Ranjeet Kendre

OOPS — OOPS Stands for object-oriented programming system/structure.

The main principles of object-oriented programming are Abstraction, Encapsulation,Inheritance and Polymorphism.

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation

1. Inheritance :
inheritance.
One class acquires property of another class with the help of extends keywords is known as

Importance of the inheritances
The most important use of inheritance in java is code reusability. The code that is present in parent class can be directly used by the child class

Types of inheritance

1. Single level inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hybrid inheritance
5. Hierarchical inheritance

1. Single level inheritance
It is an operation where inheritance takes place between 2 classes. To perform single level inheritance only two classes are required.
The class from where properties are acquiring/inheriting is called super class/base class/parent class
The class to where properties are inherited/delivered is called sub class/child class.

2. Multilevel inheritance
Multilevel Inheritance takes place between 3 or more than 3 classes.
In Multilevel Inheritance 1 sub class acquires properties of another super class & that class acquires properties of another super class & phenomenon continuous.

3. Hierarchical Inheritance:
Multiple sub classes can acquire properties of 1 super class is known as hierarchical Inheritance.

4. Multiple Inheritance:
1 subclass acquiring properties of 2 super classes at the same time is known as Multiple Inheritance.
Java doesn't support Multiple Inheritance using class because of diamond ambiguity problem.
By using interface we can achieve Multiple Inheritance.

Note: object class is the super most class in all java

5.Hybrid inheritance :
The hybrid inheritance is the combination of two or more than two types of inheritance.

Question - Why multiple & hybrid inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
Example - Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile- time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

2. Polymorphism

It is one of the oops principle where one object showing different Behaviour at different stage is known as **Polymorphism**.

Types of Polymorphism

1 Compile time Polymorphism
2.Run time Polymorphism

Compile time Polymorphism

In Compile time Polymorphism method declaration and Definition are binded during the compilation time based on argument or input parameter is known as compile time poly.

Example - Method overloading
Method overloading
When the method name is same with different argument/input param with different data types within the same class is called as method overloading.

It is also known as early binding.
We can overload the non static and static method. We can overload the main method
You can any number of method in a class but JVM class main () methods which receives string array as arguments only.

Run Time Polymorphism

In Run time Polymorphism method declaration and Definition are binded during the run time or execution time based on input parameter or argument.

Example - Method overriding

When the method is present in parent class as well as in child class with same name and same number of arguments/input parameter is called as method overriding.
In overriding method resolution always take care by JVM based on
Run time object no based on reference type that why it is also known as run time polymorphism or dynamic Polymorphism or late binding.
The process of compiler trying to resolve the method call from given overloaded method definitions is called overload resolution.

In Java, **method overloading** is a feature that allows a class to have more than one method with the same name, but with different parameter lists (method signatures). It is a way to achieve compile-time polymorphism, where the correct method to call is determined at compile time based on the method's parameters.

Rules for Method Overloading in Java:

1. Different Number of Parameters: Methods can be overloaded by changing the number of parameters.

2. **Different Parameter Types**: Methods can be overloaded by changing the type of parameters.

3. **Order of Parameters**: Methods can also be overloaded by changing the order of parameters if the types are different.

4.**Return Type Does Not Matter**: Overloading is not determined by the return type of the method. Changing only the return type will result in a compilation error.

5.**Access Modifiers and Static Methods**: The access modifier and the static keyword don't affect method overloading. You can overload static, private, final methods as well as non-static methods.

In Java, **method overriding** allows a subclass to provide a specific implementation of a method that is already defined by its parent class. The overriding method must have the same name, return type, and parameters as the method in the parent class.

Method overriding is used to achieve **runtime polymorphism**.

Rules for Method Overriding in Java:

1. **Same Method Signature**: The method in the child class must have the same name, return type, and parameter list as the method in the parent class.
2. **Access Modifier**: The access modifier of the overriding method cannot be more restrictive than the method in the parent class.
For example:
If the parent method is protected, the overriding method can be protected or public, but not private.
3. **Return Type**: The return type of the overriding method should either be the same as the parent method or a subtype (Child type also allowed) (covariant return type).
4.**Static Methods**: Static methods cannot be overridden. If a static method is redefined in a subclass, it's considered method hiding, not overriding.
5.**Final Methods**: A final method cannot be overridden. If a method in the parent class is marked as final, the subclass cannot provide its own implementation.
6. **Private Methods**: Private methods cannot be overridden as they are not accessible outside their class.
7. **Constructor Cannot Be Overridden**: Constructors cannot be overridden because they are not inherited by subclasses.

Difference between Method overloading and method overriding

Parameter	Method Overloading	Method Overriding
Definition	When the same method name is used with different input parameters (number,	When the child class provides a specific implementation of a method already defined in the parent class.

Purpose	To increase method flexibility (compile-time	To customize/modify parent method behavior (runtime polymorphism).
Method Name	Must be same.	Must be same.
Input Parameters	Must be different (number, type, or order).	Must be same (exact signature).
Return Type	No restriction — can be same or different.	Must be same OR covariant (returning child class object).
Access Modifier	No rules. Any modifier allowed.	Cannot reduce visibility (private → default → protected → public order).
Private/Final/ Static	Can be overloaded.	Cannot be overridden (but static and private methods can be hidden, not
Method	Done by Compiler.	Done by JVM at runtime.
Polymorphism	Compile-time	Runtime polymorphism
Binding Type	Static binding / Early	Dynamic binding / Late binding
Class Involved	Only one class	Always two classes (inheritance

Difference between Method hiding and Method Overriding

Parameter	Method Hiding	Method Overriding
Method Type	Both parent & child methods must be static.	Both parent & child methods must be non-static.
Inheritance?	Yes, but static methods are hidden, not overridden.	Yes, overriding requires inheritance.
Binding Type	Compile-time (static binding).	Runtime (dynamic binding).
Method Resolution	Done by compiler based on reference type.	Done by JVM based on object type.
Polymorphism Type	Compile-time polymorphism / Early binding.	Runtime polymorphism / Late binding.
Execution Based On	Reference type (Parent ref → Parent method).	Object type (Parent ref → Child method).
Usage	To give separate implementations for static behaviors.	To change or enhance inherited method logic.

3. Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract Class

A class which is declared with abstract keyword is known as abstract class.
It can have abstract and non abstract methods (Method body).

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

There are two ways to achieve abstraction in java

1.Abstract class (0 to 100%)
2. Interface (100%)

Rules of abstract class
1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non abstract methods
3. We can not crate the object of abstract class
4. It can have constructor and static method methodAbstract method
A method which is declared as abstract and does not have imply is known as the abstract method.

Concrete class:
A class which provides definations for all the incomplete methods which are present in abstract class with the help of extends keywords is called concrete /child class.

Abstraction real time example :
Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators 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.
There are two ways in which the objects can be initialised while inheriting the properties of the parent and child classes. They are:

Child c = new Child() : The use of this initialisation is to access all the members present in both parent and child classes, as we are inheriting the properties.

Parent p = new Child() : This type of initialisation is used to access only the members present in the parent class and the methods which are overridden in the child class. This is because the parent class is upcasted/top casting to the child class.

Interface :

The interface in Java is a mechanism to achieve abstraction. There can be only abstract

methods in the Java interface, not method body.
It is used to achieve abstraction and multiple+hybrid inheritance in Java.
In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also represents the IS-A relationship.

Features of interface.

1. Variables declared inside the interface are by default **static** and **final**.
2. Methods declared inside the interface are by default **public** and **abstract**.
3. Constructor concept is not applicable for interface.
4. Object of interface can not be created.
5. Interface supports multiple inheritance.
6. We can achieve 100% abstraction in interface till (1.7)

Implementation class :

A class which provides definitions for all the incomplete methods which are present in interface with the help of implements keyword is called Implementation class
Java 8 & 9 interface Improvement :
Since Java 8, we can have default and static methods in an interface. Since Java 9, we can have private methods in an interface.

Key Rules of Interfaces in Java

1. Method Signatures Only (No Method Bodies):
◦ By default, methods in interfaces are public and abstract, meaning they have
no body.

2◦ Implementing classes must provide the method body.No Instance Fields (Only Constants Allowed):
◦ Interfaces can only define constants, which are public, static, and final by
default.

3. Multiple Inheritance via Interfaces:
◦ Java does not support multiple inheritance with classes, but a class can implement
multiple interfaces.

4. Default and Static Methods:
◦ Since Java 8, interfaces can have default methods (with a method body),
allowing you to add new functionality to interfaces without breaking existing implementations.
◦ static methods can also be defined in interfaces.

5. No Constructors:
◦ Interfaces do not have constructors because they cannot be instantiated directly.
Instead, classes that implement the interface instantiate it.

Difference between Interface and Abstraction

Interface	Abstract Class
If we only know *what* to do but not *how*, we use Interface.	If we know partial implementation (some logic) and want to force children to implement the rest, we use Abstract Class.
All methods are public and abstract by default (before Java 8). From Java 8 → default & static	Methods can be abstract or concrete (normal method). No restriction on modifiers.
Methods **cannot** use private, protected; cannot be final, static, synchronized for abstract	Abstract class methods can use any modifier (private, protected, public, static, final).
Variables are **public, static, final** always (constants).	Variables can be normal (instance), static, final — any modifier allowed.
Interface variables **must be initialized** during declaration.	Abstract class variables **do not need initialization** at declaration time.
Interface **cannot have constructors**.	Abstract class **can have constructors**.
Cannot declare instance blocks or static blocks.	Can have instance blocks and static blocks.
A class can implement **multiple interfaces** → supports multiple inheritance.	A class can extend **only one abstract class** → single inheritance.
Used for **100% abstraction** (conceptually), because no state/data except constants.	Used for **0–100% abstraction**, can contain shared logic + abstract methods.
Supports multiple inheritance through	Does **not** support multiple inheritance through
Designed for behavior specification.	Designed for code reuse + behavior specification.

4. Encapsulation?

Encapsulation is the process of protecting data inside a class and allowing access only through methods.
It binds data (variables) + methods (functions) together in one unit (class).

Encapsulation = Data Hiding + Controlled Access
It ensures:

- Users cannot access data directly
- Users can access data **only through methods**
- Internal data remains safe, consistent, and secure

Why Encapsulation is Important?

1. **Protects data from unauthorized access**
   Example: Bank balance cannot be changed directly.
2. **Controls how data is modified**
   Example: Set age only if it is > 0.
3. **Helps maintain data integrity**
   Example: Salary should not be negative.
4. **Widely used in real-world applications**
   Banking apps, ATM systems, medical systems, billing systems, etc.

How to achieve Encapsulation in Java?
To achieve encapsulation:

1. Declare variables as private
2. Provide public getter and setter methods to access them

Encapsulation hides these variables as **private**, and allows updates only through **setter methods with validation**.

Encapsulation is one of the pillars of OOP.
It means wrapping data (variables) and methods in a single unit and restricting direct access to internal data.
We achieve this by declaring variables as private and accessing them using getters and setters.
It provides data security, data hiding, and helps maintain consistency."

A good real-time example is banking applications.
The balance is private; users cannot modify it directly.
Only deposit() and withdraw() methods can change the balance, ensuring proper validation and secure access to data."

Access Modifiers or specifiers

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	**Same Class**	**Same Package (Non-Subclass)**	**Subclass (Different**	**Different Package (Non-Subclass)**	**Description**
public	✔ Allo	✔ Allowed	✔ Allowed	✔ Allowed	Accessible everywhere.
protected	✔ Allowed	✔ Allowed	✔Allowed (through inheritance)	❌ Not accessible without inheritance	Visible within same package and subclasses.
default *(no modifier)*	✔ Allowed	✔ Allowed	❌ Not allowed	❌ Not allowed	Package-private: only within the same package.
private	✔ Allo	❌ Not allowed	❌ Not allowed	❌ Not allowed	Accessible only inside the same class.