**OOP Principles**

**Object-Oriented Programming System (OOPs)** is a programming concept that works on the principles of **abstraction**, **encapsulation**, **inheritance**, and **polymorphism**.

The main aim of object-oriented programming is to implement **real-world entities.**

**Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object has state and behaviour. An object can be defined as an instance of a class, and there can be multiple instances of a class in a program.

A **class** can also be defined as a blueprint from which you can create an individual object. **Class** doesn't consume any space. It is only a logical component and not the physical entity.

**Abstraction** is a process of **hiding** the implementation details and providing only pure functionality to the user.

In Java, abstraction is achieved by using the **abstract** keyword for classes and interfaces.

Abstraction describes selecting data from a larger pool to show only **relevant** details of the object to the user. It helps to reduce programming complexity and efforts.

An **interface** can have all the methods and variables like the class do, but the **methods** in the interface are **abstract** by default.

**Interfaces** in Java are used to specify a behaviour that **classes** must **implement**.

We use **interfaces** when we want to achieve **security**.

As we know, Java does not support "**multiple inheritance**". However, we can achieve this easy with **interfaces**, because the class can **implement** multiple interfaces.

In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.

**Default interface methods** are an efficient way to deal with this issue. They allow us to add new methods to an interface that are automatically available in the implementations. Therefore, we don't need to modify the implementing classes.

**Since static interface methods** don't belong to a particular object, they're not part of the API of the classes implementing the interface; therefore, they have to be called by using the interface name preceding the method name. Static methods in interfaces **make it possible to group related utility methods**, without having to create artificial **utility classes** that are simply placeholders for static methods.

**Inheritance** is a method in which one object acquires/inherits another object’s properties, and inheritance also supports hierarchical classification. The idea behind this is that we can create new classes built on existing classes, i.e., when you inherit from an existing class, **we can reuse methods and fields of the parent class**. Inheritance represents the **parent-child relationship**.

**Encapsulation** allows to keep fields within a class private, then providing access to those fields via public methods. Encapsulation is a protective barrier that keeps the data and code safe within the class itself.

**Access Modifier**: Defines the access type of the method i.e. from where it can be accessed in your application. In Java, there are 4 types of access specifiers:

**public**: Accessible in all classes in your application.

**protected**: Accessible within the package in which it is defined and in its subclass(es) (including subclasses declared outside the package).

**private**: Accessible only within the class in which it is defined.

**default -package private:** **When we don't use any keyword explicitly** (declared/defined without using any modifier): Accessible within the same class and package within which its class is defined.

**Polymorphism** is the ability of an object to take on different forms. In Java, polymorphism refers to the ability of a class to provide different implementations of a method, depending on the type of object that is passed to the method.

There are two types of polymorphism dynamic and static A.K.A runtime and compile time.

The **static** polymorphism can be achieved by method overloading. **Overloading** allows different methods to have **the same name**, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to **compile-time (or static) polymorphism**.

**Dynamic polymorphism**: Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature, and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

**SOLID Principles**

In Java, the classes are the **building blocks** of any application.

If these blocks are not strong, the building (i.e. the **application**) is going to face a tough time in the future.

When the developer builds a software following the **bad design**, the code can become inflexible and more brittle, **small changes in the software can result in bugs**.

For these reasons, we should follow **SOLID** Principles.

**SOLID** Principles is a **coding standard** that all developers should have a **clear concept** for developing software in a **proper way** to **avoid a bad design**.

It was promoted by **Robert Martin** and is used across the object-oriented design spectrum.

When applied properly it makes your code more **extendable**, **logical** and **easier to read**.

**S.O.L.I.D** represents five principles of Java which are:

* **S** – **Single responsibility** principle – class should only have one responsibility
* **O** – **Open–closed** principle – open for extension, but closed for modification
* **L** – **Liskov substitution** principle – objects should be replaceable with instances of their subtypes without altering the correctness of that program
* **I** – **Interface segregation** principle – many specific interfaces are better than one general interface
* **D** – **Dependency inversion** principle – one should depend upon abstractions, not concretions

**For more detailed explanation of each SOLID Principle see the “solid\_principles\_explained” folder in the java**-**opp repository.**[**https://github.com/Kaloyan86/java-oop/tree/master/src/solid\_principles\_explained**](https://github.com/Kaloyan86/java-oop/tree/master/src/solid_principles_explained)