

Chapter 6

A Closer Look at Methods and Classes



Key Skills & Concepts

- Control access to members
- Pass objects to a method
- Return objects from a method
- Overload methods
- Overload constructors
- Use recursion

- Apply **static**
 - Use inner classes
 - Use varargs
-

This chapter resumes our examination of classes and methods. It begins by explaining how to control access to the members of a class. It then discusses the passing and returning of objects, method overloading, recursion, and the use of the keyword **static**. Also described are nested classes and variable-length arguments.

Controlling Access to Class Members

In its support for encapsulation, the class provides two major benefits. First, it links data with the code that manipulates it. You have been taking advantage of this aspect of the class since [Chapter 4](#). Second, it provides the means by which access to members can be controlled. It is this feature that is examined here.

Although Java's approach is a bit more sophisticated, in essence, there are two basic types of class members: public and private. A public member can be freely accessed by code defined outside of its class. A private member can be accessed only by other methods defined by its class. It is through the use of private members that access is controlled.

Restricting access to a class' members is a fundamental part of object-oriented programming because it helps prevent the misuse of an object. By allowing access to private data only through a well-defined set of methods, you can prevent improper values from being assigned to that data—by performing a range check, for example. It is not possible for code outside the class to set the value of a private member directly. You can also control precisely how and when the data within an object is used. Thus, when correctly implemented, a class creates a “black box” that can be used, but the inner workings of which are not open to tampering.

Up to this point, you haven't had to worry about access control because Java provides a default access setting in which, for the types of programs shown earlier, the members of a class are freely available to the other code in the program. (Thus, for the preceding examples, the default access setting is essentially public.) Although convenient for simple classes (and example programs in books such as this one), this default setting is inadequate for many real-world situations. Here we introduce Java's other access control features.

Java's Access Modifiers

Member access control is achieved through the use of three *access modifiers*: **public**, **private**, and **protected**. As explained, if no access modifier is used, the default access setting is assumed. In this chapter, we will be concerned with **public** and **private**. The **protected** modifier applies only when inheritance is involved and is described in [Chapter 8](#).

When a member of a class is modified by the **public** specifier, that member can be

accessed by any other code in your program. This includes by methods defined inside other classes.

When a member of a class is specified as **private**, that member can be accessed only by other members of its class. Thus, methods in other classes cannot access a **private** member of another class.

The default access setting (in which no access modifier is used) is the same as **public** unless your program is broken down into packages. A *package* is, essentially, a grouping of classes. Packages are both an organizational and an access control feature, but a discussion of packages must wait until [Chapter 8](#). For the types of programs shown in this and the preceding chapters, **public** access is the same as default access.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here are some examples:

```
public String errMsg;  
private accountBalance bal;  
  
private boolean isError(byte status) { // ...
```

To understand the effects of **public** and **private**, consider the following program:

```
// Public vs private access.  
class MyClass {  
    private int alpha; // private access  
    public int beta; // public access  
    int gamma; // default access  
  
    /* Methods to access alpha. It is OK for a  
       member of a class to access a private member  
       of the same class.  
    */  
    void setAlpha(int a) {  
        alpha = a;  
    }  
}
```

```

    int getAlpha() {
        return alpha;
    }
}

class AccessDemo {
    public static void main(String[] args) {
        MyClass ob = new MyClass();

        /* Access to alpha is allowed only through
           its accessor methods. */
        ob.setAlpha(-99);
        System.out.println("ob.alpha is " + ob.getAlpha());

        // You cannot access alpha like this:
        // ob.alpha = 10; // Wrong! alpha is private! ← Wrong—alpha is private!

        // These are OK because beta and gamma are public.
        ob.beta = 88; ← OK because these are public.
        ob.gamma = 99;
    }
}

```

As you can see, inside the **MyClass** class, **alpha** is specified as **private**, **beta** is explicitly specified as **public**, and **gamma** uses the default access, which for this example is the same as specifying **public**. Because **alpha** is private, it cannot be accessed by code outside of its class. Therefore, inside the **AccessDemo** class, **alpha** cannot be used directly. It must be accessed through its public accessor methods: **setAlpha()** and **getAlpha()**. If you were to remove the comment symbol from the beginning of the following line,

```
// ob.alpha = 10; // Wrong! alpha is private!
```

you would not be able to compile this program because of the access violation. Although access to **alpha** by code outside of **MyClass** is not allowed, methods defined within **MyClass** can freely access it, as the **setAlpha()** and **getAlpha()** methods show.

The key point is this: A private member can be used freely by other members of its class, but it cannot be accessed by code outside its class.

To see how access control can be applied to a more practical example, consider the following program that implements a “fail-soft” **int** array, in which boundary errors are prevented, thus avoiding a run-time exception from being generated. This is accomplished by encapsulating the array as a private member of a class, allowing access to the array only through member methods. With this approach, any attempt to access the array beyond its boundaries can be prevented, with such an attempt failing gracefully (resulting in a “soft” landing rather than a “crash”). The fail-soft array is implemented by the **FailSoftArray** class, shown here:

```
/* This class implements a "fail-soft" array which prevents  
runtime errors.  
*/
```

```

class FailSoftArray {
    private int[] a; // reference to array
    private int errval; // value to return if get() fails
    public int length; // length is public

    /* Construct array given its size and the value to
       return if get() fails. */
    public FailSoftArray(int size, int errv) {
        a = new int[size];
        errval = errv;
        length = size;
    }

    // Return value at given index.
    public int get(int index) {
        if(indexOK(index)) return a[index]; ← Trap on out-of-bounds index.
        return errval;
    }

    // Put a value at an index. Return false on failure.
    public boolean put(int index, int val) {
        if(indexOK(index)) { ←
            a[index] = val;
            return true;
        }
        return false;
    }

    // Return true if index is within bounds.
    private boolean indexOK(int index) {
        if(index >= 0 & index < length) return true;
        return false;
    }
}

// Demonstrate the fail-soft array.
class FSDemo {
    public static void main(String[] args) {
        FailSoftArray fs = new FailSoftArray(5, -1);
        int x;

        // show quiet failures
        System.out.println("Fail quietly.");
        for(int i=0; i < (fs.length * 2); i++)
            fs.put(i, i*10); ← Access to array must be through its accessor methods.

        for(int i=0; i < (fs.length * 2); i++) {
            x = fs.get(i); ←
        }
    }
}

```



```

        if(x != -1) System.out.print(x + " ");
    }
    System.out.println("");

    // now, handle failures
    System.out.println("\nFail with error reports.");
    for(int i=0; i < (fs.length * 2); i++)
        if(!fs.put(i, i*10))
            System.out.println("Index " + i + " out-of-bounds");

    for(int i=0; i < (fs.length * 2); i++) {
        x = fs.get(i);
        if(x != -1) System.out.print(x + " ");
        else
            System.out.println("Index " + i + " out-of-bounds");
    }
}
}

```

The output from the program is shown here:

```

Fail quietly.
0 10 20 30 40

```

```

Fail with error reports.
Index 5 out-of-bounds
Index 6 out-of-bounds
Index 7 out-of-bounds
Index 8 out-of-bounds
Index 9 out-of-bounds
0 10 20 30 40 Index 5 out-of-bounds
Index 6 out-of-bounds
Index 7 out-of-bounds
Index 8 out-of-bounds
Index 9 out-of-bounds

```

Let's look closely at this example. Inside **FailSoftArray** are defined three **private** members. The first is **a**, which stores a reference to the array that will actually hold information. The second is **errval**, which is the value that will be returned when a call to **get()** fails. The third is the **private** method **indexOK()**, which determines whether an index is within bounds. Thus, these three members can be used only by other members of the **FailSoftArray** class. Specifically, **a** and **errval** can be used only by other methods in the class, and **indexOK()** can be called only by other members of **FailSoftArray**. The rest of the class members are **public** and can be called by any other code in a program that uses **FailSoftArray**.

When a **FailSoftArray** object is constructed, you must specify the size of the array and the value that you want to return if a call to **get()** fails. The error value must be a value that would otherwise not be stored in the array. Once constructed, the actual array referred to by **a** and the error value stored in **errval** cannot be accessed by users of the **FailSoftArray** object. Thus, they are not open to misuse. For example, the user cannot try to index **a** directly, possibly exceeding its bounds. Access is available only through the **get()** and **put()** methods.

The **indexOK()** method is **private** mostly for the sake of illustration. It would be harmless to make it **public** because it does not modify the object. However, since it is used internally by the **FailSoftArray** class, it can be **private**.

Notice that the **length** instance variable is **public**. This is in keeping with the way Java implements arrays. To obtain the length of a **FailSoftArray**, simply use its **length** member.

To use a **FailSoftArray** array, call **put()** to store a value at the specified index. Call **get()** to retrieve a value from a specified index. If the index is out-of-bounds, **put()** returns **false** and **get()** returns **errval**.

For the sake of convenience, the majority of the examples in this book will continue to use default access for most members. Remember, however, that in the real world, restricting access to members—especially instance variables—is an important part of successful object-oriented programming. As you will see in [Chapter 7](#), access control is even more vital when inheritance is involved.

NOTE

The modules feature added by JDK 9 can also play a role in accessibility. Modules are discussed in [Chapter 15](#).

Try This 6-1 Improving the Queue Class

`Queue.java`

You can use the **private** modifier to make a rather important improvement to the **Queue** class developed in [Chapter 5](#), Try This 5-2. In that version, all members of the **Queue** class use the default access. This means that it would be possible for a program that uses a **Queue** to directly access the underlying array, possibly accessing its elements out of turn. Since the entire point of a queue is to provide a first-in, first-out list, allowing out-of-order access is not desirable. It would also be possible for a malicious programmer to alter the values stored in the **putloc** and **getloc** indices, thus corrupting the queue. Fortunately, these types of problems are easy to prevent by applying the **private** specifier.

1. Copy the original **Queue** class in Try This 5-2 to a new file called **Queue.java**.
2. In the **Queue** class, add the **private** modifier to the **q** array, and the indices **putloc** and **getloc**, as shown here:


```
// An improved queue class for characters.
class Queue {
    // these members are now private
    private char[] q; // this array holds the queue
    private int putloc, getloc; // the put and get indices

    Queue(int size) {
        q = new char[size]; // allocate memory for queue
        putloc = getloc = 0;
    }

    // Put a character into the queue.
    void put(char ch) {
        if(putloc==q.length) {
            System.out.println(" - Queue is full.");
            return;
        }

        q[putloc++] = ch;
    }

    // Get a character from the queue.
    char get() {
        if(getloc == putloc) {
            System.out.println(" - Queue is empty.");
            return (char) 0;
        }

        return q[getloc++];
    }
}
```

3. Changing **q**, **putloc**, and **getloc** from default access to private access has no effect on a program that properly uses **Queue**. For example, it still works fine with the **QDemo** class from Try This 5-2. However, it prevents the improper use of a **Queue**. For example, the following types of statements are illegal:

```
Queue test = new Queue(10);

test.q[0] = 99; // wrong!
test.putloc = -100; // won't work!
```

4. Now that **q**, **putloc**, and **getloc** are private, the **Queue** class strictly enforces the first-in, first-out attribute of a queue.

Pass Objects to Methods

Up to this point, the examples in this book have been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, the following program defines a class called **Block** that stores the dimensions of a three-dimensional block:

```
// Objects can be passed to methods.
class Block {
    int a, b, c;
    int volume;
```

```

Block(int i, int j, int k) {
    a = i;
    b = j;
    c = k;
    volume = a * b * c;
}

// Return true if ob defines same block.
boolean sameBlock(Block ob) { ← Use object type for parameter.
    if((ob.a == a) & (ob.b == b) & (ob.c == c)) return true;
    else return false;
}

// Return true if ob has same volume.
boolean sameVolume(Block ob) { ←
    if(ob.volume == volume) return true;
    else return false;
}
}

class PassOb {
    public static void main(String[] args) {
        Block ob1 = new Block(10, 2, 5);
        Block ob2 = new Block(10, 2, 5);
        Block ob3 = new Block(4, 5, 5);

        System.out.println("ob1 same dimensions as ob2: " +
                           ob1.sameBlock(ob2)); ← Pass an object.
        System.out.println("ob1 same dimensions as ob3: " +
                           ob1.sameBlock(ob3)); ←
        System.out.println("ob1 same volume as ob3: " +
                           ob1.sameVolume(ob3)); ←
    }
}

```

This program generates the following output:

```

ob1 same dimensions as ob2: true
ob1 same dimensions as ob3: false
ob1 same volume as ob3: true

```

The **sameBlock()** and **sameVolume()** methods compare the **Block** object passed as a parameter to the invoking object. For **sameBlock()**, the dimensions of the objects are compared and **true** is returned only if the two blocks are the same. For **sameVolume()**, the two blocks are compared only to determine whether they have the same volume. In both cases, notice that the parameter **ob** specifies **Block** as its type. Although **Block** is a class type created by the program, it is used in the same way as Java's built-in types.

How Arguments Are Passed

As the preceding example demonstrated, passing an object to a method is a straightforward task. However, there are some nuances of passing an object that are not shown in the example. In certain cases, the effects of passing an object will be different from those experienced when passing non-object arguments. To see why, you need to understand in a general sense the two ways in which an argument can be passed to a subroutine.

The first way is *call-by-value*. This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument in the call. The second way an argument can be passed is *call-by-reference*. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter *will* affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type, such as **int** or **double**, to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
// Primitive types are passed by value.
class Test {
    /* This method causes no change to the arguments
       used in the call. */
    void noChange(int i, int j) {
        i = i + j;
        j = -j;
    }
}

class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a and b before call: " +
                           a + " " + b);

        ob.noChange(a, b);

        System.out.println("a and b after call: " +
                           a + " " + b);
    }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **noChange()** have no effect on the values of **a** and **b** used in the call.

When you pass an object to a method, the situation changes dramatically, because objects are implicitly passed by reference. Keep in mind that when you create a variable of a class type, you are creating a reference to an object. It is the reference, not the object itself, that is actually passed to the method. As a result, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }
    /* Pass an object. Now, ob.a and ob.b in object
       used in the call will be changed. */
    void change(Test ob) {
        ob.a = ob.a + ob.b;
        ob.b = -ob.b;
    }
}

class PassObRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);

        System.out.println("ob.a and ob.b before call: " +
                           ob.a + " " + ob.b);

        ob.change(ob);

        System.out.println("ob.a and ob.b after call: " +
                           ob.a + " " + ob.b);
    }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 35 -20
```

As you can see, in this case, the actions inside **change()** have affected the object used as an argument.

Ask the Expert

Q: Is there any way that I can pass a primitive type by reference?

A: Not directly. However, Java defines a set of classes that *wrap* the primitive types in

objects. These are **Double**, **Float**, **Byte**, **Short**, **Integer**, **Long**, and **Character**. In addition to allowing a primitive type to be passed by reference, these wrapper classes define several methods that enable you to manipulate their values. For example, the numeric type wrappers include methods that convert a numeric value from its binary form into its human-readable **String** form, and vice versa.

Remember, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object referred to by its corresponding argument.

Returning Objects

A method can return any type of data, including class types. For example, the class **ErrorMsg** shown here could be used to report errors. Its method, **getErrorMsg()**, returns a **String** object that contains a description of an error based upon the error code that it is passed.

```
// Return a String object.
class ErrorMsg {
    String[] msgs = {
        "Output Error",
        "Input Error",
        "Disk Full",
        "Index Out-Of-Bounds"
    };

    // Return the error message.
    String getErrorMsg(int i) { ← Return an object of type String.
        if(i >=0 & i < msgs.length)
            return msgs[i];
        else
            return "Invalid Error Code";
    }
}

class ErrMsg {
    public static void main(String[] args) {
        ErrorMsg err = new ErrorMsg();

        System.out.println(err.getErrorMsg(2));
        System.out.println(err.getErrorMsg(19));
    }
}
```

Its output is shown here:

```
Disk Full
Invalid Error Code
```

You can, of course, also return objects of classes that you create. For example, here is a reworked version of the preceding program that creates two error classes. One is called **Err**, and it encapsulates an error message along with a severity code. The second is called **ErrorInfo**. It defines a method called **getErrorInfo()**, which returns an **Err** object.

```
// Return a programmer-defined object.
class Err {
    String msg; // error message
    int severity; // code indicating severity of error

    Err(String m, int s) {
        msg = m;
        severity = s;
    }
}

class ErrorInfo {
    String[] msgs = {
        "Output Error",
        "Input Error",
        "Disk Full",
        "Index Out-Of-Bounds"
    };
    int[] howBad = { 3, 3, 2, 4 };

    Err getErrorInfo(int i) { ← Return an object of type Err.
        if(i >= 0 & i < msgs.length)
            return new Err(msgs[i], howBad[i]);
        else
            return new Err("Invalid Error Code", 0);
    }
}

class ErrInfo {
    public static void main(String[] args) {
        ErrorInfo err = new ErrorInfo();
        Err e;
```

```

    e = err.getErrorInfo(2);
    System.out.println(e.msg + " severity: " + e.severity);

    e = err.getErrorInfo(19);
    System.out.println(e.msg + " severity: " + e.severity);
}
}

```

Here is the output:

```

Disk Full severity: 2
Invalid Error Code severity: 0

```

Each time **getErrorInfo()** is invoked, a new **Err** object is created, and a reference to it is returned to the calling routine. This object is then used within **main()** to display the error message and severity code.

When an object is returned by a method, it remains in existence until there are no more references to it. At that point, it is subject to garbage collection. Thus, an object won't be destroyed just because the method that created it terminates.

Method Overloading

In this section, you will learn about one of Java's most exciting features: method overloading. In Java, two or more methods within the same class can share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java implements polymorphism.

In general, to overload a method, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction: the type and/or number of the parameters of each overloaded method must differ. It is not sufficient for two methods to differ only in their return types. (Return types do not provide sufficient information in all cases for Java to decide which method to use.) Of course, overloaded methods *may* differ in their return types, too. When an overloaded method is called, the version of the method whose parameters match the arguments is executed.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.
class Overload {
    void ovlDemo() { ←————— First version
        System.out.println("No parameters");
    }

    // Overload ovlDemo for one integer parameter.
    void ovlDemo(int a) { ←————— Second version
        System.out.println("One parameter: " + a);
    }
}
```

```

// Overload ovlDemo for two integer parameters.
int ovlDemo(int a, int b) { ←————— Third version
    System.out.println("Two parameters: " + a + " " + b);
    return a + b;
}

// Overload ovlDemo for two double parameters.
double ovlDemo(double a, double b) { ←————— Fourth version
    System.out.println("Two double parameters: " +
        a + " " + b);
    return a + b;
}

}

class OverloadDemo {
    public static void main(String[] args) {
        Overload ob = new Overload();
        int resI;
        double resD;

        // call all versions of ovlDemo()
        ob.ovlDemo();
        System.out.println();

        ob.ovlDemo(2);
        System.out.println();

        resI = ob.ovlDemo(4, 6);
        System.out.println("Result of ob.ovlDemo(4, 6): " +
            resI);
        System.out.println();

        resD = ob.ovlDemo(1.1, 2.32);
        System.out.println("Result of ob.ovlDemo(1.1, 2.32): " +
            resD);
    }
}

```

This program generates the following output:

No parameters

One parameter: 2

Two parameters: 4 6

Result of ob.ovlDemo(4, 6): 10

Two double parameters: 1.1 2.32

Result of ob.ovlDemo(1.1, 2.32): 3.42

As you can see, **ovlDemo()** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes two **double** parameters. Notice that the first two versions of **ovlDemo()** return **void**, and the second two return a value. This is perfectly valid, but as explained, overloading is not affected one way or the other by the return type of a method. Thus, attempting to use the following two versions of **ovlDemo()** will cause an error:

```
// One ovlDemo(int) is OK.
void ovlDemo(int a) {
    System.out.println("One parameter: " + a);
}

/* Error! Two ovlDemo(int)s are not OK even though
   return types differ.
*/
int ovlDemo(int a) {
    System.out.println("One parameter: " + a);
    return a * a;
}
```

Return types cannot be used to differentiate overloaded methods.

As the comments suggest, the difference in their return types is insufficient for the purposes of overloading.

As you will recall from [Chapter 2](#), Java provides certain automatic type conversions. These conversions also apply to parameters of overloaded methods. For example, consider the following:


```

/* Automatic type conversions can affect
   overloaded method resolution.
*/
class Overload2 {
    void f(int x) {
        System.out.println("Inside f(int): " + x);
    }

    void f(double x) {
        System.out.println("Inside f(double): " + x);
    }
}

class TypeConv {
    public static void main(String[] args) {
        Overload2 ob = new Overload2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.f(i); // calls ob.f(int)
        ob.f(d); // calls ob.f(double)

        ob.f(b); // calls ob.f(int) - type conversion
        ob.f(s); // calls ob.f(int) - type conversion
        ob.f(f); // calls ob.f(double) - type conversion
    }
}

```

The output from the program is shown here:

```

Inside f(int): 10
Inside f(double): 10.1
Inside f(int): 99
Inside f(int): 10
Inside f(double): 11.5

```

In this example, only two versions of **f()** are defined: one that has an **int** parameter and one that has a **double** parameter. However, it is possible to pass **f()** a **byte**, **short**, or **float** value. In the case of **byte** and **short**, Java automatically converts them to **int**. Thus, **f(int)** is invoked. In the case of **float**, the value is converted to **double** and **f(double)** is called.

It is important to understand, however, that the automatic conversions apply only if there is no direct match between a parameter and an argument. For example, here is the preceding program with the addition of a version of **f()** that specifies a **byte** parameter:

```
// Add f(byte).
class Overload2 {
    void f(byte x) { ←————— This version specifies
        System.out.println("Inside f(byte): " + x);      a byte parameter.
    }

    void f(int x) {
        System.out.println("Inside f(int): " + x);
    }

    void f(double x) {
        System.out.println("Inside f(double): " + x);
    }
}

class TypeConv {
    public static void main(String[] args) {
        Overload2 ob = new Overload2();

        int i = 10;
        double d = 10.1;

        byte b = 99;
        short s = 10;
        float f = 11.5F;

        ob.f(i); // calls ob.f(int)
        ob.f(d); // calls ob.f(double)

        ob.f(b); // calls ob.f(byte) - now, no type conversion

        ob.f(s); // calls ob.f(int) - type conversion
        ob.f(f); // calls ob.f(double) - type conversion
    }
}
```

Now when the program is run, the following output is produced:

```
Inside f(int): 10
Inside f(double): 10.1
Inside f(byte): 99
Inside f(int): 10
Inside f(double): 11.5
```

In this version, since there is a version of **f()** that takes a **byte** argument, when **f()** is called with a **byte** argument, **f(byte)** is invoked and the automatic conversion to **int** does not occur.

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. To understand how, consider the following: In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function **abs()** returns the absolute value of an integer, **labs()** returns the absolute value of a long integer, and **fabs()** returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java’s standard class library includes an absolute value method, called **abs()**. This method is overloaded by Java’s **Math** class to handle all of the numeric types. Java determines which version of **abs()** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* that is being performed. It is left to the compiler to choose the correct *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should overload only closely related operations.

Ask the Expert

Q: I've heard the term *signature* used by Java programmers. What is it?

A: As it applies to Java, a signature is the name of a method plus its parameter list. Thus, for the purposes of overloading, no two methods within the same class can have the same signature. Notice that a signature does not include the return type, since it is not used by Java for overload resolution.

Overloading Constructors

Like methods, constructors can also be overloaded. Doing so allows you to construct objects in a variety of ways. For example, consider the following program:

```
// Demonstrate an overloaded constructor.
class MyClass {
    int x;

    MyClass() { ←————— Construct objects in a variety of ways.
        System.out.println("Inside MyClass().");
        x = 0;
    }

    MyClass(int i) { ←—————
        System.out.println("Inside MyClass(int).");
        x = i;
    }

    MyClass(double d) { ←—————
        System.out.println("Inside MyClass(double).");
        x = (int) d;
    }

    MyClass(int i, int j) { ←—————
        System.out.println("Inside MyClass(int, int).");
        x = i * j;
    }
}
```

```

class OverloadConsDemo {
    public static void main(String[] args) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass(88);
        MyClass t3 = new MyClass(17.23);
        MyClass t4 = new MyClass(2, 4);

        System.out.println("t1.x: " + t1.x);
        System.out.println("t2.x: " + t2.x);
        System.out.println("t3.x: " + t3.x);
        System.out.println("t4.x: " + t4.x);
    }
}

```

The output from the program is shown here:

```

Inside MyClass().
Inside MyClass(int).
Inside MyClass(double).
Inside MyClass(int, int).
t1.x: 0
t2.x: 88
t3.x: 17
t4.x: 8

```

MyClass() is overloaded four ways, each constructing an object differently. The proper constructor is called based upon the parameters specified when **new** is executed. By overloading a class' constructor, you give the user of your class flexibility in the way objects are constructed.

One of the most common reasons that constructors are overloaded is to allow one object to initialize another. For example, consider this program that uses the **Summation** class to compute the summation of an integer value:

```
// Initialize one object with another.
class Summation {
    int sum;

    // Construct from an int.
    Summation(int num) {
        sum = 0;
        for(int i=1; i <= num; i++)
            sum += i;
    }

    // Construct from another object.
    Summation(Summation ob) { ← Construct one object from another.
        sum = ob.sum;
    }
}

class SumDemo {
    public static void main(String[] args) {
        Summation s1 = new Summation(5);
        Summation s2 = new Summation(s1);

        System.out.println("s1.sum: " + s1.sum);
        System.out.println("s2.sum: " + s2.sum);
    }
}
```

The output is shown here:

```
s1.sum: 15
s2.sum: 15
```

Often, as this example shows, an advantage of providing a constructor that uses one object to initialize another is efficiency. In this case, when `s2` is constructed, it is not necessary to recompute the summation. Of course, even in cases when efficiency is not an issue, it is often useful to provide a constructor that makes a copy of an object.

Try This 6-2 Overloading the Queue Constructor

`QDemo2.java`

In this project, you will enhance the **Queue** class by giving it two additional constructors. The first will construct a new queue from another queue. The second will construct a queue, giving it initial values. As you will see, adding these constructors enhances the usability of **Queue** substantially.

1. Create a file called **QDemo2.java** and copy the updated **Queue** class from Try This 6-1 into it.
2. First, add the following constructor, which constructs a queue from a queue.

```
// Construct a Queue from a Queue.
Queue(Queue ob) {
    putloc = ob.putloc;
    getloc = ob.getloc;
    q = new char[ob.q.length];

    // copy elements
    for(int i=getloc; i < putloc; i++)
        q[i] = ob.q[i];
}
```

Look closely at this constructor. It initializes **putloc** and **getloc** to the values contained in the **ob** parameter. It then allocates a new array to hold the queue and copies the elements from **ob** into that array. Once constructed, the new queue will be an identical copy of the original, but both will be completely separate objects.

3. Now add the constructor that initializes the queue from a character array, as shown here:

```
// Construct a Queue with initial values.
Queue(char[] a) {
    putloc = 0;
    getloc = 0;
    q = new char[a.length];

    for(int i = 0; i < a.length; i++) put(a[i]);
}
```

This constructor creates a queue large enough to hold the characters in **a** and then stores those characters in the queue.

4. Here is the complete updated **Queue** class along with the **QDemo2** class, which demonstrates it:

```

// A queue class for characters.
class Queue {
    private char[] q; // this array holds the queue
    private int putloc, getloc; // the put and get indices

    // Construct an empty Queue given its size.
    Queue(int size) {
        q = new char[size]; // allocate memory for queue
        putloc = getloc = 0;
    }

    // Construct a Queue from a Queue.
    Queue(Queue ob) {
        putloc = ob.putloc;
        getloc = ob.getloc;
        q = new char[ob.q.length];

        // copy elements
        for(int i=getloc; i < putloc; i++)
            q[i] = ob.q[i];
    }

    // Construct a Queue with initial values.
    Queue(char[] a) {
        putloc = 0;
        getloc = 0;
        q = new char[a.length];

        for(int i = 0; i < a.length; i++) put(a[i]);
    }
}

```

```

// Put a character into the queue.
void put(char ch) {
    if(putloc==q.length) {
        System.out.println(" - Queue is full.");
        return;
    }

    q[putloc++] = ch;
}

// Get a character from the queue.
char get() {
    if(getloc == putloc) {
        System.out.println(" - Queue is empty.");
        return (char) 0;
    }

    return q[getloc++];
}
}

// Demonstrate the Queue class.
class QDemo2 {
    public static void main(String[] args) {
        // construct 10-element empty queue
        Queue q1 = new Queue(10);

        char[] name = {'T', 'o', 'm'};
        // construct queue from array
        Queue q2 = new Queue(name);

        char ch;
        int i;

        // put some characters into q1
        for(i=0; i < 10; i++)
            q1.put((char) ('A' + i));

        // construct queue from another queue
        Queue q3 = new Queue(q1);

        // Show the queues.
        System.out.print("Contents of q1: ");
        for(i=0; i < 10; i++) {
            ch = q1.get();

```

```

        System.out.print(ch);
    }

    System.out.println("\n");

    System.out.print("Contents of q2: ");
    for(i=0; i < 3; i++) {
        ch = q2.get();
        System.out.print(ch);
    }

    System.out.println("\n");

    System.out.print("Contents of q3: ");
    for(i=0; i < 10; i++) {
        ch = q3.get();
        System.out.print(ch);
    }
}
}

```

The output from the program is shown here:

Contents of q1: ABCDEFGHIJ

Contents of q2: Tom

Contents of q3: ABCDEFGHIJ

Recursion

In Java, a method can call itself. This process is called *recursion*, and a method that calls itself is said to be *recursive*. In general, recursion is the process of defining something in terms of itself and is somewhat similar to a circular definition. The key component of a recursive method is a statement that executes a call to itself. Recursion is a powerful control mechanism.

The classic example of recursion is the computation of the factorial of a number. The *factorial* of a number N is the product of all the whole numbers between 1 and N . For example, 3 factorial is $1 \times 2 \times 3$, or 6. The following program shows a recursive way to compute the factorial of a number. For comparison purposes, a nonrecursive equivalent is also included.

```
// A simple example of recursion.
class Factorial {
    // This is a recursive function.
    int factR(int n) {
        int result;

        if(n==1) return 1;
        result = factR(n-1) * n;
        return result;
    }
    // This is an iterative equivalent.
    int factI(int n) {
        int t, result;

        result = 1;
        for(t=1; t <= n; t++) result *= t;
        return result;
    }
}

class Recursion {
    public static void main(String[] args) {
        Factorial f = new Factorial();

        System.out.println("Factorials using recursive method.");
        System.out.println("Factorial of 3 is " + f.factR(3));
        System.out.println("Factorial of 4 is " + f.factR(4));
        System.out.println("Factorial of 5 is " + f.factR(5));
        System.out.println();

        System.out.println("Factorials using iterative method.");
        System.out.println("Factorial of 3 is " + f.factI(3));
        System.out.println("Factorial of 4 is " + f.factI(4));
        System.out.println("Factorial of 5 is " + f.factI(5));
    }
}
```

↑ Execute the recursive call to **factR()**.

The output from this program is shown here:

Factorials using recursive method.

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

Factorials using iterative method.

Factorial of 3 is 6

Factorial of 4 is 24

Factorial of 5 is 120

The operation of the nonrecursive method **factI()** should be clear. It uses a loop starting at 1 and progressively multiplies each number by the moving product.

The operation of the recursive **factR()** is a bit more complex. When **factR()** is called with an argument of 1, the method returns 1; otherwise, it returns the product of **factR(n-1)*n**. To evaluate this expression, **factR()** is called with **n-1**. This process repeats until **n** equals 1 and the calls to the method begin returning. For example, when the factorial of 2 is calculated, the first call to **factR()** will cause a second call to be made with an argument of 1. This call will return 1, which is then multiplied by 2 (the original value of **n**). The answer is then 2. You might find it interesting to insert **println()** statements into **factR()** that show at what level each call is, and what the intermediate results are.

When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method. Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive methods could be said to “telescope” out and back.

Recursive versions of many routines may execute a bit more slowly than their iterative equivalents because of the added overhead of the additional method calls. Too many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not encounter this unless a recursive routine runs wild. The main advantage to recursion is that some types of algorithms can be implemented more clearly and simply recursively than they can be iteratively. For example, the Quicksort sorting algorithm is quite difficult to implement in an iterative way. Also, some problems, especially AI-related ones, seem to lend themselves to recursive solutions. When writing recursive methods, you must have a conditional statement, such as an **if**, somewhere to force the method to return without the recursive call being executed. If you don’t do this, once you call the method, it will never return. This type of error is very common when working with recursion. Use **println()** statements liberally so that you can watch what is going on and abort execution if you see that you have made a mistake.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main()**. **main()** is declared as **static** because it must be called by the JVM when your program begins. Outside the class, to use a **static** member, you need only specify the name of its class followed by the dot operator. No object needs to be created. For example, if you want to assign the value 10 to a **static** variable called **count** that is part of the **Timer** class, use this line:

```
Timer.count = 10;
```

This format is similar to that used to access normal instance variables through an object, except that the class name is used. A **static** method can be called in the same way—by use of the dot operator on the name of the class.

Variables declared as **static** are, essentially, global variables. When an object is declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable. Here is an example that shows the differences between a **static** variable and an instance variable:

```

// Use a static variable.
class StaticDemo {
    int x; // a normal instance variable
    static int y; // a static variable ← There is one copy of y
                                           for all objects to share.

    // Return the sum of the instance variable x
    // and the static variable y.
    int sum() {
        return x + y;
    }
}

class SDemo {
    public static void main(String[] args) {
        StaticDemo ob1 = new StaticDemo();
        StaticDemo ob2 = new StaticDemo();

        // Each object has its own copy of an instance variable.
        ob1.x = 10;
        ob2.x = 20;
        System.out.println("Of course, ob1.x and ob2.x " +
                           "are independent.");
        System.out.println("ob1.x: " + ob1.x +
                           "\nob2.x: " + ob2.x);
        System.out.println();

        // Each object shares one copy of a static variable.
        System.out.println("The static variable y is shared.");
        StaticDemo.y = 19;
        System.out.println("Set StaticDemo.y to 19.");

        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();

        StaticDemo.y = 100;
        System.out.println("Change StaticDemo.y to 100");

        System.out.println("ob1.sum(): " + ob1.sum());
        System.out.println("ob2.sum(): " + ob2.sum());
        System.out.println();
    }
}

```

The output from the program is shown here:

Of course, ob1.x and ob2.x are independent.

```
ob1.x: 10
```

```
ob2.x: 20
```

The static variable y is shared.

Set StaticDemo.y to 19.

```
ob1.sum(): 29
```

```
ob2.sum(): 39
```

Change StaticDemo.y to 100

```
ob1.sum(): 110
```

```
ob2.sum(): 120
```

As you can see, the **static** variable **y** is shared by both **ob1** and **ob2**. Changing it affects the entire class, not just an instance.

The difference between a **static** method and a normal method is that the **static** method is called through its class name, without any object of that class being created. You have seen an example of this already: the **sqrt()** method, which is a **static** method within Java's standard **Math** class. Here is an example that creates a **static** method:

```
// Use a static method.
class StaticMeth {
    static int val = 1024; // a static variable

    // a static method
    static int valDiv2() { ←———— A static method.
        return val/2;
    }
}

class SDemo2 {
    public static void main(String[] args) {

        System.out.println("val is " + StaticMeth.val);
        System.out.println("StaticMeth.valDiv2(): " +
                            StaticMeth.valDiv2());

        StaticMeth.val = 4;
        System.out.println("val is " + StaticMeth.val);
        System.out.println("StaticMeth.valDiv2(): " +
                            StaticMeth.valDiv2());
    }
}
```

The output is shown here:

```
val is 1024
StaticMeth.valDiv2(): 512
val is 4
StaticMeth.valDiv2(): 2
```

Methods declared as **static** have several restrictions:

- They can directly call only other **static** methods in their class.
- They can directly access only **static** variables in their class.
- They do not have a **this** reference.

For example, in the following class, the **static** method **valDivDenom()** is illegal:

```
class StaticError {
    int denom = 3; // a normal instance variable
    static int val = 1024; // a static variable

    /* Error! Can't access a non-static variable
       from within a static method. */
    static int valDivDenom() {
        return val/denom; // won't compile!
    }
}
```

Here, **denom** is a normal instance variable that cannot be accessed within a **static** method.

Static Blocks

Sometimes a class will require some type of initialization before it is ready to create objects. For example, it might need to establish a connection to a remote site. It also might need to initialize certain **static** variables before any of the class' **static** methods are used. To handle these types of situations, Java allows you to declare a **static** block. A **static** block is executed when the class is first loaded. Thus, it is executed before the class can be used for any other purpose. Here is an example of a **static** block:

```
// Use a static block
class StaticBlock {
    static double rootOf2;
    static double rootOf3;

    static { ←————— This block is executed
        System.out.println("Inside static block.");
        rootOf2 = Math.sqrt(2.0);
        rootOf3 = Math.sqrt(3.0);
    }
}
```

when the class is loaded.

```

    StaticBlock(String msg) {
        System.out.println(msg);
    }
}

class SDemo3 {
    public static void main(String[] args) {
        StaticBlock ob = new StaticBlock("Inside Constructor");

        System.out.println("Square root of 2 is " +
                           StaticBlock.rootOf2);
        System.out.println("Square root of 3 is " +
                           StaticBlock.rootOf3);

    }
}

```

The output is shown here:

```

Inside static block.
Inside Constructor
Square root of 2 is 1.4142135623730951
Square root of 3 is 1.7320508075688772

```

As you can see, the **static** block is executed before any objects are constructed.

Try This 6-3 The Quicksort

QSDemo.java

In [Chapter 5](#) you were shown a simple sorting method called the Bubble sort. It was mentioned at the time that substantially better sorts exist. Here you will develop a version of one of the best: the Quicksort. The Quicksort, invented and named by C.A.R. Hoare, is arguably the best general-purpose sorting algorithm currently available. The reason it could not be shown in [Chapter 5](#) is that the best implementations of the Quicksort rely on recursion. The version we will develop sorts a character array, but the logic can be adapted to sort any type of object you like.

The Quicksort is built on the idea of partitions. The general procedure is to select a value, called the *comparand*, and then to partition the array into two sections. All elements greater than or equal to the partition value are put on one side, and those less than the value are put on the other. This process is then repeated for each remaining section until the array is sorted. For example, given the array **fedacb** and using the value **d** as the comparand, the first pass of the Quicksort would rearrange the array as follows:

Initial	f e d a c b
Pass1	b c a d e f

This process is then repeated for each section—that is, **bca** and **def**. As you can see, the process is essentially recursive in nature, and indeed, the cleanest implementation of Quicksort is recursive.

Assuming that you have no information about the distribution of the data to be sorted, there are a number of ways you can select the comparand. Here are two. You can choose a value at random from within the data, or you can select it by averaging a small set of values taken from the data. For optimal sorting, you want a value that is precisely in the middle of the range of values. However, this is often not practical. In the worst case, the value chosen is at one extremity. Even in this case, however, Quicksort still performs correctly. The version of Quicksort that we will develop selects the middle element of the array as the comparand.

1. Create a file called **QSDemo.java**.
2. First, create the **Quicksort** class shown here:


```
// Try This 6-3: A simple version of the Quicksort.
class Quicksort {

    // Set up a call to the actual Quicksort method.
    static void qsort(char[] items) {
        qs(items, 0, items.length-1);
    }

    // A recursive version of Quicksort for characters.
    private static void qs(char[] items, int left, int right)
    {
        int i, j;
        char x, y;

        i = left; j = right;
        x = items[(left+right)/2];

        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        } while(i <= j);

        if(left < j) qs(items, left, j);
        if(i < right) qs(items, i, right);
    }
}
```

To keep the interface to the Quicksort simple, the **Quicksort** class provides the **qsort()** method, which sets up a call to the actual Quicksort method, **qs()**. This enables the Quicksort to be called with just the name of the array to be sorted, without having to provide an initial partition. Since **qs()** is only used internally, it is specified as **private**.

3. To use the **Quicksort**, simply call **Quicksort.qsort()**. Since **qsort()** is specified as **static**, it can be called through its class rather than on an object. Thus, there is no need to create a **Quicksort** object. After the call returns, the array will be sorted. Remember, this version works only for character arrays, but you can adapt the logic to sort any type of

arrays you want.

4. Here is a program that demonstrates **Quicksort**:

```
// Try This 6-3: A simple version of the Quicksort.
class Quicksort {

    // Set up a call to the actual Quicksort method.
    static void qsort(char[] items) {
        qs(items, 0, items.length-1);
    }

    // A recursive version of Quicksort for characters.
    private static void qs(char[] items, int left, int right)
    {
        int i, j;
        char x, y;

        i = left; j = right;
        x = items[(left+right)/2];

        do {
            while((items[i] < x) && (i < right)) i++;
            while((x < items[j]) && (j > left)) j--;

            if(i <= j) {
                y = items[i];
                items[i] = items[j];
                items[j] = y;
                i++; j--;
            }
        } while(i <= j);

        if(left < j) qs(items, left, j);
        if(i < right) qs(items, i, right);
    }
}

class QSDemo {
```

```

public static void main(String[] args) {
    char[] a = { 'd', 'x', 'a', 'r', 'p', 'j', 'i' };
    int i;

    System.out.print("Original array: ");
    for(i=0; i < a.length; i++)
        System.out.print(a[i]);

    System.out.println();

    // now, sort the array
    Quicksort.qsort(a);

    System.out.print("Sorted array: ");
    for(i=0; i < a.length; i++)
        System.out.print(a[i]);
}
}

```

Introducing Nested and Inner Classes

In Java, you can define a *nested class*. This is a class that is declared within another class. Frankly, the nested class is a somewhat advanced topic. In fact, nested classes were not even allowed in the first version of Java. It was not until Java 1.1 that they were added. However, it is important that you know what they are and the mechanics of how they are used because they play an important role in many real-world programs.

A nested class does not exist independently of its enclosing class. Thus, the scope of a nested class is bounded by its outer class. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two general types of nested classes: those that are preceded by the **static** modifier and those that are not. The only type that we are concerned about in this book is the non-static variety. This type of nested class is also called an *inner class*. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-**static** members of the outer class do.

Sometimes an inner class is used to provide a set of services that is needed only by its enclosing class. Here is an example that uses an inner class to compute various values for its enclosing class:

```

// Use an inner class.
class Outer {
    int[] nums;

```

```

Outer(int[] n) {
    nums = n;
}

void analyze() {
    Inner inOb = new Inner();

    System.out.println("Minimum: " + inOb.min());
    System.out.println("Maximum: " + inOb.max());
    System.out.println("Average: " + inOb.avg());
}

// This is an inner class.
class Inner { ← An inner class
    int min() {
        int m = nums[0];

        for(int i=1; i < nums.length; i++)
            if(nums[i] < m) m = nums[i];

        return m;
    }

    int max() {
        int m = nums[0];
        for(int i=1; i < nums.length; i++)
            if(nums[i] > m) m = nums[i];

        return m;
    }

    int avg() {
        int a = 0;
        for(int i=0; i < nums.length; i++)
            a += nums[i];

        return a / nums.length;
    }
}

class NestedClassDemo {
    public static void main(String[] args) {
        int[] x = { 3, 2, 1, 5, 6, 9, 7, 8 };
        Outer outOb = new Outer(x);

        outOb.analyze();
    }
}

```

The output from the program is shown here:

```
Minimum: 1  
Maximum: 9  
Average: 5
```

In this example, the inner class **Inner** computes various values from the array **nums**, which is a member of **Outer**. As explained, an inner class has access to the members of its enclosing class, so it is perfectly acceptable for **Inner** to access the **nums** array directly. Of course, the opposite is not true. For example, it would not be possible for **analyze()** to invoke the **min()** method directly, without creating an **Inner** object.

As mentioned, it is possible to nest a class within a block scope. Doing so simply creates a localized class that is not known outside its block. The following example adapts the **ShowBits** class developed in Try This 5-3 for use as a local class.

```
// Use ShowBits as a local class.
class LocalClassDemo {
    public static void main(String[] args) {

        // An inner class version of ShowBits.
        class ShowBits { ←———— A local class nested within a method
            int numbits;

            ShowBits(int n) {
                numbits = n;
            }

            void show(long val) {
                long mask = 1;

                // left-shift a 1 into the proper position
                mask <=<= numbits-1;

                int spacer = 0;
                for(; mask != 0; mask >>= 1) {
                    if((val & mask) != 0) System.out.print("1");
                    else System.out.print("0");
                    spacer++;
                    if((spacer % 8) == 0) {
                        System.out.print(" ");
                        spacer = 0;
                    }
                }
                System.out.println();
            }
        }

        for(byte b = 0; b < 10; b++) {
            ShowBits byteval = new ShowBits(8);

            System.out.print(b + " in binary: ");
            byteval.show(b);
        }
    }
}
```

The output from this version of the program is shown here:


```
0 in binary: 00000000
1 in binary: 00000001
2 in binary: 00000010
3 in binary: 00000011
4 in binary: 00000100
5 in binary: 00000101
6 in binary: 00000110
7 in binary: 00000111
8 in binary: 00001000
9 in binary: 00001001
```

In this example, the **ShowBits** class is not known outside of **main()**, and any attempt to access it by any method other than **main()** will result in an error.

One last point: You can create an inner class that does not have a name. This is called an *anonymous inner class*. An object of an anonymous inner class is instantiated when the class is declared, using **new**. Anonymous inner classes are discussed further in [Chapter 17](#).

Varargs: Variable-Length Arguments

Sometimes you will want to create a method that takes a variable number of arguments, based on its precise usage. For example, a method that opens an Internet connection might take a user name, password, file name, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. To create such a method implies that there must be some way to create a list of arguments that is variable in length, rather than fixed.

In the early days of Java, methods that required a variable-length argument list could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some situations, it applies to only a narrow class of situations. In cases where the maximum number of potential arguments is larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. Frankly, both of these approaches often resulted in clumsy solutions, and it was widely acknowledged that a better approach was needed.

Ask the Expert

Q: What makes a static nested class different from a non-static one?

A: A **static** nested class is one that has the **static** modifier applied. Because it is **static**, it can access only other **static** members of the enclosing class directly. It must

access other members of its outer class through an object reference.

Fortunately, today, Java includes a feature that greatly simplifies the creation of methods that require a variable number of arguments. This feature is called *varargs*, which is short for variable-length arguments. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*. The parameter list for a varargs method is not fixed, but rather variable in length. Thus, a varargs method can take a variable number of arguments.

Varargs Basics


A variable-length argument is specified by three periods (...). For example, here is how to write a method called **vaTest()** that takes a variable number of arguments:

```
// vaTest() uses a vararg.
static void vaTest(int ... v) {
    System.out.println("Number of args: " + v.length);
    System.out.println("Contents: ");

    for(int i=0; i < v.length; i++)
        System.out.println(" arg " + i + ": " + v[i]);

    System.out.println();
}
```

Declare a variable-length argument list.



Notice that **v** is declared as shown here:

```
int ... v
```

This syntax tells the compiler that **vaTest()** can be called with zero or more arguments. Furthermore, it causes **v** to be implicitly declared as an array of type **int[]**. Thus, inside **vaTest()**, **v** is accessed using the normal array syntax.

Here is a complete program that demonstrates **vaTest()**:

```
// Demonstrate variable-length arguments.
class VarArgs {

    // vaTest() uses a vararg.
    static void vaTest(int ... v) {
        System.out.println("Number of args: " + v.length);
        System.out.println("Contents: ");

        for(int i=0; i < v.length; i++)
            System.out.println(" arg " + i + ": " + v[i]);
    }
}
```

```

    System.out.println();
}

public static void main(String[] args)
{
    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);           // 1 arg
    vaTest(1, 2, 3);      // 3 args
    vaTest();             // no args
}

```

Call with different numbers of arguments.

The output from the program is shown here:

Number of args: 1

Contents:

arg 0: 10

Number of args: 3

Contents:

arg 0: 1

arg 1: 2

arg 2: 3

Number of args: 0

Contents:

There are two important things to notice about this program. First, as explained, inside **vaTest()**, **v** is operated on as an array. This is because **v** is an array. The **...** syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by **v**. Second, in **main()**, **vaTest()** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to **v**. In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to **doIt()** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to **vals**.

Here is a reworked version of the **vaTest()** method that takes a regular argument and a

variable-length argument:

```
// Use varargs with standard arguments.
class VarArgs2 {
```

The output from this program is shown here:

```
// Here, msg is a normal parameter and v is a
// varargs parameter.
static void vaTest(String msg, int ... v) { ← A "normal" and
    System.out.println(msg + v.length);      vararg parameter
    System.out.println("Contents: ");

    for(int i=0; i < v.length; i++)
        System.out.println("  arg " + i + ": " + v[i]);

    System.out.println();
}

public static void main(String[] args)
{
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
}
}
```

Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

```
One vararg: 1
Contents:
  arg 0: 10
```

```
Three varargs: 3
Contents:
  arg 0: 1
  arg 1: 2
  arg 2: 3
```

```
No varargs: 0
Contents:
```

Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal. There is one more restriction to be aware of: there must be only one varargs

parameter. For example, this declaration is also invalid:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
```

The attempt to declare the second varargs parameter is illegal.

Overloading Varargs Methods

You can overload a method that takes a variable-length argument. For example, the following program overloads **vaTest()** three times:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
```

```
// Varargs and overloading.
```

```
class VarArgs3 {
```

First version of **vaTest()**

```
static void vaTest(int ... v) {  
    System.out.println("vaTest(int ...): " +  
        "Number of args: " + v.length);  
    System.out.println("Contents: ");  
  
    for(int i=0; i < v.length; i++)  
        System.out.println("  arg " + i + ": " + v[i]);  
  
    System.out.println();  
}
```

Second version of **vaTest()**

```
static void vaTest(boolean ... v) {  
    System.out.println("vaTest(boolean ...): " +  
        "Number of args: " + v.length);  
    System.out.println("Contents: ");  
  
    for(int i=0; i < v.length; i++)  
        System.out.println("  arg " + i + ": " + v[i]);  
  
    System.out.println();  
}
```

Third version of **vaTest()**

```
static void vaTest(String msg, int ... v) {  
    System.out.println("vaTest(String, int ...): " +  
        msg + v.length);  
    System.out.println("Contents: ");  
  
    for(int i=0; i < v.length; i++)  
        System.out.println("  arg " + i + ": " + v[i]);  
  
    System.out.println();  
}
```

```
public static void main(String[] args)  
{  
    vaTest(1, 2, 3);  
    vaTest("Testing: ", 10, 20);  
    vaTest(true, false, false);  
}
```


The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3
Contents:
  arg 0: 1
  arg 1: 2
  arg 2: 3

vaTest(String, int ...): Testing: 2
Contents:
  arg 0: 10
  arg 1: 20

vaTest(boolean ...): Number of args: 3
Contents:
  arg 0: true
  arg 1: false
  arg 2: false
```

This program illustrates both ways that a varargs method can be overloaded. First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...)** and **vaTest(boolean ...)**. Remember, the ... causes the parameter to be treated as an array of the specified type. Therefore, just as you can overload methods by using different types of array parameters, you can overload varargs methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with **vaTest(String, int ...)**. In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method. For example, consider the following program:

```

// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

    // Use an int vararg parameter.
    static void vaTest(int ... v) { ← An int vararg
        // ...
    }

    // Use a boolean vararg parameter.
    static void vaTest(boolean ... v) { ← A boolean vararg
        // ...
    }

    public static void main(String[] args)
    {
        vaTest(1, 2, 3); // OK
        vaTest(true, false, false); // OK

        vaTest(); // Error: Ambiguous! ← Ambiguous!
    }
}

```

In this program, the overloading of **vaTest()** is perfectly correct. However, this program will not compile because of the following call:

```
vaTest(); // Error: Ambiguous!
```

Because the vararg parameter can be empty, this call could be translated into a call to **vaTest(int ...)** or to **vaTest(boolean ...)**. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of **vaTest()** are inherently ambiguous even though one takes a normal parameter:

```

static void vaTest(int ... v) { // ...

static void vaTest(int n, int ... v) { // ...

```

Although the parameter lists of **vaTest()** differ, there is no way for the compiler to resolve the following call:

```
vaTest(1)
```

Does this translate into a call to **vaTest(int ...)**, with one varargs argument, or into a call to

vaTest(int, int ...) with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Because of ambiguity errors like those just shown, sometimes you will need to forego overloading and simply use two different method names. Also, in some cases, ambiguity errors expose a conceptual flaw in your code, which you can remedy by more carefully crafting a solution.

✓ Chapter 6 Self Test

1. Given this fragment,

```
class X {  
    private int count;
```

is the following fragment correct?

```
class Y {  
    public static void main(String[] args) {  
        X ob = new X();  
  
        ob.count = 10;
```

2. An access modifier must _____ a member's declaration.
3. The complement of a queue is a stack. It uses first-in, last-out accessing and is often likened to a stack of plates. The first plate put on the table is the last plate used. Create a stack class called **Stack** that can hold characters. Call the methods that access the stack **push()** and **pop()**. Allow the user to specify the size of the stack when it is created. Keep all other members of the **Stack** class private. (Hint: You can use the **Queue** class as a model; just change the way the data is accessed.)
4. Given this class,

```
class Test {  
    int a;  
    Test(int i) { a = i; }  
}
```

write a method called **swap()** that exchanges the contents of the objects referred to by two **Test** object references.

5. Is the following fragment correct?

```
class X {  
    int meth(int a, int b) { ... }  
    String meth(int a, int b) { ... }
```

6. Write a recursive method that displays the contents of a string backwards.
7. If all objects of a class need to share the same variable, how must you declare that variable?
8. Why might you need to use a **static** block?
9. What is an inner class?
10. To make a member accessible by only other members of its class, what access modifier must be used?
11. The name of a method plus its parameter list constitutes the method's _____.
12. An **int** argument is passed to a method by using call-by-_____.
13. Create a varargs method called **sum()** that sums the **int** values passed to it. Have it return the result. Demonstrate its use.
14. Can a varargs method be overloaded?
15. Show an example of an overloaded varargs method that is ambiguous.