As you can see, each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine.

The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Recursion

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

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 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

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

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

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

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

If you are unfamiliar with recursive methods, then the operation of **fact( )** may seem a bit confusing. Here is how it works. When **fact( )** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(n–1)*n**. To evaluate this expression, **fact( )** is called with **n–1**. This process repeats until **n** equals 1 and the calls to the method begin returning.

To better understand how the **fact( )** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact( )** will cause a second call to be made with an argument of 2. This invocation will cause **fact( )** to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact( )** and multiplied by 3 (the original value of **n**). This yields the answer, 6. You might find it interesting to insert **println( )** statements into **fact( )** which will show at what level each call is and what the intermediate answers 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 the iterative equivalent because of the added overhead of the additional function calls. 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 have to worry about this unless a recursive routine runs wild.

The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives. For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Some problems, especially AI-related ones, seem to lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively.

When writing recursive methods, you must have an **if** statement 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 is a very common error in working with recursion. Use **println( )** statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.

class RecTest {
  int values[];

  RecTest(int i) {
    values = new int[i];
  }

  // display array -- recursively
  void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println("[" + (i-1) + "] " + values[i-1]);
  }
}

class Recursion2 {
  public static void main(String args[]) {
    RecTest ob = new RecTest(10);
    int i;

    for(i=0; i<10; i++) ob.values[i] = i;

    ob.printArray(10);
  }
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

```
[7] 7
[8] 8
[9] 9
```

Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of Chapter 6. While it is true that the methods **push( )** and **pop( )** do provide a controlled interface to the stack, this interface is not enforced. That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section you will be introduced to the mechanism by which you can precisely control access to the various members of a class.

How a member can be accessed is determined by the *access specifier* that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy.

Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access specifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)

In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods which are private to a class.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;

private int myMethod(int a, char b) { // ...
```

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

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
  int a; // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) { // set c's value
    c = i;
  }
  int getc() { // get c's value
    return c;
  }
}

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

    // These are OK, a and b may be accessed directly
    ob.a = 10;
    ob.b = 20;

    // This is not OK and will cause an error
//  ob.c = 100; // Error!

    // You must access c through its methods
    ob.setc(100); // OK
```

```
    System.out.println("a, b, and c: " + ob.a + " " +
                        ob.b + " " + ob.getc());
  }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**. If you were to remove the comment symbol from the beginning of the following line,

```
//  ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of Chapter 6.

```
// This class defines an integer stack that can hold 10 values.
class Stack {
  /* Now, both stck and tos are private.  This means
     that they cannot be accidentally or maliciously
     altered in a way that would be harmful to the stack.
  */
  private int stck[] = new int[10];
  private int tos;

  // Initialize top-of-stack
  Stack() {
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
```

```
  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push( )** and **pop( )**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array.

The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // push some numbers onto the stack
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<10; i++)
       System.out.println(mystack2.pop());

    // these statements are not legal
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
```

```
    }
  }
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

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 only in conjunction with an object of its class. However, 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 before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

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

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block which gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
```

```
  static int a = 3;
  static int b;

  static void meth(int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }

  static {
    System.out.println("Static block initialized.");
    b = a * 4;
  }

  public static void main(String args[]) {
    meth(42);
  }
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**. Then **main()** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

ember *It is illegal to refer to any instance variables inside of a* ***static*** *method.*

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of **final** is described in the next chapter, when inheritance is described.

Arrays Revisited

Arrays were introduced earlier in this book, before classes had been discussed. Now that you know about classes, an important point can be made about arrays: they are implemented as objects. Because of this, there is a special array attribute that you will want to take advantage of. Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable. All arrays have this variable, and it will always hold the size of the array. Here is a program that demonstrates this property:

```
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

You can put the length member to good use in many situations. For example, is an improved version of the Stack class. As you might recall, the earlier version this class always created a ten-element stack. The following version lets you creat stacks of any size. The value of **stck.length** is used to prevent the stack from overflowing.

```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }

  // Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);
```

```
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep. As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

static
non st

Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

1

There are two types of nested classes: *static* and *non-static*. A static nested class is one which has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

2 The most important type of nested class is the *inner* class. An inner class is a non-static nested 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. Thus, an inner class is fully within the scope of its enclosing class.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;
```

```
    void test() {
      Inner inner = new Inner();
      inner.display();
    }

    // this is an inner class
    class Inner {
      void display() {
        System.out.println("display: outer_x = " + outer_x);
      }
    }
  }

  class InnerClassDemo {
    public static void main(String args[]) {
      Outer outer = new Outer();
      outer.test();
    }
  }
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

It is important to realize that class **Inner** is known only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**. Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope.

As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    int y = 10; // y is local to Inner
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus it is not known outside of that class and it cannot be used by **showy( )**.

Although we have been focusing on nested classes declared within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows.

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;
```

```
        void test() {
          for(int i=0; i<10; i++) {
            class Inner {
              void display() {
                System.out.println("display: outer_x = " + outer_x);
              }
            }
            Inner inner = new Inner();
            inner.display();
          }
        }
      }

      class InnerClassDemo {
        public static void main(String args[]) {
          Outer outer = new Outer();
          outer.test();
        }
      }
```

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

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet. We will return to the topic of nested classes in Chapter 20. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name.

One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.