**SOLID Principles :** the SOLID principles of Object-Oriented Design.

what is SOLID and how does it help us write better code? Simply put, Martin's and Feathers' design principles encourage us to create more maintainable, understandable, and flexible software.

The following 5 concepts make up our SOLID principles:

* Single Responsibility
* Open/Closed
* Liskov Substitution
* Interface Segregation
* Dependency Inversion

**Single Responsibility :** this principle states that ****a class should only have one responsibility.****we have a separate class dedicated to this one concern.

How does this principle help us to build better software? Let's see a few of its benefits:

**Testing**– A class with one responsibility will have far fewer test cases

**Lower coupling** – Less functionality in a single class will have fewer dependencies

**Organization**– Smaller, well-organized classes are easier to search than monolithic ones

Take, for example, a class to represent a simple book:

in this code, we store the name, author, and text associated with an instance of a Book.

public class Book {

private String name;

private String author;

private String text;

//constructor, getters and setters

// methods that directly relate to the book properties

public String replaceWordInText(String word){

return text.replaceAll(word, text);

}

public boolean isWordInText(String word){

return text.contains(word);

}

}

public class BookPrinter {

// methods for outputting text

void printTextToConsole(String text){

//our code for formatting and printing the text

}

void printTextToAnotherMedium(String text){

// code for writing to any other location..

}

}

**Open/Closed : Open for Extension, Closed for Modification**

Now, time for the ‘O' – more formally known as the open-closed principle.classes should be open for extension, but closed for modification. In doing so, we stop ourselves from modifying existing code

As part of a new project, imagine we've implemented a Guitar class.We launch the application, However, after a few months, we decide the add some new funcitonality in Guitar Class .  just open up the Guitar class and add a new functionality  – but who knows what errors that might throw up in our application.

Instead, of modify Guitar class let's stick to the open-closed principle and **simply extend our Guitar class:**

public class SuperCoolGuitarWithFlames extends Guitar {

private String flameColor;

//constructor, getters + setters

}

By extending the Guitar class we can be sure that our existing application won't be affected.

**Liskov Substitution :** I**f class**A**is a subtype of class**B**, then we should be able to replace**B **with**A **without disrupting the behavior of our program.**

public interface Car {

void turnOnEngine();

void accelerate();

}

Above, we define a simple Car interface with a couple of methods that all cars should be able to fulfill – turning on the engine, and accelerating forward.

Let's implement our interface and provide some code for the methods.

**Interface Segregation :** The ‘I ‘ in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

**Dependency Inversion : The principle of Dependency Inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.**