- OOPS in Java:
  - Simula : first object oriented language
  - Smalltalk : first truly object oriented language
  - Object oriented programming is a methodology or paradigm for designing a program using classes & objects.
  - Object : Any real time entity that has state and behavior. It can be physical or logical. It can be defined as an instance of a class.
    Example : A dog ; states like color, name, breed ; behaviors like wagging the tail, barking, eating, etc.

- Primary element of object :
  State - embodied in object properties
  - Reflects an objects attributes
  Behavior - a representation of it
  - How it reacts with other objects
  Method - A method is a group of statements that carry out a certain operation and give an outcome

- Class : Template that describes the data and behavior associated with class instantiation. A class can also be defined as a blue print from which we can create an individual object.
  : Class does not consume any space
  : Class name must begin with an initial letter (capitalized by convention).
  : class is not real world entity until instantiated
  : Java class may implement multiple interfaces by implementing abstract methods defined by an interface allowing java to implement multiple inheritance and interface based abstraction. to provide particular functionality

- INHERITANCE :
  - Enables a class to inherit properties and action from another class called a superclass or parentclass. called subclass / child class.
  - Represents IS-A relationship / parent child relⁿ
  - Inheritance used for :

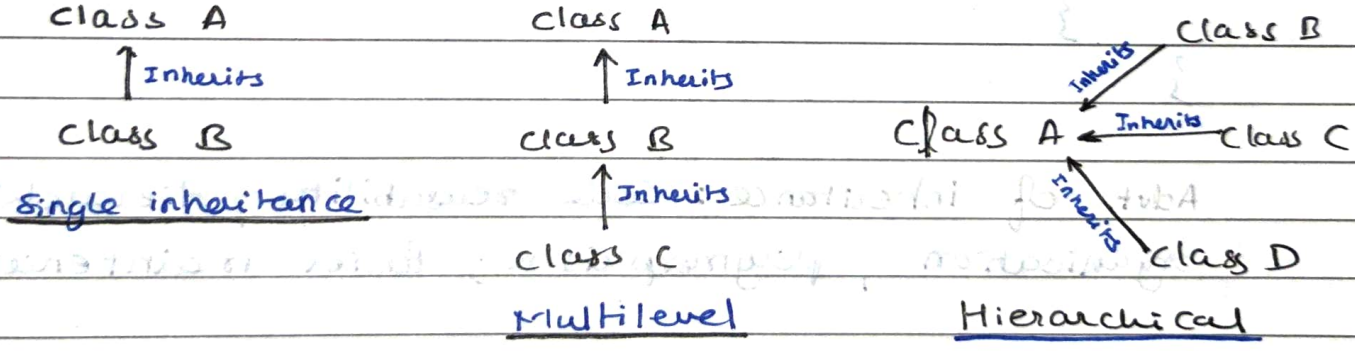
- For method overriding (so that runtime polymorphism can be achieved.
- For code reusability.

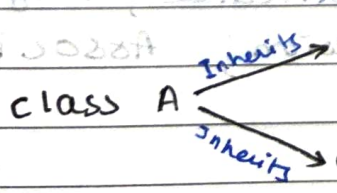
Real Life example - Child inheriting hair color from parents.
Java example -

```
class Employee {
    float salary = 40000;
}
class Programmer Extends Employee {
    int bonus = 10000;
}
```

main block :- `Programmer p = new Programmer();`
`Syso(p.salary + " " + p.bonus);`

- Types : single, multilevel, hierarchical; multiple & hybrid both are supported through interface only (if hybrid consist multiple inheritance).
  - multiple → not supported through class





Multiple inheritance

results into diamond problem (super classes sharing same method / field)
∴ interfaces are used

- Multiple inheritance using interface

```
interface A {
    void methodA();
}
interface B {
    void methodB();
}
class C implements A, B {
    public void methodA() {
        System.out.println("Method A");
    }
    public void methodB() {
        System.out.println("Method B");
    }
}
public class Main {
    public static void main(String[] args) {
        C obj = new C();
        obj.methodA();
        obj.methodB();
    }
}
```

Advt. of inheritance: Code reusability, Hierarchical organisation, polymorphism, Easier maintenance.

Terms other than Abstraction, Inheritance, polymorphism Encapsulation; Coupling, Cohesion, Association, Aggregation, Composition.

Aggregation is a type of association that represents a weak ownership between two objects. One class contains reference to another class, but both can exist independent

- AGGREGATION : represents HAS-A relationship between 2 class

```
class Employee {
    int id;
    String name;
    Address address;    //Address is a class
}
```

Here, Employee HAS-A entity reference address, so relationship is Employee HAS-A address.

```
class Operation {
    int square(int n) {
        return n*n;
    }
}
class Circle {
    Operation op;
    double pi = 3.14;
    double area(int radius) {
        op = new Operation();
        int rsquare = op.square(radius);
        return pi* rsquare;
    }
}
public class Main {
    public static void main(String[] args[]) {
        Circle c = new Circle();
        double result = c.area(5);
        System.out.println(result);
    }
}
```

- code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if relationship is a is maintained throughout the lifetime of the objects involved; otherwise aggregation.

- **POLYMORPHISM** (many + form) : Allows objects to behave differently based on their specific class type.

Features ....

Multiple behaviors - The same method can behave differently depending on the object that calls this method.

Method Overriding - A child class can redefine its method derived from parents class.

Method overloading - can define multiple method with same name but diff. parameters.

Runtime Decision - At runtime, Java determines which method to call depending on the objects actual class.

Example: A man acts as father, husband, employee each of these roles defines different behavior of man depending on object (home, office) calling it.

```
class Person {
    void role() {
        System.out.println("I am a person");
    }
}
```

```
class Father extends Person {
    @Override
    void role() {
        System.out.println("Father");
    }
}
public class Main {
    public static void main(String[] args) {
        Person p = new Father();
        p.role();          // Father
    }
}
```

- Use of Polymorphism : Code reusability, flexibility, Abstraction, Dynamic behavior.

- Types :
  - compile time polymorphism (static) → method Overloading
  - Runtime Polymorphism (Dynamic) → Method Overriding

CTP - also static polymorphism; is a type of polymorphism resolved during compile time i.e method call is determined at compile time based on method signature.

Method Overloading - multiple methods with same name but different parameter lists (different type, number, or order of parameters) within the same class

~~but different~~ type

✓ String a, int b ↔ int a, String b

✗ int a, int b ↔ int b, int a

- Method overloading is the implementation of CTP.
- compiler chooses the correct one based on signature. ↳ done at compile time

```
class Calculator {
    int add(int a, int b) {
        return a+b;
    }
    double add(double a, double b) {
        return a+b;
    }
    int add(int a, int b, int c) {
        return a+b+c;
    }
}
```

RTP - also dynamic polymorphism; RTP is resolved at runtime; occurs when child class overrides a method of its parent class and call to method is determined by object type at runtime, no the reference type.

Method overriding :-
It allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
- Overriden method must have same name, return type and parameters as parent class method.
- Method overriding is the implementation of RTP.
- Executes method based on actual object type

not the reference type.

```
class Animal {
    void sound() {
        System.out.println("Animals sound");
    }
}
class Dog extends Animal {
    void sound() {
        System.out.println("Dog's sound");
    }
}
class Cat extends Animal {
    void sound() {
        System.out.println("Cat's sound");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal a;
        a = new Dog();
        a.sound();
        a = new Cat();
        a.sound();
    }
}
```

reference type

reference type: Animal

RTE not possible Here (can be achieved by data members)

```
class Bike {
    int speedlimit = 90;
}
```

```
class Honda extends Bike {
    int speedlimit = 150;
}
```

// output : 90

- **ABSTRACTION :-**

is the process of hiding the implementation details and only exposing the features or capabilities that are necessary to the user.

- It emphasizes what an object does rather than how it does it.

Example :

ATM - we use simple interface - enter PIN, select option and withdraw, but do not see the complex backend processes like authentication validation & bank server communication.

- Abstraction can be achieved by :
  1) Using Abstract Class (for partial Abstraction)
  2) Using Interface (For 100% Abstraction)

1) Abstract class

An abstract class is a class that cannot be instantiated directly. It serves as a blueprint for other classes.

- An abstract method is a method declared without any implementation.
- An abstract class can contain both abstract & concrete methods (where implementation is present)
- A class whose parent is an abstract class must implement the abstract method. If child class cannot provide the implementation, then it must be declared as abstract.
- Any class that contains one or more abstract methods must be declared as abstract.
- Instantiation of an abstract class is not allowed in Java i.e cannot create an object

of an abstract class using the new keyword.
- If no constructor provided, abstract class has a default constructor. An abstract class can also include user-defined constructors.
- More than one constructor permitted in abstract class.

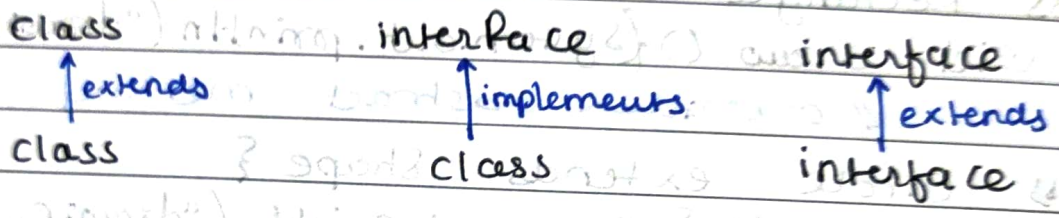
Example:

```
abstract class Shape {      // abstract class
    abstract void draw();   // abstract method with no implementation
}
class Rectangle extends Shape {
    void draw(){System.out.println("drawing rectangle");
}       // overriden abstract method
class Circle extends Shape {
    void draw() { System.out.println("drawing circle");
}
public class Main {
    public static void main(String[] args){
        Shape s = new Circle();
        s.draw();
    }
}
```

- When to use abstract class:
  - If multiple related classes share common functionality, it provides default implementations.
  - some implemented, some non implemented method for overriding in sub class.
  - If program design follows inheritance based design and has IS-A, e.g: Parent is a superclass for Father and Son.
  - To avoid code duplication by centralized shared logic

2) Interface

- An interface in Java is a blueprint of a class.
- It has static constants and abstract methods.
- There can be only abstract methods in Interface
- Used to achieve abstraction and multiple Inheritance
- Interface represents IS-A relationship
- Cannot be instantiated
- It is used to achieve loose coupling.
- By default, fields are public, static & final; methods are public & abstract.

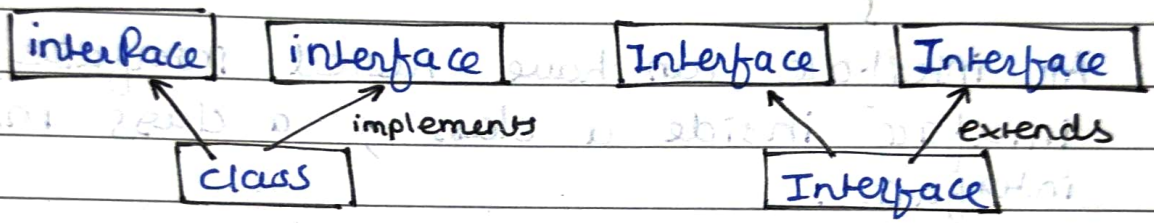


Example:

```
interface Bank {
    float rateOfInterest();
}
class SBI implements Bank {
    public float rateOfInterest() { return 9.15f; }
}
class PNB implements Bank {
    public float rateOfInterest() { return 9.7f; }
}
class HDFC implements Bank {
    public float rateOfInterest() { return 8.7f; }
}
public class Main {
    public static void main(String[] args) {
        Bank b;
```

```
        b = new SBI();
        System.out.println("SBI ROI:" + b.rateOfInterest());
        b = new PNB();
        System.out.println("PNB ROI:" + b.rateOfInterest());
        b = new HDFC();
        System.out.println("HDFC ROI:" + b.rateOfInterest());
    }
}
```

* Multiple Inheritance by Interface



In case of an interface there is no ambiguity, because the implementation class provides its implementation. Example.

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

- Inheritance of interfaces : A class implements an interface but one interface can extend another interface.
- Interface support polymorphism.
- Interface can have method body but we need to declare it default method.
- Interface can have static method in it

```
interface Drawable {
    void draw();
    static int cube(int x) { return x*x*x; }
}
```

- An interface can have nested interface; an interface inside a class, a class inside an interface :

- **MARKER OR TAG INTERFACE**
- An empty interface is known as marker/tag interface
- It delivers run time type information about an object. It is the reason that JVM and compiler has additional information about an object. In short it indicates a signal/command to JVM.
- Example: Serializable, Clonable, Remote Interface

- There are two alternatives of marker interface that produces the same result as marker interface.

1) Internal flags :- It can be used in place of marker interface to indicate any specific operation.

2) Annotation :- Since Java 5, marker interfaces are omitted insted annotations help achieve

the same results. It allows flexible metadata capability. ∴ by applying annotation to any class, we can perform specific action.

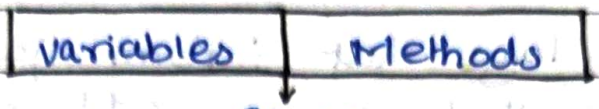
- Used as a tag that inform java compiler by a message so that it can add some special behavior to the class implementing it.
- Implementing an empty interface tells the compiler to do some operations.

- Difference betn

Abstract class	Interface
- can have abstract, non abstract methods.	- can have only abstract methods (can be default/static)
- does not support multiple inheritance.	- Supports multiple inheritance
- can have final, non-final, static & non static variable	- has only static and final variables
- can provide implementation of the interface	- cannot provide implementation of an abstract class.
- can extend another java class and implement multiple Java interfaces.	- Can extend another Java interface only.
- uses extends	- uses implements
- can have privat, protected etc class members.	- Members here are public default.

- **Encapsulation**

Encapsulation in Java is a process of wrapping code and data together into a single unit.



We can create a fully encapsulated class in Java by making all the data members of class private, and we can use setter and getter methods to set and get data in it.

- Encapsulation helps to protect data and control access to it.
- Protects sensitive data from being access directly.
- It hides unecessary data from user of a class and only shows functionality of a class.
- Easier to scale applications since it provides flexibility to add or modify features without impacting entire codebase.
- Encapsulated classes can be reused across projects.
- Example:

```
package com.tpointtech;
public class Student {
    private String name;          //private data member
    public String getName() {     //getter method for name
        return name;
    }
    public void setName(String name) {   //setter method
        this.name = name;                 //for name
    }
}
```

Real time Example:

```
class Account {
    private long acc_no;
    private String name, email;
    private float amount;

    public long getAcc_no() {
        return acc_no;
    }
    public void setAcc_no(long acc_no) {
        this.acc_no = acc_no;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getEmail(String email) {
        this.email = email;
    }
    public float getAmount() {
        return amount;
    }
    public void setAmount(float amount) {
        this.amount = amount;
    }
}
public class Main {
    public static void main(String[] args) {
        Account acc = new Account();
```

```
        acc.setAcc_no(7560504001L);
        acc.setName("Sonal Jaiswal");
        acc.setEmail("sonal123@gmail.com");
        acc.setAmount(500000f);

        System.out.println(acc.getAcc_no() + " " +
        acc.getEmail() + " " + acc.getAmount());
    }
}
```

- Advantages of Encapsulation:
  - Data protection
  - Enhanced security
  - Improved maintainability
  - Increased flexibility
  - Encourages Modularity
  - Promotes Abstraction

- Disadvantages of Encapsulation:
  - Increase code complexity
  - Decrease performance
  - Impractical for small scale programs
  - Mismanagement of setters and getters
  - Learning curve / challenging for beginners.