

Table of Contents

Object-Oriented Programming in Java.....	1
Class Definition Syntax.....	2
Fields and Methods.....	3
Creating and Initializing Objects.....	11
Destroying and Finalizing Objects.....	15
Subclasses and Inheritance.....	19
Data Hiding and Encapsulation.....	29
Abstract Classes and Methods.....	35
Important Methods of java.lang.Object.....	37
Interfaces.....	41
Nested Types.....	47
Modifier Summary.....	64
C++ Features Not Found in Java.....	65

Chapter 3. Object-Oriented Programming in Java

Now that we've covered fundamental Java syntax, we are ready to begin object-oriented programming in Java. All Java programs use objects, and the type of an object is defined by its *class* or *interface*. Every Java program is defined as a class, and nontrivial programs usually include a number of classes and interface definitions. This chapter explains how to define new classes and interfaces and how to do object-oriented programming with them.^[1]

[1] If you do not have object-oriented (OO) programming background, don't worry; this chapter does not assume any prior experience. If you do have experience with OO programming, however, be careful. The term "object-oriented" has different meanings in different languages. Don't assume that Java works the same way as your favorite OO language. This is particularly true for C++ programmers. Although Java and C++ borrow much syntax from C, the similarities between the two languages do not go far beyond the level of syntax. Don't let your experience with C++ lull you into a false familiarity with Java.

This is a relatively long and detailed chapter, so we begin with an overview and some definitions. A *class* is a collection of fields that hold values and methods that operate on those values. Classes are the most fundamental structural element of all Java programs. You cannot write Java code without defining a class. All Java statements appear within methods, and all methods are implemented within classes.

A class defines a new reference type, such as the `Point` type defined in [Chapter 2](#). An *object* is an *instance* of a class. The `Point` class defines a type that is the set of all possible two-dimensional points. A `Point` object is a value of that type: it represents a single two-dimensional point.

Objects are usually created by *instantiating* a class with the `new` keyword and a constructor invocation, as shown here:

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

Constructors are covered in [Section 3.3](#) later in this chapter.

A class definition consists of a *signature* and a *body*. The class signature defines the name of the class and may also specify other important information. The body of a class is a set of *members* enclosed in curly braces. The members of a class may include fields and methods, constructors and initializers, and nested types.

Members can be *static* or nonstatic. A static member belongs to the class itself while a nonstatic member is associated with the instances of a class (see [Section 3.2](#) later in this chapter).

The signature of a class may declare that the class *extends* another class. The extended class is known as the *superclass* and the extension is known as the *subclass*. A subclass *inherits* the members of its superclass and may declare new members or *override* inherited methods with new implementations.

The signature of a class may also declare that the class *implements* one or more interfaces. An *interface* is a reference type that defines method signatures but does not include method bodies to implement the methods. A class that implements an interface is required to provide bodies for the interface's methods. Instances of such a class are also instances of the interface type that it implements.

The members of a class may have *access modifiers* `public`, `protected`, or `private`, which specify their visibility and accessibility to clients and to subclasses. This allows classes to hide members that are not part of their public API. When applied to fields, this ability to hide members enables an object-oriented design technique known as *data encapsulation*.

Classes and interfaces are the most important of the five fundamental reference types defined by Java. Arrays, enumerated types (or "enums") and annotation types are the other three. Arrays are covered in [Chapter 2](#). Enumerated types and annotation types were introduced in Java 5.0 (see [Chapter 4](#)). Enums are a specialized kind of class and annotation types are a specialized kind of interface.

3.1. Class Definition Syntax

At its simplest level, a class definition consists of the keyword `class` followed by the name of the class and a set of class members within curly braces. The `class` keyword may be preceded by modifier keywords and annotations (see [Chapter 4](#)). If the class extends another class, the class name is followed by the `extends` keyword and the name of the class being extended. If the class implements one or more interfaces then the class name or the `extends` clause is followed by the `implements` keyword and a comma-separated list of interface names. For example:

```
public class Integer extends Number implements Serializable, Comparable {  
    // class members go here  
}
```

Generic class declarations include additional syntax that is covered in [Chapter 4](#).

Class declarations may include zero or more of the following modifiers:

`public`

A `public` class is visible to classes defined outside of its package. See [Section 3.6](#) later in this chapter.

`abstract`

An `abstract` class is one whose implementation is incomplete and cannot be instantiated. Any class with one or more `abstract` methods must be declared `abstract`.

`final`

The `final` modifier specifies that the class may not be extended. Declaring a class `final` may enable the Java VM to optimize its methods.

`strictfp`

If a class is declared `strictfp`, all its methods behave as if they were declared `strictfp`. This rarely used modifier is discussed in [Section 2.6](#) in Chapter 2.

A class cannot be both `abstract` and `final`. By convention, if a class has more than one modifier, they appear in the order shown.

3.2. Fields and Methods

A class can be viewed as a collection of data and code to operate on that data. The data is stored in fields, and the code is organized into methods. This section covers fields and methods, the two most important kinds of class members. Fields and methods come in two distinct types: class members (also known as static members) are associated with the class itself, while instance members are associated with individual instances of the class (i.e., with objects). This gives us four kinds of members:

- Class fields
- Class methods
- Instance fields
- Instance methods

The simple class definition for the class `Circle`, shown in [Example 3-1](#), contains all four types of members.

Example 3-1. A simple class and its members

```

public class Circle {
    // A class field
    public static final double PI= 3.14159;    // A useful constant

    // A class method: just compute a value based on the arguments
    public static double radiansToDegrees(double rads) {
        return rads * 180 / PI;
    }

    // An instance field
    public double r;    // The radius of the circle

    // Two instance methods: they operate on the instance fields of an object
    public double area() {    // Compute the area of the circle
        return PI * r * r;
    }
    public double circumference() {    // Compute the circumference of the circle
        return 2 * PI * r;
    }
}

```

The following sections explain all four kinds of members. First, however, we cover field declaration syntax. (Method declaration syntax is covered in [Section 2.6](#) later in this chapter.)

3.2.1. Field Declaration Syntax

Field declaration syntax is much like the syntax for declaring local variables (see [Chapter 2](#)) except that field definitions may also include modifiers. The simplest field declaration consists of the field type followed by the field name. The type may be preceded by zero or more modifier keywords or annotations (see [Chapter 4](#)), and the name may be followed by an equals sign and initializer expression that provides the initial value of the field. If two or more fields share the same type and modifiers, the type may be followed by a comma-separated list of field names and initializers. Here are some valid field declarations:

```

int x = 1;
private String name;
public static final DAYS_PER_WEEK = 7;
String[] daynames = new String[DAYS_PER_WEEK];
private int a = 17, b = 37, c = 53;

```

Field modifiers are comprised of zero or more of the following keywords:

```
public ,protected,private
```

These access modifiers specify whether and where a field can be used outside of the class that defines it. These important modifiers are covered in [Section 3.6](#) later in this chapter. No more than one of these access modifiers may appear in any field declaration.

`static`

If present, this modifier specifies that the field is associated with the defining class itself rather than with each instance of the class.

`final`

This modifier specifies that once the field has been initialized, its value may never be changed. Fields that are both `static` and `final` are compile-time constants that the compiler can inline. `final` fields can also be used to create classes whose instances are immutable.

`transient`

This modifier specifies that a field is not part of the persistent state of an object and that it need not be serialized along with the rest of the object. Serialization is covered in [Chapter 5](#).

`volatile`

Roughly speaking, a `volatile` field is like a `synchronized` method: safe for concurrent use by two or more threads. More accurately, `volatile` says that the value of a field must always be read from and flushed to main memory, and that it may not be cached by a thread (in a register or CPU cache).

3.2.2. Class Fields

A *class field* is associated with the class in which it is defined rather than with an instance of the class. The following line declares a class field:

```
public static final double PI = 3.14159;
```

This line declares a field of type `double` named `PI` and assigns it a value of 3.14159. As you can see, a field declaration looks quite a bit like a local variable declaration. The difference, of course, is that variables are defined within methods while fields are members of classes.

The `static` modifier says that the field is a class field. Class fields are sometimes called static fields because of this `static` modifier. The `final` modifier says that the value of the field does not change. Since the field `PI` represents a constant, we declare it `final` so that it cannot be changed. It is a convention in Java (and many other languages) that constants are named with capital letters, which is why our field is named `PI`, not `pi`.

Defining constants like this is a common use for class fields, meaning that the `static` and `final` modifiers are often used together. Not all class fields are constants, however. In other words, a field can be declared `static` without being declared `final`. Finally, the `public` modifier says that anyone can use the field. This is a visibility modifier, and we'll discuss it and related modifiers in more detail later in this chapter.

The key point to understand about a static field is that there is only a single copy of it. This field is associated with the class itself, not with instances of the class. If you look at the various methods of the `Circle` class, you'll see that they use this field. From inside the `Circle` class, the field can be referred to simply as `PI`. Outside the class, however, both class and field names are required to uniquely specify the field. Methods that are not part of `Circle` access this field as `Circle.PI`.

A `public` class field is essentially a global variable. The names of class fields are qualified by the unique names of the classes that contain them, however. Thus, Java does not suffer from the name collisions that can affect other languages when different modules of code define global variables with the same name.

3.2.3. Class Methods

As with class fields, *class methods* are declared with the `static` modifier:

```
public static double radiansToDegrees(double rads) { return rads * 180 / PI; }
```

This line declares a class method named `radiansToDegrees()`. It has a single parameter of type `double` and returns a `double` value. The body of the method is quite short; it performs a simple computation and returns the result.

Like class fields, class methods are associated with a class, rather than with an object. When invoking a class method from code that exists outside the class, you must specify both the name of the class and the method. For example:

```
// How many degrees is 2.0 radians?  
double d = Circle.radiansToDegrees(2.0);
```

If you want to invoke a class method from inside the class in which it is defined, you don't have to specify the class name. However, it is often good style to specify the class name anyway, to make it clear that a class method is being invoked.

Note that the body of our `Circle.radiansToDegrees()` method uses the class field `PI`. A class method can use any class fields and class methods of its own class (or of any other class). But it cannot use any instance fields or instance methods because class methods are not associated with an instance of the class. In other words, although the `radiansToDegrees()` method is defined in the `Circle` class, it does not use any

`Circle` objects. The instance fields and instance methods of the class are associated with `Circle` objects, not with the class itself. Since a class method is not associated with an instance of its class, it cannot use any instance methods or fields.

As we discussed earlier, a class field is essentially a global variable. In a similar way, a class method is a global method, or global function. Although `radiansToDegrees()` does not operate on `Circle` objects, it is defined within the `Circle` class because it is a utility method that is sometimes useful when working with circles. In many nonobject-oriented programming languages, all methods, or functions, are global. You can write complex Java programs using only class methods. This is not object-oriented programming, however, and does not take advantage of the power of the Java language. To do true object-oriented programming, we need to add instance fields and instance methods to our repertoire.

3.2.4. Instance Fields

Any field declared without the `static` modifier is an *instance field*:

```
public double r;    // The radius of the circle
```

Instance fields are associated with instances of the class, rather than with the class itself. Thus, every `Circle` object we create has its own copy of the `double` field `r`. In our example, `r` represents the radius of a circle. Thus, each `Circle` object can have a radius independent of all other `Circle` objects.

Inside a class definition, instance fields are referred to by name alone. You can see an example of this if you look at the method body of the `circumference()` instance method. In code outside the class, the name of an instance method must be prefixed with a reference to the object that contains it. For example, if the variable `c` holds a reference to a `Circle` object, we use the expression `c.r` to refer to the radius of that circle:

```
Circle c = new Circle(); // Create a Circle object; store a reference in c
c.r = 2.0;               // Assign a value to its instance field r
Circle d = new Circle(); // Create a different Circle object
d.r = c.r * 2;           // Make this one twice as big
```

Instance fields are key to object-oriented programming. Instance fields hold the state of an object; the values of those fields make one object distinct from another.

3.2.5. Instance Methods

Any method not declared with the `static` keyword is an instance method. An *instance method* operates on an instance of a class (an object) instead of operating on the class itself. It is with instance methods that object-oriented programming starts to get interesting. The `Circle` class defined in [Example 3-1](#) contains two instance methods, `area()` and

`circumference()`, that compute and return the area and circumference of the circle represented by a given `Circle` object.

To use an instance method from outside the class in which it is defined, we must prefix it with a reference to the instance that is to be operated on. For example:

```
Circle c = new Circle(); // Create a Circle object; store in variable c
c.r = 2.0;               // Set an instance field of the object
double a = c.area();     // Invoke an instance method of the object
```

If you're new to object-oriented programming, that last line of code may look a little strange. We do not write:

```
a = area(c);
```

Instead, we write:

```
a = c.area();
```

This is why it is called object-oriented programming; the object is the focus here, not the function call. This small syntactic difference is perhaps the single most important feature of the object-oriented paradigm.

The point here is that we don't have to pass an argument to `c.area()`. The object we are operating on, `c`, is implicit in the syntax. Take a look at [Example 3-1](#) again. You'll notice the same thing in the signature of the `area()` method: it doesn't have a parameter. Now look at the body of the `area()` method: it uses the instance field `r`. Because the `area()` method is part of the same class that defines this instance field, the method can use the unqualified name `r`. It is understood that this refers to the radius of whatever `Circle` instance invokes the method.

Another important thing to notice about the bodies of the `area()` and `circumference()` methods is that they both use the class field `PI`. We saw earlier that class methods can use only class fields and class methods, not instance fields or methods. Instance methods are not restricted in this way: they can use any member of a class, whether it is declared `static` or not.

3.2.5.1. How instance methods work

Consider this line of code again:

```
a = c.area();
```

What's going on here? How can a method that has no parameters know what data to operate on? In fact, the `area()` method does have a parameter. All instance methods are implemented with an implicit parameter not shown in the method signature. The

implicit argument is named `this`; it holds a reference to the object through which the method is invoked. In our example, that object is a `Circle`.

The implicit `this` parameter is not shown in method signatures because it is usually not needed; whenever a Java method accesses the instance fields in its class, it is implicit that it is accessing fields in the object referred to by the `this` parameter. The same is true when an instance method invokes another instance method in the same class. I said earlier that to invoke an instance method you must prepend a reference to the object to be operated on. When an instance method is invoked within another instance method in the same class, however, you don't need to specify an object. In this case, it is implicit that the method is being invoked on the `this` object.

You can use the `this` keyword explicitly when you want to make it clear that a method is accessing its own fields and/or methods. For example, we can rewrite the `area()` method to use `this` explicitly to refer to instance fields:

```
public double area() { return Circle.PI * this.r * this.r; }
```

This code also uses the class name explicitly to refer to class field `PI`. In a method this simple, it is not necessary to be explicit. In more complicated cases, however, you may find that it increases the clarity of your code to use an explicit `this` where it is not strictly required.

In some cases, the `this` keyword is required, however. For example, when a method parameter or local variable in a method has the same name as one of the fields of the class, you must use `this` to refer to the field since the field name used alone refers to the method parameter or local variable. For example, we can add the following method to the `Circle` class:

```
public void setRadius(double r) {  
    this.r = r;           // Assign the argument (r) to the field (this.r)  
                           // Note that we cannot just say r = r  
}
```

Finally, note that while instance methods can use the `this` keyword, class methods cannot. This is because class methods are not associated with objects.

3.2.5.2. Instance methods or class methods?

Instance methods are one of the key features of object-oriented programming. That doesn't mean, however, that you should shun class methods. In many cases, it is perfectly reasonable to define class methods. When working with the `Circle` class, for example, you might find that you often want to compute the area of a circle with a given radius but don't want to bother creating a `Circle` object to represent that circle. In this case, a class method is more convenient:

```
public static double area(double r) { return PI * r * r; }
```

It is perfectly legal for a class to define more than one method with the same name, as long as the methods have different parameters. Since this version of the `area()` method is a class method, it does not have an implicit `this` parameter and must have a parameter that specifies the radius of the circle. This parameter keeps it distinct from the instance method of the same name.

As another example of the choice between instance methods and class methods, consider defining a method named `bigger()` that examines two `Circle` objects and returns whichever has the larger radius. We can write `bigger()` as an instance method as follows:

```
// Compare the implicit "this" circle to the "that" circle passed
// explicitly as an argument and return the bigger one.
public Circle bigger(Circle that) {
    if (this.r > that.r) return this;
    else return that;
}
```

We can also implement `bigger()` as a class method as follows:

```
// Compare circle a to circle b and return the one with the larger radius
public static Circle bigger(Circle a, Circle b) {
    if (a.r > b.r) return a;
    else return b;
}
```

Given two `Circle` objects, `x` and `y`, we can use either the instance method or the class method to determine which is bigger. The invocation syntax differs significantly for the two methods, however:

```
Circle biggest = x.bigger(y);           // Instance method: also y.bigger(x)
Circle biggest = Circle.bigger(x, y);    // Static method
```

Both methods work well, and, from an object-oriented design standpoint, neither of these methods is "more correct" than the other. The instance method is more formally object-oriented, but its invocation syntax suffers from a kind of asymmetry. In a case like this, the choice between an instance method and a class method is simply a design decision. Depending on the circumstances, one or the other will likely be the more natural choice.

3.2.6. Case Study: `System.out.println()`

Throughout this book, we've seen a method named `System.out.println()` used to display output to the terminal window or console. We've never explained why this method has such a long, awkward name or what those two periods are doing in it. Now that you understand class and instance fields and class and instance methods, it is easier to understand what is going on: `System` is a class. It has a class field named `out`. The field

`System.out` refers to an object. The object `System.out` has an instance method named `println()`. If you want to explore this in more detail, you can look up the `java.lang.System` class in the reference section. The class synopsis there tells you that the field `out` is of type `java.io.PrintStream`, and you can look up that class to find out about the `println()` method.

3.3. Creating and Initializing Objects

Now that we've covered fields and methods, we move on to other important members of a class. Constructors and initializers are class members whose job is to initialize the fields of a class.

Take another look at how we've been creating `Circle` objects:

```
Circle c = new Circle();
```

What are those parentheses doing there? They make it look like we're calling a method. In fact, that is exactly what we're doing. Every class in Java has at least one *constructor*, which is a method that has the same name as the class and whose purpose is to perform any necessary initialization for a new object. Since we didn't explicitly define a constructor for our `Circle` class in [Example 3-1](#), Java gave us a default constructor that takes no arguments and performs no special initialization.

Here's how a constructor works. The `new` operator creates a new, but uninitialized, instance of the class. The constructor method is then called, with the new object passed implicitly (a `this` reference, as we saw earlier) as well as whatever arguments that are specified between parentheses passed explicitly. The constructor can use these arguments to do whatever initialization is necessary.

3.3.1. Defining a Constructor

There is some obvious initialization we could do for our circle objects, so let's define a constructor. [Example 3-2](#) shows a new definition for `Circle` that contains a constructor that lets us specify the radius of a new `Circle` object. The constructor also uses the `this` reference to distinguish between a method parameter and an instance field of the same name.

Example 3-2. A constructor for the Circle class

```

public class Circle {
    public static final double PI = 3.14159; // A constant
    public double r; // An instance field that holds the radius of the circle

    // The constructor method: initialize the radius field
    public Circle(double r) { this.r = r; }

    // The instance methods: compute values based on the radius
    public double circumference() { return 2 * PI * r; }
    public double area() { return PI * r*r; }
}

```

When we relied on the default constructor supplied by the compiler, we had to write code like this to initialize the radius explicitly:

```

Circle c = new Circle();
c.r = 0.25;

```

With this new constructor, the initialization becomes part of the object creation step:

```

Circle c = new Circle(0.25);

```

Here are some important notes about naming, declaring, and writing constructors:

- The constructor name is always the same as the class name.
- Unlike all other methods, a constructor is declared without a return type, not even `void`.
- The body of a constructor should initialize the `this` object.
- A constructor may not return `this` or any other value. A constructor may include a `return` statement, but only one that does not include a return value.

3.3.2. Defining Multiple Constructors

Sometimes you want to initialize an object in a number of different ways, depending on what is most convenient in a particular circumstance. For example, we might want to initialize the radius of a circle to a specified value or a reasonable default value. Since our `Circle` class has only a single instance field, we can't initialize it too many ways, of course. But in more complex classes, it is often convenient to define a variety of constructors. Here's how we can define two constructors for `Circle`:

```

public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }

```

It is perfectly legal to define multiple constructors for a class, as long as each constructor has a different parameter list. The compiler determines which constructor you wish to use based on the number and type of arguments you supply. This is simply an example of method overloading, as we discussed in [Chapter 2](#).

3.3.3. Invoking One Constructor from Another

A specialized use of the `this` keyword arises when a class has multiple constructors; it can be used from a constructor to invoke one of the other constructors of the same class. In other words, we can rewrite the two previous `Circle` constructors as follows:

```
// This is the basic constructor: initialize the radius
public Circle(double r) { this.r = r; }
// This constructor uses this() to invoke the constructor above
public Circle() { this(1.0); }
```

The `this ()` syntax is a method invocation that calls one of the other constructors of the class. The particular constructor that is invoked is determined by the number and type of arguments, of course. This is a useful technique when a number of constructors share a significant amount of initialization code, as it avoids repetition of that code. This would be a more impressive example, of course, if the one-parameter version of the `Circle ()` constructor did more initialization than it does.

There is an important restriction on using `this ()`: it can appear only as the first statement in a constructor. It may, of course, be followed by any additional initialization a particular version of the constructor needs to do. The reason for this restriction involves the automatic invocation of superclass constructor methods, which we'll explore later in this chapter.

3.3.4. Field Defaults and Initializers

Not every field of a class requires initialization. Unlike local variables, which have no default value and cannot be used until explicitly initialized, the fields of a class are automatically initialized to the default value `false`, `'\u0000'`, `0`, `0.0`, or `null`, depending on their type. These default values are guaranteed by Java and apply to both instance fields and class fields.

If the default field value is not appropriate for your field, you can explicitly provide a different initial value. For example:

```
public static final double PI = 3.14159;
public double r = 1.0;
```

Field declarations and local variable declarations have similar syntax, but there is an important difference in how their initializer expressions are handled. As described in [Chapter 2](#), a local variable declaration is a statement that appears within a Java method; the variable initialization is performed when the statement is executed. Field declarations, however, are not part of any method, so they cannot be executed as statements are. Instead, the Java compiler generates instance-field initialization code automatically and puts it in the constructor or constructors for the class. The initialization code is inserted into a

constructor in the order in which it appears in the source code, which means that a field initializer can use the initial values of any fields declared before it. Consider the following code excerpt, which shows a constructor and two instance fields of a hypothetical class:

```
public class TestClass {
    public int len = 10;
    public int[] table = new int[len];

    public TestClass() {
        for(int i = 0; i < len; i++) table[i] = i;
    }

    // The rest of the class is omitted...
}
```

In this case, the code generated for the constructor is actually equivalent to the following:

```
public TestClass() {
    len = 10;
    table = new int[len];
    for(int i = 0; i < len; i++) table[i] = i;
}
```

If a constructor begins with a `this()` call to another constructor, the field initialization code does not appear in the first constructor. Instead, the initialization is handled in the constructor invoked by the `this()` call.

So, if instance fields are initialized in constructor methods, where are class fields initialized? These fields are associated with the class, even if no instances of the class are ever created, so they need to be initialized even before a constructor is called. To support this, the Java compiler generates a class initialization method automatically for every class. Class fields are initialized in the body of this method, which is invoked exactly once before the class is first used (often when the class is first loaded by the Java VM.)^[2] As with instance field initialization, class field initialization expressions are inserted into the class initialization method in the order in which they appear in the source code. This means that the initialization expression for a class field can use the class fields declared before it. The class initialization method is an internal method that is hidden from Java programmers. In the class file, it bears the name `<clinit>`.

^[2] It is actually possible to write a class initializer for a class C that calls a method of another class that creates an instance of C. In this contrived recursive case, an instance of C is created before the class C is fully initialized. This situation is not common in everyday practice, however.

3.3.4.1. Initializer blocks

So far, we've seen that objects can be initialized through the initialization expressions for their fields and by arbitrary code in their constructor methods. A class has a class initialization method, which is like a constructor, but we cannot explicitly define the body of this method as we can for a constructor. Java does allow us to write arbitrary code for the initialization of class fields, however, with a construct known as a *static initializer*. A static initializer is simply the keyword `static` followed by a block of code in curly braces.

A static initializer can appear in a class definition anywhere a field or method definition can appear. For example, consider the following code that performs some nontrivial initialization for two class fields:

```
// We can draw the outline of a circle using trigonometric functions
// Trigonometry is slow, though, so we precompute a bunch of values
public class TrigCircle {
    // Here are our static lookup tables and their own simple initializers
    private static final int NUMPTS = 500;
    private static double sines[] = new double[NUMPTS];
    private static double cosines[] = new double[NUMPTS];

    // Here's a static initializer that fills in the arrays
    static {
        double x = 0.0;
        double delta_x = (Circle.PI/2)/(NUMPTS-1);
        for(int i = 0, x = 0.0; i < NUMPTS; i++, x += delta_x) {
            sines[i] = Math.sin(x);
            cosines[i] = Math.cos(x);
        }
    }
    // The rest of the class is omitted...
}
```

A class can have any number of static initializers. The body of each initializer block is incorporated into the class initialization method, along with any static field initialization expressions. A static initializer is like a class method in that it cannot use the `this` keyword or any instance fields or instance methods of the class.

In Java 1.1 and later, classes are also allowed to have instance initializers. An instance initializer is like a static initializer, except that it initializes an object, not a class. A class can have any number of instance initializers, and they can appear anywhere a field or method definition can appear. The body of each instance initializer is inserted at the beginning of every constructor for the class, along with any field initialization expressions. An instance initializer looks just like a static initializer, except that it doesn't use the `static` keyword. In other words, an instance initializer is just a block of arbitrary Java code that appears within curly braces.

Instance initializers can initialize arrays or other fields that require complex initialization. They are sometimes useful because they locate the initialization code right next to the field, instead of separating into a constructor method. For example:

```
private static final int NUMPTS = 100;
private int[] data = new int[NUMPTS];
{ for(int i = 0; i < NUMPTS; i++) data[i] = i; }
```

In practice, however, this use of instance initializers is fairly rare. Instance initializers were introduced in Java 1.1 to support anonymous inner classes, which are not allowed to define constructors. (Anonymous inner classes are covered in [Section 3.10](#) later in this chapter.)

3.4. Destroying and Finalizing Objects

Now that we've seen how new objects are created and initialized in Java, we need to study the other end of the object life cycle and examine how objects are finalized and destroyed. *Finalization* is the opposite of initialization.

In Java, the memory occupied by an object is automatically reclaimed when the object is no longer needed. This is done through a process known as *garbage collection*. Garbage collection is a technique that has been around for years in languages such as Lisp. It takes some getting used to for programmers accustomed to such languages as C and C++, in which you must call the `free()` function or the `delete` operator to reclaim memory. The fact that you don't need to remember to destroy every object you create is one of the features that makes Java a pleasant language to work with. It is also one of the features that makes programs written in Java less prone to bugs than those written in languages that don't support automatic garbage collection.

3.4.1. Garbage Collection

The Java interpreter knows exactly what objects and arrays it has allocated. It can also figure out which local variables refer to which objects and arrays and which objects and arrays refer to which other objects and arrays. Thus, the interpreter is able to determine when an allocated object is no longer referred to by any other active object or variable. When the interpreter finds such an object, it knows it can safely reclaim the object's memory and does so. The garbage collector can also detect and destroy cycles of objects that refer to each other, but are not referenced by any other active objects. Any such cycles are also reclaimed.

Different VM implementations handle garbage collection in different ways. It is reasonable, however, to imagine the garbage collector running as a low-priority background thread, so it does most of its work when nothing else is going on, such as during idle time while waiting for user input. The only time the garbage collector must run while something high-priority is going on (i.e., the only time it actually slows down the system) is when available memory has become dangerously low. This doesn't happen very often because the low-priority thread cleans things up in the background.

3.4.2. Memory Leaks in Java

The fact that Java supports garbage collection dramatically reduces the incidence of a class of bugs known as *memory leaks*. A memory leak occurs when memory is allocated and never reclaimed. At first glance, it might seem that garbage collection prevents all memory leaks because it reclaims all unused objects. A memory leak can still occur in Java, however, if a valid (but unused) reference to an unused object is left hanging around. For example, when a method runs for a long time (or forever), the local variables in that method can

retain object references much longer than they are actually required. The following code illustrates:

```
public static void main(String args[]) {
    int big_array[] = new int[100000];

    // Do some computations with big_array and get a result.
    int result = compute(big_array);

    // We no longer need big_array. It will get garbage collected when there
    // are no more references to it. Since big_array is a local variable,
    // it refers to the array until this method returns. But this method
    // doesn't return. So we've got to explicitly get rid of the reference
    // ourselves, so the garbage collector knows it can reclaim the array.
    big_array = null;

    // Loop forever, handling the user's input
    for(;;) handle_input(result);
}
```

Memory leaks can also occur when you use a hash table or similar data structure to associate one object with another. Even when neither object is required anymore, the association remains in the hash table, preventing the objects from being reclaimed until the hash table itself is reclaimed. If the hash table has a substantially longer lifetime than the objects it holds, this can cause memory leaks.

The key to avoiding memory leaks is to set object references to `null` when they are no longer needed if the object that contains those references is going to continue to exist. One common source of leaks is in data structures in which an `Object` array is used to represent a collection of objects. It is common to use a separate `size` field to keep track of which elements of the array are currently valid. When removing an object from the collection, it is not sufficient to simply decrement this `size` field: you must also set the appropriate array element to `null` so that the obsolete object reference does not live on.

3.4.3. Object Finalization

A *finalizer* in Java is the opposite of a constructor. While a constructor method performs initialization for an object, a finalizer method can be used to perform cleanup or "finalization" for the object. Garbage collection automatically frees up the memory resources used by objects, but objects can hold other kinds of resources, such as open files and network connections. The garbage collector cannot free these resources for you, so you may occasionally want to write a finalizer method for any object that needs to perform such tasks as closing files, terminating network connections, deleting temporary files, and so on. This is particularly true for classes that use native methods: these classes may need a `native` finalizer to release native resources (including memory) that are not under the control of the Java garbage collector.

A finalizer is an instance method that takes no arguments and returns no value. There can be only one finalizer per class, and it must be named `finalize()`.^[3] A finalizer can throw

any kind of exception or error, but when a finalizer is automatically invoked by the garbage collector, any exception or error it throws is ignored and serves only to cause the finalizer method to return. Finalizer methods are typically declared `protected` (which we have not discussed yet) but can also be declared `public`. An example finalizer looks like this:

^[3] C++ programmers should note that although Java constructor methods are named like C++ constructors, Java finalization methods are not named like C++ destructor methods. As we will see, they do not behave quite like C++ destructor methods either.

```
protected void finalize() throws Throwable {
    // Invoke the finalizer of our superclass
    // We haven't discussed superclasses or this syntax yet
    super.finalize();

    // Delete a temporary file we were using
    // If the file doesn't exist or tempfile is null, this can throw
    // an exception, but that exception is ignored.
    tempfile.delete();
}
```

Here are some important points about finalizers:

- If an object has a finalizer, the finalizer method is invoked sometime after the object becomes unused (or unreachable), but before the garbage collector reclaims the object.
- Java makes no guarantees about when garbage collection will occur or in what order objects will be collected. Therefore, Java can make no guarantees about when (or even whether) a finalizer will be invoked, in what order finalizers will be invoked, or what thread will execute finalizers.
- The Java interpreter can exit without garbage collecting all outstanding objects, so some finalizers may never be invoked. In this case, resources such as network connections are closed and reclaimed by the operating system. Note, however, that if a finalizer that deletes a file does not run, that file will not be deleted by the operating system.
- To ensure that certain actions are taken before the VM exits, Java 1.1 provided the `Runtime` method `runFinalizersOnExit()`. Unfortunately, however, this method can cause deadlock and is inherently unsafe; it was deprecated in 1.2. In Java 1.3 and later, the `Runtime` method `addShutdownHook()` can safely execute arbitrary code before the Java interpreter exits.
- After a finalizer is invoked, objects are not freed right away. This is because a finalizer method can resurrect an object by storing the `this` pointer somewhere so that the object once again has references. Thus, after `finalize()` is called, the garbage collector must once again determine that the object is unreferenced before it can garbage-collect it. However, even if an object is resurrected, the finalizer method is never invoked more than once. Resurrecting an object is never a useful thing to do—just a strange quirk of object finalization.
- The `finalize()` method is an instance method, and finalizers act on instances. There is no equivalent mechanism for finalizing a class.

In practice, it is quite rare for an application-level class to require a `finalize()` method. Finalizer methods are more useful, however, when writing Java classes that interface to native platform code with `native` methods. In this case, the native implementation can allocate memory or other resources that are not under the control of the Java garbage collector and need to be reclaimed explicitly by a `native finalize()` method.

Furthermore, because of the uncertainty about when and whether a finalizer runs, it is best to avoid dependence on finalizers. For example, a class that includes a reference to a network socket should define a public `close()` method, which calls the `close()` method of the socket. This way, when the user of your class is done with it, she can call `close()` and be sure that the network connection is closed. You might, however, define a `finalize()` method as backup in case the user of your class forgets to call `close()` and allows an unclosed instance to be garbage-collected.

3.5. Subclasses and Inheritance

The `Circle` defined earlier is a simple class that distinguishes circle objects only by their radii. Suppose, instead, that we want to represent circles that have both a size and a position. For example, a circle of radius 1.0 centered at point 0,0 in the Cartesian plane is different from the circle of radius 1.0 centered at point 1,2. To do this, we need a new class, which we'll call `PlaneCircle`. We'd like to add the ability to represent the position of a circle without losing any of the existing functionality of the `Circle` class. This is done by defining `PlaneCircle` as a subclass of `Circle` so that `PlaneCircle` inherits the fields and methods of its superclass, `Circle`. The ability to add functionality to a class by subclassing, or extending, is central to the object-oriented programming paradigm.

3.5.1. Extending a Class

Example 3-3 shows how we can implement `PlaneCircle` as a subclass of the `Circle` class.

Example 3-3. Extending the Circle class

```

public class PlaneCircle extends Circle {
    // We automatically inherit the fields and methods of Circle,
    // so we only have to put the new stuff here.
    // New instance fields that store the center point of the circle
    public double cx, cy;

    // A new constructor method to initialize the new fields
    // It uses a special syntax to invoke the Circle() constructor
    public PlaneCircle(double r, double x, double y) {
        super(r);        // Invoke the constructor of the superclass, Circle()
        this.cx = x;      // Initialize the instance field cx
        this.cy = y;      // Initialize the instance field cy
    }

    // The area() and circumference() methods are inherited from Circle
    // A new instance method that checks whether a point is inside the circle
    // Note that it uses the inherited instance field r
    public boolean isInside(double x, double y) {
        double dx = x - cx, dy = y - cy;        // Distance from center
        double distance = Math.sqrt(dx*dx + dy*dy); // Pythagorean theorem
        return (distance < r);                  // Returns true or false
    }
}

```

Note the use of the keyword `extends` in the first line of [Example 3-3](#). This keyword tells Java that `PlaneCircle` extends, or subclasses, `Circle`, meaning that it inherits the fields and methods of that class.^[4] The definition of the `isInside()` method shows field inheritance; this method uses the field `r` (defined by the `Circle` class) as if it were defined right in `PlaneCircle` itself. `PlaneCircle` also inherits the methods of `Circle`. Thus, if we have a `PlaneCircle` object referenced by variable `pc`, we can say:

^[4] C++ programmers should note that `extends` is the Java equivalent of `:` in C++; both are used to indicate the superclass of a class.

```
double ratio = pc.circumference() / pc.area();
```

This works just as if the `area()` and `circumference()` methods were defined in `PlaneCircle` itself.

Another feature of subclassing is that every `PlaneCircle` object is also a perfectly legal `Circle` object. If `pc` refers to a `PlaneCircle` object, we can assign it to a `Circle` variable and forget all about its extra positioning capabilities:

```
PlaneCircle pc = new PlaneCircle(1.0, 0.0, 0.0); // Unit circle at the origin
Circle c = pc; // Assigned to a Circle variable without casting
```

This assignment of a `PlaneCircle` object to a `Circle` variable can be done without a cast. As we discussed in [Section 2.9.6](#) in Chapter 2 a widening conversion like this is always legal. The value held in the `Circle` variable `c` is still a valid `PlaneCircle` object, but the compiler cannot know this for sure, so it doesn't allow us to do the opposite (narrowing) conversion without a cast:

```
// Narrowing conversions require a cast (and a runtime check by the VM)
PlaneCircle pc2 = (PlaneCircle) c;
boolean origininside = ((PlaneCircle) c).isInside(0.0, 0.0);
```

3.5.1.1. Final classes

When a class is declared with the `final` modifier, it means that it cannot be extended or subclassed. `java.lang.String` is an example of a `final` class. Declaring a class `final` prevents unwanted extensions to the class: if you invoke a method on a `String` object, you know that the method is the one defined by the `String` class itself, even if the `String` is passed to you from some unknown outside source. Because `String` is `final`, no one can create a subclass of it and change the meaning or behavior of its methods.

Declaring a class `final` also allows the compiler to make certain optimizations when invoking the methods of a class. We'll explore this when we talk about method overriding later in this chapter.

3.5.2. Superclasses, Object, and the Class Hierarchy

In our example, `PlaneCircle` is a subclass from `Circle`. We can also say that `Circle` is the superclass of `PlaneCircle`. The superclass of a class is specified in its `extends` clause:

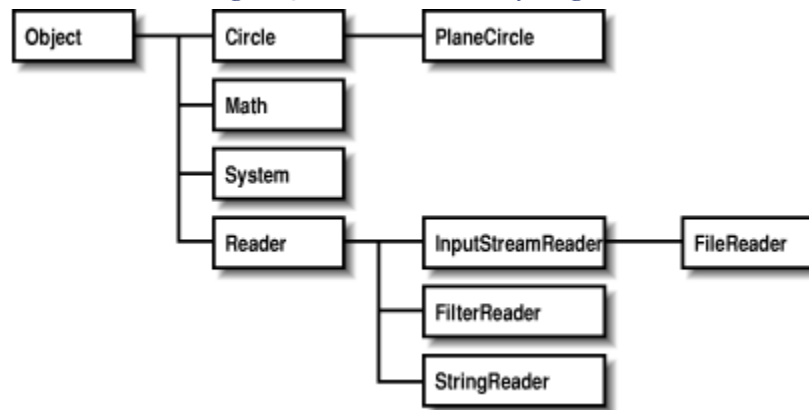
```
public class PlaneCircle extends Circle { ... }
```

Every class you define has a superclass. If you do not specify the superclass with an `extends` clause, the superclass is the class `java.lang.Object`. `Object` is a special class for a couple of reasons:

- It is the only class in Java that does not have a superclass.
- All Java classes inherit the methods of `Object`.

Because every class has a superclass, classes in Java form a class hierarchy, which can be represented as a tree with `Object` at its root. [Figure 3-1](#) shows a partial class hierarchy diagram that includes our `Circle` and `PlaneCircle` classes, as well as some of the standard classes from the Java API.

Figure 3-1. A class hierarchy diagram



3.5.3. Subclass Constructors

Look again at the `PlaneCircle()` constructor method of [Example 3-3](#):

```

public PlaneCircle(double r, double x, double y) {
    super(r);        // Invoke the constructor of the superclass, Circle()
    this.cx = x;      // Initialize the instance field cx
    this.cy = y;      // Initialize the instance field cy
}

```

This constructor explicitly initializes the `cx` and `cy` fields newly defined by `PlaneCircle`, but it relies on the superclass `Circle()` constructor to initialize the inherited fields of the class. To invoke the superclass constructor, our constructor calls `super()`. `super` is a reserved word in Java. One of its uses is to invoke the constructor method of a superclass from within the constructor method of a subclass. This use is analogous to the use of `this()` to invoke one constructor method of a class from within another constructor method of the same class. Invoking a constructor using `super()` is subject to the same restrictions as is using `this()`:

- `super()` can be used in this way only within a constructor method.
- The call to the superclass constructor must appear as the first statement within the constructor method, even before local variable declarations.

The arguments passed to `super()` must match the parameters of the superclass constructor. If the superclass defines more than one constructor, `super()` can be used to invoke any one of them, depending on the arguments passed.

3.5.4. Constructor Chaining and the Default Constructor

Java guarantees that the constructor method of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, Java must ensure that

every constructor method calls its superclass constructor method. Thus, if the first statement in a constructor does not explicitly invoke another constructor with `this()` or `super()`, Java implicitly inserts the call `super()`, that is, it calls the superclass constructor with no arguments. If the superclass does not have a constructor that takes no arguments, this implicit invocation causes a compilation error.

Consider what happens when we create a new instance of the `PlaneCircle` class. First, the `PlaneCircle` constructor is invoked. This constructor explicitly calls `super(r)` to invoke a `Circle` constructor, and that `Circle()` constructor implicitly calls `super()` to invoke the constructor of its superclass, `Object`. The body of the `Object` constructor runs first. When it returns, the body of the `Circle()` constructor runs. Finally, when the call to `super(r)` returns, the remaining statements of the `PlaneCircle()` constructor are executed.

What all this means is that constructor calls are chained; any time an object is created, a sequence of constructor methods is invoked, from subclass to superclass on up to `Object` at the root of the class hierarchy. Because a superclass constructor is always invoked as the first statement of its subclass constructor, the body of the `Object` constructor always runs first, followed by the constructor of its subclass and on down the class hierarchy to the class that is being instantiated. There is an important implication here; when a constructor is invoked, it can count on the fields of its superclass to be initialized.

3.5.4.1. The default constructor

There is one missing piece in the previous description of constructor chaining. If a constructor does not invoke a superclass constructor, Java does so implicitly. But what if a class is declared without a constructor? In this case, Java implicitly adds a constructor to the class. This default constructor does nothing but invoke the superclass constructor. For example, if we don't declare a constructor for the `PlaneCircle` class, Java implicitly inserts this constructor:

```
public PlaneCircle() { super(); }
```

If the superclass, `Circle`, doesn't declare a no-argument constructor, the `super()` call in this automatically inserted default constructor for `PlaneCircle()` causes a compilation error. In general, if a class does not define a no-argument constructor, all its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments.

If a class does not declare any constructors, it is given a no-argument constructor by default. Classes declared `public` are given `public` constructors. All other classes are given a default constructor that is declared without any visibility modifier: such a

constructor has default visibility. (The notion of visibility is explained later in this chapter.) If you are creating a `public` class that should not be publicly instantiated, you should declare at least one non-`public` constructor to prevent the insertion of a default `public` constructor. Classes that should never be instantiated (such as `java.lang.Math` or `java.lang.System`) should define a `private` constructor. Such a constructor can never be invoked from outside of the class, but it prevents the automatic insertion of the default constructor.

3.5.4.2. Finalizer chaining?

You might assume that since Java chains constructor methods, it also automatically chains the finalizer methods for an object. In other words, you might assume that the finalizer method of a class automatically invokes the finalizer of its superclass, and so on. In fact, Java does *not* do this. When you write a `finalize()` method, you must explicitly invoke the superclass finalizer. (You should do this even if you know that the superclass does not have a finalizer because a future implementation of the superclass might add a finalizer.)

As we saw in our example finalizer earlier in the chapter, you can invoke a superclass method with a special syntax that uses the `super` keyword:

```
// Invoke the finalizer of our superclass
super.finalize();
```

We'll discuss this syntax in more detail when we consider method overriding. In practice, the need for finalizer methods, and thus finalizer chaining, rarely arises.

3.5.5. Hiding Superclass Fields

For the sake of example, imagine that our `PlaneCircle` class needs to know the distance between the center of the circle and the origin (0,0). We can add another instance field to hold this value:

```
public double r;
```

Adding the following line to the constructor computes the value of the field:

```
this.r = Math.sqrt(cx*cx + cy*cy); // Pythagorean theorem
```

But wait; this new field `r` has the same name as the radius field `r` in the `Circle` superclass. When this happens, we say that the field `r` of `PlaneCircle` *hides* the field `r` of `Circle`. (This is a contrived example, of course: the new field should really be called `distanceFromOrigin`. Although you should attempt to avoid it, subclass fields do sometimes hide fields of their superclass.)

With this new definition of `PlaneCircle`, the expressions `r` and `this.r` both refer to the field of `PlaneCircle`. How, then, can we refer to the field `r` of `Circle` that holds the radius of the circle? A special syntax for this uses the `super` keyword:

```
r          // Refers to the PlaneCircle field
this.r     // Refers to the PlaneCircle field
super.r    // Refers to the Circle field
```

Another way to refer to a hidden field is to cast `this` (or any instance of the class) to the appropriate superclass and then access the field:

```
((Circle) this).r // Refers to field r of the Circle class
```

This casting technique is particularly useful when you need to refer to a hidden field defined in a class that is not the immediate superclass. Suppose, for example, that classes `A`, `B`, and `C` all define a field named `x` and that `C` is a subclass of `B`, which is a subclass of `A`. Then, in the methods of class `C`, you can refer to these different fields as follows:

```
x          // Field x in class C
this.x     // Field x in class C
super.x    // Field x in class B
((B)this).x // Field x in class B
((A)this).x // Field x in class A
super.super.x // Illegal; does not refer to x in class A
```

You cannot refer to a hidden field `x` in the superclass of a superclass with `super.super.x`. This is not legal syntax.

Similarly, if you have an instance `c` of class `C`, you can refer to the three fields named `x` like this:

```
c.x          // Field x of class C
((B)c).x     // Field x of class B
((A)c).x     // Field x of class A
```

So far, we've been discussing instance fields. Class fields can also be hidden. You can use the same `super` syntax to refer to the hidden value of the field, but this is never necessary since you can always refer to a class field by prepending the name of the desired class. Suppose that the implementer of `PlaneCircle` decides that the `Circle.PI` field does not express to enough decimal places. She can define her own class field `PI`:

```
public static final double PI = 3.14159265358979323846;
```

Now, code in `PlaneCircle` can use this more accurate value with the expressions `PI` or `PlaneCircle.PI`. It can also refer to the old, less accurate value with the expressions `super.PI` and `Circle.PI`. Note, however, that the `area()` and `circumference()` methods inherited by `PlaneCircle` are defined in the `Circle` class, so they use the value `Circle.PI`, even though that value is hidden now by `PlaneCircle.PI`.

3.5.6. Overriding Superclass Methods

When a class defines an instance method using the same name, return type, and parameters as a method in its superclass, that method *overrides* the method of the superclass. When the method is invoked for an object of the class, it is the new definition of the method that is called, not the superclass's old definition. In Java 5.0 and later, the return type of the overriding method may be a subclass of return type of the overridden method instead of being exactly the same type. This is known as a *covariant return* and is described in [Section 2.6.5](#) in Chapter 2.

Method overriding is an important and useful technique in object-oriented programming. `PlaneCircle` does not override either of the methods defined by `Circle`, but suppose we define another subclass of `Circle`, named `Ellipse`.^[5] In this case, it is important for `Ellipse` to override the `area()` and `circumference()` methods of `Circle` since the formulas used to compute the area and circumference of a circle do not work for ellipses.

^[5] Mathematical purists may argue that since all circles are ellipses, `Ellipse` should be the superclass and `Circle` the subclass. A pragmatic engineer might counter that circles can be represented with fewer instance fields, so `Circle` objects should not be burdened by inheriting unnecessary fields from `Ellipse`. In any case, this is a useful example here.

The upcoming discussion of method overriding considers only instance methods. Class methods behave quite differently, and there isn't much to say. Like fields, class methods can be hidden by a subclass but not overridden. As noted earlier in this chapter, it is good programming style to always prefix a class method invocation with the name of the class in which it is defined. If you consider the class name part of the class method name, the two methods have different names, so nothing is actually hidden at all. It is, however, illegal for a class method to hide an instance method.

Before we go any further with the discussion of method overriding, you should understand the difference between method overriding and method overloading. As we discussed in [Chapter 2](#), method overloading refers to the practice of defining multiple methods (in the same class) that have the same name but different parameter lists. This is very different from method overriding, so don't get them confused.

3.5.6.1. Overriding is not hiding

Although Java treats the fields and methods of a class analogously in many ways, method overriding is not like field hiding at all. You can refer to hidden fields simply by casting an object to an instance of the appropriate superclass, but you cannot invoke overridden instance methods with this technique. The following code illustrates this crucial difference:

```
class A {                                // Define a class named A
    int i = 1;                          // An instance field
    int f() { return i; }                // An instance method
    static char g() { return 'A'; }     // A class method
}

class B extends A {                    // Define a subclass of A
```

```

    int i = 2;                                // Hides field i in class A
    int f() { return -i; }                    // Overrides instance method f in class A
    static char g() { return 'B'; }          // Hides class method g() in class A
}

public class OverrideTest {
    public static void main(String args[]) {
        B b = new B();                       // Creates a new object of type B
        System.out.println(b.i);             // Refers to B.i; prints 2
        System.out.println(b.f());           // Refers to B.f(); prints -2
        System.out.println(b.g());           // Refers to B.g(); prints B
        System.out.println(B.g());           // This is a better way to invoke B.g()

        A a = (A) b;                         // Casts b to an instance of class A
        System.out.println(a.i);             // Now refers to A.i; prints 1
        System.out.println(a.f());           // Still refers to B.f(); prints -2
        System.out.println(a.g());           // Refers to A.g(); prints A
        System.out.println(A.g());           // This is a better way to invoke A.g()
    }
}

```

While this difference between method overriding and field hiding may seem surprising at first, a little thought makes the purpose clear. Suppose we are manipulating a bunch of `Circle` and `Ellipse` objects. To keep track of the circles and ellipses, we store them in an array of type `Circle[]`. (We can do this because `Ellipse` is a subclass of `Circle`, so all `Ellipse` objects are legal `Circle` objects.) When we loop through the elements of this array, we don't have to know or care whether the element is actually a `Circle` or an `Ellipse`. What we do care about very much, however, is that the correct value is computed when we invoke the `area()` method of any element of the array. In other words, we don't want to use the formula for the area of a circle when the object is actually an ellipse! Seen in this context, it is not surprising at all that method overriding is handled differently by Java than is field hiding.

3.5.6.2. Dynamic method lookup

If we have a `Circle[]` array that holds `Circle` and `Ellipse` objects, how does the compiler know whether to call the `area()` method of the `Circle` class or the `Ellipse` class for any given item in the array? In fact, the compiler does not know this because it cannot know it. The compiler knows that it does not know, however, and produces code that uses dynamic method lookup at runtime. When the interpreter runs the code, it looks up the appropriate `area()` method to call for each of the objects in the array. That is, when the interpreter interprets the expression `o.area()`, it checks the actual type of the object referred to by the variable `o` and then finds the `area()` method that is appropriate for that type. It does not simply use the `area()` method that is statically associated with the type of the variable `o`. This process of dynamic method lookup is sometimes also called virtual method invocation.^[6]

^[6] C++ programmers should note that dynamic method lookup is what C++ does for virtual functions. An important difference between Java and C++ is that Java does not have a `virtual` keyword. In Java, methods are virtual by default.

3.5.6.3. Final methods and static method lookup

Virtual method invocation is fast, but method invocation is faster when no dynamic lookup is necessary at runtime. Fortunately, Java does not always need to use dynamic method lookup. In particular, if a method is declared with the `final` modifier, it means that the method definition is the final one; it cannot be overridden by any subclasses. If a method cannot be overridden, the compiler knows that there is only one version of the method, and dynamic method lookup is not necessary.^[7] In addition, all methods of a `final` class are themselves implicitly final and cannot be overridden. As we'll discuss later in this chapter, `private` methods are not inherited by subclasses and, therefore, cannot be overridden (i.e., all `private` methods are implicitly `final`). Finally, class methods behave like fields (i.e., they can be hidden by subclasses but not overridden). Taken together, this means that all methods of a class that is declared `final`, as well as all methods that are `final`, `private`, or `static`, are invoked without dynamic method lookup. These methods are also candidates for inlining at runtime by a just-in-time compiler (JIT) or similar optimization tool.

^[7] In this sense, the `final` modifier is the opposite of the `virtual` modifier in C++. All non-`final` methods in Java are `virtual`.

3.5.6.4. Invoking an overridden method

We've seen the important differences between method overriding and field hiding. Nevertheless, the Java syntax for invoking an overridden method is quite similar to the syntax for accessing a hidden field: both use the `super` keyword. The following code illustrates:

```
class A {
    int i = 1;           // An instance field hidden by subclass B
    int f() { return i; } // An instance method overridden by subclass B
}

class B extends A {
    int i;               // This field hides i in A
    int f() {            // This method overrides f() in A
        i = super.i + 1; // It can retrieve A.i like this
        return super.f() + i; // It can invoke A.f() like this
    }
}
```

Recall that when you use `super` to refer to a hidden field, it is the same as casting `this` to the superclass type and accessing the field through that. Using `super` to invoke an overridden method, however, is not the same as casting `this`. In other words, in the previous code, the expression `super.f()` is not the same as `((A)this).f()`.

When the interpreter invokes an instance method with this `super` syntax, a modified form of dynamic method lookup is performed. The first step, as in regular dynamic method lookup, is to determine the actual class of the object through which the method is invoked. Normally, the dynamic search for an appropriate method definition would begin with this

class. When a method is invoked with the `super` syntax, however, the search begins at the superclass of the class. If the superclass implements the method directly, that version of the method is invoked. If the superclass inherits the method, the inherited version of the method is invoked.

Note that the `super` keyword invokes the most immediately overridden version of a method. Suppose class `A` has a subclass `B` that has a subclass `C` and that all three classes define the same method `f()`. The method `C.f()` can invoke the method `B.f()`, which it overrides directly, with `super.f()`. But there is no way for `C.f()` to invoke `A.f()` directly: `super.super.f()` is not legal Java syntax. Of course, if `C.f()` invokes `B.f()`, it is reasonable to suppose that `B.f()` might also invoke `A.f()`. This kind of chaining is relatively common when working with overridden methods: it is a way of augmenting the behavior of a method without replacing the method entirely. We saw this technique in the the example `finalize()` method shown earlier in the chapter: that method invoked `super.finalize()` to run its superclass finalization method.

Don't confuse the use of `super` to invoke an overridden method with the `super()` method call used in constructor methods to invoke a superclass constructor. Although they both use the same keyword, these are two entirely different syntaxes. In particular, you can use `super` to invoke an overridden method anywhere in the overriding class while you can use `super()` only to invoke a superclass constructor as the very first statement of a constructor.

It is also important to remember that `super` can be used only to invoke an overridden method from within the class that overrides it. Given an `Ellipse` object `e`, there is no way for a program that uses an object (with or without the `super` syntax) to invoke the `area()` method defined by the `Circle` class on this object.

3.6. Data Hiding and Encapsulation

We started this chapter by describing a class as a collection of data and methods. One of the important object-oriented techniques we haven't discussed so far is hiding the data within the class and making it available only through the methods. This technique is known as *encapsulation* because it seals the data (and internal methods) safely inside the "capsule" of the class, where it can be accessed only by trusted users (i.e., the methods of the class).

Why would you want to do this? The most important reason is to hide the internal implementation details of your class. If you prevent programmers from relying on those details, you can safely modify the implementation without worrying that you will break existing code that uses the class.

Another reason for encapsulation is to protect your class against accidental or willful stupidity. A class often contains a number of interdependent fields that must be in a consistent state. If you allow a programmer (including yourself) to manipulate those fields directly, he may change one field without changing important related fields, leaving the class in an inconsistent state. If instead he has to call a method to change the field, that method can be sure to do everything necessary to keep the state consistent. Similarly, if a class defines certain methods for internal use only, hiding these methods prevents users of the class from calling them.

Here's another way to think about encapsulation: when all the data for a class is hidden, the methods define the only possible operations that can be performed on objects of that class. Once you have carefully tested and debugged your methods, you can be confident that the class will work as expected. On the other hand, if all the fields of the class can be directly manipulated, the number of possibilities you have to test becomes unmanageable.

Other reasons to hide fields and methods of a class include:

- Internal fields and methods that are visible outside the class just clutter up the API. Keeping visible fields to a minimum keeps your class tidy and therefore easier to use and understand.
- If a field or method is visible to the users of your class, you have to document it. Save yourself time and effort by hiding it instead.

3.6.1. Access Control

All the fields and methods of a class can always be used within the body of the class itself. Java defines access control rules that restrict members of a class from being used outside the class. In a number of examples in this chapter, you've seen the `public` modifier used in field and method declarations. This `public` keyword, along with `protected` and `private`, are *access control modifiers*; they specify the access rules for the field or method.

3.6.1.1. Access to packages

A package is always accessible to code defined within the package. Whether it is accessible to code from other packages depends on the way the package is deployed on the host system. When the class files that comprise a package are stored in a directory, for example, a user must have read access to the directory and the files within it in order to have access to the package. Package access is not part of the Java language itself. Access control is usually done at the level of classes and members of classes instead.

3.6.1.2. Access to classes

By default, top-level classes are accessible within the package in which they are defined. However, if a top-level class is declared `public`, it is accessible everywhere (or everywhere that the package itself is accessible). The reason that we've restricted these statements to top-level classes is that, as we'll see later in this chapter, classes can also be defined as members of other classes. Because these inner classes are members of a class, they obey the member access-control rules.

3.6.1.3. Access to members

The members of a class are always accessible within the body of the class. By default, members are also accessible throughout the package in which the class is defined. This implies that classes placed in the same package should trust each other with their internal implementation details. This default level of access is often called *package access*. It is only one of four possible levels of access. The other three levels of access are defined by the `public`, `protected`, and `private` modifiers. Here is some example code that uses these modifiers:

```
public class Laundromat {           // People can use this class.
    private Laundry[] dirty;        // They cannot use this internal field,
    public void wash() { ... }      // but they can use these public methods
    public void dry() { ... }       // to manipulate the internal field.
    protected int temperature;     // A subclass might want to tweak this field
}
```

These access rules apply to members of a class:

- If a member of a class is declared with the `public` modifier, it means that the member is accessible anywhere the containing class is accessible. This is the least restrictive type of access control.
- If a member of a class is declared `private`, the member is never accessible, except within the class itself. This is the most restrictive type of access control.
- If a member of a class is declared `protected`, it is accessible to all classes within the package (the same as the default package accessibility) and also accessible within the body of any subclass of the class, regardless of the package in which that subclass is defined. This is more restrictive than `public` access, but less restrictive than package access.
- If a member of a class is not declared with any of these modifiers, it has the default package access: it is accessible to code within all classes that are defined in the same package but inaccessible outside of the package.

`protected` access requires a little more elaboration. Suppose class `A` declares a `protected` field `x` and is extended by a class `B`, which is defined in a different package (this last point is important). Class `B` inherits the `protected` field `x`, and its code can access that field in the current instance of `B` or in any other instances of `B` that the code

can refer to. This does not mean, however, that the code of class B can start reading the protected fields of arbitrary instances of A! If an object is an instance of A but is not an instance of B, its fields are obviously not inherited by B, and the code of class B cannot read them.

3.6.1.4. Access control and inheritance

The Java specification states that a subclass inherits all the instance fields and instance methods of its superclass accessible to it. If the subclass is defined in the same package as the superclass, it inherits all `non-private` instance fields and methods. If the subclass is defined in a different package, however, it inherits all `protected` and `public` instance fields and methods. `private` fields and methods are never inherited; neither are class fields or class methods. Finally, constructors are not inherited; they are chained, as described earlier in this chapter.

The statement that a subclass does not inherit the inaccessible fields and methods of its superclass can be a confusing one. It would seem to imply that when you create an instance of a subclass, no memory is allocated for any `private` fields defined by the superclass. This is not the intent of the statement, however. Every instance of a subclass does, in fact, include a complete instance of the superclass within it, including all inaccessible fields and methods. It is simply a matter of terminology. Because the inaccessible fields cannot be used in the subclass, we say they are not inherited. Earlier in this section we said that the members of a class are always accessible within the body of the class. If this statement is to apply to all members of the class, including inherited members, we must define "inherited members" to include only those members that are accessible. If you don't care for this definition, you can think of it this way instead:

- A class inherits *all* instance fields and instance methods (but not constructors) of its superclass.
- The body of a class can always access all the fields and methods it declares itself. It can also access the *accessible* fields and members it inherits from its superclass.

3.6.1.5. Member access summary

Table 3-1 summarizes the member access rules.

Table 3-1. Class member accessibility

Accessible to	Member visibility			
	Public	Protected	Package	Private
Defining class	Yes	Yes	Yes	Yes
Class in same package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

Here are some simple rules of thumb for using visibility modifiers:

- Use `public` only for methods and constants that form part of the public API of the class. Certain important or frequently used fields can also be `public`, but it is common practice to make fields non-`public` and encapsulate them with `public` accessor methods.
- Use `protected` for fields and methods that aren't required by most programmers using the class but that may be of interest to anyone creating a subclass as part of a different package. Note that `protected` members are technically part of the exported API of a class. They should be documented and cannot be changed without potentially breaking code that relies on them.
- Use the default package visibility for fields and methods that are internal implementation details but are used by cooperating classes in the same package. You cannot take real advantage of package visibility unless you use the `package` directive to group your cooperating classes into a package.
- Use `private` for fields and methods that are used only inside the class and should be hidden everywhere else.

If you are not sure whether to use `protected`, `package`, or `private` accessibility, it is better to start with overly restrictive member access. You can always relax the access restrictions in future versions of your class, if necessary. Doing the reverse is not a good idea because increasing access restrictions is not a backward-compatible change and can break code that relies on access to those members.

3.6.2. Data Accessor Methods

In the `Circle` example, we declared the circle radius to be a `public` field. The `Circle` class is one in which it may well be reasonable to keep that field publicly accessible; it is a simple enough class, with no dependencies between its fields. On the other hand, our current implementation of the class allows a `Circle` object to have a negative radius, and circles with negative radii should simply not exist. As long as the radius is stored in a `public` field, however, any programmer can set the field to any value she wants, no matter how unreasonable. The only solution is to restrict the programmer's direct access to the field and define `public` methods that provide indirect access to the field. Providing `public` methods to read and write a field is not the same as making the field itself `public`. The crucial difference is that methods can perform error checking.

Example 3-4 shows how we might reimplement `Circle` to prevent circles with negative radii. This version of `Circle` declares the `r` field to be `protected` and defines accessor methods named `getRadius()` and `setRadius()` to read and write the field value while enforcing the restriction on negative radius values. Because the `r` field is `protected`, it is directly (and more efficiently) accessible to subclasses.

Example 3-4. The Circle class using data hiding and encapsulation

```

package shapes;                // Specify a package for the class

public class Circle {          // The class is still public
    // This is a generally useful constant, so we keep it public
    public static final double PI = 3.14159;

    protected double r;        // Radius is hidden but visible to subclasses

    // A method to enforce the restriction on the radius
    // This is an implementation detail that may be of interest to subclasses
    protected void checkRadius(double radius) {
        if (radius < 0.0)
            throw new IllegalArgumentException("radius may not be negative.");
    }

    // The constructor method
    public Circle(double r) {
        checkRadius(r);
        this.r = r;
    }

    // Public data accessor methods
    public double getRadius() { return r; }
    public void setRadius(double r) {
        checkRadius(r);
        this.r = r;
    }

    // Methods to operate on the instance field
    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
}

```

We have defined the `Circle` class within a package named `shapes`. Since `r` is `protected`, any other classes in the `shapes` package have direct access to that field and can set it however they like. The assumption here is that all classes within the `shapes` package were written by the same author or a closely cooperating group of authors and that the classes all trust each other not to abuse their privileged level of access to each other's implementation details.

Finally, the code that enforces the restriction against negative radius values is itself placed within a `protected` method, `checkRadius()`. Although users of the `Circle` class cannot call this method, subclasses of the class can call it and even override it if they want to change the restrictions on the radius.

Note particularly the `getRadius()` and `setRadius()` methods of [Example 3-4](#). It is a common convention in Java that data accessor methods begin with the prefixes "get" and "set." If the field being accessed is of type `boolean`, however, the `get()` method may be replaced with an equivalent method that begins with "is." For example, the accessor method for a `boolean` field named `readable` is typically called `isReadable()` instead of `getReadable()`. In the programming conventions of the JavaBeans component model (covered in [Chapter 7](#)), a hidden field with one or more data accessor methods whose

names begin with "get," "is," or "set" is called a *property*. An interesting way to study a complex class is to look at the set of properties it defines. Properties are particularly common in the AWT and Swing APIs, which are covered in *Java Foundation Classes in a Nutshell* (O'Reilly).

3.7. Abstract Classes and Methods

In [Example 3-4](#), we declared our `Circle` class to be part of a package named `shapes`. Suppose we plan to implement a number of shape classes: `Rectangle`, `Square`, `Ellipse`, `Triangle`, and so on. We can give these shape classes our two basic `area()` and `circumference()` methods. Now, to make it easy to work with an array of shapes, it would be helpful if all our shape classes had a common superclass, `Shape`. If we structure our class hierarchy this way, every shape object, regardless of the actual type of shape it represents, can be assigned to variables, fields, or array elements of type `Shape`. We want the `Shape` class to encapsulate whatever features all our shapes have in common (e.g., the `area()` and `circumference()` methods). But our generic `Shape` class doesn't represent any real kind of shape, so it cannot define useful implementations of the methods. Java handles this situation with *abstract methods*.

Java lets us define a method without implementing it by declaring the method with the `abstract` modifier. An abstract method has no body; it simply has a signature definition followed by a semicolon.^[8] Here are the rules about abstract methods and the abstract classes that contain them:

^[8] An abstract method in Java is something like a pure virtual function in C++ (i.e., a virtual function that is declared `= 0`). In C++, a class that contains a pure virtual function is called an abstract class and cannot be instantiated. The same is true of Java classes that contain abstract methods.

- Any class with an abstract method is automatically abstract itself and must be declared as such.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a *concrete* subclass, to emphasize the fact that it is not abstract.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract and must be declared as such.
- `static`, `private`, and `final` methods cannot be abstract since these types of methods cannot be overridden by a subclass. Similarly, a `final` class cannot contain any abstract methods.
- A class can be declared abstract even if it does not actually have any abstract methods. Declaring such a class abstract indicates that the implementation is

somehow incomplete and is meant to serve as a superclass for one or more subclasses that complete the implementation. Such a class cannot be instantiated.

There is an important feature of the rules of abstract methods. If we define the `Shape` class to have abstract `area()` and `circumference()` methods, any subclass of `Shape` is required to provide implementations of these methods so that it can be instantiated. In other words, every `Shape` object is guaranteed to have implementations of these methods defined. [Example 3-5](#) shows how this might work. It defines an abstract `Shape` class and two concrete subclasses of it.

Example 3-5. An abstract class and concrete subclasses

```
public abstract class Shape {
    public abstract double area();           // Abstract methods: note
    public abstract double circumference();  // semicolon instead of body.
}

class Circle extends Shape {
    public static final double PI = 3.14159265358979323846;
    protected double r;                    // Instance data
    public Circle(double r) { this.r = r; } // Constructor
    public double getRadius() { return r; } // Accessor
    public double area() { return PI*r*r; }  // Implementations of
    public double circumference() { return 2*PI*r; } // abstract methods.
}

class Rectangle extends Shape {
    protected double w, h;                  // Instance data
    public Rectangle(double w, double h) {   // Constructor
        this.w = w; this.h = h;
    }
    public double getWidth() { return w; }   // Accessor method
    public double getHeight() { return h; }  // Another accessor
    public double area() { return w*h; }     // Implementations of
    public double circumference() { return 2*(w + h); } // abstract methods.
}
```

Each abstract method in `Shape` has a semicolon right after its parentheses. They have no curly braces, and no method body is defined. Using the classes defined in [Example 3-5](#), we can now write code such as:

```
Shape[] shapes = new Shape[3];           // Create an array to hold shapes
shapes[0] = new Circle(2.0);              // Fill in the array
shapes[1] = new Rectangle(1.0, 3.0);
shapes[2] = new Rectangle(4.0, 2.0);

double total_area = 0;
for(int i = 0; i < shapes.length; i++)
    total_area += shapes[i].area();       // Compute the area of the shapes
```

Notice two important points here:

- Subclasses of `Shape` can be assigned to elements of an array of `Shape`. No cast is necessary. This is another example of a widening reference type conversion (discussed in [Chapter 2](#)).

- You can invoke the `area()` and `circumference()` methods for any `Shape` object, even though the `Shape` class does not define a body for these methods. When you do this, the method to be invoked is found using dynamic method lookup, so the area of a circle is computed using the method defined by `Circle`, and the area of a rectangle is computed using the method defined by `Rectangle`.

3.8. Important Methods of `java.lang.Object`

As we've noted, all classes extend, directly or indirectly, `java.lang.Object`. This class defines several important methods that you should consider overriding in every class you write. [Example 3-6](#) shows a class that overrides these methods. The sections that follow the example document the default implementation of each method and explain why you might want to override it. You may also find it helpful to look up `Object` in the reference section for an API listing.

Some of the syntax in [Example 3-6](#) may be unfamiliar to you. The example uses two Java 5.0 features. First, it implements a parameterized, or generic, version of the `Comparable` interface. Second, the example uses the `@Override` annotation to emphasize (and have the compiler verify) that certain methods override `Object`. Parameterized types and annotations are covered in [Chapter 4](#).

Example 3-6. A class that overrides important Object methods

```

// This class represents a circle with immutable position and radius.
public class Circle implements Comparable<Circle> {
    // These fields hold the coordinates of the center and the radius.
    // They are private for data encapsulation and final for immutability
    private final int x, y, r;

    // The basic constructor: initialize the fields to specified values
    public Circle(int x, int y, int r) {
        if (r < 0) throw new IllegalArgumentException("negative radius");
        this.x = x; this.y = y; this.r = r;
    }

    // This is a "copy constructor"--a useful alternative to clone()
    public Circle(Circle original) {
        x = original.x;    // Just copy the fields from the original
        y = original.y;
        r = original.r;
    }

    // Public accessor methods for the private fields.
    // These are part of data encapsulation.
    public int getX() { return x; }
    public int getY() { return y; }
    public int getR() { return r; }

    // Return a string representation
    @Override public String toString() {
        return String.format("center=(%d,%d); radius=%d", x, y, r);
    }

    // Test for equality with another object
    @Override public boolean equals(Object o) {
        if (o == this) return true;           // Identical references?
        if (!(o instanceof Circle)) return false; // Correct type and non-null?
        Circle that = (Circle) o;             // Cast to our type
        if (this.x == that.x && this.y == that.y && this.r == that.r)
            return true;                       // If all fields match
        else
            return false;                      // If fields differ
    }

    // A hash code allows an object to be used in a hash table.
    // Equal objects must have equal hash codes. Unequal objects are allowed
    // to have equal hash codes as well, but we try to avoid that.
    // We must override this method since we also override equals().
    @Override public int hashCode() {
        int result = 17;                      // This hash code algorithm from the book
        result = 37*result + x;                // _Effective Java_, by Joshua Bloch
        result = 37*result + y;
        result = 37*result + r;
        return result;
    }

    // This method is defined by the Comparable interface.
    // Compare this Circle to that Circle. Return a value < 0 if this < that.
    // Return 0 if this == that. Return a value > 0 if this > that.
    // Circles are ordered top to bottom, left to right, and then by radius
    public int compareTo(Circle that) {
        long result = that.y - this.y; // Smaller circles have bigger y values
        if (result == 0) result = this.x - that.x; // If same compare l-to-r
        if (result == 0) result = this.r - that.r; // If same compare radius

        // We have to use a long value for subtraction because the differences
        // between a large positive and large negative value could overflow
        // an int. But we can't return the long, so return its sign as an int.
        return Long.signum(result); // new in Java 5.0
    }
}

```

Chapter 3. Object-Oriented Programming in Java

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

Prepared for Ronald Fischer, Safari ID: ronald.fischer@fussshuhn.de
 User number: 628024 Copyright 2008, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.

3.8.1. toString()

The purpose of the `toString()` method is to return a textual representation of an object. The method is invoked automatically on objects during string concatenation and by methods such as `System.out.println()`. Giving objects a textual representation can be quite helpful for debugging or logging output, and a well-crafted `toString()` method can even help with tasks such as report generation.

The version of `toString()` inherited from `Object` returns a string that includes the name of the class of the object as well as a hexadecimal representation of the `hashCode()` value of the object (discussed later in this chapter). This default implementation provides basic type and identity information for an object but is not usually very useful. The `toString()` method in [Example 3-6](#) instead returns a human-readable string that includes the value of each of the fields of the `Circle` class.

3.8.2. equals()

The `=` operator tests two references to see if they refer to the same object. If you want to test whether two distinct objects are equal to one another, you must use the `equals()` method instead. Any class can define its own notion of equality by overriding `equals()`. The `Object.equals()` method simply uses the `==` operator: this default method considers two objects equal only if they are actually the very same object.

The `equals()` method in [Example 3-6](#) considers two distinct `Circle` objects to be equal if their fields are all equal. Note that it first does a quick identity test with `==` as an optimization and then checks the type of the other object with `instanceof`: a `Circle` can be equal only to another `Circle`, and it is not acceptable for an `equals()` method to throw a `ClassCastException`. Note that the `instanceof` test also rules out `null` arguments: `instanceof` always evaluates to `false` if its left-hand operand is `null`.

3.8.3. hashCode()

Whenever you override `equals()`, you must also override `hashCode()`. This method returns an integer for use by hash table data structures. It is critical that two objects have the same hash code if they are equal according to the `equals()` method. It is important (for efficient operation of hash tables) but not required that unequal objects have unequal hash codes, or at least that unequal objects are unlikely to share a hash code. This second criterion can lead to `hashCode()` methods that involve mildly tricky arithmetic or bit-manipulation.

The `Object.hashCode()` method works with the `Object.equals()` method and returns a hash code based on object identity rather than object equality. (If you ever need

an identity-based hash code, you can access the functionality of `Object.hashCode()` through the static method `System.identityHashCode()`. When you override `equals()`, you must always override `hashCode()` to guarantee that equal objects have equal hash codes. Since the `equals()` method in [Example 3-6](#) bases object equality on the values of the three fields, the `hashCode()` method computes its hash code based on these three fields as well. It is clear from the code that if two `Circle` objects have the same field values, they will have the same hash code.

Note that the `hashCode()` method in [Example 3-6](#) does not simply add the three fields and return their sum. Such an implementation would be legal but not efficient because two circles with the same radius but whose X and Y coordinates were reversed would then have the same hash code. The repeated multiplication and addition steps "spread out" the range of hash codes and dramatically reduce the likelihood that two unequal `Circle` objects have the same code. *Effective Java Programming Guide* by Joshua Bloch (Addison Wesley) includes a helpful recipe for constructing efficient `hashCode()` methods like this one.

3.8.4. Comparable.compareTo()

[Example 3-6](#) includes a `compareTo()` method. This method is defined by the `java.lang.Comparable` interface rather than by `Object`. (It actually uses the generics features of Java 5.0 and implements a parameterized version of the interface: `Comparable<Circle>`, but we can ignore that fact until [Chapter 4](#).) The purpose of `Comparable` and its `compareTo()` method is to allow instances of a class to be compared to each other in the way that the `<`, `<=`, `>` and `>=` operators compare numbers. If a class implements `Comparable`, we can say that one instance is less than, greater than, or equal to another instance. Instances of a `Comparable` class can be sorted.

Since `compareTo()` is defined by an interface, the `Object` class does not provide any default implementation. It is up to each individual class to determine whether and how its instances should be ordered and to include a `compareTo()` method that implements that ordering. The ordering defined by [Example 3-6](#) compares `Circle` objects as if they were words on a page. Circles are first ordered from top to bottom: circles with larger Y coordinates are less than circles with smaller Y coordinates. If two circles have the same Y coordinate, they are ordered from left to right. A circle with a smaller X coordinate is less than a circle with a larger X coordinate. Finally, if two circles have the same X and Y coordinates, they are compared by radius. The circle with the smaller radius is smaller. Notice that under this ordering, two circles are equal only if all three of their fields are equal. This means that the ordering defined by `compareTo()` is consistent with the equality defined by `equals()`. This is very desirable (but not strictly required).

The `compareTo()` method returns an `int` value that requires further explanation. `compareTo()` should return a negative number if the `this` object is less than the object passed to it. It should return 0 if the two objects are equal. And `compareTo()` should return a positive number if `this` is greater than the method argument.

3.8.5. clone()

`Object` defines a method named `clone()` whose purpose is to return an object with fields set identically to those of the current object. This is an unusual method for two reasons. First, it works only if the class implements the `java.lang.Cloneable` interface. `Cloneable` does not define any methods, so implementing it is simply a matter of listing it in the `implements` clause of the class signature. The other unusual feature of `clone()` is that it is declared `protected` (see [Section 3.6](#) earlier in this chapter). This means that subclasses of `Object` can call and override `Object.clone()`, but other code cannot call it. Therefore, if you want your object to be cloneable, you must implement `Cloneable` and override the `clone()` method, making it `public`.

The `Circle` class of [Example 3-6](#) does not implement `Cloneable`; instead it provides a *copy constructor* for making copies of `Circle` objects:

```
Circle original = new Circle(1, 2, 3); // regular constructor
Circle copy = new Circle(original);    // copy constructor
```

It can be difficult to implement `clone()` correctly, and it is usually easier and safer to provide a copy constructor. To make the `Circle` class cloneable, you would add `Cloneable` to the `implements` clause and add the following method to the class body:

```
@Override public Object clone() {
    try { return super.clone(); }
    catch(CloneNotSupportedException e) { throw new AssertionError(e); }
}
```

See *Effective Java Programming Guide* by Joshua Bloch for a detailed discussion of the ins and outs of `clone()` and `Cloneable`.

3.9. Interfaces

Like a class, an *interface* defines a new reference type. Unlike classes, however, interfaces provide no implementation for the types they define. As its name implies, an interface specifies only an API: all of its methods are `abstract` and have no bodies. It is not possible to directly instantiate an interface and create a member of the interface type. Instead, a class must *implement* the interface to provide the necessary method bodies. Any instances of that class are members of both the type defined by the class and the type defined by the interface. Interfaces provide a limited but very powerful alternative to *multiple inheritance*.^[9] Classes in Java can inherit members from only a single superclass, but they

can implement any number of interfaces. Objects that do not share the same class or superclass may still be members of the same type by virtue of implementing the same interface.

[9] C++ supports multiple inheritance, but the ability of a class to have more than one superclass adds a lot of complexity to the language.

3.9.1. Defining an Interface

An interface definition is much like a class definition in which all the methods are abstract and the keyword `class` has been replaced with `interface`. For example, the following code shows the definition of an interface named `Centered`. A `Shape` class, such as those defined earlier in the chapter, might implement this interface if it wants to allow the coordinates of its center to be set and queried:

```
public interface Centered {
    void setCenter(double x, double y);
    double getCenterX();
    double getCenterY();
}
```

A number of restrictions apply to the members of an interface:

- An interface contains no implementation whatsoever. All methods of an interface are implicitly `abstract` and must have a semicolon in place of a method body. The `abstract` modifier is allowed but, by convention, is usually omitted. Since static methods may not be abstract, the methods of an interface may not be declared `static`.
- An interface defines a public API. All members of an interface are implicitly `public`, and it is conventional to omit the unnecessary `public` modifier. It is an error to define a `protected` or `private` method in an interface.
- An interface may not define any instance fields. Fields are an implementation detail, and an interface is a pure specification without any implementation. The only fields allowed in an interface definition are constants that are declared both `static` and `final`.
- An interface cannot be instantiated, so it does not define a constructor.
- Interfaces may contain nested types. Any such types are implicitly `public` and `static`. See [Section 3.10](#) later in this chapter.

3.9.1.1. Extending interfaces

Interfaces may extend other interfaces, and, like a class definition, an interface definition may include an `extends` clause. When one interface extends another, it inherits all the abstract methods and constants of its superinterface and can define new abstract methods and constants. Unlike classes, however, the `extends` clause of an interface definition may

include more than one superinterface. For example, here are some interfaces that extend other interfaces:

```
public interface Positionable extends Centered {
    void setUpperRightCorner(double x, double y);
    double getUpperRightX();
    double getUpperRightY();
}
public interface Transformable extends Scalable, Translatable, Rotatable {}
public interface SuperShape extends Positionable, Transformable {}
```

An interface that extends more than one interface inherits all the abstract methods and constants from each of those interfaces and can define its own additional abstract methods and constants. A class that implements such an interface must implement the abstract methods defined directly by the interface, as well as all the abstract methods inherited from all the superinterfaces.

3.9.2. Implementing an Interface

Just as a class uses `extends` to specify its superclass, it can use `implements` to name one or more interfaces it supports. `implements` is a Java keyword that can appear in a class declaration following the `extends` clause. `implements` should be followed by a comma-separated list of interfaces that the class implements.

When a class declares an interface in its `implements` clause, it is saying that it provides an implementation (i.e., a body) for each method of that interface. If a class implements an interface but does not provide an implementation for every interface method, it inherits those unimplemented abstract methods from the interface and must itself be declared `abstract`. If a class implements more than one interface, it must implement every method of each interface it implements (or be declared `abstract`).

The following code shows how we can define a `CenteredRectangle` class that extends the `Rectangle` class from earlier in the chapter and implements our `Centered` interface.

```
public class CenteredRectangle extends Rectangle implements Centered {
    // New instance fields
    private double cx, cy;

    // A constructor
    public CenteredRectangle(double cx, double cy, double w, double h) {
        super(w, h);
        this.cx = cx;
        this.cy = cy;
    }

    // We inherit all the methods of Rectangle but must
    // provide implementations of all the Centered methods.
    public void setCenter(double x, double y) { cx = x; cy = y; }
    public double getCenterX() { return cx; }
    public double getCenterY() { return cy; }
}
```

Suppose we implement `CenteredCircle` and `CenteredSquare` just as we have implemented this `CenteredRectangle` class. Since each class extends `Shape`, instances of the classes can be treated as instances of the `Shape` class, as we saw earlier. Since each class implements the `Centered` interface, instances can also be treated as instances of that type. The following code demonstrates how objects can be members of both a class type and an interface type:

```
Shape[] shapes = new Shape[3];           // Create an array to hold shapes

// Create some centered shapes, and store them in the Shape[]
// No cast necessary: these are all widening conversions
shapes[0] = new CenteredCircle(1.0, 1.0, 1.0);
shapes[1] = new CenteredSquare(2.5, 2, 3);
shapes[2] = new CenteredRectangle(2.3, 4.5, 3, 4);

// Compute average area of the shapes and average distance from the origin
double totalArea = 0;
double totalDistance = 0;
for(int i = 0; i < shapes.length; i++) {
    totalArea += shapes[i].area();        // Compute the area of the shapes
    if (shapes[i] instanceof Centered) { // The shape is a Centered shape
        // Note the required cast from Shape to Centered (no cast would
        // be required to go from CenteredSquare to Centered, however).
        Centered c = (Centered) shapes[i]; // Assign it to a Centered variable
        double cx = c.getCenterX();       // Get coordinates of the center
        double cy = c.getCenterY();       // Compute distance from origin
        totalDistance += Math.sqrt(cx*cx + cy*cy);
    }
}
System.out.println("Average area: " + totalArea/shapes.length);
System.out.println("Average distance: " + totalDistance/shapes.length);
```

This example demonstrates that interfaces are data types in Java, just like classes. When a class implements an interface, instances of that class can be assigned to variables of the interface type. Don't interpret this example to imply that you must assign a `CenteredRectangle` object to a `Centered` variable before you can invoke the `setCenter()` method or to a `Shape` variable before you can invoke the `area()` method. `CenteredRectangle` defines `setCenter()` and inherits `area()` from its `Rectangle` superclass, so you can always invoke these methods.

3.9.2.1. Implementing multiple interfaces

Suppose we want shape objects that can be positioned in terms of not only their center points but also their upper-right corners. And suppose we also want shapes that can be scaled larger and smaller. Remember that although a class can extend only a single superclass, it can implement any number of interfaces. Assuming we have defined appropriate `UpperRightCornered` and `Scalable` interfaces, we can declare a class as follows:

```
public class SuperDuperSquare extends Shape
    implements Centered, UpperRightCornered, Scalable {
    // Class members omitted here
}
```

When a class implements more than one interface, it simply means that it must provide implementations for all abstract methods in all its interfaces.

3.9.3. Interfaces vs. Abstract Classes

When defining an abstract type (e.g., `Shape`) that you expect to have many subtypes (e.g., `Circle`, `Rectangle`, `Square`), you are often faced with a choice between interfaces and abstract classes. Since they have similar features, it is not always clear which to use.

An interface is useful because any class can implement it, even if that class extends some entirely unrelated superclass. But an interface is a pure API specification and contains no implementation. If an interface has numerous methods, it can become tedious to implement the methods over and over, especially when much of the implementation is duplicated by each implementing class.

An abstract class does not need to be entirely abstract; it can contain a partial implementation that subclasses can take advantage of. In some cases, numerous subclasses can rely on default method implementations provided by an abstract class. But a class that extends an abstract class cannot extend any other class, which can cause design difficulties in some situations.

Another important difference between interfaces and abstract classes has to do with compatibility. If you define an interface as part of a public API and then later add a new method to the interface, you break any classes that implemented the previous version of the interface. If you use an abstract class, however, you can safely add nonabstract methods to that class without requiring modifications to existing classes that extend the abstract class.

In some situations, it is clear that an interface or an abstract class is the right design choice. In other cases, a common design pattern is to use both. Define the type as a totally abstract interface, then create an abstract class that implements the interface and provides useful default implementations that subclasses can take advantage of. For example:

```
// Here is a basic interface. It represents a shape that fits inside
// of a rectangular bounding box. Any class that wants to serve as a
// RectangularShape can implement these methods from scratch.
public interface RectangularShape {
    void setSize(double width, double height);
    void setPosition(double x, double y);
    void translate(double dx, double dy);
    double area();
    boolean isInside();
}

// Here is a partial implementation of that interface. Many
// implementations may find this a useful starting point.
public abstract class AbstractRectangularShape implements RectangularShape {
    // The position and size of the shape
    protected double x, y, w, h;

    // Default implementations of some of the interface methods
    public void setSize(double width, double height) { w = width; h = height; }
```

Chapter 3. Object-Oriented Programming in Java

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

Prepared for Ronald Fischer, Safari ID: ronald.fischer@fussshuhn.de
User number: 628024 Copyright 2008, Safari Books Online, LLC.

This PDF is exclusively for your use in accordance with the Safari Terms of Service. No part of it may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates the Safari Terms of Service is strictly prohibited.


```

    public void setPosition(double x, double y) { this.x = x; this.y = y; }
    public void translate (double dx, double dy) { x += dx; y += dy; }
}

```

3.9.4. Marker Interfaces

Sometimes it is useful to define an interface that is entirely empty. A class can implement this interface simply by naming it in its `implements` clause without having to implement any methods. In this case, any instances of the class become valid instances of the interface. Java code can check whether an object is an instance of the interface using the `instanceof` operator, so this technique is a useful way to provide additional information about an object.

The `java.io.Serializable` interface is a marker interface of this sort. A class implements `Serializable` interface to tell `ObjectOutputStream` that its instances may safely be serialized. `java.util.RandomAccess` is another example:

`java.util.List` implementations implement this interface to advertise that they provide fast random access to the elements of the list. `ArrayList` implements `RandomAccess`, for example, while `LinkedList` does not. Algorithms that care about the performance of random-access operations can test for `RandomAccess` like this:

```

// Before sorting the elements of a long arbitrary list, we may want to make
// sure that the list allows fast random access. If not, it may be quicker
// make a random-access copy of the list before sorting it.
// Note that this is not necessary when using java.util.Collections.sort().
List l = ...; // Some arbitrary list we're given
if (l.size() > 2 && !(l instanceof RandomAccess)) l = new ArrayList(l);
sortListInPlace(l);

```

3.9.5. Interfaces and Constants

As noted earlier, constants can appear in an interface definition. Any class that implements an interface inherits the constants it defines and can use them as if they were defined directly in the class itself. Importantly, there is no need to prefix the constants with the name of the interface or provide any kind of implementation of the constants.

When a set of constants is used by more than one class, it is tempting to define the constants once in an interface and then have any classes that require the constants implement the interface. This situation might arise, for example, when client and server classes implement a network protocol whose details (such as the port number to connect to and listen on) are captured in a set of symbolic constants. As a concrete example, consider the `java.io.ObjectStreamConstants` interface, which defines constants for the object serialization protocol and is implemented by both `ObjectInputStream` and `ObjectOutputStream`.

The primary benefit of inheriting constant definitions from an interface is that it saves typing: you don't need to specify the type that defines the constants. Despite its use with `ObjectStreamConstants`, this is not a recommended technique. The use of constants is an implementation detail that is not appropriate to declare in the `implements` clause of a class signature.

A better approach is to define constants in a class and use the constants by typing the full class name and the constant name. In Java 5.0 and later, you can save typing by importing the constants from their defining class with the `import static` declaration. See [Section 2.10](#) in Chapter 2 for details.

3.10. Nested Types

The classes, interfaces, and enumerated types we have seen so far in this book have all been defined as top-level classes. This means that they are direct members of packages, defined independently of other types. However, type definitions can also be nested within other type definitions. These *nested types*, commonly known as "inner classes," are a powerful and elegant feature of the Java language. A type can be nested within another type in four ways:

Static member types

A static member type is any type defined as a `static` member of another type. A `static` method is called a class method, so, by analogy, we could call this type of nested type a "class type," but this terminology would obviously be confusing. A static member type behaves much like an ordinary top-level type, but its name is part of the namespace, rather than the package, of the containing type. Also, a static member type can access the `static` members of the class that contains it. Nested interfaces, enumerated types, and annotation types are implicitly static, whether or not the `static` keyword appears. Any type nested within an interface or annotation is also implicitly `static`. Static member types may be defined within top-level types or nested to any depth within other static member types. A static member type may not be defined within any other kind of nested type, however.

Nonstatic member classes

A "nonstatic member type" is simply a member type that is not declared `static`. Since interfaces, enumerated types, and annotations are always implicitly static, however, we usually use the term "nonstatic member class" instead. Nonstatic member classes may be defined within other classes or enumerated types and are analogous to instance methods or fields. An instance of a nonstatic member class is

always associated with an instance of the enclosing type, and the code of a nonstatic member class has access to all the fields and methods (both `static` and `non-static`) of its enclosing type. Several features of Java syntax exist specifically to work with the enclosing instance of a nonstatic member class.

Local classes

A local class is a class defined within a block of Java code. Interfaces, enumerated types, and annotation types may not be defined locally. Like a local variable, a local class is visible only within the block in which it is defined. Although local classes are not member classes, they are still defined within an enclosing class, so they share many of the features of member classes. Additionally, however, a local class can access any `final` local variables or parameters that are accessible in the scope of the block that defines the class.

Anonymous classes

An anonymous class is a kind of local class that has no name; it combines the syntax for class definition with the syntax for object instantiation. While a local class definition is a Java statement, an anonymous class definition (and instantiation) is a Java expression, so it can appear as part of a larger expression, such as method invocation. Interfaces, enumerated types, and annotation types cannot be defined anonymously.

Nested types have no universally adopted nomenclature. The term "inner class" is commonly used. Sometimes, however, inner class is used to refer to a nonstatic member class, local class, or anonymous class, but not a static member type. Although the terminology for describing nested types is not always clear, the syntax for working with them is, and it is usually clear from context which kind of nested type is being discussed.

Now we'll describe each of the four kinds of nested types in greater detail. Each section describes the features of the nested type, the restrictions on its use, and any special Java syntax used with the type. These four sections are followed by an implementation note that explains how nested types work under the hood.

3.10.1. Static Member Types

A *static member type* is much like a regular top-level type. For convenience, however, it is nested within the namespace of another type. [Example 3-7](#) shows a helper interface defined as a static member of a containing class. The example also shows how this interface is used both within the class that contains it and by external classes. Note the use of its hierarchical name in the external class.

Example 3-7. Defining and using a static member interface

```
// A class that implements a stack as a linked list
public class LinkedStack {
    // This static member interface defines how objects are linked
    // The static keyword is optional: all nested interfaces are static
    public static interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // The head of the list is a Linkable object
    Linkable head;

    // Method bodies omitted
    public void push(Linkable node) { ... }
    public Object pop() { ... }
}

// This class implements the static member interface
class LinkableInteger implements LinkedStack.Linkable {
    // Here's the node's data and constructor
    int i;
    public LinkableInteger(int i) { this.i = i; }

    // Here are the data and methods required to implement the interface
    LinkedStack.Linkable next;
    public LinkedStack.Linkable getNext() { return next; }
    public void setNext(LinkedStack.Linkable node) { next = node; }
}
```

3.10.1.1. Features of static member types

A static member type is defined as a `static` member of a containing type. Any type (class, interface, enumerated type, or annotation type) may be defined as a static member of any other type. Interfaces, enumerated types, and annotation types are implicitly static, whether or not the `static` keyword appears in their definition.

A static member type is like the other static members of a class: static fields and static methods. Like a class method, a static member type is not associated with any instance of the containing class (i.e., there is no `this` object). A static member type does, however, have access to all the `static` members (including any other static member types) of its containing type. A static member type can use any other static member without qualifying its name with the name of the containing type.

A static member type has access to all static members of its containing type, including `private` members. The reverse is true as well: the methods of the containing type have access to all members of a static member type, including the `private` members. A static member type even has access to all the members of any other static member types, including the `private` members of those types.

Top-level types can be declared with or without the `public` modifier, but they cannot use the `private` and `protected` modifiers. Static member types, however, are members and can use any access control modifiers that other members of the containing type can. These

modifiers have the same meanings for static member types as they do for other members of a type. In [Example 3-7](#), the `Linkable` interface is declared `public`, so it can be implemented by any class that is interested in being stored on a `LinkedList`. Recall that all members of interfaces (and annotation types) are implicitly `public`, so static member types nested within interfaces or annotation types cannot be `protected` or `private`.

3.10.1.2. Restrictions on static member types

A static member type cannot have the same name as any of its enclosing classes. In addition, static member types can be defined only within top-level types and other static member types. This is actually part of a larger prohibition against `static` members of any sort within member, local, and anonymous classes.

3.10.1.3. Syntax for static member types

In code outside the containing class, a static member type is named by combining the name of the outer type with the name of the inner type (e.g., `LinkedList.Linkable`). You can use the `import` directive to import a static member type:

```
import pkg.LinkedList.Linkable; // Import a specific nested type
import pkg.LinkedList.*;       // Import all nested types of LinkedList
```

In Java 5.0 and later, you can also use the `import static` directive to import a static member type. See [Section 2.10](#) in Chapter 2 for details on `import` and `import static`. Note that importing a nested type obscures the fact that that type is closely associated with its containing type, and it is not commonly done.

3.10.2. Nonstatic Member Classes

A *nonstatic member class* is a class that is declared as a member of a containing class or enumerated type without the `static` keyword. If a static member type is analogous to a class field or class method, a nonstatic member class is analogous to an instance field or instance method. [Example 3-8](#) shows how a member class can be defined and used. This example extends the previous `LinkedList` example to allow enumeration of the elements on the stack by defining an `iterator()` method that returns an implementation of the `java.util.Iterator` interface. The implementation of this interface is defined as a member class. The example uses Java 5.0 generic type syntax in a couple of places, but this should not prevent you from understanding it. (Generics are covered in [Chapter 4](#).)

Example 3-8. An iterator implemented as a member class

```

import java.util.Iterator;

public class LinkedStack {
    // Our static member interface
    public interface Linkable {
        public Linkable getNext();
        public void setNext(Linkable node);
    }

    // The head of the list
    private Linkable head;

    // Method bodies omitted here
    public void push(Linkable node) { ... }
    public Linkable pop() { ... }

    // This method returns an Iterator object for this LinkedStack
    public Iterator<Linkable> iterator() { return new LinkedIterator(); }

    // Here is the implementation of the Iterator interface,
    // defined as a nonstatic member class.
    protected class LinkedIterator implements Iterator<Linkable> {
        Linkable current;
        // The constructor uses the private head field of the containing class
        public LinkedIterator() { current = head; }
        // The following 3 methods are defined by the Iterator interface
        public boolean hasNext() { return current != null; }
        public Linkable next() {
            if (current == null) throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }
}

```

Notice how the `LinkedIterator` class is nested within the `LinkedStack` class. Since `LinkedIterator` is a helper class used only within `LinkedStack`, there is real elegance to having it defined so close to where it is used by the containing class.

3.10.2.1. Features of member classes

Like instance fields and instance methods, every instance of a nonstatic member class is associated with an instance of the class in which it is defined. This means that the code of a member class has access to all the instance fields and instance methods (as well as the static members) of the containing class, including any that are declared `private`.

This crucial feature is illustrated in [Example 3-8](#). Here is the `LinkedStack.LinkedIterator()` constructor again:

```
public LinkedIterator() { current = head; }
```

This single line of code sets the `current` field of the inner class to the value of the `head` field of the containing class. The code works as shown, even though `head` is declared as a `private` field in the containing class.

A nonstatic member class, like any member of a class, can be assigned one of three visibility levels: `public`, `protected`, or `private`. If none of these visibility modifiers is specified, the default package visibility is used. In [Example 3-8](#), the `LinkedListIterator` class is declared `protected`, so it is inaccessible to code (in a different package) that uses the `LinkedList` class but is accessible to any class that subclasses `LinkedList`.

3.10.2.2. Restrictions on member classes

Member classes have three important restrictions:

- A nonstatic member class cannot have the same name as any containing class or package. This is an important rule, one not shared by fields and methods.
- Nonstatic member classes cannot contain any `static` fields, methods, or types, except for constant fields declared both `static` and `final`. `static` members are top-level constructs not associated with any particular object while every member class is associated with an instance of its enclosing class. Defining a `static` top-level member within a member class that is not at the top level would cause confusion, so it is not allowed.
- Only classes may be defined as nonstatic members. Interfaces, enumerated types, and annotation types are all implicitly static, even if the `static` keyword is omitted.

3.10.2.3. Syntax for member classes

The most important feature of a member class is that it can access the instance fields and methods in its containing object. We saw this in the `LinkedList.Iterator` constructor of [Example 3-8](#):

```
public LinkedList.Iterator() { current = head; }
```

In this example, `head` is a field of the `LinkedList` class, and we assign it to the `current` field of the `LinkedList.Iterator` class. What if we want to make these references explicit? We could try code like this:

```
public LinkedList.Iterator() { this.current = this.head; }
```

This code does not compile, however. `this.current` is fine; it is an explicit reference to the `current` field in the newly created `LinkedList.Iterator` object. It is the `this.head` expression that causes the problem; it refers to a field named `head` in the `LinkedList.Iterator` object. Since there is no such field, the compiler generates an error. To solve this problem, Java defines a special syntax for explicitly referring to the containing instance of the `this` object. Thus, if we want to be explicit in our constructor, we can use the following syntax:

```
public LinkedList.Iterator() { this.current = LinkedList.this.head; }
```

The general syntax is `classname.this`, where `classname` is the name of a containing class. Note that member classes can themselves contain member classes, nested to any depth. Since no member class can have the same name as any containing class, however, the use of the enclosing class name prepended to `this` is a perfectly general way to refer to any containing instance. This syntax is needed only when referring to a member of a containing class that is hidden by a member of the same name in the member class.

3.10.2.3.1. Accessing superclass members of the containing class

When a class shadows or overrides a member of its superclass, you can use the keyword `super` to refer to the hidden member. This `super` syntax can be extended to work with member classes as well. On the rare occasion when you need to refer to a shadowed field `f` or an overridden method `m` of a superclass of a containing class `C`, use the following expressions:

```
C.super.f
C.super.m()
```

3.10.2.3.2. Specifying the containing instance

As we've seen, every instance of a member class is associated with an instance of its containing class. Look again at our definition of the `iterator()` method in [Example 3-8](#):

```
public Iterator<Linkable> iterator() { return new LinkedIterator(); }
```

When a member class constructor is invoked like this, the new instance of the member class is automatically associated with the `this` object. This is what you would expect to happen and exactly what you want to occur in most cases. Occasionally, however, you may want to specify the containing instance explicitly when instantiating a member class. You can do this by preceding the `new` operator with a reference to the containing instance. Thus, the `iterator()` method shown earlier is shorthand for the following:

```
public Iterator<Linkable> iterator() { return this.new LinkedIterator(); }
```

Let's pretend we didn't define an `iterator()` method for `LinkedStack`. In this case, the code to obtain an `LinkedIterator` object for a given `LinkedStack` object might look like this:

```
LinkedStack stack = new LinkedStack(); // Create an empty stack
Iterator i = stack.new LinkedIterator(); // Create an Iterator for it
```

The containing instance implicitly specifies the containing class; it is a syntax error to explicitly specify the containing class name:

```
Iterator i = stack.new LinkedStack.LinkedIterator(); // Syntax error
```


One other special piece of Java syntax specifies an enclosing instance for a member class explicitly. Before we consider it, however, let me point out that you should rarely, if ever, need to use this syntax. It is one of the pathological cases that snuck into the language along with all the elegant features of nested types.

As strange as it may seem, it is possible for a top-level class to extend a member class. This means that the subclass does not have a containing instance, but its superclass does. When the subclass constructor invokes the superclass constructor, it must specify the containing instance. It does this by prepending the containing instance and a period to the `super` keyword. If we had not declared our `LinkedIterator` class to be a protected member of `LinkedList`, we could subclass it. Although it is not clear why we would want to do so, we could write code like the following:

```
// A top-level class that extends a member class
class SpecialIterator extends LinkedList.LinkedIterator {
    // The constructor must explicitly specify a containing instance
    // when invoking the superclass constructor.
    public SpecialIterator(LinkedList s) { s.super(); }
    // Rest of class omitted...
}
```

3.10.2.4. Scope versus inheritance

We've just noted that a top-level class can extend a member class. With the introduction of nonstatic member classes, two separate hierarchies must be considered for any class. The first is the *inheritance hierarchy*, from superclass to subclass, that defines the fields and methods a member class inherits. The second is the *containment hierarchy*, from containing class to contained class, that defines a set of fields and methods that are in the scope of (and are therefore accessible to) the member class.

The two hierarchies are entirely distinct from each other; it is important that you do not confuse them. This should not be a problem if you refrain from creating naming conflicts, where a field or method in a superclass has the same name as a field or method in a containing class. If such a naming conflict does arise, however, the inherited field or method takes precedence over the field or method of the same name in the containing class. This behavior is logical: when a class inherits a field or method, that field or method effectively becomes part of that class. Therefore, inherited fields and methods are in the scope of the class that inherits them and take precedence over fields and methods by the same name in enclosing scopes.

A good way to prevent confusion between the class hierarchy and the containment hierarchy is to avoid deep containment hierarchies. If a class is nested more than two levels deep, it is probably going to cause more confusion than it is worth. Furthermore, if a class has a deep class hierarchy (i.e., it has many ancestors), consider defining it as a top-level class rather than as a nonstatic member class.

3.10.3. Local Classes

A *local class* is declared locally within a block of Java code rather than as a member of a class. Only classes may be defined locally: interfaces, enumerated types and annotation types must be top-level or static member types. Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer of a class. Because all blocks of Java code appear within class definitions, all local classes are nested within containing classes. For this reason, local classes share many of the features of member classes. It is usually more appropriate, however, to think of them as an entirely separate kind of nested type. A local class has approximately the same relationship to a member class as a local variable has to an instance variable of a class.

The defining characteristic of a local class is that it is local to a block of code. Like a local variable, a local class is valid only within the scope defined by its enclosing block. If a member class is used only within a single method of its containing class, for example, there is usually no reason it cannot be coded as a local class rather than a member class. [Example 3-9](#) shows how we can modify the `iterator()` method of the `LinkedList` class so it defines `LinkedListIterator` as a local class instead of a member class. By doing this, we move the definition of the class even closer to where it is used and hopefully improve the clarity of the code even further. For brevity, [Example 3-9](#) shows only the `iterator()` method, not the entire `LinkedList` class that contains it.

Example 3-9. Defining and using a local class

```
// This method returns an Iterator object for this LinkedList
public Iterator<Linkable> iterator() {
    // Here's the definition of LinkedListIterator as a local class
    class LinkedListIterator implements Iterator<Linkable> {
        Linkable current;

        // The constructor uses the private head field of the containing class
        public LinkedListIterator() { current = head; }
        // The following 3 methods are defined by the Iterator interface
        public boolean hasNext() { return current != null; }
        public Linkable next() {
            if (current == null) throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }

    // Create and return an instance of the class we just defined
    return new LinkedListIterator();
}
```

3.10.3.1. Features of local classes

Local classes have the following interesting features:

- Like member classes, local classes are associated with a containing instance and can access any members, including `private` members, of the containing class.
- In addition to accessing fields defined by the containing class, local classes can access any local variables, method parameters, or exception parameters that are in the scope of the local method definition and are declared `final`.

3.10.3.2. Restrictions on local classes

Local classes are subject to the following restrictions:

- The name of a local class is defined only within the block that defines it; it can never be used outside that block. (Note however that instances of a local class created within the scope of the class can continue to exist outside of that scope. This situation is described in more detail later in this section.)
- Local classes cannot be declared `public`, `protected`, `private`, or `static`. These modifiers are for members of classes; they are not allowed with local variable declarations or local class declarations.
- Like member classes, and for the same reasons, local classes cannot contain `static` fields, methods, or classes. The only exception is for constants that are declared both `static` and `final`.
- Interfaces, enumerated types, and annotation types cannot be defined locally.
- A local class, like a member class, cannot have the same name as any of its enclosing classes.
- As noted earlier, a local class can use the local variables, method parameters, and even exception parameters that are in its scope but only if those variables or parameters are declared `final`. This is because the lifetime of an instance of a local class can be much longer than the execution of the method in which the class is defined. For this reason, a local class must have a private internal copy of all local variables it uses (these copies are automatically generated by the compiler). The only way to ensure that the local variable and the private copy are always the same is to insist that the local variable is `final`.

3.10.3.3. Syntax for local classes

In Java 1.0, only fields, methods, and classes could be declared `final`. The addition of local classes in Java 1.1 required a liberalization in the use of the `final` modifier. As of Java 1.1, `final` can be applied to local variables, method parameters, and even the exception parameter of a `catch` statement. The meaning of the `final` modifier remains the same in these new uses: once the local variable or parameter has been assigned a value, that value cannot be changed.

Instances of local classes, like instances of nonstatic member classes, have an enclosing instance that is implicitly passed to all constructors of the local class. Local classes can use the same `this` syntax as nonstatic member classes to refer explicitly to members of enclosing classes. Because local classes are never visible outside the blocks that define them, however, there is never a need to use the `new` and `super` syntax used by member classes to specify the enclosing instance explicitly.

3.10.3.4. Scope of a local class

In discussing nonstatic member classes, we saw that a member class can access any members inherited from superclasses and any members defined by its containing classes. The same is true for local classes, but local classes can also access `final` local variables and parameters. The following code illustrates the many fields and variables that may be accessible to a local class:

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
    private char c = 'c';           // Private fields visible to local class
    public static char d = 'd';
    public void createLocalObject(final char e)
    {
        final char f = 'f';
        int i = 0;                 // i not final; not usable by local class
        class Local extends B
        {
            char g = 'g';
            public void printVars()
            {
                // All of these fields and variables are accessible to this class
                System.out.println(g); // (this.g) g is a field of this class
                System.out.println(f); // f is a final local variable
                System.out.println(e); // e is a final local parameter
                System.out.println(d); // (C.this.d) d -- field of containing class
                System.out.println(c); // (C.this.c) c -- field of containing class
                System.out.println(b); // b is inherited by this class
                System.out.println(a); // a is inherited by the containing class
            }
        }
        Local l = new Local();      // Create an instance of the local class
        l.printVars();              // and call its printVars() method.
    }
}
```

3.10.3.5. Local variables, lexical scoping, and closures

A local variable is defined within a block of code that defines its scope. A local variable ceases to exist outside of its scope. Java is a *lexically scoped* language, which means that its concept of scope has to do with the way the source code is written. Any code within the curly braces that define the boundaries of a block can use local variables defined in that block.^[10]

^[10] This section covers advanced material; first-time readers may want to skip it for now and return to it later.

Lexical scoping simply defines a segment of source code within which a variable can be used. It is common, however, to think of a scope as a temporal scope—to think of a local

variable as existing from the time the Java interpreter begins executing the block until the time the interpreter exits the block. This is usually a reasonable way to think about local variables and their scope.

The introduction of local classes confuses the picture, however, because local classes can use local variables, and instances of a local class can have a lifetime much longer than the time it takes the interpreter to execute the block of code. In other words, if you create an instance of a local class, the instance does not automatically go away when the interpreter finishes executing the block that defines the class, as shown in the following code:

```
public class Weird {
    // A static member interface used below
    public static interface IntHolder { public int getValue(); }

    public static void main(String[] args) {
        IntHolder[] holders = new IntHolder[10]; // An array to hold 10 objects
        for(int i = 0; i < 10; i++) {           // Loop to fill the array up
            final int fi = i;                    // A final local variable
            class MyIntHolder implements IntHolder { // A local class
                public int getValue() { return fi; } // It uses the final variable
            }
            holders[i] = new MyIntHolder();      // Instantiate the local class
        }

        // The local class is now out of scope, so we can't use it. But we have
        // 10 valid instances of that class in our array. The local variable
        // fi is not in our scope here, but it is still in scope for the
        // getValue() method of each of those 10 objects. So call getValue()
        // for each object and print it out. This prints the digits 0 to 9.
        for(int i = 0; i < 10; i++) System.out.println(holders[i].getValue());
    }
}
```

The behavior of the previous program is pretty surprising. To make sense of it, remember that the lexical scope of the methods of a local class has nothing to do with when the interpreter enters and exits the block of code that defines the local class. Here's another way to think about it: each instance of a local class has an automatically created private copy of each of the final local variables it uses, so, in effect, it has its own private copy of the scope that existed when it was created.

The local class `MyIntHolder` is sometimes called a *closure*. In general terms, a closure is an object that saves the state of a scope and makes that scope available later. Closures are useful in some styles of programming, and different programming languages define and implement closures in different ways. Java's closures are relatively weak (and some would argue that they are not truly closures) because they retain the state of only `final` variables.

3.10.4. Anonymous Classes

An *anonymous class* is a local class without a name. An anonymous class is defined and instantiated in a single succinct expression using the `new` operator. While a local class definition is a statement in a block of Java code, an anonymous class definition is an

expression, which means that it can be included as part of a larger expression, such as a method call. In practice, anonymous classes are much more common than local classes. If you find yourself defining a short local class and then instantiating it exactly once, consider rewriting it using anonymous class syntax, which places the definition and use of the class in exactly the same place.

Consider [Example 3-10](#), which shows the `LinkedList` class implemented as an anonymous class within the `iterator()` method of the `LinkedList` class. Compare it with [Example 3-9](#), which shows the same class implemented as a local class. The generic syntax in this example is covered in [Chapter 4](#).

Example 3-10. An enumeration implemented with an anonymous class

```
public Iterator<Linkable> iterator() {
    // The anonymous class is defined as part of the return statement
    return new Iterator<Linkable>() {
        Linkable current;
        // Replace constructor with an instance initializer
        { current = head; }

        // The following 3 methods are defined by the Iterator interface
        public boolean hasNext() { return current != null; }
        public Linkable next() {
            if (current == null) throw new java.util.NoSuchElementException();
            Linkable value = current;
            current = current.getNext();
            return value;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }; // Note the required semicolon. It terminates the return statement
}
```

One common use for an anonymous class is to provide a simple implementation of an adapter class. An *adapter class* is one that defines code that is invoked by some other object. Take, for example, the `list()` method of the `java.io.File` class. This method lists the files in a directory. Before it returns the list, though, it passes the name of each file to a `FilenameFilter` object you must supply. This `FilenameFilter` object accepts or rejects each file. When you implement the `FilenameFilter` interface, you are defining an adapter class for use with the `File.list()` method. Since the body of such a class is typically quite short, it is easy to define an adapter class as an anonymous class. Here's how you can define a `FilenameFilter` class to list only those files whose names end with *.java*:

```
File f = new File("/src"); // The directory to list

// Now call the list() method with a single FilenameFilter argument
// Define and instantiate an anonymous implementation of FilenameFilter
// as part of the method invocation expression.
String[] filelist = f.list(new FilenameFilter() {
    public boolean accept(File f, String s) { return s.endsWith(".java"); }
}); // Don't forget the parenthesis and semicolon that end the method call!
```

As you can see, the syntax for defining an anonymous class and creating an instance of that class uses the `new` keyword, followed by the name of a class and a class body definition in curly braces. If the name following the `new` keyword is the name of a class, the anonymous class is a subclass of the named class. If the name following `new` specifies an interface, as in the two previous examples, the anonymous class implements that interface and extends `Object`. The syntax does not include any way to specify an `extends` clause, an `implements` clause, or a name for the class.

Because an anonymous class has no name, it is not possible to define a constructor for it within the class body. This is one of the basic restrictions on anonymous classes. Any arguments you specify between the parentheses following the superclass name in an anonymous class definition are implicitly passed to the superclass constructor. Anonymous classes are commonly used to subclass simple classes that do not take any constructor arguments, so the parentheses in the anonymous class definition syntax are often empty. In the previous examples, each anonymous class implemented an interface and extended `Object`. Since the `Object()` constructor takes no arguments, the parentheses were empty in those examples.

3.10.4.1. Features of anonymous classes

Anonymous classes allow you to define a one-shot class exactly where it is needed. Anonymous classes have all the features of local classes but use a more concise syntax that can reduce clutter in your code.

3.10.4.2. Restrictions on anonymous classes

Because an anonymous class is just a type of local class, anonymous classes and local classes share the same restrictions. An anonymous class cannot define any `static` fields, methods, or classes, except for `static final` constants. Interfaces, enumerated types, and annotation types cannot be defined anonymously. Also, like local classes, anonymous classes cannot be `public`, `private`, `protected`, or `static`.

Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class. If your class requires a constructor, you must use a local class instead. However, you can often use an instance initializer as a substitute for a constructor.

The syntax for defining an anonymous class combines definition with instantiation. Using an anonymous class instead of a local class is not appropriate if you need to create more than a single instance of the class each time the containing block is executed.

3.10.4.3. Syntax for anonymous classes

We've already seen examples of the syntax for defining and instantiating an anonymous class. We can express that syntax more formally as:


```
new class-name ( [ argument-list ] ) { class-body }
```

or:

```
new interface-name () { class-body }
```

Although they are not limited to use with anonymous classes, instance initializers were introduced into the language for this purpose. As described earlier in this chapter in [Section 3.3.4](#), an instance initializer is a block of initialization code contained within curly braces inside a class definition. The contents of all instance initializers for a class are automatically inserted into all constructors for the class, including any automatically created default constructor. An anonymous class cannot define a constructor, so it gets a default constructor. By using an instance initializer, you can get around the fact that you cannot define a constructor for an anonymous class.

3.10.4.4. When to use an anonymous class

As we've discussed, an anonymous class behaves just like a local class and is distinguished from a local class merely in the syntax used to define and instantiate it. In your own code, when you have to choose between using an anonymous class and a local class, the decision often comes down to a matter of style. You should use whichever syntax makes your code clearer. In general, you should consider using an anonymous class instead of a local class if:

- The class has a very short body.
- Only one instance of the class is needed.
- The class is used right after it is defined.
- The name of the class does not make your code any easier to understand.

3.10.4.5. Anonymous class indentation and formatting

The common indentation and formatting conventions we are familiar with for block-structured languages like Java and C begin to break down somewhat once we start placing anonymous class definitions within arbitrary expressions. Based on their experience with nested types, the engineers at Sun recommend the following formatting rules:

- The opening curly brace should not be on a line by itself; instead, it should follow the closing parenthesis of the `new` operator. Similarly, the `new` operator should, when possible, appear on the same line as the assignment or other expression of which it is a part.
- The body of the anonymous class should be indented relative to the beginning of the line that contains the `new` keyword.
- The closing curly brace of an anonymous class should not be on a line by itself either; it should be followed by whatever tokens are required by the rest of the expression.

Often this is a semicolon or a closing parenthesis followed by a semicolon. This extra punctuation serves as a flag to the reader that this is not just an ordinary block of code and makes it easier to understand anonymous classes in a code listing.

3.10.5. How Nested Types Work

The preceding sections explained the features and behavior of the four kinds of nested types. Strictly speaking, that should be all you need to know about nested types. You may find it easier to understand nested types if you understand how they are implemented, however.

Nested types were added in Java 1.1. Despite the dramatic changes to the Java language, the introduction of nested types did not change the Java Virtual Machine or the Java class file format. As far as the Java interpreter is concerned, there is no such thing as a nested type: all classes are normal top-level classes. In order to make a nested type behave as if it is actually defined inside another class, the Java compiler ends up inserting hidden fields, methods, and constructor arguments into the classes it generates. You may want to use the *javap* disassembler to disassemble some of the class files for nested types so you can see what tricks the compiler has used to make the nested types work. (See [Chapter 8](#) for information on *javap*.)

3.10.5.1. Static member type implementation

Recall our first `LinkedList` example ([Example 3-7](#)), which defined a static member interface named `Linkable`. When you compile this `LinkedList` class, the compiler actually generates two class files. The first one is `LinkedList.class`, as expected. The second class file, however, is called `LinkedList$Linkable.class`. The `$` in this name is automatically inserted by the Java compiler. This second class file contains the implementation of the static member interface.

As we discussed earlier, a static member type can access all the `static` members of its containing class. If a static member type does this, the compiler automatically qualifies the member access expression with the name of the containing class. A static member type is even allowed to access the `private static` fields of its containing class. Since the static member type is compiled into an ordinary top-level class, however, there is no way it can directly access the `private` members of its container. Therefore, if a static member type uses a `private` member of its containing type (or vice versa), the compiler generates synthetic non-`private` access methods and converts the expressions that access the `private` members into expressions that invoke these specially generated methods. These methods are given the default package access, which is sufficient, as the member class and its containing class are guaranteed to be in the same package.

3.10.5.2. Nonstatic member class implementation

A nonstatic member class is implemented much like a static member type. It is compiled into a separate top-level class file, and the compiler performs various code manipulations to make interclass member access work correctly.

The most significant difference between a nonstatic member class and a static member type is that each instance of a nonstatic member class is associated with an instance of the enclosing class. The compiler enforces this association by defining a synthetic field named `this$0` in each member class. This field is used to hold a reference to the enclosing instance. Every nonstatic member class constructor is given an extra parameter that initializes this field. Every time a member class constructor is invoked, the compiler automatically passes a reference to the enclosing class for this extra parameter.

As we've seen, a nonstatic member class, like any member of a class, can be declared `public`, `protected`, or `private`, or given the default package visibility. Member classes are compiled to class files just like top-level classes, but top-level classes can have only `public` or package access. Therefore, as far as the Java interpreter is concerned, member classes can have only `public` or package visibility. This means that a member class declared `protected` is actually treated as a public class, and a member class declared `private` actually has package visibility. This does not mean you should never declare a member class as `protected` or `private`. Although the Java VM cannot enforce these access control modifiers, the modifiers are stored in the class file and conforming Java compilers do enforce them.

3.10.5.3. Local and anonymous class implementation

A local class is able to refer to fields and methods in its containing class for exactly the same reason that a nonstatic member class can; it is passed a hidden reference to the containing class in its constructor and saves that reference away in a `private` synthetic field added by the compiler. Also, like nonstatic member classes, local classes can use `private` fields and methods of their containing class because the compiler inserts any required accessor methods.

What makes local classes different from member classes is that they have the ability to refer to local variables in the scope that defines them. The crucial restriction on this ability, however, is that local classes can reference only local variables and parameters that are declared `final`. The reason for this restriction becomes apparent in the implementation. A local class can use local variables because the compiler automatically gives the class a `private` instance field to hold a copy of each local variable the class uses. The compiler also adds hidden parameters to each local class constructor to initialize these automatically created `private` fields. A local class does not actually access local variables but merely its own `private` copies of them. The only way this can work correctly is if the local variables

are declared `final` so that they are guaranteed not to change. With this guarantee, the local class can be assured that its internal copies of the variables are always in sync with the real local variables.

Since anonymous classes have no names, you may wonder what the class files that represent them are named. This is an implementation detail, but Sun's Java compiler uses numbers to provide anonymous class names. If you compile the example code shown in [Example 3-10](#), you'll find that it produces a class file for the anonymous class with a name like *LinkedStack\$1.class*.

3.11. Modifier Summary

As we've seen, classes, interfaces, and their members can be declared with one or more *modifiers*—keywords such as `public`, `static`, and `final`. [Table 3-2](#) lists the Java modifiers, explains what types of Java constructs they can modify, and explains what they do. See also [Section 3.1](#) and [Section 3.2.1](#) earlier in this chapter, as well as [Section 2.6.2](#) in Chapter 2.

Table 3-2. Java modifiers

Modifier	Used on	Meaning
abstract	Class	The class contains unimplemented methods and cannot be instantiated.
	Interface	All interfaces are <code>abstract</code> . The modifier is optional in interface declarations.
abstract	Method	No body is provided for the method; it is provided by a subclass. The signature is followed by a semicolon. The enclosing class must also be <code>abstract</code> .
final	Class	The class cannot be subclassed.
	Method	The method cannot be overridden (and is not subject to dynamic method lookup).
	Field	The field cannot have its value changed. <code>static final</code> fields are compile-time constants.
	Variable	A local variable, method parameter, or exception parameter cannot have its value changed. Useful with local classes.
native	Method	The method is implemented in some platform-dependent way (often in C). No body is provided; the signature is followed by a semicolon.
None (package)	Class	A <code>non-public</code> class is accessible only in its package.
	Interface	A <code>non-public</code> interface is accessible only in its package.
	Member	A member that is not <code>private</code> , <code>protected</code> , or <code>public</code> has package visibility and is accessible only within its package.
private	Member	The member is accessible only within the class that defines it.
protected	Member	The member is accessible only within the package in which it is defined and within subclasses.
public	Class	The class is accessible anywhere its package is.
	Interface	The interface is accessible anywhere its package is.
	Member	The member is accessible anywhere its class is.
strictfp	Class	All methods of the class are implicitly <code>strictfp</code> .
strictfp	Method	All floating-point computation done by the method must be performed in a way that strictly conforms to the IEEE 754 standard. In particular, all values, including intermediate results, must be expressed as IEEE <code>float</code> or <code>double</code> values and cannot take advantage of any extra precision or range offered by native platform floating-point formats or hardware. This modifier is rarely used.

Modifier	Used on	Meaning
<code>static</code>	Class	An inner class declared <code>static</code> is a top-level class, not associated with a member of the containing class.
	Method	A <code>static</code> method is a class method. It is not passed an implicit <code>this</code> object reference. It can be invoked through the class name.
	Field	A <code>static</code> field is a class field. There is only one instance of the field, regardless of the number of class instances created. It can be accessed through the class name.
	Initializer	The initializer is run when the class is loaded rather than when an instance is created.
<code>synchronized</code>	Method	The method makes nonatomic modifications to the class or instance, so care must be taken to ensure that two threads cannot modify the class or instance at the same time. For a <code>static</code> method, a lock for the class is acquired before executing the method. For a non- <code>static</code> method, a lock for the specific object instance is acquired.
<code>transient</code>	Field	The field is not part of the persistent state of the object and should not be serialized with the object. Used with object serialization; see <code>java.io.ObjectOutputStream</code> .
<code>volatile</code>	Field	The field can be accessed by unsynchronized threads, so certain optimizations must not be performed on it. This modifier can sometimes be used as an alternative to <code>synchronized</code> . This modifier is very rarely used.

3.12. C++ Features Not Found in Java

This chapter indicates similarities and differences between Java and C++ in footnotes. Java shares enough concepts and features with C++ to make it an easy language for C++ programmers to pick up. Several features of C++ have no parallel in Java, however. In general, Java does not adopt those features of C++ that make the language significantly more complicated.

C++ supports multiple inheritance of method implementations from more than one superclass at a time. While this seems like a useful feature, it actually introduces many complexities to the language. The Java language designers chose to avoid the added complexity by using interfaces instead. Thus, a class in Java can inherit method implementations only from a single superclass, but it can inherit method declarations from any number of interfaces.

C++ supports templates that allow you, for example, to implement a `Stack` class and then instantiate it as `Stack<int>` or `Stack<double>` to produce two separate types: a stack of integers and a stack of floating-point values. Java 5.0 introduces parameterized types or "generics" that provide similar functionality in a more robust fashion. Generics are covered in [Chapter 4](#).

C++ allows you to define operators that perform arbitrary operations on instances of your classes. In effect, it allows you to extend the syntax of the language. This is a nifty feature, called operator overloading, that makes for elegant examples. In practice, however, it tends to make code quite difficult to understand. After much debate, the Java language designers decided to omit such operator overloading from the language. Note, though, that the use

of the + operator for string concatenation in Java is at least reminiscent of operator overloading.

C++ allows you to define conversion functions for a class that automatically invokes an appropriate constructor method when a value is assigned to a variable of that class. This is simply a syntactic shortcut (similar to overriding the assignment operator) and is not included in Java.

In C++, objects are manipulated by value by default; you must use & to specify a variable or function argument automatically manipulated by reference. In Java, all objects are manipulated by reference, so there is no need for the & syntax.