# OOPs

26 February 2024 18:04

## *What is Object Oriented Programming?*

- *OOP is abbreviated as object oriented programming*
- *Object oriented Programming is a style of developing real world application.*
- *In OOP, we will consider everything has a object and class*
- *In Order to implement OOP,*
    - *First, we need to identify the objects*
    - *Then, We need to describe about those objects*
    - *Then, we need to establish the relationship between the objects*
    - *Once these are done we will implement it via class and object using programming language that supports OOPS*
- *Using OOPS we can make our code scalable/maintainable and easy to refactor*
- *Apart from Classes and Object, Encapsulation, Inheritance, Polymorphism and Abstraction are the major principles of OOPS where,*
    - *Inheritance allows us to get the properties of the super class*
    - *Encapsulation provides data security by hiding the implementation details of classes and it also binds the data and method into single unit*
    - *Abstraction allows the creation of complex systems by hiding unnecessary details*
    - *And polymorphism enables the use of the same code for different types of objects.*

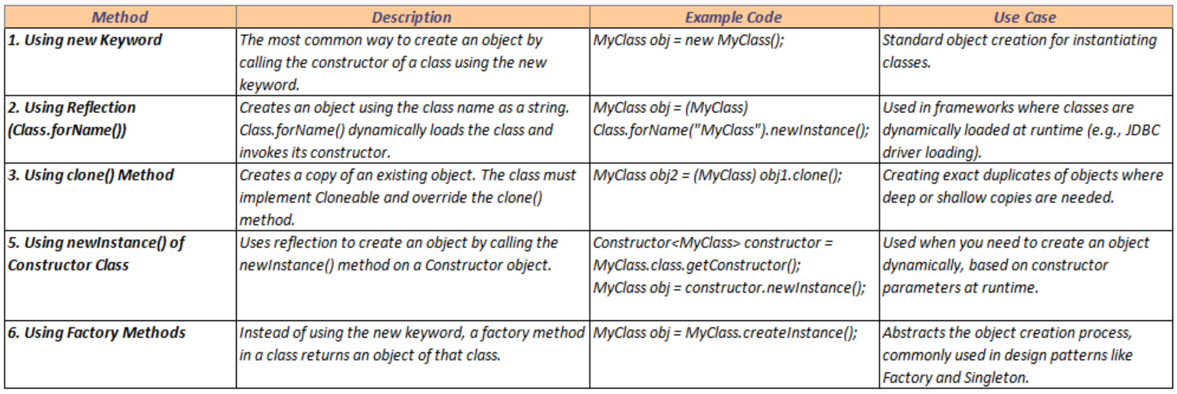
## *What is a Class?*

- *A class is a blueprint or template for creating objects. It defines the structure and behaviour of the objects that will be created from it. A class can contain:*
    - ***Fields** (also known as attributes, properties, or instance variables) which represent the data or state of the object.*
    - ***Methods**, which define the behaviour or actions that the object can perform*

## *What is an Object?*

- *An **object** is an instance of a class. When you create an object from a class, the object has its own values for the attributes defined in the class, and it can use the methods to interact with or manipulate these values. An object represents a **real-world entity** or a thing in your program. It has **state** (represented by attributes/fields) and **behaviour** (represented by methods).*

- *Different ways to create the object in java*



| Method | Description | Example Code | Use Case |
|---|---|---|---|
| **1. Using new Keyword** | The most common way to create an object by calling the constructor of a class using the new keyword. | MyClass obj = new MyClass(); | Standard object creation for instantiating classes. |
| **2. Using Reflection (Class.forName())** | Creates an object using the class name as a string. Class.forName() dynamically loads the class and invokes its constructor. | MyClass obj = (MyClass) Class.forName("MyClass").newInstance(); | Used in frameworks where classes are dynamically loaded at runtime (e.g., JDBC driver loading). |
| **3. Using clone() Method** | Creates a copy of an existing object. The class must implement Cloneable and override the clone() method. | MyClass obj2 = (MyClass) obj1.clone(); | Creating exact duplicates of objects where deep or shallow copies are needed. |
| **5. Using newInstance() of Constructor Class** | Uses reflection to create an object by calling the newInstance() method on a Constructor object. | Constructor<MyClass> constructor = MyClass.class.getConstructor(); MyClass obj = constructor.newInstance(); | Used when you need to create an object dynamically, based on constructor parameters at runtime. |
| **6. Using Factory Methods** | Instead of using the new keyword, a factory method in a class returns an object of that class. | MyClass obj = MyClass.createInstance(); | Abstracts the object creation process, commonly used in design patterns like Factory and Singleton. |

## *ENCAPSULATION*

- *Encapsulation is programming concept which is used to hide data and controlling the access. We can achieve this by declaring the data members as private and providing the getter and setter method to access the data member. In short it is more about data protection.*
- ***In our framework we are implemented encapsulation for web elements. We are declaring the web elements as private and provided a getter method for that. If some needs to access the web element they have to access via getter of that particular web element***
- ***Access Specifier or modifiers:***
    - ***public**: Accessible from anywhere.*
    - ***protected**: Accessible within the same package and by subclasses.*
    - ***default**: Accessible only within the same package (no modifier).*
    - ***private**: Accessible only within the same class.*

## *ABSTRACTION*

- *Abstraction is a programming concept that involves hiding unnecessary details and exposing only essential information. It allows the creation of complex systems by breaking them down into smaller, manageable parts. In order to achieve abstraction we need to create a interface or abstract class and Implement abstract methods in subclasses and create object for the sub class and access the required methods. **A typical example can i say is in API testing we can create a interface along the abstract method for each type of request like get, post etc. Then we will implement these Interface in Different Classes like RestAPI, SoapAPI***
- ***Have to use abstract classes** when you want to provide some default behaviour and let subclasses fill in the specific details.*
- ***Have to use interfaces** when you want to define a contract that multiple classes can implement, regardless of their position in the class hierarchy.*
- *In Java, abstraction is achieved by interfaces and abstract classes. Using interfaces, we can achieve 100% abstraction.*

## *INHERITANCE*

- *Inheritance is a programming concept. It allows the subclass or child class to inherit the fields (variables) and methods (functions) from the superclass or parent class. It is mainly used for code reusability, Maintainability and Extensibility. To implement this , we have to write the common functionality in the superclass and shared with all subclasses using the extends keyword.*
- *We can't inherit the private and static members*
- ***For example, In our framework, we have a BaseTest class that contains common methods like setupDriver(), tearDown(), and takeScreenshot(). Every individual test class (like LoginTest, SearchTest) can inherit these methods and use them without needing to redefine them in every test class by extending the BaseTest.***

- ***Different types of Inheritance***

  ***Single-level**: When a class inherits from one parent class*

  ***Multi-level**: When a class inherits from a class that is itself inherited from another class.*

  ***Hierarchy** - When multiple classes inherit from one parent class*

  ***Multiple** - When a class inherits from more than one parent class (Java does not support this type of inheritance (i.e., a class cannot inherit from more than one class) to avoid complexity and ambiguity. However, this can be achieved using interfaces)*
- *We can  implement from interface to class but vice versa in not possible ie., class to interface*

- ***Why multiple Inheritance is not supported in Java?***
  - *Syntax is not supported*
  - *Multiple super statements are not allowed in constructor*
  - *Ambiguity of diamond problem*

## *POLYMORPHISM*

- *Process of allowing objects to take on multiple forms and enabling a single method or interface to work in different ways depending on the context. It used to achieve the code flexibility, maintainability, and reusability in Java.*
- *There are two types of Polymorphism in Java*

  ***1. Compile time polymorphism (Static binding)***
  - *Method declaring and definition binded by complier at the time of compiling*
  - *Method overloading is good example for this*

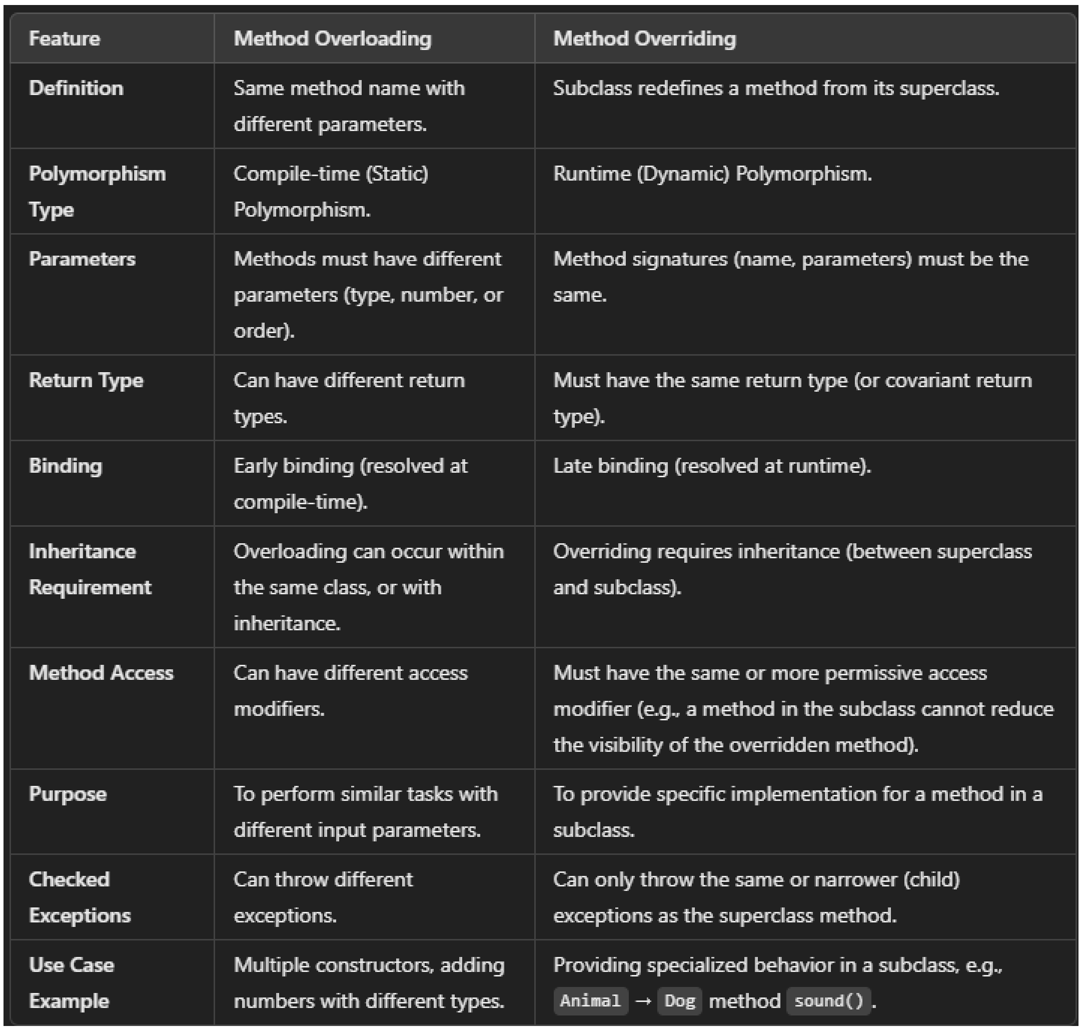
  ***2. Runtime polymorphism (Dynamic binding)***
  - *Method declaring and definition binded by JVM at run time.*
  - *Method overriding is good example for this*

## *METHOD OVERLOADING*

- *Process of developing the methods with the same name but with different arguments is known as method overloading where arguments can differ in length, sequence or by datatype. If we want to perform same tasks but with different input we use method overloading*
- *We can't overload the private, static methods outside the class*
- ***Implicit wait** is an example of overloading. In Implicit wait we use different time stamps such as SECONDS, MINUTES, HOURS etc.,*
- ***Action class** in Selenium, **Assert class** in TestNG are good example of overloading.*

## *METHOD OVERRIDING*

- *Process of inheriting the method and changing its implementation in sub class according to the sub class specification is known as method overriding while doing this we are not allowed to change the method signature meaning return type, method name and argument list should not be altered. For doing this inheritance is must. If we want to provide different implementation in different classes we go for method overriding*
- *We can't override the private, static and final members*
- ***Handling different browsers (Chrome, Firefox, Safari, etc.) in Selenium WebDriver is good example for this***



| Feature | Method Overloading | Method Overriding |
|---|---|---|
| **Definition** | Same method name with different parameters. | Subclass redefines a method from its superclass. |
| **Polymorphism Type** | Compile-time (Static) Polymorphism. | Runtime (Dynamic) Polymorphism. |
| **Parameters** | Methods must have different parameters (type, number, or order). | Method signatures (name, parameters) must be the same. |
| **Return Type** | Can have different return types. | Must have the same return type (or covariant return type). |
| **Binding** | Early binding (resolved at compile-time). | Late binding (resolved at runtime). |
| **Inheritance Requirement** | Overloading can occur within the same class, or with inheritance. | Overriding requires inheritance (between superclass and subclass). |
| **Method Access** | Can have different access modifiers. | Must have the same or more permissive access modifier (e.g., a method in the subclass cannot reduce the visibility of the overridden method). |
| **Purpose** | To perform similar tasks with different input parameters. | To provide specific implementation for a method in a subclass. |
| **Checked Exceptions** | Can throw different exceptions. | Can only throw the same or narrower (child) exceptions as the superclass method. |
| **Use Case Example** | Multiple constructors, adding numbers with different types. | Providing specialized behavior in a subclass, e.g., `Animal` → `Dog` method `sound()`. |