Java Fundamentals

Siddharth Sharma

Constructors

Constructors:

- It can be tedious to initialise all of the variables in a class each time an instance is created. Even when you add convenience functions like setDim( ), it would be simpler and more concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialisation is so common, Java allows objects to initialise themselves when they are created. This **automatic initialisation is performed through the use of a constructor**.
- **A constructor initialises 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.**
- The *implicit return type of a class' constructor is the class type itself*.
- It is the constructor's job to initialise the internal state of an object so that the code creating an instance will have a fully initialised, usable object immediately.

```
class box
{
   int height;
   int width;
   int length;
   box( )
   {
      System.out.println("Constructing box");
      height=10;
      width=10;
      length=10;
   }
   int volume()
   {
      return height*width*length;
   }
}
class example
{
   public static void main(String args[])
   {
      box box1= new box();
      box box2= new box();
      System.out.println("Volume is "+box1.volume());
```

```
        System.out.println("Volume is "+box2.volume());
    }
}
```

OUTPUT:

```
Constructing box
Constructing box
Volume is 1000
Volume is 1000
```

Parameterised Constructors :

```
class box
{
    int height;
    int width;
    int length;
    box(int h, int w, int l)
    {
        System.out.println("Constructing box");
        height=h;
        width=w;
        length=l;
    }
    int volume()
    {
        return height*width*length;
    }
}
class example
{
    public static void main(String args[])
    {
        box box1= new box(10,20,30);
        box box2= new box(10,10,10);
        System.out.println("Volume is "+box1.volume());
        System.out.println("Volume is "+box2.volume());
    }
}
```

OUTPUT:

```
Constructing box
Constructing box
Volume is 6000
Volume is 1000
```

The values 10, 20, and 30 are passed to the **Box( )** constructor when **new** creates the object. Thus, box1's copy of **width**, **height**, and **depth** will contain the values 10, 20, and 30, respectively.

The values 10, 10, and 10 are passed to the **Box( )** constructor when **new** creates the object. Thus, box2's copy of **width**, **height**, and **depth** will contain the values 10, 10, and 10, respectively.

The this Keyword :

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.

```
class box
{
   int height;
   int width;
   int length;
   box(int h, int w,int l)
   {
      this.height=h;
      this.width=w;
      this.length=l;
   }
}
```

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

Instance Variable Hiding :

- It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. **However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.**
- **This** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

```
class box
{
   int height;
   int width;
   int length;
   box(int height, int width, int length)
   {
      System.out.println("Constructing box");
      this.height=height;
      this.width=width;
```

```
        this.length=length;
    }
    int volume()
    {
        return height*width*length;
    }
}
class example
{
    public static void main(String args[])
    {
        box box1= new box(10,20,30);
        box box2= new box(10,10,10);
        System.out.println("Volume is "+box1.volume());
        System.out.println("Volume is "+box2.volume());
    }
}
```

OUTPUT:

```
Constructing box
Constructing box
Volume is 6000
Volume is 1000
```

NOTE:

Garbage Collection:

Since objects are dynamically allocated by using the **new** operator, 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.