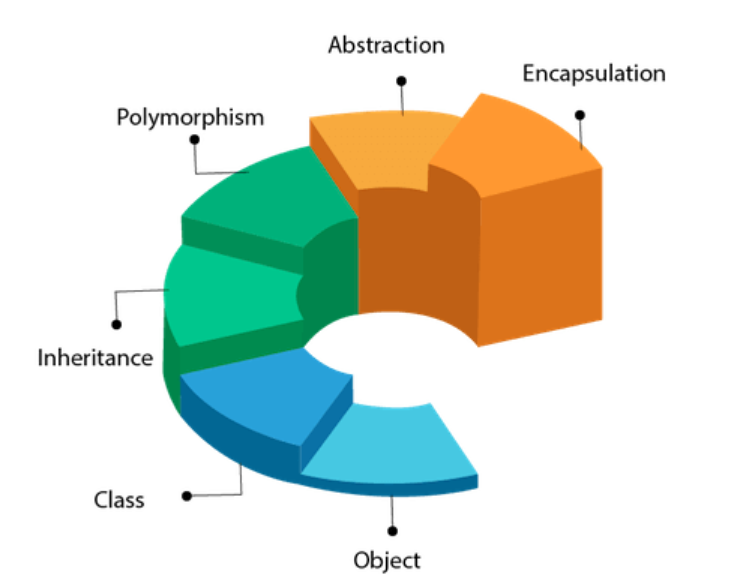
**OOPS CONCEPTS IN JAVA**

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

* [Object](https://www.javatpoint.com/object-and-class-in-java)
* Class
* [Inheritance](https://www.javatpoint.com/inheritance-in-java)
* [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
* [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
* [Encapsulation](https://www.javatpoint.com/encapsulation)



**Object**

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

**Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviours like wagging the tail, barking, eating, etc.



## Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

**Inheritance in Java:**

Inheritance in Java is a mechanism in which one object acquires   
all the properties and behaviours of a parent object. Inheritance   
represents the IS-A relationship which is also known as a   
parent-child relationship. We use inheritance for Method Overriding.  
So, runtime polymorphism can be achieved and also for Code Reusability.

**Types of inheritance :**

There are five types of inheritance.

1. Single Inheritance  
A single subclass extends from a single superclass.

2. Multilevel Inheritance  
A subclass extends from a   
superclass and then the same subclass acts as a superclass   
for another class.

3. Hierarchical Inheritance  
multiple subclasses extend from a single superclass.

4. Multiple Inheritance  
A single subclass extends from multiple superclasses.

5. Hybrid Inheritance  
Hybrid inheritance is a combination of two or more types  
of inheritance.

**The syntax of Java Inheritance:**

class Subclass-name extends Superclass-name   
{   
//methods and fields   
}

The extends keyword indicates that you are making a new class that   
derives from an existing class. In the terminology of Java, a class which   
is inherited is called a parent or superclass, and the new class is called   
child or subclass.

**Example of Inheritance**

class Animal {

// field and method of the parent class  
String name;  
public void eat() {  
System.out.println("I can eat");  
}  
}

// inherit from Animal  
class Dog extends Animal {

// new method in subclass  
public void display() {  
System.out.println("My name is " + name);  
}  
}

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

// create an object of the subclass  
Dog labrador = new Dog();

// access field of superclass  
labrador.name = "Rohu";  
labrador.display();

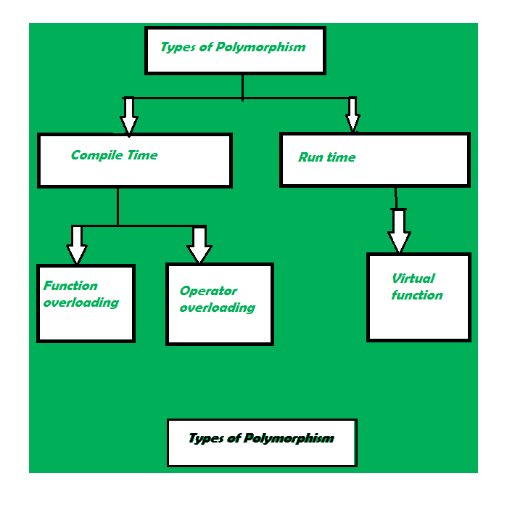
// call method of superclass  
// using object of subclass  
labrador.eat();

}  
}

Output is :  
My name is Rohu  
I can eat

Here we have derived a subclass Dog from superclass Animal.  
labrador is an object of Dog. However, name and eat() are   
the members of the Animal class. Since Dog inherits the field   
and method from Animal, we are able to access the field and   
method using the object of the Dog.

**Polymorphism in Java**

  
  
The word “poly” means many and “morphs” means forms, So it means many forms.  
In simple words, we can define polymorphism as the ability of a message to be   
displayed in more than one form. Polymorphism is the ability of an object to   
take on many forms. The most common use of polymorphism in OOP occurs when a   
parent class reference is used to refer to a child class object.  
Any Java object that can pass more than one IS-A test is considered to be   
polymorphic. In Java, all Java objects are polymorphic since any object   
will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is   
through a reference variable.

**Example**  
Polymorphism can be demonstrated by various examples, such as a polygon class   
that can render different shapes, an animal class that can have different eating  
behaviors, or a bank account class that can have different interest rates.

class Polygon {

// method to render a shape  
public void render() {  
System.out.println("Rendering Polygon...");  
}  
}

class Square extends Polygon {

// renders Square  
public void render() {  
System.out.println("Rendering Square...");  
}  
}

class Circle extends Polygon {

// renders circle  
public void render() {  
System.out.println("Rendering Circle...");  
}  
}

class Main {  
public static void main(String[] args) {  
  
// create an object of Square  
Square s1 = new Square();  
s1.render();

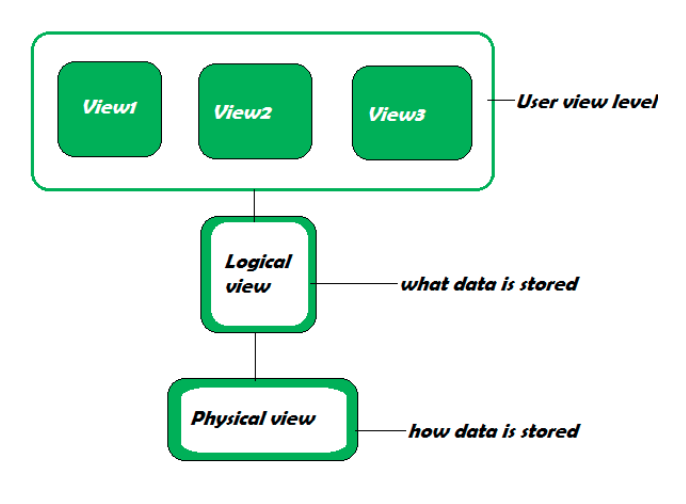
// create an object of Circle  
Circle c1 = new Circle();  
c1.render();  
}  
}

In Java polymorphism is mainly divided into two types:  
compile-time polymorphism and runtime polymorphism,   
which differ in when and how they are resolved.

Compile-time polymorphism, also known as static polymorphism, occurs when the   
compiler determines the method to be invoked based on the reference type and   
arguments. It is achieved by method overloading and operator overloading.

Runtime polymorphism, also known as dynamic polymorphism, occurs when the   
method to be invoked is determined by the actual object type at runtime.   
It is achieved by method overriding and inheritance.

**ABSTRACTION IN JAVA**



Abstraction is a feature of OOPs. The feature allows us to hide the implementation detail from the user and shows only the functionality of the programming to the user. Because the user is not interested to know the implementation. The best example of abstraction is a car. When we derive a car, we do not know how is the car moving or how internal components are working? But we know how to derive a car. It means it is not necessary to know how the car is working, but it is important how to derive a car. The same is an abstraction.

The same principle (as we have explained in the above example) also applied in Java programming and any OOPs. In the language of programming, the code implementation is hidden from the user and only the necessary functionality is shown or provided to the user. We can achieve abstraction in two ways:

1.Using Abstract Class

Abstract classes are the same as normal Java classes the difference is only that an abstract class uses abstract keyword while the normal Java class does not use. We use the abstract keyword before the class name to declare the class as abstract. Using an abstract class, we can achieve 0-100% abstraction. An abstract class contains abstract methods as well as concrete methods. If we want to use an abstract class, we have to inherit it from the base class. If the class does not have the implementation of all the methods of the interface, we should declare the class as abstract. It provides complete abstraction. It means that fields are public static and final by default and methods are empty. Example of an Abstract Class: Main.java   
//abstract class   
abstract class Demo   
{   
//abstract method   
abstract void display();   
}   
//extends the abstract class   
public class MainClass extends Demo   
{   
//defining the body of the method of the abstract class   
void display()   
{   
System.out.println("Abstract method called.");   
}   
public static void main(String[] args)   
{   
MainClass obj = new MainClass ();   
//invoking abstract method

2.Using Interface

In Java, an interface is similar to Java classes. The difference is only that an interface contains empty methods (methods that do not have method implementation) and variables. In other words, it is a collection of abstract methods (the method that does not have a method body) and static constants. The important point about an interface is that each method is public and abstract and does not contain any constructor. Along with the abstraction, it also helps to achieve multiple inheritance. The implementation of these methods provided by the clients when they implement the interface. Using interface, we can achieve 100% abstraction. Separating interface from implementation is one way to achieve abstraction. The Collection framework is an excellent example of it.

Features of Interface:We can achieve total abstraction.We can use multiple interfaces in a class that leads to multiple inheritance.It also helps to achieve loose coupling.To use an interface in a class, Java provides a keyword called implements. We provide the necessary implementation of the method that we have declared in the interface.

Let's see an example of an interface.

Car.java

interface CarStart   
{   
void start();   
}   
interface CarStop   
{   
void stop();   
}   
public class Car implements CarStart, CarStop   
{   
public void start()   
{   
System.out.println("The car engine has been started.");   
}   
public void stop()   
{   
System.out.println("The car engine has been stopped.");   
}   
public static void main(String args[])   
{   
Car c = new Car();   
c.start();   
c.stop();   
}   
}   
Output: The car engine has been started. The car engine has been stopped.

**Encapsulation**

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
* In encapsulation, the data in a class is hidden from other classes, which is similar to what **data-hiding** does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.
* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.

Example:

class Area {

// fields to calculate area

int length;

int breadth;

// constructor to initialize values

Area(int length, int breadth) {

this.length = length;

this.breadth = breadth;

}

// method to calculate area

public void getArea() {

int area = length \* breadth;

System.out.println("Area: " + area);

}

}

class Main {

public static void main(String[] args) {

// create object of Area

// pass value of length and breadth

Area rectangle = new Area(5, 6);

rectangle.getArea();

}

}

Output:

Area: 30

In the above example, we have created a class named Area. The main purpose of this class is to calculate the area.

To calculate an area, we need two variables: length and breadth and a method: getArea(). Hence, we bundled these fields and methods inside a single class.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.