

## Class

- A class is a **template** that specifies the **attributes** and **behavior** of things or objects.
- A class is a **blueprint** or **prototype** from which objects are created.
- A class is the implementation of an **abstract data type** (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively.

### Syntax:

```
class classname
{
    Datatype variable1;
    Datatype variable2;
    // ...
    Datatype variableN;

    return_type methodname1(parameter-list)
    {
        // body of method
    }

    return_type methodname2(parameter-list)
    {
        // body of method
    }

    return_type methodnameN(parameter-list)
    {
        // body of method
    }
}
```

- A class is declared by use of the **class** keyword.
- The **data**, or **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 of one object is **separate** and **unique** from the data of another.
- The actual code contained within **methods**.
- The **methods** and **variables** defined within a class are called members of the class.

## Methods

### Syntax:

```
return_type method_name(parameter-list)
{
    // body of method
}
```

- Here, **return\_type** specifies the type of data **returned** by the method. This can be any **valid** type. If the method does not **return** a value, its return type must be **void**.

- The name of the method is specified by **method\_name**. This can be any legal **identifier** other than those already used by other member 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**.

### Return value

- Method that have a **return type** other than **void** return a value to the calling **method** using the following form of the return statement:

#### Syntax:

**return** value;

- Here, **value** is the value returned.

#### Example:

```
class Box
{
    double width = 1;
    double height = 2;
    double depth = 3;

    double volume()
    {
        return (width * height * depth);
    }
}
```

### Method Call

#### Syntax:

var\_name = **object\_name**.method\_name(parameter-list);

#### Example:

vol = b1.volume();

- In above example, **b1** is an object and when **volume( )** is called, it is put on the right side of an **assignment** statement. On the left is a **variable**, in this case **vol**, that will receive the value returned by **volume( )**.

### Declaring Object

- To obtain an **object** of class, **first** you 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**, you must obtain an **actual** copy of the object and **assign** it to that variable. You 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 an **address** in memory of the object allocated by **new**.

### Syntax:

**Step - 1 :** `class_name class_var;`

**Step - 2:** `class_var = new class_name( );`

- Here, **class\_var** is a variable of the class type being created. The **class\_name** is the name of the class that is being instantiated.

### Assigning Object Reference Variables

- **Object reference** variables acts differently than you might expect when an assignment takes place.

#### Example:

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

```
Box b2 = b1;
```

- Here, **b1** and **b2** will both refer to the **same** object. The assignment of **b1** to **b2** did not allocate any memory of the original object.
- It simply makes **b2** refer to the same object as **b1** does. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same **object**.

#### Example:

```
class Box
{
    double width = 1.0;
    double height = 2.0;
    double depth = 3.0;

    void volume()
    {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo
{
    public static void main(String args[])
    {
        Box b1 ;           // declare reference to object
        b1 = new Box()      // allocate a Box object
        //Box b1 = new Box();  We can also combine above two statement.
        b1.volume();
    }
}
```

#### Output:

Volume is 6.0



## Visibility Controls of JAVA

- Java provides a number of **access modifiers** to set access levels for **classes, variables, methods** and **constructors**. The four access levels are:
  - 1) **Package/friendly (default)** -Visible to the **package**. **No modifiers are needed**.
  - 2) **Private** - Visible to the **class** only.
  - 3) **Public**- Visible to the **class** as well as **outside the class**.
  - 4) **Protected**- Visible to the **package** and all **subClasses**.

### Default Access Modifier – No Keyword

- **Default** access modifier means no need to declare an access modifier for a **class, field, method** etc.
- A variable or method declared without any access control modifier is **available to any other class in the same package**. The **default** modifier cannot be used for methods in an **interface** because the methods in an interface are by default **public**.

**Example:**

```
String str = "Hi";  
void a()  
{  
    System.out.println(str);  
}
```

### Private Access Modifier – private

- Methods, Variables and Constructors that are declared **private** can only be accessed within the **declared class itself**.
- Private access modifier is the most **restrictive access level**. **Class** and **interfaces** cannot be **private**.
- Variables that are declared private can be accessed outside the class if **public getter** methods are present in the **class**.
- Using the private modifier, an object **encapsulates** itself and **hides** data from the outside world.

**Example:**

```
class A  
{  
    private String s1 = "Hello";  
    public String getName()  
    {  
        return this.s1;  
    }  
}
```

- Here, **s1** variable of **A** class is private, so there's no way for other classes to retrieve. So, to make this variable available to the outside world, we defined public methods: **getName()**, which returns the value of **s1**.

## Public Access Modifier - public

- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members.
- When a class member is preceded by **public**, then that member may be accessed by code **outside the class**.
- A class, method, constructor, interface etc declared public can be accessed from any other class.
- Therefore, methods or blocks declared inside a public class can be accessed from any class belonging to the Java world.
- However, if the public class we are trying to access is in a different package, and then the public class still need to be **imported**. Because of class inheritance, all public methods and variables of a class are inherited by its **subClasses**.

### Example:

```
public static void main(String[] args)
{
    // ...
}
```

- The **main( )** method of an application needs to be **public**. Otherwise, it could not be called by a **Java interpreter** (such as java) to run the class.

## Protected Access Modifier – Protected

- Variables, methods and constructors which are declared **protected** in a super class can be accessed only by the **subClasses** in other package or any class within the package of the protected members' class.
- The protected access modifier cannot be applied to **class** and **interfaces**. **Methods** can be declared **protected**, however **methods** in a **interface** cannot be declared **protected**.
- Protected access gives chance to the subClass to use the helper method or variable, while prevents a non-related class from trying to use it.

### Example:

```
package p1;
class A
{
    float f1;
    protected int i1;
}
```

**// note that class B belongs to the same package as class A**

```
package p1;
class B
{
    public void getData()
    {
        // create an instance of class A
    }
}
```



```

        A a1 = new A();
        a1.f1 = 19;
        a1.i1 = 12;
    }
}

```

- In above example, class **A** and **B** are in same package **p1**. Class **A** has **i1** variable which is declared as protected. So, it can be accessed through entire package and all its subclasses. Thus, in **getData()** method of class **B** we can access variable **f1** as well as **i1**.

## this Keyword

- Sometimes a method will need to refer to the object that invoked it.
- To allow this, Java defines the **this** keyword. Keyword **this** can be used inside any method or constructor of class to refer to the current object.
- It means, **this** is always a reference to the object on which the method was invoked.
- **this** keyword can be very useful in case of Variable Hiding.
- You can use **this** anywhere a reference to an object of the current class' type is permitted.
- We cannot create two **Instance/Local** variables with same name. But it is legal to create one instance variable & one local variable or method parameter with same name.
- **Local Variable** will hide the instance variable which is called Variable Hiding.

Example:

```

class A
{
    int v = 5;
    public static void main(String args[])
    {
        A a1 = new A();
        a1.method(20);
        a1.method();
    }
    void method(int variable)
    {
        int v = 10;
        System.out.println("Value of Instance variable :" + this.v);
        System.out.println("Value of Local variable :" + v);
    }
    void method()
    {
        int v = 40;
        System.out.println("Value of Instance variable :" + this.v);
        System.out.println("Value of Local variable :" + v);
    }
}

```

Output:

```

Value of Instance variable :5
Value of Local variable :10

```

Value of Instance variable :5

Value of Local variable :40

## static Keyword

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**.
- **main( )** is declared as **static** because it must be called before any objects exist.
- When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.
- **Methods** declared as **static** have several restrictions:
  - 1) They can only call other **static** methods.
  - 2) They must only access **static** data.
  - 3) They cannot refer to **this** or **super** in any way.

### Example:

```
class staticDemo
{
    static int count=0;    //will get memory only once and retain its value

    staticDemo()
    {
        count++;
        System.out.println(count);
    }
    static
    {
        System.out.println("Static block initialized...");
    }
    static void display()
    {
        System.out.println("Static method call...");
    }
    public static void main(String args[])
    {
        staticDemo s1=new staticDemo();
        staticDemo s2=new staticDemo();
        staticDemo s3=new staticDemo();
        display();
    }
}
```

### Output:

Static block initialized...

1

2

3

Static method call...

- If you wish to call a **static** method from outside its class, you can do so using the following general form:

**classname.method( );**

- Here, **classname** is the name of the class in which the **static** method is declared. No need to call static method through object of that class.

## final Keyword

- A **variable** can be declared as **final**. You cannot change the value of final variable. It means, final variable act as **constant** and value of that variable can never be changed.
- If you declared any **method** as **final** then you cannot **override** it.
- If you declared any **class** as **final** then you cannot **inherit** it.

**Example:**

```
class finalDemo
{
    final int b = 100;
    void m1()
    {
        b = 200; /* Error generate because we cannot change the value of
                  final variable*/
    }
    public static void main(String args[])
    {
        finalDemo f1 = new finalDemo();
        f1.m1();
    }
}
```

**Output:**

Compile Time Error

## Method Overloading

- If class have multiple methods with **same name** but **different parameters** is known as **Method Overloading**.
- Method overloading is also known as **compile time (static) polymorphism**.
- The same method name will be used with **different number of parameters and parameters of different type**.
- Overloading of methods with **different return types is not allowed**.
- Compiler identifies which method should be called among all the methods have same name using the **type** and **number of arguments**.
- However, the two functions with the same name must differ in at least one of the following,



- 1) The number of parameters
- 2) The data type of parameters
- 3) The order of parameter

**Example:**

```
class overloadingDemo
{
    void sum(int a,int b)
    {
        System.out.println("Sum of (a+b) is:: "+(a+b));
    }
    void sum(int a,int b,int c)
    {
        System.out.println("Sum of (a+b+c) is:: "+(a+b+c));
    }
    void sum(double a,double b)
    {
        System.out.println("Sum of double (a+b) is:: "+(a+b));
    }
    public static void main(String args[])
    {
        overloadingDemo o1 = new overloadingDemo();
        o1.sum(10,10);      // call method1
        o1.sum(10,10,10);   // call method2
        o1.sum(10.5,10.5);  // call method3
    }
}
```

**Output:**

```
Sum of (a+b) is:: 20
Sum of (a+b+c) is:: 30
Sum of double (a+b) is:: 21.0
```

## Constructor

- Constructor is special type of method that is used to initialize the object.
- It is invoked at the time of object creation.
- **There are two rules to define constructor as given below:**
  - 1) Constructor name must be same as its class name.
  - 2) Constructor must not have return type.
- Return type of class constructor is the **class type itself**.
- **There are two type of constructor :**
  - 1) Default Constructor
  - 2) Parameterized constructor

## Default Constructor

- A constructor that has **no parameter** is known as **default constructor**.
- If we don't explicitly declare a constructor for any class, the compiler creates a default constructor for that class.

**Example:**

```
class A
{
    A()           //Default constructor
    {
        System.out.println("Default constructor called..");
    }
    public static void main(String args[])
    {
        A a = new A();
    }
}
```

**Output:**

Default constructor called..

## Parameterized Constructor

- A constructor that has parameters is known as **parameterized constructor**.
- It is used to provide different values to the distinct objects.
- It is required to pass parameters on creation of objects.
- If we define only parameterized constructors, then we cannot create an object with **default constructor**. This is because compiler will not create default constructor. You need to create default constructor explicitly.

**Example:**

```
class A
{
    int a;
    String s1;
    A(int b,String s2)    //Parameterized constructor
    {
        b = a;
        s2 = s1;
    }
    void display()
    {
        System.out.println("Value of parameterized constructor is :: "+a+" and "+b);
    }
    public static void main(String args[])
    {
        A a = new A(10,"Hello");
    }
}
```

```

        a.display();
    }
}

```

**Output:**

Value of parameterized constructor is :: 10 and Hello

## Copy Constructor

- A copy constructor is a constructor that takes only **one parameter** which is the same type as the class in which the copy constructor is defined.
- A copy constructor is used to create another **object** that is a copy of the object that it takes as a parameter. But, the newly created copy is totally **independent** of the original object.
- It is **independent** in the sense that the copy is located at different address in memory than the original.

## Overloading Constructor

- Constructor overloading in java allows to **more than one constructor** inside one Class.
- It is not much different than **method overloading**. In Constructor overloading you have multiple constructors with **different signature** with only difference that constructor doesn't have **return type**.
- These types of constructor known as **overloaded constructor**.

## Passing object as a parameter

- If you want to construct a new object, that is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter.

**Example:**

```

class Box
{
    double width;
    double height;
    double depth;
    // It takes an object of type Box. Copy constructor
    Box(Box ob)
    {
        // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    // Parameterized constructor
    Box(double w, double h, double d)
    {
        width = w;
        height = h;
        depth = d;
    }
}

```



```

    }
    // Default constructor
    Box()
    {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }
    // constructor used when cube is created
    Box(double len)
    {
        width = height = depth = len;
    }
    // compute and return volume
    double volume()
    {
        return width * height * depth;
    }
}

class DemoAllCons
{
    public static void main(String args[])
    {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        Box myclone = new Box(mybox1); // create copy of mybox1
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of cube is " + vol);
        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}

```

**Output:**

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of cube is 343.0

Volume of clone is 3000.0