



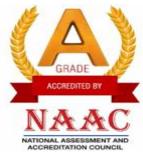
Sri Sai Vidya Vikas Shikshana Samithi ®

SAI VIDYA INSTITUTE OF TECHNOLOGY

Approved by AICTE, New Delhi, Affiliated to VTU, Recognized by Govt. of Karnataka
Accredited by NBA

RAJANUKUNTE, BENGALURU 560 064, KARNATAKA

Phone: 080-28468191/96/97/98 ,Email: info@saividya.ac.in, URL www.saividya.ac.in



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (CSE)

Module -2

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.
- The class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Class Fundamentals

- The classes created exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax.
- A class is that it 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.

The General Form of a Class

- A class is declared by use of the **class** keyword.
- The classes that have been used up to this point are actually very limited examples of its complete form.
- Classes can (and usually do) get much more complex.

A simplified general form of a **class** definition is shown here:

```
class classname {  
    type instance-  
    variable1; type  
    instance-variable2;  
    // ...  
    type instance-variableN;  
  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(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.
- The general rule, it is the methods that determine how a class' data can be used.
- 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.
- The general form of a class does not specify a **main()** method.
- Java classes do not need to have a **main()** method.
- You only specify one if that class is the starting point for your program.

A Simple Class

Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods.

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

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. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

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

After this statement executes, **mybox** will refer to an instance of **Box**. Thus, it will have “physical” reality.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. 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;
```

In general, we use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the **Box** class:

```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
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);
    }
}

```

You should call the file that contains this program **BoxDemo.java**, because the **main()** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

output:

```
Volume is 3000.0
```

Each object has its own copies of the instance variables. If you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```

// This program declares two Box objects.

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

class BoxDemo2 {
    public static void main(String[] args) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
    }
}

```

```

    // compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
}
}

```

The output produced by this program is shown here:

```

Volume is 3000.0
Volume is 162.0

```

mybox1's data is completely separate from the data contained in **mybox2**.

Declaring Object:

- When you create a class, we are creating a new data type. 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, essentially, 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

Declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described.

```

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**. At this point, **mybox** does not yet refer to an actual object. The next line allocates an 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, in essence, the memory address of the actual **Box** object.

A Closer Look at new

The **new** operator dynamically allocates memory for an object. In the context of an assignment, it has this general form:

class-var = new *classname* ();

- 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.
- Most real-world classes explicitly define their own constructors within their class definition.
- However, if no explicit constructor is specified, then Java will automatically supply a default constructor.

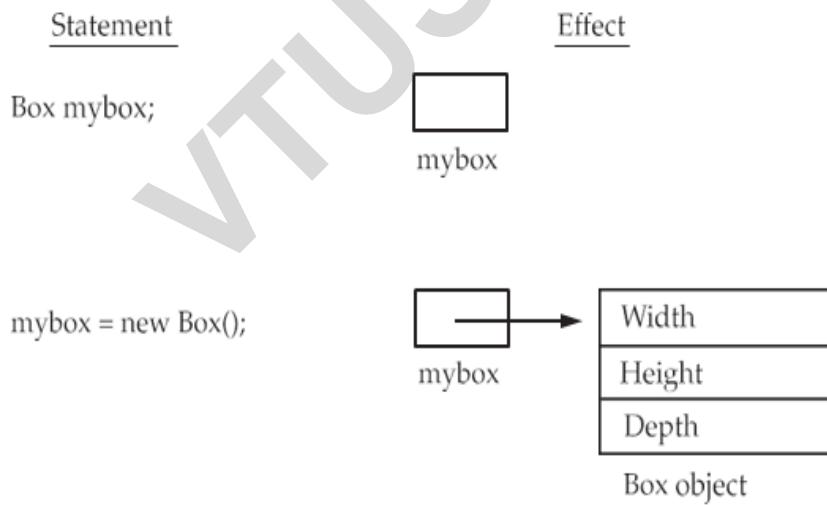


Figure: Declaring an object of type **Box**

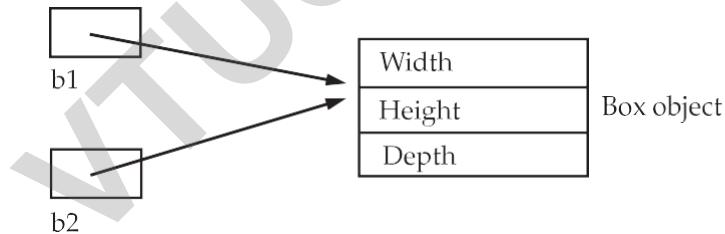
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

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.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.

For example:

```
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

Classes usually consist of two things: instance variables and methods, to add methods to your classes.

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

Adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// This program includes a method inside 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();
        Box mybox2 = new Box();

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // display volume of first box
        mybox1.volume();

        // display volume of second box
        mybox2.volume();
    }
}
```

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

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

```
Here, mybox1.volume();
mybox2.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**, and the call to **mybox2.volume()** displays the volume of the box defined by **mybox2**. Each time **volume()** is invoked, it displays the volume for the specified box.

Returning a Value

A better way to implement **volume()** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume() returns the volume of a box.

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();
        Box mybox2 = new Box();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* assign different values to mybox2's
           instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 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);
    }
}
```

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.

The call to **volume()** could have been used in the **println()** statement directly, as shown here:

```
System.out.println("Volume is " + mybox1.volume());
```

In this case, when **println()** is executed, **mybox1.volume()** will be called automatically and its value will be passed to **println()**.

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. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
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

// 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.

Constructors

- 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 when the object is created, before the **new** operator completes.
- Constructors have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

Types of Constructors in Java

- Default Constructor
- Parameterized Constructor

1. Default Constructor in Java

A constructor that has no parameters is known as default the constructor. A default constructor is invisible.

2. Parameterized Constructor in Java

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then we use a parameterized constructor.

You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor.

This version is shown here:

```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
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

```

Parameterized Constructors

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then we use a parameterized constructor.

For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

```

/* Here, Box uses a parameterized constructor to initialize the dimensions of a box.*/
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;
    }
}

```

```

// 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. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box()**:

```

// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}

```

The this Keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

To better understand what **this** refers to, consider the following version of **Box()**:

```
// A redundant use of this.  
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

Instance Variable Hiding

- Instance variable hiding refers to a state when instance variables of the same name are present in superclass and subclass.
- When a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class.
- Because **this** lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.
- For example, here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

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

Garbage Collection

- Java takes an approach for deallocation of memory 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 need to explicitly destroy objects.
- 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.

A Stack Class

- A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used.
- Stacks are controlled through two operations traditionally called *push* and *pop*.
- To put an item on top of the stack, you will use *push*.
- To take an item off the stack, you will use *pop*. As you will see, it is easy to encapsulate the entire stack mechanism.
- Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values

class Stack {
    int[] stck = new int[10];
    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 stck[tos--];
    }
}
```

```
        return 0;
    }
    else
        return stck[tos--];
}
}

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.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.  
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
```

As you can see, **test()** 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 one **double** parameter. The fact that the fourth version of **test()** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

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: 88.0
Inside test(double) a: 123.2
```

Overloading Constructors

Java supports Constructor Overloading in addition to overloading methods. In Java, overloaded constructor is called based on the parameters specified when a [new](#) is executed.

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
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;

        // get volume of first box
        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

```

// 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 equalTo(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.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}

```

This program generates the following output:

```

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

```

The **equalTo()** 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 **equalTo()** 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.

For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

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

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.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 OverloadCons2 {
    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);

        Box myclone = new Box(mybox1); // create copy of mybox1

        double vol;

        // get volume of first box
        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 cube is " + vol);

        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}
```

Argument Passing

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.

When you pass a primitive type 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 {
    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

```

Java Program for Objects are passed through their references.

```

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 PassObjRef {
    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

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 $1 \times 2 \times 3 \times$, 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
```

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
```

Access Control

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.

- Java's access modifiers 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 **public**, 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** modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier 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 Chapter 9.)
- 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 that are private to a class.

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());
    }
}
```

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;  
    }  
}
```

Understanding static

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

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 parameter **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 field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second,

you can assign it a value within a constructor. The first approach is probably the most common. Here is an 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** fields, as this example shows.

In addition to fields, both method parameters and local variables can be declared **final**. Declaring a parameter **final** prevents it from being changed within the method. Declaring a local variable **final** prevents it from being assigned a value more than once.

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

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 does not exist independently 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. 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 types of nested classes: *static* and *non-static*. A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly.

The second 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.

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 an instance of **Inner** can be created only in the context of class **Outer**. The Java compiler generates an error message otherwise. In general, an inner class instance is often created by code within its enclosing scope, as the example does.

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 inner classes declared as members 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
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events. We will return to the topic of nested classes in Chapter 25. 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.