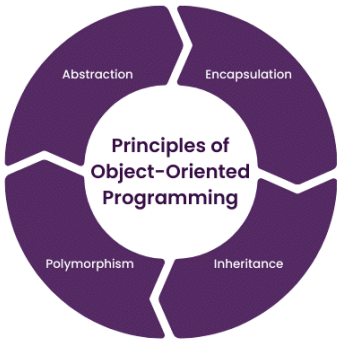
PRINCIPLES OF OBJECT ORIENTED PROGRAMMING

Four Main Object Oriented Programming Concepts of Java



Object-oriented programming generally referred to as OOPS is the backbone of java as java is not a purely object oriented language but it is object oriented language. Java organizes a program around the various objects and well-defined interfaces. There are four pillars been here in OOPS which are listed below. These concepts aim to implement real-world entities in programs.

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Inheritance

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Why Do We Need Java Inheritance?

Inheritance is a key feature of OOP, allowing you to create flexible and reusable code. Understanding how to implement it effectively can greatly improve the structure of your programs.

- Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- Abstraction: The concept of abstract where we do not have to provide all details, is achieved through inheritance.

Important Terminologies Used in Java Inheritance

- Class: Class is a set of objects which shares common characteristics/ behaviour and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Super Class/Parent Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).
- Sub Class/Child Class: The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to Use Inheritance in Java?

The **extends keyword** is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, "extends" refers to increased functionality.

Java Inheritance Types

Below are the different types of inheritance which are supported by Java.

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behaviour of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.

Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the grandparent's members.

Hierarchical Inheritance

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class C is derived from interfaces A and B.

Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces if we want to involve multiple inheritance to implement Hybrid inheritance.

However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.

Java IS-A type of Relationship

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance, refer notes.

"super" Keyword

The super keyword in Java is a reference variable that is used to refer to parent class when we're working with objects. The Keyword **"super"** came into the picture with the concept of Inheritance.

Characteristics of Super Keyword in Java
In Java, super keyword is used to refer to the parent class of a subclass. Here are some of its key characteristics:

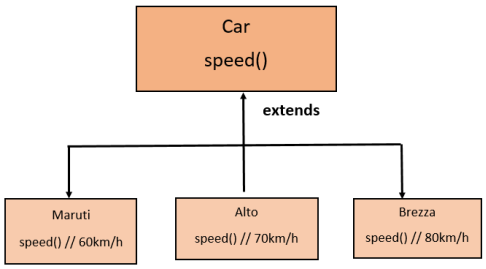
- **super is used to call a superclass constructor:** When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.
- **super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.
- **super is used to access a superclass field:** A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class's version of a field.
- **super must be the first statement in a constructor:** When calling a superclass constructor, the super() statement must be the first statement in the constructor of the subclass.
- **super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.
- **super is not required to call a superclass method:** While it is possible to use the super keyword to call a method in the parent class, it is not required. If a method is not overridden in the subclass, then calling it without the super keyword will invoke the parent class's implementation.

Use of super keyword in Java
It is majorly used in the following contexts as mentioned below:

- Use of super with Variables
- Use of super with Methods
- Use of super with Constructors

Polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.



Types of Java Polymorphism

In Java Polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

Compile-Time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. *But Java doesn't support the Operator Overloading.*

Method Overloading/Early Binding When there are multiple functions with the same name but different parameters, type, return type or order of parameter then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Runtime Polymorphism in Java

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

How it works internally When we create an object of Child Type but Data Type is of Parent. When the method is called JVM checks the method exists in Parent Class or not. If the method didn't exist then is throws an error. If the method exists in Parent but not in Child it runs the Parent method. But if the method exists in both then, this is a case of Overriding where the method in Child class has been Upcasted to Parent Class. This process takes place at runtime which is known as Dynamic Method Dispatch/Late Binding.

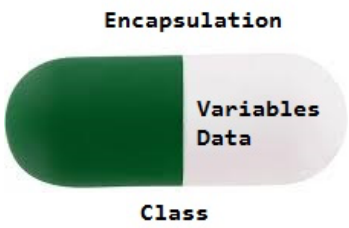
'final' keyword used at parent's method where overriding is being done to avoid overriding. 'final' keyword is also used to avoid to inheritance. If a class is declared final, then all of it's methods are also declared final. Methods with 'static' keyword are not overridden.

Advantages of Polymorphism in Java

1. Increases code reusability by allowing objects of different classes to be treated as objects of a common class.
2. Enables objects to be treated as a single type, making it easier to write generic code that can handle objects of different types.

Encapsulation

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.



Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together 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.

For example: The Programmer of the edit menu code in a text-editor GUI might at first, implement the cut and paste operations by copying actual screen images in and out of an external buffer. Later, he/she may be dissatisfied with this implementation, since it does not allow compact storage of the selection,

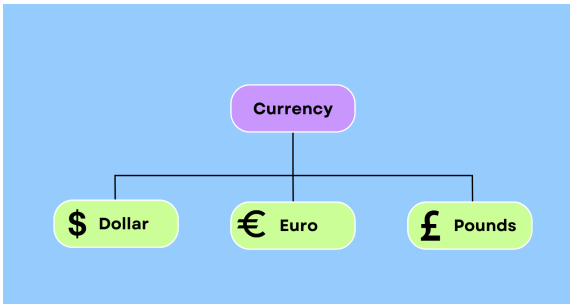
and it does not distinguish text and graphic objects. If the programmer has designed the cut-and-paste interface with encapsulation in mind, switching the underlying implementation to one that stores text as text and graphic objects in an appropriate compact format should not cause any problems to functions that need to interface with this GUI. Thus encapsulation yields adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts.

Advantages of Encapsulation

- Data Hiding: it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
- Increased Flexibility: We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program

Abstraction

In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.



Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviour of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerator will increase the speed of a car or applying brakes will stop the car, but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc. in the car. This is what abstraction is.