Unit: 2

Universal Class

→ **Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

→ The Object class is beneficial if we want to refer any object whose type we don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

→ 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.For example:
Object obj=getObject();

→ Constructors:

Constructor	Description
Object()	No argument constructor

→ Methods:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Access Specifiers

→ There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

❖ 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:

1) **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2) **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If we do not specify any access level, it will be the default.
3) **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If we do not make the child class, it cannot be accessed from outside the package.
4) **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
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

❖ non-access modifiers.

→ There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Inheritance

→ Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

→ The idea behind inheritance in Java is that we can create new classes that are built upon existing classes. When we inherit from an existing class, we can reuse methods and fields of the parent class. Moreover, we can add new methods and fields in our current class also.

→ Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

→ Use of Inheritance:
- For Method overriding(to achieve runtime polymorphism)
- For Code reusability

→ **Terminology:**
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates we to reuse the fields and methods of the existing class when we create a new class. We can use the same fields and methods already defined in the previous class.

→ **Syntax:**

class **Subclass-name** extends **Superclass-name**
{
//methods and fields
}

→ The extends keyword indicates that we are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

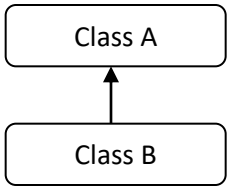
→ Types of Inheritance:

→ On the basis of class, there can be three types of inheritance in java:

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

❖ Single

- In single inheritance, subclasses inherit the features of one superclass.

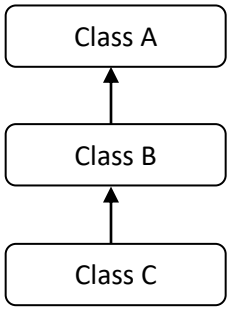


- **Example:**

```
class Animal{
        void eat(){System.out.println("Eating food...");}
}
class Cat extends Animal{
        void drink(){System.out.println("Drinking Milk...");}
}
class SingleInheritanceEx{
        public static void main(String args[]){
        Cat c=new Cat();
        c.drink();
        c.eat();
}}
```

❖ Multilevel

- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.



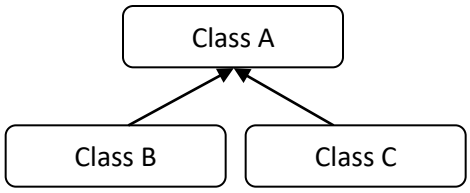
- **Example:**

```
class Animal{
        void eat(){System.out.println("Eating food...");}
}
class Dog extends Animal{
        void bark(){System.out.println("Barking...");}
}
class BabyDog extends Dog{
        void weep(){System.out.println("Weeping...");}
}
class MultilevelInheritanceEx{
```

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

❖ Hierarchical.

- In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.



- **Example:**

```
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 args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

→ To reduce the complexity and simplify the language, multiple inheritance is not supported in java through class.

→ Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and we call it from child class object, there will be ambiguity to call the method of A or B class.

→ Example:

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

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

```
    C obj=new C();
    obj.msg();//Now which msg() method would be invoked?
  }
  }
```

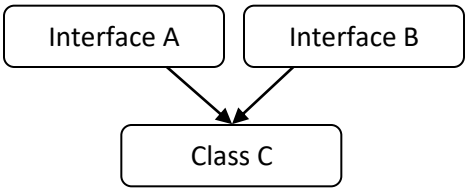
→ Since compile-time errors are better than runtime errors, Java renders compile-time error if we inherit 2 classes. So whether we have same method or different, there will be compile time error.

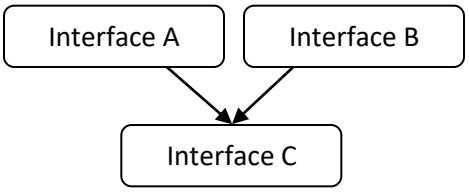
→ Multiple inheritance does not support by the java class. But can be implement using java interface.

→ On the basis of interface we can implement multiple and hybrid inheritance also other inheritance can be performed by interface.

❖ Multiple

- In Multiple inheritances, one class can have more than one interface and inherit features from all parent interfaces.



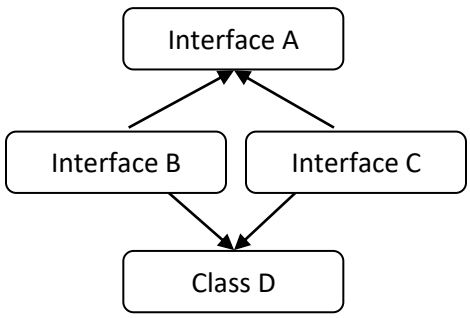


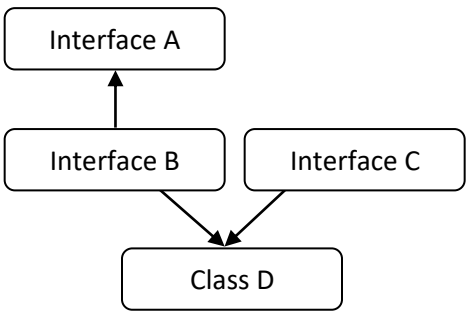
- **Example:**

```
interface Printable{
        void print();
}
interface Showable{
void show();
}
class MultipleInheritanceEx implements Printable,Showable{
public void print(){System.out.println("BCA4");}
public void show(){System.out.println("Sarvodaya");}
public static void main(String args[]){
MultipleInheritanceEx  obj = new MultipleInheritanceEx  ();
obj.print();
obj.show();
 }
}
```

❖ Hybrid

- Hybrid inheritance is a combination of hierarchical and multiple inheritance and also combination of single and multiple inheritance.
- In java, we can achieve hybrid inheritance only through Interfaces.





- **Example:**

```
interface A{
        public void dispA();
}
interface B extends A{
        public void dispB();
}
interface C extends A{
        public void dispC();
}
class HybridInheritanceEx implements B,C{
        public void dispA(){
                System.out.println("Method of Interface A");
        }
        public void dispB(){
                System.out.println("Method of Interface B");
        }
        public void dispC(){
                System.out.println("Method of Interface C");
        }
        public static void main(String[] args){
                HybridInheritanceEx obj=new HybridInheritanceEx();
                obj.dispA();
                obj.dispB();
                obj.dispC();
        }}
```

Constructor in Inheritance

→ Constructors are not members, so they are not inherited by sub classes. But the constructor of the super class can be invoked from the sub class.

→ If a class is inheriting the properties of another class, then sub class automatically acquires the default constructor of the super class.

→ But if we want to call parameterized constructor of a super class we need to use “super” keyword to invoke the parameterized constructor of the super class.

→ **Default Constructor in Inheritance:**

```
class A
{
        A(){
                System.out.println("Constructor of Class A");
        }
}
class B extends A
{
        B(){
                System.out.println("Constructor of Class B");
        }
}
class C extends B
```

```
{
    C(){
        System.out.println("Constructor of Class C");
    }
    public static void main(String[] args){
        new C();
    }
}
```

→ **Parameterized Constructor in Inheritance:**

```
class Base{
    int no;
    Base(int no){
        this.no=no;
    }
    void get(){
        System.out.println("Value of No :"+no);
    }
}
class Derived extends Base
{
    Derived(int val){
        super(val);
    }
    public static void main(String[] args){
        new Derived(25).get();
    }
}
```

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, it is known as method overriding.

→ Usage of Java Method Overriding

- 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).

→ **Example:**

```
class Vehicle{
    void run(){
        System.out.println("Vehical is running");
    }
}
class Bike extends Vehicle{
```

```
    void run(){
        System.out.println("Bike is running");

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

Super Keyword

→ The super keyword in Java is a reference variable which is used to refer immediate parent class object.

→ Whenever we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

→ Usage of super Keyword

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

❖ super is used to refer immediate parent class instance variable.

→ We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

→ **Example:**

```
class Animal{
    String color="white";
}
class Dog extends Animal{
    String color="black";
    void printColor(){
        System.out.println(color);//prints color of Dog class
        System.out.println(super.color);//prints color of Animal class
    }
}
class SuperInstanceEx{
    public static void main(String args[]){
    Dog d=new Dog();
    d.printColor();
}}
```

❖ super can be used to invoke parent class method

→ The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

→ **Example:**

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
```

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

❖ super is used to invoke parent class constructor

→ The super keyword can also be used to invoke the parent class constructor.

→ **Example:**

```
class Animal{
        Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
        Dog(){
            super();
            System.out.println("dog is created");
        }
}
class SuperConsEx{
        public static void main(String args[]){
        Dog d=new Dog();
}}
```

Interface

→ An interface in Java is a blueprint of a class. It has static constants and abstract methods.

→ The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

→ In other words, we 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.

→ Since Java 8, we can have default and static methods in an interface.

→ Since Java 9, we can have private methods in an interface.

→ There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

→ Syntax :

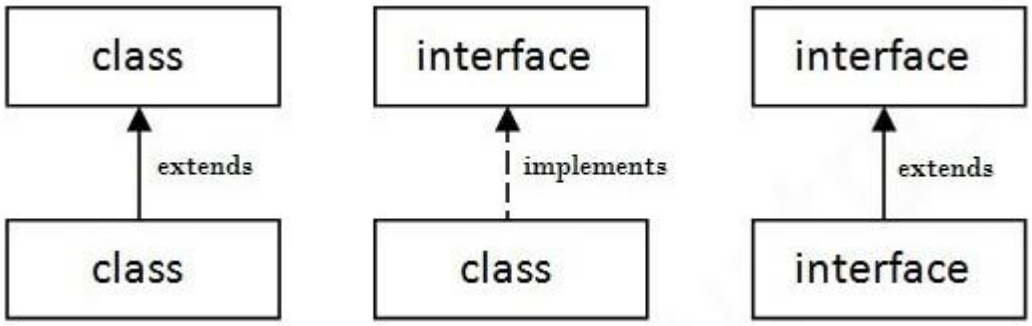
```
interface <interface_name> {

  // declare constant fields
```

```
    // declare methods that abstract
    // by default.
}
```

→ To declare an interface, use interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use implements keyword.

→ **The relationship between classes and interfaces**

- A class extends another class, an interface extends another interface, but a class implements an interface.



→ **Example:**

```
interface printable{
void print();
}
class Example implements printable{
public void print(){System.out.println("Hello");}

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

Abstract Class

→ A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

→ Abstraction is a process of hiding the implementation details and showing only functionality to the user.

→ A class which is declared as abstract is known as an abstract class. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

→ Rules for Java Abstract class:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

→ **Syntax**

```
abstract class <class_name>{}
```

→ **Example**

```
abstract class Test{}
```

→ **Abstract Method**

A method which is declared as abstract and does not have implementation is known as an abstract method.

- **Syntax**

abstract return_type method_name(parameter list);

- **Example**

abstract void printStatus();//no method body and abstract

→ **Example:**

```
abstract class Base {
   abstract void fun();
}
class Derived extends Base {
   void fun()
   {
      System.out.println("Derived fun() called");
   }
}
class AbstractEx {
   public static void main(String args[])
   {
      Base b = new Derived();
      b.fun();
   }
}
```

Difference between abstract class and interface

→ Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

→ But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.
Example: public abstract class Shape{	**Example:** public interface Drawable{

public abstract void draw();}	void draw();}

→ **Example:**

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Nested and Inner Class

→ In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable us to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.

→ The scope of a nested class is bounded by the scope of its enclosing class.

→ A nested class has access to the members, including private members, of the class in which it is nested. However, the reverse is not true i.e., the enclosing class does not have access to the members of the nested class.

→ A nested class is also a member of its enclosing class.

→ As a member of its enclosing class, a nested class can be declared private, public, protected, or package private(default).

→ Nested classes are divided into two categories:

- static nested class : Nested classes that are declared static are called static nested classes.
- inner class : An inner class is a non-static nested class.
  - Member inner class
  - Anonymous inner class
  - Local inner class

1. static nested class

→ A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

→ It can access static data members of outer class including private.

→ Static nested class cannot access non-static (instance) data member or method.

→ Java static nested class example with instance method

```
class TestOuter1{
static int data=30;
 static class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
}
```

→ Java static nested class example with static method

```
class TestOuter2{
 static int data=30;
 static class Inner{
  static void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter2.Inner.msg();//no need to create the instance of static nested class
 }
}
```

2. Inner Class

1) Java Member inner class

→ A non-static class that is created inside a class but outside a method is called member inner class.

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

2) Java Anonymous inner class

→ A class that have no name is known as anonymous inner class in java. It should be used if we have to override method of class or interface. Java Anonymous inner class can be created by two ways:

- Class (may be abstract or concrete).

- Interface

→ Java anonymous inner class example using class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

→ Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

3) Java Local inner class

→ A class i.e. created inside a method is called local inner class in java. If we want to invoke the methods of local inner class, we must instantiate this class inside the method.

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }}
```

Static – Normal(Non-static) import

❖ Static import

→ static import feature allows to access the static members of a class without the class qualification. static import provides accessibility to static members of the class.

→ we can access all the static members (variables and methods) of a class directly without explicitly calling class name.

→ **Example:**

```
import static java.lang.System.*;
import static java.lang.Math.*;
class StaticImportExample{
  public static void main(String args[]){

    out.println("Welcome to BCA SEM 4");//Now no need of System
    out.println("Absolute value of -56 is "+abs(-56));//no need of Math

  }
}
```

❖ Normal import

→ The import allows the java programmer to access classes of a package without package qualification whereas the The import provides accessibility to classes and interface.

→ With the help of import, we are able to access classes and interfaces which are present in any package.

→ **Example:**

```
import java.util.*;
public class Main
{
   public static void main(String[] args)
   {
      Date d1 = new Date();
      System.out.println("Current date is " + d1);
   }
}
```

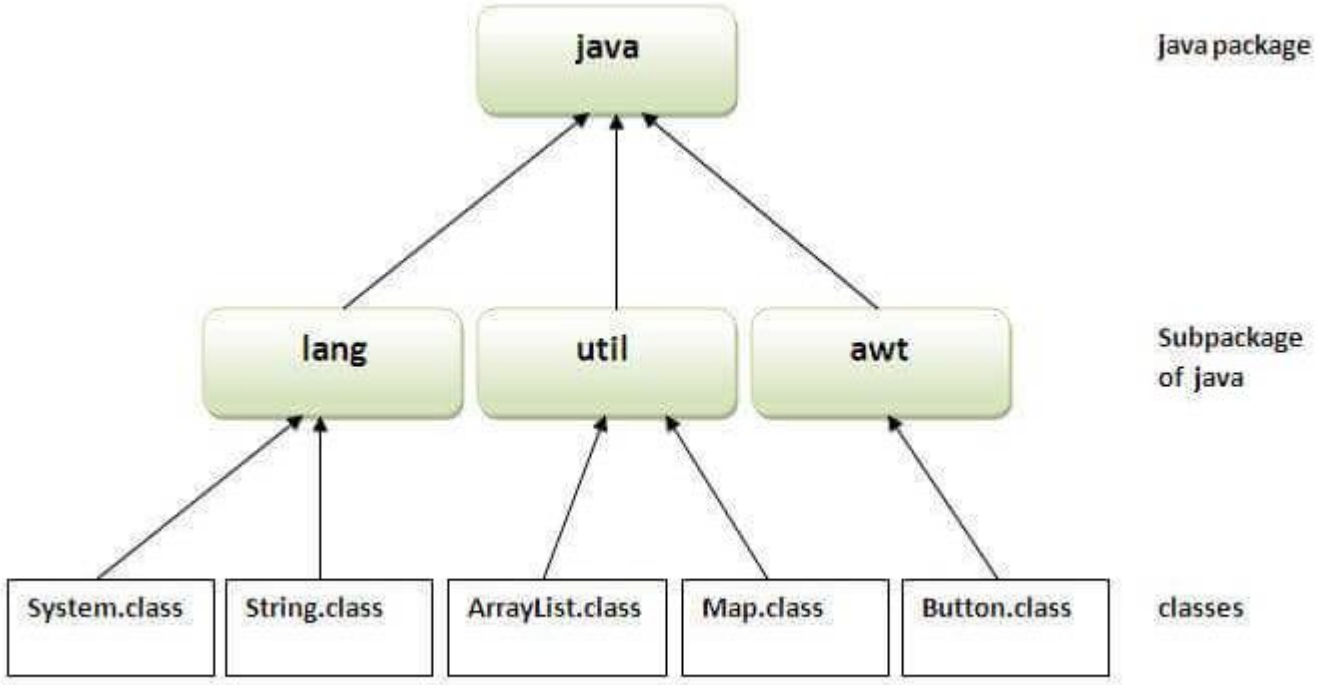
Java API packages

→ Package **in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:**

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

→ Package in java can be categorized in two form

- built-in package
- user-defined package.



1. Built in package:

→ There are many built-in packages available in java such as lang,util,io,awt,net,sql,applet etc.

→ To use a class or a package from the library, we need to use the import keyword:

→ **Syntax:**

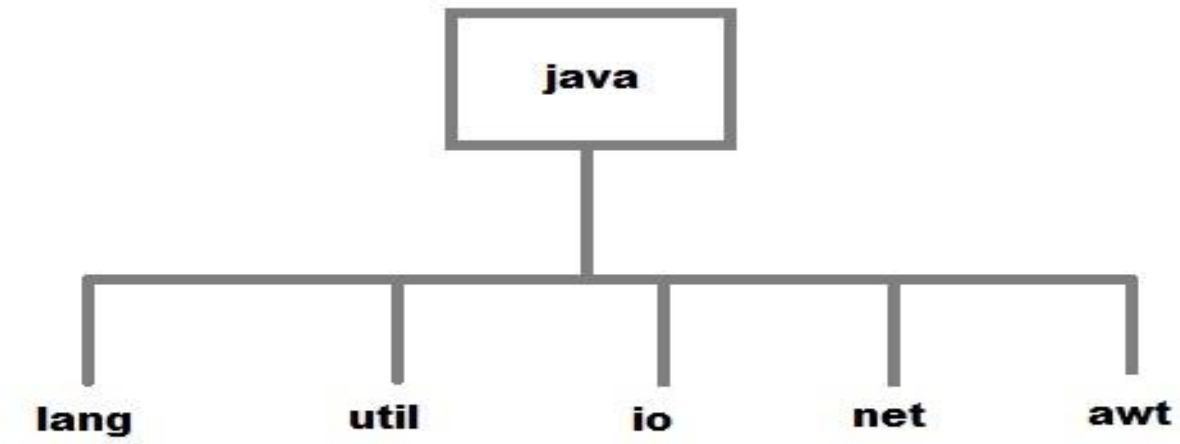
import *package.name.Class*; // Import a single class

import *package.name*.*; // Import the whole package

→ **Example:**

import java.util.Date;

import java.sql.*;



Package Name	Detail
java.lang.*	This package is used for achieving the language functionality such as conversation of data from string to fundamental data, displaying the result on the console, obtaining the garbage collector etc. This package is by default implemented for each and every java program.
java.util.*	This package is also known as "Collection Framework",it is used for developing quality or reliable application in java. It contains various classes and interfaces like Date, Vector, Stack, Queue, HashTable etc.
java.io.*	This package is used to perform Input and Output operations. It is used for developing file handling application such as opening file, read and write etc.
java.net.*	This package is used for developing networking application like client-server application.
java.awt.*	This package is used for developing GUI components such as Button, TextField, Checkbox, Scrollbar etc.
java.sql.*	This package is used for retrieving data from database and performing various operations on the database.
java.applet.*	This package is used for developing browser oriented application. An applet is a java program which runs in the context of www on browser.

2. User defined package:

→ To create our own package, we need to understand that Java uses a file system directory to store them. Just like folders on our computer.

→ To create a package, use the "package" keyword. It must be first statement in source code.

→ **Syntax:**

package packagename;

→ **Example:**

package mypack;

→ The package statement define a namespace in which classes are stored.

→ If we omit package statement, the classes are put into the default package which has no name.

→ We can create a hierarchy of package. To do that separate each package name with "." Sign.

→ **Syntax:**

package packagename.subpackagename;

→ **Example:**

package mypack.test;

→ **Compilation of Java Package:**

javac –d directory javafilename

Here, -d switch specifies the destination where to put the generated class file. We can use any directory name like /home(in case of Linux),d:/javaex(in case of windows) etc. If we want to keep the package within the same directory we can use "." Sign.

Example:

To compile : javac –d . FirstPackage.java

To run: java mypack.FirstPackage

→ **Execution of Java Package:**

There are 3 different ways to import package in Java file.

1) import packagename.*;
2) import packagename.classname;
3) fully qualified name

1) Using import packagename.*

→ If we 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.

→ **Example:**

```
//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();
  }
}
```

→ **Execution:**

```
javac –d . A.java
javac –d . B.java
java mypack.B
```

2) Using import packagename.classname;

→ If we import package.classname then only declared class of this package will be accessible.

→ **Example:**

```
//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();
  }
}
```

→ **Execution:**

```
javac –d . A.java
javac –d . B.java
java mypack.B
```

3) Using fully qualified name

→ If we use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But we need to use fully qualified name every time when we 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:**

```
//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();
   obj.msg();
  }
}
```

→ **Execution:**

```
javac –d . A.java
javac –d . B.java
java mypack.B
```

❖ Subpackage

→ Package inside the package is called the subpackage. It should be created to categorize the package further.

→ For example, Sun Microsystem has definded 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:**

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

→ **Execution:**

```
javac –d . A.java
```

```
javac –d . B.java
java mypack.B
```

Object Cloning

→ 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.

→ The clone() method is defined in the Object class. Syntax of the clone() method is as follows:

```
protected Object clone() throws CloneNotSupportedException
```

→ 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 of Object cloning:

- We don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
- 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 of Object cloning

- 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.
- Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
- Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
- If we want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
- Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

→ **Example of clone() method (Object cloning)**

```
class Student implements Cloneable{
int rollno;
String name;

Student(int rollno,String name){
this.rollno=rollno;
this.name=name;
}

public Object clone()throws CloneNotSupportedException{
return super.clone();
}

public static void main(String args[]){
try{
Student s1=new Student(101,"Ronak");
```

```
Student s2=(Student)s1.clone();

System.out.println(s1.rollno+" "+s1.name);
System.out.println(s2.rollno+" "+s2.name);

}catch(CloneNotSupportedException c){}

}
}
```

Copy Constructor

→ Like C++, Java also supports the **copy constructor**. But in C++ it is created by default. While in Java we define copy constructor our own.

→ In Java, a copy constructor is a special type of constructor that creates an object using another object of the same Java class. It returns a duplicate copy of an existing object of the class.

→ We can assign a value to the final field but the same cannot be done while using the clone() method. It is used if we want to create a deep copy of an existing object. It is easier to implement in comparison to the clone() method.

→ Use of Copy Constructor

- Create a copy of an object that has multiple fields.
- Generate a deep copy of the heavy objects.
- Avoid the use of the Object.clone() method.

→ Advantages of Copy Constructor

- If a field declared as final, the copy constructor can change it.
- There is no need for typecasting.
- Its use is easier if an object has several fields.
- Addition of field to the class is easy because of it. We need to change only in the copy constructor.

→ Example:

```
public class Fruit
{
        private double fprice;
        private String fname;
        //constructor to initialize roll number and name of the student
        Fruit(double fPrice, String fName)
        {
                fprice = fPrice;
                fname = fName;
        }
        //creating a copy constructor
        Fruit(Fruit fruit)
        {
                System.out.println("\nAfter invoking the Copy Constructor:\n");
                fprice = fruit.fprice;
                fname = fruit.fname;
        }
        //creating a method that returns the price of the fruit
        double showPrice()
```

```
        {
                return fprice;
        }
        //creating a method that returns the name of the fruit
        String showName()
        {
                return fname;
        }
        //class to create student object and print roll number and name of the student
        public static void main(String args[])
        {
                Fruit f1 = new Fruit(399, "Ruby Roman Grapes");
                System.out.println("Name of the first fruit: "+ f1.showName());
                System.out.println("Price of the first fruit: "+ f1.showPrice());
                //passing the parameters to the copy constructor
                Fruit f2 = new Fruit(f1);
                System.out.println("Name of the second fruit: "+ f2.showName());
                System.out.println("Price of the second fruit: "+ f2.showPrice());
        }
}
```