**Notes - SOLID principles, Object Oriented Design, and Design Patterns**

Contents

[SOLID 1](#_Toc499898216)

[Best Practices for OOPs Concepts 1](#_Toc499898217)

[What Is an Interface? 2](#_Toc499898218)

[Interface Vs Abstract Class? 2](#_Toc499898219)

# SOLID

It is a [mnemonic](https://en.wikipedia.org/wiki/Mnemonic) [acronym](https://en.wikipedia.org/wiki/Acronym) for five [design](https://en.wikipedia.org/wiki/Object-oriented_design) principles intended to make software designs more understandable, flexible and [maintainable](https://en.wikipedia.org/wiki/Software_maintenance).

|  |  |
| --- | --- |
| **Initial** | **Concept** |
| **S** | [Single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle)[[4]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-4)  A [class](https://en.wikipedia.org/wiki/Class_(computer_science)) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class). |
| **O** | [Open/closed principle](https://en.wikipedia.org/wiki/Open/closed_principle)[[5]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-5)  “software entities … should be open for extension, but closed for modification.” |
| **L** | [Liskov substitution principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle)[[6]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-6)  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also [design by contract](https://en.wikipedia.org/wiki/Design_by_contract). |
| **I** | Interface segregation principle[7]  “many client-specific interfaces are better than one general-purpose interface.”[[8]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-martin-design-principles-8) |
| **D** | [Dependency inversion principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle)[[9]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-9)  one should “depend upon abstractions, [not] concretions.”[[8]](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)#cite_note-martin-design-principles-8) |

[**Design patterns**](https://sourcemaking.com/design_patterns) are generally **reusable solutions** for certain scenarios, that can **come in handy to solve commonly occurring problems**, and can hugely help us optimize our code.

**KISS: Keep it Simple, Stupid and Loosely Coupled Code**

Any code you write needs to stay clean and simple. Easy to read by other software developers and easy to change.

# Best Practices for OOPs Concepts

Since the aim of OOPs concepts in Java is to save time without sacrificing security and ease of use, the best practices are all oriented toward advancing that main goal.

* **DRY (Don’t Repeat Yourself).** This is the core concept in Java. You should never have two blocks of identical code in two different places. Instead, have one method you use for different applications.
* If you expect your Java code to change in the future, encapsulate it by making all variables and methods private at the outset. As the code changes, increase access to **“protected”** as needed, but not too **public.**
* **Single Responsibility.** Another best practice for OOPs concepts in Java is the Single Responsibility Principle. Simply put, a class should always have only one functionality. That way, it can be called and/or extended on its own when new uses arise for it, without causing **coupling** between different functionalities.
* **Open Closed Design.** Make all methods and classes Closed for modification but Open for an extension. That way, tried and tested code can remain static but can be modified to perform new tasks as needed.

# What Is an Interface?

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies.

|  |  |
| --- | --- |
|  | If you work with interfaces then you can remain independent of the actual implementation. |
|  | One simple example: you want to communicate with a device but you don't know if it will be via Ethernet, COM or USB. You define one interface and 3 implementations. At runtime you could then select which method you want and the factory will give you the appropriate implementation. |

# Interface Vs Abstract Class?

The abstract class inheritance is used when the derived class shares the core properties and behaviour of the abstract class. The kind of behaviour that actually defines the class.

On the other hand interface inheritance is used when the classes share peripheral behaviour, ones which do not necessarily define the derived class.

For eg. A Car and a Truck share a lot of core properties and behaviour of an Automobile abstract class, but they also share some peripheral behaviour like Generate exhaust which even non automobile classes like Drillers or PowerGenerators share and doesn't necessarily defines a Car or a Truck, so Car, Truck, Driller and PowerGenerator can all share the same interface IExhaust.

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |