

Chapter 6

Introducing Classes

General form of Class

```
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  
    }  
}
```

General form of Class

- Data and variable defined within class are called instance variable
- 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 general rule, it is the methods that determine how a class' data can be used
- Java classes do not need to have a main() method

A simple Class

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

- . A class defines a new data type
- . A class declaration only creates a template; it does not create an actual object
- . To actually create a object
- . `Box mybox = new Box();` // create a Box object called mybox
- . To assign a value to width of mybox
- . `Mybox.width = 125`

A simple 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);
    }
}
```

A simple Class

```
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);  
    }  
}
```

A simple Class

- The java program should be stored in file named as BoxDemo.java
- Compiling the program generates two .class files separate for each defined
- Both classes can be defined in two separate .java files also

A simple Class

```
// 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;


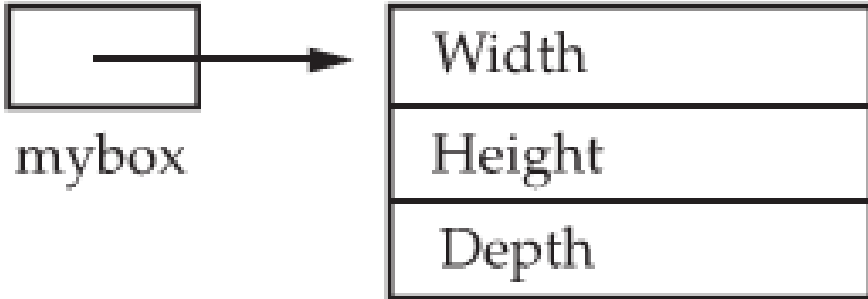
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);

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


Declaring Objects

- Obtaining objects of a class is a two-step process. First, you must declare a variable of the class type and second, you must acquire an actual, physical copy of the object and assign it to that variable
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it
- ```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```
- After first line, the mybox contains value **null**, which indicates that it does not yet point to an actual object. Any attempt to use mybox here results in compile time error
- The next line allocates an actual object and assigns a reference to it to **mybox**
- Now mybox can be used as object of Box
- Initializing other variables like int, short, char etc does not need **new** keyword as they are not classes
- Due to finite memory available, if new can not allocate memory, it generates exception

# Declaring Objects

| <u>Statement</u>                | <u>Effect</u>                                                                                      |
|---------------------------------|----------------------------------------------------------------------------------------------------|
| <code>Box mybox;</code>         | <br>mybox       |
| <code>mybox = new Box();</code> | <br>Box object |

# Assigning Object Reference Variables

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

- 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.
- Basically they are same object with different names
- However they are not linked in any way. 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;
```

---

**REMEMBER** *When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.*

# Methods

```
// This program declares two Box objects.

// 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();
 }
}
```

# Return values from methods

- Instead of printing the results of computations within a method, it is always desirable to return the results.

```
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);
 }
}
```

# Parameterized Methods

- Most of the methods needs parameters

```
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);
 }
}
```

# Constructors

- Most of the objects need initialization
- Java allows objects to initialize themselves when they are created
- A **constructor** initializes an object immediately upon creation
- It has the same name as the class in which it resides and is syntactically similar to a method
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes
- Constructors never return values, its return type is not even **void**
- This is because the implicit return type of a class' constructor is the class type itself

# Constructors

```
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);
 }
}
```



# Parameterized 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;
 }
}
```

```
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);
 }
}
```

# Garbage Collection

- Objects are dynamically allocated by using the new operator
- In C++, dynamically allocated objects must be manually released by use of a delete operator
- Java takes a different approach; it handles deallocation for you automatically
- 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
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects
- There is no explicit need to destroy objects as in C++
- Normally a programmer should not think about destroying objects in java

# 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
- You can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector
- Right before an asset is freed, the Java run time calls the **finalize()** method on the object

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

- the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class
- It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope

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
- You can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector
- Right before an asset is freed, the Java run time calls the **finalize()** method on the object

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

- the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class
- It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope

# Case study: The Stack class

```
// 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--];
 }
}
```

# Case study: The Stack class

```
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());
 }
}
```

# Case study: The Stack class

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
```