PROGRAMMING IN JAVA

RAMAIAH
Institute of Technology

Dr. Manish Kumar
Assistant Professor
Department of Master of Computer Applications
M S Ramaiah Institute of Technology
Bangalore-54

Chapter 6 – Introducing Classes

Class Fundamentals

- Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class.

The General Form of a Class

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

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

- The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*.
- Collectively, the methods and variables defined within a class are called *members* of the class. 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.
- 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.
- You only specify one if that class is the starting point for your program. Further, some kinds of Java applications, such as applets, don't require a **main( )** method at all.

A Simple Class

Let’s begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

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

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

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

- 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);
}
}
```

To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, 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**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height *
mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height *
mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

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

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

Declaring Objects

- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let’s look at the details of this procedure.

- In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

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

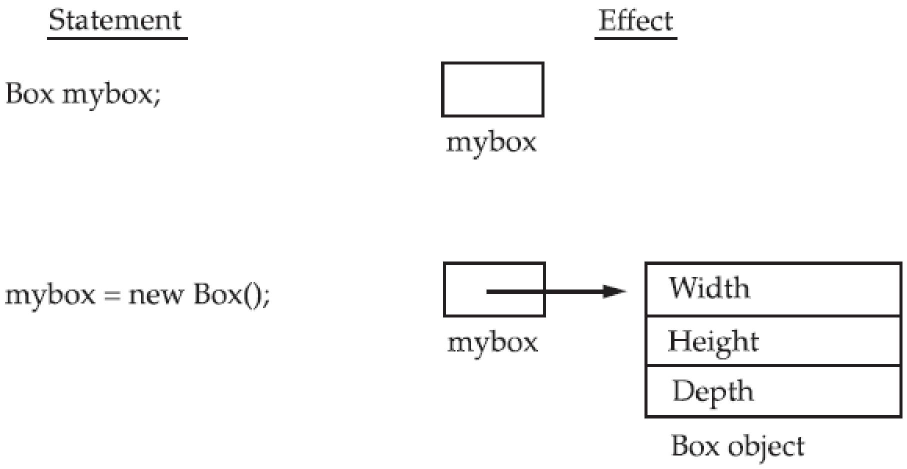
This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new *classname* ( );



Statement

Effect

Box mybox;

mybox

mybox = new Box();

mybox

Width

Height

Depth

Box object

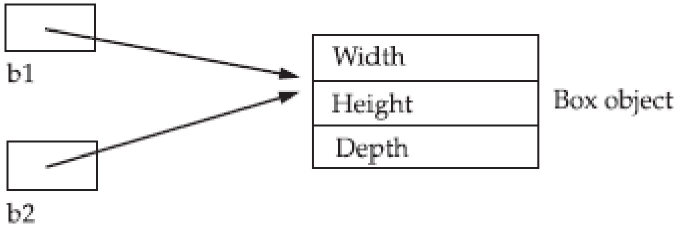
-1 Declaring an object of type **Box**

Assigning Object Reference Variables

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

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

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
}
```

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
```

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);
}
}
```

Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

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

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

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

```
// 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;
}
}
```

```
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

- 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
```

Parameterized Constructors

```
/* 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 you 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

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

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

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.

- 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 for you automatically.

- The technique that accomplishes this is called *garbage collection*. It works like this:

- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

- There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program.

- It will not occur simply because one or more objects exist that are no longer used.

- 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
}
```

- It is important to understand that **finalize( )** is only called just prior to garbage collection.

- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed.

- Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

A 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--];
}
}
```

```
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:           Stack in mystack2:
9                            19
8                            18
7                            17
6                            16
5                            15
4                            14
3                            13
2                            12
1                            11
0                            10
```

Questions !

Thank You!