

Classes and Objects in Java

Basics of Classes in Java

Introduction

- Java is an OO language and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the “state” and “behaviour” of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them. They provide a convenient method for packaging a group of logically related data items and functions that work on them.
- A class essentially serves as a template for an object and behaves like a basic data type “int”. It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic OO concepts such as encapsulation, inheritance, and polymorphism.

Is Java not a purely Object-Oriented Language???

- There are seven qualities to be satisfied for a programming language to be pure Object Oriented. They are:
 1. Encapsulation/Data Hiding
 2. Inheritance
 3. Polymorphism
 4. Abstraction
 5. All predefined types are objects
 6. All user defined types are objects
 7. All operations performed on objects must be only through methods exposed at the objects.
- Example: Smalltalk, and Eiffel.

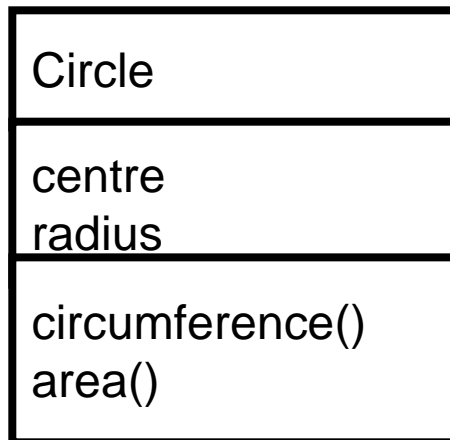
Java is not a pure OOP language due to two reasons:

- The first reason is that the Object oriented programming language **should only have objects** whereas java contains 8 primitive data types like char, boolean, byte, short, int, long, float, double which are not objects. These primitive data types can be used without the use of any object. (Eg. `int x=10; System.out.print(x.toString());`)

- The second reason related to the **static keyword**. In pure object oriented language ,we should access everything by message passing (through objects). But java contains static variables and methods which can be accessed directly without using objects. That means, when we declare a class as '**static**' then it can be referenced without the use of an object.

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.



Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

```
class ClassName [extends SuperClassName]  
{  
    [fields declaration]  
    [methods declaration]  
}
```

Adding Fields: Class Circle with fields

- Add *fields*

```
public class Circle {  
    public double x, y; // centre coordinate  
    public double r;    // radius of the circle  
  
}
```

- The fields (data) are also called the *instance* variables.

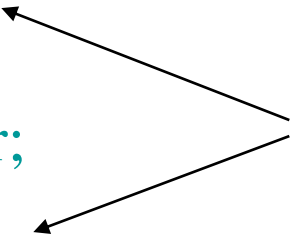
Adding Methods

- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
    Method-body;
}
```

Adding Methods to Class Circle

```
public class Circle {  
  
    public double x, y; // centre of the circle  
    public double r;    // radius of circle  
  
    //Methods to return circumference and area  
    public double circumference() {  
        return 2*3.14*r;  
    }  
    public double area() {  
        return 3.14 * r * r;  
    }  
}
```



Method Body

Class Fundamentals

A class is a *template* for an object, and an object is an *instance* of a class

The General Form of a 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  
    }  
}
```

Here is a **class** called **Box** that defines three instance variables: **width**, **height**, and **depth**.

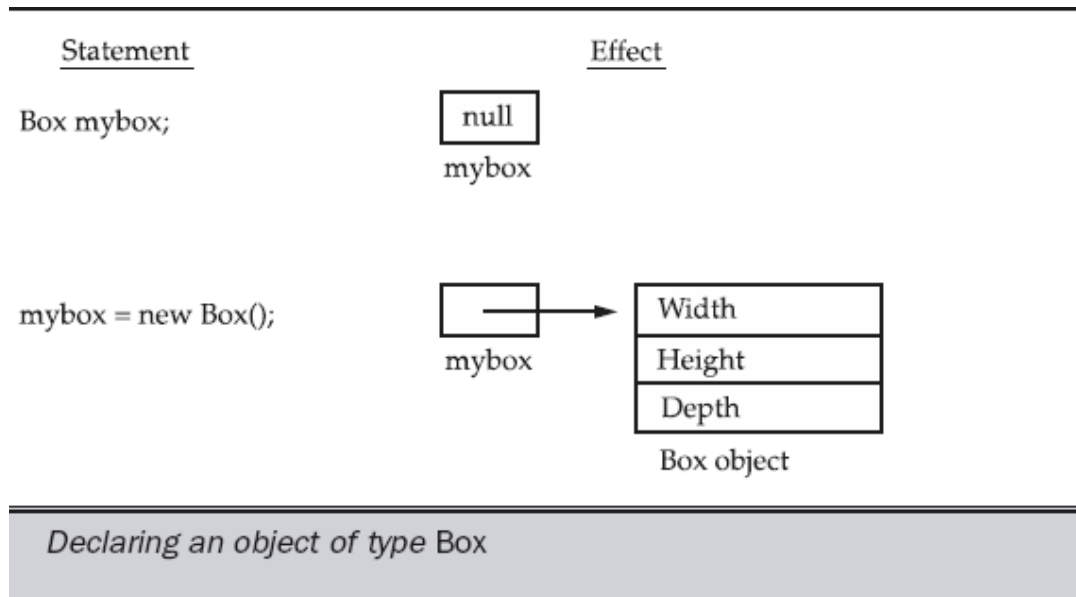
```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

to create a **Box** object called **mybox**

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

to assign the **width** variable of **mybox** the value 100

```
mybox.width = 100;  
mybox.height = 50;  
mybox.depth = 60;
```



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);  
    }  
}
```

new Operator

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

class-var = new *classname*();

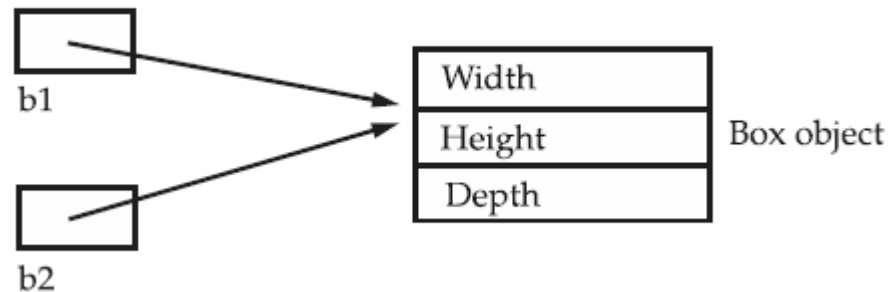
- *class-var* is a variable of the class type being created.
- *classname* is the name of the class that is being instantiated.
- The class name followed by parentheses specifies the *constructor* for the class.

Distinction between 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.

Assigning Object Reference Variables

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



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

Here, **b1** has been set to **null**, but **b2** still points to the original object.

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

```
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);  
    }  
}
```


What is a Constructor?

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Defining a Constructor

- Like any other method

```
public class ClassName {  
  
    // Data Fields...  
  
    // Constructor  
    public ClassName()  
    {  
        // Method Body Statements initialising Data Fields  
    }  
  
    //Methods to manipulate data fields  
}
```

- Invoking:
 - There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

```
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;  
    }  
}
```

Output:
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

```
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);  
    }  
}
```

Output:
Volume is 3000.0
Volume is 162.0

Garbage Collection

- 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.
- 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