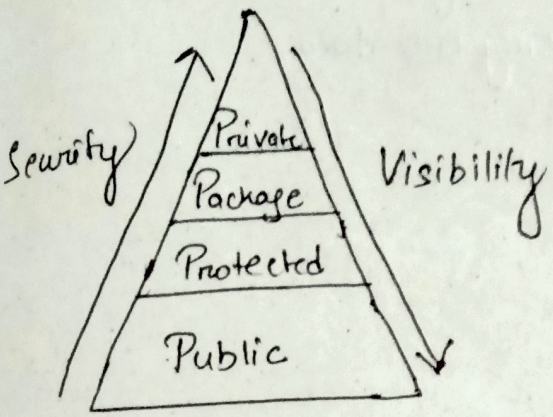


Object class → entire
- Supermost class at ↑ java level of hierarchy.
- Available in lang. package
- By default given, if the class is not declared.

Object Class –

→ It is the supermost class of the entire java class hierarchy.

→ It is a non-final class (means it can be extended).

→ It is available in java.lang package.

→ Every class developed or created, by default, extends object class.

→ Object class has only one constructor that is no argument constructor (default constructor).

Methods of Object class:

– Public	String	toString()
– Public	Boolean	equal (object obj)
– public	int	hashcode()
– public	final void	wait()
– public	final void	wait (long m)
– public	final void	wait (long m, int n)
– public	final void	notify()
– public	final void	notifyAll()
– public	final class	getClass()
– protected	void	finalize() [Garbage Collection]
– protected	Object	clone()

Multi-threading

Note — When we have to create an exact copy of a object we use clone(), when we copy only one data member we use copy constructor.

```
class A
{
}
public class Test{
        psvm(--){
            A a = new A();
            SOPln (a.getClass().getSimpleName());
        }
}
```

Method Chaining

```
A a = new A();
class c = a.getClass();
String s = c.getName();
SOPln(s);
```

a.getClass().getName();

13/08/23

List of some final methods in Object class:-

- → wait()
- → wait(long m)
- → wait(long m, int n)
- → notify()
- → notifyAll()
- → getClass()

List of some final classes in Object class :-

- String
- StringBuffer
- StringBuilder

- Integer
- Float
- Boolean
- Double
- Short
- Byte
- Char
- Long

} Wrapper Class

Eg

```
class A {
    int i;
    public A(int i){
        this.i;
    }
    @override
    public String toString() {
        return "A [i=" + i + "]";
    }
}
```

Overriding to String method :-

→ toString() method belongs to object class and when used in our classes, it returns the address of the object.

→ toString() method is overriden in our (user-defined) classess, to provide the string representation of the state (content) of the object.

Eg

```
class A {
    int i;
    public A(int i) {
        this.i = i;
    }
```

```
@override
public String toString(){
        return "A[i=" + i + "]";
}     }
}

public class Test{

        PSVM(--){

            A a1 = new A(10);
            SOPln(i);
            A a2 = new A(15);
            SOPln(i);

        }
}
```

i = a1.toString()

i = a2.toString()

Address

Q) class student{

```
int id;
String name;
public student(int id, String name){
        this.id = id;
        this.name = name;
    }

@override
public String toString(){
        return "id=" + id + "," + "name=" + name;
    }

public class Test{
        PSVM(--){
            Student s1 = new student(02, Rai);
            SOPln(s1.toString());
            Student s2 = new Student(03, Minu);
            SOPln(s2.toString());
} }
```

Case Study / Rules to follow for overriding :-

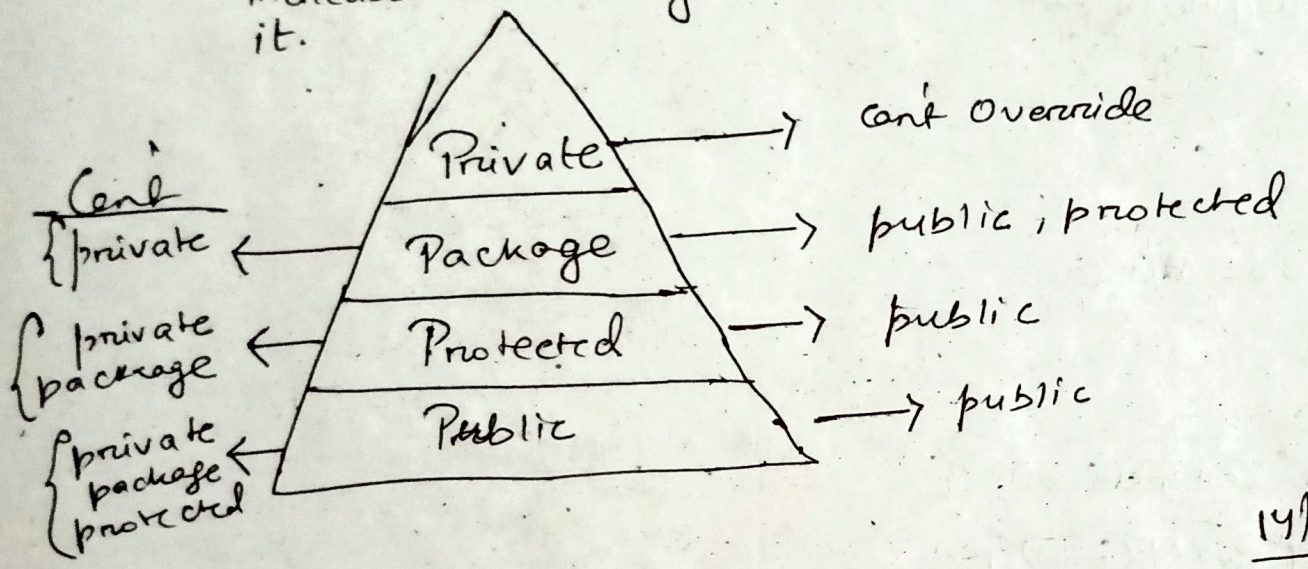
Test Case 1) Private methods cannot be overriden.

Test Case 2) Static methods cannot be overriden.

Test Case 3) Final methods cannot be overriden.

Test Case 4) Non-Static methods should be overriden as non-static but not, static methods.

Test Case 5) while ~~By~~ overriding, Retain the same access modifier or increase the visibility of access modifier but do not decrease it.



14/03/23

Test Case 6) - The return type of the sub class can be same as the return type of the superclass.

- If the return type of the superclass is parent type then the return type of the subclass can be parent type or child type. This is known as Co-Variant return types.

```
public class Test {
    public static void m1(int choice){
        if (choice==1)
        {
            SOPln("Hi");
            SOPln("Hello");
            SOPln("welcome");
        } return;
```

```
        SOPln("Hi Hello");
        SOPln("Hi welcome");
    }
public static void main(String args[]){
        m1(1);
    }
```

→ return statement returns with respect to return type in the method.

'InstanceOf' keyword:-

```
class A
{
}
class B extends A
{
}
public class TestInstanceOf {
    PSVM(--){
        Object obj = new Object();
        SOPln(obj instanceof Object); //T
        SOPln(obj instanceof A); //F
        SOPln(obj instanceof B); //F
        A a = new A();
        SOPln(a instanceof Object); //T
        SOPln(a instanceof A); //T
        SOPln(a instanceof B); //F
        B b = new B();
        SOPln(b instanceOf Object); //T
        SOPln(b instanceOf A); //T
        SOPln(b instanceOf B); //T
    }
}
```

→ 'instanceOf' keyword in java is used to check whether the object belongs to their classes or superclasses or not.

→ It returns boolean value (True or False).

→ 'IS A' relationship is mandatory to use instanceOf keyword, otherwise throws compile-time error.

→ It returns true if the object can access or belong to their class or superclasses, otherwise returns false.

'Super' keyword -

→ 'Super' keyword in java is used to access the properties of the immediate super class.

→ 'IS A' relationship is mandatory.

→ If the subclass and superclass has the properties with the same name, to access the properties of parent/super class 'super' keyword is used.

```
class A {
    int i = 500;
    public void m1(){
            SOPln ("Learn Java");
        }
    }
Class B extends A {
        int i = 800;
        public void getI() {
                SOPln ("i=" + this.i);
                SOPln ("i=" + super.i);
            }
    }
public class Test {
        psvm(...){
```

```
        B b= new B();
    public void m1()
    {
        super.m1();
        SOPln ("Isfiders will give icb");
    }
}
public class Test {
        psvm(--){
        B b= new B();
        b.getI();
        b.m1();
        }
    }
```

→ 'super' keyword can be used in any line of the method.

→ It is used in non-static methods and constructors not in static methods.

15/03/23.

```
class A {
    public A (){
            SOPln ("class A NO Arg");
        }
    public A (int m) {
            SOPln ("class A 1 Arg");
        }
class B extends A {
    //B(){
```

```
    // super();  } - Getting executed internally
    // }           - Implicit / Explicit
}
public class Test {
    P S V M(--) {
        new B();
    }
}
```

Call to super - `super();` '()' → Constructor

Call to this - `this();`

→ Call to super is used to call the immediate super class constructor.

→ Every constructor created, the first line of code executed is super call to ~~this~~, with no arguments.

→ If a class doesn't have any constructor, JVM calls a constructor known as 'default constructor', which is calling 'call to super'.

→ Call to super can be used only inside constructors not a non-static method or static methods.

→ Call to super should be the first line of the constructor.

→ By default, every constructor calls 'call to super' with no arguments.

→ To call parameterized constructor of the parent class from child class 'call to super' is explicit.

```
Class A extends Object {
    public A() {
        SOPln("class A No Arg");
    }
    public A(int m) {
        SOPln("class A 1 Arg");
    }
}
class B extends A {
    public B() {
        super(5);
        SOPln("class B No Arg");
    }
}
public class Test {
    P.S.V.M(--) {
        new B();
        new B("Java");
    }
}
```

```
public B(String Name)
{
    super();
    SOPln("classB String
           Arg");
}
```

class is Empty → default constructor → classload{ super(); }

class with no-arg → super();

class with arg → super();

```
class A
{
   int i;
   int j;
   public void displayAValues()
   {
      SOPln (i+" "+ j);
   }

}
class B extends A
 {
     int i;
     int j;
     public B (int i, int j)
      { this.i=i;
        this.j = j;
        super.i = i;
        super.j = j;
      }
      public void showBValues()
      {
         SOPln (i + " " + j);
      }
 }
 public class Test {
          PSVM(--){
                B b = new B(12,45);
                b.displayAValues();
                b. showBValues();
            }
        }
```

```
Class A {
    int i;
    int j;
    public A (int i, int j){
        this.i = i;
        this.j = j;
    }
    public void displayAValues () {
        SOPln (*i+" "+j);
    }
}
class B extends A {
    int i;
    int j;
    public B (int i, int j){
        super (i,j);
        this.i = i;
        this.j = j;
    }
    public void showBvalues(){
        SOPln (i+" "+j);
    }
}
public class Test{
    PSVM(--){
        B b = new B (12, 45);
        b.displayAValues();
        b.showBValues ();
    }
}
```

```
class Human {
    String name;
    int age;
    String gender;
    public Human (String name, int age, String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
class Doctor extends Human {
    String specialist;
    public Doctor (String name, int age, String gender,
                   String specialist)
    {
        super (name, age, gender);
        this.specialist = specialist;
    }
    public void displayDoctorDetails () {
        SOPln ("Doctor name: " + name);
        SOPln ("Doctor Age: " + age);
        SOPln ("Doctor Gender: " + gender);
        SOPln ("Doctor specialized: " + specialist);
    }
}
class Teacher extends Human {
    String subject;
    public Doctor (String name, int age, String gender, String subject)
    {
        super (name, age, gender);
        this.subject = subject;
    }
```

```
public void displayTeacherDetails(){
        SOPln("Teacher Name:" + name);
        SOPln("Teacher Age:" + age);
        SOPln("Teacher Genden:" + genden);
        SOPln("Teacher's subject:" + subject);
    }
}
public class EmployeeInfo {
    PSVM(--){
        Doctor doctor1 = new Doctor("Mahesh", 33,
                                    "Male", "Ortho");
        doctor1.displayDoctorDetails();
        Teacher teacher1 = new Teacher("Ramesh", 29,
                                       "Male", "Maths");
        teacher1.displayTeacherDetails();
    }
}
```

— X —

[why call to super for every constructor by default?]

```
class A {
    int i;
    A(){
        i=100;
    }
}
class B extends A {
    int j;
    B(){
        j=78;
    }
}
```

```
public class Test {
    PSVM(--){
        B b = new B();
        SOPln(b.i);
        SOPln(b.j);
    }
}
```

Assgn

Difference bet" this/this() and super/super().

this()	super()
i) this() is used to ~~case~~ call the constructor of the same class.	i) super() is used to call the constructor of immediate super class.
ii) IS A relationship is required.	ii) IS A relationship is mandatory.
iii) this() is explicit.	iii) super() is implicit and also explicit.
iv) Only used in constructor.	iv) only used in constructor.
v) should be the first line of the constructor.	v) should be the first line of the constructor.
vi) can call only one constructor of the same class.	vi) can call only one constructor of super class.
vii) this() is used to pass value to another constructors within the class.	~~vii)~~
viii) Also known as constructor chaining.	vii) Also known as constructor chaining.

Data Abstraction :-

→ Hiding the implementation (essential features) from the user and proving only the functionality (behaviour) to the user is known as data abstraction.

→ It is also known implementation ~~fun~~ hiding.

→ It can be achieved in two ways namely -

i) Abstract class. (0 to 100 % abstraction)

ii) Interface (100 % abstraction)

Data Encapsulation :-

→ Wrapping the data members and the member function into a single unit called class is known as data encapsulation.

→ It is also known as data hiding or information hiding.

```
class A {
    private int i;
    Public int getI() // getter method / accessor
    {
     return i;
    }
    Public void setI (int i) // setter method / mutation
    {
     this.i = i;
    }
}
Public class TestDataEncap {
        PSVM(--){
            A a = new A();
            SOP(a.i); // not possible
            SOP(a.getI());
```

```
        a.setI (45);
        SOP (a.getI());
    }
}
```

→ To create a data encapsulated class we should

i) Declare the data member as private

ii) Define a public method to fetch the value of private data member known as getter method (accessor).

iii) Define a public method to set of the value of a private data member known as setter method (mutator).

Private Constructor :-

→ It is a set of statement that is executed when the object is created.

→ Private constructor is defined to hide the constructor within the class.

→ When the constructor is declared as private, objects cannot be created outside the class but it can be created within the class.

→ 'N' number of objects can be created within the class.

→ To object or for a private constructor we should make use of (or) define helper method known as factory method.

→ To define a factory method we should

i) Declare access modifier as public

ii) Declare modifier as static.

iii) Return type should be class type.

iv) Declare a user defined method name (recommended getInstance).

```
Class A {
        Static int count;
        Private A() {
            count ++;
        }
    Public static A getInstance() // helper method → Factory metho
    {
    return new A();
    }
}
Public class Private Constructor {
    P S VM(--) {
        A a = new A(); // not possible when constructor
                       is private
        A a1 = A.getInstance();
        A a2 = A.getInstance();
        A a3 = A.getInstance();

        SOP (a1==a2); // False
        SOP (a2==a3); // False
        SOP (a3==a1); // False
        SOP (A.count); // 3
```

For counting object:

Comparing the address of the object.

how many object is there.

SingleTon class:-

→ A class which consists of atmost one object is known as SingleTon class.

→ SingleTon class is one of the design pattern in Java.

→ To create a singleTon class we should

i) Define a static type class reference variable.

ii) Define private constructor.

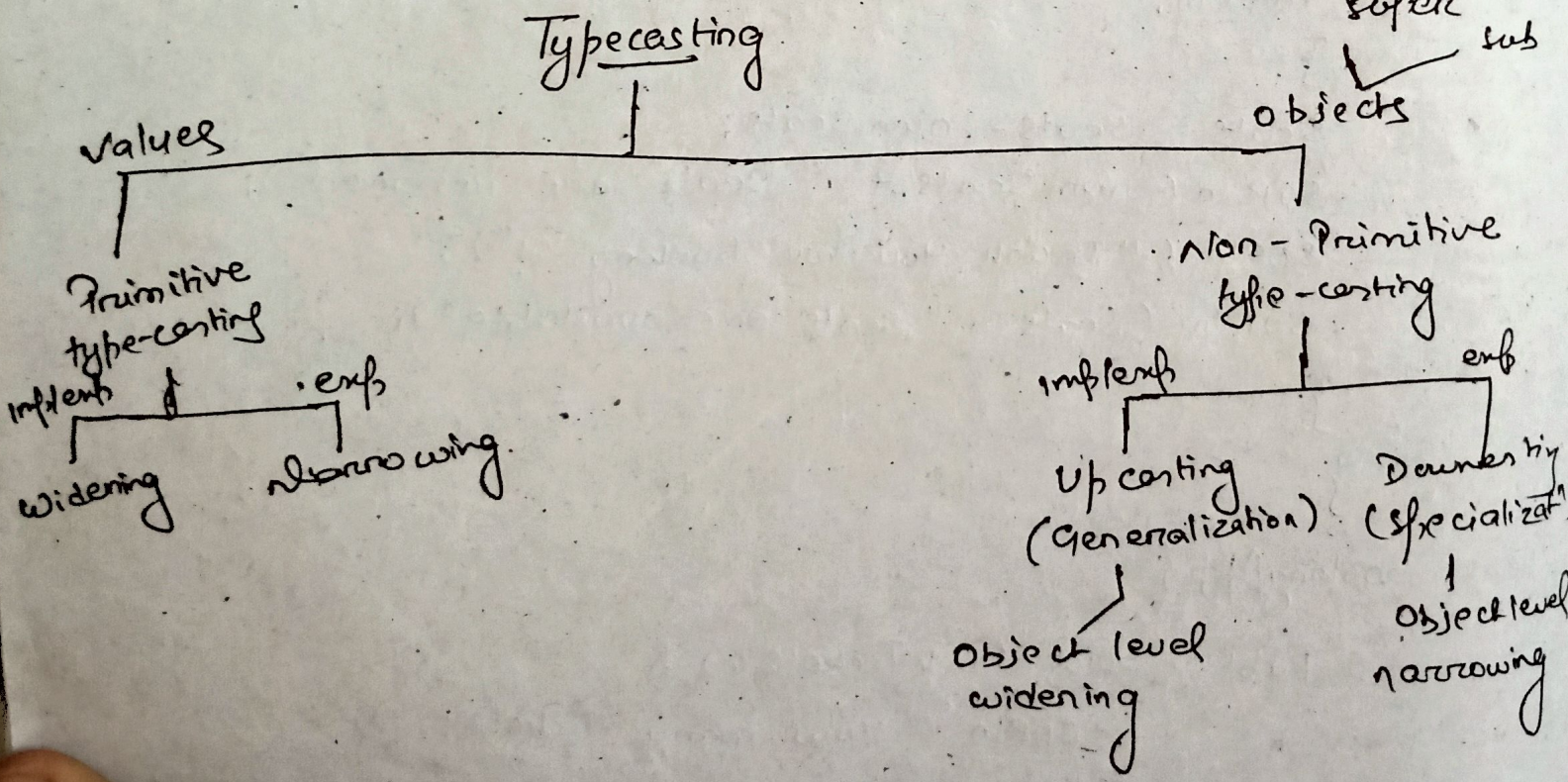
iii) Define a factory method which returns only one object using condition.

Ticket Booking Program

```
import java.util.Scanner;
class Theatre{
    int seats = 50;
    static Theatre t = null;
    Private Theatre()
    {
    }
    Public static Theatre getInstance()
    {
      if (t == null) t = new Theatre();
      return t;
    }
    Public void reserveSeats (int numSeats)
    {if (numSeats > seats)
      {
      SOPln (numSeats + " seats are not available");
      SOPln (seats + " seats are available");
      return;
      }
      seats = seats - numSeats;
      SOPln (numSeats + " seats are reserved");
      SOPln ("Thank you for booking--!");
      SOPln (seats + " seats are available");
    }
    }
class BookingApp{
    Public void bookTickets(){
            SOPln ("How many Tickets: ");
            Scanner scan = new Scanner (System.in)
            int tickets = scan.nextInt();
```

```
Theatre theatre = Theatre.getInstance();
theatre.reserveSeats(tickets);
}
}
Public class BookMyShow {
    Public static void main(String args){
    BookingApp cust1 = new BookingApp();
    cust1.bookTickets();
    BookingApp cust2 = new BookingApp();
    cust2.bookTickets();
    BookingApp cust3 = new BookingApp();
    cust3.bookTickets();
    }
}
```

Type Casting :-



Type-Casting - 20/03/23

[P.N.Folio]

Conversion of one data type to another is known as typecasting.

Two types -

I) Primitive :-

↙ Converting Values

ⓐ Widening - Smaller data type to bigger
- No data loss

ⓑ Narrowing - Bigger data type to smaller
- Data Loss

II) Non-Primitive :-

↙ Converting Object/ Object-level Typecasting

ⓐ Upcasting - Object is creating for subclass with reference to superclass.
- to achieve data abstraction
- hiding implementation
- to only achieve parent class member/properties.

ⓑ Downcasting - converting superclass reference to subclass.

→ Without upcasting, downcasting can't be achieved.

↙ Beoz for downcasting parent-class reference is required.

Type Casting [Notes] -

Sood-Rhid

The process of converting one data type to another data type is known as typecasting.

There are two types of typecasting -

(1) Primitive TypeCasting
(11) Non-Primitive TypeCasting

1) Primitive TypeCasting -

(a) The process of converting one primitive data type to another primitive data type, is known as primitive typecasting.

(b) Primitive typecasting are of two types -
- Widening
- Narrowing

(c) → Widening ⇒
- The process of converting smaller ~~type~~ of data into bigger type of data, is known as widening.
- Widening is done implicitly or explicitly.
- The main advantage of widening process is, ~~here~~ will be no data loss.
- This widening process is also known as auto-widening.

(d) → Narrowing ⇒
- The process of converting bigger data type into smaller data type is known as narrowing.
- Narrowing is done explicitly.
- The main disadvantage of narrowing is data loss.
- Always narrowing is done explicitly by using typecast operator.

11) Non-Primitive Typecasting -

(a) The process of converting one non-primitive type to another non-primitive data type is known as non-primitive typecasting.

(b) Non-Primitive typecasting is also known as object-level typecasting.

(c) Non-Primitive typecasting are of two types -

1. Upcasting
2. Downcasting

(d) Upcasting -

→ Converting subclass type to superclass type is known as upcasting.

→ Object is created for subclass, but stored in reference of superclass.

→ Upcasting is done implicitly or explicitly.

→ The main advantage of upcasting is generalization (Accessing general properties from parent class).

→ Without doing downcasting we can do upcasting.

→ Once it is upcasted, we can't access child class properties, but we can access overridden methods of child class.

(e) Downcasting -

→ The process of converting superclass reference type to subclass reference type, is known as downcasting.

→ Downcasting is done explicitly by using typecast operator.

→ Without upcasting we can't achieve downcasting.

→ The main advantage of downcasting is specialization (we can access parent class properties as well as child class properties)

```
class Animal
{
  public void drink() {
        SOPln ("drinks water");
      }
}
class Dog extends Animal
{
    public void sound () {
          SOPln (" bow bow bow ... ");
        }
    public void eat () {
          SOPln (" veg & non-veg ");
        }
}
Class Lion extends Animal
{
      public void hunt () {
            SOPln (" Lion hunts ");
          }
      public void sound () {
            SOPln (" Lion roars ");
          }
}
```

```
public class AnimalApplication {
    PSVM (--){
        Animal A1 = new Animal();
        A1.drink();
        SOPln ("xxxxxxxxxxxxxxx");
Downcasting {
        Animal A = new Dog();
        A1.drink();
        Dog d = (Dog) A;
        d.drink();
        d.sound();
        d.eat();
        Lion L1 = (Lion) SOPln ("xxxxxxxxxx");
        Animal A2 = new Lion();
        A2.drink();
        Lion L1 = (Lion) A2;
        L1.drink();
        L2.hunt();
        L1.hunt();
        L1.sound();
    }
}
```

```
class Shape {
    public void shapeOfDiagram(){
        SOPln ("Shape");
    }
}
class circle extends Shape
{
    public void radius(){
```

```
            SOPln (" radius is given");
        }
    }
class Rectangle extends Shape
{
    public void length() {
            SOPln (" Length is given");
            }
    public void breadth(){
            SOPln (" Breadth is given");
            }
}
class Triangle extends Shape
{
    public void height () {
            SOPln (" Height is given");
            }
}
public class ShapeApp {
        PSVM(--) {
            Shape s1 = new Shape ();
            s1. ShapeOfDiagram ();
            SOPln (" * * * * * * * * * ");
            Shape s = new Circle();
            s1. ShapeOfDiagram();
            ~~Circle c1 = new s~~
            Circle c1 = (Circle) s;
```

```
        C1.ShapeOfDiagram();
        C1.Radius();
        SOPln("* ** *** **** *****");
        Shape S2 = new Rectangle();
        S2.ShapeOfDiagram();
        Shape
        Rectangle R1 = (Rectangle) S2;
        R1.ShapeOfDiagram();
        R1.length();
        R1.breadth();
    }
}
```

23/03/23

```
class Mechanic{
    public void service(Bike bike)
    {
    }
}
class Bike {

}
class Pulsar extends Bike
{

}
```

```
class Test{
    PSVM(--){
        Mechanic mech = new Mechanic();
        mech.service(new Pulsar());
    }
}
```

```
class Mechanic
{
 public void service(Bike bike){

 }
 public Bike Purchase(){
        return new Pulsar();
        }
```

```
class Bike {

    }
class Pulsar extends Bike
 {

 }
```

```
class Test{
    PSVM(--){
        Mechanic mech = new Mechanic();
        mech.service(new Pulsar());
        Bike bike = mech.purchase();
    }
}
```

Binding :-

→ If a method call is bound to the method defination, it is known as binding.

→ Binding are of two types -

i) Early binding
ii) Late binding

Early Binding -

i) Early binding is also known as static binding or compile-time binding.

ii) If a method call is bound to the method defination at compile-time by Javac (Compiler) based on the arguments is known as Early binding.

iii) Javac as a compiler will bind all the methods and give it to JVM for execution.

iv) In some context Javac doesn't bind certain reference and those are given JVM data.

v) Example of early binding -

i) Static methods
ii) Final methods
iii) Private methods
iv) Method overloading
v) Constructor overloading

Late Binding -.

i) If a method call is bound to the method defination at run-time by JVM (interpreter) based on the objects created is known as late binding.

ii) It is also known as dynamic binding or runtime binding.

28/03/23

Polymorphism :-

(i) Polymorphism is also one of the feature of OOPs.

(ii) It is derived from greek words, poly → many and morph → forms.

(iii) The ability of an object performing different tasks at different situations, is known as polymorphism.

(iv) The method call is ~~dowm t~~ bound to the method ~~dif~~ definition, is known as polymorphism.

(v) It is of two types-

a) Compile-Time Polymorphism

b) Run-Time Polymorphism

a) Compile-Time Polymorphism -

→ The binding is achieved at compile time, and the same behaviour ~~is~~ should execute at run-time, is known as compile-time polymorphism.

→ Compile-Time polymorphism is also known as static polymorphism.

→ Compile-Time Polymorphism is achieved by;

(i) Method Overloading

(ii) Constructor Overloading

~~(iii) Method hiding~~

b) Run-Time Polymorphism -

→ The binding is achieved at compile time and different behaviour is executed at run-time, is known as run-time polymorphism.

→ Run-Time polymorphism is also known as dynamic polymorphism.

→ Run-Time polymorphism is achieved by using method overriding.

Method Hiding -

→ In Super Class and Sub Class, if it is having same static method name with same signature then we can call it as method hiding.

→ Whenever we want to ~~act~~ do method hiding, Is-A relationship is mandatory.

```
class A {
    static int i=10;
    public static void m1()
    {
        SOPln("hi");
    }
}
Class B extends A {
        static int i= 20;
        public static void m1() {
                SOPln("hello");
        }
}
class C extends B {
        static int i= 30;
        p s v m1(){
                SOPln("Gd Afternoon");
        }
}
```

```
public class Demo {
    p s vm (--) {
        A a1 = new A();
        a1.m1(); // hello
        A a2 = new B();
        a1.m1(); // hello
        A a3 = new C();
        a3.m1(); // hello
        SOPln(a1.i); // i 10
    }
}
```

29/03/23

Compile-Time Polymorphism :-

=

① The call to the overloaded methods is resolved at compile time by Javac (Java Compiler) based on the type or reference of the class, is known as compile-time polymorphism.

② Ex- Overloading

- Method Overloading
  - Static
  - Non-static
- Constructor Overloading

Run-Time Polymorphism :-

=

① The call to overridden methods is resolved at run-time by Java interpreter (JVM) based on the objects is known as run-time polymorphism.

② Ex- Overriding

Method Hiding :-

=

① If two static methods in a context of overriding with same names, its not method overriding, instead it is known as method hiding.

② The sub class static method is hidden from JVM.

★ Upcasting :

→ Converting sub class type to super class type is known as upcasting.

→ Creating the object of sub class and storing in the reference of super class is known as upcasting.

→ Also known as Generalization or Object Level Widening.

→ It is implicit and also explicit.

→ Without downcasting, upcasting can be achieved.

```
class A
{
}
class B extends A
{
}
A a = (A) new B();
      or
A a = new B();
```

→ While upcasting we can access :

- Generic properties of super class.
- overriding methods of sub class.

→ We cannot access specific properties of sub class.

★ Downcasting :

→ Converting super class type to sub class type is known as downcasting.

→ Creating a reference of subclass and pointing it to the upcasted reference is known as downcasting.

-> Also known a specialization or Object Level Narrowing.

-> It is always explicit.

-> Cannot be achieved without upcasting.

```
class A {
    }
class B extends A {
        }
B b = (B) a;
b.sm();
((B)a).sm();
```

-> Downcasting is achieved to access the specific properties of sub class and inherited properties of sub class.

30/03/23

Concrete Methods -

1) A method which consists of implementation is known as concrete method.

Abstract Method -

1) A method which has only declaration but not defination / implementation is known as abstract method.

11) Abstract keyword is used.

⇒ A class which consists of only concrete methods is known as Concrete Class.

→ Concrete classes can be instantiated.

⇒ A class which consists of atleast one abstract method, is known as Abstract Class.

→ Abstract classes are used to achieve data abstraction.

→ Abstract keyword is used.

→ Abstract class consists of both concrete methods and abstract methods.

→ Abstract class cannot be instantiated.

→ Abstract class is a incomplete class.

→ The abstract methods of the abstract class should be mandatorily overriden (implemented) by the subclass.

→ If the subclass does not override all the abstract methods of the abstract class, then the subclass becomes abstract class.

Test.java

```
abstract class A {
        void m1()
        {
            sopln ("m1() defined for class A");
        }
        abstract void m2();
        abstract void m3();
    }
Class B extends A
{
```

```
@override
void m2(){
    SOPln("m2() defined for class B");
}
@override
void m3(){
    SOPln("m3() defined for class B");
}
void m4(){
    SOPln("m4() defined for class B");
}
}
public class TestAbstract{
    P S V M (--){
        A a = new B();
        a.m1();
        a.m2();
        a.m3();
        B b = (B) a;
        b.m4();
    }
}
```

or,

```
abstract class B extends A{
    @override
    void m2(){
```

```
        sopln ("m2() defined");
    }
    void m4()
    {
        sopln ("m4() defined");
    }

class C extends B
{
    void m3(){
        sopln ("m3() defined");
    }
    void m5(){
        sopln ("m5() defined");
    }
}

Public classTestA {
    Public static void Main (String [ ]Args){
        B b = new C();
        b.m1();
        b.m2();
        b.m3();
        b.m4();

        C c = (C) b;

        c.m3();
        c.m5();
```

1Q) Can we have static data members in abstract class?

2Q) Can we have non-static data members in abstract class?

3Q) Can we have blocks (non-static and static) " " "?

4Q) Can we have final variables " " "?

5Q) " " constructors " " "?

6Q) " " private methods " " "?

7Q) " " static methods " " "?

8Q) " " final methods " " "?

9Q) Can we declare abstract method private?

10Q) " " " " final?

11Q) " " " " static?

Ans

A1) Yes

A2) Yes

A3) Static, Non-Static - Yes

A4) Yes

A5) No/Y

A6) No/Y

A7) Yes

A8) Yes

A9) No

A10) No

A11) No

3/4/23

Q) Why Abstract class cannot achieve 100% abstraction?

A) - Abstract class contains/~~both~~ consists of abstract methods and also concrete methods.

- When the no. of concrete methods increases in abstract class abstraction decreases.

- when the no. of abstract methods increases in abstract class abstraction increases.

Note: To achieve 100% abstraction, interfaces are introduced.

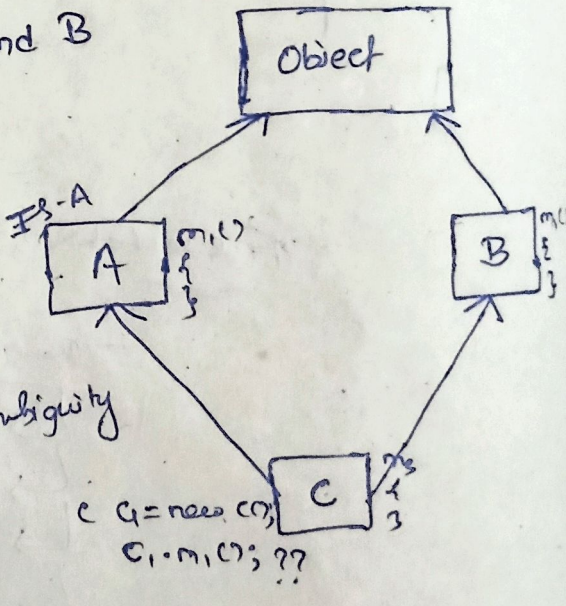
Q) Why multiple-inheritance cannot be achieved through classes in java?

A) - According to the diagram, class A and B are extended by the child class C.

- When C class object is created, construction is called which internally calls, call to super().

- Since, child class C has two parent A and B, JVM doesn't know which construction to call.

- This creates a confusion known as Ambiguity Error.



- Since, class A and class B are extending from object class, this creates a hybrid inheritance.

- Since, Hybrid Inheritance consists of multiple inheritance, this also creates Ambiguity Error and the problem is known as

diamond problem.
- To solve this problem in Java, interface is introduced.

Tight Coupling :-
- A class extending from another class achieves tight coupling.
- Implementation of the super class will (change the implementation) affect (of) the sub class when extended in the sub class.

Loose Coupling :-
- A class implementing an interface achieves loose coupling.
- The sub class (implementation class) is not affected by the interface.

Interface :-
→ Interface is used to achieve 100% abstraction.
→ Interface is used to achieve loose coupling.
→ Interface is used to achieve multiple inheritance.

Defination : It is an interface between service provider and end user.
→ Interface consists of static constants and abstract methods.
→ The data members of an interface is by default, public static and final.

→ The methods of an interface is by default public and abstract.

→ Interfaces cannot be instantiated.

→ "Interface" keyword is used.

→ The class which provides implementation for the abstract methods of an interface is known as implementation class.

→ "Implements" keyword is used.

→ An implementation class can implement from more than one interface but can extend from only one class.

→ An implementation class must override all the abstract methods of an interface, if it does not override the implementation class will become abstract.

→ From Java 1.8 version, interface consists of default methods and static methods (concrete methods) and also functional interfaces.

05/04/23

```
Interface Iter
{
    public static final double PI = 3.14;
    public abstract void m1();
    public abstract void m2();

}
```

→ One interface can be extended by other interface.

```
interface Iter
{
    double PI = 3.14;
    void m1();
    void m2();
    default void default_method()
    {
        SOPln("Im a default method of interface");
    }
    static void staticmethod()
    {
        SOPln("Im a static method of interface");
    }
}
class B implements Iter
{
    @override
    public void m1()
    {
        SOPln("m1() defined");
    }
    @override
    public void m2()
    {
        SOPln("m2() defined");
    }
    public void m3() // Sm
    {
```

```
        SOPln ("m3() defined");
    }
}
class TestInterface {
    PS VM (--) {
        Iter It = new B();
        It.m1();
        It.m2();
        It.defaultMethod();
        It.del.
        It.staticMethod();
        ((B)iter).m3();
    }
}
```

Q) A = 5467

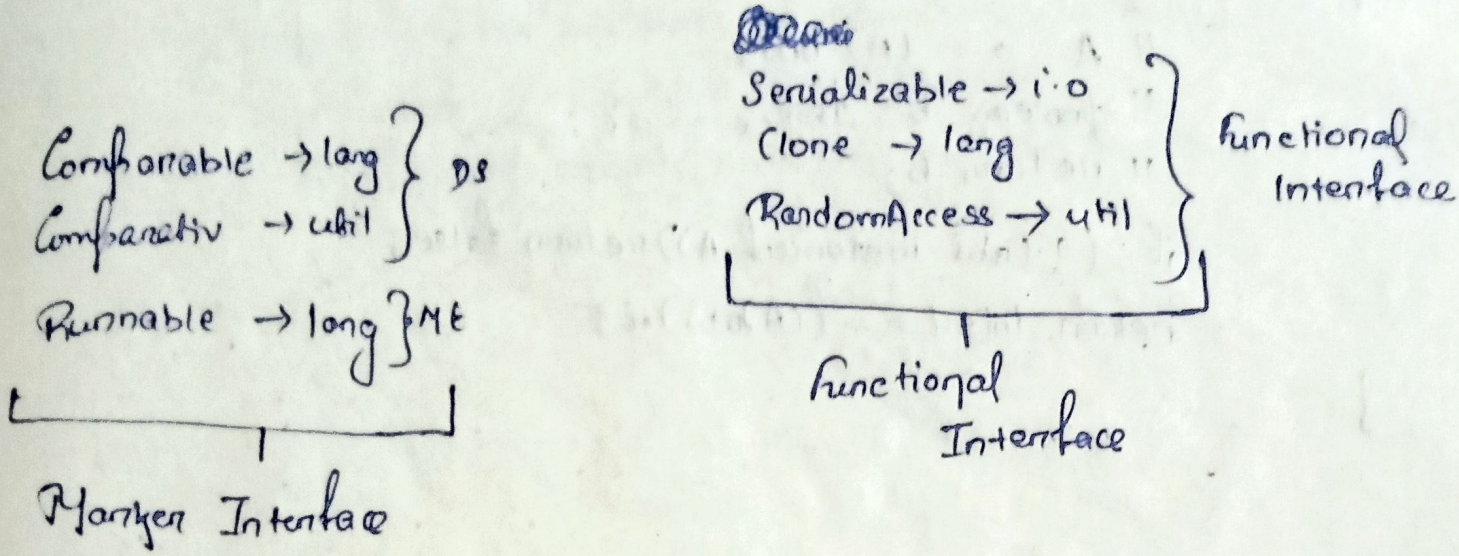
B = 22

A - 5 = B

B = 5445

Java 8 features

07/04/23



- Functional Interface -

- An interface which consists of atmost one abstract method is known as functional interface.
- Default methods & Static methods can be defined also inside functional interface.
- '@FunctionalInterface' annotation is used for functional interface available in java.lang package.

```
Class A
{
    int i;
    public A(int i){
            this.i;
    }
    @Override
    public String toString()
    { return "A[i=" + i + "]";
    }
```

```
public boolean equals (Object obj)
{
    // A a = (A) obj;
    // boolean b = this.i == a.i;
    // return b;
    if (!(obj instanceof A)) return false;
    return this.i == ((A)obj).i;
}
}

public class Test {
    public static void main (String [] args)
    {
        A a1 = new A(10);
        A a2 = new A(10);
        // B b1 = new B();
        SOPln(a1);
        SOPln(a2);
        SOPln(a1 == a2);
        SOPln(a1.equals(a2));
```

— x —

08/04/23

- **Wrapper Class :-**

→ Non-primitive versions of primitive data types.

→ There are 8 wrapper class present.

→ All wrapper classes are final classes present in lang package.

1) Byte
2) Short
3) Integer
4) Long
5) Float
6) Double
7) Char
8) Boolean

Auto-boxing =
Converting primitive type to non-primitive type is known as Auto-boxing.

```
Ex-   int a = 10;  //Primitive
      Integer i = new Integer(a);
      SOP(i); //10, Non-Primitive
```

Auto-Unboxing =
Converting non-primitive to primitive type is known as auto-unboxing.

```
Ex-  Integer a = new Integer(10);
     SOP(a); //10
     int b = a; // unboxing
     SOP(b); //10, Primitive
```

10/04/2023

Parse Methods -

- Parse Method is used to convert string to current type.
- It is a static method, by using class name we get the method.

Ex -

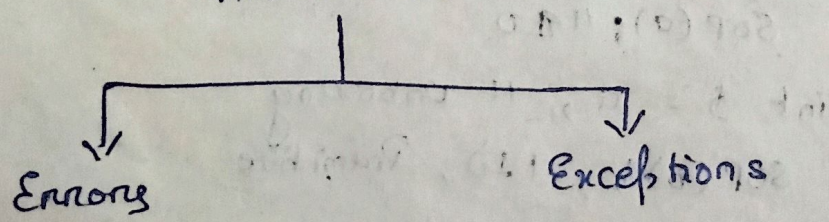
```
String s = "12.0";
SOP(s); // 12.0"
double n = Double.ParseDouble(s);
SOP(n); // 12.0
```

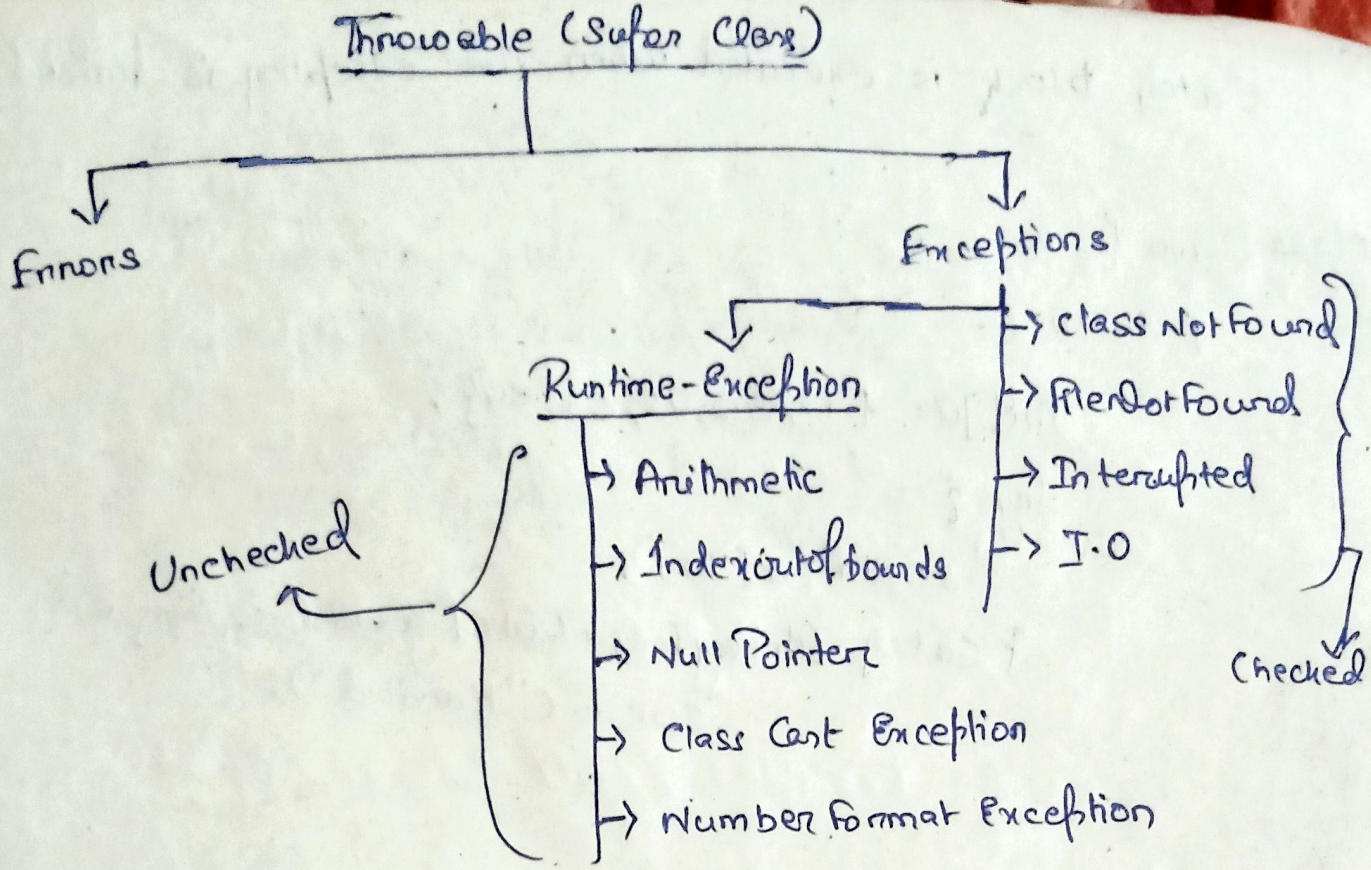
Ex -

```
SOP("main starts");
SOP(10/0); // Exception
           -->
           Arithmetic
SOP("main ends");
```

Exception Handling :-

- → Exception is an event which will occur during runtime.
- → Exception is also known as abnormal termination.
- → In Java, every exception is a pre-defined class.
- → when exception will occur, the remaining lines of code will not execute.
- → we can use try, & catch block, handle the exceptions.
- → We can't handle errors, we have to debug it.





11/04/23

```
public class Demo2{
    psvm(--){
        SOPln("Main Start");
        try{
            SOPln(10/0); // AE a = new AE();
        }catch(ArithmeticException e)
        {
            SOPln(e);
            SOPln("Handled ---:)");
        }
        SOPln("Main End");
    }
}
```

Catch block is executed when the exception is raised.

```
class Demo {
    p s v m (--) {
        int[]a= {12,80,45,67,84};
        try {
            sOPln (a[40]);
        } catch (ArrayIndexOutOfBoundsException e){
                    sOPln ("Handled");
        }
    },
}
```

```
class Demo {
    p s vm (--) {
        String s = null;
        try {
            sOPln (s.length());
        } catch (nUllPointerException e) {
                    sOPln ("Handled");
        }
    }
}
```

```
main {
    String s = null;
    try {
        sopln(s.length());
    } catch(NullPointerException e)
    {
        s = "Ram";
        SOPln(s.length());
    }
    SOPln(s);
}
}
```

```
public class A {
    ps vm (--) {
        String s = "Ram";
        int n = 0;
        try {
            n = Integer.parseInt(s);
            SOPln("Hello");
        } catch (ArrayIndexOutOfBoundsException e) {
            .SOPln(e);
        } catch (ArithmeticException e) {
            SOPln(e);
        } catch (NumberFormatException e) {
            SOPln(e);
        }
        SOPln(n);
    }
}
```

12/04/23

- Unchecked Exceptions -

- The classes which are subclasses to runtime exception class, those exceptions we call it as unchecked exception.
- In Java, every exception is ~~pre-defined~~ a class.
- Unchecked exceptions are also known as compiler unknown exceptions.
- Compiler is not forcing to handle unchecked exception.
- Example: IndexOfIndexOutOfBoundException.

- Checked Exception -

- The classes which are sub classes to exception class directly, those are called checked exceptions.
- checked exceptions are also known as compiler known exceptions.
- Compiler is forcing to handle checked exceptions.
- Example: ClassNotFoundException, InterruptedException
- Handling or propagation for a checked exception is mandatory.

Ex
```
public class Test1
{
    psvm(--){
        try {
            SOPln(10/2);
            String s = "Ram";
            int n = Integer.ParseInt(s);
```

```
            int [] a = {1,2,3,4,5,6};
            SOPln (a[100]);
        } catch (ArithmeticException e){
                SOPln ("Arithmetic");
        } catch (ArrayOutOfBoundsException e){
                SOPln ("Array Index");
        } catch (RuntimeException e) {
                SOPln ("Exception");
        }
    }
}
```

—X—

```
public class Test1 {
    p s v m (--) {
        try {
            int [] a = {23,4,5,7,8,9};
            SOPln (a[-1]);
        } catch (Exception e) {
            SOPln (e);
            SOPln (e.getMessage());
            e.printStackTrace();
        }
    }
}
```

```
{
  psvm(--){
        sopln("Main open");
        try{
          sopln(10/2);
        }catch(ArithmeticException e){
              sopln(e);
        }finally{
              sopln("Main close");
        }
  }
}
```

→ 'finally' block is a block of code which will execute every time.
→ Before executing return statement, finally block gets executed.
→ We can write, try with finally also.

14/04/23

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Test2{
      p.s v m(--){
            try{
            FileReader fr = new FileReader("Address");
            }catch(FileNotFoundException e){
                  sopln("Executed");
            }
      }
}
```

```
try {
    FileReader n = new FileReader ("Address/Path");
    try {
        int n = n.read();
        while (n!=-1) {
                SOP ((char)n);
                n = n.read();
            }
    } catch (IOException e) {
            }
    } catch (FileNotFoundException e) {

    }
}
}
```

```
public class Demo5 {
    psvm(--){
        SOPln ("Hi");
        try{
            Thread.sleep(800);
          }catch (InterruptedException e) {
          }
        try{ SOPln("Are you mad?");
        try {
            Thread.sleep(800);
          } catch (InterruptedException e) {
          }
        SOPln ("Get lost");
    } }
```

⇒ For checked exceptions, compiler will check user provided handlers or not.

⇒ For checked exceptions, handling is mandatory.

⇒ If we not handle checked expression, compiler will throw compile-time error.

```
public class Demo5 {
        static void m1(){
                SOPln(10/0);
        }
        psvm(--){
            try{
                m1();
            } catch(ArithmeticException e) {
                        SOPln(e);
            }
        }
    }
```

—X—

```
public class Demo5 {
        static void m1() throws InterruptedException {
                SOPln("Java is My first Language");
                Thread.sleep(800);
                SOPln("That's why I choose Java");
        }
        psvm(--){
            try{
                m1();
            } catch(InterruptedException e) {
            }
        }
}
```

Exception e
or,

→ 'Throws' is a keyword, used to declare an exception.

⇒ Propagation :-

Declaring unhandled exception by using throws keyword, is known as exception propagation.

```
public class Test2{
        psvm(--) throws FileNotFoundException{
            FileReader fr = new FileReader("Path");
            SOPln("Hi");
        }
}
```

15/04/23

```
import java.io FileNotFoundException
import java.io.FileReader

public class Test2 {
        psvm(--){
            FileReader fr = new FileReader("WrongPath");
            SOPln("Hi");
        }
}
```

'Throws'

① It is a keyword used to declare an exception.

② By using Throws keyword we can declare for checked exception mostly.

③ By using Throws keyword we can declare more than one exception.

④ We are using throws keyword with method signature.

'Throw'

① It is a keyword used to throw an exception.

② By using throw, we can throw unchecked and checked exception.

③ By using throw, we can throw only once a exception at time.

④ We use throw keyword, inside the method.

Note: It is not compulsory to give value for exception but it is good if we give value.

User Castwise Exception - Create Custom Exception

```
public class Gmail {
    psvm (--) {
        String username = "R@m123";
        int pswd = 54321;
        if (username.equals("R@m123")){
            { if (pswd== 54321){
                SOPln ("Login Done...");
            }
        else {
            try {
                throw new WrongPswdException();
```

```
            } catch (WrongPswdException e) {
                    SOPln ("Entered Pswd Wrong");
            }
        }
    }else {
        try {
            throw new WrongNameException ();
        }catch (Exception e){
                SOPln ("Entered name Wrong");
        }
    }
```

→ 2 types of exception - User-defined & Pre-defined.
→ getmessage method belongs to throwable class.

```
public class AgeInvalidException extends RuntimeException {
        String s;
    public AgeInvalidException (String s) {
            super ();
            this.s = s;
        }
    @override
    public String getMessage () {
            return s;
        }
}
```

```
public class MatriMony {
    ps vm (--) {
        int age = 46;
        if (age >= 25) {
            if (age <= 35) {
                SOPln ("Ready To Marriage ...");
            } else {
                SOPln (" Too Late :( ");
            }
        } else {
            try {
                throw new AgeInvalidException ("You have time");
            } catch (AgeInvalidException e) {
                SOPln (e.getMessage());
            }
        }
    }
}
```

Note = Every default is no argument constructor but every no argument constructor is not default constructor.

```
public class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException() {
        super ("Invalid Balance");
    }
```

```
    public InsufficientBalanceException(String s) {
        super(s);
    }
}
```

```
public class Account {
    private int bal;
    public Account(int bal) {
        super();
        this.bal = bal;
    }
    public int getBal() {
        return bal;
    }
    void withdraw(int amount) throws InsufficientBalanceExcep
    {
        if (amount > bal) {
            throw new InsufficientBalanceException("LowBal");
        }
        bal -= amount;
    }
}
```

```
public class AccountDriver {
    psvm (--) {
        Account a = new Account(1000);
        int amount = 500;
        try {
            a.withdraw(amount);
            SOPln("withdraw Done :" + amount);
            sopln("Bal is :" + a.getBal());
        } catch (InsufficientBalanceException e) {
            SOPln(e.getMessage());
        }
    }
}
```

17/04/23

- **Recursion :-**

```
public class Test1 {
    psvm (--) {
        SOPln("Main Start");
        SOPln(m1(5));
        SOPln("Main Ends");
    }
    public static int m1(int i) {
        SOPln(i++);
        int j = m2(i);
        SOPln(i);
        SOPln(j);
        return j;
    }
```

```
public static int m2 (int j) {
        SOP (i++);
        int j= m3 (i);
        SOP (i);
        SOP (j);
        return j;
    }
public static int m3 (int i) {
        SOP (i);
        return i;
    }
```

— x —

① Sum of number = 12345

```
SOP ( Sum (12345));
        ↓
    5+ Sum (1234)
        ↓
    4+ Sum (123)
        ↓
    3+ Sum (12)
        ↓
    2+ Sum (1)
        ↓
    1 + Sum (0)
```

```
int Sum (int n)
    {
        if (n==0) return 0;
        return n%10 + sum (n/10);
                5 + sum (1234);
```

```
Sum (1234)
if (1234==0)
  4+ Sum (123)
```

Sum (123)
123 == 0
3 + Sum (12)

Sum (12)
2 + sum(1)

Sum (1)
1 + Sum (0)

```
Public class num {
    p s vm (--) {
        m1 (10,1);
    }
    public static void m1 (int i, int j) {
        if (i<j)
            return ;
        sop(i);
        m1 (--i,j);
    }
}
```

(2) Sum of numbers until get single digit.

```
public class Num {
    p s vm (--) {
        sop( sum(12345));
    }
```

5) Factorial Program :-

```
static int fact (int n) {
        if (n==0) return 1;
        return n* fact (n-1);
    }
public class Test
{
    psvm (--) {
            sop (fact (5));
    }
    public static int fact (int n)
    {   if (n==0)
            return 1;
        return n * fact (n-1);
    }
}
```

fact (5)
↓
5 * fact (4)
↓
4 * fact (3)
↓
3 * fact (2)
↓
2 * fact (1)
↓
1 * fact (0)

6) Strongnum :-

```
public class Strongnum {
    psvm (--) {
        int n = 40586;
        if (n == isStrong (n))
            sop ("strong No");
        else
            sop ("Not a strong Num");
    }
```

Strong (145)
↓
5! + Strong (14)
↓
4! + Strong (1)
↓
1! + Strong (0)

```
public static int isStrong(int n) {
        if (n==0)
            return 0;
        return fact(n%10) + isStrong(n/10);
    }
public static int fact(int n){
        if (n==0)
            return 1;
        return n*fact(n-1);
    }
```

⑤ Armstrong Num :-

=

```
public class Armstrong {
    psvm(--){
        int n=153;
        if (n== isArmstrong(n, count(n))
            sop("Armstrong");
        else
            sop("Not a Armstrong");
    }
    public static int count(int n){
        if (n==0)
            return 0;
        return 1+ count(n/10);
    }
```

```
public static int isArmstrong (int n, int pow) {
        if (n==0)
            return 0;
        return;
    }
```

19/04/23

Palindrome number :-

```
public class Palindrome {
    public static void main (String[] args) {
        int n = 151;
        if (n == rev(n,0) sop("Palindrome");
        else sop("not Palindrome");
    }
    public static int rev (int n, int temp)
    {
        if (n==0)
            return sum;
        temp = (temp*10) + n%10;
        return rev(n/10, temp);
    }
```

→ Why we do multithreading?

↓↑

To achieve multi-tasking

Imp - ① Exception
② Multi-Threading
③ File-Handling

→ Once a thread is started, thread will execute simultaneously

→ Threads are independent.

→ In process based multi-tasking, we get separate memory for each task, but in thread based we get one memory for multiple task.

Java (JVM) → Main Thread → main() } Execution

Thread Class → Having one name for Thread (Java.lang).

⇒ Multi-tasking = ↓ Multiple task executing simultaneously.

It is divided into 2 types -

i) Process based multi-tasking
ii) Thread based multi tasking

① Multiple processes executing simultaneously is known as process based multi-tasking.

Multitasking

(i) - Process is nothing but a task.
(ii) - It is a heavy weight process.
(iii) - It is used in OS level.

Thread Based Multitasking =

① Multiple threads are executed at same time is known as thread based multi tasking.

(i) It is a light weight sub-process.
(ii) It is used at programatic level.

(iv) Thread based multitasking means Main threading.

→ JVM creates 3 threads-

1) Main Thread
2) Thread Scheduler Thread
3) Garbage Collector Thread

→ Main method is executed by Main Thread.

→ For every thread they have priority between 1-10.

→ Every thread execution is based on (5) default (1) min (10) max

20/04/23

→ We use multithreading in animation.

→ In Thread is a program.

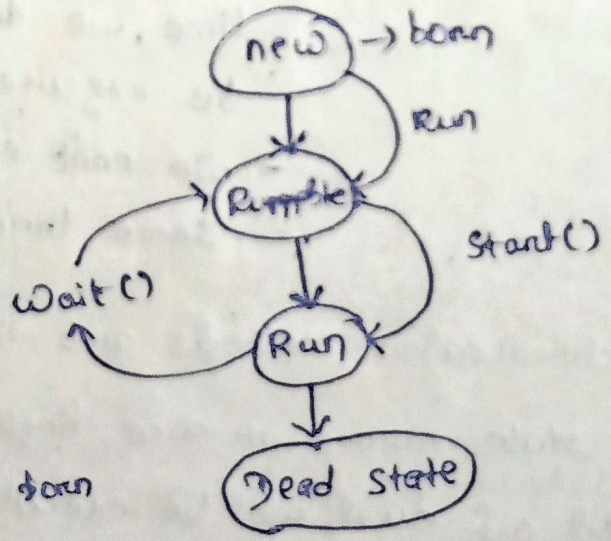
→ When we override Run method, at that time thread is runnable.

→ When we call start(), then thread is changed to Run.

→ In thread, every time we get different O/P.

→ Main thread priority (5).

→ Thread runnable decides which method is executed.



new Thread born

```
T1 t = new T1()
    t.start();
```

- The first time we create an object for thread, at that time new object is born.

→ If we want to create a thread, first we have to create a class.

→ Then we have to extend Thread class for making thread.

→ when we override run() in child class, then thread is in runnable state. we then call start(), at that time thread is in Run state. After program execution thread is in dead state.

⇒ Main Thread Priority -

Threads are executed based on priority.

- default thread priority is 5.
- child thread will get priority by parent thread (Means child class extends thread class).
- All threads are executed at the same time, we dont know which thread will be executed first.
- We can't call start() 2 times, by using same thread reference.

→ All synchronization methods are thread state.

→ Thread state means at once first one ~~three~~ thread is executed and dual will be executed.

⇒ Synchronization -

Synchronization is a mechanism to avoid thread race condition.

Multiple Threads trying to access same object at same point.

→ we can achieve Synchronization by using keyword called 'synchronized'.

→ Synchronized keyword is used for method & blocks.

→ we can avoid data inconsistency by synchronization.

⇒ Object Level lock :-

- when we are creating non-static method as synchronizing method internally object level lock is created.
- Every object has one unique lock (same as class name).
- After completion of execution, Thread releases the lock.

Eg)
```
class A {
    public synchronized void m1(String s)
    {
        for (int i = 1; i<= 5 ;i++){
                    sop ("m1 executed");
```

=> Class level Lock :-

=

- When we are creating static method as Synchronized method, internally class level lock is created.

- Class level lock is unique.

26/04/23

FileWriter ("Path")

- To write the data
- in I:O ~~sta~~ package

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class WritingFile {
    public static void main (String[] args) throws IOException{
        File f = new File ("C:\\ . . . . ");
        FileWriter file = new FileWriter(f);
        if (f.canWrite()) {
            file.write("Jafidens...\n");
            file.write("qafidens...\n");
            char[] ch = {'a','y','v','a','u'};
            file.write(ch);
            file.close();
        } else {
            sop\n ("we can't write");
        }
    }
}
```

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class WritingFile2 {
        public static void main(String[] args) throws Exception{
                File f = new File(" Path -- ");
                FileOutputStream fo = new FileOutputStream(f);
                if (f.canWrite()){
                        String s = " nfanatheli";
                        byte[] b = s.getBytes();
                        fo.write(b);
                }else{
                        sopln (" cant write");
                }
                fo.close();
        }
}
```