

Unit:2

Inheritance, java packages

Universal class(object class)

Access Specifiers

Constructor in inheritance

Method overriding

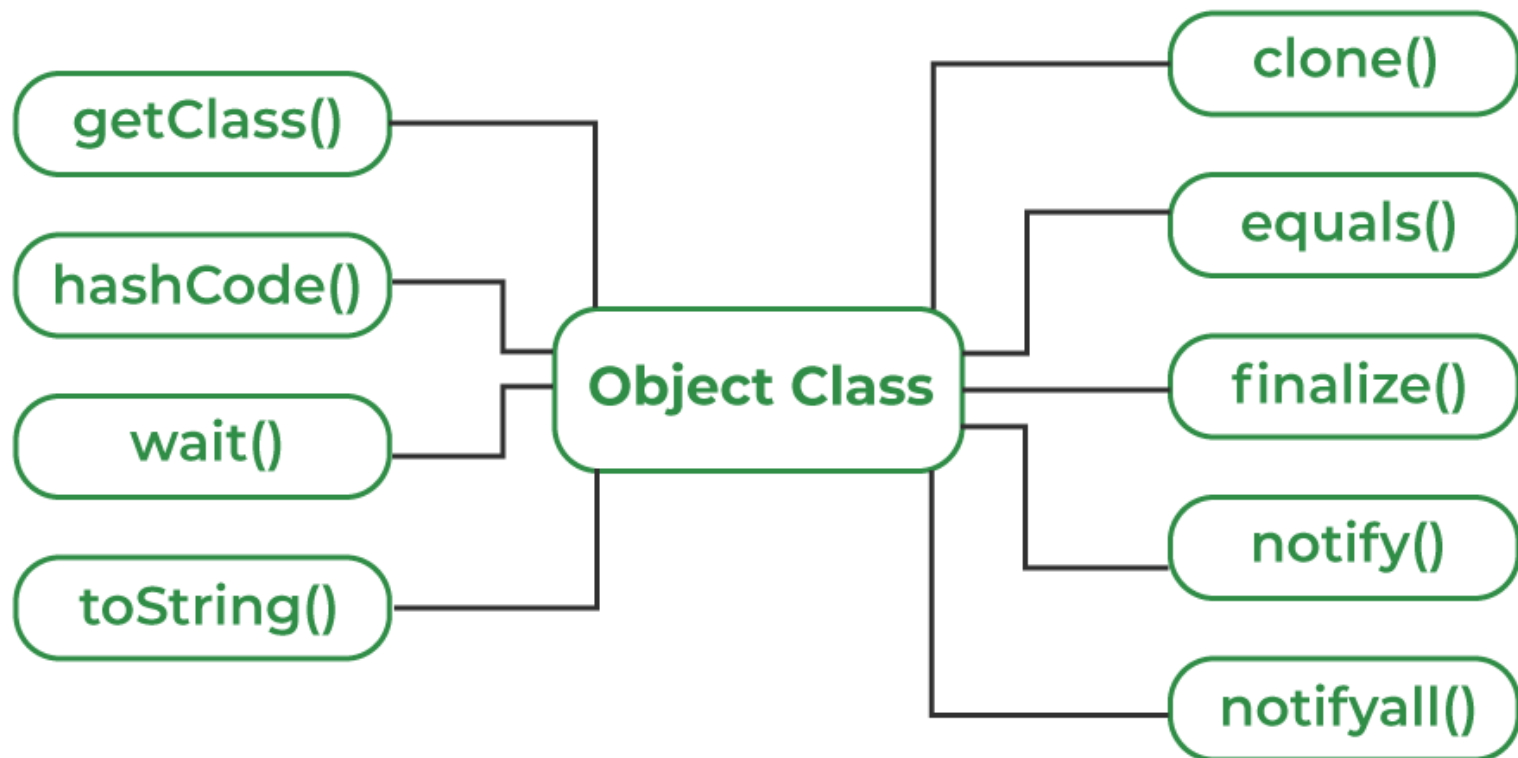
Universal Class(Object class in Java)

- Object class is present in java.lang package. The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.
- Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object.
- `Object obj = getObject();`
`// we don't know what object will be returned from this method`

The Object class provides some common behaviours to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Object class acts as a root of the inheritance hierarchy in any Java Program.

Using Object Class Methods



1. toString() method

- The toString() provides a String representation of an object and is used to convert an object to a String.
- Default behavior of toString() is to print class name, then @, then unsigned hexadecimal representation of the hash code of the object.
- ```
public String toString()
{
 return getClass().getName() + "@" +
 Integer.toHexString(hashCode());
}
```

## 2.hashCode() method

- For every object, JVM generates a unique number which is a hashcode. It returns distinct integers for distinct objects.
- A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find the address of an object, so it uses native languages like C/C++ to find the address of the object.

## **Use of hashCode() method:**

It returns a hash value that is used to search objects in a collection.

JVM(Java Virtual Machine) uses the hashCode () method while saving objects into hashing-related data structures like HashSet, HashMap, Hashtable, etc.

The main advantage of saving objects based on hash code is that searching becomes easy.

```
public int hashCode()
{
 return roll_no;
}
```

### 3. getClass() method

- It returns the class object of “this” object and is used to get the actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don't override it.

```
public class Test {
 public static void main(String[] args)
 {
 Object obj = new String("GeeksForGeeks");
 Class c = obj.getClass();
 System.out.println("Class of Object obj is : " + c.getName());
 }
}
```

## 4. equals(Object obj) method

- It compares the given object to “this” object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override the **equals(Object obj)** method to get our own equality condition on Objects.
- Syntax:

**public boolean** equals(Object obj)



## 5. finalize() Method

- In Java, garbage means unreferenced object.
- There are 3 ways to unreferenced object:
  - 1. by nulling the reference( c=null;)
  - 2.by assigning a reference to another (c1=c2;)
  - 3.by annonyms object (new crickter();)
- The purpose of finalize method is to realese the resources that is allocated by unused object, before removing unused object by garbage collector(gc()).
- Syntax:  
**protected void finalize()**

## 6.clone() method

- It returns a new object that is exactly the same as this object.
- The **object cloning** is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.
- The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.
- Syntax:

**protected** Object clone() **throws** CloneNotSupportedException

## Why use clone() method ?

- The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.
- Advantage:
- It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
- Clone() is the fastest way to copy array.

## Disadvantage

- To use the `Object.clone()` method, we have to change a lot of syntaxes to our code, like implementing a `Cloneable` interface, defining the `clone()` method and handling `CloneNotSupportedException`, and finally, calling `Object.clone()` etc.
- We have to implement `cloneable` interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform `clone()` on our object.

# Access Modifiers in Java

- There are two types of modifiers in Java:  
**access modifiers** and **non-access modifiers**.
- The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.
- There are four types of Java **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.

- **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 Modifier  | within class | within package | outside package by subclass only | outside package |
|------------------|--------------|----------------|----------------------------------|-----------------|
| <b>Private</b>   | Y            | N              | N                                | N               |
| <b>Default</b>   | Y            | Y              | N                                | N               |
| <b>Protected</b> | Y            | Y              | Y                                | N               |
| <b>Public</b>    | Y            | Y              | Y                                | Y               |

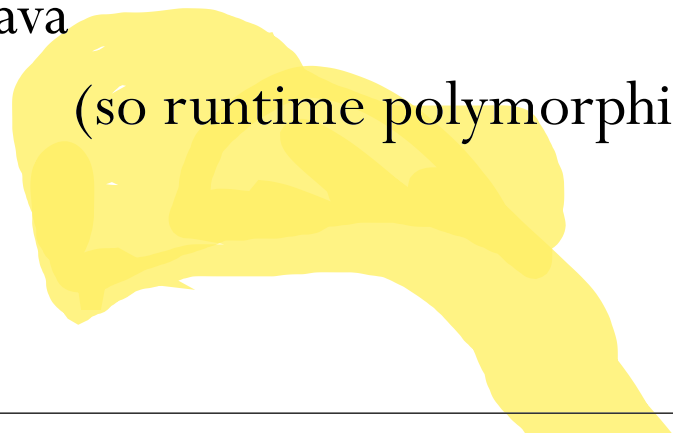
Note: A class cannot be private or protected except nested class.

## Private:

```
class A
{
 //private A() {} //private constructor
 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(); //Compile Time Error
 A obj=new A();
 System.out.println(obj.data); //Compile Time Error
 obj.msg(); //Compile Time Error
 } }
 }
```

# Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).
  - The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class.
  - Inheritance represents the **IS-A** relationship which is also known as a parent-child relationship.
  - Why use inheritance in java
  - For **Method Overriding** (so runtime polymorphism can be achieved).
  - For Code Reusability.
- 



```
class Employee
```

```
{
```

```
 float salary=40000;
```

```
}
```

```
class Programmer extends Employee
```

```
{
```

```
 int bonus=10000;
```

```
 public static void main(String args[])
```

```
{
```

```
 Programmer p=new Programmer();
```

```
 System.out.println("Programmer salary is:"+p.salary);
```

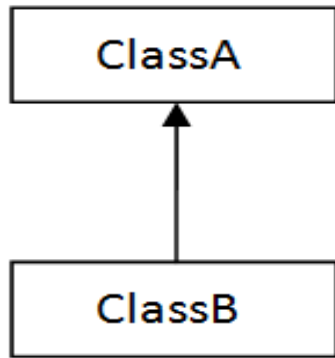
```
 System.out.println("Bonus of Programmer is:"+p.bonus);
```

```
}
```

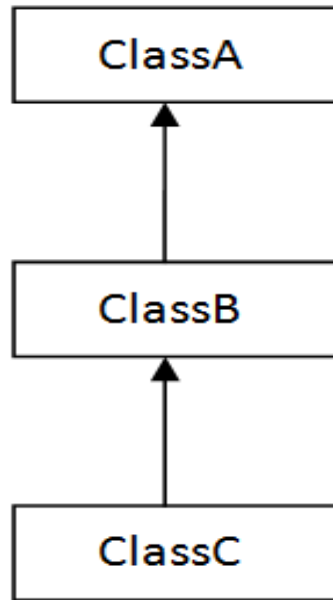
```
}
```

## Types of inheritance in java

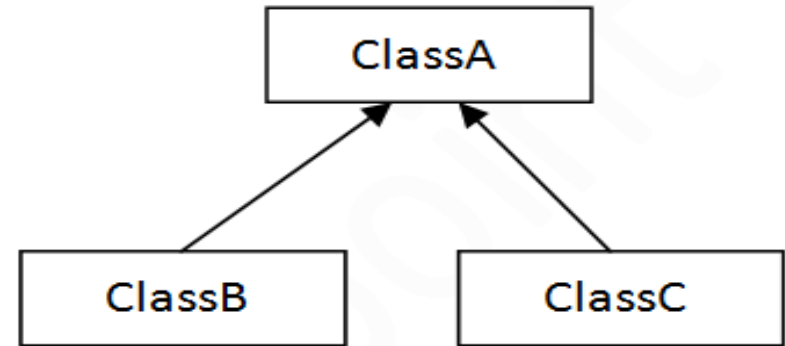
- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.



1) Single



2) Multilevel



3) Hierarchical

## Single Inheritance

```
class Animal
{
 void eat() {System.out.println("eating..."); }
}
class Dog extends Animal
{
 void bark() {System.out.println("barking..."); }
}
class TestInheritance
{
 public static void main(String args[]) {
 Dog d=new Dog();
 d.bark();
 d.eat();
 }
}
```

## Order of execution of constructor in Single inheritance

```
class ParentClass
```

```
{
 ParentClass()
 {
 System.out.println("ParentClass
 constructor executed.");
 }
}
```

```
class ChildClass extends ParentClass
```

```
{
 ChildClass()
 {
 System.out.println("ChildClass
 constructor executed.");
 }
}
```

```
public class OrderofExecution1
```

```
{
 public static void main(String
 args[])
 {
 System.out.println("Order of
 constructor execution...");

 ChildClass obj= new
 ChildClass();
 }
}
```

## Multilevel Inheritance

```
class Animal
```

```
{
 void eat()

 {
 System.out.println("eating...");
 }
}
```

```
class Dog extends Animal
```

```
{
 void bark()

 {
 System.out.println("barking...");
 }
}
```

```
class BabyDog extends Dog
```

```
{
 void weep()
```

```
{
 System.out.println("weeping...");
} }
```

```
class TestInheritance2
```

```
{
 public static void main(String args
 [])
 {
 BabyDog d=new BabyDog();
 d.weep();
 d.bark();
 d.eat();
 }
}
```

## Order of execution of constructor in Multilevel inheritance

```
class ParentClass
{
 ParentClass()
 {
 System.out.println("ParentClass
 constructor executed.");
 } }
```

```
class ChildClass extends
 ParentClass
```

```
{
 ChildClass()
 {
 System.out.println("ChildClass
 constructor executed.");
 } }
```

```
class BabyClass extends ChildClass
```

```
{
 BabyClass()
 {
 System.out.println("BabyChildCla
 ss constructor executed.");
 } }
 public class OrderofExecution1
 {
 public static void main(Stringar[])
 {
 System.out.println("Order of
 constructor execution...");
 BabyClass obj = new
 BabyClass();
 }
 }
```

## Hierarchical Inheritance

```
class Animal {
 void eat()
 {
 System.out.println("eating...");
 }
 class Dog extends Animal {
 void bark()
 {
 System.out.println("barking...");
 }
 }
 class Cat extends Animal {
 void meow()
 {
 System.out.println("meowing...")
```

```
 ;}
 }
 class TestInheritance3 {
 public static void main(String a
 rgs[])
 {
 Cat c=new Cat();
 c.meow();
 c.eat();
 //c.bark(); //C.T error
 }
 }
```

# Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- Usage:
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.
- Rules for Java Method Overriding:
- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).



**class** Vehicle

```
{
 void run() //defining a method
 {
 System.out.println("Vehicle is running");
 }
}
```

**class** Bike2 **extends** Vehicle // Creating a child class

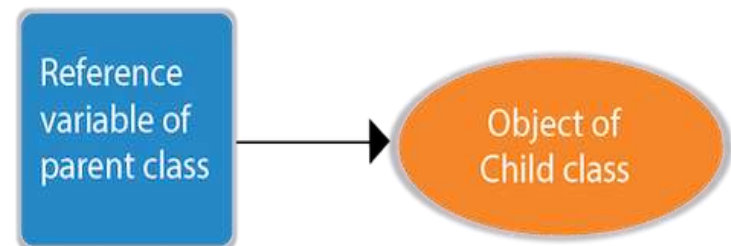
```
{
 //defining the same method as in the parent class
 void run()
 {
 System.out.println("Bike is running safely");
 }
}
```

**public static void** main(String args[])

```
{
 Bike2 obj = new Bike2(); //creating object
 obj.run(); //calling method
}
}
```

# Dynamic method dispatch

- **Runtime polymorphism or Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- **Upcasting:** If the reference variable of Parent class refers to the object of Child class, it is known as upcasting.
- `class A {}`
- `class B extends A {}`
- `A a=new B(); //upcasting`



# Abstract method and classes

- Abstract in English means- exciting in thought or as in idea without concrete existence.
- **Abstract method:** A method that is declared without implementation.

```
abstract void move (double x, double y
```

- **Abstract class:** If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
Public abstract class phoneModel
{
 Abstract void switchoff();
 // more code
}
```

## Program:

```
abstract class base //don't have object{
```

```
{
 base2()
 {
 System.out.println("i m base2
 constructor!");
 }
}
```

```
public void sayhello()
{
```

```
 System.out.println("Hello!");
}
```

```
abstract public void greet();
}
```

```
class derived extends base
```

```
{
 public void greet()
 {
```

```
 System.out.println("Good
 morning!");
 } }
```

```
class absta
```

```
{
```

```
 public static void main(String[] args)
```

```
{
 // base obj=new base();
```

```
 derived obj1=new derived();
```

```
 obj1.greet();
 } }
```

# interface

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Why use Java interface?
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

# How to declare an interface?

## Syntax:

**interface** <interface\_name>

{

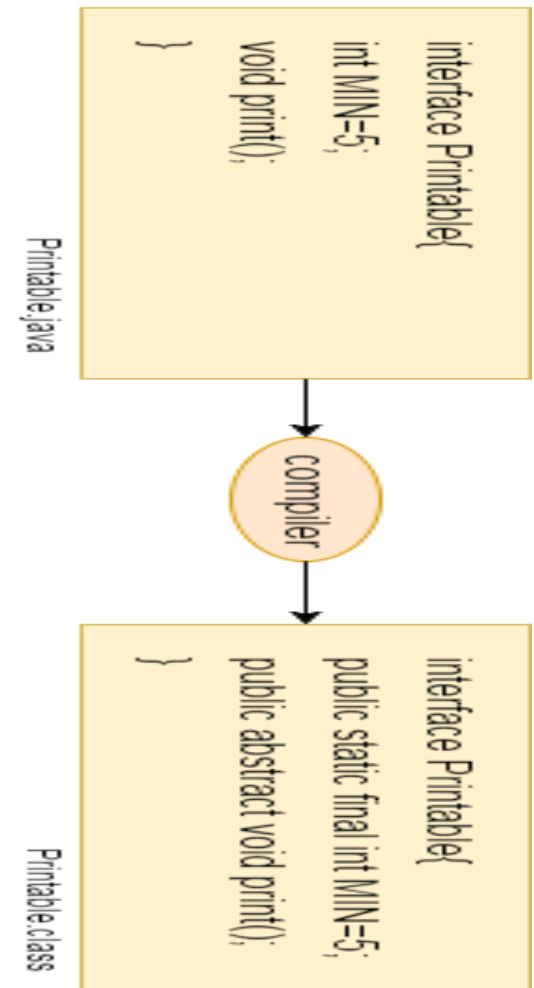
// declare constant fields

// declare methods that abstract

// by default.

}

- The Java compiler adds public and abstract keywords before the **interface method**. Moreover, it adds public, static and final keywords before **data members**.



## interface Drawable

```
//Interface declaration: by first user
{
void draw();
}
```

## class Rectangle implements Drawable

```
//Implementation: by second user
{
public void draw()
{
System.out.println("drawing
rectangle");
} }
}
```

## class Circle implements Drawable

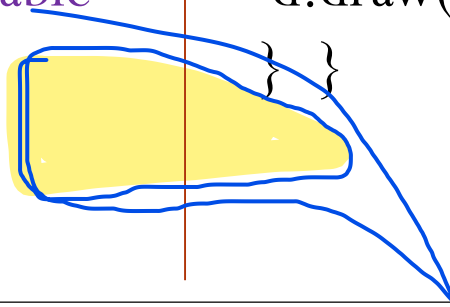
```
{
public void draw()
{

```

```
System.out.println("drawing
circle");
} }
```

## class A6

```
//Using interface: by third user
{
public static void main(String args[])
{
 //A6 d=new A6();
 //In real scenario, object is
provided by method getDrawable()
 //Drawable d=new Circle();
 Circle d= new Circle();
 d.draw();
} }
```



# Relationship Between Class and Interface

- A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.

| class                                                                       | interface                                                              |
|-----------------------------------------------------------------------------|------------------------------------------------------------------------|
| In class, you can instantiate variables and create an object                | In an interface, you can't instantiate variables and create an object. |
| A class can contain concrete (with implementation) methods.                 | The interface cannot contain concrete (with implementation) methods.   |
| The access specifiers used with classes are private, protected, and public. | In Interface only one specifier is used- Public.                       |



# Advantages of Interfaces in Java

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

## Multiple Inheritance in Java Using Interface

- Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. let us check this with an example.

# Interpreter vs compiler

- Interpreter translate one statement at a time into machine code.
- Compiler seams the entire programme and translate whole of it into machine code.

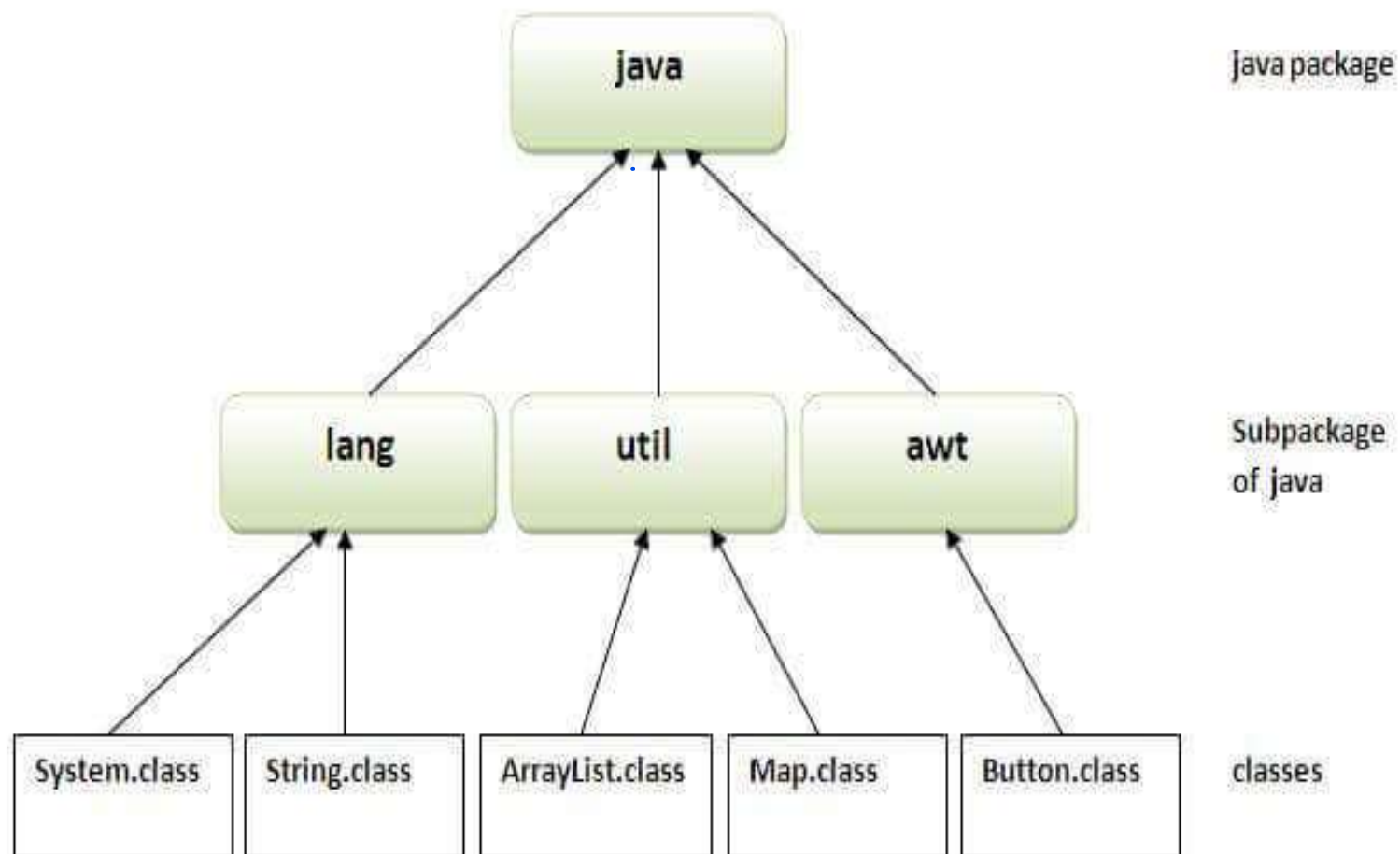
| interpreter                          | compiler                                                    |
|--------------------------------------|-------------------------------------------------------------|
| One statement at a time              | Entire programme at a time                                  |
| It is needed everytime               | Once compiled it is not needed                              |
| Partial execution if error           | No execution if an an error occur                           |
| Easy fro programmers<br>(eg. python) | Usually not as easy as interpreted<br>onces<br>(eg. c, c++) |

# Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form,
  1. built-in package (java API)
  2. user-defined package(custom packages)
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

## Advantage of Java Package

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



## Simple example of java package

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

How to compile java package

**javac -d directory javafilename**

How to run java package program

**To Compile:** javac -d . Simple.java

**To Run:** java mypack.Simple

# Built-in Packages

- **java.sql:** Provides the classes for accessing and processing data stored in a database. Classes like Connection, DriverManager, PreparedStatement, ResultSet, Statement, etc. are part of this package.
- **java.lang:** Contains classes and interfaces that are fundamental to the design of the Java programming language. Classes like String, StringBuffer, System, Math, Integer, etc. are part of this package.
- **java.util:** Contains the collections framework, some internationalization support classes, properties, random number generation classes. Classes like ArrayList, LinkedList, HashMap, Calendar, Date, Time Zone, etc. are part of this package.

- **java.net:** Provides classes for implementing networking applications. Classes like Authenticator, HTTP Cookie, Socket, URL, URLConnection, URLEncoder, URLDecoder, etc. are part of this package.
- **java.io:** Provides classes for system input/output operations. Classes like BufferedReader, BufferedWriter, File, InputStream, OutputStream, PrintStream, Serializable, etc. are part of this package.
- **java.awt:** Contains classes for creating user interfaces and for painting graphics and images. Classes like Button, Color, Event, Font, Graphics, Image, etc. are part of this package.

# How to access package from another package?

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

## 1) Using `package.*`

- If you use `package.*` then all the classes and interfaces of this package will be accessible but not subpackages.
- The `import` keyword is used to make the classes and interface of another package accessible to the current package.



## 2) Using `packagename.classname`

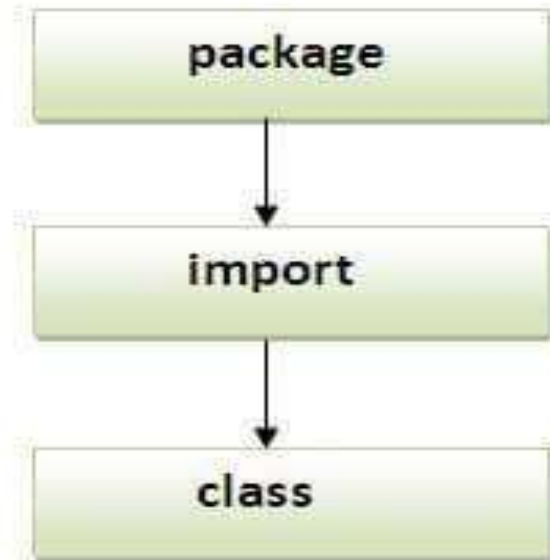
- If you import `package.classname` then only declared class of this package will be accessible.

## 3) Using fully qualified name

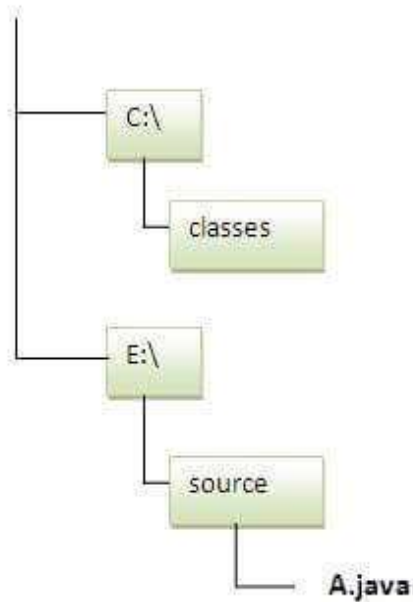
- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. `java.util`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

Note: Sequence of the program must be package then import then class.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages . Hence, you need to import the subpackage as well.



# How to send the class file to another directory or drive?



To Compile:

```
e:\sources> javac -d c:\classes Simple.java
```

To Run:

```
e:\sources> set classpath=c:\classes;.;
```

```
e:\sources> java mypack.Simple
```

Another way to run this program by -classpath switch of java:

```
e:\sources> java -classpath c:\classes mypack.Simple
```

## Subpackage in java

- Package inside the package is called the subpackage. It should be created to categorize the package further.
- Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc.
- These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

## Example of Subpackage

```
package com.javatpoint.core;
class Simple
{
 public static void main(String args[]) {
 System.out.println("Hello subpackage");
 }
}
```

- **To Compile:** javac -d . Simple.java
- **To Run:** java com.javatpoint.core.Simple
- **Output:** Hello subpackage

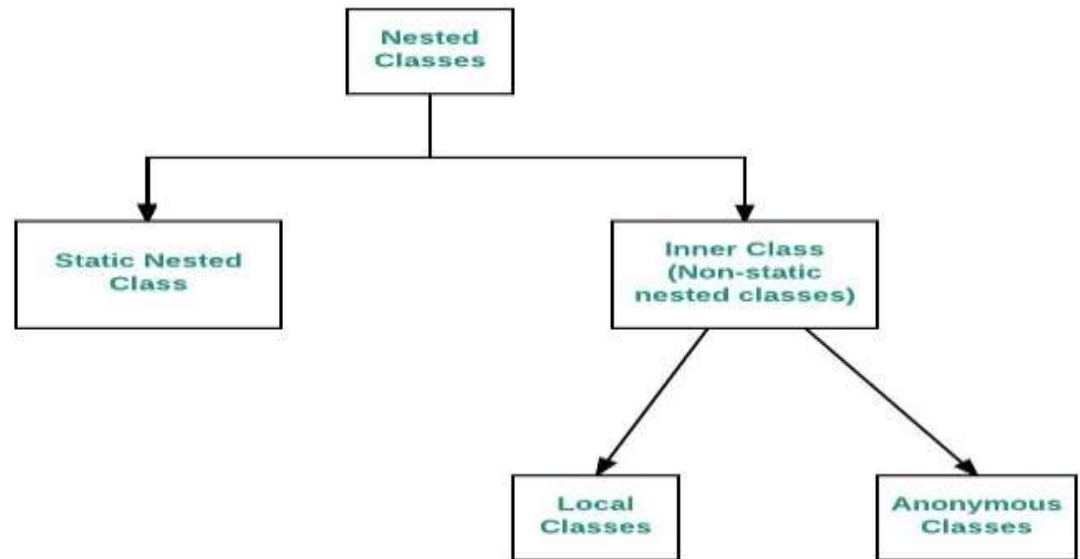
## Java Inner Classes (Nested Classes)

- Java inner class or nested class is a class that is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.
- **Advantage of Java inner classes**
  1. Nested classes represent a particular type of relationship that is it can access all the members (data members and methods) of the outer class, including private.
  2. Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
  3. Code Optimization: It requires less code to write.

## Syntax of Inner class

```
class Java_Outer_class {
 //code
 class Java_Inner_class {
 //code
 }
}
```

## Types of nested class:



| Type                         | Description                                                                                           |
|------------------------------|-------------------------------------------------------------------------------------------------------|
| <u>Member Inner Class</u>    | A class created within class and outside method.                                                      |
| <u>Anonymous Inner Class</u> | A class created for implementing an interface or extending class. The java compiler decides its name. |
| <u>Local Inner Class</u>     | A class was created within the method.                                                                |
| <u>Static Nested Class</u>   | A static class was created within the class.                                                          |
| <u>Nested Interface</u>      | An interface created within class or interface.                                                       |



# Static nested class

```
class OuterClass {
 static int outer_x = 10;
 int outer_y = 20;
 private static int outer_private =
 30;

 static class StaticNestedClass {
 void display()
 {
 System.out.println("outer_x = "
 + outer_x);
 System.out.println("outer_private
 =" + outer_private);

 OuterClass out = new
 OuterClass();
 }
 }
}
```

```
System.out.println("outer_y = "
 + out.outer_y);
 }
}
```

```
public class StaticNestedClassDemo
{
 public static void main(String[]
 args)
 {
 OuterClass.StaticNestedClass
 nestedObject = new
 OuterClass.StaticNestedClass();
 nestedObject.display();
 }
}
```

## member inner class

```
class outer
```

```
{
```

```
 int data=30;
```

```
 class Inner //member inner class
```

```
{
```

```
 void msg()
```

```
{
```

```
 System.out.println("member
inner class "+data);
```

```
}
```

```
}
```

```
public static void main(String
 args[])
```

```
{
```

```
 outer obj=new outer();
```

```
 outer.Inner in=obj.new Inner();
```

```
 in.msg();
```

```
 obj.show();
```

```
}
```

```
}
```

## local inner class

```
class outer
{
 int data=30;
 void msg() {
 class Inner //local inner class
 {
 void msg1()
 {
 System.out.println("local inner
 class "+data);
 }
 }
 }
}
```

```
 Inner in=new Inner();
 in.msg1();
 }

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