

## **CORE JAVA**

### **Features of Java**

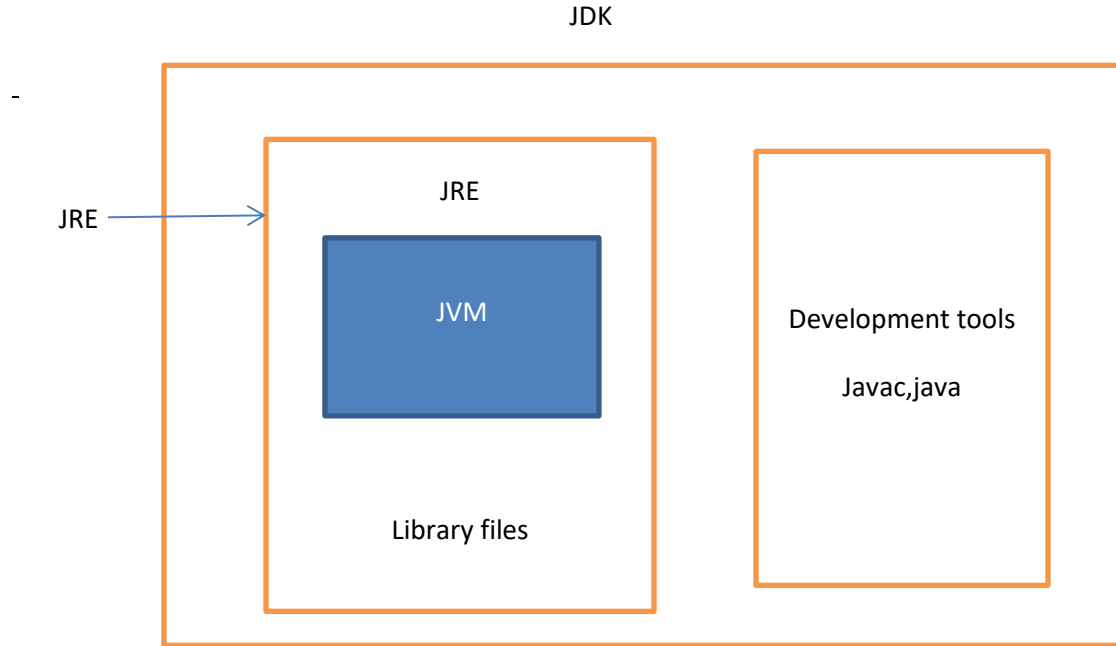
- Simple : Java is very easy to learn, and its syntax is simple, clean and easy to understand.
- Object-Oriented: Everything in Java is an object.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

- Platform Independent: Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms.
- Secured: Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
  - No explicit pointer
  - Java Programs run inside a virtual machine sandbox
- Robust: Robust simply means strong.
  - It uses strong memory management.
  - There is a lack of pointers that avoids security problems.
- Multi-threaded: A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads.

## JDK,JRE and JVM



JDK -> Java Development Kit

JRE -> Java Runtime Environment

JVM -> Java Virtual Machine

JVM -> Load  
verify  
Execute  
Provide runtime environment

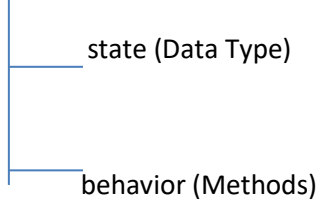
### **Instalation**

- Windows64 is to be used for 64 bit pc. Once installed the kit needs to be configured with the pc using the step below.
- My computer-> Right click->properties->Advanced system settings-> Environment variables->new->variable name: PATH->Variable value: C folder->prgm files->java->jdk->bin-copt the address and past as variable value.

## Java

Java is an **Object Oriented Programing Language**

**Object** –it is an entity



eg: int a

int → datatype

**class** → blue print of object

### Sample Program

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

Class name

- Save the file as Sample.java ,where **Sample** is the class name.
- Always ensure the **file name is same as class name**. while the file is saved.
- C:\users\java>d:
- D:\>cd javaprograms
- D:\javaprograms>javac Sample.java →Compilation
- D:\javaprograms>java Sample →Run

Save → class name.java

Compilation → javac classname.java

Run → java classname

## Errors

- 2-type: **Compile time Error and Run Time Error**
- Compile time error → deals with syntax mistakes
- Run time error → deals with Logical mistakes(eg:divisible by zero)

## Data Types :

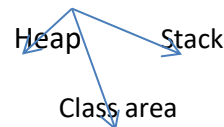
→ Define what type of data to be stored in a variable

- 2 types: **Primitive and Non Primitive**
- Primitive data types are → int, short, byte, char, float, boolean, double
- Non Primitive Data type → String, Array, Integer, Double
- Non primitive data types are also known as **wrapper classes or predefined data types**

## Variables

Container which holds the value, name of memory location.

- 3 types: **Local , instance, static**
- **Local:** declare inside the method
- **Instance:** declare outside method and inside class
- **Static:** declare outside method and inside class
  - Declare with static keyword
  - It saves memory
  - It get memory only once in class area(memory in jvm)



```
Class Sample{  
  
    static int b=20; //static variable  
  
    int c=30; //instance variable
```

```
Public static void main(String args[]){  
  
    Int a=10; //local variable  
  
}  
  
}
```

### **CommandLine Arguments**

```
Class Classname{  
  
public static void main(String args[]){  
  
int a=Integer.parseInt(args[0]);  
  
int b=Integer.parseInt(args[1]);  
  
int c=a+b;  
  
sop( c);  
  
}}
```

### **Conditional Statements**

The [Java if statement](#) is used to test the condition.

It checks [boolean](#) condition: *true* or *false*. There are various types of if statement in Java.

1. Simple if

```
If(condition)  
{  
    True  
}
```

2. IfElse

```
If(condition)  
{  
    true  
}  
else{  
    false  
}
```

3. If else if

```
If(condition 1)
{ //stmt
}
else if(condition 2)
{ //stmt
}
Else if(condition n)
{ //stmt
}
else
{ //stmt
}
```

4. Switch : The Java *switch statement* executes one statement from multiple conditions.

```
switch(expression)
{
case value1: //stmt
    break;
case value2: //stmt
    break;
case value3: //stmt
    break;
.
.
.
case value n: //stmt
    break;
Default: //stmt    (code to be executed if all cases are not matched)
    break;
}
```

## Loops

loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in Java.

- for loop
- while loop
- do-while loop

1. **For loop** : The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed.

```
for(initialization;condition;increment/decrement)
{

    Stmt;

}
```

2. **While loop** : The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed.

```
Initialization;
While(condition)
{

    Stmt;

}
```

→ It is entry controlled

3. **Do While loop** : The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once.

```
do
{

    Stmt;

}while(condition);
```

→ It is exit controlled

→ It runs the loop once without checking the condition due to the usage of “do” initially.

```
class DowhileLoopExample {
    public static void main(String args[]){
        int i=10;
        do{
            System.out.println(i);
            i--;
        }while(i>1);
    }
}
```

## Branching Statements

- **Break** → It breaks the current flow of the program at specified condition.

```
public class BreakExample {
    public static void main(String[] args) {
        //using for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //breaking the loop
                break;
            }
            System.out.println(i);
        }
    }
}
```

- **Continue** : The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition.

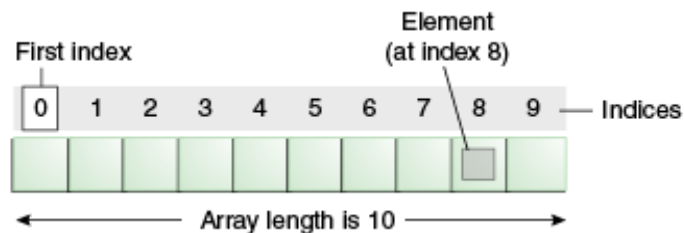
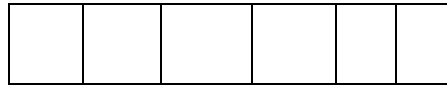
```
public class ContinueExample {
    public static void main(String[] args) {
        //for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //using continue statement
                continue;//it will skip the rest statement
            }
            System.out.println(i);
        }
    }
}
```

## Java Array

- **Java array** is an object which contains elements of a similar data type.
- We can store only a fixed set of elements in a Java array.



- Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



#### Advantage:

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

#### Disadvantage:

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

#### Types of Array:

- Single Dimensional array
- Multidimensional Array

Syntax,

Data Type[] arr; (or)

Data Type []arr; (or)

Data Type arr[];

**Initialization** → `int x[]={1,2,3,4};`

**Instatiation→ arrayRefere.variable=new data type[size];**

**Int x[]=new int[4];**

Java Program to illustrate the use of declaration, instantiation and initialization of Java array in a single line

```
class TestArray{  
  
public static void main(String args[]){  
  
    int a[]={1,2,3,4,5};    //declaration, instantiation and initialization  
  
    for(int i=0;i<a.length;i++)  
  
    {  
        System.out.print(a[i]+" ");    a[0]=1,a[1]=2,  
  
    }  
    }}  

```

Output:1

2  
3  
4  
5

### **Multidimensional Array**

data is stored in row and column based index (also known as matrix form).

Syntax,

```
data type[][] arrayRefVar; (or)  
Data Type [][]arrayRefVar; (or)  
Data Type arrayRefVar[][]; (or)  
Data Type []arrayRefVar[];
```

```
int[][] arr=new int[3][3]; //3 row and 3 column
```

- simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
class TestMulti{  
  
public static void main(String args[]){  
  
    int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
```

```
        row  
        ↑  
for(int i=0;i<3;i++)  
  
{  
    column  
    ↗  
    for(int j=0;j<3;j++)  
  
    {  
  
        System.out.print(arr[i][j]+" ");  
    }  
    System.out.println();  
}  
}}
```

```
Output: 1 2 3  
        2 4 5  
        4 4 4
```

## **String**

String is a sequence of characters. But in Java, string is an **object** that represents a sequence of characters.

There are two ways to create String object:

1. By string literal
2. By new keyword

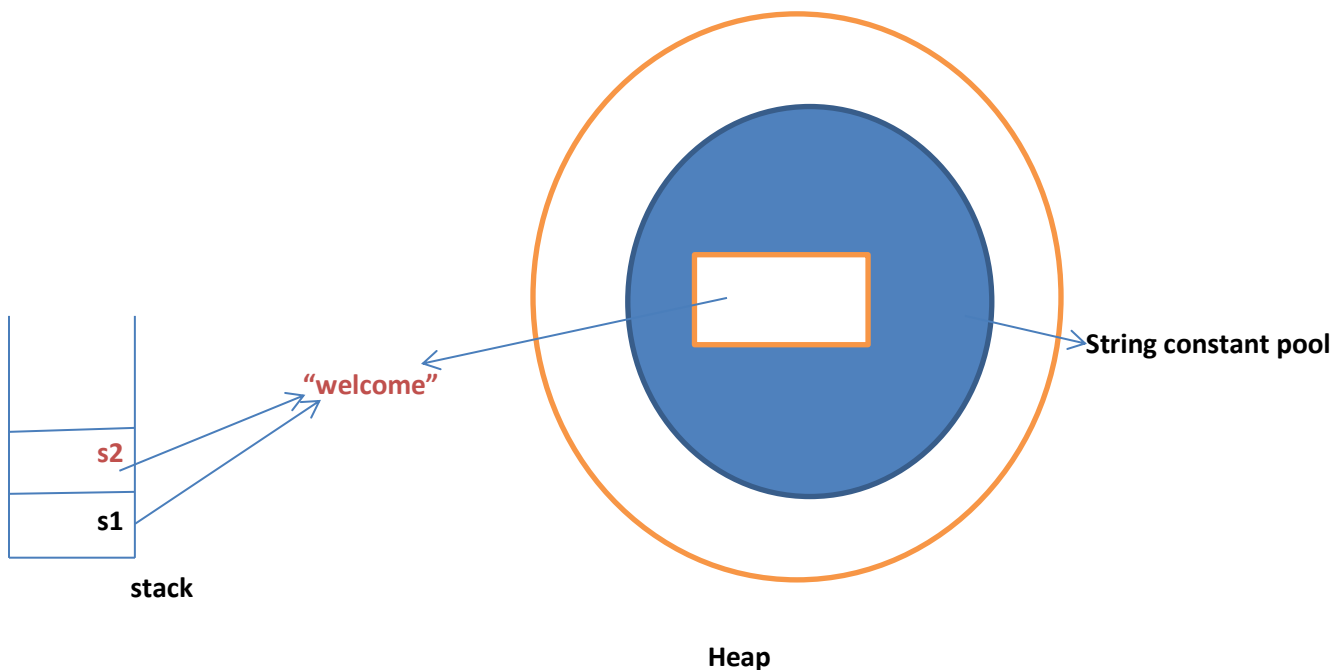
1. String literal

java String literal is created by using double quotes.

Eg: String s="welcome";

(Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.)

```
String s1="Welcome";  
String s2="Welcome";//It doesn't create a new instance
```



- String objects are stored in a special memory area known as the **"String constant pool"**
- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

## 2. By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

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

```
String s1="java";//creating string by java string literal
char ch[]={ 's','t','r','i','n','g','s' };
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:     java  
          strings  
          example

## String Methods:

### String length()

The string length() method returns length of the string.

```
String s="Sachin";
System.out.println(s.length());//6
```

### String charAt()

The string charAt() method returns a character at specified index.

```
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
```

### StringvalueOf()

The string valueOf() method converts given type such as int, long, float, double, boolean, char and char array into string.

```
int a=10;
String s=String.valueOf(a);
System.out.println(s);
System.out.println(s+10);
```

### String equals()

The **java string 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.

```
public boolean equals(Object anotherObject)
```

```
public class EqualsExample{  
public static void main(String args[]){  
    String s1="java";  
    String s2="java";  
    String s3="JAVA";  
    String s4="selenium";  
    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  
}}
```

### String isEmpty()

The **java string isEmpty()** method checks if this string is empty or not. It returns *true*, if length of string is 0 otherwise *false*. In other words, true is returned if string is empty otherwise it returns false.

```
public class IsEmptyExample{  
public static void main(String args[]){  
    String s1="";  
    String s2="javat";  
  
    System.out.println(s1.isEmpty()); //true  
    System.out.println(s2.isEmpty()); //false  
}}
```

### String equalsIgnoreCase()

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string. It is like equals() method but doesn't check case. If any character is not matched, it returns false otherwise it returns true.

```
public class EqualsIgnoreCaseExample{
```

```

public static void main(String args[]){
String s1="java";
String s2="java";
String s3="JAVA";
String s4="Selenium";
System.out.println(s1.equalsIgnoreCase(s2));//true because content and case both are
same
System.out.println(s1.equalsIgnoreCase(s3));//true because case is ignored
System.out.println(s1.equalsIgnoreCase(s4));//false because content is not same
}}

```

## String Buffer and String Builder

Java provides three classes to represent a sequence of characters: String, StringBuffer, and StringBuilder. The String class is an **immutable(It changes value)** class whereas StringBuffer and StringBuilder classes are mutable.

### StringBuffer class

- StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.
- StringBuffer is *less efficient* than StringBuilder.

```

class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"java");//now original string is changed
sb.replace(1,3,"java");// Hjavao
sb.delete(1,3); // Hlo
sb.reverse(); //olleH
System.out.println(sb);//prints Hello Java
} }      Hjavaello

```

### StringBuilder

- - StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
  - **StringBuilder is *more efficient* than StringBuffer.**

## ➔ Methods of StringBuilder and String Buffer

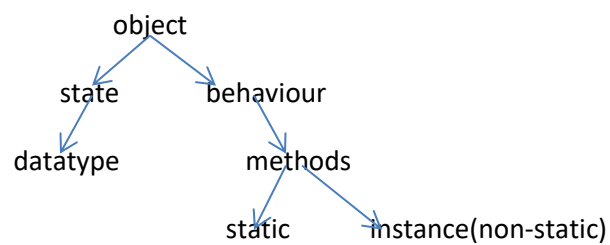
1. `append()`
2. `insert()`
3. `replace()`
4. `delete()`
5. `reverse()`

## METHODS

Class : blue print of object

Variables:

- Local variable
- Static variable
- Instance variable



- static- invoked by using class
- Instance/Non static- invoked by using object

method syntax:

```
access modifier  non-access modifier  return type  methodName (Parameter list)
{
    // method body
}
```

- Access modifier
  - \* To set visibility
  - \* eg: public ,protected, private, default.
- Non access modifier
  - \* To set restriction



- \* eg: static, abstract, final
- Return type
  - \* returned value
  - \* void, int, char
- Static method- non access modifier- static
- Instance method- No non access modifier

### Rules of method

- Name must start with lower case letter
- If it contains multiple words then the first word will have lower case letter and remaining will start with upper case letter
- Eg : getData

```
public class Addition{
    public static int addNumbers()
    {
        int a=10, b=20;
        int sum=a+b;
        //System.out.println(sum);
        return sum;
    }
    public static void main(String args[]){
        int c= Addition.addNumber();
    }
}
```

- Method call – **classname.methodname();** //invoke static method

### Parameterized static method

- Method take an argument from the main and passes a value to the main

```
Class AdditionParameter
{
    public static void add(int a, int b)
    {
        Int c=a+b;
        System.out.println(c);
    }
    Public static void addFloat(float p,float q){
    float m=p+q;
    Sop(m);
    }
    public static void main(String args[]){
        AdditionParameter.add(15,20);
        AdditionParameter.addFloat(10.5f,20.5f);
    }
}
```

### Return Type

- It returns the data from a method to main method
- The same variable name can also be used in the main method to hold the return value
- The return type needs to be changed accordingly.

```
Class Returntype{
    public static int add()
    {
        int a=10;
        int b=20;
        int c=a+b;
        return(c);
    }
    public static void main(String args[])
    {
        int r=Returntype.add();
        System.out.println(r);
    }
}
```

## Method Overloading

- If a class have multiple method with same method name and different arguments is called method overloading.
- Arguments changed in 2 type
  - > By changing no.of arguments
  - > By changing datatype
- Method overloading takes place in static and instance method
- Use of methgod overloading: To increase readability of program

```
class OverMethod
{
Public int add(int a,int b) //instance method
{
int c=a+b;
    //System.out.println(c);

return(c);
}

Public float add(float m, float n) //instance method
{
float p=m+n;

//System.out.println(p);
return(p);
}
public static void main(String args[])
{
OverMethod ob=new OverMethod();

//OverMethod ob1=new Overmethod(5.0f,5.0f);

//System.out.println(ob.add);

//System.out.println(ob1.add);

int q=ob.add(10,20);

float s=ob.add(10.0f,5.0f);

System.out.println(q);
```

```
System.out.println(s);  
  
}  
  
}
```

### Instance methods

#### Syntax:

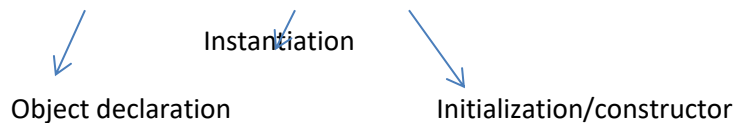
```
Access modifier  return type  methodName(Parameter list)  
{  
  
    //method body  
}
```

❖ Invoked by using object

#### Object creation

1. Object declaration
2. Object instantiation
3. Object initialization

**Classname** **Objectname**=new **classname**();



- Instantiation: It refers to the memory allocation for object
- Initialization: Constructor stores a copy of argument in new
- Instance method is invoked as : **Objectname.methodName();**

### Constructor

- ❖ Constructor is construct an object
- ❖ It is used to initialize a object
- ❖ The constructor has same name as class name

- ❖ Constructor doesn't have return type.
- ❖ Constructor 2 types:
  - ◆ Default Constructor :- constructor with no arguments.
  - ◆ Parameterized Constructor :- constructor with arguments.

#### Syntax for default constructor

```
Access modifier className()
{
}

```

#### Syntax for parameterized constructor

```
Access modifier classname(arguments/parameterlist)
{
}

```

#### Scanner class

- Entering values from keyboard
- It is achieved by making use of an inbuilt class is called scanner class.
- **Import java.util.Scanner**
- Syntax,  
**Scanner s=new Scanner(System.in);**

Here instead of int different data types can also be used

- Float e=nextFloat();
- Double c=nextDouble();
- String e=s.next  
OR  
String e=s.nextLine();

## "This" Keyword

- This is a reference variable that refers to the Current class object
- `this.instance variable;` → to refer instance variable of current class, the instance variable can be used as a Local variable.
- `this.methodname();` → to invoke current class method
- `this();` → to invoke current class constructor
- This must be included as the first statement whenever it is used.

## Inheritance

### Introduction

#### Object Oriented Programing

- Object- (state, behavior)
- Methods-(behavior)
  - Static
  - Instance
- Instance
  - instance variable
  - instance method
  - (syntax)
- OOP's Concept
  - 4 types
    1. Inheritance
    2. Polymorphism
    3. Encapsulation
    4. Abstraction

- **Inheritance**

Inheritance is a mechanism in which one class acquires the property of another class.

We can reuse the fields(variables) and methods of existing class(parent class).

It is **parent - child** relationship(**IS-A relationship**), child class inherit the features of parent class.

**Use of inheritance : - code reusability and method overriding.**

**Syntax:**

```
Class  child class  extends  parent class
{
}
```

- **extends** keyword is used by child class to inherit the features of parent class.
- Parent class is also known as base class or super class.
- Child class is also known as derived class, extended class or sub class.

Public Class A

```
{
Public void test()          //instance method (jvm create default constructor)
{
    System.out.println("Parent class");
}
}
```

Public Class B extends A

```
{
    public void print()      //another instance method
    {
```

```
System.out.println("Child class");  
}
```

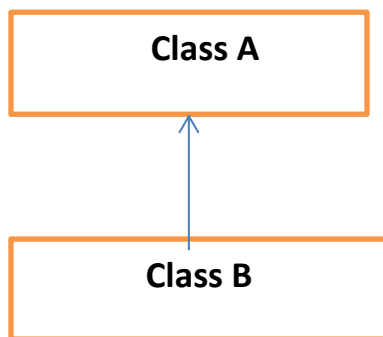
```
public static void main(String args[])  
{  
    B ob=new B(); //object creation  
    ob.test();    //call parent class method  
  
    ob.print();   //call child class method  
  
}  
}
```

Output:

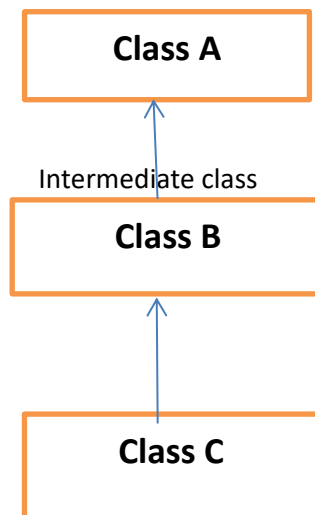
Parent class

Child class

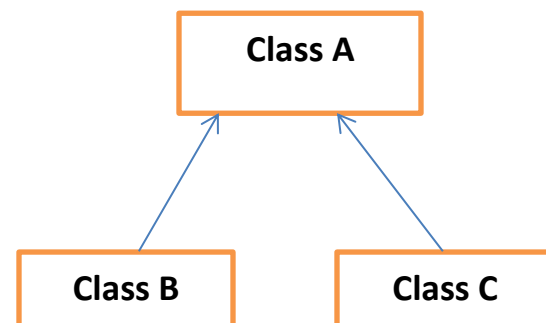
## Types of inheritance



a) Single Inheritance



b) Multilevel Inheritance



c) Hierarchical



- **Single inheritance**

```
class A
{
}
class B extends A
{
}
```

- **Multilevel inheritance**

```
class A
{
}
class B extends A
{
}
class C extends B
{
}
```

- **Hierarchical inheritance**

```
Class A
{
}

Class B extends A
{
}

class C extends A
{
}
```

## Super Keyword

- It is a reference variable used to **refer immediate parent class object.**

### Usage of super keyword

- Super can be used for invoke parent class instance variable: `super.variableName;`
- Super can be used for invoke parent class method: `super.methodName();`
- Super can be used for invoke parent class constructor: `super();`

### Parent class:

```
public class ABC {  
    int a=10;  
    int b=20;  
    public void print() {  
        System.out.println("Parent class");  
    }  
}
```

### Child class:

```
public class XYZ extends ABC {  
  
    public void show() {  
  
        super.print(); // invoke parent class method  
        Sop(super.a);   //invoke parent class variables  
        Sop(super.b);  
  
        System.out.println("Child class");  
    }  
    public static void main(String args[]) {  
  
        XYZ ob=new XYZ();  
        ob.show();  
        //System.out.println(ob.a);  
        //System.out.println(ob.b);    // invoke parent class variable in  
main method  
    }  
}
```

## Scanner class

- **Entering value from keyboard**
- It is achieved by making use of an inbuilt class called scanner class
- **import java.util.Scanner;** , must be included outside the class
- Syntax :,

```
Scanner s=new Scanner(System.in);  
Sop(" ")  
int a=s.nextInt();
```

- Here instead of int different data types can also be used
- float f=s.nextFloat();
- double d=s.nextDouble();
- String b=s.nextLine();

## Access modifiers

Access Modifier	Within class	Within Package	Outside Package by subclass only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

## Polymorphism

Is a concept by which we can perform a *single action in different ways*.

### ➔ Compiletime polymorphism

Eg: Method Overloading

### ➔ Runtime polymorphism

Eg: method Overriding

## Method Overriding (Runtime polymorphism)

- The method must have the same name as the parent class method.
- Method must have the same parameters.
- Method overriding is used for runtime polymorphism
- There must be IS-A relationship.
- Static method can't be over ridden since its invoked by class not by object
- Private and final method can't be over ridden
- Java main method can't be over ridden.

Public Class A

```
{  
  
    public void test()          //instance method  
    {  
        System.out.println("Parent class");  
    }  
}
```

Public Class B extends A

```
{  
    public void test()    //same instance method  
    {  
  
        System.out.println("Child class");  
  
    }  
    public static void main(String args[])  
    {  
        B ob=new B();  
        ob.test();  
  
    }  
}
```

Output: Child class

## Aggregation

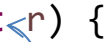
If a class have an entity reference of another class, it is known as Aggregation.

Aggregation represents **HAS-A relationship**.

```
public class Operation {  
    public int square(int r) {  
        return (r*r);  
    }  
}
```

```
import p1.Operation;
```

```
public class Circle {
```



```

double pi=3.14;
public double area(int radius) {
    Operation op=new Operation();    //create operation
class object
    int s=op.square(radius);
    System.out.println(s);
    return(pi*s);
}

public static void main(String[] args) {
    Circle c=new Circle();           //create circle class
object
    double result=c.area(5);
    System.out.println(result);
}
}

```

## Encapsulation

- It is the mechanism of **wrapping code (methods) and data (variables) together as a single unit**. In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.
- Declare all its variable as **private**
- Use **setter** and **getter methods** to set (modify) and get (view) the data in it.

**Advantage:-**

**Data hiding:** Other class will not be able to access the data through private variables and hence it act as protective shield.

Package1:

```
public class Student {  
  
    private String name;  
    private int age;          //instance variable  
  
    public void getName() {    //method is used to get value  
        //age=age*10;  
        System.out.println(name);  
        System.out.println(age);  
    }  
    public void setName(String name,int age) { //method is used to  
set value  
  
        this.name=name;  
        this.age=age;  
    }  
}
```

Package2:

```
public class Test {  
  
    public static void main(String[] args) {  
  
        Student s=new Student();    //create student class object  
        s.setName("ABC",5);          //call student class method  
        s.getName(); //call student class method2  
    }  
}
```

### Final keyword

- Restrict the use of editing the data
- Final variable :Can't be change
- Final method : Can't be override, it result compile time error
- Final class : can't be extended by inheritance

## Abstraction

- It is the process of hiding the internal details showing only the functionality to user.

### 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 provides 0-100% data hiding based on the methods used within the class.
- The definition of all the abstract methods must be provided in the extended class which is not abstract

Syntax:

```
public abstract class classname
{
    access modifier abstract returntype methodname();    // Abstract method
}
```

```
public abstract class Test {           //abstract class
```



```

    public abstract void print();    //Abstract method

    public void show()              // Non-abstract method
    {
        System.out.println("Non Abstract method");
    }
}

```

```
package abstraction;
```

```

public class Sample extends Test {

    public void print()            //definition of abstract method
    {
        System.out.println("Abstract method");
    }

    public static void main(String args[])
    {
        Sample s=new Sample();
        s.print();
        s.show();
    }

}

```

Output:

```

    Abstract method
    Non Abstract method

```

## Interface

Class →blue print of object

**Interface – >blue print of class**

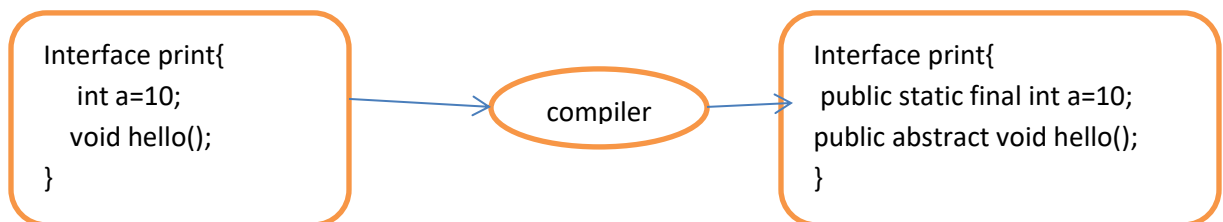
Class has **method** and **variables**

- Method in interface – **public abstract return type method name();**
- Variables in interface – **public static final datatype variable name;**
- It can't be instantiated
- It doesn't have constructor
- Interface is 100% data hiding(abstraction)
- It support Multiple and Hybrid Inheritance

Syntax,

```
interface <interface name>{  
  
    // declare constant fields → public static final int a=10;  
  
    // declare methods that abstract → public abstract void print();  
}
```

By default,



**Class**----extends----->**class**

**Class** ----implements--->**interface**

**Interface**---extends----->**interface**

```
public interface Test {
```

```

    int a=10; //variable declaration
    public abstract void hello(); //abstract method
}

public class Sample implements Test{

    public static void main(String[] args) {
        // TODO Auto-generated method stub

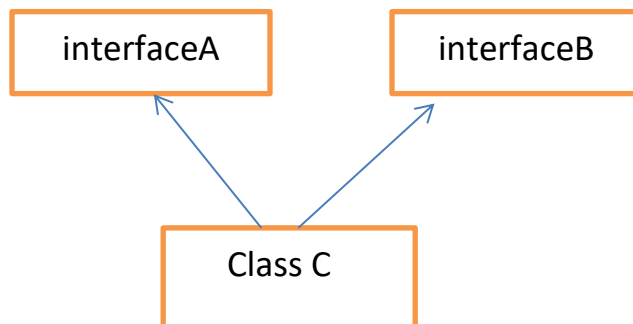
        Sample ob=new Sample();

        ob.hello();           //call interface method
        System.out.println(ob.a); //invoke variable
    }

    public void hello() {
        System.out.println("Abstract method of interface");
    }
}

```

### Multiple Inheritance



```

interface Printable{ // interface 1
void print();
}
interface Showable extends Printable { // interface 2
void show();
}
class Example implements Showable { //

public void print(){
    System.out.println("Hello");
}
}

```

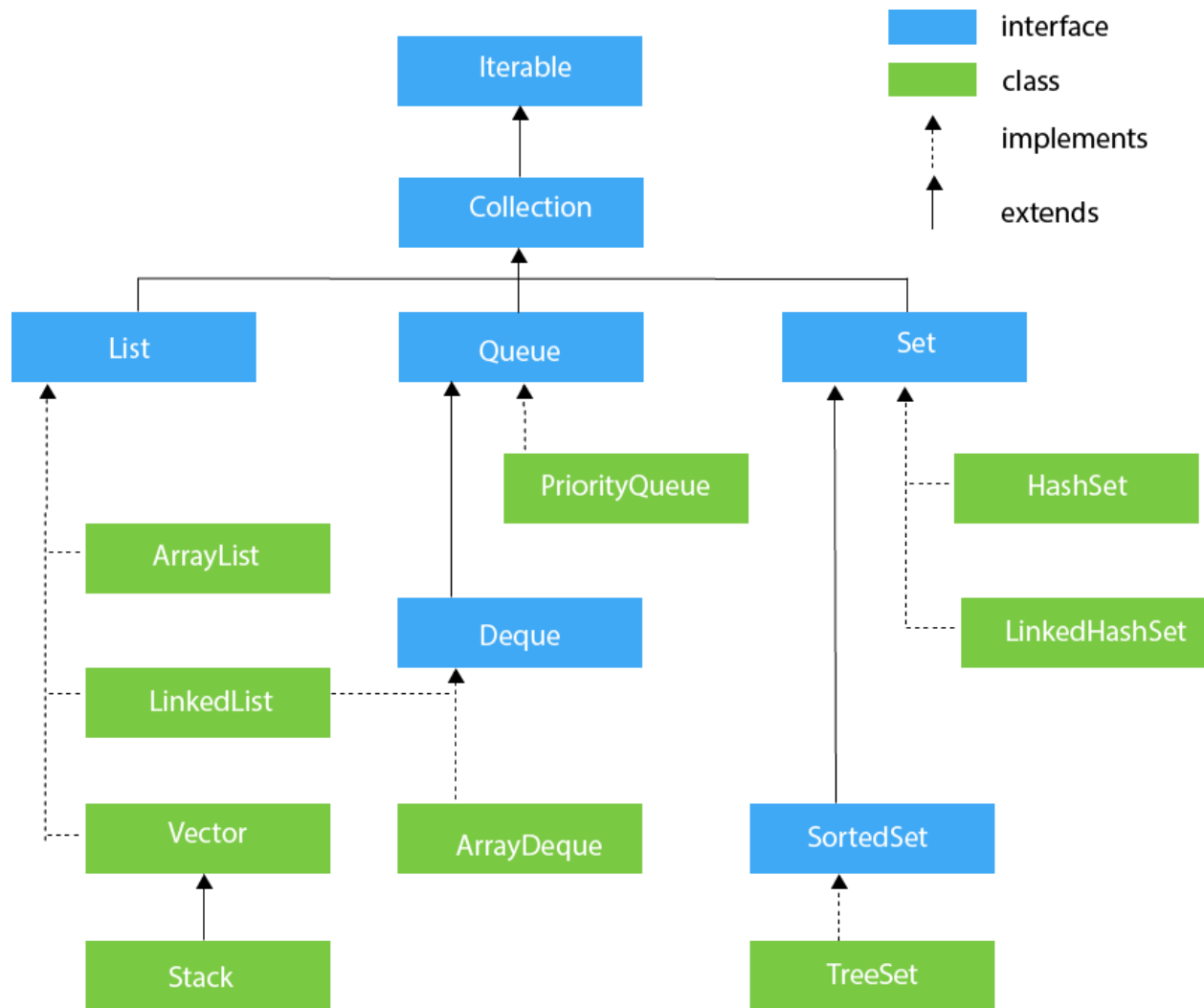
```
}  
public void show(){  
System.out.println("Welcome");  
}  
  
public static void main(String args[]){  
Example obj = new Example();  
obj.print(); //call interface1 method  
obj.show(); //call interface2 method  
}  
}
```

## **Collection**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

A Collection represents a single unit of objects, i.e., a group.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](#), [Vector](#), [LinkedList](#), [PriorityQueue](#), [HashSet](#), [LinkedHashSet](#), [TreeSet](#)).



## ArrayList

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.

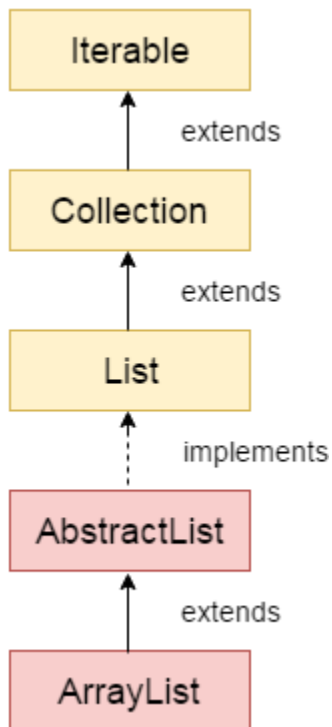
**ArrayList al =new ArrayList();** //create old non-generic arrayList. Type of array should not be mentioned.(JDK 1.5 feature)

**ArrayList<String> al =new ArrayList<String>();** //Create generic arraylist. Type of array should be mentioned.

**List<Integer>a1=new ArrayList<Integer>();**

**List<Integer>a1=new ArrayList();**

//There are JDK 1.7 features



## Methods in ArrayList

- `a1.add(object);` //adding object in arraylist
- `a1.remove(index);` //remove element in the index position
- `a1.size();` //print the arraylist size
- `a1.addAll(collection);` // It is used to append all the elements in the specified collection.
- `a1.removeAll();` // It is used to remove all the elements from the list.
- `boolean contains(Object o)` // It returns true if the list contains the specified element
- `a1.get(index);` //Retrive the element from collection

```
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>(); //Creating arraylist
    list.add("Ravi");    //Adding object in arraylist
    list.add("Vijay");
    list.add("Ravi");
    list.add("Ajay");
        //Invoking arraylist object
    System.out.println(list);
}
}
```

Output: [Ravi,Vijay,Ravi,Ajay]

```
public class p1 {

public static void main(String[] args) {

ArrayList<Integer>a1=new ArrayList<Integer>();

a1.add(10);
a1.add(20);
a1.add(30);
```

```

a1.add(40);
a1.add(50);
System.out.println("Added values are:"+a1);
a1.remove(2); //remove element in the ArrayList
System.out.println("After remove :"+a1);
int length=a1.size(); //calculate the size of the ArrayList
System.out.println("Length="+length);

ArrayList<Integer>a2=new ArrayList<Integer>();
a2.addAll(a1); // used to append all the elements in the specified collection

System.out.println("Values of the a2 ArrayList is:"+a2);

a1.removeAll(a1); // to remove all the elements from the list.
System.out.println("a1 values :"+a1);
boolean b=a2.contains(20);// returns true if the list contains the specified
element
System.out.println("Contains:"+b);

}}

```

Output:

```

After remove :[10, 20, 40, 50]
Length=4
Values of the a2 ArrayList is :[10, 20, 40, 50]
a1 values :[]
Contains:true

```

## Iterator Interface

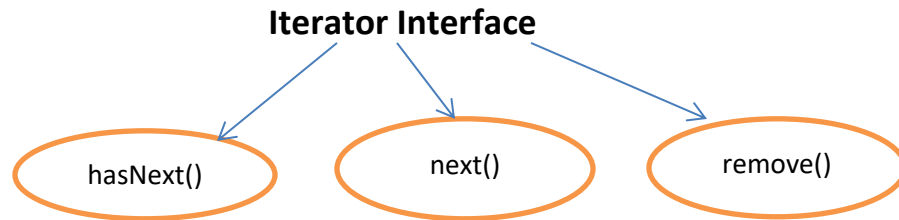
Iterator are used in Collection framework in java to retrieve elements one by one. There are three iterators.

Syntax:

```
Iterator iterator()
```

**Return Value:** This method returns an **iterator** over the element in this list in proper sequence.





**hasNext():** it is used to return true if the given list iterator contains more number of elements during traversing the given list in the forward direction.

**Next():** Return the next element in the given list.

**Remove():** remove the last element in the given list.

*//Traversing list through iterator*

```
Iterator itr=a2.iterator();
while(itr.hasNext()){
    System.out.println(itr.next());
}
```

## ArrayList elements using the **for-each loop**

The syntax of the Java **for-each** loop is:

```
for(dataType item : array) {
    ...
}
```

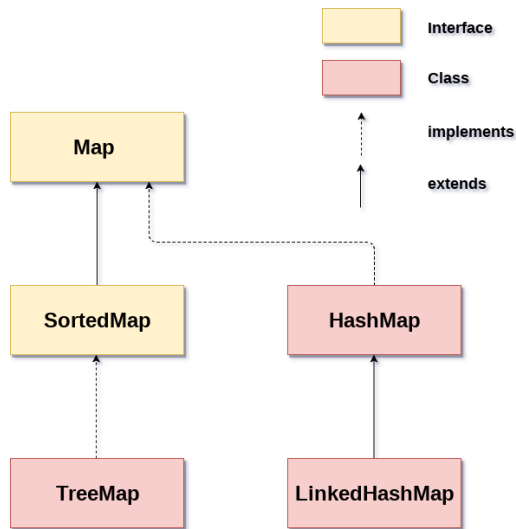
Here,

- **array** - an array or a collection
- **item** - each item of array/collection is assigned to this variable
- **dataType** - the data type of the array/collection

```
// Traversing list through for-each loop
for(String obj:al) {
    System.out.println(obj);
}
```

## Map

- A map contains values on the basis of key, i.e. **key and value pair**.
- Each key and value pair is known as an entry.
- A Map contains unique keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.



- A Map doesn't allow duplicate keys, but you can have duplicate values.
- HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.
- HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

## Class

**HashMap** : HashMap is the implementation of Map, but it doesn't maintain any order.

**LinkedHashMap** : LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.

**TreeMap** : TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

## **Exception Handling**

**Exception:** It is an unwanted condition that disrupts the normal flow of the program.

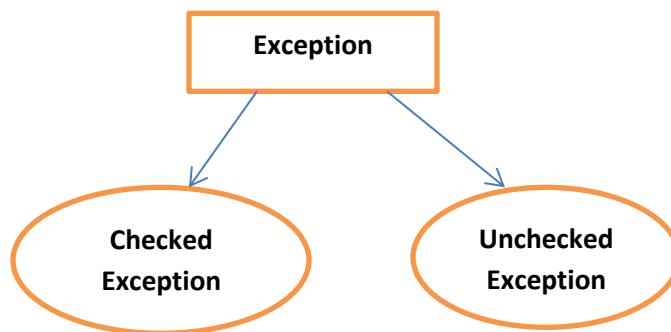
- An exception is an error event that can happen during the execution of a program and disrupts its normal flow. The exception can arise from different kinds of situations such as wrong data entered by the user, hardware failure, network connection failure, etc.

**It is a class**

**Exception Handling:** one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

Advantage: **to maintain the normal flow of the application**

**Exceptions are 2 Type: Checked Exception And Unchecked Exception**



➔ **Checked** : it is compile time exception, program doesn't run due to external errors. It can be corrected at compile time

Eg: Input-Output exception

File not found-->file can't be read path is different

SQL exception -> error in data base

➔ **Unchecked** : It is runtime exception, program doesn't run due to error, created by the programmer. It cannot be corrected in compile time as issue occur in run time. It is corrected at runtime

**Eg:** Arithmetic exception

Null pointer exception

Array out of bound

**There are 5 keywords which are used in handling exceptions in Java.**

**try:**

- The "try" keyword is used to specify a block where we **should place exception code**.
- The try block must be followed by either catch or finally. It means, we can't use try block alone.
- try{  
    }  
}

**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 must be used within the method
- catch(ClassName objname) {  
    //stmt  
}

**finally:**

- The "finally" block is used to execute the important 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 doesn't throw an exception.
- It specifies that there may occur an exception in the method. It is always used with method signature.

```
class JavaExcep{
    public void test()
    {
        try{
            int a=10/0;    //exception occure
        }catch(Exception e){ //catch Exception
            System.out.println(e);
            System.out.println("Exception Handilled");
        }
    }
    public static void main(String args[]){
        JavaExcep ob=new JavaExcep();
        ob.test();
        System.out.println("Rest of the code");
    }
}
```

### Multiple catch block

```
class MultipleCatch{

    public static void main(String args[]){
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
```

```

System.out.println("Arithmetic Exception occure");

    }
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("ArrayIndexOutOfBoundsException occure");
}
catch(Exception e){

System.out.println("ParentException occure");
}
System.out.println("Rest of the code");
}
}

```

Output:

```

Arithmetic Exception Occure
Rest of the code

```

### throw:

- The "throw" keyword is used to throw an exception.
- The "throw" keyword is used to **explicitly throw an exception**.
- We can throw unchecked exception.
- The throw keyword is mainly used to throw **custom exception**.

Syntax,

**throw exception;**

eg: `throw new ArithmeticException("msg");`

```

public class TestThrow {
    public static void age(int age) {
        if (age<18){

            throw new ArithmeticException("Not valid"); //throw anexception
        }
        else
            System.out.println("Eligible for vate");
    }

    public static void main(String args[]) {
        TestThrow.age(15);
    }
}

```

```

        System.out.println("rest of the code");
    }
}

```

## Throws

- throws keyword is used to **declare an exception**.
- It is always used with method signature.
- It specifies that there may occur an exception in the method.
- It gives an information to the programmer that there may occur an exception.
- mainly used to handle the checked exceptions.

Syntax,

```

return_type method_name() throws exception_class_name{

    //method code
}

```

- **public class** Propagation {

**void** print3() {

int data=50/0; //exception hit  
        //throw new IOException("Error");

    }

**void** print2() {

        print3();

    }

**void** print1() {

**try** {  
            print2();

        }

**catch**(Exception e) {

```

        System.out.println("Exception handled");
    }
}
public static void main(String[] args) {

    Propagation ob=new Propagation();
    ob.print1();
    System.out.println("Rest of the code");

}}

```

### Custom Exception

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

```

public class Licenceexception extends Exception{

    public Licenceexception(String s) {
        super(s);
    }
}

public class Test {

    public void Chinnu(int age)throws Licenceexception {

        if(age<18) {
            throw new Licenceexception("Not Valid");
        }
        else {
            System.out.println("Eligible");
        }
    }

    public static void main(String args[]) {

        Test t=new Test();
    }
}

```



```

    try {
        t.Chinnu(10);
    } catch (Exception e) {
        System.out.println("Exception occure:" + e);
    }
}

```

- Final is a keyword

**final** class can't be inherited, **final** method can't be overridden and **final** variable value can't be changed.

- Finally is a block

It is used to place important code, it will be executed whether exception is handled or not.

- Finalize is a method.

**Finalize** is used to perform clean up processing just before object is garbage collected.

The **finalize()** method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection, so as to perform clean-up activity.

### Difference between List & Set

	List	Set
1	The list implementation allows us to add the same or duplicate elements.	The set implementation doesn't allow us to add the same or duplicate elements.
2	The insertion order is maintained by the List	It doesn't maintain the insertion order of elements.
3	List allows us to add any number of null values.	Set allows us to add at least one null value in it.
4	The List implementation classes are LinkedList and ArrayList.	The Set implementation classes are TreeSet, HashSet and LinkedHashSet.
5	We can get the element of a specified index from the list using the get() method.	We cannot find the element from the Set based on the index because it doesn't provide any get method().

### Difference between ArrayList and LinkedList

ArrayList	LinkedList
ArrayList internally uses a <b>dynamic array</b> to store the elements.	LinkedList internally uses a <b>doubly linked list</b> to store the elements.
Manipulation with ArrayList is <b>slow</b> because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is <b>faster</b> than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
An ArrayList class can <b>act as a list</b> only because it implements List only	LinkedList class can <b>act as a list and queue</b> both because it implements List and Deque interfaces.
ArrayList is <b>better for storing and accessing</b> data.	LinkedList is <b>better for manipulating</b> data.

