STUDY MATIRIAL OF JAVA

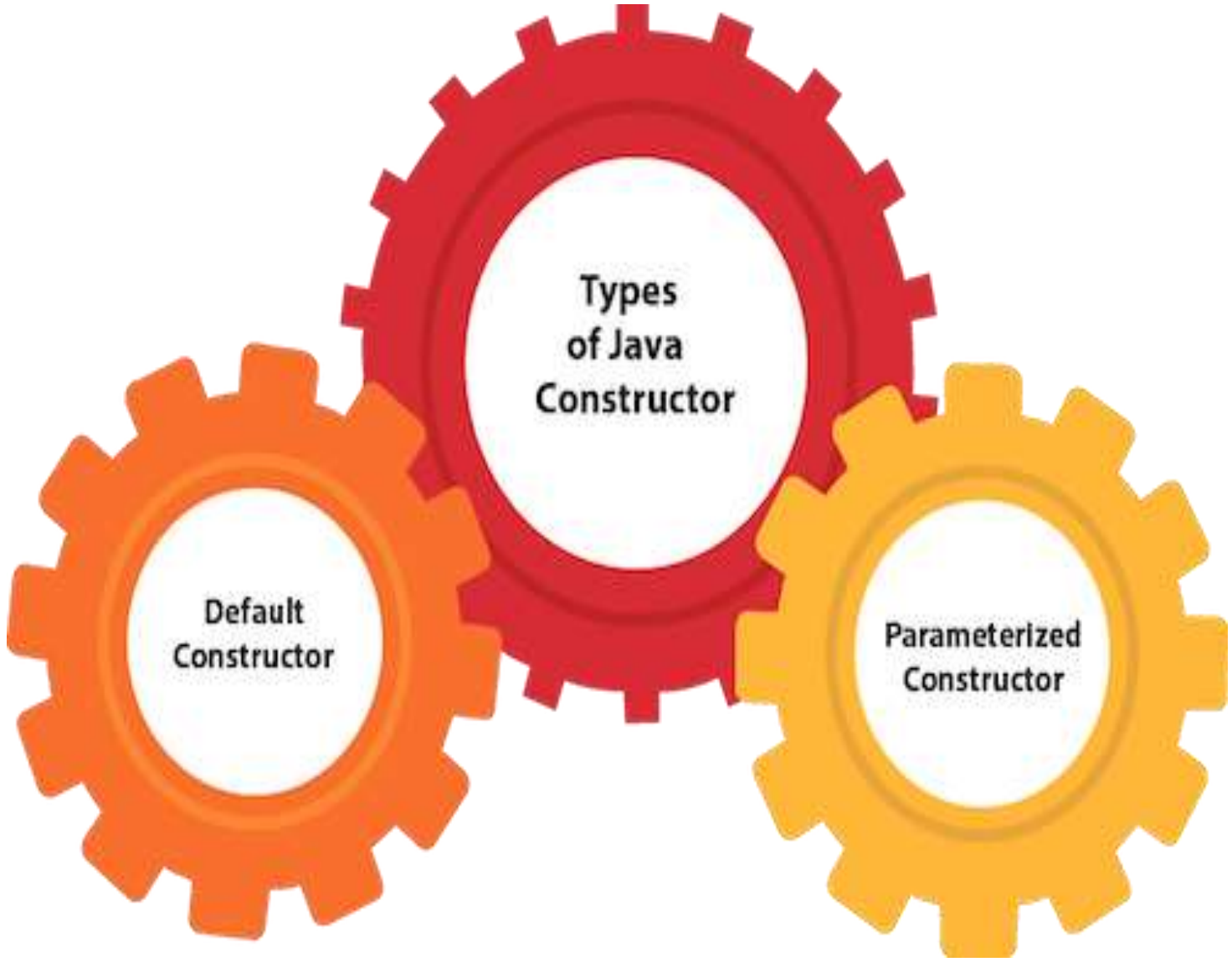
Constructor

- It is special type of method that is used to initialize the object.
- Java constructor is invoked at the time of object creation. It constructs the value.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

- There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized


Types
of Java
Constructor
Default
Constructor
Parameterized
Constructor

- **Java Default Constructor**
  A constructor is called "Default Constructor" when it doesn't have any parameter.

  Syntax of default constructor:
  <class_name>(){}

  What is the purpose of a default constructor?
  The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example:

- **//Java Program to create and call a default constructor**

  class Bike1{
- **//creating a default constructor**

  Bike1(){System.out.println("Bike is created");}
- **//main method**

  public static void main(String args[]){
- **//calling a default constructor**

  Bike1 b=new Bike1(); } }

- **Java Parameterized Constructor**
  A constructor which has a specific number of parameters is called a parameterized constructor.

  Why use the parameterized constructor?
  The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

```
class Student4{
    int id;
    String name;
 //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
 //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
    }
}
```

Difference between constructor and method in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

No-arg Constructor in Java

- Constructor without arguments is called no-arg constructor. Default constructor in java is always a no-arg constructor.
- class GfG
- {
- public GfG()
- {
- //No-arg constructor
- }
- }r.

- **How a no – argument constructor is different from default Constructor?**
- If a class contains no constructor declarations, then a default constructor with no formal parameters and no throws clause is implicitly declared.
- If the class being declared is the primordial class Object, then the default constructor has an empty body. Otherwise, the default constructor simply invokes the superclass constructor with no arguments.

- **What are private constructors and where are they used?**
- Like any method we can provide access specifier to the constructor. If it’s made private, then it can only be accessed inside the class.
- The major scenarios where we use private constructor:
- Internal Constructor chaining
- Singleton class design pattern

Constructor Overloading in Java

- **What is constructor?**

  - A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created.

- **What do you mean by overloading?**

  - It is a concept of Java in which we can create multiple methods of the same name in the same class, and all methods work in different ways.

- Overloading is a form of **polymorphism** in OOP. Polymorphism allows objects or methods to act in different ways, according to the means in which they are used.

- It is a technique of having more than one constructor with different parameter lists.

- They are arranged in a way that each constructor performs a different task.

Constructor Overloading in Java

- **When do we need Constructor Overloading?**
- Sometimes there is a need of initializing an object in different ways. This can be done using constructor overloading. Different constructors can do different work by implementing different line of codes and are called based on the type and no of parameters passed.
- According to the situation , a constructor is called with specific number of parameters among overloaded constructors.
- **Do we have destructors in Java?**
- No, Because Java is a garbage collected language you cannot predict when (or even if) an object will be destroyed. Hence there is no direct equivalent of a destructor.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
      id = i;
    name = n;
    age=a;
    }
void display(){System.out.println(id+"
"+name+" "+age);}

    public static void main(String
args[]){
    Student5 s1 = new
Student5(111,"Karan");
    Student5 s2 = new
Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

- **Constructor Overloading**

1. Writing more than 1 constructor in a class with a unique set of arguments is called as Constructor Overloading
2. All constructors will have the name of the class
3. Overloaded constructor will be executed at the time of instantiating an object
4. An overloaded constructor cannot be static as a constructor relates to the creation of an object
5. An overloaded constructor cannot be final as constructor is not derived by subclasses it won't make sense
6. An overloaded constructor can be private to prevent using it for instantiating from outside of the class

- **Method Overloading**

1. Writing more than one method within a class with a unique set of arguments is called as method overloading
2. All methods must share the same name
3. An overloaded method if not static can only be called after instantiating the object as per the requirement
4. Overloaded method can be static, and it can be accessed without creating an object
5. An overloaded method can be final to prevent subclasses to override the method
6. An overloaded method can be private to prevent access to call that method outside of the class

static keyword

- The static keyword in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object),

For example: the company name of employees, college name of students, etc.

List the restrictions associated with static.

- Restrictions on static blocks and static methods
- You cannot access a non-static member (method or, variable) from a static context.
- This and super cannot be used in static context.
- The static method can access only static type data (static type instance variable).
- You cannot override a static method.

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

- Suppose there are 500 students in college, now all instance data members will get memory each time when the object is created.
- All students have its unique rollno and name, so instance data member is good in such case.
- Here, "college" refers to the common property of all objects.
- If we make it static, **this field will get the memory only once.**

Example for Static keyword

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n; }
```

```
//Test class to create and display the values of object
public class TestStaticMethod{
    public static void main(String args[]){
    Student.change();//calling change method
    //creating objects
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    Student s3 = new Student(333,"Sonoo");
    //calling display method
    s1.display();
    s2.display();
    s3.display();}}
```

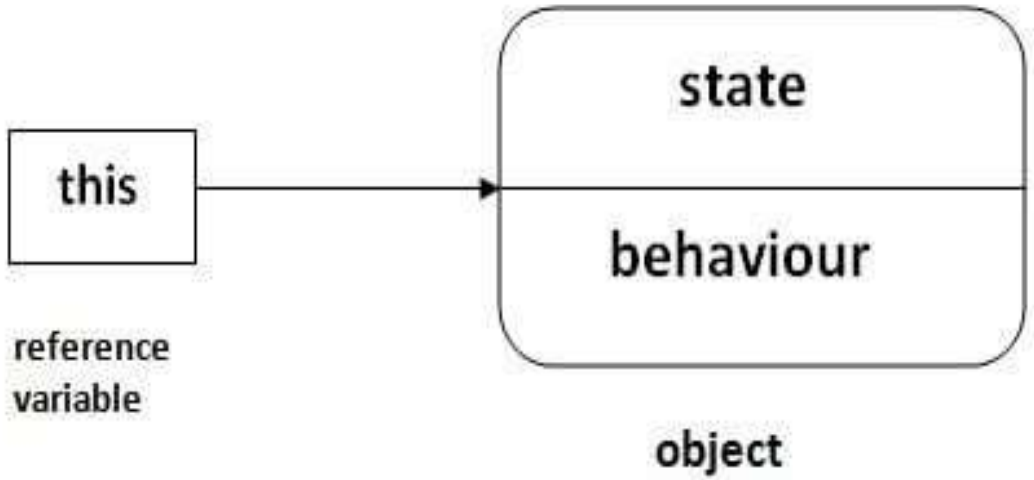
Another example of a static method that performs a normal calculation.

//Java Program to get the cube of a given number using the static method

```
class Calculate{
static int cube(int x){
return x*x*x;
}
public static void main(String args[]){
int result=Calculate.cube(5);
System.out.println(result);
}
}
```

“this” keyword

- “this” is a reference variable that refers to the current object.



- **Usage of Java this keyword**
- Here is given the 6 usage of java this keyword.

- this can be used to refer current class instance variable.
- this can be used to invoke current class method (implicitly)
- this() can be used to invoke current class constructor.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.
- this can be used to return the current class instance from the method.

“this”: to refer current class instance variable

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
```

```
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Method overloading

- In Java it is possible to define two or more methods within the same class that are having the same name, but their parameter declarations are different.
- In the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method overloading is one of the ways of Java implementation of polymorphism.

- If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

- Overloaded methods can be either static or non static.

- Compiler uses method signature to check whether the method is overloaded or duplicated.
- Duplicate methods will have same method signatures.

  i.e same name, same number of arguments and same types of arguments.
- Overloaded methods will also have same name but differ in number of arguments or else types of arguments.
- It is not possible to have two methods in a class with same method signature but different return types because compiler will give duplicate method error.
- Compiler checks only method signature for duplication not the return types.
- If two methods have same method signature, straight away it gives compile time error.

- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int, int ,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.
- So, we perform method overloading to figure out the program quickly.

Example:

```
int add(int x, int y)   //version1
{
    cal = x + y;
    return(cal);
}
int add(int z)   //version2
{
   cal = z + 10;
   return(cal);
}
float add(float x, float y)  //version3
{
   val = x + y;
   return(val);
}
```

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Advantage of method overloading

- Method overloading *increases the readability of the program*.

Different ways to overload the method

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

Method Overloading: changing no. of arguments

```
class Adder
{
static int add(int a, int b)
{
return a+b;
}
static int add(int a, int b, int c)
{
return a+b+c;}
}
```

```
class TestOverloading1
{
public static void main(String[] args)
{
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}
}
```

- we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

Method Overloading: changing data type of arguments

```
class Adder
{
static int add(int a, int b)
{return a+b;}
static double add(double a, double b)
{return a+b;}
}
```

Inheritance in Java

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

- 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.

- **Terms used in Inheritance.**

- **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.

New class = Derived class/Sub class/ 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.

Old class = Base class/Super class/Parent class

- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

- **Inheritance Syntax in Java**

```
class derived_class extends base_class
{
   //methods
   //fields}
```

- **General format for Inheritance**

```
class superclass
{
 // superclass data variables
 // superclass member functions
}
class subclass extends superclass
{
 // subclass data variables
 // subclass member functions }
```

- **Inheritance Program Example**

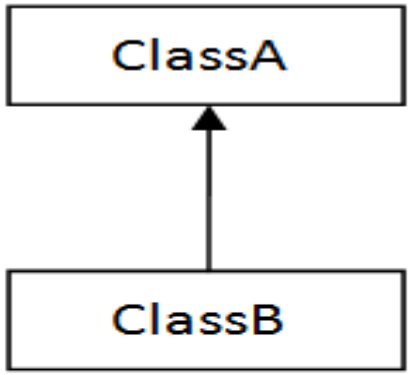
```
class Base
 {
 public void M1()
 {
 System.out.println(“ Base Class Method ”);
 }
 }
class Derived extends Base
{
 public void M2()
 {
 System.out.printIn(“ Derived Class Methods “);
 }
}
```

```
class Test
{
 public static void main(String[] args)
 {
 Derived d = new Derived(); // creating
object
 d.M1(); // print Base Class Method
 d.M2(); // print Derived Class Method
 }
}
```

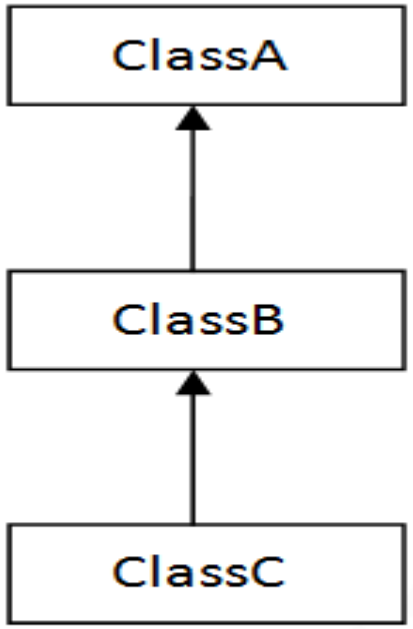
```
Class PetAnimal {

  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }
}

// inherit from PetAnimal
class Dog extends PetAnimal {

  // new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }
}
```

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

    // create an object of the
subclass
    Dog labrador = new Dog();

    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();

    // call method of superclass
    // using object of subclass
    labrador.eat();
  }
}
```

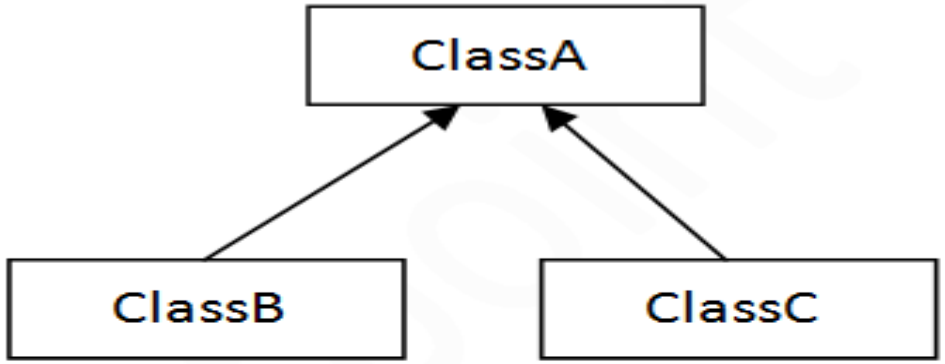
Types of inheritance in java



1) Single



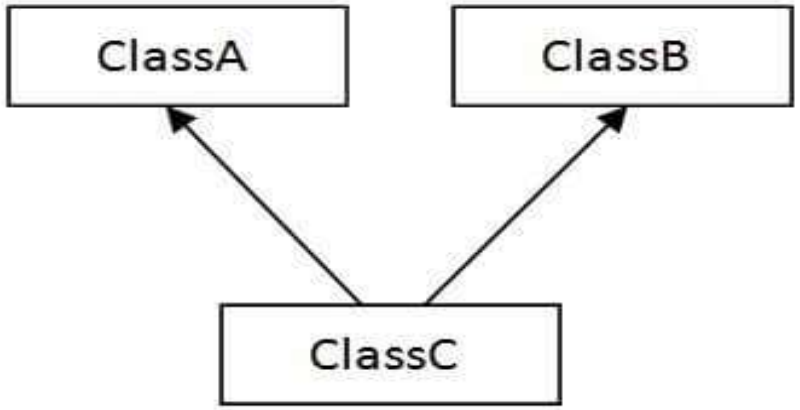
2) Multilevel



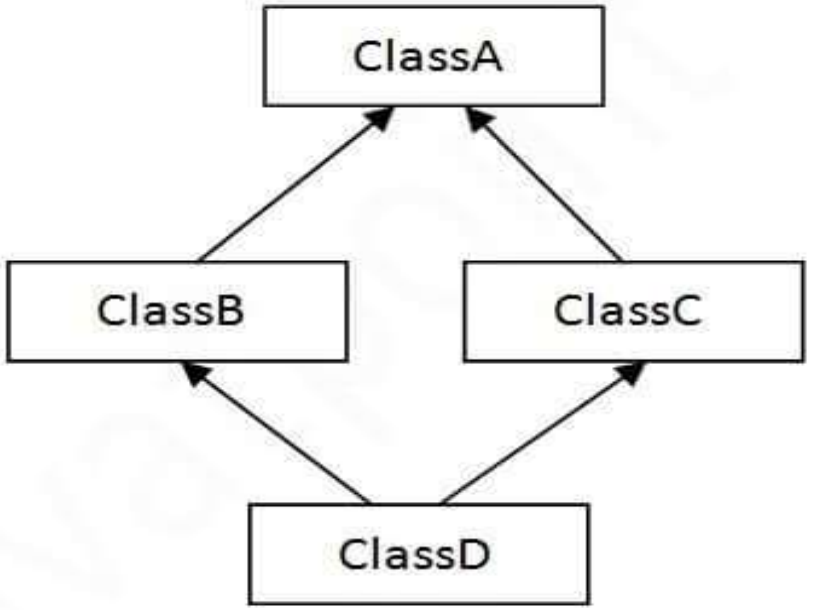
3) Hierarchical

When one class inherits multiple classes, it is known as multiple inheritance.

- For Example:



4) Multiple



5) Hybrid

Single Inheritance Example

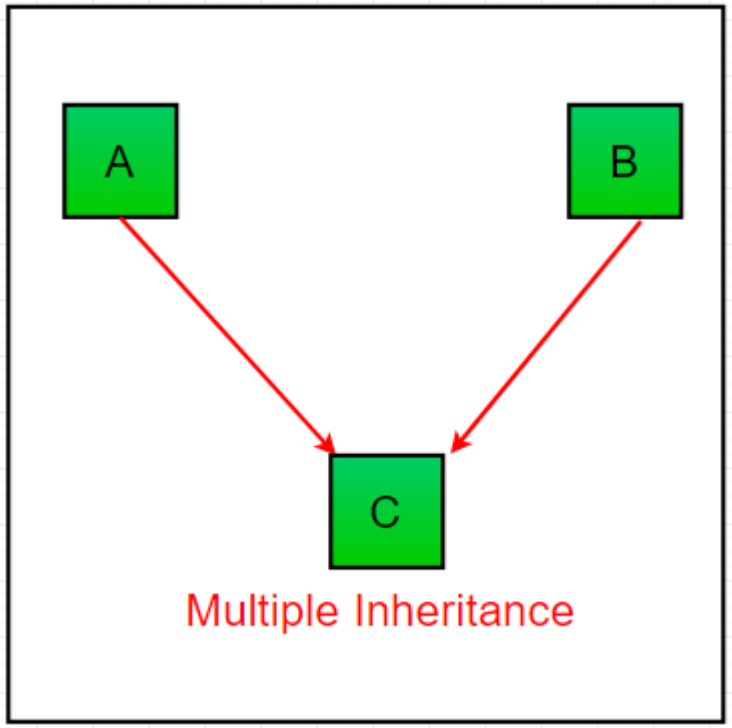
- It is the simplest type of inheritance in java.
- In this, a class inherits the properties from a single class.
- The class which inherits is called the derived class or child class or subclass, while the class from which the derived class inherits is called the base class or superclass or parent class
- In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Example for 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();
}}
```

Multiple Inheritance in Java

- It is the capability of creating a single class with multiple superclasses.



Example.

```
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();
}}
```

Hierarchical Inheritance Example

- Hierarchical inheritance is one of the types of inheritance where multiple child classes inherit the methods and properties of the same parent class.
- Hierarchical inheritance not only reduces the code length but also increases the code modularity.

```
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
}}
```

Super keyword in Java

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable
- There are three usages of super keyword in Java:
- 1. We can invoke the superclass variables.
- 2. We can invoke the superclass methods.
- 3. We can invoke the superclass constructor

```
class Superclass
{
 int i =20;
void display()
{
 System.out.println(“Superclass display method”);
 }
}
class Subclass extends Superclass
{
 int i = 100;
 void display()
 {
 super.display();
 System.out.println(“Subclass display method”);
 System.out.println(“ i value =”+i);
 System.out.println(“superclass i value =”+super.i);
 }
}
class SuperUse
{
 public static void main(String args[])
{
 Subclass obj = new Subclass();
 obj.display();
}
}
```

1) super is used to refer immediate parent class instance variable.

```
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 TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

1) super is used to refer immediate parent class instance variable.

```
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 TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

2) super can be used to invoke parent class method.

```
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 TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();  }}
```

3) super is used to invoke parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

SUPER KEYWORD : REAL WORLD USE

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name;
}
}
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary;
}
void display(){System.out.println(id+" "+name+" "+salary);}
}
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

Difference between “super” and “this” keyword.

super() keyword	this() keyword
super() calls the parent constructor	this() can be used to invoke the current class constructor
It can be used to call methods from the parent.	It can be passed as an argument in the method call.
It is returned with no arguments.	It can be passed as an argument in the constructor call.
It can be used with instance members.	It is used to return the current class instance from the method.

Recursion

- Recursion is the technique of making a function call itself.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.
- Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

- **Syntax:**

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

- **What do you mean by Direct recursion?**

When in the body of a method if there is a call to the same method, we say that the method is directly recursive. That means Direct recursion occurs when a method invokes itself.

- **What do you mean by Indirect recursion?**

If method A calls method B, method B calls method C, and method C calls method A we call the methods A, B and C indirectly recursive or mutually recursive.

.

Direct Recursion	Indirect Recursion
In the direct recursion, only one function is called by itself.	In indirect recursion more than one function are by the other function and number of times.
direct recursion makes overhead.	The indirect recursion does not make any overhead as direct recursion
The direct recursion called by the same function	While the indirect function called by the other function
In direct function, when function called next time, value of local variable will stored	but in indirect recursion, value will automatically lost when any other function is called local variable
Direct function engaged memory location	while local variable of indirect function not engaged it

Structure of direct function
`int num()`
`{`
`...`
`...`
`int num();`
`}`
	Structure of indirect function
	`int num()`
	`{`
	`...`
	`...`
	`int sum();`
	`}`
	`int sum()`
	`{`

Example on Recursion:Printing “Hello”

```
public class RecursionExample1 {
static void p(){
System.out.println("hello");
p();
}

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

Example on Recursion:Factorial of the given number

```
public class Recursion {
    static int factorial(int n){
        if (n == 1)
          return 1;
        else
          return(n * factorial(n-1));
    }

public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5)); }
}
```

Access Modifiers

- It is used to set the accessibility of classes, constructors, methods, and other members of Java.
- **Four Types of Access Modifiers**
- **Private:** We can access the private modifier only within the same class and not from outside the class.
- **Default:** We can access the default modifier only within the same package and not from outside the package. And also, if we do not specify any access modifier it will automatically consider it as default.
- **Protected:** We can access the protected modifier within the same package and also from outside the package with the help of the child class. If we do not make the child class, we cannot access it from outside the package. So inheritance is a must for accessing it from outside the package.
- **Public:** We can access the public modifier from anywhere. We can access public modifiers from within the class as well as from outside the class and also within the package and outside the package.

Difference between access modefiers

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Non- Access Modifiers

- It is a keywords that allow us to modify our classes by fine-tuning the accessibility of its members.

- **Non-Access modifiers are:**
- static.
- final.
- abstract.
- synchronized.
- volatile.
- transient.
- native.

“Final” keyword

- The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).
- The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).It is also called as "modifier".
- The final keyword in java is used to restrict the user.
- Final can be:
- variable
- method
- class

1) Java final variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).
- **For example:**

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

2) Java final method

- If you make any method as final, you cannot override it.

- **For example:**

```
class Bike{
  final void run(){System.out.println("running");}
}
class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

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

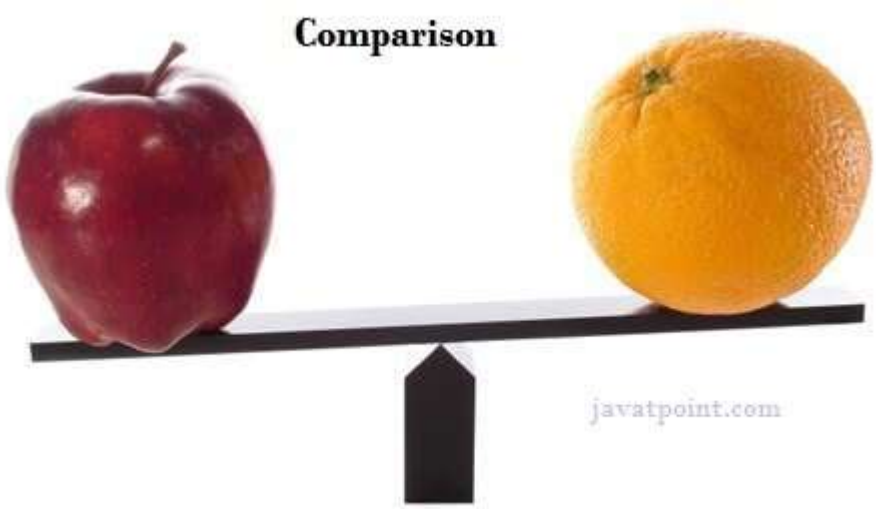
3) Java final class

- If you make any class as final, you cannot extend it.
- **For example:**

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Java String compare

- It is used in authentication (by equals() method), sorting (by compareTo() method), reference matching (by == operator) etc
- The index of the first character is 0, the second character is 1, and so on.



String Concatenation in Java

- In Java, String concatenation forms a new String that is the combination of multiple strings.
- There are two ways to concatenate strings in Java:
- **1) String Concatenation by + (String concatenation) operator**

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 }
}
```

String Concatenation in Java

2) String Concatenation by concat() method.

- **public String concat(String another)**

Example:

```
class TestStringConcatenation3{
 public static void main(String args[]){
   String s1="Sachin ";
   String s2="Tendulkar";
   String s3=s1.concat(s2);
   System.out.println(s3);//Sachin Tendulkar
  }
}
```

Java StringBuffer Class

- Java StringBuffer class is used to create mutable (modifiable) String objects.

- **Important Constructors of StringBuffer Class:**
- **StringBuffer()-**It creates an empty String buffer with the initial capacity of 16.
- **StringBuffer(String str)-**It creates a String buffer with the specified string..
- StringBuffer(int capacity)-It creates an empty String buffer with the specified capacity as length.
- There are some important methods of StringBuffer class those are:
- insert(),delete(),replace(),reverse(),capacity(),length(),ensureCapacity(),char At(),substring(),etc.

- **Example:delete(int startIndex, int endIndex)-**It is used to delete the string from specified startIndex and endIndex.

Java StringBuilder Class

- The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. However, the StringBuilder class differs from the StringBuffer class on the basis of synchronization.

- The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

- String is immutable whereas StringBuffer and StringBuilder are mutable classes.

Java StringBuilder Class

- **1) StringBuilder append() method.**
- The StringBuilder append() method concatenates the given argument with this String.

```
class StringBuilderExample{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```

Output:HelloJava

3) StringBuilder replace() method.

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

Example:

```
class StringBuilderExample3{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

Output:HJavalo

4) StringBuilder delete() method.

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

Example:

```
class StringBuilderExample4{
public static void main(String args[]){
StringBuilder sb=new StringBuilder("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```

Output:Hlo

Java String charAt() Method

- The charAt() method returns the character at the specified index in a string
- The index of the first character is 0, the second character is 1, and so on.
- A char value at the specified index of this string.
- The first char value is at index 0
- **StringIndexOutOfBoundsException**, it will return if the given index number is greater than or equal to this string length or a negative number.

- **Syntax**

  public char charAt(int index)

- **Example:**
- String myStr = "Hello";
- char result = myStr.charAt(0);
- System.out.println(result);

Java String charAt() Method

- The charAt() method returns the character at the specified index in a string
- The index of the first character is 0, the second character is 1, and so on.
- A char value at the specified index of this string.
- The first char value is at index 0
- **StringIndexOutOfBoundsException**, it will return if the given index number is greater than or equal to this string length or a negative number.

- **Syntax**

  public char charAt(int index)

- **Example:**
- String myStr = "Hello";
- char result = myStr.charAt(0);
- System.out.println(result);

Java String charAt() Method

- The charAt() method returns the character at the specified index in a string
- The index of the first character is 0, the second character is 1, and so on.
- A char value at the specified index of this string.
- The first char value is at index 0
- **StringIndexOutOfBoundsException**, it will return if the given index number is greater than or equal to this string length or a negative number.

- **Syntax**

  public char charAt(int index)

- **Example:**
- String myStr = "Hello";
- char result = myStr.charAt(0);
- System.out.println(result);

Java String compareTo()

- This is a method compares the given string with the current string.
- If the first string is lexicographically greater than the second string, it returns a positive number (difference of character value).
- If the first string is less than the second string lexicographically, it returns a negative number.
-
- If the first string is lexicographically equal to the second string, it returns 0.
- **Syntax**
- public int compareTo(String anotherString)

Example for compareTo()

```
int compareTo(String anotherString) {
int length1 = value.length;
int length2 = anotherString.value.length;
int limit = Math.min(length1, length2);
char v1[] = value;
char v2[] = anotherString.value;
int i = 0;
while (i < limit) {
    char ch1 = v1[i];
    char ch2 = v2[i];
    if (ch1 != ch2) {
        return ch1 - ch2;  }
    i++;
}
return length1 - length2;  }
```

Java String equals()

- The Java String class equals() method compares the two given strings based on the content of the string.
- If any character is not matched, it returns false.If all characters are matched, it returns true.

Example:

```
public class EqualsExample{
public static void main(String args[]){
String s1="javatpoint";
String s2="javatpoint";
String s3="JAVATPOINT";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}}
```

Output:

1. **true**
2. **false**
3. **false**

Java String getChars()

- The Java String class getChars() method copies the content of this string into a specified char array.
- There are four arguments passed in the getChars() method. **Example:**

alse because content is not same.

- **Signature.**

public void getChars(int srcBeginIndex, int srcEndIndex, char[] destination, int dstBeginIndex)

- **Parameters:**
- int srcBeginIndex: The index from where copying of characters is started.

- int srcEndIndex: The index which is next to the last character that is getting copied.

- Char[] destination: The char array where characters from the string that invokes the getChars() method is getting copied.

- **Java String getChars() Method Example**

```
public class StringGetCharsExample{
public static void main(String args[]){
 String str = new String("hello javatpoint how r u");
      char[] ch = new char[10];
      try{
         str.getChars(6, 16, ch, 0);
         System.out.println(ch);
      }catch(Exception ex){System.out.println(ex);}
}}
```

Java String length()

- The Java String class length() method finds the length of a string.

- **Signature**
- public int length()

- **Java String length() method example:**
- public class LengthExample{
- public static void main(String args[]){
- String s1="javatpoint";
- String s2="python";
- System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
- System.out.println("string length is: "+s2.length());//6 is the length of python string
- }}

Java String length()

- **Internal implementation:**

```
public int length() {
return value.length;
}
```

- **Java String length() method example:**
- public class LengthExample{
- public static void main(String args[]){
- String s1="javatpoint";
- String s2="python";
- System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string
- System.out.println("string length is: "+s2.length());//6 is the length of python string
- }}

https://www.javatpoint.com/java-string

Students you can also refer this link for more detail explanation of string concatenation and its data types.

Packages In Java

- It is a mechanism to encapsulate a group of classes, sub packages and interfaces.
- A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
- Packages are used for:

1. 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
2. Making searching/locating and usage of classes, interfaces, enumerations and annotations easier

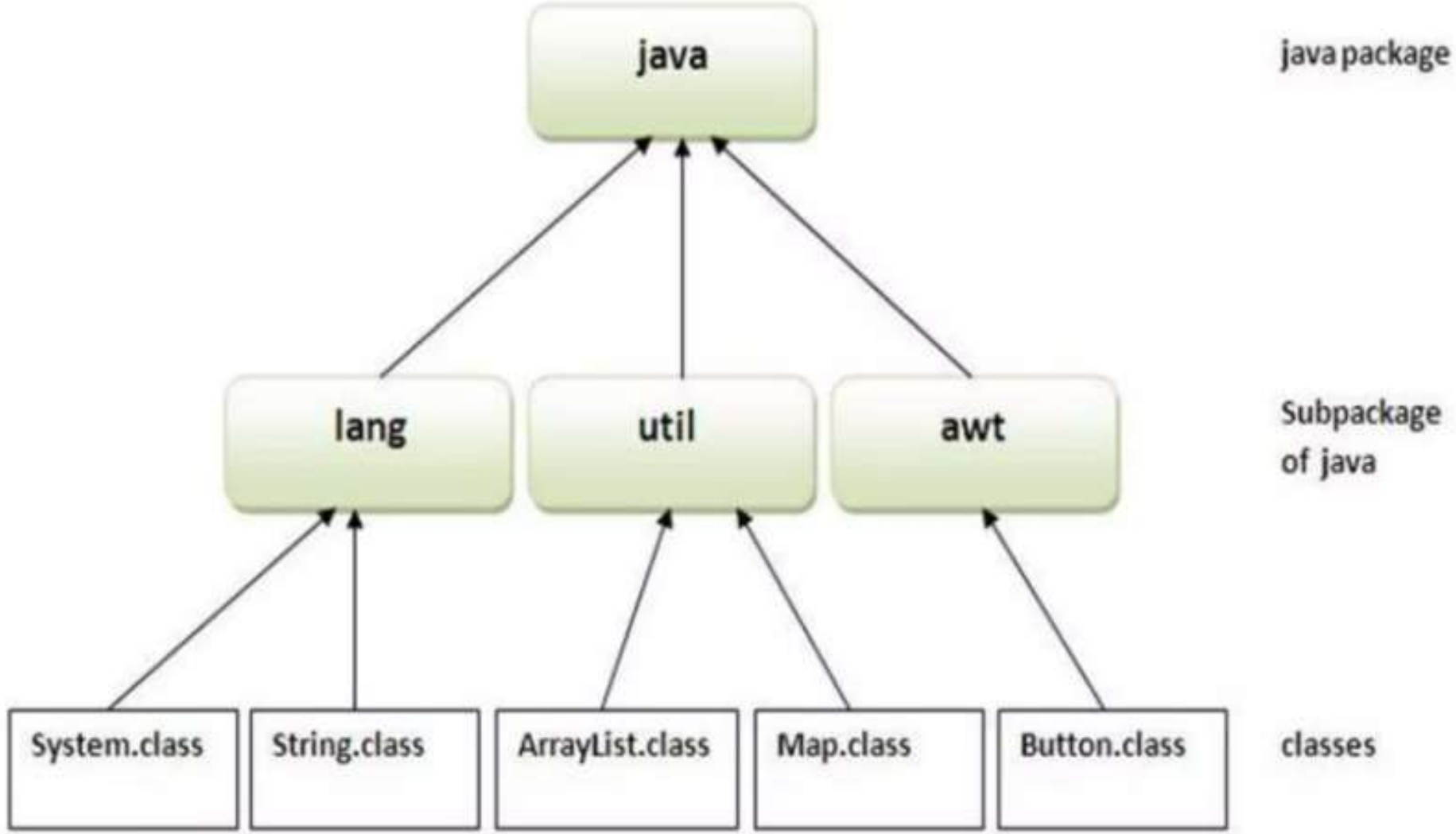
PACKAGES

- Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages.
- In java there are already many predefined packages that we use while programming.
    - For example: java.lang, java.io, java.util etc.
- One of the most useful feature of java is that we can define our own packages

PACKAGES Cont...

Advantages of using a package

- Reusability: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- Easy to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to “name-space collisions”. Packages are a way of avoiding “name-space collisions”.


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

PACKAGES Cont...

Defining a Package:

- This statement should be used in the beginning of the program to include that program in that particular package.

  package <package name>;

- The **package keyword** is used to create a package in java.

To Compile: javac -d directory javafilename.java

To Run: java packagename.classname

- The -d switch specifies the destination where to put the generated class file
- If you want to keep the package within the same directory, you can use . (dot).

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.

Using packagename.*

- 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.

Using packagename.classname

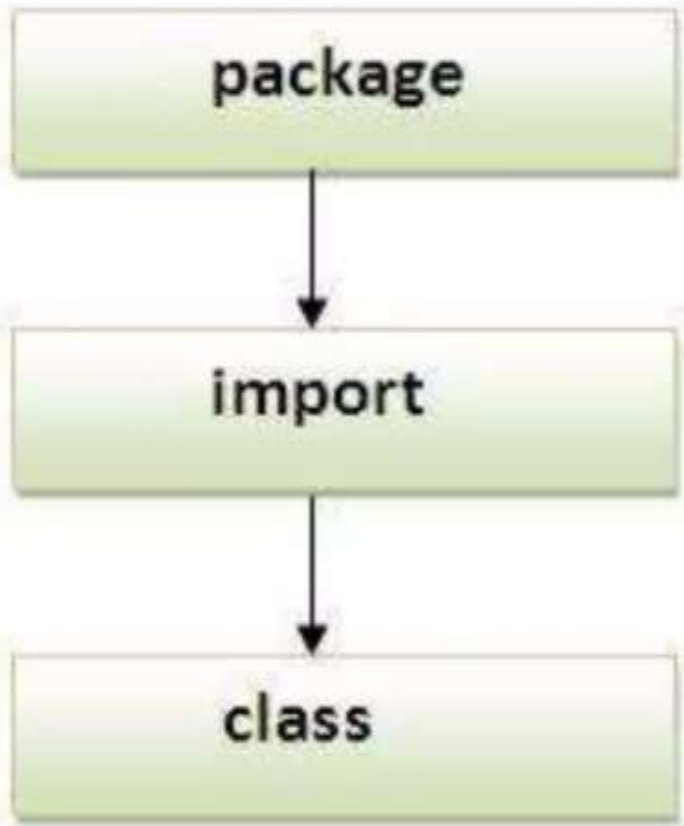
- If you import package.classname then only declared class of this package will be accessible.

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: 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.

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



Packaging up multiple classes

Package with Multiple Public Classes

- A Java source file can have only one class declared as **public**, we cannot put two or more public classes together in a**.java** file. This is because of the restriction that the file name should be same as the name of the public class with **.java**extension.
- If we want to multiple classes under consideration are to be declared as **public**, we have to store them in **separate source files** and attach the **package** statement as the first statement in those source files.

```
//Save as A.java
package javapoint;
Public class B{}
```

```
//Save as B.java
package java
public class a{}
```

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

To Compile:

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

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

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

e:\sources> java mypack.Simple

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

- The -classpath switch can be used with javac and java tool.
- To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:
    - **e:\sources> java -classpath c:\classes mypack.Simple**

Access Protection in Packages

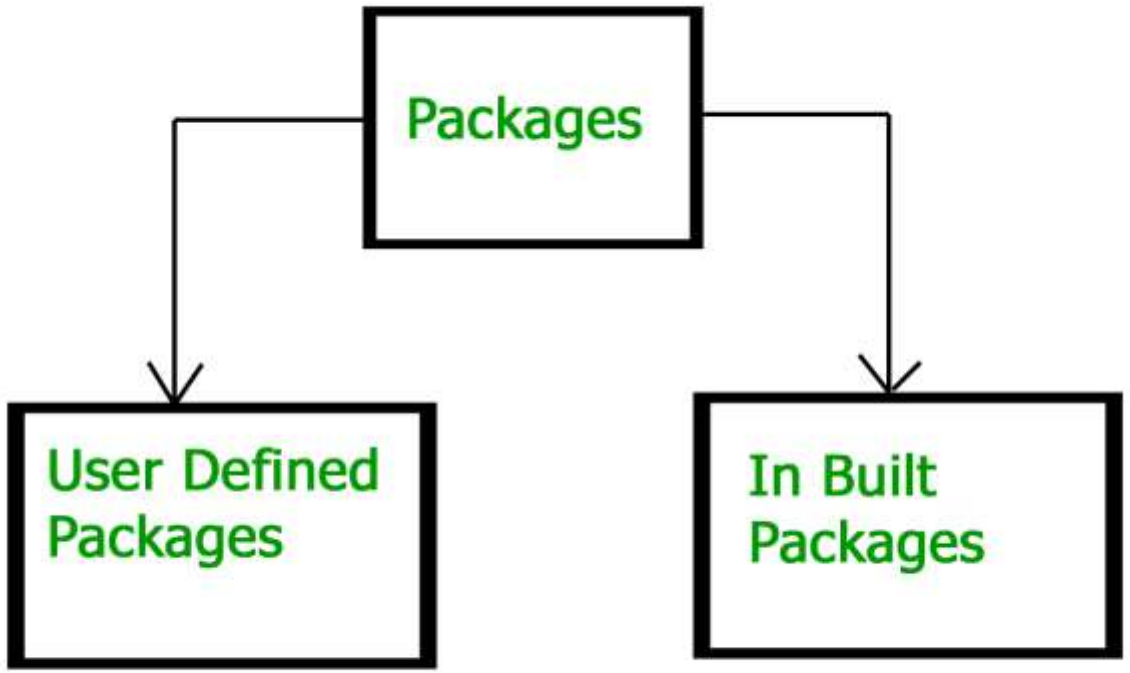
- Access modifiers define the scope of the class and its members (data and methods).
- There are four categories, provided by Java regarding the visibility of the class members between classes and packages:
    - Subclasses in the same package
    - Non-subclasses in the same package
    - Subclasses in different packages
    - Classes that are neither in the same package nor subclasses

Subpackages

- Packages that are inside another package are the subpackages. These are not imported by default, they have to imported explicitly.
- Also, members of a subpackage have no access privileges,

  i.e., they are considered as different package for protected and default access specifiers.
- Example :
- **import java.util.*;**
- util is a subpackage created inside java package.

Types of packages.



Built-in Packages:

These packages consist of a large number of classes which are a part of Java API.

1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.

2) **java.io:** Contains classed for supporting input / output operations.

3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.

4) **java.applet:** Contains classes for creating Applets.

5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

6) **java.net:** Contain classes for supporting networking operations.

User-defined packages

- These are the packages that are defined by the user.
- First we create a directory myPackage (name should be same as the name of the package).
- Then create the MyClass inside the directory with the first statement being the package names.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

```
/* import 'MyClass' class from 'names'
myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

Directory structure

- The package name is closely associated with the directory structure used to store the classes.
- The classes belonging to a specific package are stored together in the same directory.
- Next they are stored in a sub-directory structure specified by its package name.
- For example:
- the class Circle of package com.zzz.project1.subproject2 is stored as “$BASE_DIR\com\zzz\project1\subproject2\Circle.class”, where $BASE_DIR denotes the base directory of the package.
- Clearly, the “dot” in the package name corresponds to a sub-directory of the file system.

- The base directory ($BASE_DIR) could be located anywhere in the file system.
- Hence, the Java compiler and runtime must be informed about the location of the $BASE_DIR so as to locate the classes.
- This is accomplished by an environment variable called CLASSPATH.
- CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

Setting CLASSPATH:

- CLASSPATH can be set by any of the following ways:
- CLASSPATH can be set permanently in the environment:
- In Windows, choose control panel ? System ? Advanced ? Environment Variables ? choose “System Variables” (for all the users) or “User Variables” (only the currently login user) ? choose “Edit” (if CLASSPATH already exists) or “New” ? Enter “CLASSPATH” as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar”).
- Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH.
- To check the current setting of the CLASSPATH, issue the following command:
- > SET CLASSPATH

CLASSPATH

- It is a environmental variable, which contains the path for the default-working directory (.).
- The specific location that java compiler will consider, as the root of any package hierarchy is, controlled by Classpath
- The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory.
- There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:

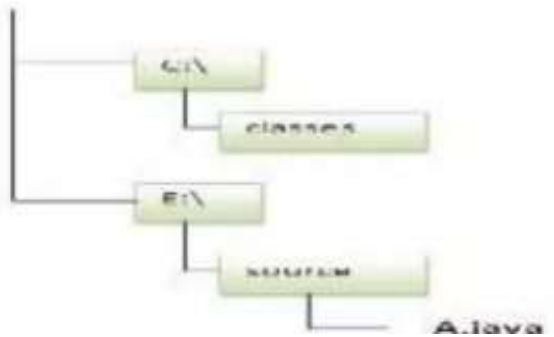


Illustration of user-defined packages:
Creating our first package:
File name – ClassOne.java

- package package_name;

- public class ClassOne {
- 	public void methodClassOne() {
- 		System.out.println("Hello there its ClassOne");
- 	}
- }

Creating our second package:
File name – ClassTwo.java

```
package package_one;


public class ClassTwo {
	public void methodClassTwo(){
		System.out.println("Hello there i am ClassTwo");
	}
}
```

Making use of both the created packages:
File name – Testing.java

```
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
	public static void main(String[] args){
		ClassTwo a = new ClassTwo();
		ClassOne b = new ClassOne();
		a.methodClassTwo();
		b.methodClassOne();
	}
}
```

Important points:

- Every class is part of some package.
- If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
- All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
- If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
- We can access public classes in another (named) package using: package-name.class-name

Access Modifiers

The three main access modifiers *private*, *public* and *protected* provides a range of ways to access required by these categories.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
same package subclass	No	Yes	Yes	Yes
same package non - subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package mon-subclass	No	No	No	Yes

Try These:

- **Built-In :**

```
import java.util.Scanner;


class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");


    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

```
import java.io.*;
class Fruit {// parent class fruit
        private String fruitName;      // field name of parent class
        public void setFruitName(String fruitName)
        {
                this.fruitName = fruitName;
        }
        public String getFruitName()
        {
                return this.fruitName + " is a Fruit";
        }}
class Mango extends Fruit{    //child class mango
        public static void main(String args[])
        {
                Fruit fruit = new Mango();    // reference of instance of obj mango
                        System.out.println("This fruit name is mango");
                fruit.setFruitName("Mango");
                System.out.println(fruit.getFruitName());}}
```

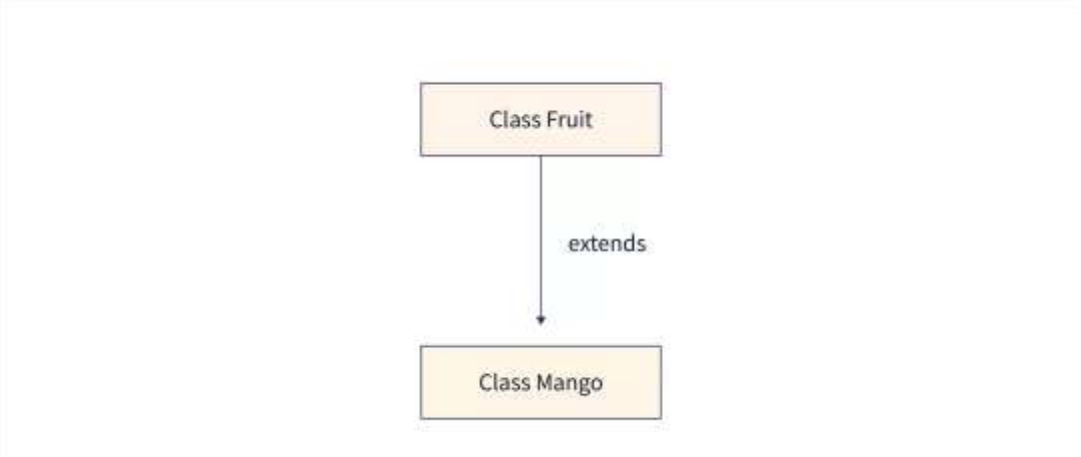
Interface in Java

- 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, 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 can be achieved by extending a class or interface by using the keyword ‘extends’.



In the given image, the class mango extends a class fruit. This means that fruit is a parent class of mango and the class mango is said to have IS-A relationship with class fruit. Hence we say that mango IS-A fruit.

Why use Java interface?

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

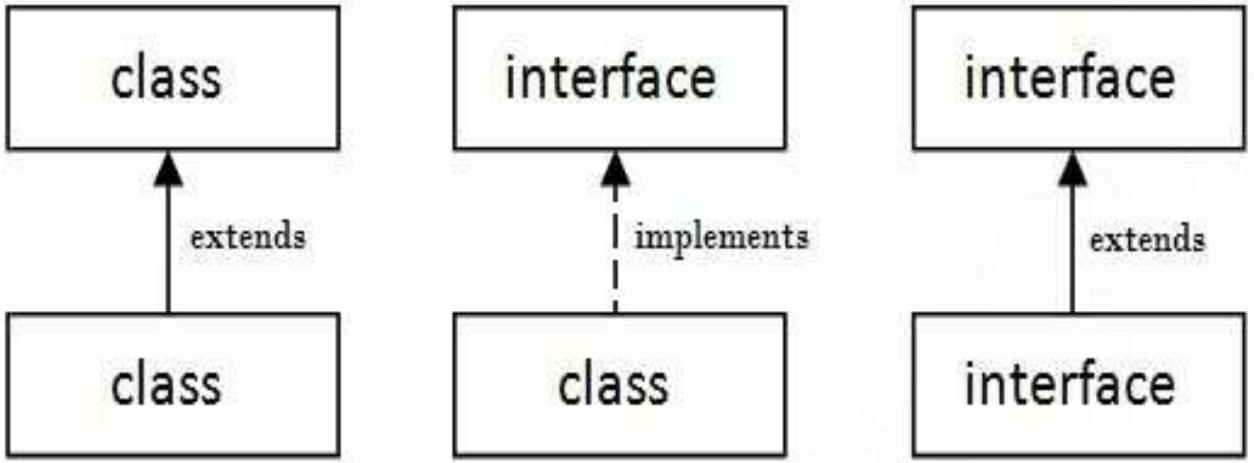
- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

How to declare an interface?

- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.
- **Syntax:**

```
interface <interface_name>{

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

The relationship between classes and interfaces



Java Interface Example: Drawable

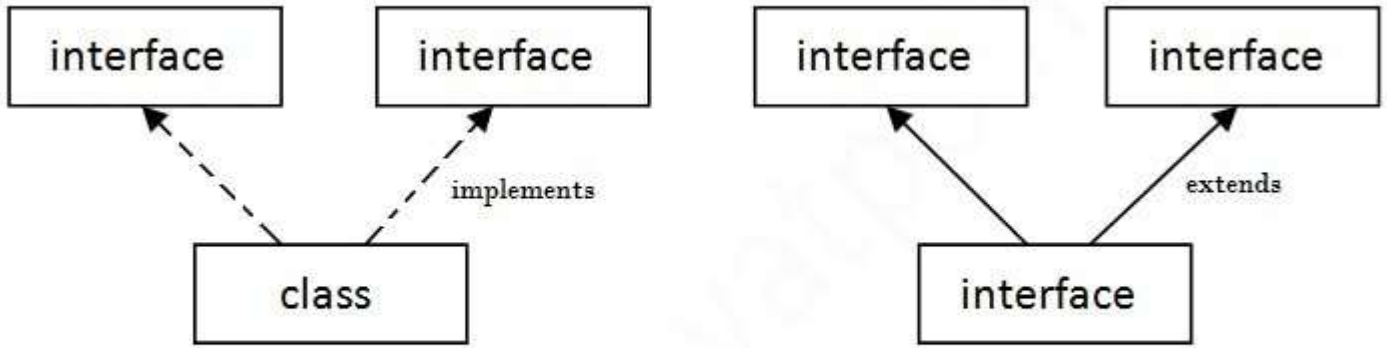
```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}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

Java Interface Example: Bank

```
interface Bank{
float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());  }}
```

Multiple inheritance in Java by interface:

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

- What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Default Method in Interface

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

Static Method in Interface

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Exception Handling in Java

- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

- **What is Exception in Java?**
  Dictionary Meaning: Exception is an abnormal condition.

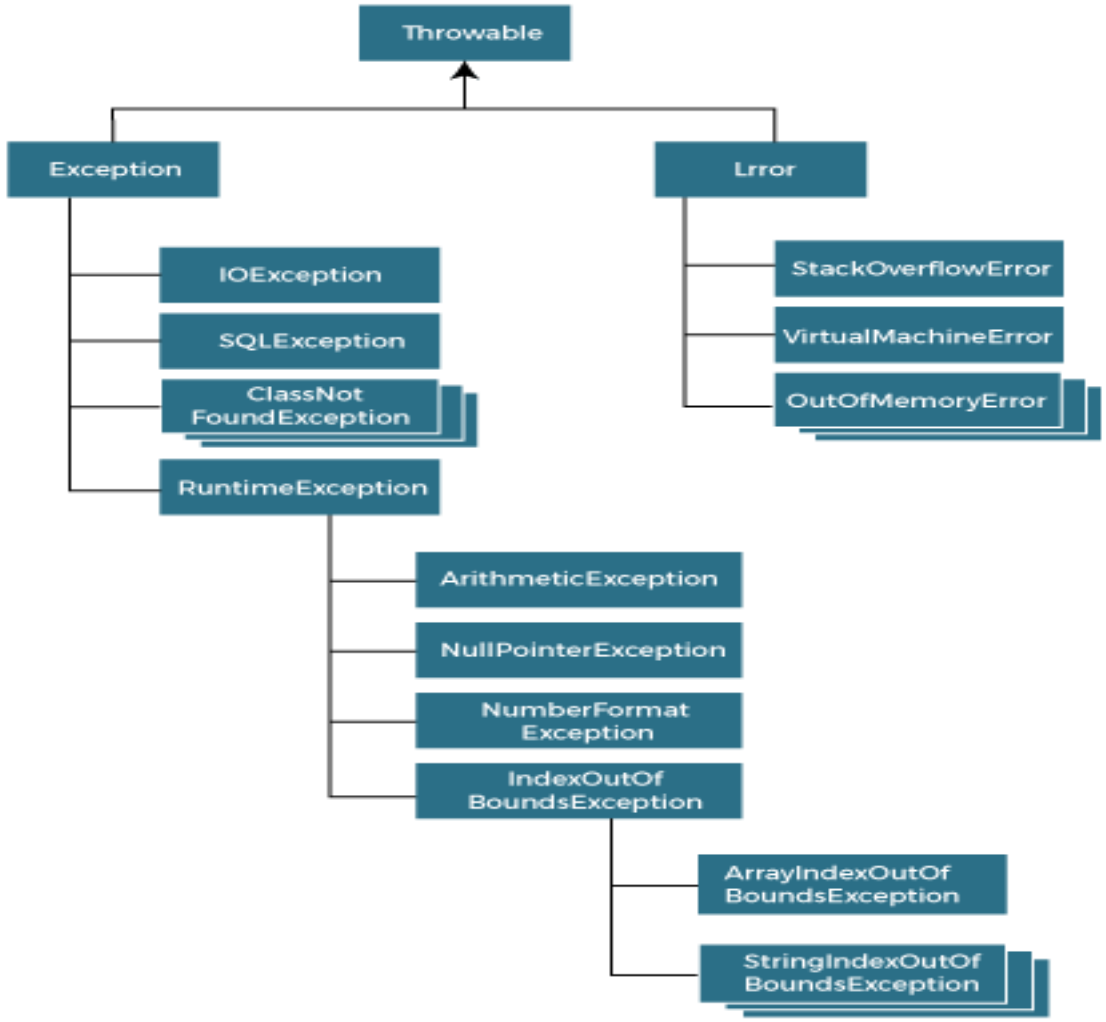
In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

- **What is Exception Handling?**

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Hierarchy of Java Exception classes

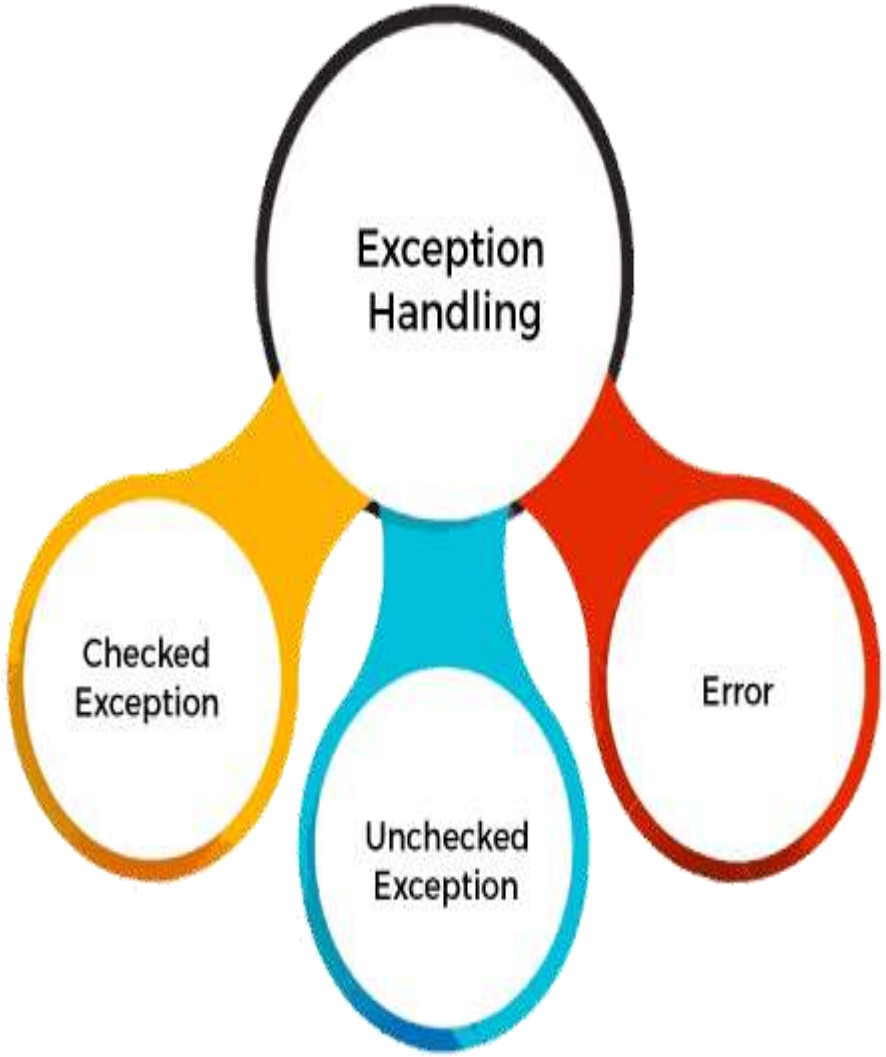
- The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error.



Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked.
- An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

- Checked Exception
- Unchecked Exception
- Error



- **Difference between Checked and Unchecked Exceptions:**
- **1) Checked Exception**
- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

- **2) Unchecked Exception**
- The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

- **3) Error**
- Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java Exception Keywords	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

- **What is Exception Handling?**

  Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Common Scenarios of Java Exceptions

- There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs.

- If we divide any number by zero, there occurs an ArithmeticException.
- int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

- If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.
- String s=null;
- System.out.println(s.length());//NullPointerException

Common Scenarios of Java Exceptions

3) A scenario where NumberFormatException occurs

- If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.
- String s="abc";
- int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

- When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.
- int a[]=new int[5];
- a[10]=50; //ArrayIndexOutOfBoundsException

Difference between Abstract Classes & Interfaces

Abstract Classes		Interfaces
Abstract classes are fast.	Speed	Interfaces are slow.
Abstract classes can extend only one class.	Multiple Inheritance	Interface can implement several interfaces.
You can define fields as well as constants.	Defined Fields	You cannot define fields in an interface.
A single abstract class can extend one & only one interface.	Extension Limit	A single interface can extend multiple interfaces.

Java Threads

- A Thread is a very light-weighted process, or it is the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.
- Threads allows a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.

- **Creating a Thread**
- There are two ways to create a thread.

Multithreading

- Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently.
- Eachpart of such a program is called a thread, and each thread defines a separate path of execution.
- A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process.
- There are two distinct types of Multitasking i.e. Processor-Based and Thread-Based multitasking.

Q 1.What is the difference between thread-based and process-based multitasking?

Ans: As both are types of multitasking there is very basic difference between the two.

- Process-Based multitasking is a feature that allows your computer to run two or moreprograms concurrently. For example you can listen to music and at the same time chat withyour friends on Facebook using browser.
- In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously.
- For example a text editor can print and at the same time you can edit text provided that those two tasks are perform by separate threads.

Benefits of Multithreading

1. Enables programmers to do multiple things at one time JAVA NOTES – ACHIN JAIN - ASSISTANT PROFESSOR, CSE(NIEC)JAVA NOTES BY ACHIN JAIN 1
2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
3. Improved performance and concurrency
4. Simultaneous access to multiple applications

Life Cycle of a Thread

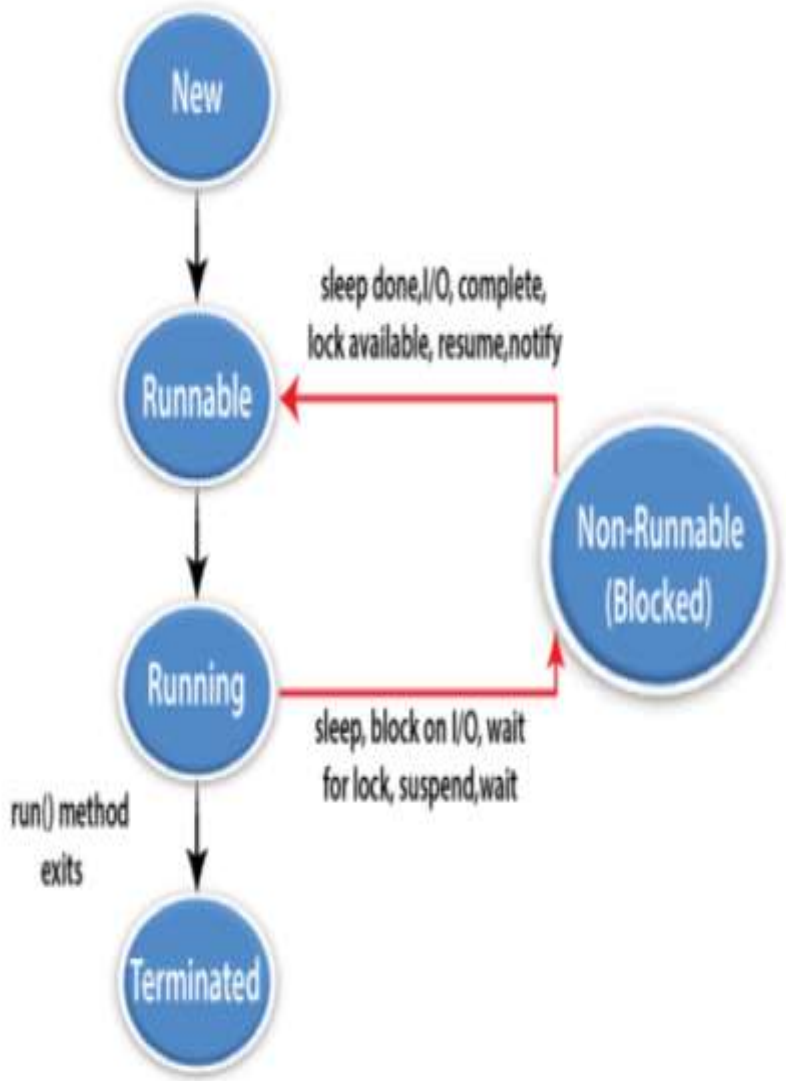
New: Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Active: When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.

Runnable: A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

Running: When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

Blocked or Waiting: Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.



- **Terminated:** A thread reaches the termination state because of the following reasons:
- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.
- A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

Implementation of Thread States

- **public static final Thread.State NEW**
- It represents the first state of a thread that is the NEW state.

- **public static final Thread.State RUNNABLE**
- It represents the runnable state.It means a thread is waiting in the queue to run.

- **public static final Thread.State BLOCKED**
- It represents the blocked state. In this state, the thread is waiting to acquire a lock.

- **public static final Thread.State WAITING**

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

public static final Thread.State TIMED_WAITING
It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

public static final Thread.State TERMINATED
It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

Creating a Thread

- Java defines two ways in which this can be accomplished:
- □ You can implement the Runnable interface.
- □ You can extend the Thread class, itself.
- **Create Thread by Implementing Runnable**
- The easiest way to create a thread is to create a class that implements the Runnable interface.
- To implement Runnable, a class need only implement a single method called run( ), which is declared like this:
- **public void run( )**

- After you create a class that implements Runnable, you will instantiate an object of type
- Thread from within that class. Thread defines several constructors. The one that we will use is shown here:
- **Thread(Runnable threadOb, String threadName);**
- Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName. After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread.
- The start( ) method is shown here:
- **void start( );**

Example to Create a Thread using Runnable Interface

```
class t1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t1.java

C:\NIEC Java>java t1
Thread is Running

C:\NIEC Java>
```

• Create Thread by Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run( ) method,JAVA NOTES – ACHIN JAIN - ASSISTANT PROFESSOR, CSE(NIEC) JAVA NOTES BY ACHIN JAIN 1 which is the entry point for the new thread.
- It must also call start( ) to begin execution of the new thread.

Example to Create a Thread by Extending Thread Class

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t2.java

C:\NIEC Java>java t2
Thread is Running

C:\NIEC Java>
```

Use of stop() Method
The stop() method kills the thread on execution

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) stop();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 stop() method is evoked causing termination of thread execution

Output

```
C:\NIEC Java>java C
A:1
```

Use of sleep() Method

Causes the currently running thread to block for at least the specified number of milliseconds.You need to handle exception while using sleep() method.

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                if(i==2) sleep(1000);
            }
            catch(Exception e)
            {
            }
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 sleep() method is evoked which halts the execution of the thread for 1000 milliseconds. When you see output there is no change but there is delay in execution.

- **Thread class:**
- Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

- **Commonly used Constructors of Thread class:**
- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.

Commonly used methods of Thread class:

- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

- **Starting a thread:**
- The start() method of Thread class is used to start a newly created thread. It performs the following tasks:
- **1) Java Thread Example by extending Thread class**

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");  }

public static void main(String args[]){

Multi t1=new Multi();

t1.start();   }  }
```

- **2) Java Thread Example by implementing Runnable interface**

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above.

```
public class MyThread1
{
// Main method
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");
// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Thread Priority

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java priorities are in the range between MIN_PRIORITY (a constant of 1) andMAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

- **Use of isAlive() and join() method**
- The java.lang.Thread.isAlive() method tests if this thread is alive. A thread is alive if it has been started and has not yet died. Following is the declaration for java.lang.Thread.isAlive() method
- **public final boolean isAlive()**
- This method returns true if this thread is alive, false otherwise.join() method waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

Example

```
class A extends Thread
{
    public void run()
    {
        System.out.println("Status:" + isAlive());
    }
}
class alivetest
{
    public static void main(String args[])
    {
        A a = new A();
        a.start();
        try
        {
            a.join();
        }
        catch(InterruptedException e)
        {}
        System.out.println("Status:" + a.isAlive());
    }
}
```

At this point Thread A is alive so the value gets printed by **isAlive()** method is "***true***"

join() method is called from Thread A which stops executing of further statement until A is Dead

Now isAlive() method returns the value false as the Thread A is complete

Output

```
C:\Achin Jain>java alivetest
Status:true
Status:false
```

- # Interthread Communication
- It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methodsof Object Class:
- **wait( ):** This method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify( ):** This method wakes up the first thread that called wait( ) on the same object.
- **notifyAll( ):** This method wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

THANK YOU