Programming in Java

Introducing Classes

Mahesh Kumar
(maheshkumar@andc.du.ac.in)

Course Web Page
(www.mkbhandari.com/mkwiki)

Outline

1. Class Fundamentals
2. Declaring Objects
3. Introducing Methods
4. Constructors
5. The this Keyword
6. Garbage Collection
7. The finalize( ) Method

Class Fundamentals

- Class is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- The class forms the basis for object-oriented programming in Java.
- Any concept you wish to implement in a Java program must be encapsulated within a class.
- 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 or blueprint* for an object, and an object is an *instance* of a class.
- Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

- When you define a class, you declare its exact form and nature (the data that it contains and the code that operates on that data).
- While very simple classes may contain only code or only data, most real-world classes contain both.
- A class' code defines the interface to its data.
- A class is declared by use of the **class** keyword.

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
        }
}
```

1. *class:* A class is declared by use of the class keyword.
2. *Instance Variables:* The data, or variables, defined within a class are called instance variables.

   Because each instance of the class contains its own copy of variables.

   Thus, the data for one object is separate and unique from the data for another.
3. *Methods:* The code is contained within methods and they determine how a class' data can be used.
4. *Members:* The methods and variables defined within a class are called members of the class.

A Simple Class

```
// A program that uses the Box 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);
        }
}
```

1. *Class Box* defines three instance variables: width, height, and depth.
2. *In this example Box* does not contain any methods.
3. As class defines a new type of data, in this case, the new data type is called Box.
4. A class declaration only creates a template; it does not create an actual object.
5. To actually create a Box object, you will use a statement like the following:

   Box mybox = new Box( );
6. After the above statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality.

A Simple Class

```
// A program that uses the Box 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);
        }
}
```

7. Every Box object will contain its own copies of the instance variables width, height, and depth.
8. To access these variables, you will use the dot (.) operator.
9. The dot operator links the name of the object with the name of an instance variable.

   mybox.width = 100;

10. In general, you use the dot operator to access both the instance variables and the methods within an object.
11. One other point: Although commonly referred to as the dot operator, the formal specification for Java categorizes the dot (.) as a separator.

A Simple Class

```
// A program that uses the Box 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);
        }
}
```

Q1 What will be the name of file for this program?

Q2 How many .class files will be created when you compile this program?

Q3 To run this program, which .class file must be executed?

Q4 What will be the output of this program?

A Simple Class

```
// A program that uses the Box 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);
        }
}
```

Q1 What will be the name of file for this program?

BoxDemo.java

Q2 How many .class files will be created when you compile this program?

*Two .class files, one for **Box** and one for **BoxDemo***

Q3 To run this program, which .class file must be executed?

BoxDemo.class

Q4 What will be the output of this program?

Volume is 3000.0

A Simple Class with two objects

```
// This program declares two Box objects.
// class Box is same as previous program
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 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);
        }
}
```

1. Each object has its own copies of the instance variables.
2. It means if you have two Box objects, each has its own copy of depth, width, and height.
3. Changes to the instance variables of one object have no effect on the instance variables of another.

Q. What will be the output of this program?

A Simple Class with two objects

```
// This program declares two Box objects.
// class Box is same as previous program
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 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);
        }
}
```

1. Each object has its own copies of the instance variables.

2. It means if you have two Box objects, each has its own copy of depth, width, and height.

3. Changes to the instance variables of one object have no effect on the instance variables of another.

Q. What will be the output of this program?

Volume is 3000.0
Volume is 162.0

T How will you declare 50 or 100 objects?

A Simple Class with two objects

```
// This program declares two Box objects.
// class Box is same as previous program
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 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);
        }
}
```

1. Each object has its own copies of the instance variables.

2. It means if you have two Box objects, each has its own copy of depth, width, and height.

3. Changes to the instance variables of one object have no effect on the instance variables of another.

Q. What will be the output of this program?

Volume is 3000.0
Volume is 162.0

T. How will you declare 50 or 100 objects?
Box nboxes[ ]= new Box[50];

Declaring Objects

- When you create a class, you are creating a new data type, and this new data type can be used to declare objects of that type.

- However, obtaining objects of a class is a two-step process.

  1. 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.

  2. Acquire an actual, physical copy of the object and assign it to that variable 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, 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.**

Declaring Objects

- 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
```

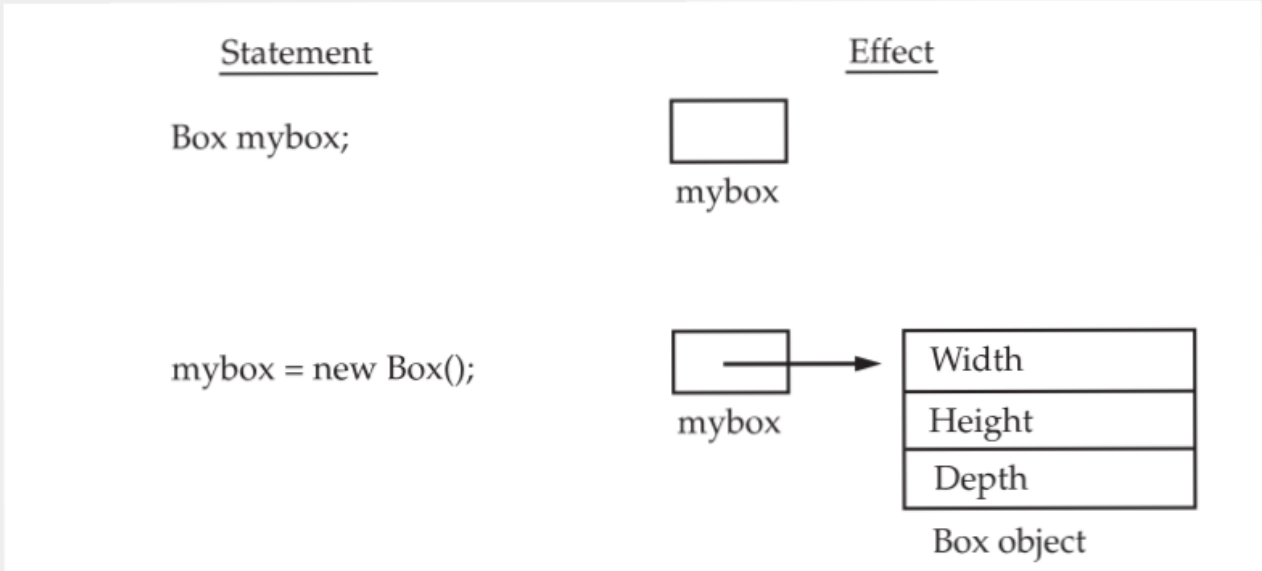


Figure 6-1 Declaring an object of type **Box**

A Closer Look at new

- The new operator dynamically allocates memory for an object. It has this general form:

```
class-var = new classname( );
```

1. *class-var* is a variable of the class type being created.
2. The *classname* is the name of the class that is being instantiated.
3. The class name followed by parentheses specifies the *constructor* for the class.
   - 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.

A Closer Look at new

Q1 What if new will not be able to allocate memory for an object because insufficient memory exists?

Q2 Why you do not need to use new for such things as integers or characters?

A Closer Look at new

Q1 What if new will not be able to allocate memory for an object because insufficient memory exists?

- If this happens, a run-time exception will occur.

Q2 Why you do not need to use new for such things as integers or characters?

- Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables.

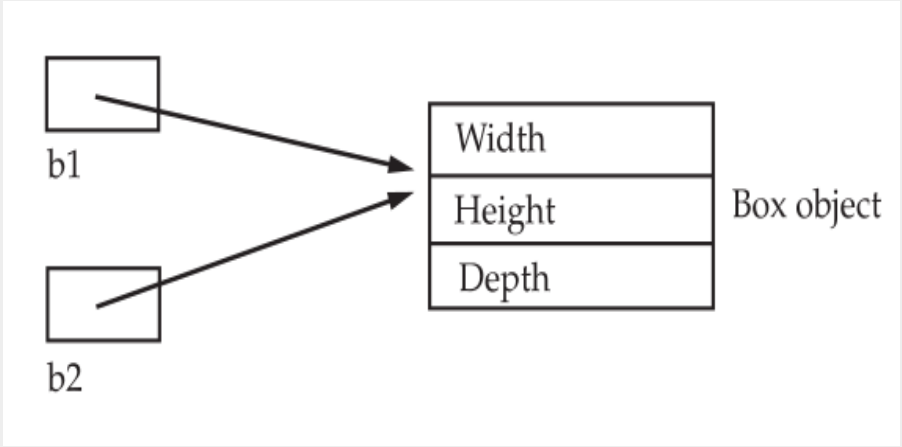
Review the distinction b/w a Class and an Object

- A class creates a new data type that can be used to create objects.
- 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

- What do you think the following fragment does?

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



1. Both b1 and b2 will refer to the *same* object.
2. The assignment of b1 to b2 do not allocate any memory or copy any part of the original object.
3. It simply makes b2 refer to the same object as does b1.
4. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

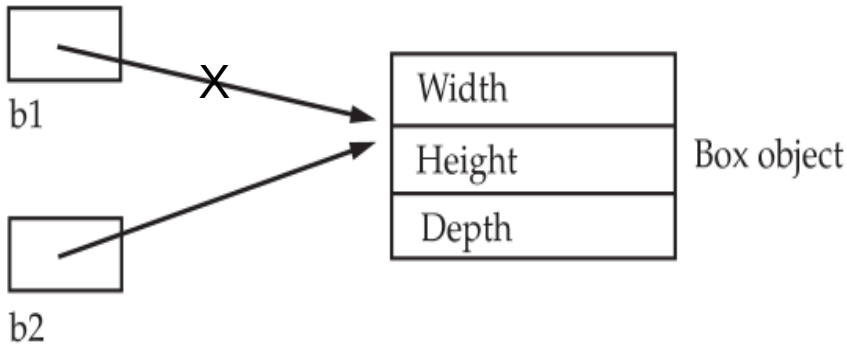
Q What will happen after following assignment?

```
Box b1 = new Box( );
Box b2 = b1;
// ...
b1 = null;
```

Assigning Object Reference Variables

- What do you think the following fragment does?

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



1. Both b1 and b2 will refer to the *same* object.
2. The assignment of b1 to b2 do not allocate any memory or copy any part of the original object.
3. It simply makes b2 refer to the same object as does b1.
4. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

Q What will happen after following assignment?

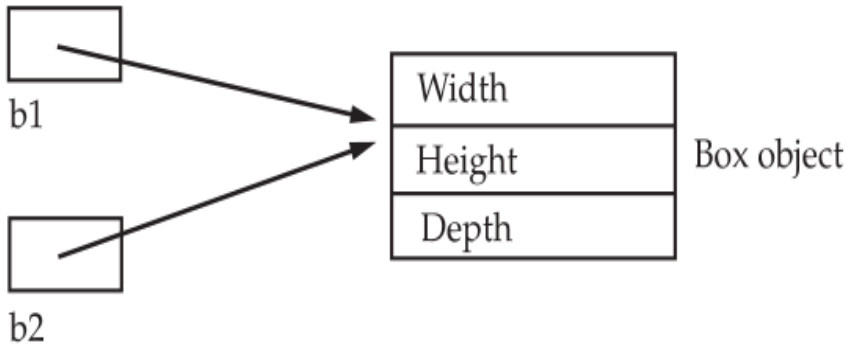
```
Box b1 = new Box( );
Box b2 = b1;
// ...
b1 = null;
```

b1 is set to **null**, b2 still points to the original object.

Assigning Object Reference Variables

- What do you think the following fragment does?

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



1. Both b1 and b2 will refer to the *same* object.
2. The assignment of b1 to b2 do not allocate any memory or copy any part of the original object.
3. It simply makes b2 refer to the same object as does b1.
4. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

Rem. 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.

Introducing Methods

- A class usually consist of two things: instance variables and methods.
- The general form of a method is:

```
type name(parameter-list)
{
        // body of method

}
```

1. *type:* specifies the type of data returned by the method. This can be:
   - Any valid type – including class types that you create.
   - void – if the method does not return a value.
2. *name:* any legal identifier.
3. 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.

Introducing Methods

- A class usually consist of two things: instance variables and methods.
- The general form of a method is:

```
type name(parameter-list)
{
        // body of method

}
```

4. 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;*

Adding a Method to the Box Class

- Most of the time, you will use methods to access the instance variables defined by the class.
- Methods define the interface to most classes.
- This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions.
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

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);
        }
}
```

1. 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.

2. 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.

3. The reason is: A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known.

Adding a Method to the Box Class

```
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 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( );
        }
}
```

Q. What will be the output of this program?

Volume is 3000.0
Volume is 162.0

Returning a Value

```
// In this program 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 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);
        }
}
```

Returning a Value

- There are two important things to understand about returning values:

  1. The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.

  2. 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.

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

- When println( ) is executed, mybox1.volume( ) will be called automatically and its value will be passed to println( ).

Adding a Method That Takes Parameters

- While some methods don't need parameters, most do.
- Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

```
// Returns the value of 10 squared
int square( ) {
        return 10 * 10;
}
```

```
// Returns the square of whatever value it is called with.
int square(int i) {
        return i * i;
}
```

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

Adding a Method That Takes Parameters

```
// 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;
        }
}
```

Adding a Method That Takes Parameters

```
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);
        }
}
```

C. Concepts in methods:
method calls/invocation,
arguments,
parameters,
return values
Return types

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 but they have no return type, not even void.
  - → Once defined, the constructor is automatically called when the object is created, before the **new** operator completes.
  - → 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.

Constructors

```
// Here, Box uses a constructor to initialize the dimensions of  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);
        }
}
```

Constructors

- The output of the program is shown here:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

- ***Default constructor***
  - → When you 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.
  - → Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

- The parameterized constructor is used to provide different values to distinct objects.
  - → *Default constructor* - provide the default values to the objects like **0, null, false**, etc., depending on the data type of the instance variables.
  - → *No-argument Constructor* - it is not very useful (**all objects will have the same values**).
  - → *Parameterized Constructor* - a way to construct objects of with different values (by adding parameters to the constructor).

Parameterized Constructors

```
// Here, Box uses a parameterized constructor to initialize the dimensions of  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);
        }
}
```

Parameterized Constructors

- The output of the 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.

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.

```
// Consider the following version of Box( ):
Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
}

// The use of this is redundant, but perfectly correct.
```

The *this* Keyword

- **Instance variable hiding**
  - → Local variables, including formal parameters to methods, which may overlap with the names of the class' instance variables. (it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes)
  - → 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.
  - → So two possible ways to resolve any namespace collision:
    1. Simply use different names
    2. Use ***this*** (because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables)

The *this* Keyword

```
// 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.

  class-var = new classname( );

- How such objects are destroyed and their memory released for later reallocation?
  - → In **C++**, dynamically allocated objects must be manually released by use of a ***delete*** operator. (in **C** *free( )* function is used)
  - → In **Java**, deallocation is performed automatically, which is known as **garbage collection**.
  - → Garbage Collection is process of reclaiming the runtime unused memory automatically.

- ***Advantage of Garbage Collection***
  - → It makes java memory efficient because garbage collector removes the unreferenced objects (*nulling the reference, assigning a reference to another, anonymous objects*) from heap memory.
  - → It is automatically done by the garbage collector(a part of JVM) so we don't need to make extra efforts.

The *finalize( )* method

- The ***finalize()*** method is invoked each time before the object is garbage collected.
  - → It can be used to perform cleanup processing
  - → It is defined in Object class as:
    protected void finalize(){ }
  - → ***Note:*** The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).
- ***gc( ) method***
  - → The ***gc( )*** method is used to invoke the garbage collector to perform cleanup processing.
  - → The ***gc( )*** method is found in System and Runtime classes.
    public static void gc(){ }
  - → ***Note:*** Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Simple Example of garbage collection in java

```
public class TestGarbage1{

    public void finalize( ){
        System.out.println("Object is garbage collected");
    }

    public static void main(String args[ ]){
        TestGarbage1 s1=new TestGarbage1( );
        TestGarbage1 s2=new TestGarbage1( );

        s1=null;
        s2=null;

        System.gc( );
    }
}
```

References

- **Reference for this topic**

1. **Book-** Java: The Complete Reference, Tenth Edition: Herbert Schildt

2. **Web-** https://www.tutorialspoint.com/java/index.htm

3. **Web-** https://www.javatpoint.com/java-tutorial