Module 3

Introducing Classes

- The class is at the core of Java.
- It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.

Class Fundamentals

- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.
- A class is declared by use of the **class** keyword

```
class classname {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
                // body of method
        }
        type methodnameN(parameter-list) {
                // body of method
        }
}
```

- The data, or variables, defined within a **class** are called *instance variables.*
- The code is contained within *methods.*
- Collectively, the methods and variables defined within a class are called *members* of the class.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

```
class Box {
        double width;
        double height;
        double depth;
}
```

- As stated, a class defines a new type of data.
- In this case, the new data type is called **Box**.
- You will use this name to declare objects of type **Box**.
- It is important to remember that a **class** declaration only creates a template; it does not create an actual object

```
Box mybox = new Box(); // create a Box object called mybox
```

- After this statement executes, **mybox** will be an instance of **Box**.
- Thus, it will have "physical" reality.
- Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**.
- To access these variables, you will use the *dot* (.) operator.
- The dot operator links the name of the object with the name of an instance variable.

For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

mybox.width = 100;

```
class Box {
        double width;
        double height;
        double depth;
}
// This class declares an object of type Box.
class BoxDemo {
        public static void main(String args[]) {
                Box mybox = new Box();
                double vol;
                // assign values to mybox's instance variables
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                // compute volume of box
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

Declaring Objects

- When you create a class, you are creating a new data type.
- However, obtaining objects of a class is a two-step process.
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator.
- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable.
- Thus, in Java, all class objects must be dynamically allocated.

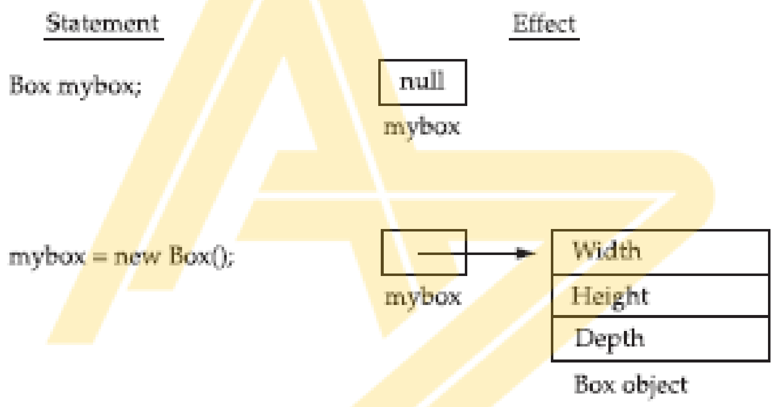
Box mybox = new Box();

- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

- The first line declares **mybox** as a reference to an object of type **Box**.
- After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object.
- Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**.
- After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.



- Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.
- A constructor defines what occurs when an object of a class is created.
- Constructors are an important part of all classes and have many significant attributes.
- It is important to understand that **new** allocates memory for an object during run time.
- The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur.
- A class creates a new data type that can be used to create objects.

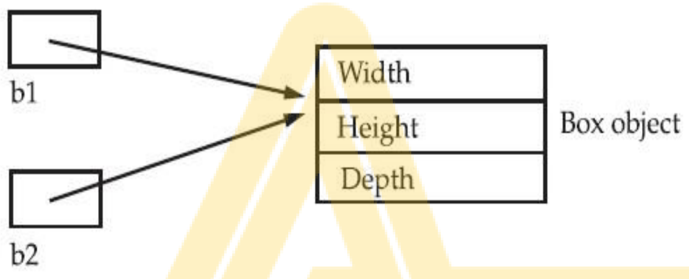
- That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class.
- Thus, a class is a logical construct. An object has physical reality.

Assigning Object Reference Variables

Object reference variables act differently when an assignment takes place.

```
Box b1 = new Box();
Box b2 = b1;
```

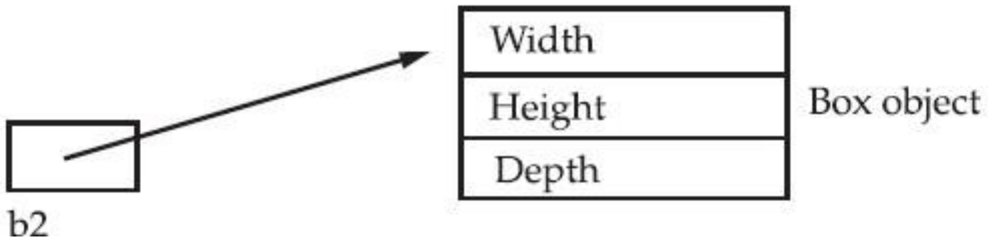
This situation is depicted here:



- After this fragment executes, **b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object.
- It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
- Although **b1** and **b2** both refer to the same object, they are not linked in any other way.

```
Box b1 = new Box(); Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.



Introducing methods

This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name*. This can be any legal identifier other than those already used by other items within the current scope.
- The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the *arguments* passed to the method when it is called.
- If the method has no parameters, then the parameter list will be empty.
- Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement:

  return *value*;
- Here, *value* is the value returned.

Adding a Method to the Box Class

```
class Box {
        double width;
        double height;
        double depth;
        // display volume of a box
        void volume() {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class BoxDemo3 {
        public static void main(String args[]) {
```

```
        Box mybox1 = new Box();
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        // display volume of first box
        mybox1.volume();
        }
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

Look closely at the following two lines of code:

```
mybox1.volume();
```

- The first line here invokes the **volume( )** method on **mybox1**.
- That is, it calls **volume( )** relative to the **mybox1** object, using the object's name followed by the dot operator.
- Thus, the call to **mybox1.volume( )** displays the volume of the box defined by **mybox1**,
- There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator.
- When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.
- This is easy to understand if you think about it. Amethod is always invoked relative to some object of its class. Once this invocation has occurred, the object is known.

Returning a Value

- While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it.

```
class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}
class BoxDemo4 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                double vol;
                // assign values to mybox1's instance variables
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
        }
}
```

- As you can see, when **volume( )** is called, it is put on the right side of an assignment statement.
- On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**.
- Thus, after vol = mybox1.volume(); executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **vol**.

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters

- While some methods don't need parameters, most do. Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations

```
int square()
{
        return 10 * 10;
}
```

- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
      return i * i;
}
```

- Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

- In the first call to **square( )**, the value 5 will be passed into parameter **i**.
- In the second call, **I** will receive the value 9.
- The third invocation passes the value of **y**, which is 2 in this example.
- As these examples show, **square( )** is able to return the square of whatever data it is passed
- A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter.

- An *argument* is a value that is passed to a method when it is Invoked.
- For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.
- Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately.
- This concept is implemented by the following program:

```
// This program uses a parameterized method.
class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
        // sets dimensions of box
        void setDim(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
}
class BoxDemo5 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                // initialize each box
                mybox1.setDim(10, 20, 15);
                mybox2.setDim(3, 6, 9);
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

- As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**.
- Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- A *constructor* initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
class Box {
        double width;
        double height;
```

```
    double depth;
    // This is the constructor for Box.
    Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo6 {
    public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

***class-var* = new *classname*( );**

- Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

  Box mybox1 = new Box();

- **new Box( )** is calling the **Box( )** constructor **new .**
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class

Parameterized Constructors

- While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions.
- What is needed is a way to construct **Box** objects of various dimensions.
- The easy solution is to add parameters to the constructor

```
class Box {
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}
class BoxDemo7 {
        public static void main(String args[]) {
                // declare, allocate, and initialize Box objects
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box(3, 6, 9);
                double vol;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

The this keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object

```
Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
}
```

Uses of this:

→ To overcome shadowing or instance variable hiding.

→ To call an overload constructor

Instance Variable Hiding

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.

- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

NOTE: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- Java takes a different approach; it handles deallocation for you automatically.
- The technique that accomplishes this is called *garbage collection.*
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

The **finalize( )** method has this general form:

```
        protected void finalize( )
        {
        // finalization code here
        }
```

- Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.
- It is important to understand that **finalize( )** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object.
- It must not rely on **finalize( )** for normal program operation.

<u>A Stack Class</u>

```
class Stack {
    int stck[] = new int[10];
    int tos;
    // Initialize top-of-stack
    Stack() {
        top = -1;
    }
// Push an item onto the stack
void push(int item) {
    if(top==9)
        System.out.println("Stack is full.");
    else
        stck[++top] = item;
    }
    // Pop an item from the stack
```

```
        int pop() {
                if(top < 0) {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return stck[top--];
        }
}
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());
        }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
```

```
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

Overloading Methods

- In Java it is possible to define two or more methods within the same class that 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 supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

```
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }
        // Overload test for one integer parameter.
```

```
    void test(int a) {
        System.out.println("a: " + a);
    }
    // Overload test for two integer parameters.
    void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
    }
    // overload test for a double parameter
    double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
        // call all versions of test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
    }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.

- However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
        void test() {
                System.out.println("No parameters");
        }
        // Overload test for two integer parameters.
        void test(int a, int b) {
                System.out.println("a and b: " + a + " " + b);
        }
        // overload test for a double parameter
        void test(double a) {
                System.out.println("Inside test(double) a: " + a);
        }
}
class Overload {
        public static void main(String args[]) {
                OverloadDemo ob = new OverloadDemo();
                int i = 88;
                ob.test();
                ob.test(10, 20);
                ob.test(i); // this will invoke test(double)
                ob.test(123.2); // this will invoke test(double)
        }
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

- When **test( )** is called with an integer argument inside **Overload**, no matching method is found.
- However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call.
- Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.
- Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.
- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.
- 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.

<u>Overloading Constructors</u>

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.
- To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {
        double width;
        double height;
        double depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box() {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}
class OverloadCons {
        public static void main(String args[]) {
                // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                // get volume of cube
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.
- For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // return true if o is equal to the invoking object
        boolean equals(Test o) {
                if(o.a == a && o.b == b) return true;
                else return false;
        }
}
class PassOb {
        public static void main(String args[]) {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equals(ob2));
                System.out.println("ob1 == ob3: " + ob1.equals(ob3));
        }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

- As you can see, the **equals( )** method inside **Test** compares two objects for equality and returns the result.
- That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals( )** specifies **Test** as its type.
- Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.
- One of the most common uses of object parameters involves constructors. Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- To do this, you must define a constructor that takes an object of its class as a parameter

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument 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.
- 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, Java uses both approaches, depending upon what is passed.
- In Java, when you pass a primitive type to a method, it is passed by value. Thus, 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 {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
        }
}
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.meth(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
```

- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Thus, 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 by reference.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
}
// pass an object
void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
}
}
class CallByRef {
        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.meth(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: 30 10
```

Returning Objects

- A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
        int a;
        Test(int i) {
                a = i;
        }
        Test incrByTen() {
                Test temp = new Test(a+10);
                return temp;
        }
}
class RetOb {
        public static void main(String args[]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: " + ob2.a);
        }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

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 method
        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
```

- 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.
- 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.

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.

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

- 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 welldefined 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
- Java's access specifiers are **public**, **private**, and **protected**.
- Java also defines a default access level. **protected** applies only when inheritance is involved 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.
- 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) { // ...

```
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!
```

<u>Understanding static</u>

- 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 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, which prints a message and then initializes **b** 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**.

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:

  ***classname.method*( )**

- Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables.
- A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.
- Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
        static int a = 42;
        static int b = 99;
        static void callme() {
                System.out.println("a = " + a);
        }
}
```

```
class StaticByName {
	public static void main(String args[]) {
		StaticDemo.callme();
		System.out.println("b = " + StaticDemo.b);
	}
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

- A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared.
- For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

- 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.

Arrays Revisited

- 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
```

- You can put the **length** member to good use in many situations. For example, here is an improved version of the **Stack** class. As you might recall, the earlier versions of this class always created a ten-element stack.
- The following version lets you create 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());
        }
```

```
}
```