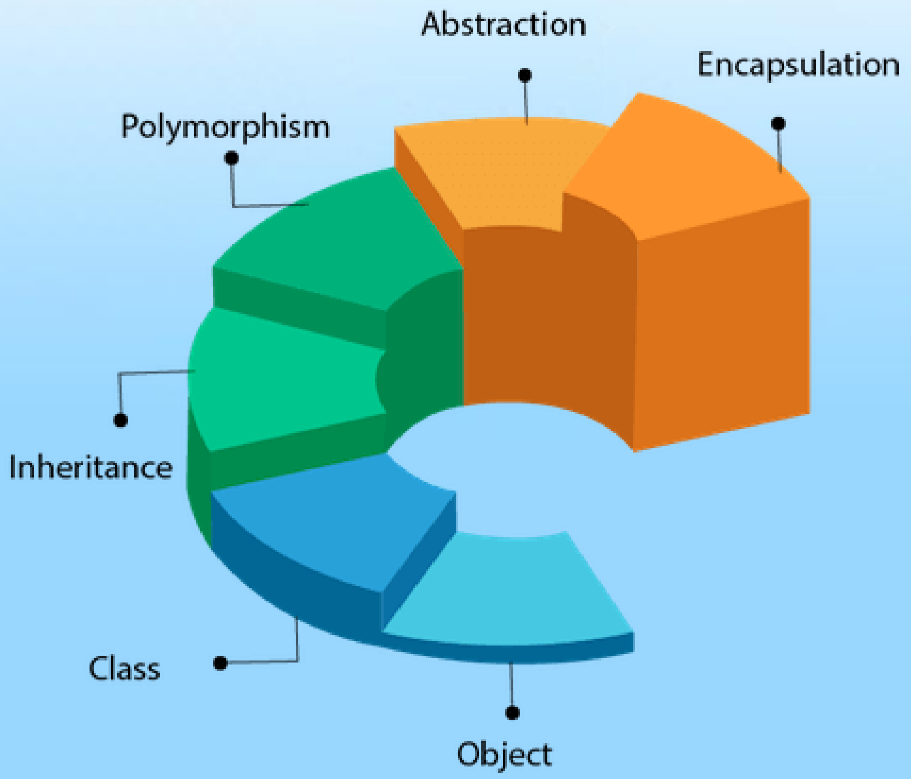
Introduction to Java

Prajul Mohandas

WHAT IS JAVA ?

- Java is Object Oriented Programming language.
- Java was developed by a team led by James Gosling at Sun Microsystems.
- Write Once - Run Anywhere
- Java is a first programming language which provide the concept of writing programs that can be executed using the web.

Object Oriented Programming



Object

- A real-world entity.
- Any entity that has state and behavior.
- It can be physical or logical.
- An Object can be defined as an instance of a class.
- **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

- Collection of objects
- It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.

Object Oriented Programming

- OOP is a methodology or paradigm to design a program using classes and objects
- A language in which everything represent in the form of Object.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Object Oriented Programming

4 Concepts in OOP

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Inheritance

- When one object acquires all the properties and behaviors of a parent object.
- It provides code reusability.
- It's creating a parent-child relationship between two classes.
- Child class is allow to inherit the features(fields and methods) of another class.

Inheritance

Syntax:

```
class derived-class extends base-class
{
   //methods and fields
}
```

Polymorphism

- refers to the ability of a variable, object or function to take on multiple forms.
- the ability to differentiate between entities with the same name efficiently.
- In Java, we use method overloading and method overriding to achieve polymorphism.

Method Overloading

- A single method may perform different functions depending on the context in which it's called.
- The class will have multiple methods having same name but different in parameters.

Advantage of Method Overloading

It increases the readability of the program

- If we have to perform only one operation, having same name of the methods increases the readability of the program.

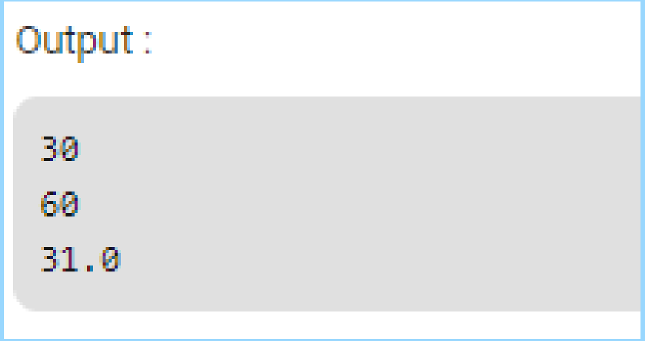
Different ways to overload the method

There are two ways to overload the method

- By changing number of arguments
- By changing the data type

Example

```
public class Sum {
    // Overloaded sum(). This sum takes two int parameters
   public int sum(int x, int y)
   {
      return (x + y);
   }
    // Overloaded sum(). This sum takes three int parameters
   public int sum(int x, int y, int z)
   {
      return (x + y + z);
   }
  // Overloaded sum(). This sum takes two double parameters
   public double sum(double x, double y)
   {
      return (x + y);
   }

   // Driver code
   public static void main(String args[])
   {
      Sum s = new Sum();
      System.out.println(s.sum(10, 20));
      System.out.println(s.sum(10, 20, 30));
      System.out.println(s.sum(10.5, 20.5));
   }
}
```



Output :

```
30
60
31.0
```

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class

Usage of Method Overloading

- used to provide the specific implementation of a method which is already provided by its superclass.
- used for runtime polymorphism

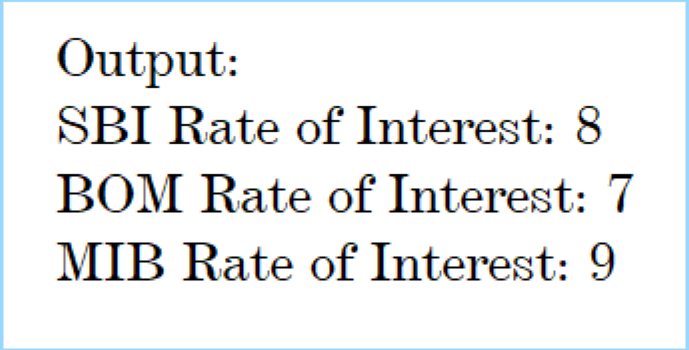
Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example

```java
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}
 class BOM extends Bank{
int getRateOfInterest(){return 7;}
}
class MIB extends Bank{
int getRateOfInterest(){return 9;}
}
```

```java
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
BOM i=new BOM();
MIB a=new MIB();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("BOM Rate of Interest: "+i.getRateOfInterest());
System.out.println("MIB Rate of Interest: "+a.getRateOfInterest());
}
}
```



```
Output:
SBI Rate of Interest: 8
BOM Rate of Interest: 7
MIB Rate of Interest: 9
```

Abstraction

- It is a process of hiding the implementation details and showing only functionality to the user.
- Another way, it shows only essential things to the user and hides the internal details.
- It lets you focus on what the object does instead of how it does it.

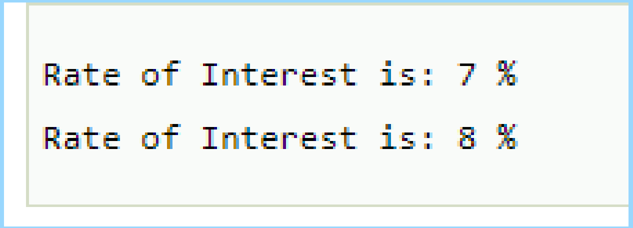
Abstraction

- For example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- There are two ways to achieve abstraction in java
  - Abstract class
  - Interface

Example

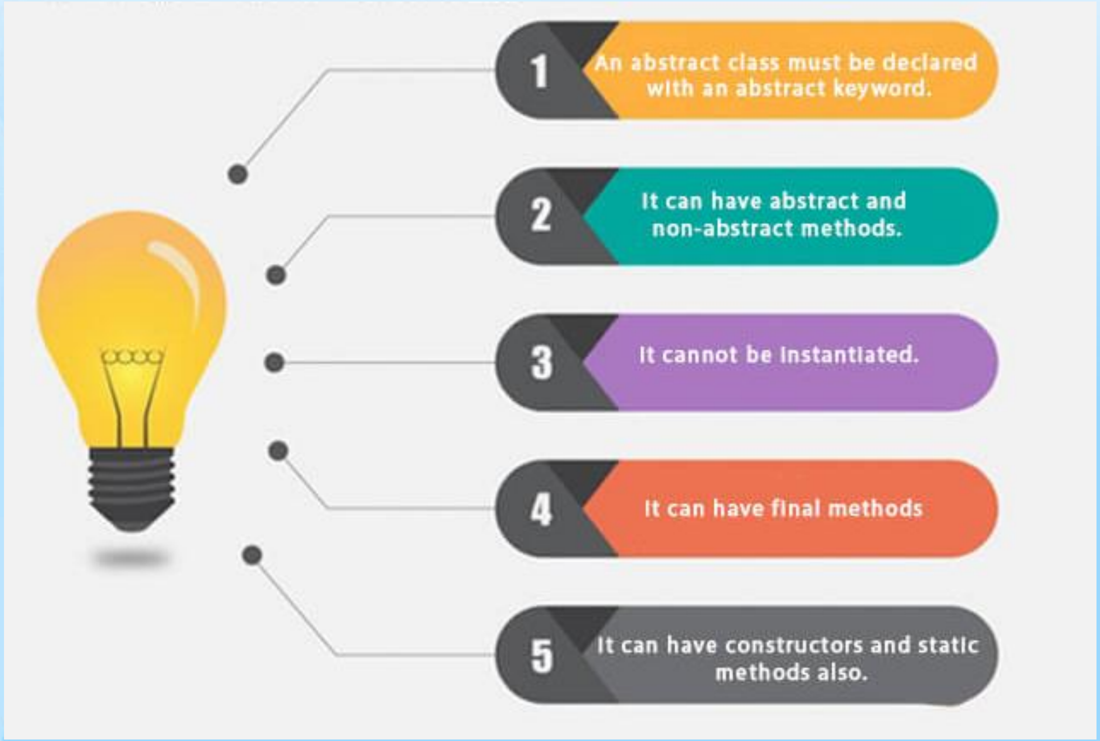
```
abstract class Bank
{
abstract int getRateOfInterest();
}
class SBI extends Bank
{
int getRateOfInterest()
{return 7;}
}
class MIB extends Bank
{
int getRateOfInterest()
{return 8;}
}
```

```
class TestBank{
public static void main(String args[])
{
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new MIB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```



```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

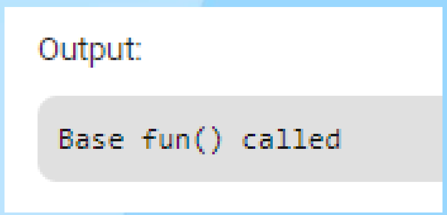
Rules for Java Abstract Class



1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

2. Can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base
{
   void fun()
  {
   System.out.println("Base fun() called");
  }
}
```



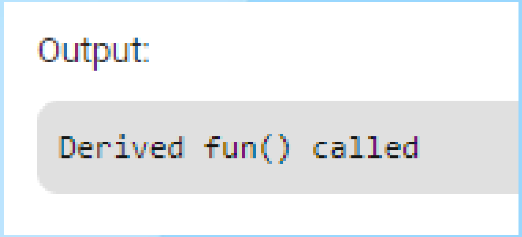
Output:

```
Base fun() called
```

```
class Derived extends Base { }

class Main
{
   public static void main(String args[])
  {
      Derived d = new Derived();
      d.fun();
  }
}
```

3. In Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base
{
   abstract void fun();
}
class Derived extends Base
{
void fun()
{System.out.println("Derived fun() called"); }
}
```



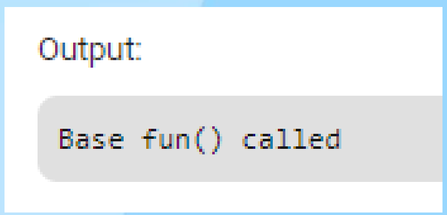
Output:

```
Derived fun() called
```

```
class Main
{
public static void main(String args[])
{
          // Uncommenting the following line will cause
// compiler error as the
     // line tries to create an instance of abstract class.
     // Base b = new Base();

     // We can have references of Base type.
     Base b = new Derived();
     b.fun();
  }
}
```

4. Can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base
 {
  final void fun()
  {
 System.out.println("Derived fun() called");
  }
}
```



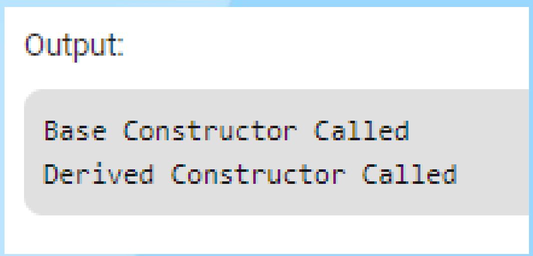
Output:

```
Base fun() called
```

```
class Derived extends Base {}

class Main
{
    public static void main(String args[])
{
    Base b = new Derived();
    b.fun();
 }
}
```

5. Can have constructor of abstract class, and is called when an instance of a inherited class is created.

```
// An abstract class with constructor
abstract class Base
{
Base()
{
 System.out.println("Base Constructor Called");
}
   abstract void fun();
}
```


Output:

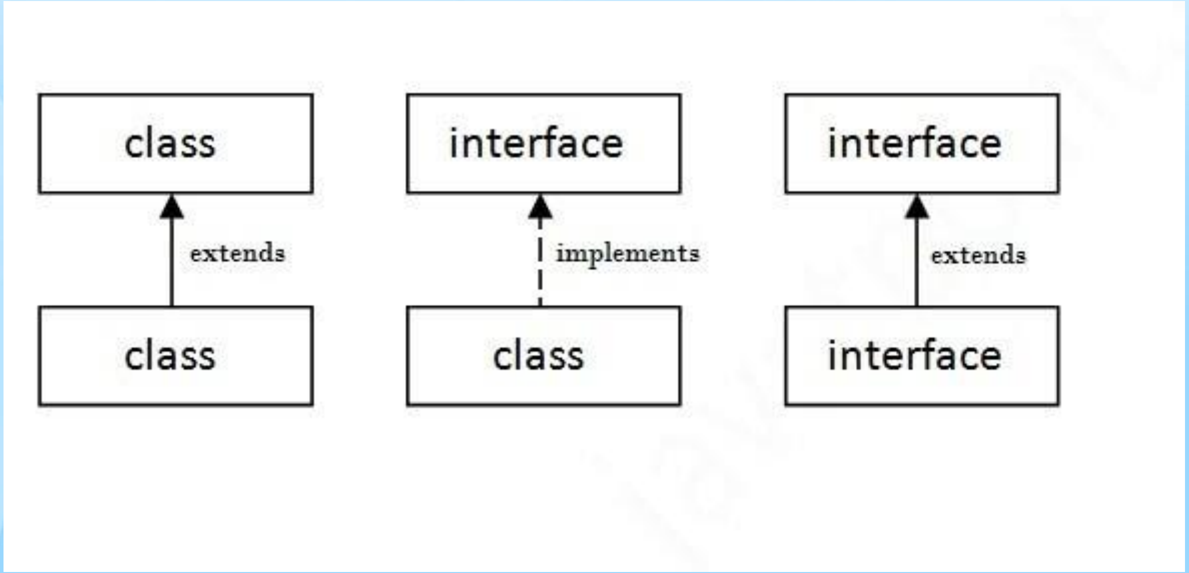
```
Base Constructor Called
Derived Constructor Called
```

```
class Derived extends Base
{
   Derived()
{ System.out.println("Derived Constructor Called"); }
   void fun()
 { System.out.println("Derived fun() called"); }
}
class Main
 {
   public static void main(String args[])
    {
      Derived d = new Derived();
    }
}
```

Interface

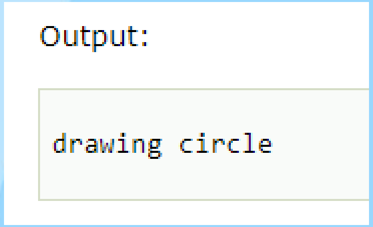
- An Interface in java is a blueprint of a class.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.

Relationship between Classes and Interfaces



Example

```
//Interface declaration: by first user
interface Drawable
{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable
{
public void draw()
{
System.out.println("drawing rectangle");
}
}
```



Output:

```
drawing circle
```

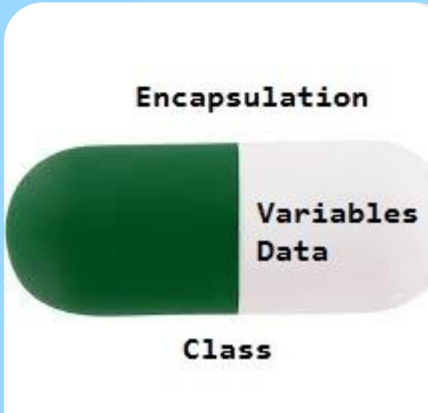
```
class Circle implements Drawable
{
        public void draw()
        {System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
        public static void main(String args[]){
        Drawable d=new Circle();
        //In real scenario, object is provided by method e.g. getDrawable()
        d.draw();
        }
}
```

Abstract Class Vs Interface

Abstract Class	Interface
An abstract class can have both abstract and non-abstract methods.	The interface can have only abstract methods.
It does not support multiple inheritances.	It supports multiple inheritances.
An abstract class can have protected and abstract public methods.	An interface can have only have public abstract methods.

Encapsulation

- It is a process of wrapping code and data together into a single unit.
- it is a protective shield that prevents the data from being accessed by the code outside this shield.



Encapsulation – How to Implement?

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Advantage of Encapsulation in Java

- It is a process of wrapping code and data together into a single unit.
- it is a protective shield that prevents the data from being accessed by the code outside this shield.

Abstraction Vs Encapsulation

Abstraction	Encapsulation
Abstraction is about hiding unwanted details while showing most essential information.	Encapsulation means binding the code and data into a single unit.
Abstraction allows focusing on what the information object must contain	Encapsulation means hiding the internal details or mechanics of how an object does something for security reasons.

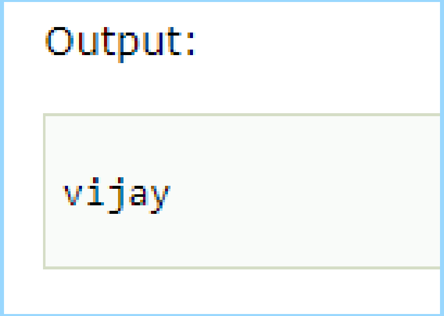
To achieve Encapsulation in Java

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example

```
public class Student{
//private data member
private String name;
//getter method for name
public String getName()
{
return name;
}
//setter method for name
public void setName(String name)
{
this.name=name
}
}
```

```
class Test{
public static void main(String[] args)
{
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```



Output:

```
vijay
```

Modifiers in Java

- Access modifiers
  - Java provides a number of access modifiers to set scope & access levels for classes, variables, methods and constructors.
- Non-access modifiers
  - Java provides a number of non-access modifiers to achieve many other functionality.

Access Modifiers

- **Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

- **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Access Modifiers

- **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

- **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Example - Private Access Modifier

```
class A
{
private int data=40;
private void msg()
{
System.out.println("Hello java");
}
}
```

```
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time
Error
   obj.msg();//Compile Time Error
   }
}
```

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

Example - Default Access Modifier

```
//save by A.java
package pack;
class A{
  void msg()
{System.out.println("Hello");}
}
```

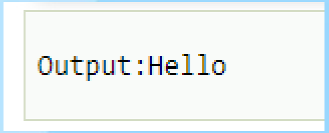
```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

Example - Protected Access Modifier

```
//save by A.java
package pack;
public class A{
protected void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```



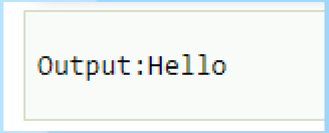
```
Output:Hello
```

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

Example - Public Access Modifier

```
//save by A.java
 package pack;
public class A{
public void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```



```
Output:Hello
```

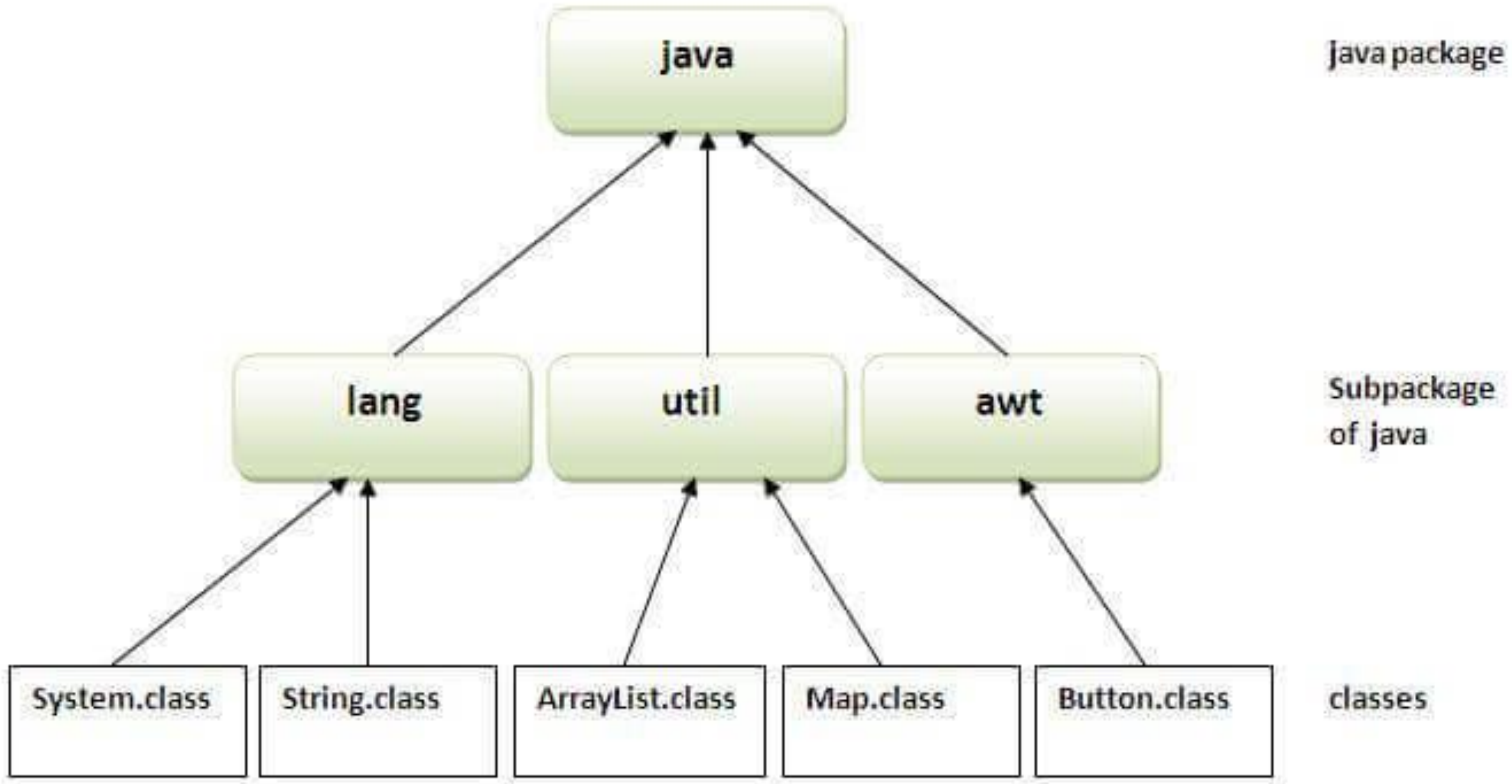
In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

Java Package

- A group of similar types of classes, interfaces and sub-packages.
  - Built-in package
  - User-defined package.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.


java
java package
lang
util
awt
Subpackage of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

Example – Java Package

```
//save as Simple.java
package mypack;
public class Simple
{
 public static void main(String args[])
   {
    System.out.println("Welcome to package");
   }
}
```

How to compile java package

If you are not using any IDE, you need to follow the syntax given below:

```
javac -d directory javafilename
```

For example

```
javac -d . Simple.java
```

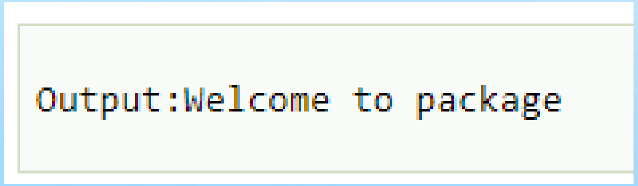
The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple



```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

- There are three ways to access the package from outside the package.
  - import package.*;
  - import package.classname;
  - fully qualified name.

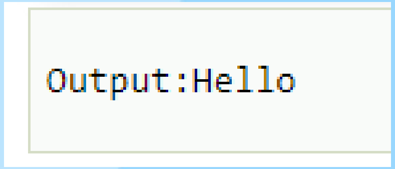
1. Using import packagename.*

- The import keyword is used to make the classes and interface of another package accessible to the current package.

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

Example – Using import packagename.*

```
//save by A.java
package pack;
public class A
{
 public void msg()
{System.out.println("Hello");}
}
```



```
Output:Hello
```

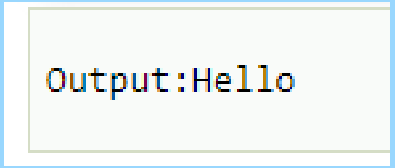
```
//save by B.java
package mypack;
import pack.*;

class B{
 public static void main(String args[])
{
  A obj = new A();
  obj.msg();
 }
}
```

2. Using packagename.classname

- If you import package.classname then only declared class of this package will be accessible.

```
//save by A.java
package pack;
public class A
{
  public void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;
class B{
  public static void main(String args[])
{     A obj = new A();
   obj.msg();    }  }
```



```
Output:Hello
```

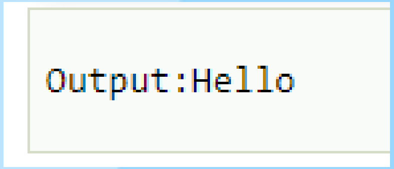
3. Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example – Using import fully qualified name

```
//save by A.java
package pack;
public class A
{
  public void msg()
{System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[])
{
   pack.A obj = new pack.A();
//using fully qualified name
   obj.msg();   }
}
```



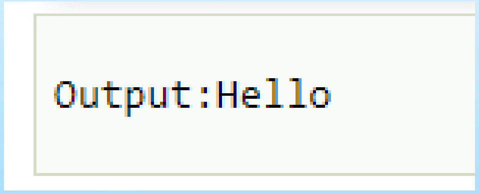
```
Output:Hello
```

Non-Access Modifiers

- For classes,
  - Final - The class cannot be inherited by other classes
  - Abstract - The class cannot be used to create objects

Example – Final Modifier

```
final class MyClass
{
 public static void main(String[] args)
 {
   System.out.println("Hello");
  }
}
```

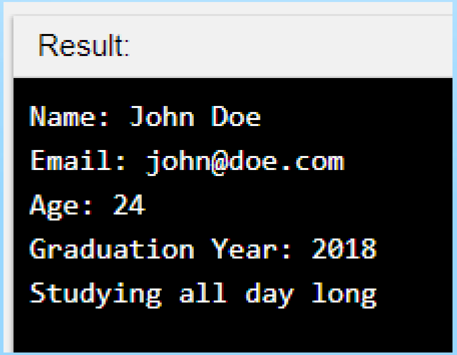


```
Output:Hello
```

The class cannot be inherited by other classes

Example – Abstract Modifier

```
class MyClass {
 public static void main(String[] args) {
  // create an object of the Student class (which inherits attributes and methods from Person)
  Student myObj = new Student();

  System.out.println("Name: " + myObj.fname + " " +myObj.lname);
  System.out.println("Email: " + myObj.email);
  System.out.println("Age: " + myObj.age);
  System.out.println("Graduation Year: " + myObj.graduationYear);
  myObj.study(); // call abstract method
 }
}
```



Result:

```
Name: John Doe
Email: john@doe.com
Age: 24
Graduation Year: 2018
Studying all day long
```

The class cannot be used to create objects

Non-Access Modifiers

- For **attributes and methods**
  - Final - Attributes and methods cannot be overridden/modified
  - Static - Attributes and methods belongs to the class, rather than an object
  - Transient - Attributes and methods are skipped when serializing the object containing them
  - Synchronized - Methods can only be accessed by one thread at a time
  - Volatile - The value of an attribute is not cached thread-locally, and is always read from the "main memory"

Constructor

- A Constructor is a special method that is used to initialize objects.
- Constructor name must match the class name.
- It is called when an object of a class is created.
- It can be used to set initial values for object attributes.

Constructor

- All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero.
- Once you define your own constructor, the default constructor is no longer used.

Rules for writing Constructor

- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructor

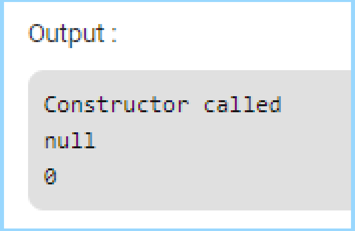
No-argument Constructor

- A constructor that has no parameter is known as default constructor.
- If we don't define a constructor in a class, then compiler creates default constructor(with no arguments) for the class.

Example

```
// Java Program to illustrate calling a
// no-argument constructor
import java.io.*;
class Geek
{
    int num;
    String name;
      // this would be invoked while an object
    // of that class is created.
    Geek()
    {
        System.out.println("Constructor called");
    }
}
```

```
class GFG
{
    public static void main (String[] args)
    {
        // this would invoke default constructor.
        Geek geek1 = new Geek();

        // Default constructor provides the default
        // values to the object like 0, null
        System.out.println(geek1.name);
        System.out.println(geek1.num);
    }
}
```



Output :

```
Constructor called
null
0
```

Types of Constructor

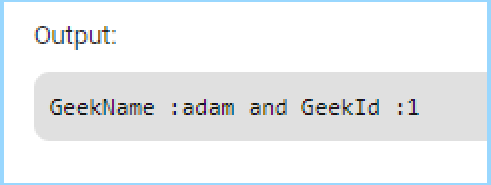
Parameterized Constructor

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with your own values, then use a parameterized constructor.

Example

```
// Java Program to illustrate calling of parameterized constructor.
import java.io.*;
class Geek
{
    // data members of the class.
    String name;
    int id;
     // constructor would initialize data members
    // with the values of passed arguments while
    // object of that class created.
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
```

```
class GFG
{
    public static void main (String[] args)
    {
        // this would invoke the parameterized constructor.
        Geek geek1 = new Geek("adam", 1);
        System.out.println("GeekName :" + geek1.name +
                           " and GeekId :" + geek1.id);
    }
}
```



Output:

```
GeekName :adam and GeekId :1
```

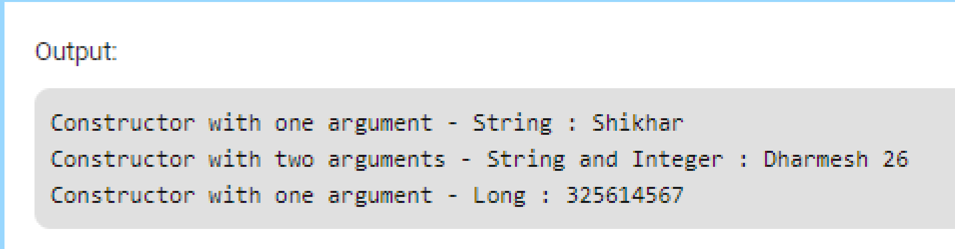
Constructor Overloading

- Like methods, we can overload constructors for creating objects in different ways.
- Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

Example

```
import java.io.*;
class Geek
{    // constructor with one argument
   Geek(String name)
   {       System.out.println("Constructor with one " +
                "argument - String : " + name);    }
    // constructor with two arguments
   Geek(String name, int age)
   {   System.out.println("Constructor with two arguments : " +
            " String and Integer : " + name + " "+ age);   }
    // Constructor with one argument but with different
   // type than previous..
   Geek(long id)
   {  System.out.println("Constructor with
        one argument : " + "Long : " + id);   } }
```

```
class GFG
{  public static void main(String[] args)
   {   // Creating the objects of the class named 'Geek'
       // by passing different arguments
       // Invoke the constructor with one argument of
       // type 'String'.
       Geek geek2 = new Geek("Shikhar");
       // Invoke the constructor with two arguments
       Geek geek3 = new Geek("Dharmesh", 26);
       // Invoke the constructor with one argument of
       // type 'Long'.
       Geek geek4 = new Geek(325614567);    } }
```



Output:

```
Constructor with one argument - String : Shikhar
Constructor with two arguments - String and Integer : Dharmesh 26
Constructor with one argument - Long : 325614567
```

How Constructors are different from Methods in Java?

Constructor	Methods
Must have the same name as the class within which it defined	It is not necessary for the method in java
Do not return any type	Have the return type or void if does not return any value
Called only once at the time of Object creation	Can be called any numbers of time.

Thank You.