, double speed,
            double direction, String name) {
  setX(x);
  setY(y);
  setSpeed(speed);
  setDirection(direction);
  setName(name);
}

/** Build a ship with default values
 *  (x=0, y=0, speed=1.0, direction=0.0).
 */

public Ship(String name) {
  setName(name);
}

/** Move ship one stepsat current speed/direction. */

public void move() {
  moveInternal(1);
}

/** Move N steps. */

public void move(int steps) {
  moveInternal(steps);
}

private void moveInternal(int steps) {
  double angle = degreesToRadians(direction);
  x = x + (double)steps * speed * Math.cos(angle);
  y = y + (double)steps * speed * Math.sin(angle);
}

private double degreesToRadians(double degrees) {
  return(degrees * Math.PI / 180.0);
}

/** Report location to standard output. */

public void printLocation() {
  System.out.println(getName() + " is at (" + getX() +
                     "," + getY() + ").");
}

/** Get current X location. */

public double getX() {
  return(x);
}

/** Set current X location. */

public void setX(double x) {
      this.x = x;
    }

    /** Get current speed. */

    public double getSpeed() {
      return(speed);
    }

    /** Set current speed. */

    public void setSpeed(double speed) {
      this.speed = speed;
    }

    /** Get current heading (0=East, 90=North, 180=West,
     *  270=South).  I.e., uses standard math angles, <B>not</B>
     *  nautical system where 0=North, 90=East, etc.
     */

    public double getDirection() {
      return(direction);
    }

    /** Set current direction (0=East, 90=North, 180=West,
     *  270=South). I.e., uses standard math angles,<B>not</B>
     *  nautical system where 0=North,90=East, etc.
     */

    public void setDirection(double direction) {
      this.direction = direction;
    }

    /** Get Ship's name. Can't be modified by user. */

    public String getName() {
      return(name);
    }

    private void setName(String name) {
      this.name = name;
    }
  }
```

Listing 7.6 `shipTest.java`

```
public class ShipTest {
  public static void main(String[] args) {
    Ship s1 = new Ship("Ship1");
    Ship s2 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship2");
    s1.move();
    s2.move(3);
    s1.printLocation();
    s2.printLocation();
  }
}
```

Compiling and Running:

```
javac ShipTest.java
java ShipTest
```

The first line calls `javac` on `Ship.java` automatically. The Java compiler automatically checks the file creation date of each `.class` file used by the class it compiles, compares it to the creation date of the source file (if available), and recompiles if necessary. Java does not require makefiles as in C/C++; all classes *directly* used are recompiled if out of date. If you want to keep a given `.class` file in use even while you are updating the `.java` source, simply keep the source in a separate directory.

Core Note

You don't need makefiles in Java. Use `javac file` *to compile a class and recompile any out-of-date classes it* ***directly*** *uses.*

Output:

```
Ship1 is at (1,0).
Ship2 is at (-4.24264,4.24264).
```

7.7 Javadoc

The `javadoc` program is distributed with Sun's JDK and by most third-party vendors. The command-line program uses comments enclosed between `/**` and `*/` to generate on-line documentation that includes hypertext links to other user-defined and system classes. You generally place `javadoc` comments above the class definition and before every nonprivate method and class. The first sentence of the variable or method description is placed in the alphabetized index at the top of the resultant page, with the full description available in the detailed sections at the bottom. To create the documentation, supply one or more file or package names to `javadoc`, as follows:

```
javadoc Foo.java
javadoc Foo.java Bar.java
javadoc graphics.newWidgets
javadoc graphics.newWidgets math.geometry
```

You can also supply various options to `javadoc`. These options are described later in this section, but here are some simple examples:

```
javadoc -author -version SomeClass.java
javadoc -noindex -notree somePackage
```

Packages are described in Section 7.10.

By default, the `javadoc` program uses a standard doclet template to generate HTML-formatted API output. The doclet template recognizes HTML markup, not just standard text, inside your comments. Thus, you can use `<A HREF="...">` to create hypertext links to your organization's home page, use the `IMG` element to include a screen dump of your program in action, and even use the `APPLET` element to load an interactive demonstration that illustrates some of the most important uses of your class. You probably want to avoid headings, however, since they are likely to break up the outline `javadoc` generates. Extra white space is ignored inside these comments unless they occur inside a space-preserving element such as `PRE`.

If you would like a different overall HTML format for the generated API, you can write your own custom doclets, since doclets are simply Java programs that use the doclet API to specify the content and format of the output from the `javadoc` tool. The following URLs provide complete documentation for the `javadoc` tool and doclets.

Javadoc 1.3 Home Page

http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/

Javadoc Tool Reference Page

http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html

http://java.sun.com/j2se/1.3/docs/tooldocs/win32/javadoc.html

Doclet Overview and Doclet API

http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html

http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/doclet/

Javadoc Tags

In addition to HTML markup and text, there are some special tags that you can insert in the documentation to get certain effects. For instance, you can use `@param` to describe individual parameters that a method expects, `@return` to describe the return value, and `@see` to reference related classes or methods. Here's an example:

```
/** Converts an angle in degrees to one in radians.
 *
 * @param degrees The input angle in degrees.
 * @return The equivalent of the input, in radians.
 * @see #radiansToDegrees.
 */

public double degreesToRadians(double degrees) {
  return(degrees * Math.PI / 180.0);
}
```

These special tags must occur at the beginning of a line, and all text following the tag and before the next tag or the end of the comment is considered part of the tagged paragraph.

The legal tags are described below. Unless otherwise noted, the tag is supported in JDK 1.1 and later.

@author

This tag is used in the class documentation to specify the author. You must use "`javadoc -author ...`" for this specification to take effect. HTML markup is allowed here; for example:

```
/** Description for SomeClass
 *   ...
 * @author <A HREF="mailto:ellison@microsoft.com">
 *         Larry Ellison</A>
 */
```

@deprecated

This tag indicates classes, fields, or methods that should no longer be used, that is, ones that are currently supported for backward compatibility but that might be dropped in future releases.

{@docRoot} [Java 1.3]

This tag represents a relative path from the generated document to the root document page in a package hierarchy. This tag is useful when you are generating documentation for a large package that has numerous subdirectories and you would like to refer to a common page, for example, a copyright or license page, located in the root directory. For example, if a link to licensing information is needed in the documentation for

`corewebprogramming/Ship.java`, then

```
<a href="{@docRoot}/license.html">
```

would resolve to

```
<a href="../license.html">
```

assuming here that `license.html` is located in the `corewebprogramming` directory.

@exception

This tag documents methods and constructors and should be followed by an exception classname, a space, and a description.

{@link} [Java 1.2]

This tag is similar to the `@see` tag; however, instead of a See Also section being created in the document, an inline hyperlink is created. This approach is useful in a descriptive paragraph for creating links to other methods. For example, in `Ship.java`, a descriptive paragraph could contain

```
To change the location of a ship use
{@link #move(int) move}.
```

which is translated to

```
To change the location of a ship use
<a href="Ship.html#move(int)">move</a>.
```

@param

The `@param` tag documents the arguments that a method (or constructor) takes. The tag should be followed by the parameter name, a space, and the description of the parameter.

@return

This tag documents the return value of a method.

@see

This tag creates hypertext links to the `javadoc` documentation for other methods or classes. Methods should normally be prefaced by the classname and a #, but the classname can be omitted for methods in the current class. For instance:

```
/**
 * ...
 * @see #getWidget
 * @see Widget
 * @see Frob#setWidget
 */
```

```
public void setWidget(Widget w) { ... }
```

@serial [Java 1.2]

@serialData [Java 1.2]

@serialField [Java 1.2]

These three tags provide a way to document the form for a `Serializable` class. The first tag describes a default field that is serializable in the class. The second tag describes any `ObjectStreamField`s, and the third tag describes the sequence and types of any data that is `Externalizable`. For detailed information on these tags, see http://java.sun.com/j2se/1.3/docs/guide/serialization/spec/serial-arch.doc6.html" .

@since

This tag creates a "Since" section used to document when the class, field, or method was added. It is typically used for classes or packages undergoing revisions to distinguish new features from preexisting ones.

@throws [Java 1.2]

This tag defines an `Exception` that the method or class may throw. The tag is immediately followed by the exception classname, a space, and then a description of the exception.

@version

This tag is used in the class documentation to list a version number. You must use "`javadoc -version ...`" for this specification to take effect.

Javadoc Command-Line Arguments

The `javadoc` tool lets you supply a number of options to customize its behavior. For instance, authors often use

```
javadoc -author -version -noindex -notree Class.java
```

to honor the `@author` and `@version` tags and to suppress generation of the index and class hierarchy. As of JDK 1.3, over 40 options flags are available for `javadoc` when the standard doclet template is used. Below we summarize only the most frequently used option flags.

-author This flags tells `javadoc` to include the author information, which is omitted by default.

-bottom [Java 1.2] The `bottom` flag specifies text to be placed at the bottom of each output file below the lower navigation bar. If the text contains white space, then enclose the text in quotes.

-classpath

-sourcepath

These flags tell `javadoc` where to look for the specified `.java` files. Multiple directories should be separated by a colon on Solaris and a semicolon on Windows. Directories specified in the `CLASSPATH` variable are searched automatically.

-d This flag supplies a destination directory for the HTML files. An absolute or relative directory can be used. If the flag is omitted, the HTML files are placed in the same directory as the source file. You often use `-d` to specify a directory that already contains the `images` subdirectory holding the GIF files used by documentation files.

-encoding

-docencoding

These flags allow you to specify the character encoding used for the source file (`-encoding`) or that should be used for the documentation (`-docencoding`).

-link [Java 1.2]

-linkoffline [Java 1.2]

The `link` flag tells `javadoc` where to look to resolve links to other packages when generating the documentation. For example, if you would like to resolve the links to the on-line documentation at java.sun.com for Java classes found in your documentation, you would use

```
-link http://java.sun.com/j2se/1.3/docs/api
```

The `javadoc` program actually uses the `package-list` file located at the above URL to correctly resolve the links for the external packages. If Internet access is not available, you can use the `linkoffline` to point to a local copy of the `package-list` file. If you've locally installed the JDK 1.3 documentation, then the list for the Java packages is located at `/root/jdk1.3/docs/api`. Thus, to resolve the external links locally on a Windows platform, you might use

```
-linkoffline http://java.sun.com/j2se/1.3/docs/api
             c:\jdk1.3\docs\api
```

-J The `javadoc` tool is itself a Java program. The `-J` flag lets you pass parameters directly to the Java runtime. For instance, if the documentation set will be very large, you might want to specify a large (24 megabyte) startup size through

```
javadoc -J-ms24m ...
```

-nodeprecated

-nodeprecatedlist [Java 1.2]

The first flag tells `javadoc` to omit all deprecated entries. The second flag omits the `deprecated-list.html` page and the deprecation link in the navigation bar but still generates deprecated information throughout the documentation.

-noindex This tells `javadoc` not to generate the `AllNames.html` index. The index is useful for packages, but you often want to omit it when building documentation for a single class.

-nonavbar [Java 1.2] This tag suppresses the generation of the navigation bar found at the top and bottom of pages.

-notree This tells `javadoc` not to generate the `tree.html` class hierarchy.

-public

-protected

-package -private

These flag options let you specify which classes and members get documented. They are ordered here from the most restrictive to the most inclusive, and each option automatically includes all of the ones above it in the list. The default is `protected`, which generates documentation only for `public` and `protected` members. This documentation is what users of your class normally want, but developers that are changing the class might want to know about `private` data as well.

-splitindex [Java 1.2] This flag specifies a separate index file for each letter in the index, instead of a single alphabetical index file. In addition, a single, separate file is generated for index entries that do not start with an alphabetical character.

-verbose This flag tells `javadoc` to print information about the time taken to parse each file.

-version The `-version` flag tells `javadoc` to include version numbers (as specified through `@version`). They are omitted by default.

-windowtitle [Java 1.2] This flag defines the title used when generating the HTML document and appears in the browser window title. The text, preceding the flag and enclosed in quotes, is placed in the `<title> ... </title>` container within the HTML document. The title cannot contain any HTML markup tags.

For example, the following `javadoc` command was used on Windows 98 with JDK 1.3 to generate the HTML documentation shown in Figure 7-1.

```
>javadoc -author -version -noindex -notree
         -linkoffline
         http://java.sun.com/j2se/1.3/docs/api
         c:\jdk1.3\docs\api Ship.java
```

For a large list of command options, you can exceed the character limit for some shells. In such cases, you need to create a makefile to use with `javadoc`. For example makefiles on Windows, see http://java.sun.com/products/jdk/javadoc/makefiles. Makefiles for other platforms are given in the Javadoc Tool Reference Page for the various platforms at http://java.sun.com/j2se/1.3/docstooldocs/javadoc/index.html.

7.8 Inheritance

Inheritance is the process by which a new class is built on top of a previously written class without the existing class's functionality needing to be rewritten. The `extends` keyword is used to indicate that one class inherits from another. The original class is usually referred to as the *parent class* or *superclass* (called base class in C++ lingo). The new class is known as the *child class* or *subclass* (called derived class in C++ lingo). The new child class inherits the nonprivate instance variables and methods but *not* the constructors of the parent class. However, a constructor in a child class can explicitly call the constructor in the parent class by using `super` on the first line of the new constructor. Here is a simplified example:

```
// ParentClass.java: original parent/base/superclass.
// Has fields a and b and methods foo and bar.
// Has two constructors.
public class ParentClass {
  public int a;
  private int b;

  public ParentClass() { ... }
  public ParentClass(double z) { ... }

  public String foo(int x, String s) { ... }

  private void bar() { ... }
}

// ChildClass.java: the new child/derived/subclass.
// Has fields a (by inheritance) and c, and methods
// foo (by inheritance) and baz. Has one constructor
// which uses ParentClass's constructor.

public class ChildClass extends ParentClass {
  public int c;

  public ChildClass(double z) {
    super(z); // call ParentClass's constructor
    ...
  }

  public void baz(boolean isReady) { ... }
}
```

Be aware that if the subclass constructor does not explicitly call a constructor in the superclass through the keyword `super`, then the zero-argument constructor in the superclass is implicitly called before any code is executed in the subclass constructor. Furthermore, if the superclass does not have a zero-argument constructor, then a compile error is produced when the subclass is compiled. Remember that the compiler will add a default, zero-argument constructor to any class that does not define constructors. Also, remember that if any constructors are explicitly defined in a class, then the compiler does not automatically add a default constructor.

Core Note

When calling a constructor in the superclass, you must make the `super` call the first line in the subclass constructor. If no constructor in the superclass is explicitly called, then the zero-argument constructor in the superclass is implicitly called.

One benefit of inheritance is the ability to "override" a method inherited from the parent class and provide a new implementation of the method in the child class. The overridden method in the child class must have the same signature: method name, parameter list, and return type. If needed, the new method can access the previous version by calling

```
super.overriddenMethodName(...)
```

You cannot invoke `super.super` to access overridden methods higher up in the inheritance hierarchy, though. Also, if desired, you can relax the visibility of an overridden method to provide greater access; a method declared `protected` in the parent class can be redeclared `public` in the child class.

An initial misconception about inheritance is that the keyword `super` is necessary to access any of the methods in the parent class. The keyword `super` is really only necessary when both the parent class and the child class have a method with the same name—the child class has overridden the method in the parent class—and you would like to access the method in the parent class from the child class. Otherwise, the inherited methods (non-overridden methods) are immediately available in the child class directly through the method name and the keyword `super` is not required.

Core Note

Inherited methods are immediately available to the child class through the method name. The keyword `super` is only necessary when you want to access the parent version of an inherited method that has been overridden in child class.

Technically, you can override variables inherited from the parent class. However, overriding a variable is *never* done in practice, since an instance of the child class would have two variables with the same name—one variable is exposed when a method is executed in the child class and the other variable is exposed when an inherited method is executed in the parent class. Because the two variables have the same name, this behavior is very difficult to debug, so, for good practice, do not override variables.

Following, we illustrate an example that uses inheritance and overrides a method. Listing 7.7 presents a `Speedboat` class that adds color methods and variables and overrides the `printLocation` method. A driver for the class is given in Listing 7.8.

Listing 7.7 `Speedboat.java`

```
/** A fast Ship. Red and going 20 knots by default. */

public class Speedboat extends Ship {
  private String color = "red";

  /** Builds a red Speedboat going N at 20 knots. */

  public Speedboat(String name) {
    super(name);
    setSpeed(20);
  }

  /** Builds a speedboat with specified parameters. */

  public Speedboat(double x, double y, double speed,
                   double direction, String name,
                   String color) {
    super(x, y, speed, direction, name);
    setColor(color);
  }

  /** Report location. Override version from Ship. */

  public void printLocation() {
    System.out.print(getColor().toUpperCase() + " ");
    super.printLocation();
  }

  /** Gets the Speedboat's color. */

  public String getColor() {
    return(color);
  }

  /** Sets the Speedboat's color. */

  public void setColor(String colorName) {
    color = colorName;
  }
}
```

Listing 7.8 `SpeedboatTest.java`

```
/** Try a couple of Speedboats and a regular Ship. */

public class SpeedboatTest {
  public static void main(String[] args) {
    Speedboat s1 = new Speedboat("Speedboat1");
    Speedboat s2 = new Speedboat(0.0, 0.0, 2.0, 135.0,
                                 "Speedboat2", "blue");
    Ship s3 = new Ship(0.0, 0.0, 2.0, 135.0, "Ship1");
    s1.move();
    s2.move();
    s3.move();
    s1.printLocation();
    s2.printLocation();
    s3.printLocation();
  }
}
```

Compiling and Running:

```
javac -depend SpeedboatTest.java
java SpeedboatTest
```

The first line above calls `javac` on `Speedboat.java` and `Ship.java` automatically.

Output:

```
RED Speedboat1 is at (20,0).
BLUE Speedboat2 is at (-1.41421,1.41421).
Ship1 is at (-1.41421,1.41421).
```

Java, unlike C++ and Lisp/CLOS but like Smalltalk, supports single inheritance only. This means that although your class can have many ancestor classes, it can have only one *immediate* parent. This behavior is normally what you want; in some situations, though, multiple inheritance is useful, but Java doesn't support it. However, many of the benefits of multiple inheritance can be attained with less complexity by the use of *interfaces,* which is discussed in the next section.

7.9 Interfaces and Abstract Classes

Suppose that you want to define a class to act as a parent of other classes, but you don't want to let people directly instantiate the class. For instance, you might want to provide some common behavior in the class, but the class will not have enough information to be used by itself. In this case, Java lets you declare a class `abstract`, and the compiler will not let you build an instance of the class. For instance, Listing 7.9 shows an abstract `Shape` class. The example says that all `Shape` subclasses will have methods to look up and set locations (`getX`, `getY`, and so forth) but that you are prohibited from directly building a `Shape` object.

Listing 7.9 `Shape.java`

```
/** The parent class for all closed, open, curved, and
 *  straight-edged shapes.
 */

public abstract class Shape {
  protected int x, y;

  public int getX() {
    return(x);
  }

  public void setX(int x) {
    this.x = x;
  }

  public int getY() {
    return(y);
  }

  public void setY(int y) {
    this.y = y;
  }
}
```

Java also lets you define abstract methods—methods that define the return type and parameters but don't provide a method body—as follows:

```
public ReturnType methodName(Type1 arg1, Type2 arg2);
```

Classes that contain an abstract method *must* be declared abstract. Their subclasses must also be abstract *unless* they implement all of the abstract methods in the superclass. Abstract methods are useful when you want to require all members of a class to have certain general categories of behavior but where each member of the class will implement the behavior slightly differently.

Since the `Shape` class does not define abstract methods, its subclasses can be abstract or concrete. In this particular case, we create two further abstract subclasses for curved shapes and for those with straight edges. See Listings 7.10 and 7.11.

Listing 7.10 `Curve.java`

```
/** A curved shape (open or closed). Subclasses will include
 *  arcs and circles.
 */

public abstract class Curve extends Shape {}
```

Listing 7.11 `StraightEdgedShape.java`

```
/** A Shape with straight edges (open or closed). Subclasses
 *  will include Line, LineSegment, LinkedLineSegments,
 *  and Polygon.
 */

public abstract class StraightEdgedShape extends Shape {}
```

Now, suppose that we want to extend the `Curve` class to create a `Circle` and we want the `Circle` to have a method to calculate its area. We could simply have `Circle` extend `Curve` (*without* including the `abstract` declaration) and include a `getArea` method. This approach works fine if circles are the only shapes whose area can be measured. But if we planned ahead for a `Rectangle` class descended from `StraightEdgedShape`, we might be dissatisfied with this approach, since the areas of rectangles can be measured as well. Creating general routines that deal with areas will now be difficult because `Rectangle` and `Circle` have no common ancestor containing `getArea`. For instance, how would we easily make an array of shapes whose areas can be summed? A `Circle[]` wouldn't allow rectangles, and a `Rectangle[]` also would be too restrictive, since it would exclude circles. But a `Shape[]` would not be restrictive enough, since it would permit shapes that didn't have a `getArea` method. Similarly, if we wanted to return the larger area from two shapes, what argument types should such a `maxArea` method take? Not `Shape`, since some shapes lack `getArea`. Putting an abstract `getArea` method into `Shape` doesn't make sense since some shapes (line segments, arcs, and such) won't have measurable areas. In fact, there *is* no clean approach if we use regular classes.

Fortunately, Java has a solution for just this type of dilemma: interfaces. Interfaces look like abstract classes where all of the methods are abstract. The big difference is that a class can directly implement *multiple* interfaces, whereas it can only directly extend a single class. A class that implements an interface must either provide definitions for all methods or declare itself abstract. To illustrate, Listing 7.12 shows a simple interface.

Listing 7.12 `Interface1.java`

```
public interface Interface1 {
  ReturnType1 method1(ArgType1 arg);
  ReturnType2 method2(ArgType2 arg);
}
```

Note that the `interface` keyword is used instead of `class` and that the method declarations end in a semicolon with no method body (just like abstract methods). Also note that the `public` declarations before methods in an interface are optional; the methods are implicitly public, so the explicit declarations are normally omitted. Listing 7.13 shows a class that uses the interface.

Listing 7.13 `Class1.java`

```
// This class is not abstract, so it must provide
// implementations of method1 and method2.

public class Class1 extends SomeClass
                    implements Interface1 {
  public ReturnType1 method1(ArgType1 arg) {
    someCodeHere();
    ...
  }

  public ReturnType2 method2(ArgType2 arg) {
    someCodeHere();
    ...
  }

  ...
}
```

Listing 7.14 presents another interface.

Listing 7.14 `Interface2.java`

```
public interface Interface2 {
 ReturnType3 method3(ArgType3 arg);
}
```

Next, Listing 7.15 outlines a class that uses *both* interfaces. It is abstract, so does not have to provide implementations of the methods of the interfaces.

Listing 7.15 `Class2.java`

```
// This class is abstract, so does not have to provide
// implementations of the methods of Interface 1 and 2.

public abstract class Class2 extends SomeOtherClass
                             implements Interface1,
                                        Interface2 {
  ...
}
```

Finally, Listing 7.16 shows a concrete subclass of `Class2`.

Listing 7.16 `Class3.java`

```
// This class is not abstract, so it must provide
// implementations of method1, method2, and method3.

public class Class3 extends Class2 {
  public ReturnType1 method1(ArgType1 arg) {
    someCodeHere();
    ...
  }

  public ReturnType2 method2(ArgType2 arg) {
    someCodeHere();
    ...
  }

  public ReturnType3 method3(ArgType3 arg) {
    someCodeHere();
    ...
  }

  ...
}
```

Since interfaces do not contain method definitions, they cannot be directly instantiated. However, an interface can `extend` (not `implement`) one or more other interfaces (use commas if you extend more than one), so that interface definitions can be built up hierarchically just like class definitions can. Interfaces cannot include normal instance variables but can include constants. The `public`, `static`, and `final` declarations for the constants are implicit, so are normally omitted. Listing 7.17 gives an example.

Listing 7.17 `Interface3.java`

```
// This interface has three methods (by inheritance) and
// two constants.

public interface Interface3 extends Interface1,
                                    Interface2 {
  int MIN_VALUE = 0;
  int MAX_VALUE = 1000;
}
```

Now, how does all this help us with the `Shape` class hierarchy? Well, the key point is that methods can refer to an interface as though it were a regular class. So, we first define a `Measurable` interface, as shown in Listing 7.18. Note that it is common to end interface names with "able" (`Runnable`, `Serializable`, `Sortable`, `Drawable`, and so on).

Listing 7.18 `Measurable.java`

```
/** Used in classes with measurable areas. */

public interface Measurable {
  double getArea();
}
```

Next, we define a `Circle` class that implements this interface, as shown in Listing 7.19.

Listing 7.19 `Circle.java`

```
/** A circle. Since you can calculate the area of
 *  circles, class implements the Measurable interface.
 */

public class Circle extends Curve implements Measurable {
  private double radius;
  public Circle(int x, int y, double radius) {
    setX(x);
    setY(y);
    setRadius(radius);
  }

  public double getRadius() {
    return(radius);
  }

  public void setRadius(double radius) {
    this.radius = radius;
  }

  /** Required for Measurable interface. */

  public double getArea() {
    return(Math.PI * radius * radius);
  }
}
```

Having `Measurable` as an interface lets us define a class that uses `Measurable` without having to know which specific classes actually implement it. Listing 7.20 gives an example.

Listing 7.20 `MeasureUtil.java`

```
/** Some operations on Measurable instances. */

public class MeasureUtil {
  public static double maxArea(Measurable m1,
                               Measurable m2) {
    return(Math.max(m1.getArea(), m2.getArea()));
  }

  public static double totalArea(Measurable[] mArray) {
    double total = 0;
    for(int i=0; i<mArray.length; i++) {
      total = total + mArray[i].getArea();
    }
    return(total);
  }
}
```

Because of this design, other classes can implement the `Measurable` interface and automatically become available to the `MeasureUtil` methods. For instance, Listings 7.21 and 7.22 show an abstract `Polygon` class and one of its concrete subclasses. All polygons should have measurable areas but will be calculated differently.

Listing 7.21 **Polygon.java**

```java
/** A closed Shape with straight edges. */

public abstract class Polygon extends StraightEdgedShape
                              implements Measurable {
  private int numSides;

  public int getNumSides() {
    return(numSides);
  }

  protected void setNumSides(int numSides) {
    this.numSides = numSides;
  }
}
```

Listing 7.22 **Rectangle.java**

```java
/** A rectangle implements the getArea method. This satisfies
 *  the Measurable interface, so rectangles can be instantiated.
 */

public class Rectangle extends Polygon {
  private double width, height;

  public Rectangle(int x, int y,
                   double width, double height) {
    setNumSides(2);
    setX(x);
    setY(y);
    setWidth(width);
    setHeight(height);
  }

  public double getWidth() {
    return(width);
  }
  public void setWidth(double width) {
    this.width = width;
  }

  public double getHeight() {
    return(height);
  }

  public void setHeight(double height) {
    this.height = height;
  }

  /** Required to implement Measurable interface. */

  public double getArea() {
    return(width * height);
  }
}
```

Although *we* took a long time explaining what was going on, using interfaces did not substantially increase the *code* required in the `Shape` hierarchy. The `Measurable` interface took three lines, and the only other thing needed was the "`implements Measurable`" declaration. However, the interface saved a large amount of work in the `MeasureUtil` class and made it immune to changes in which classes actually have `getArea` methods. Listing 7.23 shows a simple test of the `MeasureUtil` class.

It has been widely claimed in the Java community that interfaces provide all of the good features of *multiple inheritance* (the ability of a class to have more than one immediate parent class) with little of the complexity. We suspect that these claims are mostly made by people without significant experience with multiple inheritance. It is absolutely true that interfaces are a convenient and useful construct, and you'll see them used in several places in the rest of the book. However, if you come from a language that supports multiple inheritance (e.g., C++, Eiffel, Lisp/CLOS, among others), you will find that they do not provide everything that you are accustomed to in other languages.

Listing 7.23 `MeasureTest.java`

```
/** Test of MeasureUtil. Note that we could change the
 *  actual classes of elements in the measurables array (as
 *  long as they implemented Measurable) without changing
 *  the rest of the code.
 */

public class MeasureTest {
  public static void main(String[] args) {
    Measurable[] measurables =
      { new Rectangle(0, 0, 5.0, 10.0),
        new Rectangle(0, 0, 4.0, 9.0),
        new Circle(0, 0, 4.0),
        new Circle(0, 0, 5.0) };
    System.out.print("Areas:");
    for(int i=0; i<measurables.length; i++)
      System.out.print(" " + measurables[i].getArea());
    System.out.println();
    System.out.println("Larger of 1st, 3rd: " +
       MeasureUtil.maxArea(measurables[0],
                           measurables[2]) +
                           "\nTotal area: " +
                           MeasureUtil.totalArea(measurables));
  }
}
```

From the viewpoint of someone who will be *using* a class (e.g., the person writing `MeasureUtil`), interfaces are better; they let you treat classes in different hierarchies as though they were part of a common class, but without all the confusion multiple inheritance can cause. However, from the viewpoint of the person *writing* the classes themselves (e.g., the person writing the `Shape` hierarchy), they may save little. In particular, interfaces do not let you inherit the implementation of methods, although such inheritance can sometimes be a timesaver. For example, suppose that you have a variety of custom buttons, windows, and textfields, each of which has a private `name` field used for debugging, and each of which should have a `debug` method that prints out the location, parent window, and name of the graphical component. In Java, you would have to repeat this code in each subclass, whereas multiple inheritance would let you place it in a single parent class (`Debuggable`) and have each graphical object simply extend this class.

The use of multiple inheritance to provide characteristics shared by classes at varying locations in the class hierarchy is sometimes known as a "mix in" style, and Java provides nothing equivalent. Whether the simplicity benefit gained by leaving out this capability is worth the cost in the relatively small number of cases where mix-ins would have been valuable is open to debate.

7.10 Packages, Classpath, and JAR Archives

Java lets you group class libraries into separate modules or "packages" to avoid naming conflicts and to simplify their handling. For instance, in a real project, the `Ship` and `Shape` class hierarchies would have been easier to create and use if placed in separate packages. To create a package called `packagename`, first make a subdirectory with the same name and place all source files there. Each of these files should contain

```
package packagename;
```

as the first noncomment line. Files that lack a package declaration are automatically placed in an unnamed package. Files in the main directory that want to use the package should include

```
import packagename.ClassName;
```

or

```
import packagename.*;
```

before the class definitions (but after the `package` declaration, if any). This statement tells the compiler that all the specified classes ("`*`" means "all") should be available for use if they are needed. Otherwise, the compiler only looks for classes it needs in the current directory or, as we will see shortly, in directories specified through the `CLASSPATH` variable.

Package names can contain dots ("."); these correspond to subdirectories. For example, assume that your application is being developed on Windows 2000 or NT in `C:\Java\classes`. Now suppose that the classes shown in Listings 7.24 through 7.27 are created. They include the following:

- `package1` package (in `C:\Java\classes\package1`) containing `Class1`
- `package2` package (in `C:\Java\classes\package2`) containing `Class2`
- `package2.package3` package (in `C:\Java\classes\package2\package3`) containing `Class3`
- `package4` package (in `C:\Java\classes\package4`) containing `Class1`

Note the name conflict between `package1` and `package4`: they both contain a class named `Class1`. If a single program needs to use both, it simply omits the `import` statement for one of them and uses `packagename.Class1` instead. This is known as the "fully qualified classname" and can be used in lieu of `import` any time.

For instance, an applet could explicitly extend `java.applet.Applet` and not import `java.applet.Applet` or `java.applet.*`.

Also notice that the `printInfo` methods of the test classes are declared `static`. You can invoke static methods by using the name of the class (like `Math.cos`) without creating an object instance.

Listing 7.24 `C:\Java\classes\package1\Class1.java`

```
package package1;

public class Class1 {
  public static void printInfo() {
    System.out.println("This is Class1 in package1.");
  }
}
```

Listing 7.25 `C:\Java\classes\package2\Class2.java`

```
package package2;

public class Class2 {
  public static void printInfo() {
    System.out.println("This is Class2 in package2.");
  }
}
```

Listing 7.26 `C:\Java\classes\package2\package3\Class3.java`

```
package package2.package3;

public class Class3 {
  public static void printInfo() {
    System.out.println("This is Class3 in " +
                       "package2.package3.");
  }
}
```

Listing 7.27 `C:\Java\classes\package4\Class1.java`

```
package package4;

public class Class1 {
  public static void printInfo() {
    System.out.println("This is Class1 in package4.");
  }
}
```

Now, let's make a test program that uses these classes (Listing 7.28). This file will be placed back in the root directory, not in the package-specific subclasses.

Listing 7.28 `C:\Java\classes\PackageExample.java`

```
import package1.*;
import package2.Class2;
import package2.package3.*;

public class PackageExample {
  public static void main(String[] args) {
    Class1.printInfo();
    Class2.printInfo();
    Class3.printInfo();
    package4.Class1.printInfo();
  }
}
```

Compiling and Running:

```
javac PackageExample.java
java PackageExample
```

The first line above compiles all four other classes automatically.

Output:

```
    This is Class1 in package1.
    This is Class2 in package2.
    This is Class3 in package2.package3.
    This is Class1 in package4.
```

In the above example, compiling `PackageExample.java` automatically compiled the other source files located in the four packages, since `PackageExample` uses these classes directly (they are automatically compiled if out of date). If you would like to compile the four source files individually, you would need to specify the full subdirectory path in the command line, as in

```
javac package1/Class1.java
javac package2/Class2.java
javac package2/package3/Class3.java
javac package4/Class1.java
```

In general, when compiling files in packages you have two choices:

1. Compile the files from the directory just above the package directory (the package directory is below the working directory). For example, you would compile the file `C:\Java\classes\package2\Class2.java` located in the `package2` directory from `C:\Java\classes\`, specifying `javac package2/Class2.java`.

2. Include the root package directory in the `CLASSPATH` and compile the file from the same directory in which the source is located. For this example, you would include `C:\Java\classes` on your `CLASSPATH`.

We describe how to set the `CLASSPATH` in the next section.

The CLASSPATH

Up to now, we've been acting as though you must have a single root directory for all your Java files. This approach would be inconvenient if you develop a large number of applications. Rather than just looking in the current directory, Java lets you supply a list of directories in which it should look for classes. This mechanism complements packages; it doesn't replace them. In particular, the directory list should contain the *roots* of package hierarchies, not the subdirectories corresponding to the individual packages. The classpath defines the starting points for the search; the `import` statements specify which subdirectories should be examined. Most Java systems also allow entries in the classpath to be compressed JAR files; this is a convenient mechanism for moving large package hierarchies around.

You can specify the classpath in two ways. The first is to supply a `-classpath` argument to `javac` and `java`. The second, and more common, approach is to use the `CLASSPATH` environment variable. If this variable is not set, Java looks for class files and package subdirectories relative to the current directory. In many cases, looking in subdirectories of the working directory is usually what you want, so if you set the `CLASSPATH` variable, be sure to include "." if you want the current directory to be examined. Directories are separated by semicolons or colons, depending on the operating system. For instance, on Unix (`csh`, `tcsh`), you could set the environment variables through:

```
setenv CLASSPATH .:~/classes:/home/mcnealy/classes
```

On Windows 98/2000/NT, you could set the environment variables through

```
set CLASSPATH=.;C:\BillGates\classes;D:\Java\classes
```

MacOS doesn't have the concept of environment variables, but you can set the classpath on an application-by-application basis by changing the `'STR '(0)` resource. Fortunately, most Mac-based Java IDEs have easier ways of doing this. It is not necessary to include the location of the standard Java classes in the `CLASSPATH`; it is included automatically. Furthermore, `java.lang.*` is automatically imported, so there is no need to do this yourself.

Many browsers, including Netscape and appletviewer, look for classes in the classpath before trying to load them from the network. These classes are then granted special privileges. If someone knew your classpath and could look at the classes in it (e.g., on a multiuser system), that person could make an applet that uses these classes to perform privileged or destructive operations from your account. to avoid this situation, you want to make sure `CLASSPATH` is not set when you start the browser.

Core Security Warning

Unset `CLASSPATH` before starting your Web browser.

Using JAR Files

The `CLASSPATH` is a nice, convenient place to specify a list of directories in which to look for class files. However, if you are using a lot of large package hierarchies, eventually your `CLASSPATH` will exceed the number of characters allowed for your environment variable. To solve this problem, Java also lets you place classes in Java ARchive (JAR) files and include the JAR files in your classpath. Included with the JDK is a `jar` tool that compresses multiple class files into a single JAR file, using a compression algorithm based on the common ZIP format.

The most common command options for the `jar` tool are summarized below.

c This option specifies creation of a new JAR file.

f The `f` option specifies which file to process or the target JAR file.

m, M The first option, lowercase `m`, means that a manifest file is provided for advanced capabilities (signed JAR files, executable JAR files). If both the `m` and `f` option are used, then the order of the manifest and JAR file must be the same order of the command options. The second option, uppercase `M`, specifies that a manifest file not be created. By default, a manifest is included in the JAR file; you can then examine the manifest by decompressing the JAR file. For additional information of manifests, see http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html.

t This option lists the contents of the JAR file.

u This option updates a JAR file, for example, adding additional files or changing the manifest.

v This option generates verbose output.

x *file* This option decompresses the *file* located in the JAR file. If no file is specified, then all files in the achieve are decompressed.

0 This option (numerical `0`) creates an uncompressed JAR file.

As an example, to place the `ClassX.class` files in the previous example in a single JAR file, you would use

```
C:\Java\classes>jar cfv example.jar package1 package2 package4
```

Output

```
added manifest
adding: package1/(in = 0) (out= 0)(stored 0%)
adding: package1/Class1.class(in = 468)
        (out= 326)(deflated 30%)
adding: package2/(in = 0) (out= 0)(stored 0%)
adding: package2/Class2.class(in = 468)
```

```
        (out= 327)(deflated 30%)
adding: package2/package3/(in = 0) (out= 0)
        (stored 0%)
adding: package2/package3/Class3.class(in = 486)
        (out=338)(deflated 30%)
adding: package4/(in = 0) (out= 0)(stored 0%)
adding: package4/Class1.class(in = 431)
        (out= 298)(deflated 30%)
```

Here, we assumed that the package directories contained only the class files; otherwise, all files, including source files, would be compressed into `example.jar`. To avoid adding all directory files, you would need to individually specify the `class` files on the command line. If necessary, you can place a list of `jar` commands in a separate file and then refer to the file on the command line through `@filename`.

Once you've created a `jar` file, you can add the file to your classpath; or if your classpath is still too long, you can place the file in the `/root/jdk1.3/ jre/lib/ext/` directory. Java automatically examines the JAR files located in this directory for class files as if they were listed in your classpath.

Core Note

If your classpath is too long, you can place your JAR files in the `/root/jdk1.3/jre/lib/ext/` directory for automatic inclusion by `javac` and `java`.

7.11 Modifiers in Declarations

We've used a variety of class, method, and instance variable modifiers in this chapter: `public`, `private`, `protected`, and `static`. These were relatively obvious from context or from the brief explanation provided when they were used, but it is worth reviewing all the possible modifiers and explicitly stating their purpose.

Visibility Modifiers

These modifiers designate the other classes that are allowed to see the data inside an object you write. Table 7.1 summarizes them, with more details given below. If you are not familiar with object-oriented programming already, just look at `public` and `private` and forget the other options for now. For people with a C++ background, consider classes within the same package to be friendly to each other.

Table 7.1. Summary of Visibility Modifiers

Variables or Methods with This Modifier Can Be Accessed by Methods in:	***Variable or Method Modifier***			
	`public`	`private`	`protected`	***no modifier (default)***
Same Class	Y	Y	Y	Y
Classes in Same Package	Y	N	Y	Y
Subclasses	Y	N	Y	N
Classes in Different Packages	Y	N	N	N

public This visibility modifier indicates that the variable or method can be accessed by anyone who can access an instance of the class. You normally use this modifier for the functionality you are deliberately providing to the outside world, and you should generally document it with `javadoc`.

Many people feel that instance variables should *never* be public and should be accessed *only* by methods. See Section 7.6 for some of the reasons. This is the approach used in the `Ship` example (Listing 7.5) and a convention necessary for JavaBeans, Java's component architecture. However, some practitioners believe that "never" is too strong and that public fields are acceptable for small classes that act simply as records (C-style structs). They think that for such classes, if the get and set methods only set and read the variables without any modification, side effects, or error checking, then the methods are an unnecessary level of abstraction and the variables should be public instead. For instance, the `java.awt.Point` class contains public `x` and `y` variables, since the whole idea of a `Point` is to wrap up two integers into a single object. At the very least, we recommend that you avoid public instance variables for all complex classes.

Core Approach

Avoid public instance variables.

A *class* can also have the designation `public`, which means that any other class can load and use the class definition. The name of a public class must match the filename.

private This visibility modifier specifies that the variable or method can only be accessed by methods within the same class. This modifier is what you normally use for internal data and methods that are needed to implement your public interface but which users need not or should not (because of potential changes) access.

protected This visibility modifier specifies that the variable or method can be accessed by methods within the class and within classes in the same package. It is inherited by subclasses. Variables and methods that are protected can cross package boundaries through inheritance. This modifier is what you use for data that is normally considered private but might be of interest to people extending your class.

no modifier (default) Omitting a visibility modifier specifies that the variable or method can be accessed by methods within the class and within classes in the same package but is not inherited by subclasses. Variables and methods with default visibility cannot cross package boundaries through inheritance. Although no modifier is the default, the other modifiers represent more common intentions.

Other Modifiers

static This modifier indicates that the variable or method is shared by the entire class. Variables or methods that are `static` can be accessed through the classname instead of just by an instance. So, if a class `Foo` had a static variable `bar` and there were two instances, `foo1` and `foo2`, `bar` could be accessed by `foo1.bar`, `foo2.bar`, or `Foo.bar`. All three would access the same, shared, data. In a similar example of `Math.cos`, `cos` is a static method of the class `Math`.

Static methods can only refer to static variables or other static methods unless they create an instance. For example, in Listing 7.29, code in `main` can refer directly to `staticMethod` but requires an instance of the class to refer to `regularMethod`.

Listing 7.29 `Statics.java`

```
public class Statics {
  public static void main(String[] args) {
    staticMethod();
    Statics s1 = new Statics();
    s1.regularMethod();
  }

  public static void staticMethod() {
    System.out.println("This is a static method.");
  }
```

```
    public void regularMethod() {
      System.out.println("This is a regular method.");
    }
  }
```

final For a class, the `final` modifier indicates that it cannot be subclassed. This declaration may let the compiler optimize to method calls on variables of this type. For a variable or method, `final` indicates that it cannot be changed at runtime or overridden in subclasses. Think of `final` as the final representation (constant).

abstract This declaration can apply to classes or methods and indicates that the class cannot be directly instantiated. See Section 7.9 (Interfaces and Abstract Classes) for some examples.

synchronized The `synchronized` declaration is used to set locks for methods in multithreaded programming. Only one thread can access a synchronized method at any given time. For more details, see Chapter 16 (Concurrent Programming with Java Threads).

volatile For multithreaded efficiency, Java permits methods to keep local copies of instance variables, reconciling changes only at lock and unlock points. For some data types (e.g., `long`), multithreaded code that does not use locking risks having one thread partially update a field before another accesses it. The `volatile` declaration prevents this situation.

transient Variables can be marked `transient` to stipulate that they should not be saved by the object serialization system when writing an object to disk or network.

native This modifier signifies that a method is implemented using C or C++ code that is linked to the Java image.

7.12 Summary

The Java programming language is pervasively and consistently object oriented. You cannot go anywhere in Java without a good grasp of how to use objects. This chapter summarized how to create objects, give them state (instance variables), and assign them behavior (methods). Using inheritance, you can build hierarchies of classes without repeating code that is shared by subclasses. Although Java lacks multiple inheritance, interfaces are a convenient mechanism for letting one method handle objects from different hierarchies in a uniform manner. Class hierarchies that are intended to be reused should be documented with `javadoc` and can be organized into packages for convenience.

Getting comfortable with objects takes a bit of a conceptual leap and may take awhile if you've never seen objects before. Chapter 8 requires no such leap; it is a laundry list of basic constructs supported by Java. It should be quick going. In fact, if you know C or C++, you can just skim through, looking for the differences.

CONTENTS