30

MODULE -2

Introducing Classes: Class Fundamentals, Declaring Objects, Assigning Object Reference Variables, Introducing Methods, Constructors, The this Keyword, Garbage Collection, The finalize( ) Method, A Stack Class, A Closer Look at Methods and Classes: Overloading Methods, Using Objects as Parameters, A Closer Look at Argument Passing, Returning Objects, Recursion, Introducing Access Control, Understanding static, Introducing final, Arrays Revisited, Inheritance: Inheritance, Using super, Creating a Multilevel Hierarchy, When Constructors Are Called, Method Overriding, Dynamic Method Dispatch, Using Abstract Classes, Using final with Inheritance, The Object Class.
Text book 1: Ch 6, Ch 7 (7.1-7.9)

➢ The General Form of a Class

- A class is declared by use of the **class** keyword. A simplified general form of a **class** definition isshown 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. In most classes, the instance variables are acted upon and accessed

by the methods defined for that class. Thus, as a 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. Thus, the data for one object is separate and unique from the data for another.
- All methods have the same general form as **main( )**, which we have been using thus far. However, most methods will not be specified as **static** or **public**.
- Notice that the general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method. We only specify one if that class is the starting point for your program. Further, applets don't require a **main( )** method at all.

➢ A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, Box does not contain anymethods (but some will be added soon).

```
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**. This name is used 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 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 be an instance of **Box**. Thus, it will have "physical" reality. For the moment, do not worry about the details of this statement.
- 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;
```

- This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you 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);
}
}
```

- 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 be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively. To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

  Volume is 3000.0
- Each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth, width,** and **height.**
- 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'sinstance 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 separate from the data contained in **mybox2.**

➢ Declaring Objects:

- When you create a class, you are creating a new data type. You can use this type to declare objects of that 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 by 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 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. In the preceding sample programs, a line like the following is used to declare an object of type **Box**:
- Box mybox = new Box();

  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, mybox simply holds the memory address of the actual **Box** object. The effect of these two lines of code is depicted.

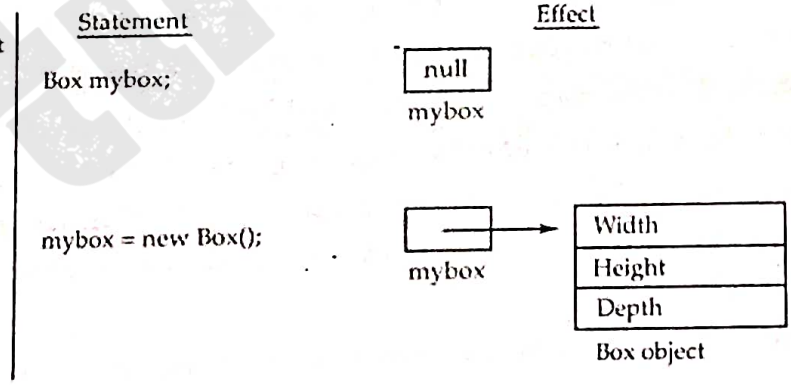


FIGURE 6-1 Declaring an object of type **Box**

- 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.

- However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. There is no need to use **new** for such things as integers or characters as Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables.
- It is important to understand that **new** allocates memory for an object during run time. The advantage of this approach is that 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. (That is, an object occupies space in memory.)

➢ Assigning Object Reference Variables

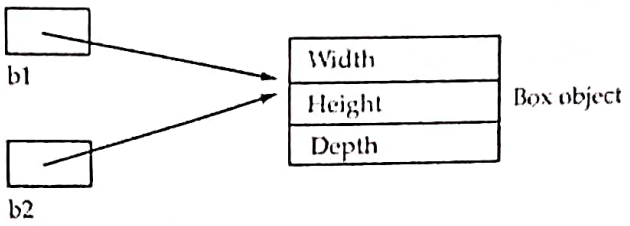
- Object reference variables act differently than you might expect when an assignment takes place. For example,

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

- You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, 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. 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. This is the general formof 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

// 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
```

Look closely at the following two lines of code:

```
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( ) isinvoked, it displays the volume for the specified box.

- When **mybox1.volume( )** is executed, the Java run-time system transfers control to the code defined inside **volume( )**. After the statements inside **volume( )** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call.
- 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. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume( )** implicitly refer to the copies of those variables found in the object that invokes **volume( )**.
- When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

➢ 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,

```
// 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
boxvol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

- 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 returnan 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 preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. 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

- Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in several slightly different situations.
- To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number10:

- int square() {

return 10 * 10;

}

- While this method does, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, 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
  25x = square(9); // x equals 81y = 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. It is important to keep the two terms *parameter* and *argument* straight.
- 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.

// 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);
}
}
```

- 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

- A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically like a method. Once defined, the constructor is automatically called immediately after 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.
- Let us begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize thedimensions 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
```

- Both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume.
- The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object when you allocate an object, you use the following general form: *class-var* = new *classname*( );
- Now you can understand why the parentheses are needed after the class name. What is 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.
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line ofcode worked in earlier versions of **Box** that did not define a constructor.
- The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

➢ Parameterized Constructors

- While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not of various dimensions.
- **/* Here, Box uses a parameterized constructor toinitialize 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;
}
// 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
```

- Each object is initialized as specified in the parameters to its constructor. For example, in the following line, **Box mybox1 = new Box(10, 20, 15);** the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

➢ This this Keyword

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines **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;
}
```

- This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object.

➢ 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.
- 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.
- If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. Because **this** lets you refer directly to the object, you can use it to resolve any name space 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

- 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.
- 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. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

➢ 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.
- To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object. 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.

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

- The Stack class defines two data items and three methods. The stack of integers is held by the array stck. This array is indexed by the variable tos, which always contains the index of the top of the stack. The Stack( ) constructor initializes tos to –1, which indicates an empty stack. The method push( ) puts an item on the stack.
- To retrieve an item, call pop( ). Since access to the stack is through push( ) and pop( ), the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by push( ) and pop( ) would remain the same.
- The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
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
```

Difference between Constructors and Methods

Constructors	Methods
A Constructor is a block of code that initializes a newly created object.	A Method is a collection of statements which returns a value upon its execution.
A Constructor can be used to initialize an object.	A Method consists of Java code to be executed.
A Constructor is invoked implicitly by the system.	A Method is invoked by the programmer.
A Constructor is invoked when a object is created using the keyword **new**.	A Method is invoked through method calls.
A Constructor doesn't have a return type.	A Method must have a return type.
A Constructor initializes a object that doesn't exist.	A Method does operations on an already created object.
A Constructor's name must be same as the name of the class.	A Method's name can be anything.
A class can have many Constructors but must not have the same parameters.	A class can have many methods but must not have the same parameters.
A Constructor cannot be inherited by subclasses.	A Method can be inherited by subclasses.

➢ Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, if 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 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 parametersa: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

- **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 ofno consequence relative to overloading, since return types do not play a role in overload resolution.
- 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 parametersa and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

- This version of **OverloadDemo** does not define **test(int)**. Therefore, 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 paradigms. The value of overloading is that it allows related methods to be accessed by use of a common name.

➢ Overloading Constructors

```
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;
} }
```

- The **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

  Box ob = new Box();

- Since **Box( )** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you. Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described.Here is a program that contains an improved version of **Box** that does just that:

- **Here, Box defines three constructors to initializethe 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 indicatean uninitialized box
height = -1;
depth = -1;
}
// 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
boxvol = 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
```

- The proper overloaded constructor is called based upon the parameters specified when **new** is executed.

➢ 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 arg2s[]) {
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
```

- 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. 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;
height = -1;
depth = -1;
}
// 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);
vol = mybox2.volume(); // get volume of second box
System.out.println("Volume of mybox2 is " + vol);
vol = mycube.volume(); // get volume of cube
System.out.println("Volume of cube is " + vol);
 vol = myclone.volume(); // get volume of clone
System.out.println("Volume of clone is " + vol);
}
}
```

➢ 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.
- Java uses both approaches, depending upon what is passed. In Java, when you passa 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

- The operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.
- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- When you create a variable of a class type, you are only creating a reference to an object. 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;
}
```

```
void meth(Test o) { // pass an object
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

- The actions inside **meth( )** have affected the object used as an argument. when an object reference is passed to a method, the reference itself is passed by use of call-by-value.
- However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

➢ 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
```

- Each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine.

➢ 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, 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 {
int fact(int n) {  // this is a recursive method
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 **fact( )** is called with an argument of 1, the function returns 1; otherwise, it returns the product of **fact(n–1)*n**.
- To evaluate this expression, **fact( )** is called with **n–1**. This process repeats until **n** equals 1 and the calls to the method begin returning. When you compute the factorial of 3, the first call to **fact( )** will cause a second call to be made with an argument of 2. This invocation will cause **fact( )** to be called a third time with an argument of 1.
- This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact( )** and multiplied by 3 (the original value of **n**). This yields the answer, 6.
- 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.
- 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. When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed. If you do not do this, once you call the method, it will never return. This is a very common error in working with recursion.
- Use println( ) statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake. Here is one more example of recursion. The recursive method printArray( ) prints the first i elements in the array values.

```
class RecTest {
int values[];
RecTest(int i) {
values = new int[i]; }
void printArray(int i) {  // display array – recursively
```

```
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);
}
    }
```

➢ Introducing Access Control

- Allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access specifiers. Let us begin by defining **public** and **private**.
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. **main( )** has always been preceded by the **public** specifier. It is called by code that isoutside 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. **Here is an example:**

```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
/* 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());
}
}
```

- Inside the **Test** class, **a** uses default access, which for this example is the sameas specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**. If you were to remove the comment symbol from the beginning of the following line,
- // ob.c = 100; // Error! then you would not be able to compile this program because of the access violation.

➢ Understanding 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:
  1. They can only call other **static** methods.
  2. They must only access **static** data.
  3. They cannot refer to **this** or **super** in any way.
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

- The **UseStatic** class is loaded, all 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 bythe dot operator.
- For example, if you wish to call a **static** method from outside its class, you cando so using the following general form: ***classname.method( )***
- Here, *classname* is the name of the class in which the **static** method is declared. 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; s
tatic 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. Thismeans 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

```
// 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

- The size of each array is displayed. Keep in mind that the value of **length** has nothing to do

with the number of elements that are in use. It only reflects the number of elements that the array is designed to hold.

INTRODUCING NESTEAD AND INNER CLASSES.

- **Java inner class** or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.
- Additionally, it can access all the members of the outer class, including private data members and methods.

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

Advantage of Java inner classes

- There are three advantages of inner classes in Java. They are as follows:
- Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
- Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
- Code Optimization: It requires less code to write.

Need of Java Inner class

- Definition: Inner classes are a type of nested class that is non-static and defined within the scope of an instance of the outer class.

Types:

- Member Inner Class: Defined at the member level of a class but not inside a method.
- Local Inner Class: Defined inside a method.
- Anonymous Inner Class: A class without a name, often used for one-time use.
- Static Nested Class: Similar to inner classes but marked as static.
- The difference between nested class and inner class is: An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

```
public class OuterClass {
  private int outerField;

  public class InnerClass {
    private int innerField;

    public void innerMethod() {
      // Access outer class members
      outerField = 10;
    }
  }
}
```