Inheritence

client-1

class Cricket

- name
- age
- matchesPlayed
- runs
- avg
- country
- no of 6's
- no of 50's
- no of 100's

class Football

- name
- age
- matchesPlayed
- country
- goalsScored
- positions
- no of cards
- of side
- tackles

def: mechanism in which one object acquires the behaviors and properties of the parent object

- idea - create new classes that are built upon existing classes.

client-2

class Player

- name
- age
- matchesPlayed
- country

class Cricket

- runs
- wickets
- avg
- no of 6's
- no of 100's
- no of 50's
- catches

class Football

- goalsScored
- positions
- no of cards
- of side
- tackles

def^n: It is a mechanism where one object acquires the features [Properties (fields) & behavior (methods)]

Single Inheritance

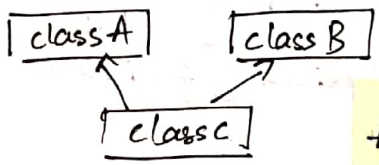
single- subclass inheriting the properties & behaviours of superclass

multilevel: a class inherits P&B of another class which again inherits P&B of a different class

ss/
ss/base
ed
s/derived
lass

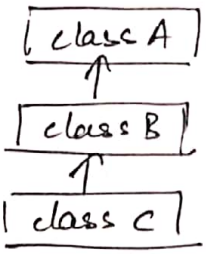
```
class B extends A {
  =
  =
}
```

Multiple [not possible in java]



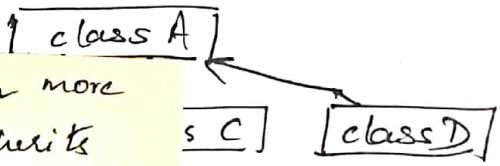

```
class A {
  =
}
class B {
  =
}
class C extends A, B {
  =
  =
}
```

Multilevel




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

Hierarchical

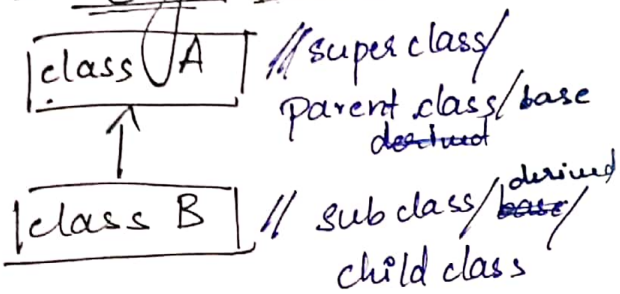


Hierarchical: when more than one class inherits P&B of the parent class

```
A {
  =
}
class C extends A {
  =
}
class D extends A {
  =
}
```

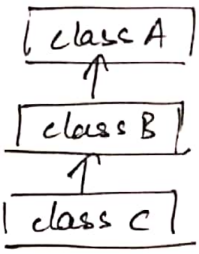
def^n: It is a mechanism where one object acquires the features [Properties (fields) & behavior (methods)]

Single Inheritance



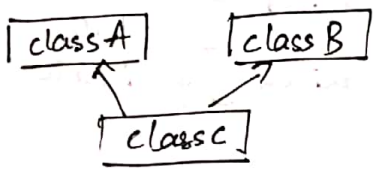

```
class A {
  ==
}

class B extends A {
  ==
}
```

Multilevel



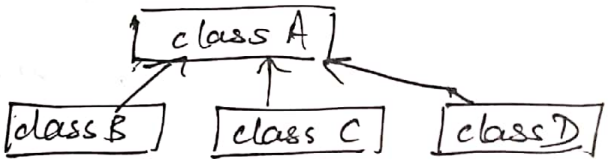

```
class A {
  ==
}
class B extends A {
  ==
}
class C extends A {
  ==
}
```

Multiple [not possible in java]




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

Hierarchical




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

Rules of Inheritence

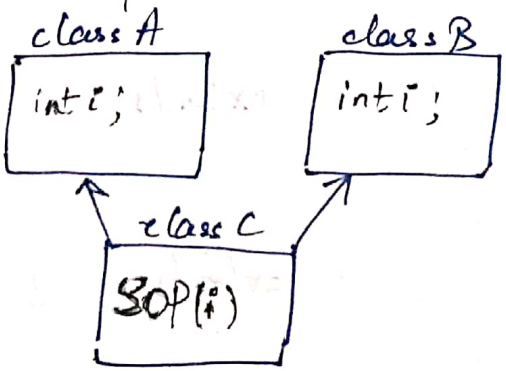
RULE-1: Private members doesn't participate in inheritence (properties & behavior)

```
class Base {
    private int x=10;
    int y=20;
    private void display(){
        SOPln("parent class");
    }
}

public class Derived extends Base{
    psvm (String[] args){
    Derived derive = new Derived();
    SOPln (derive.y); // 20
    SOPln (derive.x); // cannot be inherited } coz its private
    derive.display(); // cannot be inherited }
}
```

[Const. cannot be inherited but can be called]

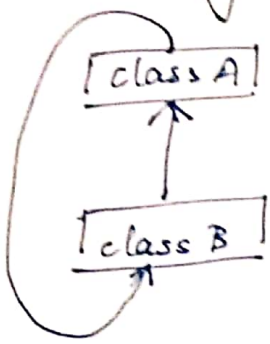
RULE-2: Constructor doesn't participate in inheritence.

RULE-3: Multiple Inheritence is not allowed in java



- There will be an ambigious situation for the above case Because variable 'i' is present in class A as well as class B.

RULE-4: Cyclic inheritance is not allowed.




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

X

```
class Base {
    int y = 20;
}
public class Derived extends Base{
  int y = 10;
  void displayY(){
   SOPln(y); //10
   SOPln(this.y); //10 Prints value available in same class.
   SOPln(super.y); //20 - super keyword is used to
                          extract the value from
                          parent class
  }
  Derived derive = new Derived();
  derive.displayY();
}
}
```

Note

Base class & derived class can have variables with the same name. We can overcome the naming conflict by using 'this' & 'super' keyword

'this' pointer to current class variable

'super' points to parent class variable

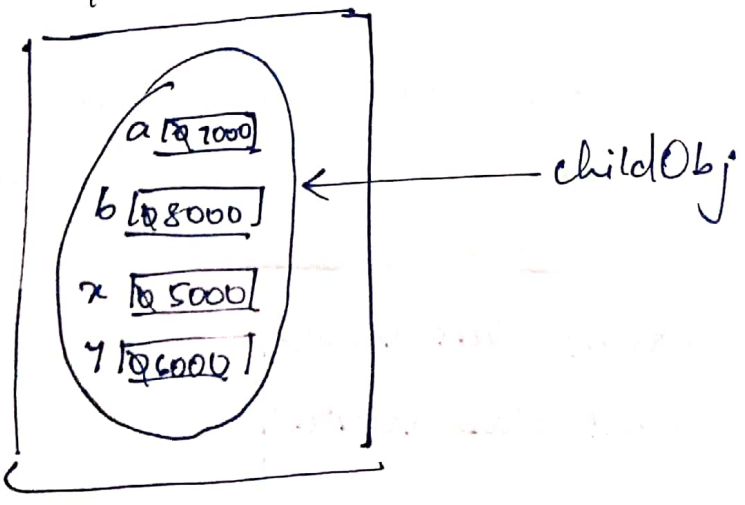
Parent & child constructor / Constructor Chaining

```
① class Parent {
    Parent() {
        SOP("Parent constructor");
    }
}
class Child extends Parent {
    Child() {
        super();
        SOP('Child constructor');
    }
    psvm(String[] args) {
        Child obj = new Child();
    }
}
```

super() is used to call the constructor of parent class. Even if not present by default super() is available

Whenever we create an object of child class the parent class constructor will be called by the super() call.

②



```
class Parent {
  int a, b;
  public Parent() {
    a = 10;
    b = 20;
    SOPln("Constructor-1 of parent");
  }
  Parent(int a, int b) {
  this.a = a;
  this.b = b;
  SOPln("Constructor-2 of parent");
}
}
class Child extends Parent {
  int x, y;
  public Child() {
    x = 999;
    y = 888;
    SOPln("Constructor-1 of child");
  }
public Child(int x, int y){
    this.x = x;
    this.y = y;
    SOPln("Constructor-2 of child");
  }
  public Child(int x, int y, int a, int b) {
  super(a, b); // instead of this if its just super()
  this.x = x;
  this.y = y;
  SOPln("Constructor-3 of child");
}
```

```
public static void main (String[] args ) {
    SOPln(" Main method start");
    Child childObj = new Child (5000, 6000, 7000, 8000);
  SOPln ( childObj.x);
  SOPln (childObj.y);
  SOPln (childObj.a);
  SOPln (childObj.b);
  SOPln(" main method stop");
 }
}
```

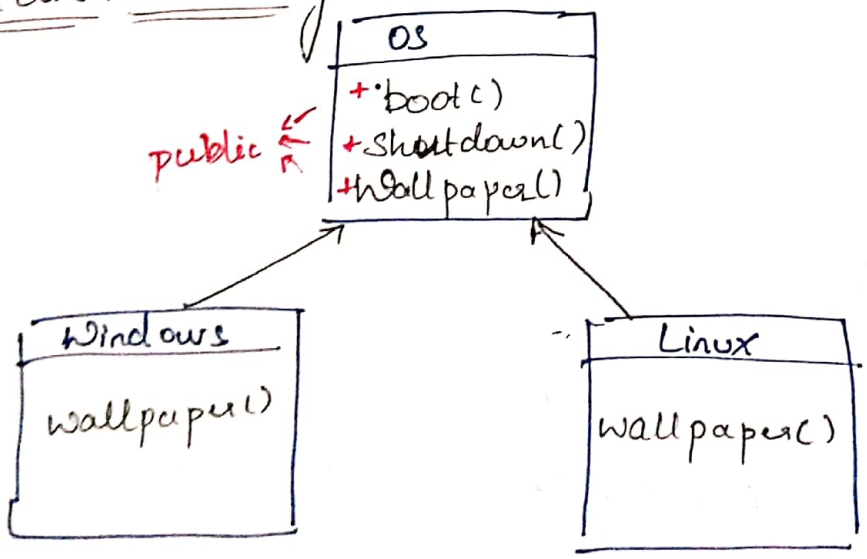
OUTPUT

Main method start
Constructor - 2 of parent
constructor - 3 of child
5000
6000
7000
8000
Main method stop

OUTPUT

Main method start
Constructor 1 of parent
Constructor 3 of child
10
20
10
20
Main method stop

Method Overriding



- Inherited methods are such methods where the body is defined in the parent class.
- Specialized methods are such methods where the body is defined in the same class
- Overridden methods are methods which is inherited from the parent class and the body is changed in the child class.

```
class OperatingSystem {
    public void boot() {
        SOPln("The system starts");
    }
    public void shutdown() {
        SOPln("The system ~~shown~~ shuts downs")
    }
    public void wallpaper() {
        SOPln("Blank wallpaper");
    }
}
class Windows extends OperatingSystem {
    public void wallpaper() {
        SOPln("Line wallpaper");
    }
}
class Linux extends OperatingSystem {
    public void wallpaper() {
        SOPln("Image");  → super.wallpaper();
    }
}
```

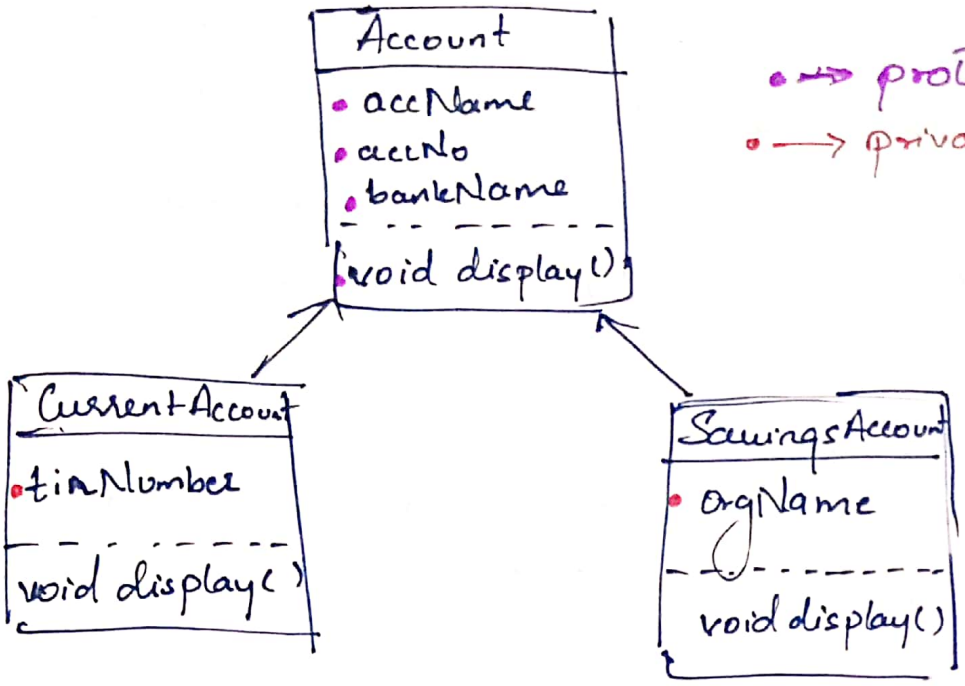
```
public class OS{
    public static void main(String[] args){
     Windows mysystem1 = new Windows();
     mysystem1.boot();
     mysystem1.wallpaper();
     mysystem1.shutdown();
     Linux mysystem2 = new Linux();
     mysystem2.boot();
     mysystem2.wallpaper();
     mysystem2.shutdown();
    }
}
```

OUTPUT

The system starts
Live wallpaper → overidden
The system shuts down

The system starts → blank wallpaper
Image → ouridden
The system shuts down

- From the above program the method wallpaper is ouridden in class 'Windows' and 'Linux'.



Object class :-

By default object class will be inherited by all the classes.

Methods in an object class:

1. clone() — creates copy of the object
2. equals() → compares if 2 objects are equal
3. getClass() → displays class name along with the package
4. hashCode() → returns int value (shows where the file is present)
5. ~~toString~~ → returns the address
6. ~~finalize~~ → executes by the garbage collection.
7. wait()
8. notify()
9. notifyAll()

'final' modifier:

- can be used for
  - ↳ variables
  - ↳ methods
  - ↳ classes

1. Final in case of variable:

If final is used in case of variable we can't change the value of the variable after assigning.

Ex: final int x = 10;
x = 100;

Ex: final int x = 5;
sop(x++)

2. For methods:

methods cannot be overridden. but can be inherited.

Ex: final void wallpaper() {
}

3. For class:

class cannot be inherited

Ex: final class Parent {
}

class Child extends Parent {
}

Rules of overridding :-

1. To override a method, signature & the method should be same as the parent class.
2. Modifiers cannot be changed while overriding the method

Order of accessibility

i private - can't be inherited
ii default - default, protected, public
iii protected - protected, public
iv public - public

less accessible ↓ more

3. If we have to change the modifier in the child class method it can be changed as given by the above table

4. We can't change the return type while overriding the method.

```
class Parent {
    protected void display() {
        SOPln("Holidays are over");
    }
}
class Child extends Parent {
    public int display() { // error - return type must be changed to void.
        SOPln("Back to class!!");
        return 10;
    }
}
```

5 But return type can be changed when the types are co-variant.

```
class Parent {
    protected Parent display() {
        SOPln("Parent");
        Parent p = new Parent();
        return p;
    }
}
class Child extends Parent {
    public Child display() {
        SOPln("child")
        Child c = new Child();
        return c
    }
}
```

co-variant

→ protected
- → private
+ → public

access modifiers : helps to restrict the scope of class, constructor, variable, method or data member

Non-Polymorphic Code:

```
import java.util.Scanner;
class OperatingSystem {
  public void boot() {
    SOPln("System booting");
  }
  public void shutDown(){
    SOPln("System shutting down");
  }
protected/public void writeCode(){
    SOPln("writing code in OS");
  }
}

class Windows extends OperatingSystem {
  public void writeCode(){
    SOPln("writing code in windows.");
  }
  public void cortana(){
    SOPln("Hi, how may I help you");
  }
}
```

```
class Mac extends OperatingSystem {
public void writeCode(){
  SOPln("Writing code in MAC!!!");
}
public void editVideo(){
  SOP("Allows to edit");
}
}

public class Main{
  p.s.v.m(String args[]){
    Scanner scan = new Scanner(System.in);
    SOPln("Enter your choice")
    SOPln("1. Windows\n2. Mac\n3. Linux\n");
    int choice = scan.nextInt();
    switch(choice){
      case1 : Windows window = new Windows();
              window.writeCode();
              break;
      case2 : Mac mac = new Mac();
              mac.writeCode();
              break;
      case3 : Linux linux = new Linux();
              linux.writeCode();
              break;
      default: SOPln("Enter the right choice")
    }
}
```

: : relation

```
class Linux extends OperatingSystem {
    public void writeCode(){
        SOPln("Writing code in Linux :> ");
    }
}
```

def: ability to display message in more than one form

Polymorphic code:

poly - many
morphs - forms

- Single action in different ways

```
OperatingSystem OS;
switch(choice) {
case 1: OS = new Windows(); // here parent reference is used to create different class objects.
        OS.writeCode();
        break;
case 2: OS = new MAC();
        OS.writeCode();
        break;
case 3: OS = new Linux();
        OS.writeCode();
        break;
default: SOPln("Enter valid choice");
}
```

(OR)

```
OperatingSystem OS = null; // OS must be initialized
switch(choice) {
case 1: OS = new Windows(); // upcasting
        break;
case 2: OS = new Mac();
        break;
case 3: OS = new Linux();
        break;
```

many relation

```
        default: SOPln("Enter valid choice");
    }
    os.writeCode();
```

```
OperatingSystem os = new Windows();
```

Upcasting ←

- Specialized methods cannot be called [Ex method cortana() cannot be called when os reference is used]
  - os.cortana();
- can be rectified as
  ((Windows) os).cortana(); // downcasting

run time polymorphism → method overridding
compile time polymorphism → method overloading

Parent → child ⇒ down casting
child → parent ⇒ up casting

Case 1: Non polymorphic code :->

We cannot achieve polymorphism in this prog coz there is 1:1 relation (tight coupling)
In other words, Since there is no 1: many relation we are not able to achieve polymorphism.

1:1 relation :- (tight coupling)

```
Linux li = new Linux();
Windows win = new Windows();
```

1: many relation :- (loose coupling)

```
OperatingSystem os
os = new Windows
os = new Linux
```

Case2: Polymorphic code:

In this program we can achieve polymorphism with the help of 1: many relation (loose coupling)

We can achieve loose coupling by pointing parent class reference to multiple child class objects.

disadv of having parent class reference to child class obj - loose coupling

- cant access specialized methods

We can overcome this prob by downcasting

Upcasting & Downcasting:

UPCASTING:- Assigning child class ~~reference~~ object to parent class reference is upcasting

Ex
```
OperatingSystem OS;
OS = new Windows();
OS = new Mac();
```

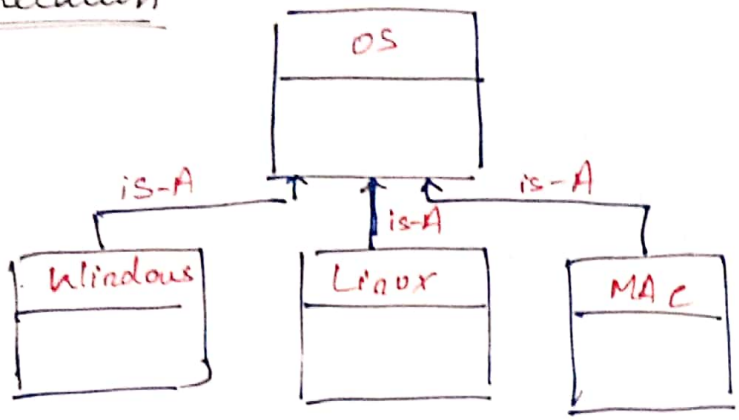
DOWNCASTING:- Process of converting the parent class reference to child class reference is downcasting.

```
Windows win = (
OperatingSystem OS : new Windows();
Windows win = ((Windows)os);  → downcasting
```

Note

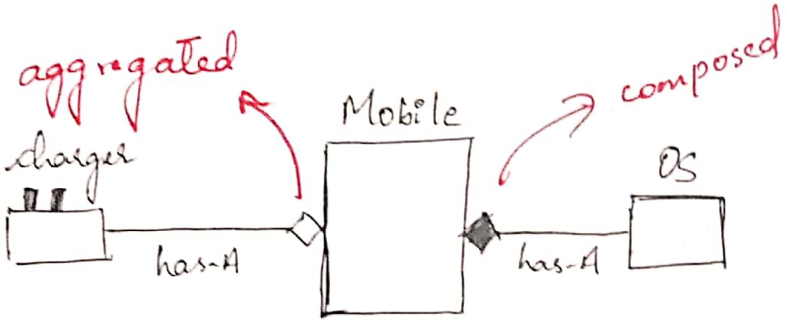
Parent child relation is considered as is-A relation

is-A relation



Delegation model:

- When the classes are related through the keyword extends then it is considered as 'is-a relation', but some relations cant be achieved through extends that is considered as 'has-a relation'
- To achieve has-a relation we can make use of delegation model

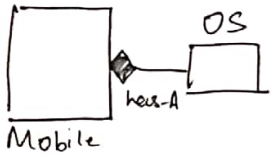


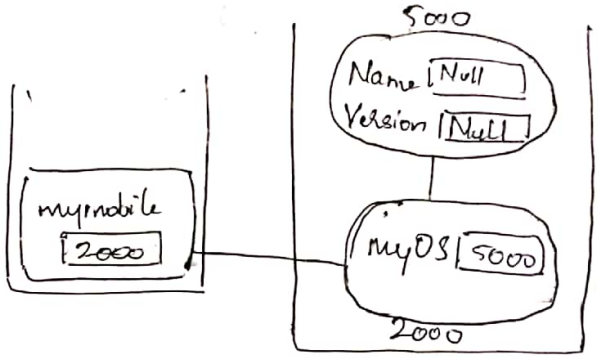
- Designation model consists of composed objects & aggregated object.
- Composed objects are such which gets destroyed when the main object is destroyed.
  Ex Mobile & operating s/m.
- Aggregated objects are such objects which continue to exists even after the main objects gets destroyed
  Ex Mobile & charger

Rules for implementing delegation model :

- All the composed objects must be declared as instance variable in main class
- The aggregated objects has to be passed as a parameter to the method of main class.

Composed Objects :





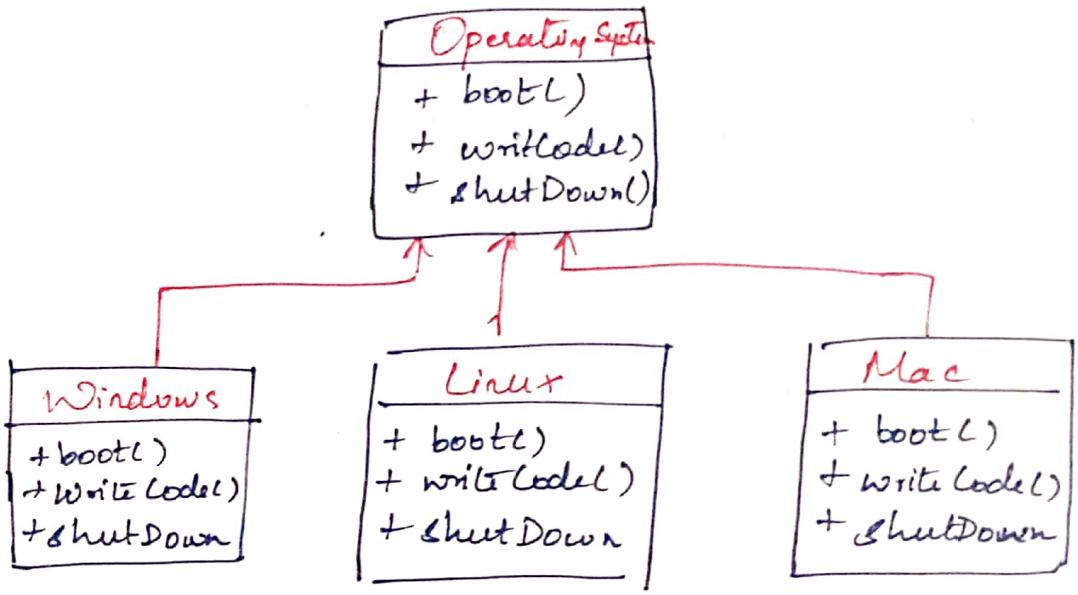

```
class OS {
        String Name;
        String Version;
    }
class Mobile{
        OS myOS;
        Mobile(){
          this.myOS = new OS()
        }
    }
class Alpha {
        p.s.v.m(String args[]){
            Mobile myMobile = new Mobile();
            myMobile.myOS.name = "android";
            myMobile.myOS.version = "pie";
            SOP(myMobile.myOS.name);
        }
}
```

~~Import~~ Factory Design Pattern:

```
class OperatingSystem {
    public void boot() {
        SOPln("System booting");
    }
    public void shutDown() {
        SOPln("system shutting down");
    }
    public void writeCode() {
        SOPln("Writing code");
    }
}
class Windows extends OperatingSystem {
    public void boot() {
        SOPln("System booting in windows");
    }
    public void writeCode() {
        SOPln("Writing code in windows");
    }
    public void shutDown() {
        SOPln("Shutting down in windows");
    }
}
class Linux extends OperatingSystem {
    public void boot() {
        SOPln("System booting in Linux");
    }
    public void shutDown() {
        SOPln("System shutting down in Linux");
    }
}
```

```
    public void writeCode() {
        SOPln("Writing code in Linux");
    }
}
class Mac extends OperatingSystem{
    public void boot(){
        SOPln("System booting in mac");
    }
    public void shutDown(){
        SOPln("System shutting down in mac");
    }
    public void writeCode(){
        SOPln("Writing code in mac");
    }
}
class Factory{
    OperatingSystem static getOS(String choice) {
        if(choice == null){
            return null;
        }
    else if(choice.equalsIgnoreCase("windows")){
        return new Windows();
    }
    else if(choice.equalsIgnoreCase("linux")){
        return new Linux();
    }
    else if(choice.equalsIgnoreCase("mac")){
        return new Mac();
    }
    else {
        return null;
    }
}
```

```
public class DesignPattern {
    public static void main(String args[]) {
        Scanner scan = new Scanner(System.in);
        SOPln("Windows\nLinux\nMac\n");
        SOPln("Enter your choice");
        OperatingSystem os = Factory.getOS(scan.next());
        if (os != null) {
            os.boot();
            os.writeCode();
            os.shutDown();
        }
        else {
            SOPln("Invalid Input");
        }
    }
}
```



Factory most used design pattern in java.
We create obj without exposing the creation logic to the client & refer to newly created obj

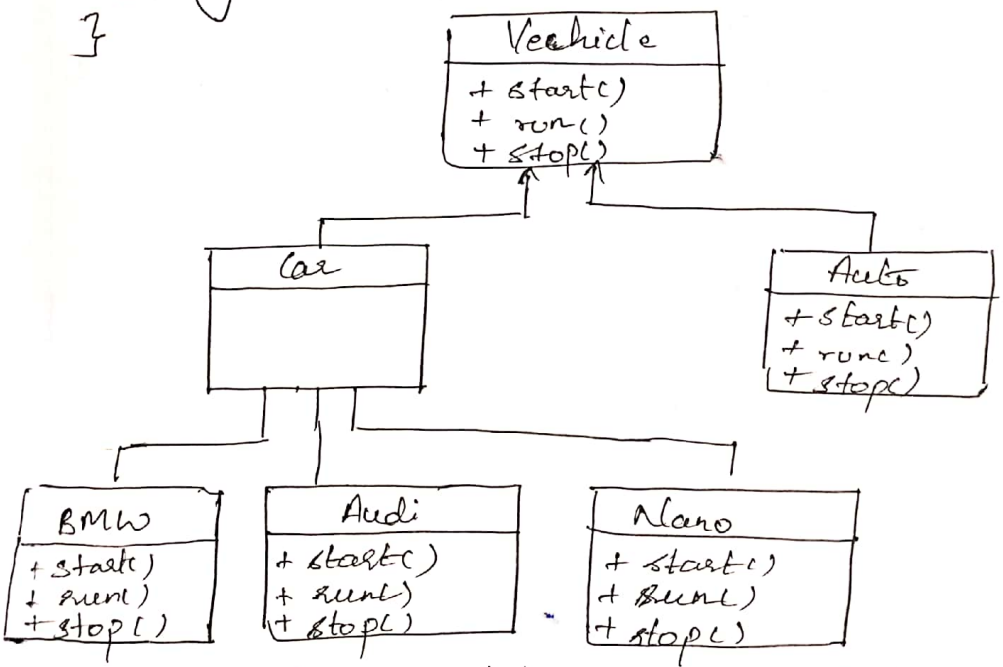
Abstract keyword:

When you dont want to provide any implementation to the method

```
abstract class Athlete {
    abstract void whatHeDoes();
}

class Swimmers extends Athlete {
    public void whatHeDoes(){
        SOPln ("He swims");
    }
}
class Sprinters extends Athlete {
    void whatHeDoes (){
        SOPln (" He runs");
    }
}
~~class~~ abstract class Shooters extends Athlete {

}

public class AbstractKeyword{
    p.s.v.m (String args[]) {
    Swimmer swim = new Swimmer();
    swim.whatHeDoes();
    Sprinters sprinters = new Sprinters();
    sprinter.whatHeDoes();
    new Athlete();
    new Shooters();
}
}
```

// cannot create the object bcoz the given class is 'abstract'

```
public class AutoMobile {
    p.s.v.m(String args[]) {
    SOPln(" BMW/Audi/Auto/Nano");
    SOPln("Enter your choice");
    Scanner in = new Scanner(System.in);
    String choice = scan.next();
Vehicle myAutoMobile = factory.getVehicle(choice);
 if (myAutoMobile != null) {
      myAutoMobile.start();
      myAutoMobile.run();
      myAutoMobile.stop();
  }
}
```




```
static vechicle getVehicle(String choice){
    if (choice == null) {
          return null;
    }else if (choice.equalsIgnoreCase("BMW")) {
        return new BMW();
    }else if (choice.equalsIgnoreCase("audi")){
        return new audi();
    }
    // for nano, auto ...
  }else { return null;
}}
```

Abstract is a keyword to create Abstract method & classes in Java.

Rules of Abstract

- Abstract methods has no body.
- If a class has a abstract method, it is compulsory to declare class as abstract.
- It is not compulsory to have abstract method inside abstract class.
- we cannot create a object of abstract class
- If a normal class inherits the abstract class it is compulsory to override the unimplemented methods of the parent abstract class & else we can make the child class as abstract.
- we can have reference of a abstract class
- we can have constructor in abstract class
- we can have concrete (non-abstract method inside the abstract class.

```
abstract class Parent{
    abstract void eat();
final void sleep(){
    SOP("sleeping");
}
class Child extends Parent{
    void eat(){
        SOP("eating");
}}
```

compulsory to be overridden by the child

not compulsory for child to override but if it shd'nt be overridden then use final

- we cannot write abstract and final together in one method.
- we cannot use final, static, private, default modifiers along with abstract.

Note: If we make the constructor as 'private' we can create object & that particular class only inside the class; outside the class you cant create the object.

```
class A {
    private A() {
        SOPln("Constructor-1");
    }
    A(int x){
        SOPln("constructor-2");
    }
}
public class MyExample{
    p.s.v.m(String args[]){
        A a = new A(5); // since its calling parameterized constructor
    }
}
```

Cant create the object but can call parameterized constructor

- You can make the constructor as final nor abstract.
- You can make a method as static only if there are no instance variables in that particular method
- Abstraction – hiding the functionality from the user.
- compile time / static / false / static binding / early binding → method overloading
- run time polymorphism → method overridding late binding

- In Method overriding we can achieve run time polymorphism, also called as actual / true / late binding / dynamic binding polymorphism
- Can static method be overridden? Method Hiding
  Ans: No
  If the reference is of Parent, parent method will be called & if the reference is of child, child method will be called

```
class Parent {
    static void sleep() {
        SOPln("human is sleeping");
    }
}
class Child1 extends Parent {
    static void sleep() {
        SOPln("Child1 is sleeping");
    }
}
class Child2 extends Parent {
    static void sleep() {
        SOPln("Child2 is sleeping");
    }
}
class Sleep {
    p.s.vm(String args[]) {
        Parent child1 = new Child1(); // parent method is called
        Child1 child = new Child1(); // child1 method is called.
        child1.sleep();
        child.sleep();
    }
}
```

→ In this case of instance method of Child1 will be called

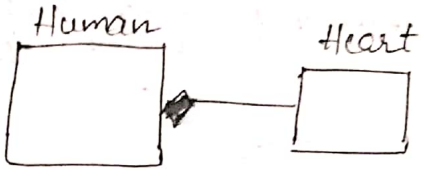
- In the above prg since the methods are made as static it cant be overridden rather the methods will be hidden.
- This In case of static the parent & child class has the method with the same name. This process is called as method hiding.

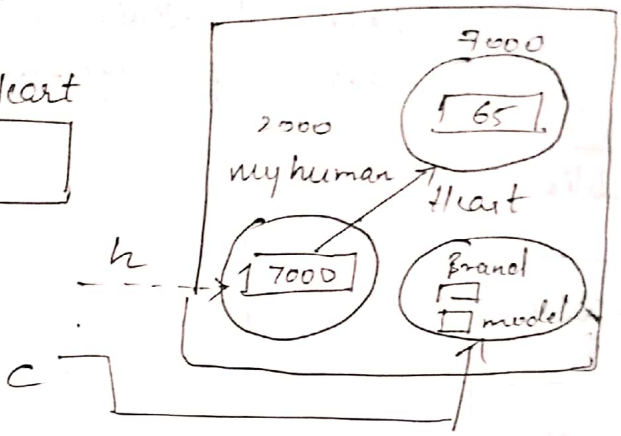
	within the class	within the package (sub class)	within the package (non-sub class)	outside the package (sub class)	outside the Package (non sub class)
public	✓	✓	✓	✓	✓
protected	✓	✓	✓	✓	X
default	✓	✓	✓	X	X
private	✓	X	X	X	X

Aggregation and composed

```
class Human {
    Heart myHeart;
    void eat() {
        SOP("Human eating");
    }
    Human() {
        myHeart = new Heart();
    }
    void myCar(Car myCar) {
        SOP(Car.brand);
        SOP(car.model);
    }
}
class Heart {
    int heartBeats = 65;
}
class Car {
    String brand;
    int model;
}
```

```
class Alpha {
    psvm(String args[]){
        Human h = new Human();
        SOP(h.myheart.Heartbeat);
        Car c = new Car();
        h.mycar(c);
    }
}
```





Interface

```
class Kala {
    void fight() {
        =
    }
}
interface Bala {
    void dance();
    void sing();
}
interface Mala {
    void dance();
    void paint();
}
abstract class Lala extends Kala implement Bala, Mala {
    void dance() {
        =
    } void paint() {
        =
    }
}
```

- class → implement / ~~extends~~ → interface
  - ↳ overidding method is compulsory (∵ they are abstract methods)
- interface → extends / ~~implement~~ → interface
- interface → implement → class → extend → class
- interfaces → extend → interface

Interface is a collection of abstract methods.

Rules of interface:

1. We can declare an interface by using the keyword interface followed by interface name

Ex
```
interface Demo {
}
```

2. A class doesn't extend interface rather it implements interface.

Ex
```
class Alpha implements Demo {
}
```

3. A interface cannot implement another interface rather it extends another interface.

Ex
```
interface Athlete {
}
interface Swimmer extends Athlete {
}
```

4. All the methods inside the interface are abstract methods (we can't provide any implementation for the methods inside interface)

Ex
```
interface Athlete {
    void pratice();
    void sleep();
}
```

5. When a class implements an interface it is compulsory to ovride all unimplemented methods the interface or the class has to be made as abstract.

```
interface Athlete{
    void practice();
    void sleep();
}

abstract class Sprinter implements Athlete{
}

class Swimmer implements Athlete{

public void practice(){
    SOP("practicing to swim");
}
public void sleep(){
    SOPln("Good night");
}
}
```

6. We cant create objects of interface

7. We can create a reference of an interface.

```
public class Demo{
    psvm(String args[]){
        Athlete swimmer = new Swimmer();
    }
}
```

Athlete is a interface

Swimmer() is a class which implements Athlete interface.

8. A class can implement multiple interfaces

Ex

```
interface Athlete{
    void practice();
    void sleep();
}

interface Human{
    void eat();
}
```

```
class Swimmer implements Athlete, Human {
    p.s.v.m (String args[]) {
        SOP("hi");
    }
    public void practice() {
        SOPln("practicing to swim");
    }
    public void sleep() {
        SOPln("sleeping");
    }
    public void eat() {
        SOP("eating");
    }
}
```

9. An interface can extend multiple interfaces.

```
interface Athlete {
    void practice();
    void sleep();
}
interface Human {
    void eat();
}
interface Sprinter extends Athlete, Human {
}
```

10 A class can extend another class & implement interface at the same time

```
interface A {
}
class B {
}
class Alpha extends B implements A {
}
```

But: class Alpha implement A extends B { } ✗ not possible

11. Methods inside the interface are public & abstract by default, whether we declare it or not

```
interface Athlete {
    abstract void practice();
    void sleep(); // or can be written as
                  abstract void sleep();
}
```

12. All the below declarations are same.

```
void sleep();
abtract void sleep();
public void sleep();
public abstrad void sleep();
abstrad public void sleep();
```

13. We cant give below modifiers to the methods inside interface. (only public & abstract can be used)

- private, protected (cant be used in place of public)
- final, static (cant be used in place of abstract)

14. We can declare variables inside an interface

Ex
```
interface Athlete {
    int x = 10;
}
```

15. It is compulsory to initialize a variable during the declaration inside the interface.

```
interface A {
    int a; ✗
}
```

```
interface A {
    int a = 5; ✓
}
```

16. By default the variables inside interface is public, static, and final.

17. All the below declarations are same.

```
int x = 10;
public int x = 10;
static int x = 10;
final int x = 10;
static public int x = 10;
public final static int x = 10;
```

18. We cannot change the value of a ~~interface~~ variable which is declared inside the interface

19. If we implement multiple interfaces in a class & if those interfaces have same name & same signature & same return type then we can provide a common implementation for such methods.

```
interface Athlete {
    void sleep();
}
interface Human {
    void sleep();
}
class Swimmer implements Athlete, Human {
    void sleep() {
        SOPln ("sleeping");
    }
}
```

20. If a class implements multiple interfaces & if those interfaces have methods with same name & different signature then we have to provide multiple implementations for those methods with same name.

```
interface Athlete {
    void sleep();
}
interface Human {
    void sleep(int x);
}
class Summer implements Athlete, Human {
    void Sleep() {
        SOP("sleeping for 16 hours");
    }
    void sleep(int x) {
        SOP("sleeping for " + x + " hours");
    }
}
```

21. If we have multiple interfaces having the method with same name, same signature & different return Type then we can't implement both these interfaces at the same time to a single class.

```
Interface Athlete {
    void sleep();
}
interface Human {
    int sleep()
}
class Summer implements Athlete {
    void sleep() {
    }
    int sleep() {
    }
}
```

or Human interface can be implemented but not both

- In the above example since there is sleep() method with same signature in both interfaces along with different return type we cant provide a common implementation in Swimmer class.
- In case if we provide 2 different implementations there will be a ambigious situation.

So in above case either Athlete or Human interface can be implemented.

22. We cannot have ~~constructors~~ no constructors inside the interface

23.
```
interface Athlete {
    int x = 50;
}
interface Human{
    int x = 999;
}
class Swimmer implements Athlete, Human {
    p s v m (String args[]) {
        SOPln (Athlete.x);
    } SOPln (Human.x);
}
```

24. If we have to access the variable x, we should access through the interface name.

If we directly access the variable 'x' it would give a compile time error bcoz 'x' exist in Athlete and Human interface & there will be an ambigious situation.

When to choose abstract class, concrete class and interface.

- When we know the complete implementation, we can go for concrete class.
- When we know the partial implementation we can go for abstract class.
- When we have no clue abt implementation we can go for interface.

Difference b/w abstract class & interface:

Abstract	Interface
• We can have concrete and abstract methods in an abstract class.	• We can have only abstract methods in interface
• We can have final methods inside abstract class	• We can't have final methods.
• It is compulsory to declare abstract for abstract methods inside abstract class.	• It is not compulsory to declare abstract for the methods in interface.
• We can have instance variables and static variables inside abstract class.	• We can have only static variables
• Initialization of variable is not compulsory during declaration inside abstract class	• Initialization of variables is compulsory during declaration.
• We can have constructors inside abstract class	• We can't have constructor in interface.
• We can have static & instance blocks inside abstract class.	• We can't have static & instance block in interface.
• We cannot achieve multiple inheritance	• We can achieve multiple inheritance.

1. What is the need of constructor in case of abstract class when it doesn't exist in interface.

2. Why constructor doesn't exist in interface

Since we can have instance variables inside abstract class it is the duty of that class to initialize the values for those instance variables

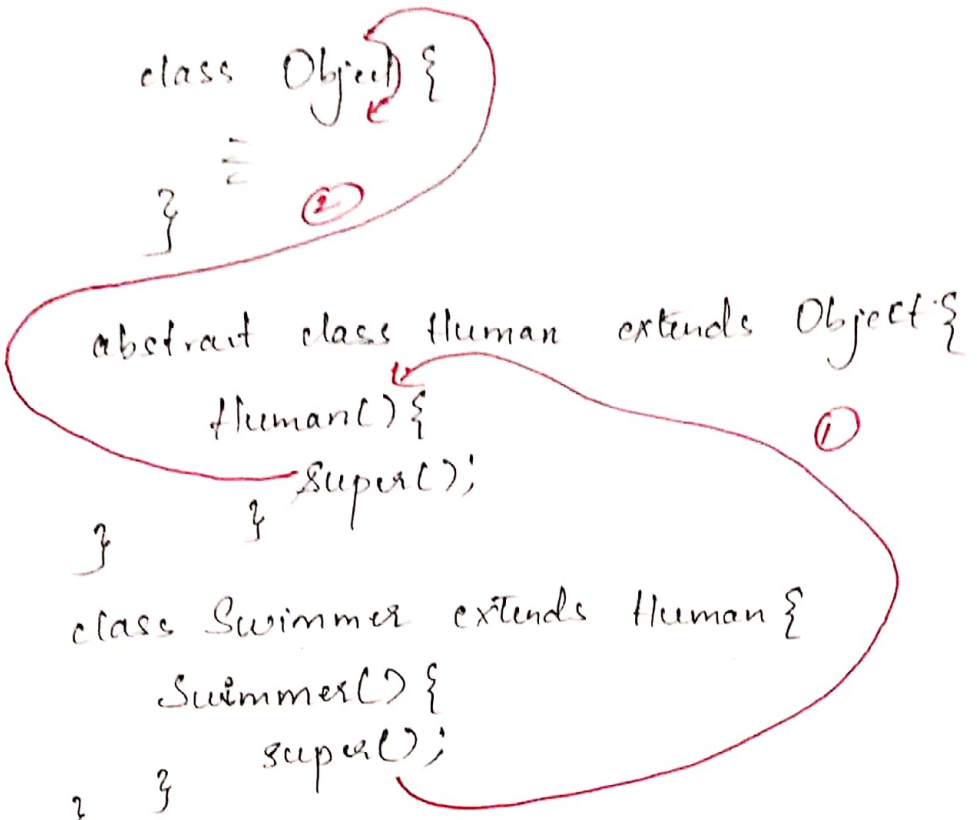
```
abstract class Human {
                    name;
                    age;
                    adhaar;
                    salary;

    Human (name, age, adhaar, salary)
    =
}
class Cricketer extends Human {
    Cricketer (name, age, adhaar, salary) {
        super (name, age, adhaar, salary);
    }
}
class Swimmer extends Human {
    Swimmer (name, age, adhaar, salary) {
        super (name, age, adhaar, salary);
    }
}
```

Why constructor is needed in abstract class

We need constructor inside abstract class ~~int~~ in order to achieve constructor chaining so that we can inherit the methods of object class, But this is not necessary incase of interface.

```
class Object {
    ...
}
```

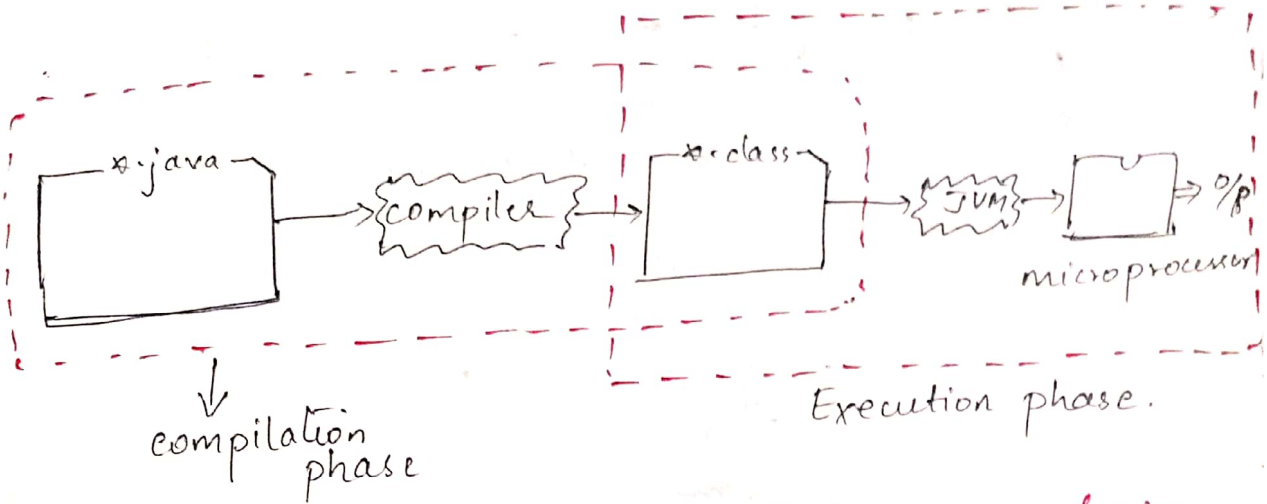
```
abstract class Human extends Object {
    Human() {
        super();
    }
}
class Swimmer extends Human {
    Swimmer() {
        super();
    }
}
```



Why is variable inside interface final?

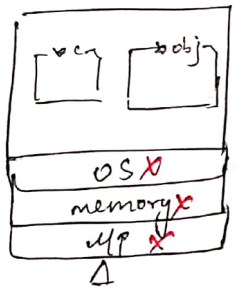
```
interface Alpha {
    int x = 50;
}
class A implements Alpha {
    void changeValue() {
        SOP(Alpha.x)
        Alpha.x = 100; // not possible.- coz value of x is
                       used by other classes as well.
    }
}
class B implements Alpha {
    void change() {
        SOP(Alpha.x);
    }
}
```

Exception Handling :-

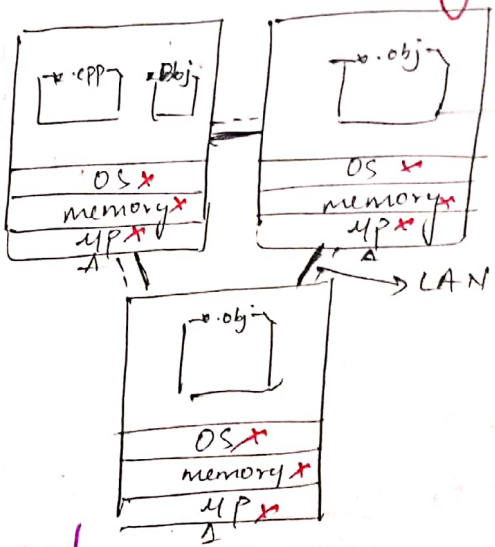


Is it compulsory to handle errors which occurs during run time / execution phase ?

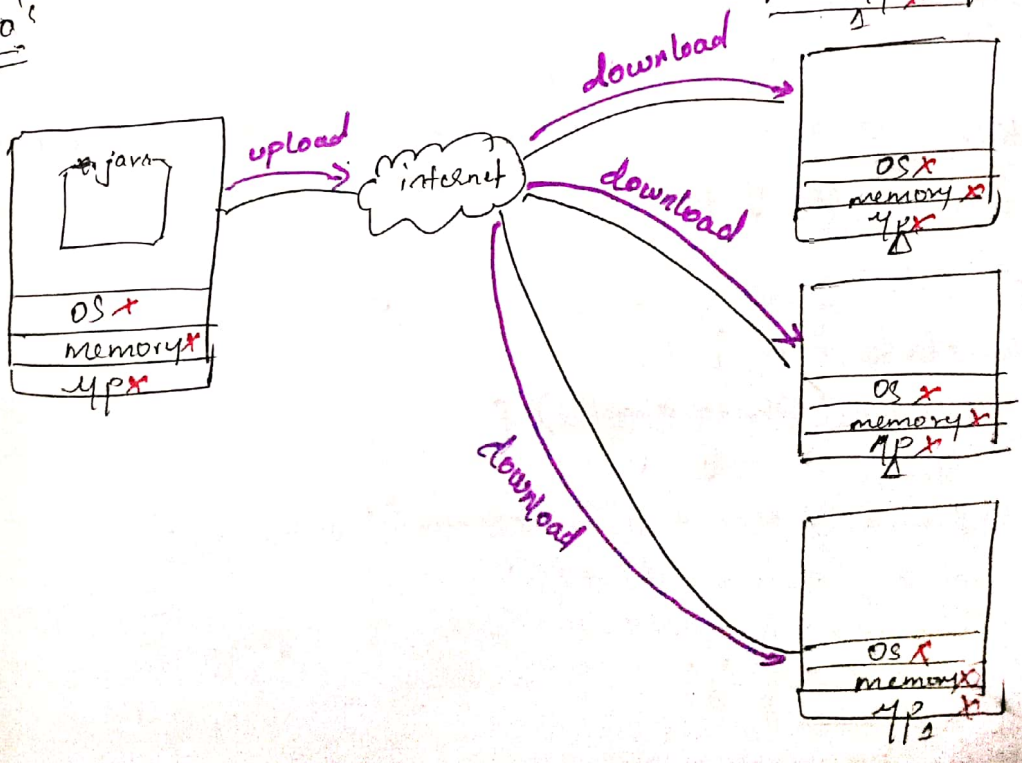
1970's
C
Dennis



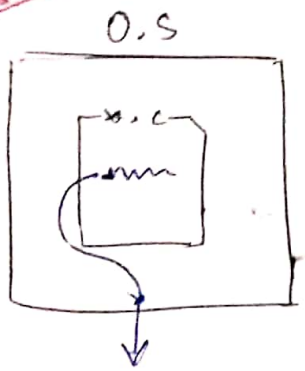
1980's
C++



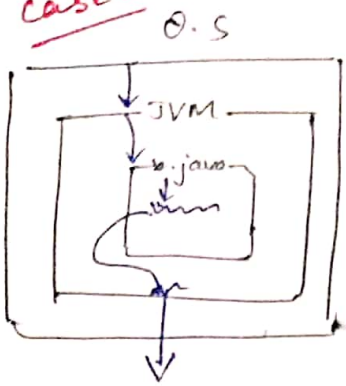
1990's



case 1



case 2

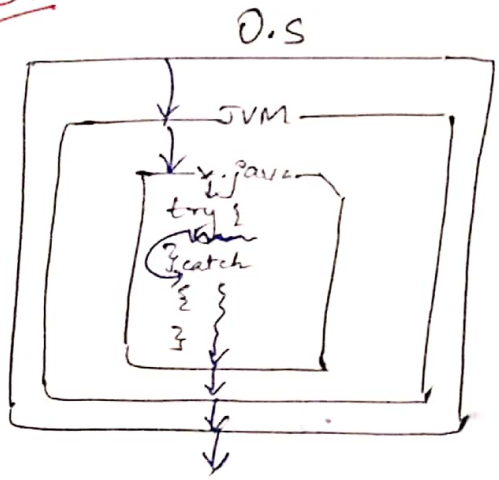


abnormal termination

```
try {
   Risk code
}
catch(xxx e)
{
}
```

(2)

case 3



Normal termination

- Exception is an unwanted / unexpected event that occurs during the execution of the program which disturbs the normal flow of the program.
- The objective of handling exception is for the graceful termination of the program
- Exception handling is providing an alternative way to continue the program normally

Example:

```
public class First {
        p s v m(String args[]) {
        SOPln()
        Scanner scan = new Scanner(System.in);
        int a = scan.nextInt();
        int b = scan.nextInt();
    } SOPln(a/b);
}
```

In the above program there is every possibility that a exception might occur in the selected like

- We can handle the exception using try and catch block

```
try{
    Risky code
}catch(xxx e){
    Alternative solution
}
```

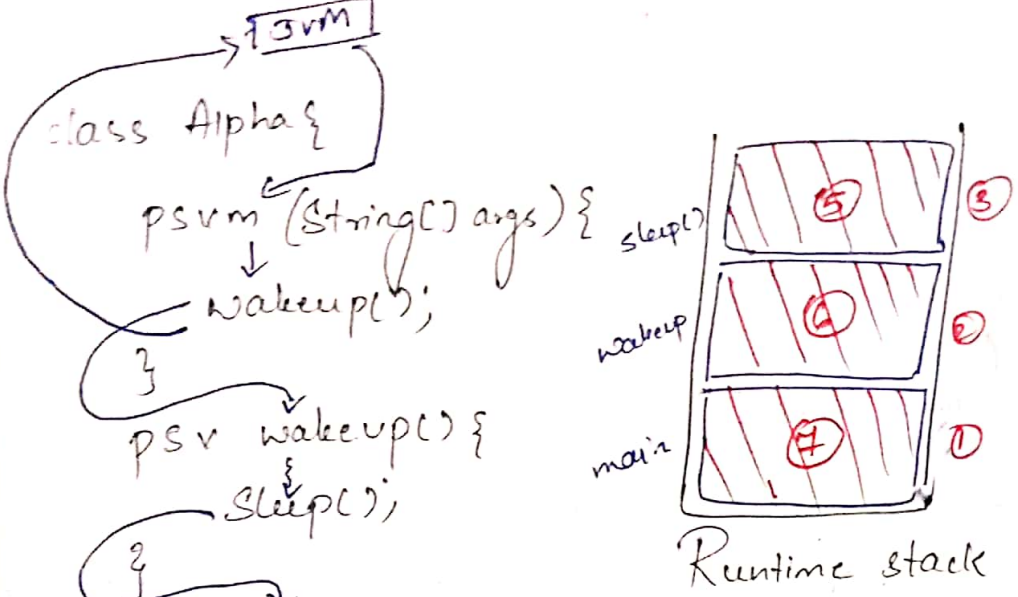
(3)

- All the risky code is written inside the try block
- The alternative sol^n has to be written inside the catch block
- If any exception occurs inside the try block an exception object get created and the control will be passed to the catch block.
- Once the catch block gets executed the prg executes normally after that.

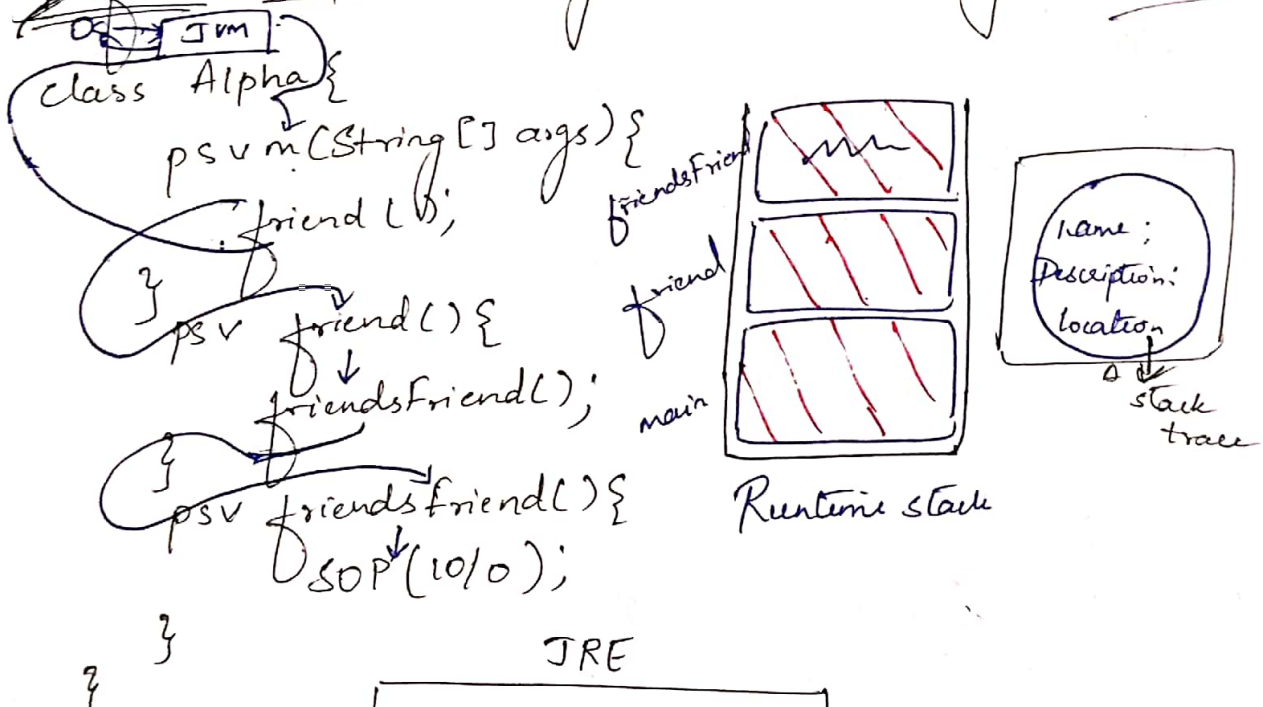
```
public class First {
    psvm(String args[]) {
    SOPln("prg starts");
    SOPln("enter a");
    int a = scan.nextInt();
    SOPln("enter b");
    int b = scan.nextInt();
    try {
        SOPln(a/b);
    }
    catch(Exception e) {
    SOPln("Exception occured bcoz of wrong i/p");
    }
    SOPln("normal termination of prg");
```

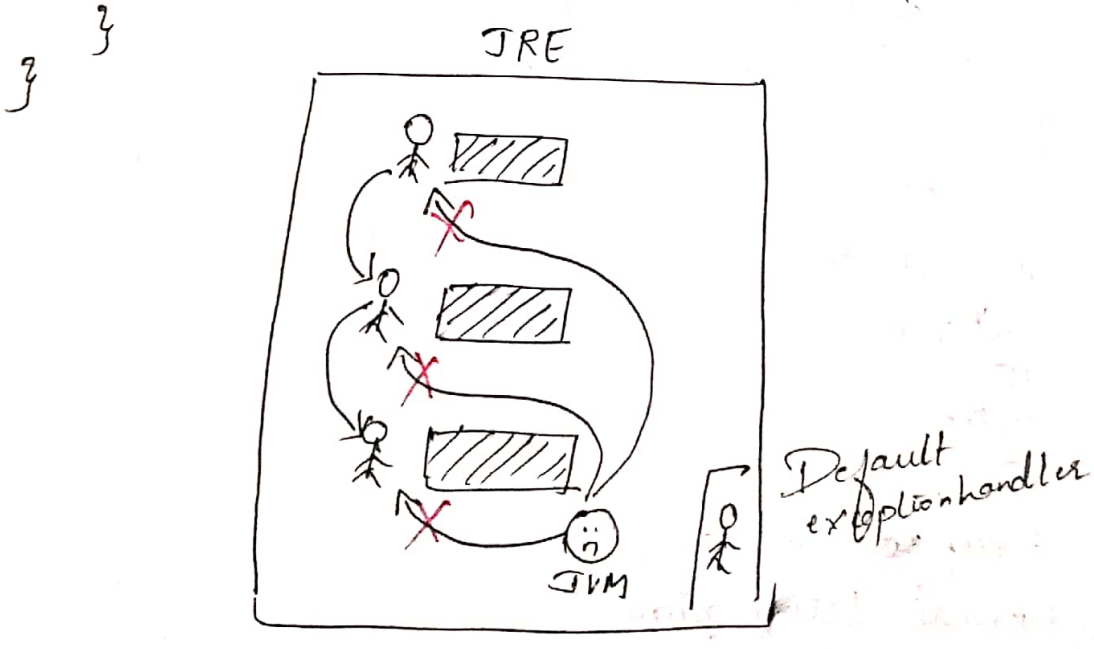
O/P

```
prg starts
enter a
10
enter b
0
Exception occured
normal termination
```



Default Exception handling & exception poopogation





Note: Whenever an exception occurs the JVM creates the exception object inside the heap area which consists of the
- ★ name of the exception
- ★ description of the exception
- ★ place where the exception occur (Stack Trace)

- If there is no code to handle the exception written by the developer then the JVM will handle the exception with the help of default exception handling.
- The default exception handler will print the details of in the exception object & the program exits abnormally.

Output

Exception in thread "main" java.lang.ArithmeticException: / by zero
at ~~com~~ First.friendsFriend
at First.friend
at First.main

Alternative code to handle the exception

```
public class First{
    psvm(String[] arg){
    try{
      friend();
    }
    catch(Exception e){
      SOPln("Exception handled");
  }}
  private s v friend(){
      friendsFriend();
  }
  private s v friendsFriend(){
     SOP(10/0);
  }
}
```

OUTPUT

Exception handled

Multiple catch block


```
public class First {
    psvm(String args[]){
    sopln();
    Scanner scan = new Scanner (System.in);
    try {
        int a = scan.nextInt();
        int b = scan.nextInt();
        SOPln(a/b);
    }
    catch (ArithmeticException e) {
      SOPln("Exception handled");
    }
    catch (InputMismatchException e) {
      SOPln("type properly");
    }
    catch (Exception e) {
      SOPln("give right input");
    }
}
}
```

OUTPUT

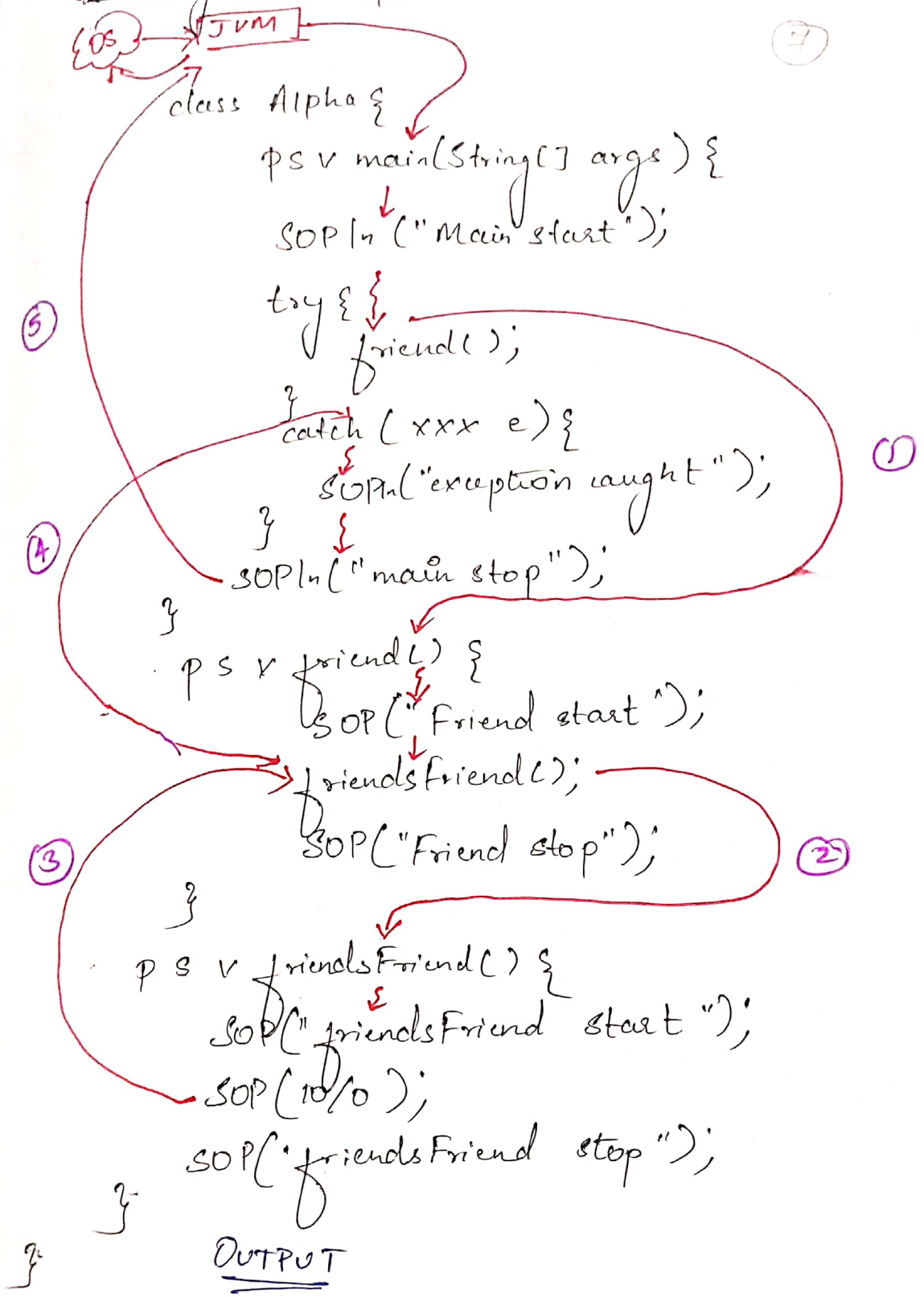
(i) 5
0
exception handled

(ii) abc
type properly

Note: We can have multiple catch blocks for a single try block. It is always a good practice to have different catch blocks for different kind of exceptions

- We cannot have multiple try blocks for a single catch block.
- It is compulsory to have a catch block for a try blo

Ducking an exception:



OUTPUT

Main start
Friend start
Friendsfriend start
exception caught
main stop

- If the called method is not handling the exception occured inside that method. rather it throws to the caller method to handle the exception is called as <u>ducking an exception</u>

<u>Rethrowing an exception</u>

```
public class Alpha {
    p s v m (String[] args) {
        SOPln("Main start");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        try {friend(n);
        } catch(Exception e) {
            SOPln(e.getMessage());
        }
        SOPln("main stop");
    }
    private static void friend(int n) throws ArithmeticExption {
        SOPln("friend start");
        friendsFriend(n);
        SOPln("Friend stop");
    }
    private static void friendsFriend(int n) throws ArithmeticExption {
        SOPln("friends friend start");
        SOPln(10/n);
        SOPln("friends friend stop");
    }
}
```

• If the exception occurs in the called method, it is the duty of the called method to inform the caller method about the exception.
• To do that the called method shd throw the exception object back to the caller method. This can be achieved with the help of 'throws' keyword.
• The process of catching the exception object & throwing it to the caller method is called as 'rethrowing'.

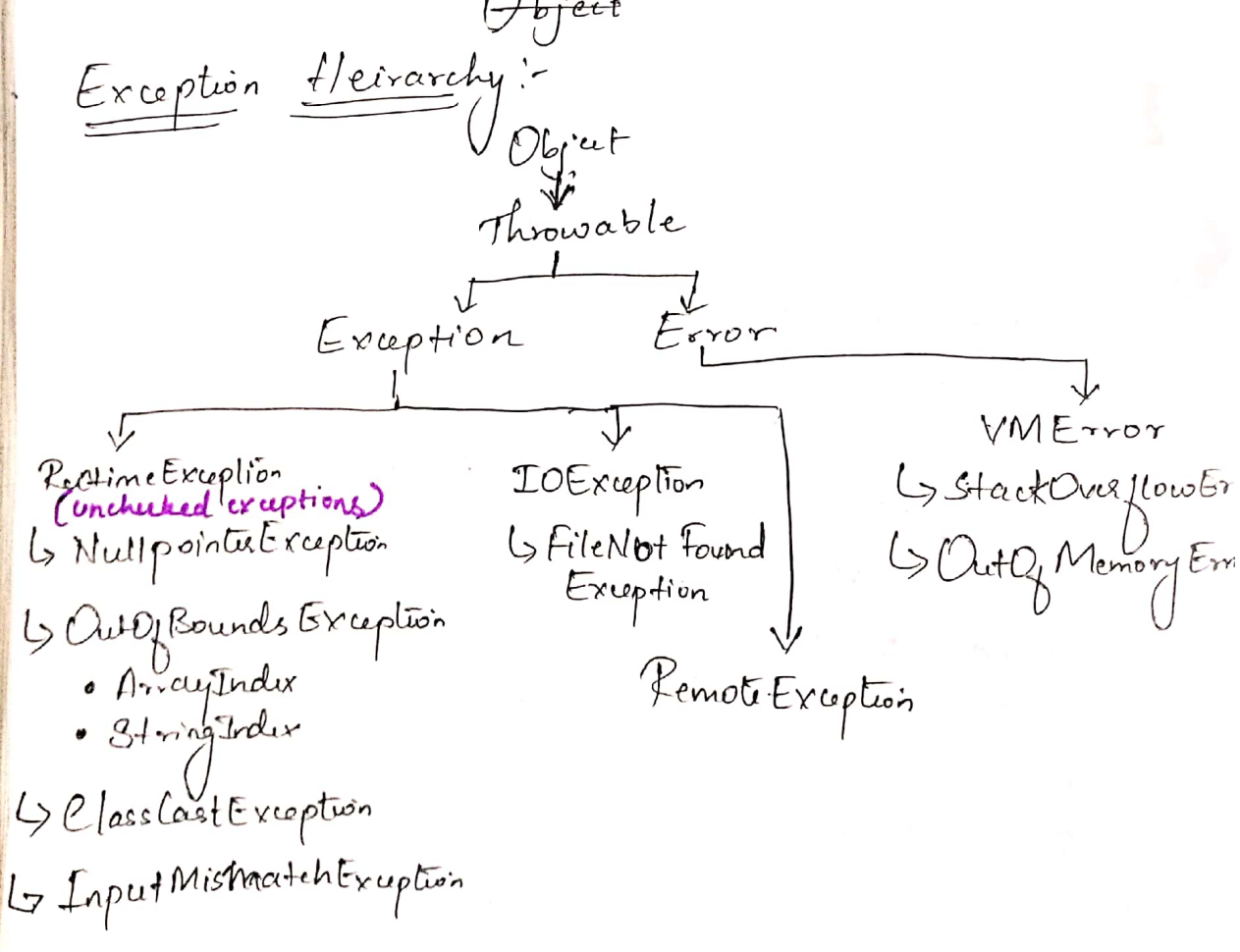
Whenever you call a method, 1st you have to create an object and then call but if its a static method you dont have to create an object & call

```
public class Alpha {
        p s v main(String args[]) {
         SOPln ("main start");
        Scanner scan = new Scanner(System.in);
        int n= scan.nextInt();
ption {  int ar[] = new int[5];
        try {
          Alpha.friend(n, ar);
        }
        catch(ArithmeticException e) {
          SOPln (" value cannot be zero");
        }
        catch(ArrayIndexOutOfBoundsException e) {
ption {  SOPln (" index given in the range of 0-4");
        }
        catch(Exception e) {
         SOPln (" Unexpected Exception ");
        }
        SOPln(" Main stop");
    }
```

```
psv friends (int n, int[] ar) throws
        ArithmeticException, ArrayIndexOutOfBoundsExcep
    SOPln("Friend start");
    friendsFriend(n, ar);
    SOPln(ar[n]);
    SOPln("Friend stop");
}

psv friendsFriend(int n, int[] ar) throws
        ArithmeticException {
    SOPln("friendsFriend start");
    SOPln(10/n);
    SOPln("friendsFriend stop");
}
}
```

(10)

Exception Heirarchy :-



Note: Errors occurs due to lack of s/m resources.

Ex StackOverflowError: This error occurs when there is no space in the stack to create the stack to push stackframe / activation record.

- Errors cannot be handled.
- Errors are always unchecked.
- The exceptions wch are checked by the compiler for the smooth execution of the prg are considered as checked exceptions.
  Ex FileNotFoundException.
- All the exceptions wch come under run time are unchecked exceptions. Rest all the exceptions are checked exceptions.
- If there is any checked exception in the program then it is compulsory to handle the exception otherwise we would get an error saying 'Error: Unresolved compilation problem: Unhandled exception type FileNotFoundException'
- The exceptions that are not checked by the compiler during the compile time are considered to be unchecked exceptions..

Methods of Exception:-

(1) printStackTrace() :- this method prints the name, description & stackTrace (location) of the exception (void)
stackTrace → places detailed info of where the exception occured and where all the exception propogated.

(2) getMessage() :- this method returns the description of the exception in String format but does not return anything

③ toString() :- this method returns the name & discription of the exception in the String format.

Explicitly creating an exception :-

public class We can explicitly create the exception object by using 'new' keyword (as we create all other objects)

We can throw the exception after creating it using the 'throw' keyword.

Ex throw new ArithmeticException("JUST FOR JOKE");

Program

```
public class Test{
    psvm(String[] args){
    SOPln("main start");

    try{friend();
    }catch(Exception dhoni){
        SOPln(dhoni);
    }
    SOPln("main stop");
}
private static void friend() throws ArithmeticException{
    throw new ArithmeticException();
}
}
```

Handling Checked exception :-

Checked exception can be handled in 2 ways

1. using try catch
2. by declaring the exception (using 'throws' keyword)

Ex
Example

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

public class Test {
    public static void main(String args[]) throws
            FileNotFoundException {
    SOPln(" Main start");
    openFile();
    SOPln("Main stop");
}
public static void openfile() throws FileNotFoundException {
    openTextFile();
}
static void openTextfile() throws FileNotFoundException {
    FileReader reader = new FileReader("/home/Desktop/a.txt");
}
}
```

Custom Exception:

We can create our own exception classes by extending exception & overridding the methods present inside the exception.

```
class myException extends Exception {
    // overridding getMessage() present in Exception class
    public String getMessage() {
    return "/ by 100"; // description of the exception
    }
}
public class CustomException {
    p s v m(String args[]){
    Scanner scan = new Scanner(System.in);
    int n = scan.nextInt();
    SOPln("Enter n");
```

```
try {
    calculate(n);
}
catch(MyException e){
    e.printStackTrace();
}
}
static void calculate(int n) throws MyException {
    if(n==100) {
        throw new MyException();
    }
    else { SOPln(100/n);
    }
}
```

OUTPUT

```
Enter value of n:
100
com.abc.exceptionHandling.MyException : / by 100
at com.abc.exceptionHandling.CustomException.calculate(line no)
```

exception name — description

package name — class name — method — location

Program to handle Owaged & underaged exception :

≤18 UnderagedException
>60 OwagedException

```
Ex
class UnderAgedException extends Exception {
    public String getMessage() {
        return "drink milk and grow fast";
    }
}
class OwnAgeException extends Exception {
    public String getMessage() {
        return "OMG, you are still alive??";
    }
}
```

```
public class Driving{
    Scanner scan = new Scanner(System.in);
    SOPln(" Enter your age ");
    int age = scan.nextInt();
    try{ canDrive(age);
    }
    catch(UnderAgedEruption e){
        e.printStackTrace();
    }catch(OverAgedEruption e){
        e.printStackTrace();
    }catch(Eruption e){
      SOPln(" unknown exception ");
    }
  static void canDrive(int age) throws UnderAgedEruption,
                                       OverAgedEruption {

    if(age<18){
        throw new UnderAgedEruption();
    }else if(age>60){
        throw new OverAgedEruption();
    }else{
      SOPln(" You can drive ");
    }
}
```

OUTPUT

```
Enter your age:
16
com.abc.exceptionHandling.UnderAgedEruption : drink milk
                                              and grow fast

at com.abc.exceptionHandling.Driving.canDrive(Driving.
                                              java:22)
```

ii) Enter your age:
65
com.abc.exceptionHandling.OverAgedException: OMG, you are still alive??
at com.abc.exceptionHandling.Driving.canDrive(Driving.java:22)

iii) Enter your age
25
You can drive

finally :-

- finally block consists set of instructions which has to be executed under any condition (whether exception occurs or doesn't)
- finally block follows try/catch block.

```
try {
} catch (xxx e) {
} finally {
}
```

(OR)

```
try {
} finally {
}
```

- for a single try block we can have only one finally block
- finally block shd always be written at the last.
- The primary purpose of the finally block is to close all the opened resources.

Ex. Connection to database, i/p streams etc.

- Handling code can also be written in the finally block (but will not be according to IT stds)

~~Adapter~~

~~A-I~~

~~Single~~

```
class Alpha{
    psvm(String args[]){
    FileReader reader = null;
    try{
      reader = new FileReader("\home\Desktop\a.txt");
    }catch(FileNotFoundException e){
     e.printStackTrace();
    }catch(Exception e){
     e.printStackTrace();
    }finally{
      try{
        reader.close();
      }catch(Exception e){
        e.printStackTrace();
      }
    }
  }
}
```

```
public class Test{
    psvm(String[] args){
     SOPln();
     int result = myCall(new Scanner(System.in).nextInt());
     SOPln(result);
    }
    static int myCall(int x){
     try{SOPln(100/x);
       return 100;
     }catch(Exception e){
       return 50;
     }finally{
       return 25;
     }
    }
}
```

OUTPUT

(i) 20 / 4 / 25

(ii) 0 / 25

try with resource (Alternate form of try/finally)

```
public class Hello {
    psvm(String[] args) {
    try (FileReader reader = new FileReader("/home/Data
                                            /a.txt");
    } catch(FileNotFoundException e) {
      e.printStackTrace();
    } catch(Exception e) {
      e.printStackTrace();
    }
}
```

The feature of try with resource is available after java 1.7.

- In the above prg the programmer doesn't have to close the resource (FileReader) explicitly using finally block.
- The resource will be close implicitly by the JVM after the execution of try block.
- If we have to open multiple resources we can open by using the below format

```
try(resource-1; resource-2; resource-3..){
}catch(xxx e){
}
```

- Any AutoClocable Interface classes can be written for the resources. (OR)

Inside try with resource we can create objects of certain classes which is implemented from AutoClosable Interface

- inside autoclosable interface there will be a method named close()

try with multicatch (JDK 1.7)

```
try{/*Risky code*/
  }
catch(ArithmeticException e){
  e.printStackTrace();
}catch(NullPointerException e){
  e.printStackTrace();
}catch(InputMismatchException e){
  e.printStackTrace();
}catch(ArrayIndexOutOfBoundsException e){
  e.printStackTrace();
}catch(Exception e){
  e.printStackTrace();
}
```

- Since the handling code for multiple exceptions are same we can use multicatch block after java 1.7 as given below

```
try{
  Risky code
  }
catch(ArithmeticException|NullPointerException|
      InputMismatchException|ArrayOutOfBoundsException e)
{ e.printStackTrace();
}catch(Exception e){
  SOPln("No idea")
}
```

- In multicatch block the list of exceptions written together shd not have any parent child relation
- The given below example gives the error because there is parent-child relation among them

- Parent child relation is not allowed in single catch block beoz the parent type will be able to handle the child object.

```
try { /* Risky code */
    }catch (StringIndexOutOfBounds |
            ArrayIndexOutOfBoundsException |
            IndexOutOfBoundsException e){
        e.printStackTrace();
    }
```

Frequently occuring exceptions:

1. Unchecked exception
   - ArithmeticException :~
     Ex SOPln(10/0);
   - InputMismatchException :~
     ↳ when user gives the wrong i/p this exception occurs.
     ex int n = scan.nextInt();
     when user enter any string value it throws this exception
   - NullPointerException
     Ex String s = null;
     SOPln(s.length());
     this exception occurs when trying to find the length of the string when its already initialized to null.
   - ClassNotFoundException
     ex java A
     We get this excep when the class is not found in the given directory
   - ArrayIndexOutOfBoundsException
     ↳ when we try to access an element beyond the array size this exception occurs
     ex int ar[] = new int[2];
     SOPln(ar[5]); /* Index 5 out of bounds for length 2 */

- StringIndexOutOfBoundsException
  (When we try to access the character beyond its string size)

ex String s = "hello";
SOP(s.charAt(10)); // String out of range: 10

- NumberFormatException

ex int n = Integer.parseInt("hello");
SOP(n);

↳ parseInt() method accepts only numeric characters in the string. If we pass any parameters apart from numeric characters it gives an exception.

- ClassCastException

Object o = new String(); // upcasting (child to parent)
(parent) (child)

String s = (String)o // downcasting
(child) (parent)

↳ The above prg is possible bcoz it has the reference of object type & and is pointing to the String [so string content is converted to string]

ex Object o = new Object();
String s = (String)o;

↳ Exception occurs in the above prg bcoz the object reference is pointing to the object itself. (Object cannot be converted to String)

String s = o; // cannot convert object to string

- IllegalThreadStateException

- IllegalArgumentException

Errors :-

- OutOfMemoryError : When there is no sufficient memory in a heap area.
- StackOverflowError :

↳ try with resource is an alternative way for try/finally :

```
try(Resource ){
}
```

(OR)

```
try {
}finally {
}
```