UNIT 1

Introducing Classes

Topics Covered

- Class Fundamentals
- Declaring Objects
- A closer look at new
- Assigning Object Reference variables
- Introducing Methods
- Constructors
- Parameterized Constructors
- The this keyword
- Instance variable hiding
- Garbage Collection()
- The finalize() method
- A Stack class

Class Fundamentals

- 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.
- As such, the class forms the basis for object-oriented programming in Java.
- Any concept we wish to implement in a Java program must be encapsulated within a class.

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

The General Form of a Class

- A class is declared by use of the class keyword.
- 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 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 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.
- Thus, the data for one object is separate and unique from the data for another.
- All methods have the same general form as main( ).
- Java classes do not need to have a main( ) method.
- We only specify one if that class is the starting point for the program.
- Further, some kinds of Java applications, such as 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 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. We 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 code does not cause any objects of type Box to come into existence.

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

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

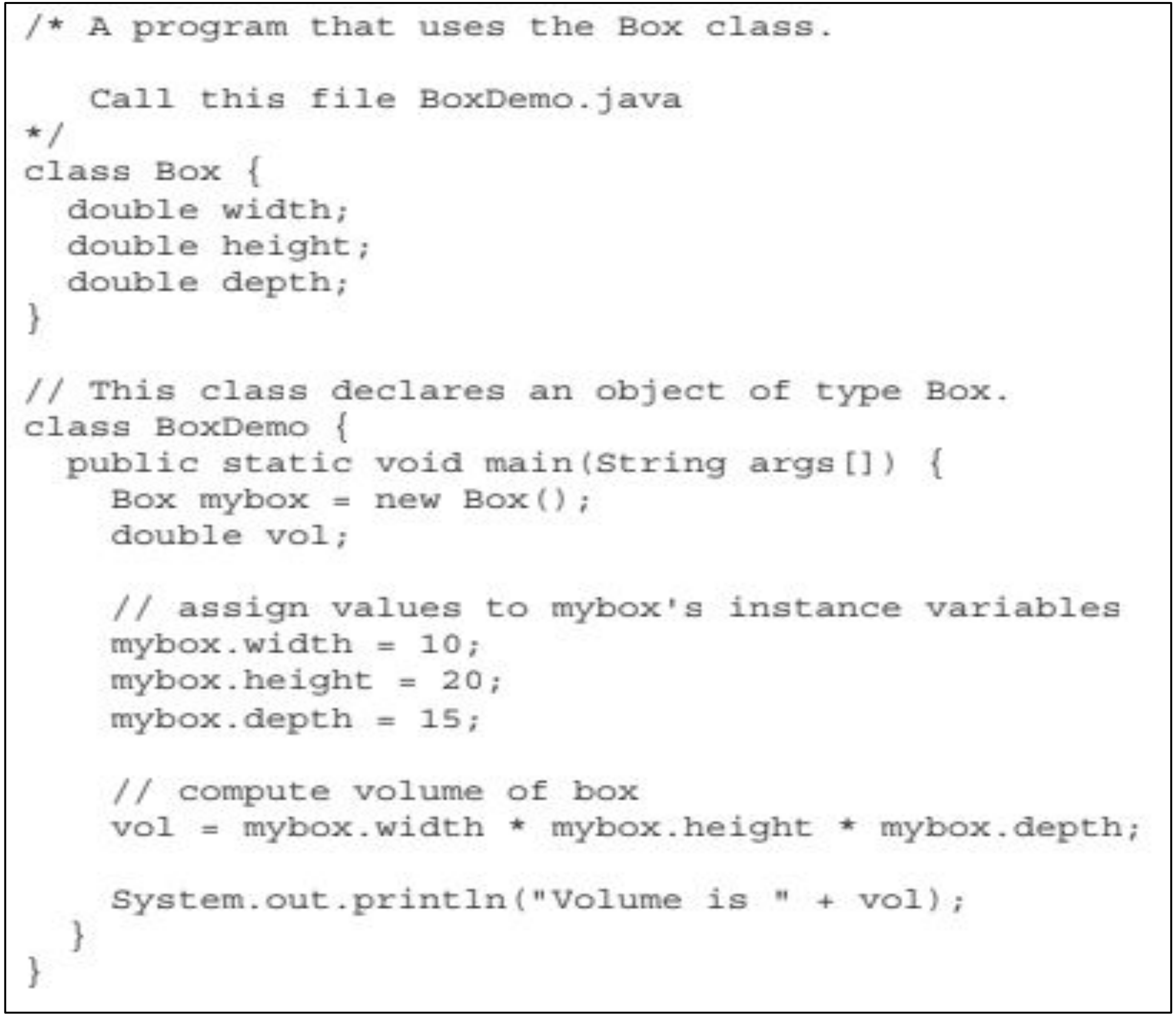
After this statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality.

- Each time we create an instance of a class, we 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, we will use the dot (.) operator.
- The dot operator links the name of the object with the name of an instance variable.

To assign the width variable of mybox the value 100, we 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.



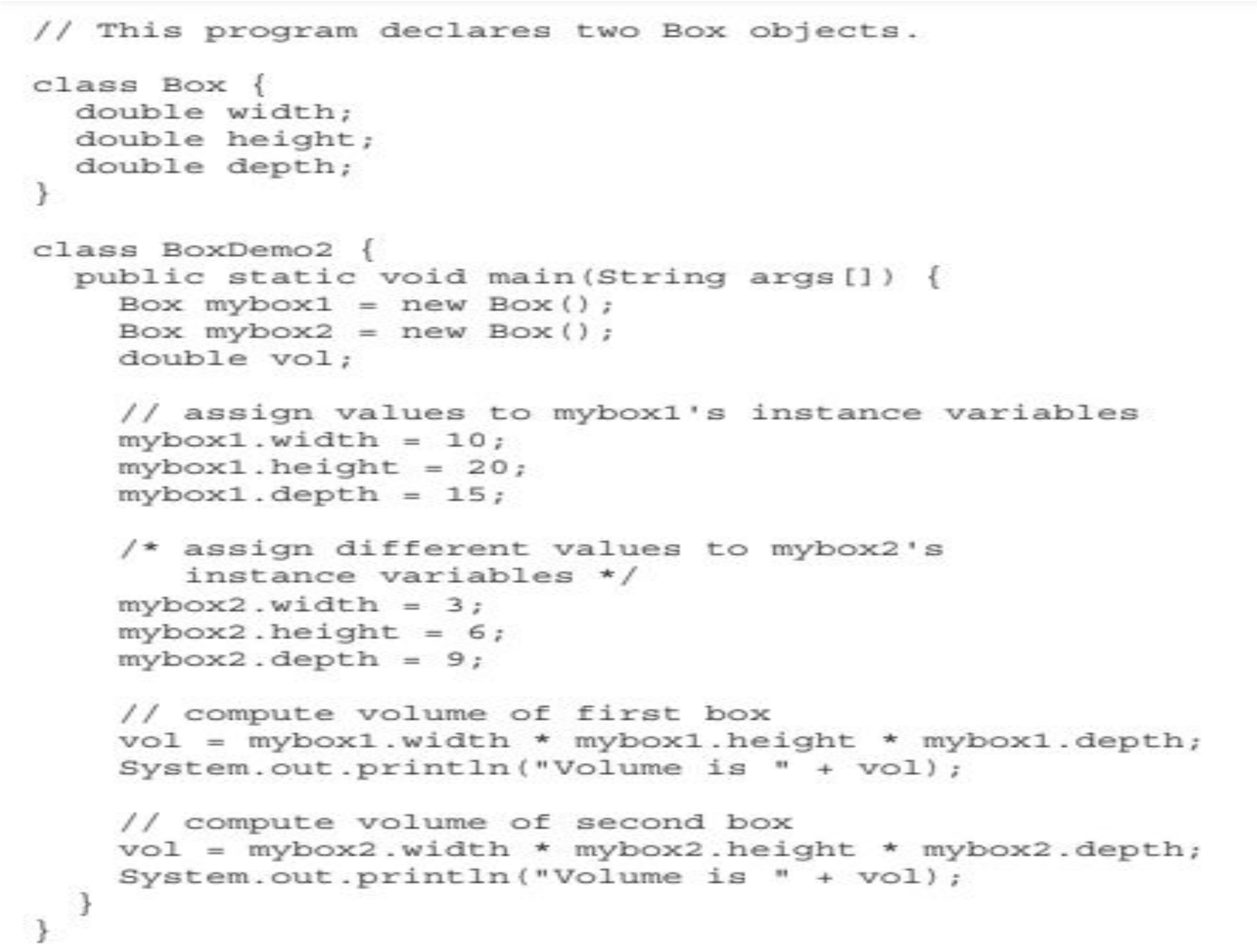
```
/* A program that uses the Box class.

   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);
  }
}
```

- We 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 we compile this program, we 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.

- To run this program, we must execute BoxDemo.class. When we do, we will see the following output:
  Volume is 3000.0

- Each object has its own copies of the instance variables.
- This means that if we 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.
- This is explained with following example



```
// 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
```

As we can see, mybox1’s data is completely separate from the data contained in mybox2.

Declaring Objects

- When we create a class, we are creating a new data type.
- We can use this type to declare objects of that type.
- However, obtaining objects of a class is a two-step process.
    - ❑ First, we 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, we must acquire an actual, physical copy of the object and assign it to that variable. We 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 the address in memory of the object allocated by new. This reference is then stored in the variable.
- Thus, in Java, all class objects must be dynamically allocated.

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

This statement combines two steps. It can be rewritten as

```
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, we can use mybox as if it were a Box object.

The effect of these two lines of code is depicted in Figure.

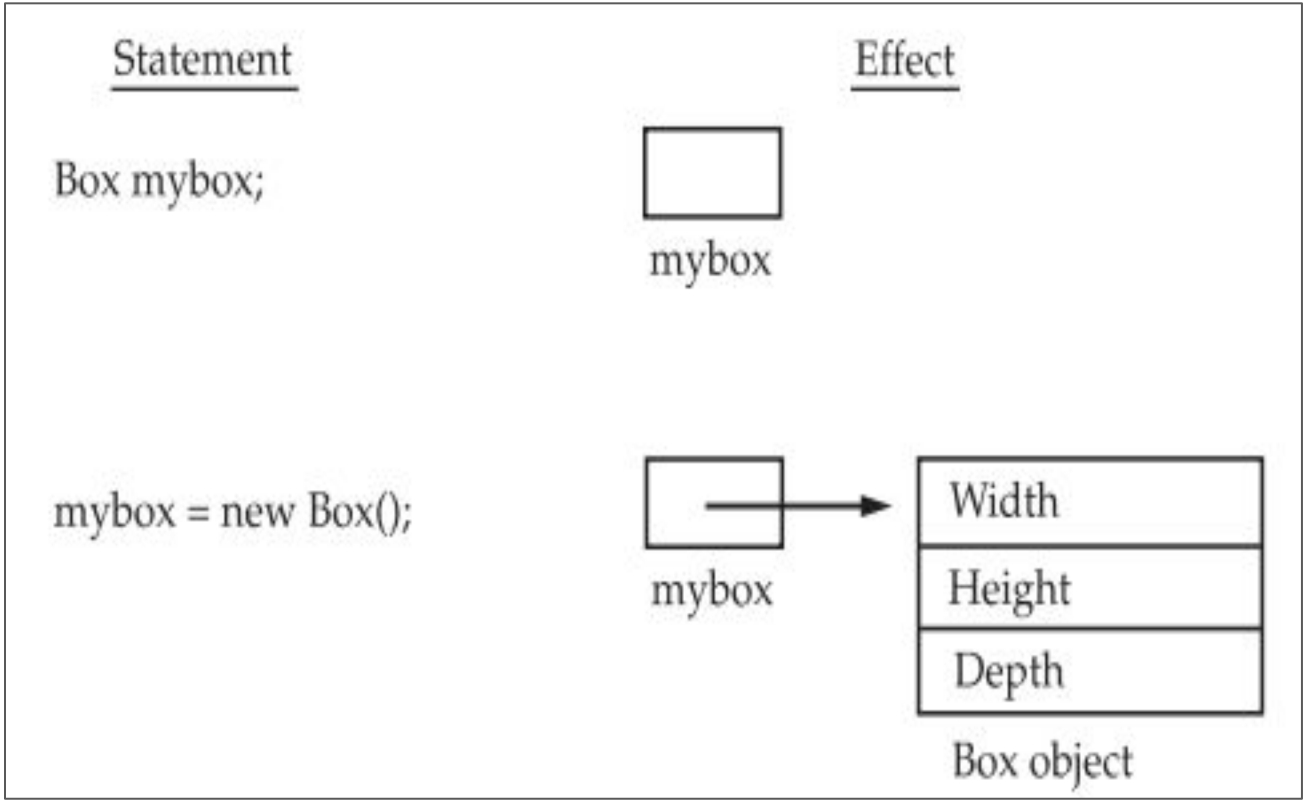


Figure: Declaring an object of type Box

A Closer Look at new

The new operator dynamically allocates memory for an object. 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. This is the case with Box.

It is important to understand that new allocates memory for an object during run time. The advantage of this approach is that the program can create as many or as few objects as it needs during the execution of the 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.

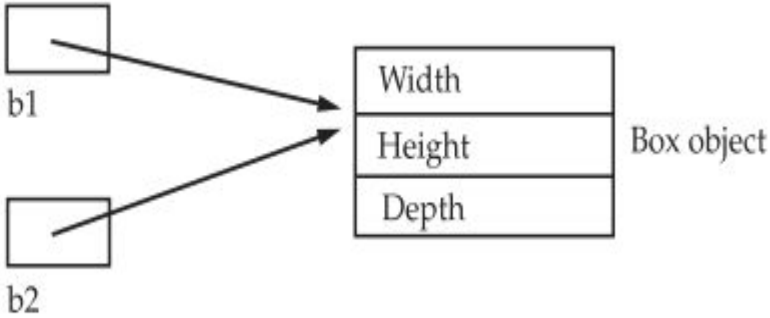
Assigning Object Reference Variables

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

After this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 do 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.

Note: When we assign one object reference variable to another object reference variable, we are not creating a copy of the object, we are only making a copy of the reference.

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

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. 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.

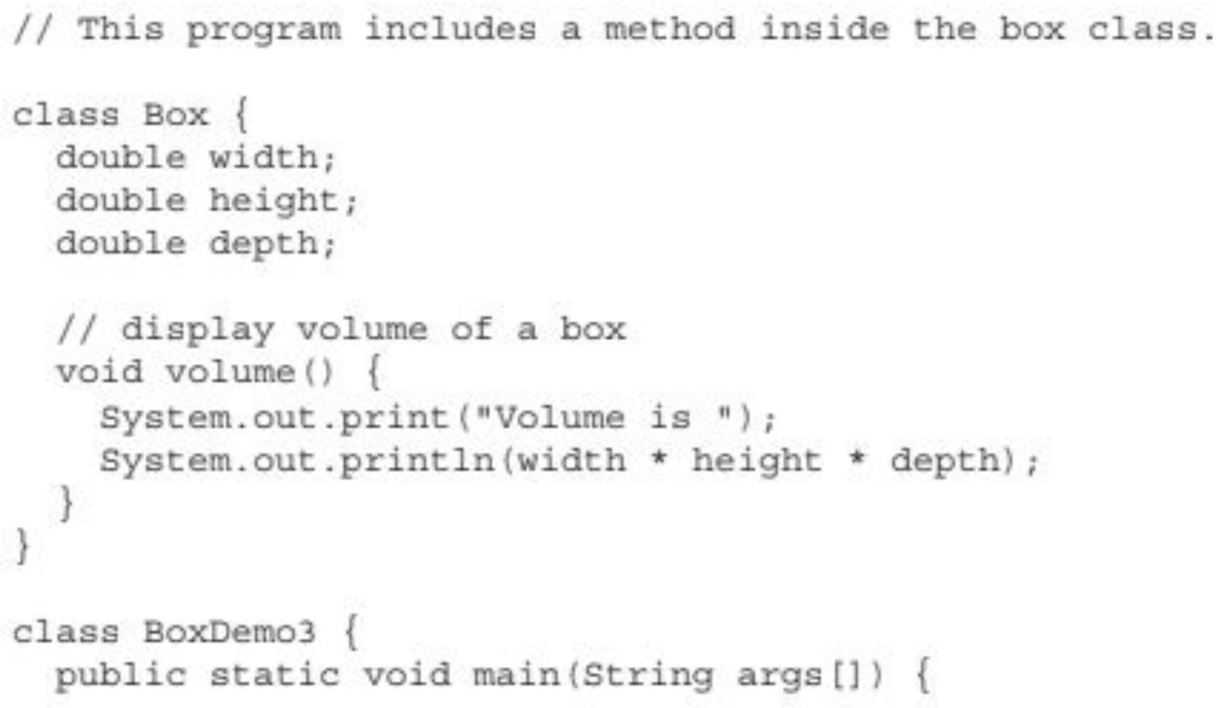
- The parameter-list is a sequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than void return a value to the calling routine
using the following form of the return statement:

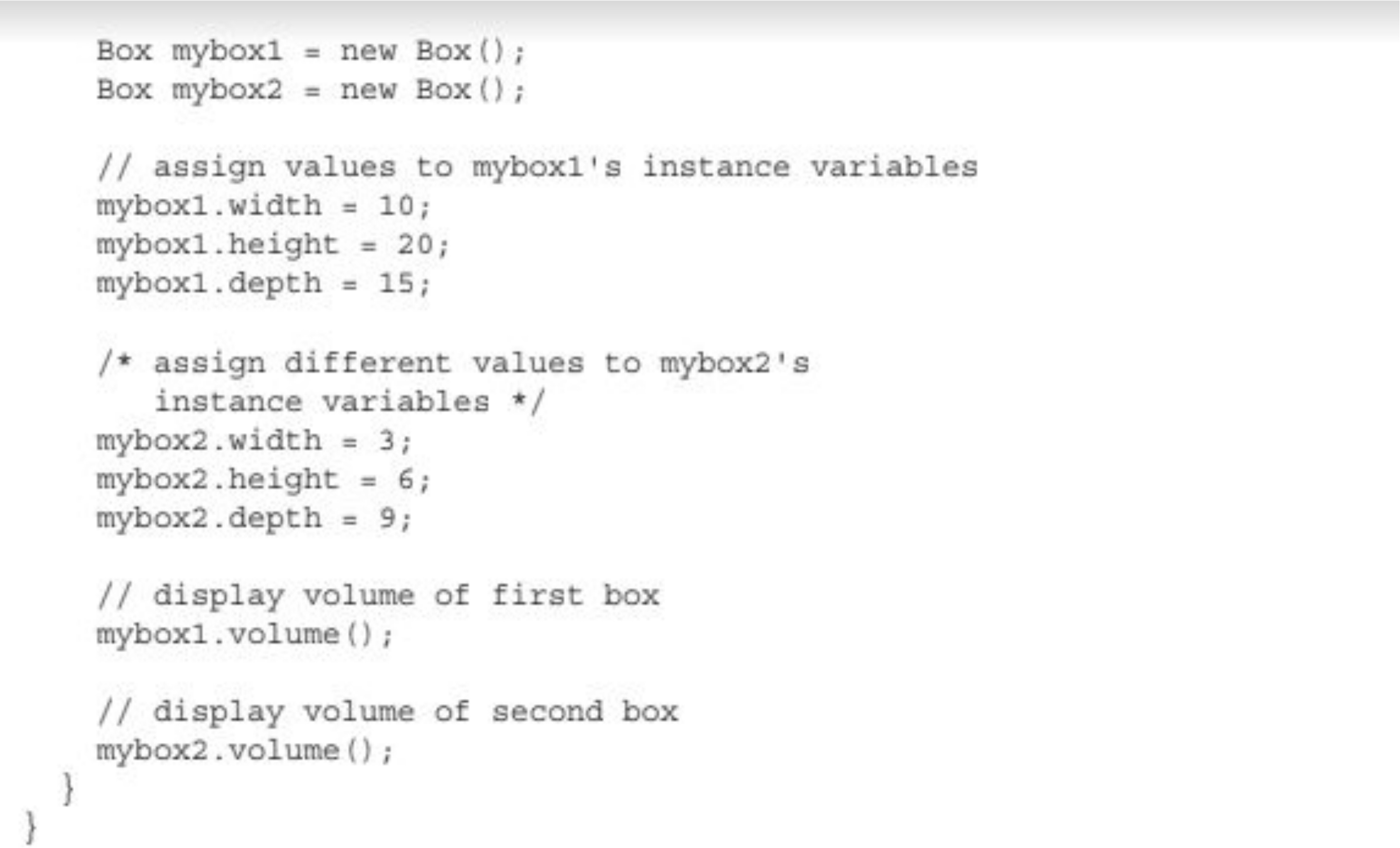
return value;

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
```

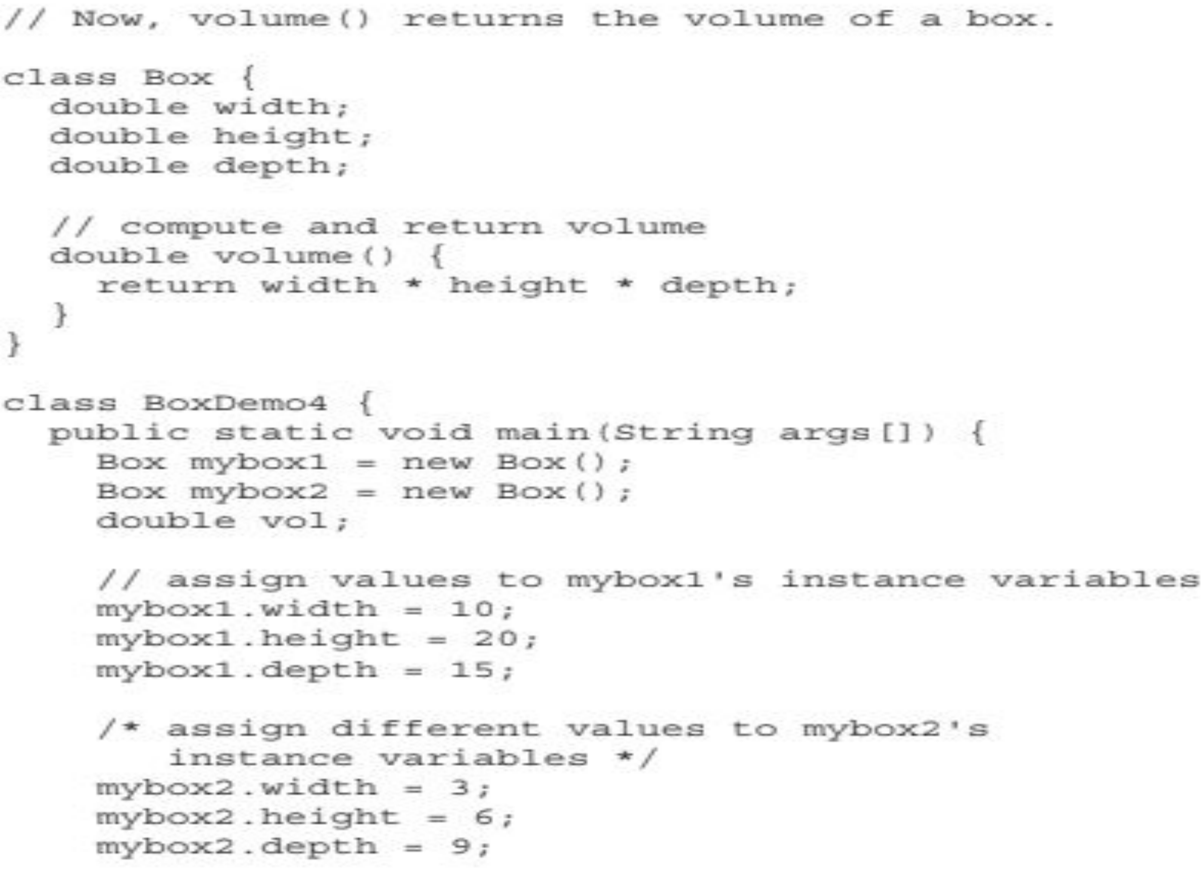
The following two lines of code:

```
mybox1.volume();
mybox2.volume();
```

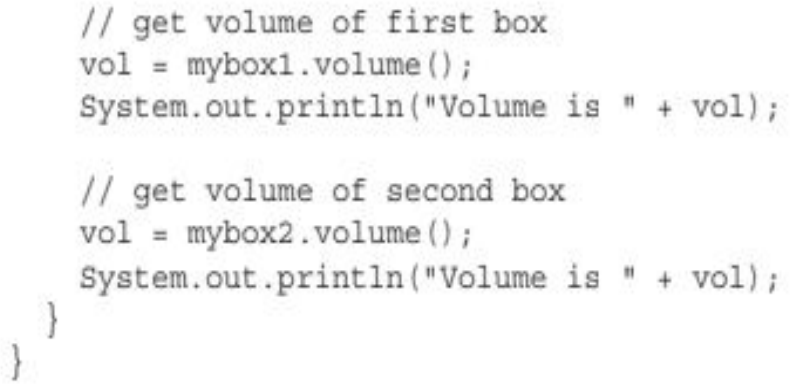
The first line here invokes the volume( ) method on mybox1. That is, it calls volume( ) relative to the mybox1 object, using the object's name followed by the dot operator. Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1, and the call to mybox2.volume( ) displays the volume of the box defined by mybox2. Each time volume( ) is invoked, it displays the volume for the specified box.

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. 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( ).

Returning a Value



```
// 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);
  }
}
```

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, we could not return an integer.
- The variable receiving the value returned by a method (such as vol, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters

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. 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 we modify the method so that it takes a parameter, as shown next, then we can make square( ) much more useful.

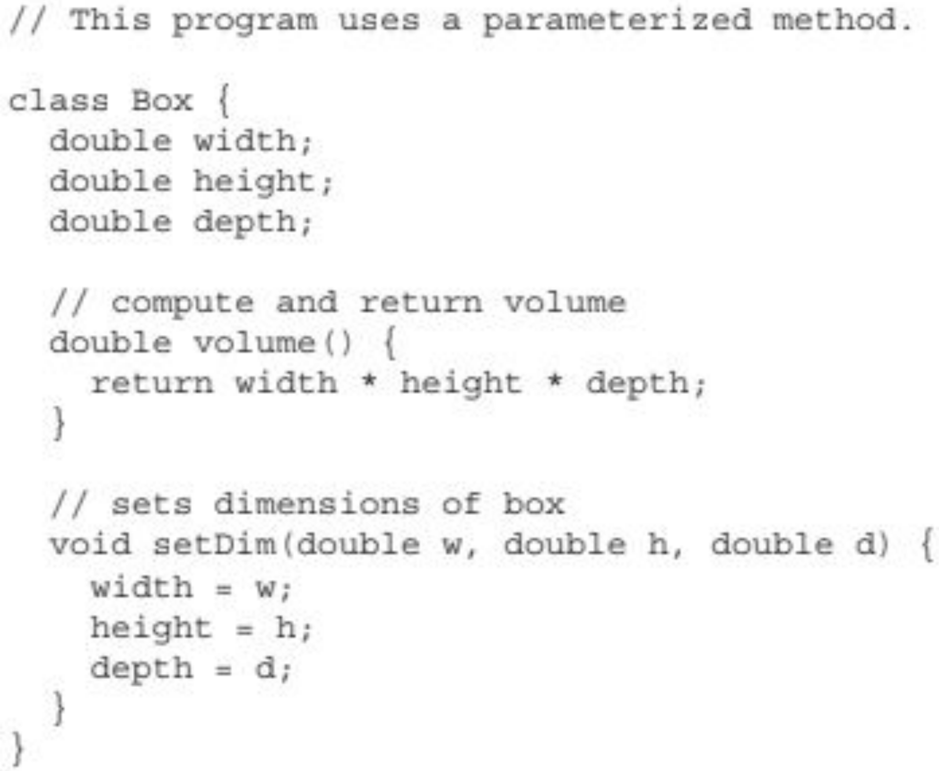
```
int square(int i)
{
return i * i;
}
```

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

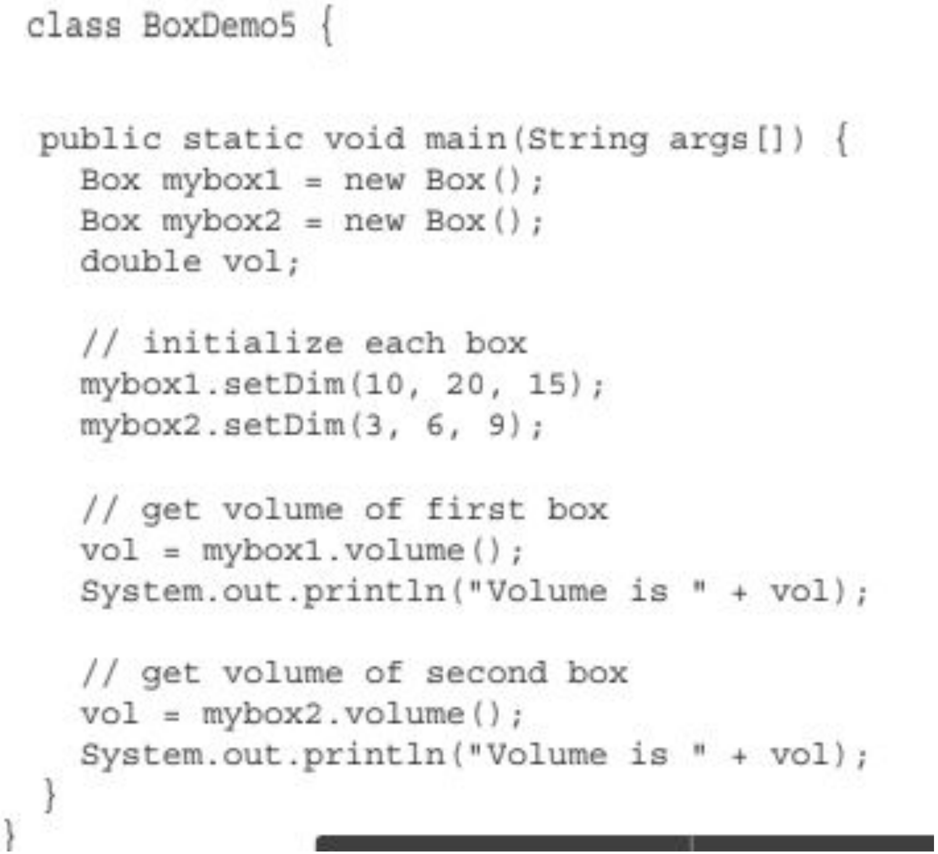
Here is an example:

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

In the first call to square( ), the value 5 will be passed into parameter i. In the second call, i will receive the value 9. The third invocation passes the value of y, which is 2 in this example. As these examples show, square( ) is able to return the square of whatever data it is passed.



```
// 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);

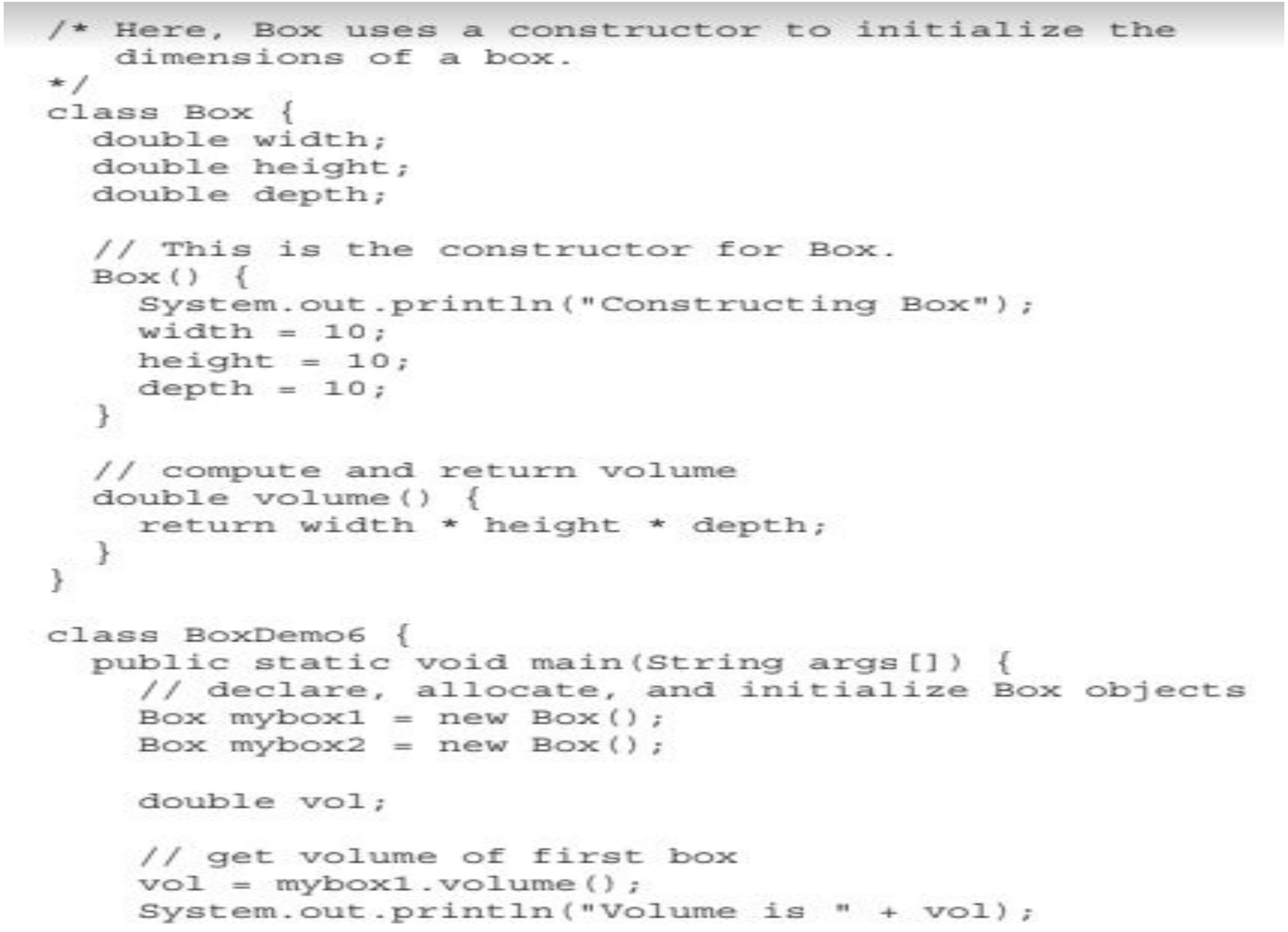
is executed, 10 is copied into parameter w, 20 is copied into h, and 15 is copied into d.

Inside ***setDim( )*** the values of w, h, and d are then assigned to width, height and depth

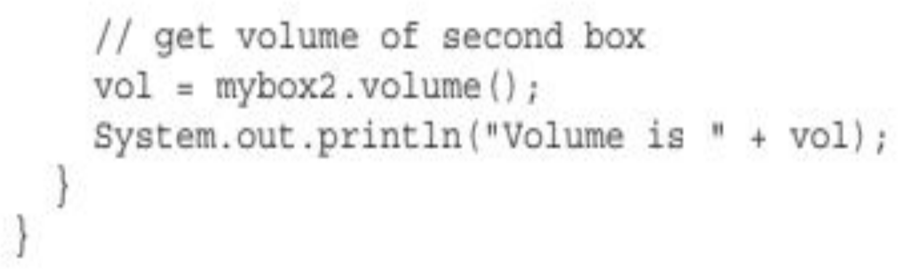
respectively.

Constructors

- 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 look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.



```
/* 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
```

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.

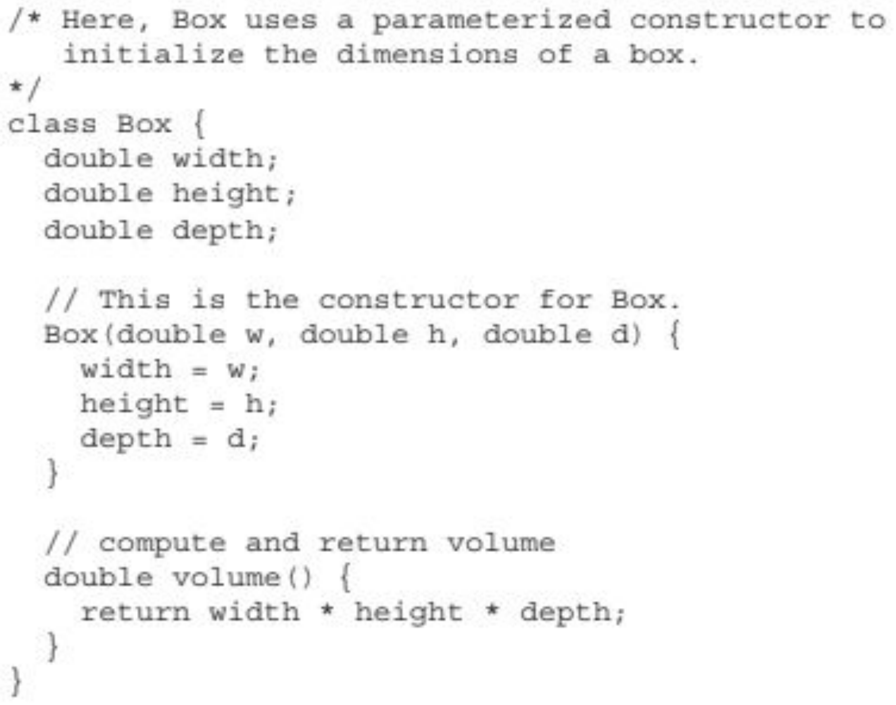
Thus, in the line

```
    Box mybox1 = new Box();
```

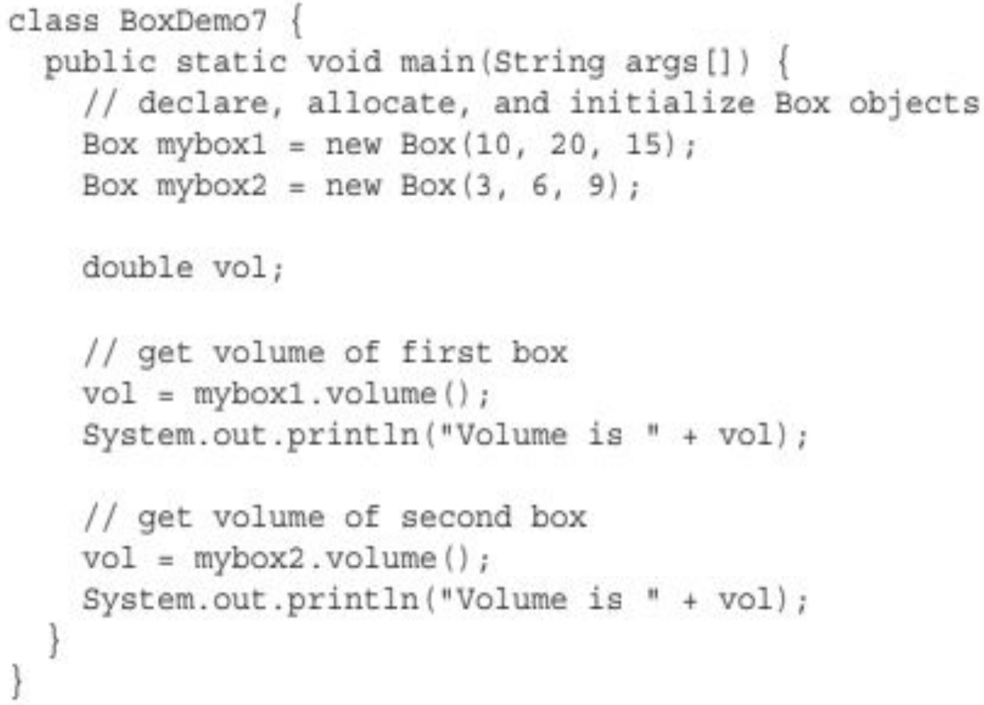
new Box( ) is calling the Box( ) constructor. When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class. The default constructor automatically initializes all instance variables to their default values, which are zero, null, and false, for numeric types, reference types and boolean, respectively. The default constructor is often sufficient for simple classes, but it usually won’t do for more sophisticated ones. Once we define our own constructor, the default constructor is no longer used.

Parameterized Constructors

In order to construct Box objects of various dimensions, the easy solution is to add parameters to the constructor.



```
/* 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;
  }

  // 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

As we can see, each object is initialized as specified in the parameters to its constructor.

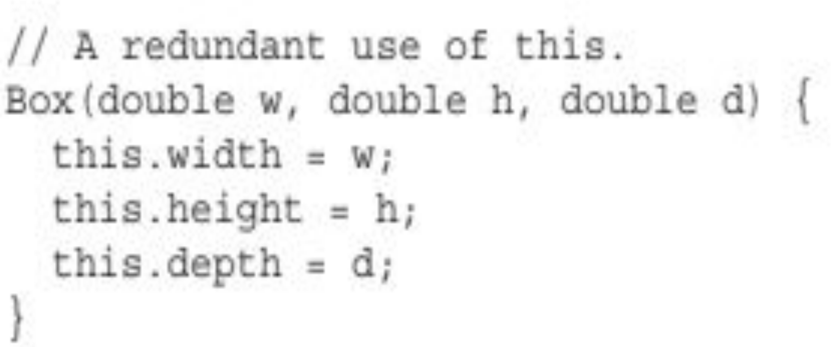
For example, in the following line,

```
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.

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.



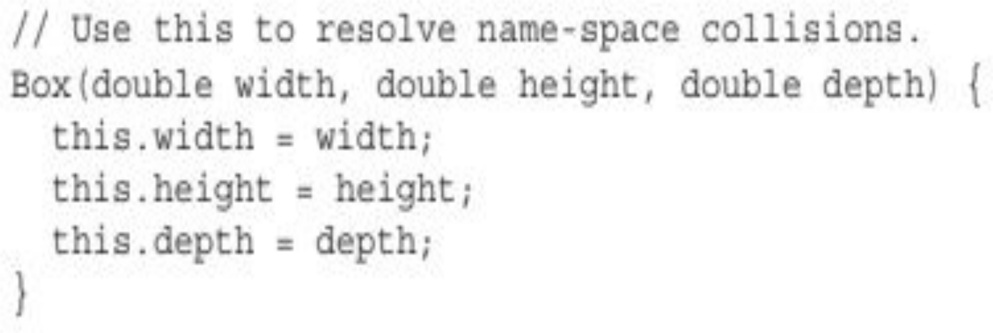
```
// A redundant use of this.
Box(double w, double h, double d) {
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

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, we can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.
- 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, for example, would have referred to the formal parameter, hiding the instance variable width.
- Because ***this*** refer directly to the object, we 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

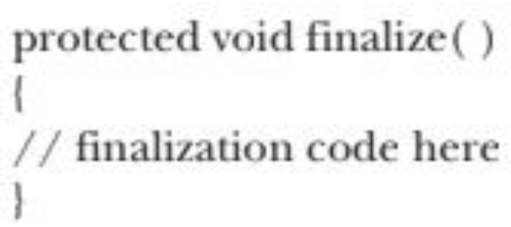
- Objects are dynamically allocated by using the ***new*** operator
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a ***delete*** operator.
- Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called **garbage collection.**
- It works like this: when no references to an object exist, that object is assumed to be no longer needed and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs occasionally during the execution of program. It will not occur simply because one or more objects exist that are no longer used.

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or character font, then we 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, we 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, we simply define the finalize( ) method.
- The Java runtime calls that method whenever it is about to recycle an object of that class.
- Inside the finalize( ) method, we 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 runtime calls the finalize( ) method on the object.

The finalize( ) method has this general form:



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

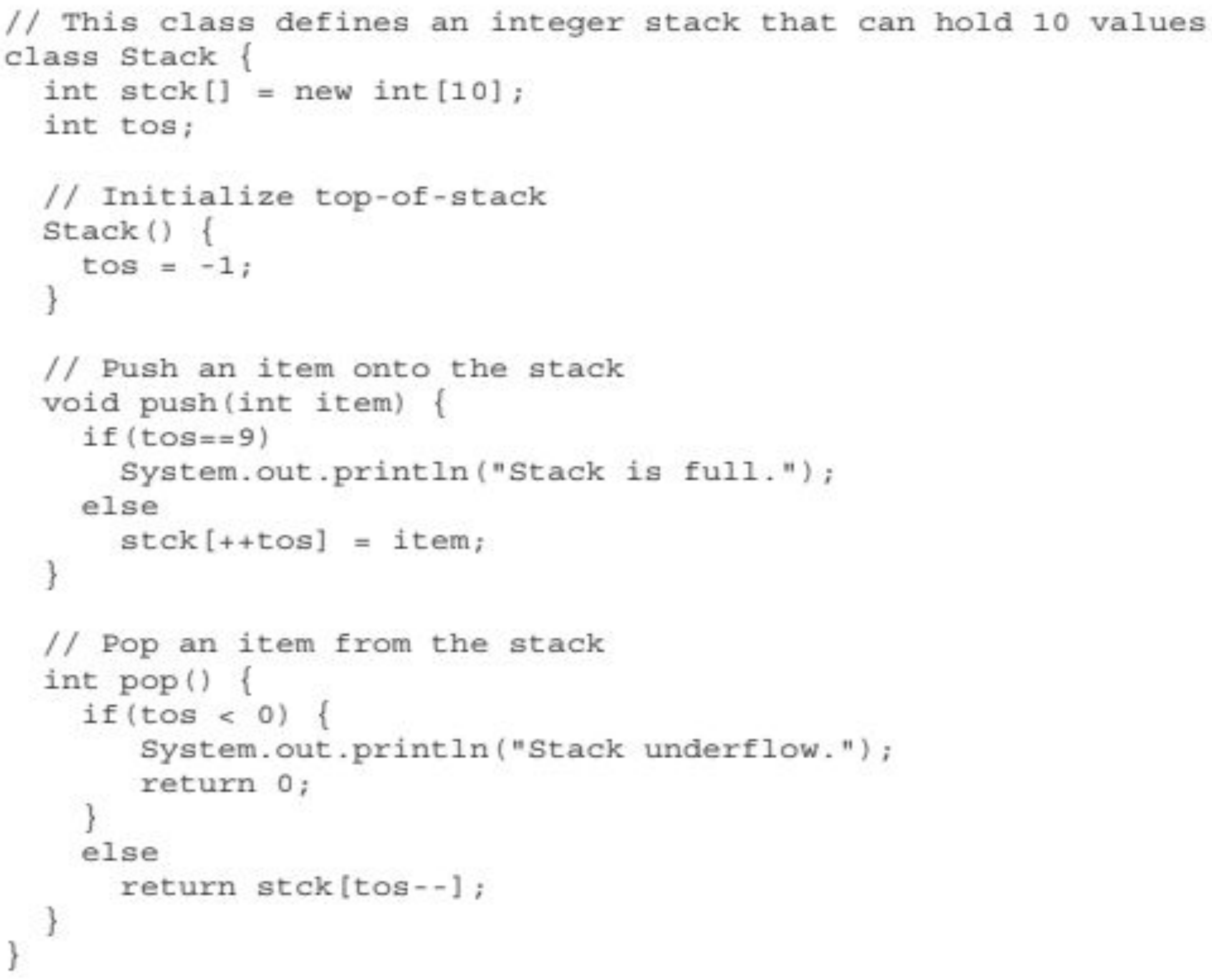
Here, the keyword protected is a specifier that limits access to finalize( ).

- **finalize( )** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example.

Note: C++ allows to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The finalize( ) method only approximates the function of a destructor.

A Stack Class

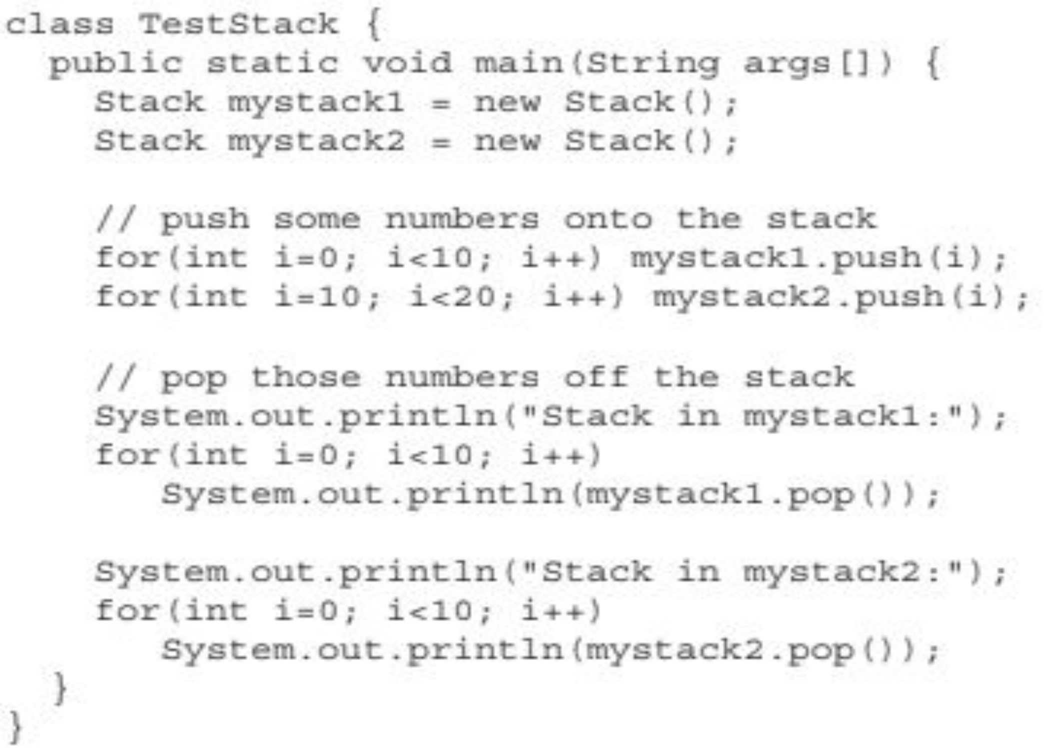
- A stack stores data using first-in, last-out ordering.
- That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used.
- Stacks are controlled through two operations traditionally called push and pop.
- To put an item on top of the stack, we will use push.
- To take an item off the stack, we will use pop.



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

Here, 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( ).

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
```