**JAVA Class**

Java file 🡪 javac 🡪 Java Byte File 🡪 JRE(jvm) 🡪 Oputput

Javac and JRE are the part of JDK.

Java Type file is also called as .class file, this is O.S independent file.

JRE is also a O.S independent.

Example Java Program:

class MainClass{

public static void main(String arg[]){

System.out.println("Hello Vishnu");

}

}

**Unary Operator:** used to increment or decrement values

-- is for decrement operation

++ is for increment operation

Primitive Data types: int, float, long, short, byte, boolean, double, char

Non-Primitive Reference types: Class, Object and String.

**Relations Operator:** used to check the relationship between the two operands, it usually gives the boolean value true or false.

>, <, >=, <=, ==, !=

**Logical Operator:** This is used to check the logical relationship of two operators.

&& (AND) , || (OR), ! (NOT)

class RelationalOperator{

public static void main(String arg[]){

int age = 2;

if(age<15)

{

System.out.println("Give a Baloon");

}

else

{

System.out.println("Allow to enter");

}

}

}

**Condition Statement:** If is a condition statement used to check the condition. It usually give the boolean value true or false, if condition is true it executes the if statement.

If 🡪 true it executes if block.

In if else condition: if is true it executes if block, else it executes the else block.

class NoDivBy5And7{

public static void main(String arg[]){

int n = 35;

if(n%5 ==0 && n%7 ==0)

{

System.out.println("Given no div by 5 and 7");

}

else

{

System.out.println("Given no not div by 5 and 7");

}

}

}

**Nested if:** if block has one more if blocks is called as nested if.

class FindBigIn3{

public static void main(String arg[]){

int a = 10;

int b = 20;

int c = 30;

if(a>=b && a>=c)

{

System.out.println("A is Big");

}

else if(b>=a && b>=c)

{

System.out.println("B is Big");

}

else

{

System.out.println("C is Big");

}

}

}

**Loops:** Used to perform repeated operations in the program. Example: for, for each, while, do while.

1. **Print 10 number using for loop.**

class ForLoop{

public static void main(String arg[]){

int i;

for(i=1;i<=10;i++)

{

System.out.println(i);

}

}

}

1. **Print number between 24 and 38**

class NoBetween24And38{

public static void main(String arg[]){

for(int i = 24; i<=38; i++)

{

System.out.println(i);

}

}

}

1. **Print odd number between 24 and 38**

class OddNoBetween24And38{

public static void main(String arg[]){

for(int i=24; i<=38; i++)

{

if(i%2 != 0)

{

System.out.println(i);

}

}

}

}

1. **Print odd or even in 1 to 20**

class OddREvenIn1To20{

public static void main(String arg[]){

for(int i=0; i<=20; i++)

{

if(i%2 == 0)

{

System.out.println(i+" is Even");

}

else

{

System.out.println(i+" is Odd");

}

}

}

}

1. **Print Even and Divisible by 7 in 1 to 100**

class EvenAndDivBy7{

public static void main(String arg[]){

for(int i=1; i<=100; i++)

{

if(i%2 == 0 && i%7 ==0)

{

System.out.println(i);

}

}

}

}

1. **Print 2 \* 1 table**

class TwosTable{

public static void main(String arg[]){

for(int i=1; i<=10; i++)

{

System.out.println("2 \* "+i+" = "+2\*i);

}

}

}

1. **Print 1234 in 4 lines**

class Print1234{

public static void main(String arg[]){

for(int i=1; i<=4; i++)

{

for(int j=1; j<=4; j++)

{

System.out.print(j +" ");

}

System.out.println("");

}

}

}

1. **Print \* triangle**

class StarTriPrint{

public static void main(String arg[]){

int line = 4;

int star = 1;

for(int i=1; i<=line; i++)

{

for(int j=1; j<=star; j++)

{

System.out.print("\*");

}

System.out.println("");

star++;

}

}

}

**While Loop:**

class WhileLoop{

public static void main(String arg[]){

int i=1;

while(i<=10)

{

System.out.println(i);

i++;

}

}

}

**Do While Loop:**

class DoWhileLoop{

public static void main(String arg[]){

int i =1;

do{

System.out.println(i);

i++;

}while(i<=5);

}

}

**Switch Case:**

class SwitchCase{

public static void main(String arg[]){

int n = 100;

switch(100){

case 100: System.out.println("100");

break;

case 80: System.out.println("80");

break;

case 60: System.out.println("60");

break;

default: System.out.println("Invalid Input");

break;

}

}

}

1. **Factorial of a given number**

class FactorialOfGivenNo{

public static void main(String arg[]){

int n = 5;

int f = 1;

for(int i=n; i>=1; i--)

{

f = f \* i;

}

System.out.println(f);

}

}

**Methods:**

\*Methods are used to perform some functionality

\*Methods may contain group of statements.

\*Methods you can call N number of times.

\*Values passed from caller method is called as Arguments.

\*Values received by the called method is called as Parameters.

**Example:**

public static void main(String arg[]){

vmethod(x); 🡪 argument

int vmethod(i) 🡪 parameter

{

System.out.println(i\*i);

}

\*Method can have only one return statement logically.

\*Method return type is “void” it should not return any values.

\*Method return type is “Not void” is should return used return type value.

\*Methods can be called from any methods.

Note: Method should not be written inside the method.

1. **Adding two values using method call.**

class KuduMethod{

public static void main(String arg[]){

kudu(4,5);

}

static void kudu(int i, int j)

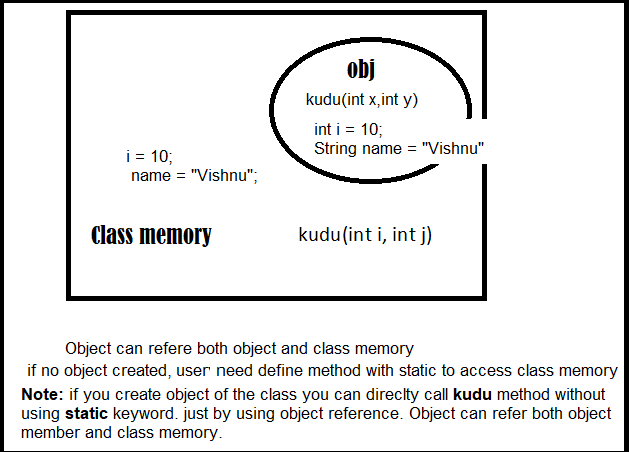
{

int Sum = i + j;

System.out.println(Sum);

}

}



**Example for above image program:**

class KuduWithoutStaticKeywordInMethod{

public static void main(String arg[]){

int i = 10;

String name = "Vishnu"

Test2 t = new Test2();

t.kudu(10,20);

}

void kudu(int x,int y)

{

System.out.println(x+y);

}

}

1. **Finding average of 3 numbers using method call.**

class AverageOf3Subjects{

public static void main(String arg[]){

getAvg(38.00,60.00,98.97);

}

static void getAvg(double i, double j, double k)

{

double avg = (i+j+k)/3;

System.out.println(avg);

}

}

1. **Fibonacci number printing using method calling.**

class FibonacciNumberPrint{

public static void main(String arg[]){

int a = 0;

int b = 1;

int c = 0;

int n = 20;

fib(n,a,b,c);

}

static void fib(int n, int a, int b, int c)

{

for(int i = 1; i<=n; i++)

{

c = a + b;

b = c;

a = b;

System.out.println(c);

}

}

}

1. **Return largest by calling methods**

class ReturnLargestByMethodCall{

public static void main(String arg[]){

int x;

int a = 10;

int b = 20;

int c = 30;

x = larg(a,b,c);

System.out.println("Biggest in given 3 number is: "+x);

}

static int larg(int i, int j, int k)

{

if(i>j && i>k)

{

return i;

}

else if(j>i && j>k)

{

return j;

}

else

{

return k;

}

}

}

1. **Print factorial of a given number using recursion.**

class RecursionFactorial{

public static void main(String arg[]){

int n = 5;

int f = fact(n);

System.out.println(f);

}

static int fact(int n)

{

if(n==0)

{

return 1;

}

return n \* fact(n-1);

}

}

**Class and Object:**

Class contains status and behaviors of the object.

Class is the blue print of the object.

Object is the instance of the class.

Object is created using new keyword.

Any variable of class or type which points to an object is called as reference variable.

Example: Van v = new Van();

Note: File contains multiple class, in that case file should be named with main class name.

Note: Java is architectural neutral language, main method can be anywhere.

class Van{

int wheel;

String name;

void move()

{

System.out.println("Van moving...");

}

void stop()

{

System.out.println("Van Stopped");

}

}

class TwoClassNMethodCall{

public static void main(String arg[]){

Van v = new Van();

int w = v.wheel = 4;

String n = v.name = "Vishnu";

v.move();

v.stop();

System.out.println(w);

System.out.println(n);

}

}

**Calculator Class:**

class CalculatorClass{

void add(int i, int j)

{

System.out.println(i+j);

}

void mul(int i, int j)

{

System.out.println(i\*j);

}

void sub(int i, int j)

{

System.out.println(i-j);

}

void div(double i, double j)

{

System.out.println(i/j);

}

}

class CalTest{

public static void main(String arg[]){

CalculatorClass cal = new CalculatorClass();

cal.add(4,5);

cal.mul(4,5);

cal.sub(100,50);

cal.div(100,3);

}

}

**Variables:**

Local variables

Global variables – a. Static Variable b. Instance Variable

**Example of Variables:**

class Cal{

int i; 🡪 Instance variable

Static int j; 🡪 Static variable

void add()

{

Int x = 2; 🡪 Local variable

}

}

Static Variable: are the one which are declared using keyword static.

Static Variables load only once when class loads first time into the memory.

Instance Variable: are the one which are declared without using keyword static.

**Example of Static and Instance Variables:**

class Branch{

int branch\_count=0;

static int total\_count=0;

void swipe()

{

branch\_count++;

total\_count++;

}

}

class ATMToTestBranchAndTotalCount{

public static void main(String arg[]){

Branch bsk = new Branch();

Branch rjn = new Branch();

Branch hsr = new Branch();

bsk.swipe();

bsk.swipe();

rjn.swipe();

hsr.swipe();

System.out.println("Branch Count :" +bsk.branch\_count);

System.out.println("Branch Count :" +rjn.branch\_count);

System.out.println("Branch Count :" +hsr.branch\_count);

System.out.println("Total Count :"+Branch.total\_count);

}

}

**Inheritance:**

One of the OOPS concept, a class deriving the properties of another class is called as Inheritance.

Inheritance can be achieved by using extends keyword.

Inheritance is also called as Is-A relationship.

Java supports 3 types of Inheritances:

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance

**Examples of inheritance:**

1. **Single Inheritance.**

class Cal{

void add(int i, int j)

{

System.out.println(i+j);

}

void mul(int i, int j)

{

System.out.println(i\*j);

}

}

class SciCal extends Cal{

void sin(double ang)

{

System.out.println("Sin: "+ang);

}

void cos(double ang)

{

System.out.println("Cos: "+ang);

}

}

class CalAndSciCalTest{

public static void main(String arg[]){

SciCal sc = new SciCal();

sc.add(4,5);

sc.mul(4,5);

sc.sin(344.56);

sc.cos(944.56);

}

}

1. **Multilevel Inheritance.**

class Basic{

void sms()

{

System.out.println("Text Message service");

}

}

class Color extends Basic{

void mms()

{

System.out.println("Multimedia Message service");

}

}

class Android extends Color{

void videocall()

{

System.out.println("Video call service");

}

}

class MobileMultilevelInheritance{

public static void main(String arg[]){

Android mob = new Android();

mob.sms();

mob.mms();

mob.videocall();

}

}

1. **Hierarchical Inheritance.**

class Animal{

void walk()

{

System.out.println("Animal which can walk");

}

}

//--------------------------------------------------------------------

class Lion extends Animal{

void eat()

{

System.out.println("Lion Eat Animals");

}

}

class Cab extends Lion{

void play()

{

System.out.println("Cab play with Lion");

}

}

//--------------------------------------------------------------------

class Cow extends Animal{

void eat()

{

System.out.println("Cow Eat Grass");

}

}

class Calf extends Cow{

void play()

{

System.out.println("Calf play with Cow");

}

}

//--------------------------------------------------------------------

class HierarchicalInheritanceWithAnimal{

public static void main(String arg[]){

Cab cb = new Cab();

Calf cf = new Calf();

cb.walk();

cf.walk();

cb.eat();

cf.eat();

cb.play();

cf.play();

}

}

Java does not support Multiple Inheritance for the below reasons:

1. Method calling conflict
2. Constructor conflict [super()]
3. Diamond Problem

**Static and Instance:**

Instance member method can access both Static and Non-Static member of the class directly.

Static member method can only access Static members of the class and can’t access non-static members of the class.

class InstanceAndStatic{

int i=5;

static int j = 10;

// static void add()

// {

// System.out.println(i+j);

// }

void mul()

{

System.out.println(i\*j);

}

}

class CheckInstanceAndStatic{

public static void main(String arg[]){

InstanceAndStatic s = new InstanceAndStatic();

// s.add(); //we can't call this as its calling static method and its using non-static(instance variable)

s.mul();

}

}

**Naming Conventions:**

The names of a class, method and variable should only the characters like: a-z, A-Z, 0-9, $, \_(underscore).

The name should start with alphabet, $, and \_ (underscore) followed by number.

**Example:**

$name 🡪 Right

\_name 🡪 Right

Name 🡪 Right

2Name 🡪 Wrong

**Class Name:** first word of the class name should be in upper case. If class have more than one word, the second word first char should be in upper case. The words in class name should be separated by using underscore(\_).

Example:

ClassName

Class\_Name

**Methods or Variables:** Should always start with lower case letter, the next word of the variable should be in upper case.

**Example:**

myMedhod()

strName

**Constants:** should be All in caps.

**Example:**

PI

RATE

**Default Values:** Compiler will assign a default value to the global variable which are not initialized.

**Example:**

**Primitive:** char🡪 space, int 🡪 0, float 🡪 0.0, double 🡪 0.0, short 🡪 0, boolean 🡪 false.

**Non-Primitive:** by default it set null value.

**Note:** Compiler will not assign any value to the local variable which are not initialized(program blow...)

class LocalVariable{

// int i; //if variable declared here it give '0' as output.

void m()

{

int i; // it give error: variable i might not have been initialized.

System.out.println(i);

}

}

class LocalVariableWithNoInitialization{

public static void main(String arg[]){

LocalVariable v = new LocalVariable();

v.m();

}

}

**Constructor:** are the special methods which have the same name as the class name without any return type.

**Example:**

class Parent{

Parent()

{

System.out.println("Zero parameter constructor");

}

}

class ConstructorClass{

public static void main(String arg[]){

new Parent();

}

}

**Default Constructor:** if class not explicitly declared any constructor, compiler internally writes one zero parameter constructor which is called as default constructor.

**Constructor Overloading:** Single class having multiple constructor with different type of parameters is constructor overloading.

class Parent{

Parent()

{

System.out.println("Zero parameter constructor");

}

Parent(int n)

{

System.out.println("Int constructor");

}

Parent(String name)

{

System.out.println("String constructor");

}

Parent(String name,int age)

{

System.out.println("String and Int constructor");

}

}

class ConstructorOverloading{

public static void main(String arg[]){

new Parent();

new Parent(5);

new Parent("Vishnu");

new Parent("Vishnu",17);

}

}

**super()** – Used to call the Parent class constructor.

super statement should be the first in the constructor.

If we don’t write super inside a constructor, compiler will write super which always points to parent class constructor.

class Parent{

int i = 10;

Parent(){

System.out.println("Parent Class Constructor");

}

Parent(int x,int y)

{

System.out.println(x+y);

}

}

class Child extends Parent{

int i = 20;

Child(){

**// super(10,20);**

System.out.println("Child class Constructor");

}

void m()

{

System.out.println(i); //it is printing present class i value.

**System.out.println(super.i); //it is printing parent class i value, used super.i**

}

}

class SuperConstructorTest{

public static void main(String arg[]){

Child c = new Child();

c.m();

}

}

**//output after Uncomment this line: super(10,20);**

//30

//Child class Constructor

//20

//10

**//output after comment this line: super(10,20);**

//Parent Class Constructor

//Child class Constructor

//20

//10

**//Note:If we are not calling super explicitly compiler automatically call super by default.**

**this()** – Used to call the current class constructor.

**this** – is the keyword used when there is a conflict between the *global and local variable*. We make use of **“this”** to indicate global variable.

**Example:**

class Person{

String name;

int age;

Person(String name, int age)

{

this.name = name;

this.age = age;

}

void m()

{

System.out.println("Name is : "+name);

System.out.println("Age is : "+age);

}

}

class ThisConstructor{

public static void main(String arg[]){

Person p = new Person("Vishnu",17);

p.m();

}

}

Parent class type reference can be given to the child class object.

Parent p = new Son(); **🡪Correct**

Child class type reference can’t be given to the Parent class object.

Son s = new Parent(); **🡪 Incorrect**

**Method Overriding:** Parent class method implementation changed in Child class is called as Method overriding.

Method overloading can be achieved only with **Is-A** relationship.

**Example1:**

class Mother{

void lifestyle()

{

System.out.println("Simple living");

}

}

class Daughter extends Mother{

void lifestyle()

{

System.out.println("Modern living");

}

}

class MethodOverriding{

public static void main(String arg[]){

Mother m = new Mother();

m.lifestyle();

Mother d= new Daughter(); //method overriding

d.lifestyle(); //method overriding

}

}

**Example2:**

class vlc1{

void play()

{

System.out.println("Play mp3");

}

}

class vlc2 extends vlc1{

void play()

{

System.out.println("Play mp4");

}

}

class MethodOverride2{

public static void main(String arg[]){

vlc2 v2 = new vlc2();//Method Overriding

v2.play();

vlc1 v1 = new vlc2();//Method Overriding

v1.play();

vlc1 v = new vlc1();

v.play();

}

}

**Method Overloading:** Method having same name with different number of parameters with different data types (different order of types) is called as method overloading.

Method overloading can be achieved with and without **Is-A** relationship.

class AddClass{

void add(int i,int j)

{

System.out.println(i+j);

}

void add(int i, int j, int k)

{

System.out.println(i+j+k);

}

void add(int i, double j)

{

System.out.println(i+j);

}

void add(double i, int j)

{

System.out.println(i+j);

}

}

class MethodOverloading{

public static void main(String arg[]){

AddClass a = new AddClass();

a.add(4,5);

a.add(4,5,6);

a.add(4,3.4);

a.add(2.1,3);

}

}

**Note:** change in signature of method implementation is not considered as method overloading.

**🡪 NOT ALLOW AS BELOW…**

**class MethodOverLoadWithDiffSignature{**

**void add(int a, int b)**

**{**

**System.out.println(a+b);**

**}**

**int add(int a, int b)**

**{**

**System.out.println(a+b);**

**}**

**}**

**Note:** change in the return type (signature) not considered as method overriding.

**🡪 NOT ALLOW AS BELOW…**

**class AddMethodClass{**

**void add(int a, int b)**

**{**

**System.out.println(a+b);**

**}**

**}**

**class MethodOverrideWithDiffSignature extends AddMethodClass{**

**int add(int a, int b)**

**{**

**System.out.println(a+b);**

**}**

**}**

**Casting:** converting one data type into another is called as casting.

There are two types of casting.

1. Type Casting
2. Reference Type Casting.

1. **Type Casting:** is used to convert one primitive data type into another primitive data type.

Again, its divided into two types:

* 1. Implicit Type Casting: Converting Lower data type into higher
  2. Explicit Type Casting: Converting Higher data type into lower.

Implicit Type Casting is: byte 🡪 short 🡪 Int 🡪 float 🡪 long 🡪 double

char🡽

Explicit Type Casting is: double 🡪 long 🡪 float 🡪 int 🡪 short 🡪 byte

🡾 char

**Implicit Type casting:**

class ImplicitTypeCasting{

public static void main(String arg[]){

int i = 2;

**double j = i;**

char a = 'V';

**int b = a;**

System.out.println(j);

System.out.println(b);

}

}

**Explicit Type casting:**

class ExplicitTypeCasting{

public static void main(String arg[]){

double i = 3.4567;

**int j = (int)i;**

int x = 66;

**char y = (char)x;**

System.out.println(j);

System.out.println(y);

}

}

2. **Reference Type Casting:** it has two different types.

a. Up casting (Implicit Up casting): Converting Child object into Parent type reference.

Example: **Parent p = new Child();**

b. Down casting (Explicit Down casting): Converting Parent object into Child type reference.

Example: Parent p = new Child();

Child c = (child)p;

Note: For downcast first we need to up cast and later we need to download case, else it throws run time exception error class cast exception. Example not allowed: Child c = (Child) new Parent();

class Parent{

void pmethod()

{

System.out.println("Parent Method");

}

}

class Child extends Parent{

void cmethod()

{

System.out.println("Child Method");

}

}

class ReferenceTypeCasting{

public static void main(String arg[]){

Parent p = new Child(); //Up Casting

Child c = (Child)p; //Down Casting by using Up Casting Object reference

p.pmethod();

// p.cmethod();// Not allowed to access child method using prent reference.

c.pmethod();

c.cmethod();

}

}

**Abstraction:**

* A method without body (terminated with semicolon ;) should be declared as abstract.
* Any class which contains abstract method should be declared as abstract class.
* Abstract class can’t be instantiated.
* Any child class which extends abstract class should override all the abstract methods or that class should be declared abstract class.
* Abstract class should not be final.
* Abstract class should not be declared as static and final.
* Any method with the body is called as concrete method.
* Any method declared with abstract keyword is called as abstract method.
* Abstract class can have static, final and concrete methods.

abstract class Hero{

void engine()

{

System.out.println("Hero engine design");

}

abstract void key();

}

class Honda extends Hero{

void key()

{

System.out.println("Honda Key design");

}

}

class AbstractClass{

public static void main(String arg[]){

Honda h = new Honda();

h.engine();

h.key();

}

}

**Interface:**

* Interface used to achieve 100% abstraction.
* All the methods declared inside the interface are public abstract by default.
* All the variable declared inside the interface are public, static, final.
* Class can implement interface.
* We can’t create object of interface.
* Interface type reference can be given to any of its child class objects. (WebDriver driver = new FireforxDriver(); )
* Class which implements interface should override all the methods in the interface or that class also to be declared as abstract class.

**Single Interface Implementation:**

interface Switch{

public void on();

public void off();

}

class TV implements Switch{

public void on()

{

System.out.println("TV ON");

}

public void off()

{

System.out.println("TV off");

}

}

class InterfaceClass{

public static void main(String arg[]){

TV t = new TV();

t.on();

t.off();

}

}

**Multiple Interface Implementation:**

interface StateTax{

double serviceTax(double amt);

}

interface CentralTax{

double roadTax(double amt);

}

class TaxCollector implements StateTax,CentralTax{

public double serviceTax(double amt)

{

double ans = (amt/100)\*2;

return ans;

}

public double roadTax(double amt)

{

double ans = amt/100;

return ans;

}

}

class InterfaceImplementClass{

public static void main(String arg[]){

TaxCollector tc = new TaxCollector();

double st = tc.serviceTax(300.45);

double rt = tc.roadTax(600.54);

System.out.println("Service Tax: "+st);

System.out.println("Road Tax: "+rt);

}

}

**Note:** Interface can extend an interface and Class can implement interface. Example as below:

interface a{

……

……

}

Interface b **extends** a{

……

……

}

class x **implements** a{

……

……

}

**Abstraction:** Hiding the business login and allowing user to access functionality is called as abstraction.

**Abstraction** canbe achieved by using interface and abstract class.

**Access Specifiers:**  are the one which are used to control the accessibility of the class members.

Access specifiers are public, protected, default and private.

**Public:** can access

1. Inside the class
2. Inside the same package
3. Outside the package

**Protected:** can access

1. Inside the class
2. Inside the same package.
3. Outside the package (only with **Is-A** relationship)

**Default:** can access

1. Inside the class
2. Inside the same package.

**Private:** can access

1. Inside the class

Private

Protected

Public

Default

Top to Bottom – Widening.

Bottom to Top – Narrowing.

**Final:** is a keyword used in java to declare constants.

Final can be used to declare Global and Local variables.

Any variable declared with final need to be initialized as soon as they created (either directly initialization or thru constructor initialization.

**Instance final variable Example1:**

final int i = 10;

**Instance final variable Example2:**

class IstanceInitBlockClass{

final int j;

IstanceInitBlockClass() //Instance Init block

{

j = 20;

System.out.println(j);

}

}

class InstanceFinalVariable{

public static void main(String arg[]){

new IstanceInitBlockClass();

}

}

**Static final variable Example1:**

static final x = 30;

**Static final variable Example2:**

class StaticInitBlockClass{

static final int y;

static{

y=40;

}

}

class StaticFinalVariable{

public static void main(String arg[]){

StaticInitBlockClass sib = new StaticInitBlockClass();

System.out.println(sib.y);

}

}

**Final Method**

Can’t be overridden

Can be overloaded

Can be inherited

**Static Method**

Can’t be overridden

**Final Class**

Can’t be inherited.

Can create object of final class

**Note:** static final variables are called as class constants.

**Encapsulation:** grouping of related set of data and binding them into a block is called as encapsulation.

In java encapsulation can be achieved using java beans.

**Rules to create Java Beans:**

* Bean class should be declared as **public** and non-abstract.
* In Bean class, all the variable should be declared as **private**.
* In Bean class, all the variable should have corresponding getters and setters.
* Bean class object is called as DTO (data transferring object)

**Java Bean Example: (Test class file and Data class file need to create compile and run)**

**Save file as: EmpDataTest.java**

public class EmpDataTest{

public static void main(String age[]){

EmpData ed = new EmpData();

ed.setName("Vishnu");

ed.setAge(17);

ed.setGender('M');

System.out.println(ed.getName());

System.out.println(ed.getAge());

System.out.println(ed.getGender());

}

}

**Save file as: EmpData.java**

public class EmpData{

//public class EmpData implements java.io.Serializable{

private String name;

private int age;

private char gender;

public void setName(String name)

{

this.name = name;

}

public void setAge(int age)

{

this.age = age;

}

public void setGender(char gender)

{

this.gender = gender;

}

public String getName()

{

return name;

}

public int getAge()

{

return age;

}

public char getGender()

{

return gender;

}

}

/////////////////////////////////////////////////////////////////////////////////////////////

**//Java Bean Explained in 3 files: (Compile all and run Main class file)**

**//Save as file: JavaBeanEmpMainTest.java**

public class JavaBeanEmpMainTest{

public static void main(String arg[]) {

JavaBeanEmpGetSet egs = new JavaBeanEmpGetSet();

egs.setName("Vishnu");

egs.setNumber(123456789);

egs.setEmail("abc@emc.com");

JavaBeanEmpDB db = new JavaBeanEmpDB();

db.receiveData(egs);

}

}

**//Save as file: JavaBeanEmpDB.java**

public class JavaBeanEmpDB {

public void receiveData(JavaBeanEmpGetSet egs)

{

System.out.println(egs.getName());

System.out.println(egs.getNumber());

System.out.println(egs.getEmail());

}

}

**//Save as file: JavaBeanEmpGetSet.java**

public class JavaBeanEmpGetSet {

private String name;

private int number;

private String email;

public void setName(String name)

{

this.name = name;

}

public void setNumber(int number)

{

this.number = number;

}

public void setEmail(String email)

{

this.email = email;

}

public String getName()

{

return name;

}

public int getNumber()

{

return number;

}

public String getEmail()

{

return email;

}

}

/////////////////////////////////////////////////////////////////////////////////////////////

**Exceptions:** Exceptions in java are used to handle expected, unexpected or undefined things.

**Handling exception using Try and Catch blocks: (Single try can have multiple catch blocks)**

When there are multiple lines need to executed in try block, if any line gets exception it terminates execution and other lines of code will not be executed.

class ExceptionClass{

void m()

{

try{

String a = null;

// System.out.println(15/0); //uncomment to see ArithmeticException

System.out.println(a.length());

}

catch(ArithmeticException ae){

System.out.println("Zero can be div");

}

catch(NullPointerException ne){

System.out.println("Can't handl null");

}

}

}

class ExceptionHandingByTryCatch{

public static void main(String arg[]){

ExceptionClass ec = new ExceptionClass();

ec.m();

}

}

If the exception is raised inside the method and not handled, of that exception will be thrown back the caller, if that caller not handled, again it throws back to its caller, it continues until the exception is handled or JVM will handle it. Because caller of the **main** method is JVM.

Unchecked Exce

Throwable

Object

Checked Exce

Error

Run time exception.

Examples:

NullPointerException

ArithmeticException

Exception

Compile time exception.

Examples:

SQLException

ClassNotFoundException

**Checked Exception:** Any exception which needs to be handled before compiling the code are called as Checked Exception.

All checked exceptions extends **Exception** class.

Examples: SQLException, ClassNotFoundException

**UnChecked Exception:** Any exception which are handled at run-time are called as Unchecked Exception.

All unchecked exceptions extends **RunTimeException** class.

Examples: NullPointerException, ArithmeticException

**Throwable:** is a parent class of Exception. Error and Exception are the child classes of the Throwable.

**throw:** is a keyword used to throw the exception.

**Example of throw:**

class ExceptionChildClass{

public void m()

{

try{

System.out.println(15/0);

}

catch(ArithmeticException ae)

{

System.out.println("Child Class ArithmeticException message");

throw ae;

}

}

}

class ExceptionMainClass{

public static void main(String arg[]){

ExceptionChildClass acc = new ExceptionChildClass();

acc.m();

try{

System.out.println("Main class called");

}

catch(ArithmeticException ae)

{

System.out.println("Main class ArithmeticException message");

}

}

}

//Output is:

//Child Class ArithmeticException message

//Exception in thread "main" java.lang.ArithmeticException: / by zero

// at ExceptionChildClass.m(ExceptionMainClass.java:5)

// at ExceptionMainClass.main(ExceptionMainClass.java:17)

**Finally:** **finally** is a block in exception handling which is going to execute after throwing the exception or even after the try/catch block execution.

class FinallyCheckAfterException{

public static void main(String arg[]){

try{

System.out.println(15/0);

}

catch(ArithmeticException ae){

System.out.println("Catch block message...");

}

**finally{**

**System.out.println("Finally block message... printed after try/catch");**

**}**

}

}

**throws:** is a keyword which is used at the method declaration to handle exception.

throws is declaration used to check the checked exception.

Checked exception can be handled inside the **try&catch** block or can be handled at method declaration using keyword **throws**.

class ThrowsClass{

void m() throws ArithmeticException

{

System.out.println(15/0);

}

//Above throws or below try catch can be used for checked exception.

// void m()

// {

// try{

// System.out.println(15/0);

// }

// catch(ArithmeticException ae){

// System.out.println("ArithmeticException Exception Vishnu");

// }

// }

}

class ThrowsException{

public static void main(String arg[]){

ThrowsClass tc = new ThrowsClass();

tc.m();

}

}

When there are multiple checked exceptions been thrown from the method we can generalize the throws by indicating **Exception** class.

**Throws Exception Example: (Simple and Good One)**

**class Cal{**

**public int div(int a, int b) throws ArithmeticException**

**{**

**int c;**

**c =a/b;**

**return c;**

**}**

**}**

**class ExceptionThrowsTest{**

**public static void main(String arg[]){**

**Cal c = new Cal();**

**try{**

**int x = c.div(500,0);**

**System.out.println(x);**

**}**

**catch(ArithmeticException ae)**

**{**

**System.out.println("THROWS: Num Can't div by zero");**

**}**

**}**

**}**

**Throw Exception Example: (Simple and Good One)**

**class Cal{**

**public int div(int a, int b)**

**{**

**int c;**

**if(b == 0)**

**{**

**throw new ArithmeticException("THROW: Num Can't div by zero");**

**}**

**else**

**{**

**c =a/b;**

**return c;**

**}**

**}**

**}**

**class ExceptionThrowTest{**

**public static void main(String arg[]){**

**Cal c = new Cal();**

**int x = c.div(5,0);**

**System.out.println(x);**

**}**

**}**

**Custom Exception:**

User defined exceptions are called as custom exceptions.

User can create custom exception by extending Exception class.

User can override exception methods toString() and getMessage() to define user messages.

**Custom exception are two types:**

**Checked Exception:** this can be achieved by extending **Exception** class.

**Unchecked Exception:** this can be achieved by extending **RunTimeException** class

Still need to work on Custom Checked and Unchecked???

**Example of Checked Custom Exception:**

**class** InvalidAgeException **extends** Exception{

InvalidAgeException(String s){

**super**(s);

}

}

**class** ExceptionCustomChecked{

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

**try**

{

*validate*(13);

}

**catch**(Exception m)

{

System.***out***.println("Exception occured: "+m);

}

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

}

**static** **void** validate(**int** age)**throws** InvalidAgeException

{

**if**(age<18)

{

**throw** **new** InvalidAgeException("not valid");

}

**else**

{

System.***out***.println("welcome to vote");

}

}

}

**Example of UnChecked Custom Exception:**

**class** NoDivbyZeroException **extends** RuntimeException{

**public** NoDivbyZeroException (String msg)

{

**super**(msg);

}

}

**class** ExceptionCustomUnChecked{

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

**int** a = 500;

**int** b = 0;

**try**

{

**if** (b==0) **throw** **new** NoDivbyZeroException("Div by ZERO not allowed");

**else**

System.***out***.println("Result: " +a/b);

}

**catch** (NoDivbyZeroException e)

{

System.***out***.println(e);

}

}

}

**Interface**

100% abstraction

Only public abstract methods

All methods are public

Variables by default: public, static final.

Multiple inheritance

Constants

No Constructor

implements

**Abstract**

50-50 abstraction

Concrete and Abstract methods

Methods can be – public, default and protected.

Variables can be any type.

No Multiple inheritance

No Constants

Constructor

extends

**Rules for Method Overriding:**

There should be a Is-A relationship.

Signature fo the method should be same.

Access specifier is private we can’t override.

Access specifier is default we can override by using default or protected or public.

Access specifier can be widening not be narrowed, example: protected can override as protected or public.

Private

Protected

Public

Default

**Constructor**

Same is class name without any return type.

Can’t be static, final, abstract

Recursion is not possible

Can’t be inherited

Default constructor

Called when we create object

Use to initialize instance members of the class.

**Method**

Can be same as class name or different, it should have return type.

It can be static, final, abstract

Recursion is possible

Can be inherited.

No default method

Called when needed.

Not initialize instance variables of the class

**Object Class:**

If the class does not extends any other class, compiler by default extends object class.

Object class present in java.lang package

Object class is the super most class in Java.

Object class type reference can be given to any other class object.

Object class has 11(+1 first one found in internet) methods:

1. Object()
2. String toString();
3. int hashCode();
4. boolean equals(Object obj);
5. void finalize();
6. final getClass();
7. Object clone();
8. final void wait();
9. final void wait(long timeout);
10. final void wait(long timeout,int nanoseconds)
11. final notify();
12. final notifyAll();

**Diamond Problem:**

Object class

class A

class B

class C extends A,B

**Scanner Class:** Scanner is a class which is present in java.util package (import java.util.Scanner;)

We can create object of Scanner class.

Scanner scan = new Scanner(System.in)

To Scan values from keyboard we have corresponding methods.

Scanner method until space or tab for next(); and until next line for nextLine();

import java.util.Scanner;

class ScannerClass{

public static void main(String arg[]){

Scanner s = new Scanner(System.in);

//number and boolen scanner examples.

int i = s.nextInt();

float f = s.nextFloat();

double d = s.nextDouble();

//long l = s.nextLong();

//boolean b = s.nextBoolean();

//string scanner examples.

String s1 = s.next();

String s3 = s.nextLine();

System.out.println(i);

System.out.println(f);

System.out.println(d);

//System.out.println(l);

//System.out.println(b);

System.out.println(s1);

System.out.println(s3);

}

}

**Has-A Relationship:**

A class having a reference to an object of another class is called as **Has-A** relationship

class Mobile{

String model;

void doCall()

{

System.out.println("Calling...");

}

}

class Person{

int age;

Mobile m = new Mobile();

void walk()

{

System.out.println("Walking...");

}

}

class Has\_A\_RelationshipTest{

public static void main(String arg[]){

Person p = new Person();

p.age = 17;

**p.m.model = "N-17"; //Has-A relation**

p.walk();

**p.m.doCall(); //Has-A relation**

}

}

**System.out.println:**

System 🡪 is a class

out 🡪 is a reference variable of PrintStream class

println 🡪 is a instance method of PrintStream class

**System.in:**

System 🡪 is a class

in 🡪 is a reference variable in InputStram class

**Access Modifiers**

final

static

abstract

trancint

volatile

**Access Specifiers**

private

protected

public

**Polymorphism**

A member of a class having a same name but behaves differently based on the input given, is called as polymorphism.

Its two types:

1. Dynamic Polymorphism / Run-Time Polymorphism 🡪 Method Overriding is an example.
2. Static Polymorphism / Compile time Polymorphism 🡪 Method OverLoading is an example.

**String:**

String is class in Java

String class is present in java.lang package.

String objects are immutable.

String objects are created inside the String pool.

When we create an object of string without using new keyword compiler will not create new object it just makes the reference to the existing object with different reference id.

String object can be created as:

class StringTest{

public static void main(String arg[]){

**String r = "Rock";**

**String s = new String("Vishnu");**

System.out.println(r+" "+s);

}

}

String a = “Vishnu”

String b = “Vishnu”

String c = “Java”

For String a and b: It just uses diff address to locate same object ex: a = 100, b = 200

For String c it create new object and refer.

Vishnu

Java

class StringTest{

public static void main(String arg[]){

String r = "Rock";

String s = new String("Vishnu");

System.out.println(r+" "+s);

//String comparision test

String x = "Vishnu";

String y = new String("Vishnu");

if (x==y)

{

System.out.println("This string compare will not print");

}

else if(x.equals(y))

{

System.out.println("This string compare text will print");

}

//String compare second example

String g1 = new String("ganesh");

String g2 = "ganesh";

String g3 = "Ganesh";

if(g1.equals(g2))

{

System.out.println("g1 = g2 compare text will print");

}

if(g1.equals(g3))

{

System.out.println("g1 = g3 compare text will NOT print");

}

if(g1.equalsIgnoreCase(g3))

{

System.out.println("g1 = g3 compare text will print After IgnoreCase");

}

//String concatination.

String c1 = "Vishnu";

String c2 = c1.concat(" GC");

System.out.println(c1);

System.out.println(c2);

//String length / upper / lower

String l1 = "Vishnuvardhana";

System.out.println(l1.length());

System.out.println(l1.toUpperCase());

System.out.println(l1.toLowerCase());

// String trim

String t1 = " Trim Space at FirstAndLast ";

System.out.println(t1.trim());

// Strings storing and printing using array

String s1 = "Nanu Neenu Aanu Tanu";

String sp[] = s1.split(" ");

for(int i=0; i<s1.length();i++)

{

System.out.println(sp[i]);

}

}

}

**String Builder and String Buffer:**

These class are final classes and there are present in java.lang package.

There class objects are mutable. (content of object can be changed or modified)

**String Builder object we can create in two ways:**

**String Buffer object we can create in two ways:**

**Example program has both examples:**

class StringBuilderAndBuffer{

public static void main(String arg[]){

//String Builder

**StringBuilder sb1 = new StringBuilder("Vishnu SB1");**

**StringBuilder sb2 = new StringBuilder();**

sb2.append("Vishnu SB2");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("String Builder Print Start");

System.out.println(sb1);

System.out.println(sb2);

System.out.println("String Builder Print End");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

//String Buffer

**StringBuffer sbf1 = new StringBuffer("Vishnu SBF1");**

**StringBuffer sbf2 = new StringBuffer();**

sbf2.append("Vishnu SBF2");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("String Buffer Print Start");

System.out.println(sbf1);

System.out.println(sbf2);

System.out.println("String Buffer Print End");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

**Wrapper Class**

For each primitive type there is one corresponding class and these classes are called as Wrapper class.

Object Class

Number Class

Boolean

Character

Byte

Short

Integer

Long

Float

Double

**Boxing:** Converting primitive type into corresponding wrapper class type is called as Boxing.

**int i = 89;**

**Integer j = new Integer(i);**

**Un-Boxing:** Converting wrapper class type into corresponding primitive type is called as Un-Boxing.

**Integer k = new Integer(33 );**

**int l = k.intValue();**

**Auto-Boxing:** Compiler internally converting primitive type into non-primitive type is called as Auto-Boxing.

**int x = 45;**

**Integer y = x;**

**Note:** Only primitive type have the wrapper classes. And String and Non-primitive have no wrapper classes.

**Example Program works in eclipse and not in command prompt.**

**class** WrapperClassTest{

**public** **static** **void** main(String arg[]){

//Boxing

**int** i = 89;

Integer j = **new** ~~Integer~~(i);

System.***out***.println(j);

//UnBoxing

Integer k = **new** ~~Integer~~(33);

**int** l = k.intValue();

System.***out***.println(l);

//Auto-Boxing

**int** x = 45;

Integer y = x;

System.***out***.println(y);

}

}

**Parsing:**

Converting a string value into corresponding wrapper class type is called as parsing.

class ParsingTest{

public static void main(String arg[]){

**String s = "111.23";**

**String ss = "222";**

**double d = Double.parseDouble(s);**

**int i = Integer.parseInt(ss);**

System.out.println(d);

System.out.println(i);

}

}

**Array:**

Array is collection of multiple elements of same type under one reference

Example: int a[] = new a[6];

class ArrayTest{

public static void main(String arg[]){

int[] a = {10,20,30,40};

for(int i = 0; i<a.length; i++)

{

System.out.println(a[i]);

}

}

}

**Receive and Print Two-Dimensional array using Scanner:**

import java.util.Scanner;

class TwoDimentionalArray{

public static void main(String arg[]){

int a[][] = new int[2][2];

Scanner s = new Scanner(System.in);

System.out.println("Enter Array.....");

for(int i=0;i<2;i++)

{

for(int j=0; j<2; j++)

{

a[i][j]=s.nextInt();

}

}

System.out.println("Now Prining Array.....");

for(int i=0;i<2;i++)

{

for(int j=0; j<2; j++)

{

System.out.println(a[i][j]);

}

}

}

}

**Collections:**

Is a Framwork used to store and manipulate group of objects.

Operations you can perform is: Search, Sort, Insert, Update and Delete.

**Note:** Below image class/interface name has I and C to indicate **interface (I) and class (C)**

Iterable I

Collection I

List I

Queue I

Set I

SortedSet I

DeQueue I

**ArrayList Example:**

import java.util.\*;

class **CollectionsArrayListTest**{

public static void main(String arg[]){

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("ArrayList Example Start..........");

ArrayList al = new ArrayList();

al.add("ABCD");

al.add(1234);

al.add(12.33);

//For loop

System.out.println("ForLoop RESULT");

for(int i = 0; i<al.size(); i++)

{

System.out.println(al.get(i));

}

//For each loop

System.out.println("ForEachLoop RESULT");

for(Object r : al)

{

System.out.println(r);

}

//While loop

System.out.println("WhileLoop RESULT");

Iterator it = al.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

System.out.println("ArrayList Example End..........");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("ArrayStringList Example Start..");

ArrayList<String> als = new ArrayList<String>();

als.add("ABC");

als.add("EFG");

als.add("XYZ");

System.out.println("For Loop of ArryString List");

for(int j = 0; j<als.size(); j++)

{

System.out.println(als.get(j));

}

System.out.println("ForEach Loop of ArryString List");

for(Object rs : als)

{

System.out.println(rs);

}

System.out.println("While Loop of ArryString List");

Iterator<String> its = als.iterator();

while(its.hasNext())

{

System.out.println(its.next());

}

System.out.println("ArrayStringList Example End......");

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.println("ArrayIntegerList Example Start..");

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

ali.add(11);

ali.add(22);

ali.add(33);

System.out.println("For Loop of Arry Int List");

for(int k = 0; k<ali.size(); k++)

{

System.out.println(ali.get(k));

}

System.out.println("ForEach Loop of Arry Int List");

for(Object ri : ali)

{

System.out.println(ri);

}

System.out.println("While Loop of Arry Int List");

Iterator<Integer> iti = ali.iterator();

while(iti.hasNext())

{

System.out.println(iti.next());

}

System.out.println("Array Int List Example End....");

}

}

**ListIterator example with hasPrevious() used after hasNext():**

import java.util.\*;

class ListIteratorTestWithArrayListInt{

public static void main(String arg[]){

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

ali.add(1111);

ali.add(2222);

ali.add(3333);

ListIterator<Integer> liti = ali.listIterator();

while(liti.hasNext())

{

System.out.println(liti.next());

}

**System.out.println("After hasNext we can use Previous");**

while(liti.hasPrevious())

{

System.out.println(liti.previous());

}

}

}

**LinkedList Example:**

import java.util.\*;

class CollectionsLinkedListTest{

public static void main(String arg[]){

LinkedList ll = new LinkedList();

ll.add("abc");

ll.add(123);

System.out.println("\*\*\*\*\*GEN LinkedList \*\*\*\*\*\*\*\*\*");

System.out.println("LinkedList with For");

for(int i=0; i<ll.size(); i++)

{

System.out.println(ll.get(i));

}

System.out.println("LinkedList with For Each");

for(Object r : ll)

{

System.out.println(r);

}

System.out.println("LinkedList with While");

Iterator it = ll.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

System.out.println("\*\*\*\*\*GEN LinkedList \*\*\*\*\*\*\*\*\*");

LinkedList<Integer> lli = new LinkedList<Integer>();

lli.add(1111);

lli.add(2222);

lli.add(3333);

System.out.println("LinkedList Int with For");

for(int j=0; j<lli.size(); j++)

{

System.out.println(lli.get(j));

}

System.out.println("LinkedList Int with For Each");

for(Object r : lli)

{

System.out.println(r);

}

System.out.println("LinkedList Int with While");

Iterator iti = lli.iterator();

while(iti.hasNext())

{

System.out.println(iti.next());

}

}

}

**Vector Example:**

import java.util.\*;

class CollectionsVectorTest{

public static void main(String arg[]){

Vector v = new Vector();

v.add("Vishnu");

v.add(17);

System.out.println("For loop printing.....");

for(int i=0; i<v.size();i++)

{

System.out.println(v.get(i));

}

System.out.println("Enumeration printing.....");

Enumeration e = v.elements();

while(e.hasMoreElements())

{

System.out.println(e.nextElement());

}

}

}

**Note: Enumeration supports only in Vector and Stack.**

**List**

Index

Duplicate

Multiple Null

Order

**Set**

Non-Index

Unique

Single Null

UnOrder

**Set:**

**HashSet Example:**

import java.util.\*;

class CollectionsHashSetTest{

public static void main(String arg[]){

System.out.println("HasSet For Print.......");

HashSet hs = new HashSet();

hs.add("Avanu");

hs.add("Evanu");

for(Object r:hs)

{

System.out.println(r);

}

System.out.println("HasSet Iterator Print.......");

Iterator it = hs.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

}

}

**LinkedHashSet Example:**

import java.util.\*;

class CollectionsLinkedSetTest{

public static void main(String arg[]){

LinkedHashSet ls = new LinkedHashSet();

ls.add(3333);

ls.add(5555);

ls.add(1111);

ls.add(4444);

System.out.println("LinkedHasSet Using for");

for(Object r : ls)

{

System.out.println(r);

}

System.out.println("LinkedHasSet Using Iterator");

Iterator it = ls.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

}

}

Note1: HashSet is thread safe and LinkedHasSet is not thread safe.

Note2: TreeSet sort only same type of objects.

**TreeSet Example:**

import java.util.\*;

class CollectionsTreeSetTest{

public static void main(String arg[]){

TreeSet ts = new TreeSet();

ts.add(9999);

ts.add(1111);

ts.add(5555);

ts.add(3333);

//ts.add(1111); //if you add duplicates it ignores and its sort same type objects.

System.out.println("TreeSet For Print");

for(Object r : ts)

{

System.out.println(r);

}

System.out.println("TreeSet Iterator Print");

Iterator it = ts.iterator();

while(it.hasNext())

{

System.out.println(it.next());

}

}

}

**PriorityQueue Example:**

import java.util.\*;

class CollectionsPriorityQueueTest{

public static void main(String arg[]) {

PriorityQueue pq = new PriorityQueue();

pq.add("Vishnu");

pq.add("Savi");

pq.add("Vaishnavi");

pq.add("Sanatan");

for(Object r : pq)

{

System.out.println(r);

}

}

}

**MAP:**

Map is a interface present in java.util in map the objects get stored in the form of key and value.

Map I

NavigableMap I

SortedMap I

**HashTable Example:**

import java.util.\*;

class MapHashTableTest{

public static void main(String arg[]){

Hashtable<Integer,String> ht = new Hashtable<Integer,String>();

ht.put(1,"Vishnu");

ht.put(2,"Savi");

ht.put(3,"Vaishnavi");

ht.put(4,"Sanatan");

for(Map.Entry<Integer,String> r : ht.entrySet())

{

System.out.println(r.getKey());

System.out.println(r.getValue());

}

}

}

**HashMap Example:**

import java.util.\*;

class MapHashMapTest{

public static void main(String arg[]){

HashMap<Integer,String> hm = new HashMap<Integer,String>();

hm.put(1,"Evanu");

hm.put(4,"Avanu");

hm.put(2,"Aanu");

hm.put(3,"Taanu");

for(Map.Entry<Integer,String> r : hm.entrySet())

{

System.out.println(r.getKey());

System.out.println(r.getValue());

}

}

}

**LinkedHashMap Example:**

import java.util.\*;

class MapLinkedHashMapTest{

public static void main(String arg[]){

LinkedHashMap<Integer,String> lhm = new LinkedHashMap<Integer,String>();

lhm.put(2,"Aanu");

lhm.put(3,"Taanu");

lhm.put(1,"Naanu");

lhm.put(4,"Avanu");

for(Map.Entry<Integer,String> r : lhm.entrySet())

{

System.out.println(r.getKey());

System.out.println(r.getValue());

}

}

}

**TreeMap Example:**

import java.util.\*;

class MapTreeMapTest{

public static void main(String arg[]){

TreeMap<Integer,String> tm = new TreeMap<Integer,String>();

tm.put(1,"Avanu");

tm.put(4,"Evanu");

tm.put(3,"Aanu");

tm.put(2,"Taanu");

for(Map.Entry<Integer,String> r : tm.entrySet())

{

System.out.println(r.getKey());

System.out.println(r.getValue());

}

}

}

**File Handing:**

**Read Write**

FileInputStream FileOutputStream

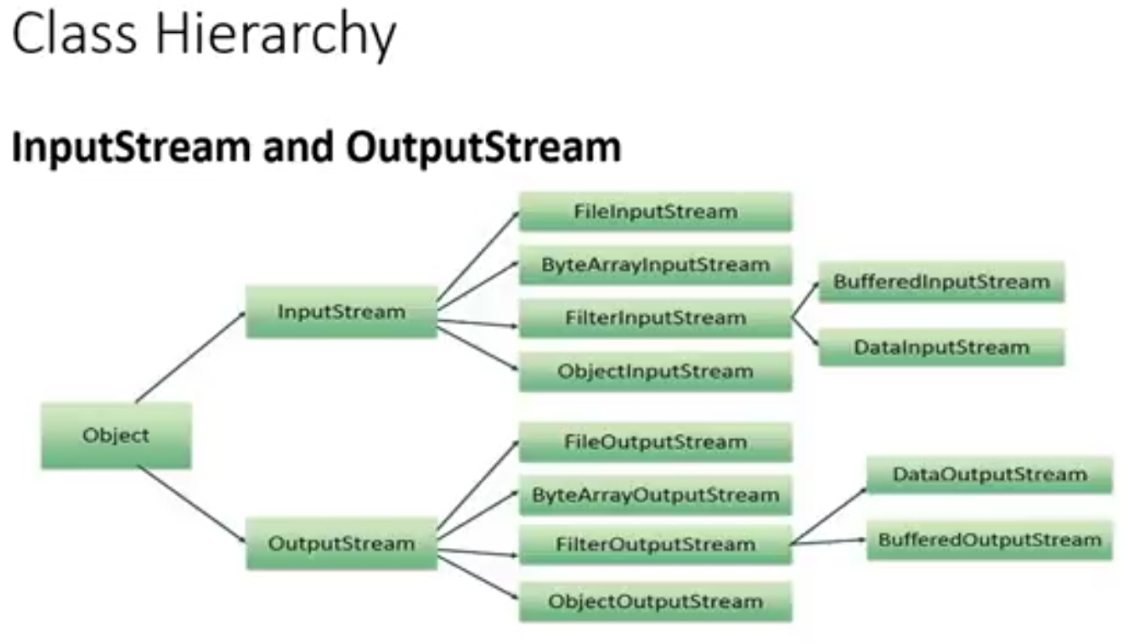
BufferInputStream BufferOutputStream

FileReader FileWriter

BufferReader BufferWriter

Byte Stream: FileInputStream and FileOutputStream

Character Stream: FileReader and FileWriter



**FileInputStream:** used to read data from the file.

FileInputStream fis = new FileInputStream("C:\\gcv\\test.txt");

read() – read byte inputstream.

available() – give number of bytes in the input.

skip() – skip n bytes

close() – close file

**FileOutputStream:** used to write data into the file.

FileOutputStream fos = new FileOutputStream("C:\\gcv\\test.txt");

write() – write byte to the output stream.

flush() – flush/clean output stream.

close() – close file