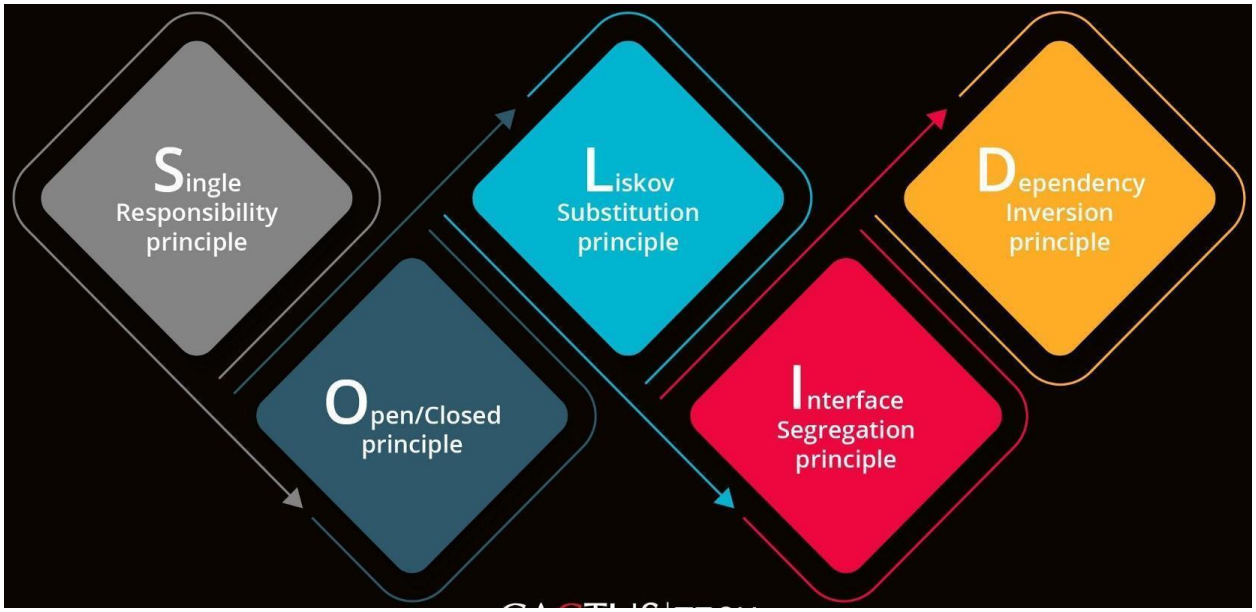
The Understanding Of The SOLID Principles With Functional Programming



Introduction

SOLID principles come into picture because these are the design principles that help us in encouraging creating more maintainable, understandable, and flexible software.

As our applications grow in size, we can reduce their complexity by using **SOLID** principles.

SOLID Principles:-

The following five concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

These five software development principles are guidelines to follow when building software so that it is easier to scale and maintain. They were made popular by a software engineer, Robert C. Martin.

BENEFITS OF SOLID PRINCIPLES-

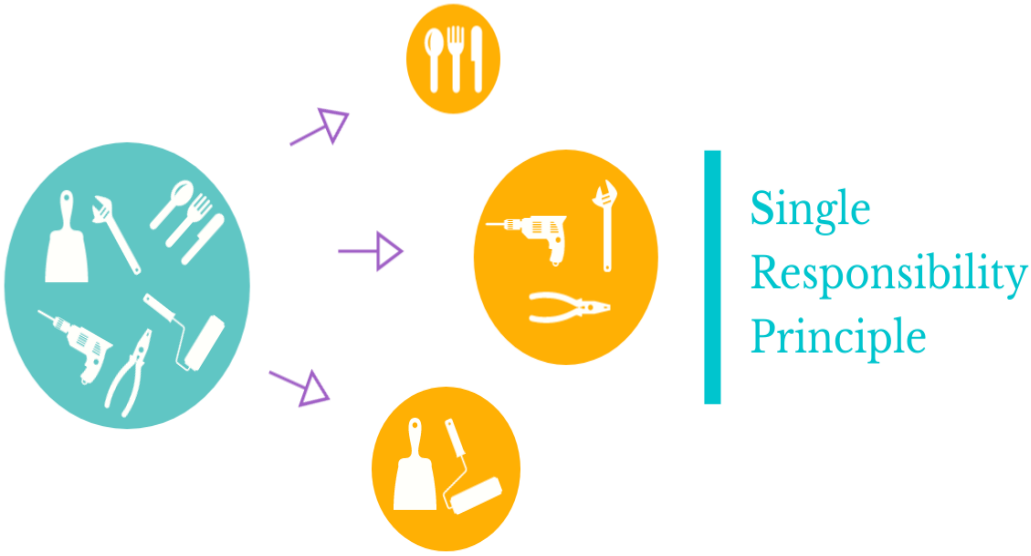
Some of the benefits SOLID Principle holds are as follows:-

- ❖ Loose Coupling
- ❖ Code Maintainability
- ❖ Dependency Management

The SOLID Principles:

1. S — Single Responsibility:

Define:- A class should have a single responsibility.



- If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities could affect the other ones without you knowing.
- functional programming languages don't have classes; the same principle holds true. Functions should be small reusable pieces of code that you can compose freely to create complex behavior.
- This can be extracted to almost anything, once your functions are small, the modules where they are located they should also form a cohesive closure that does only one thing and does it well.
- As long as your function or class or module has only one reason to change then you are applying this principle.

Code Snippet:-

```java
package SingleResponsibility;
public class Book_issued {
    String date= "09/08/2021";
    void issuedBook(String name,String n){

        System.out.println("The book "+name+" issued to "+n+" on "+date);
    }
    public static void main(String[] args) {
        Book B1 =new Book();
        User u1 = new User();
        Book_issued i1 =new Book_issued();
        i1.issuedBook(B1.name,u1.name);
    }
}
```

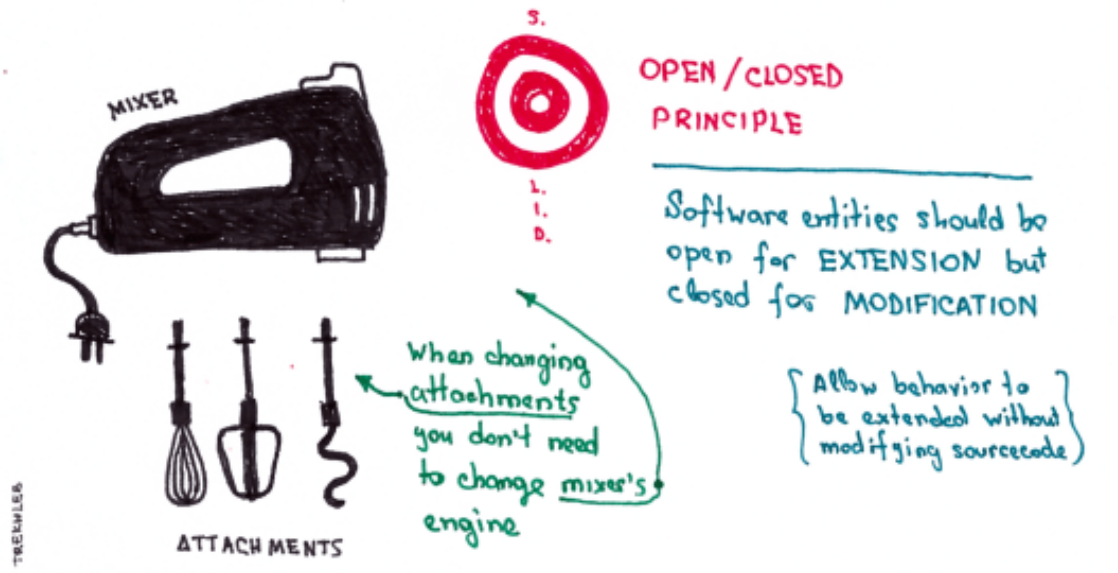
```java
    public String getName()
    {return this.name;}

    public String getAuthor()
    {return this.author;}

    public void setId(int id)
    {this.id=id;}

    public void setName(String name)
    {this.name=name;}

    public void setAuthor(String author)
    {this.author=author;}
    public Book(){
        setId(123);
        setName("Harry Potter");
        setAuthor("J.K. Rowlings");
    }

    void showDetails(){

        System.out.println("The ID of the Book is "+getId());
        System.out.println("The Name of the Book is "+getName());
        System.out.println("The Author of the Book is "+getAuthor());
    }
}
```

Goal:

This principle aims to separate behaviours so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.

2. O — Open-Closed:

Define:- Classes should be open for extension, but closed for modification



- Changing the current behaviour of a Class will affect all the systems using that Class.

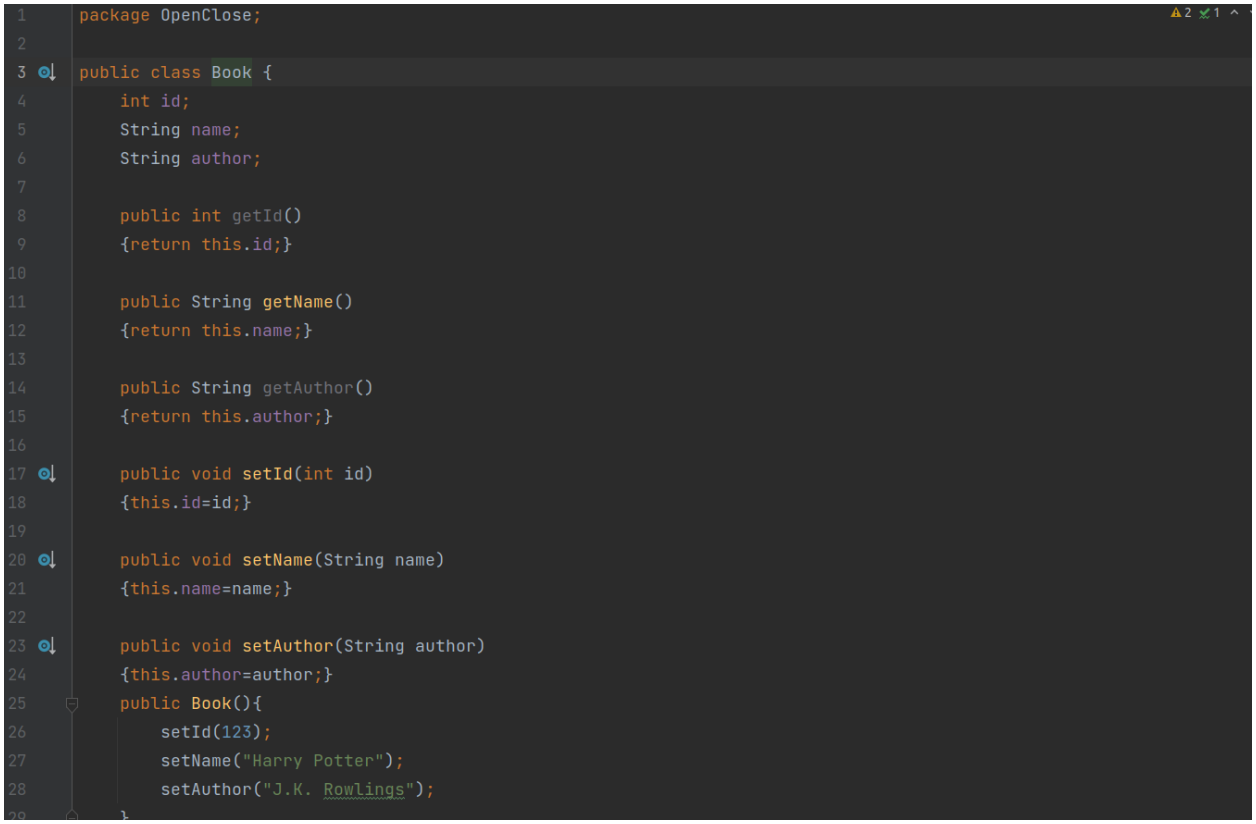
- If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.
- Instead of using inheritance, Functional Programming achieves this by using two tools. Composition to create new

behaviors from previously defined functions and higher-order functions to change functionality at runtime.

Code Snippet:-

```java
package OpenClose;



public class booksOpenClosed extends Book {
    String category;
    public String getCategory() { return this.category; }
    public void setId(int id) { this.id=id; }

    public void setName(String name) { this.name=name; }

    public void setAuthor(String author) { this.author=author; }

    public void setCategory(String category) { this.category=category; }

       booksOpenClosed(){
        setId(124);
        setName("Harry Potter");
       setAuthor("J.K. Rowlings");
        setCategory("Fiction");

    }
}


```

```java
package OpenClose;

public class Book_issued {
    String date= "09/08/2021";
    void issuedBook(String name,String userName,String c){

        System.out.println("The book "+name+" issued to "+userName+" on "+date);
        System.out.println("The category of "+name+" is "+c);
    }

    public static void main(String[] args) {
        booksOpenClosed B1 =new booksOpenClosed();
        User u1 = new User();
        Book_issued i1 =new Book_issued();
        i1.issuedBook(B1.getName(),u1.getName(),B1.category);
    }
}


```



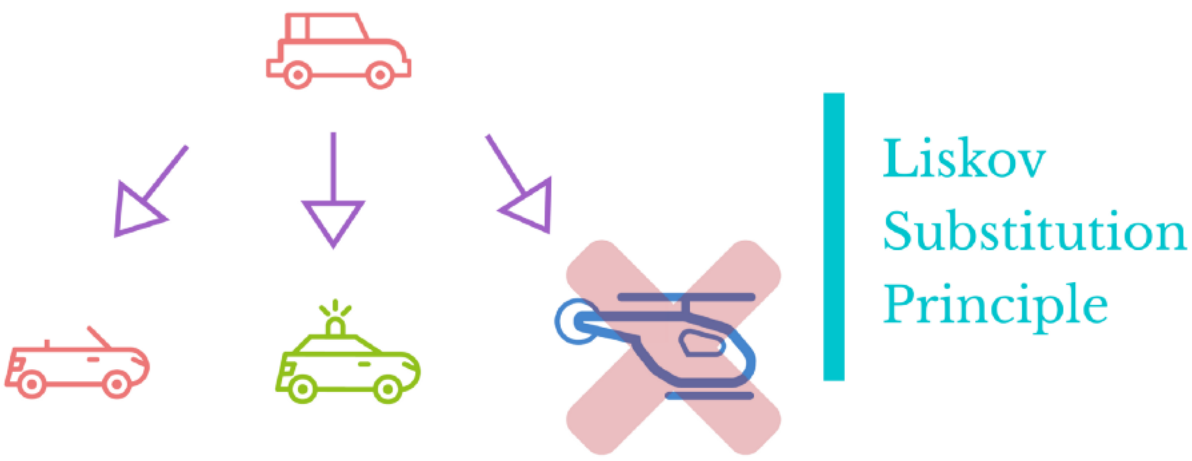
```java
package OpenClose;

public class Book {
    int id;
    String name;
    String author;

    public int getId()
    {return this.id;}

    public String getName()
    {return this.name;}

    public String getAuthor()
    {return this.author;}

    public void setId(int id)
    {this.id=id;}

    public void setName(String name)
    {this.name=name;}

    public void setAuthor(String author)
    {this.author=author;}
    public Book(){
        setId(123);
        setName("Harry Potter");
        setAuthor("J.K. Rowlings");
    }
```

Goal:

This principle aims to extend a Class's behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.

3. L — Liskov Substitution:

Define:- If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.



- When a child Class cannot perform the same actions as its parent Class, this can cause bugs.
- If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a child. The child Class should be able to do everything the parent Class can do. This process is called Inheritance.
- The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.
- The picture shows that the parent Class delivers Coffee(it could be any type of coffee). It is acceptable for the child Class to deliver Cappuccino because it is a specific type of Coffee, but it is NOT acceptable to deliver Water.
- If the child Class doesn't meet these requirements, it means the child Class is changed completely and violates this principle.
- LSP also applies in case we use generic or parametric programming where we create functions that work on a

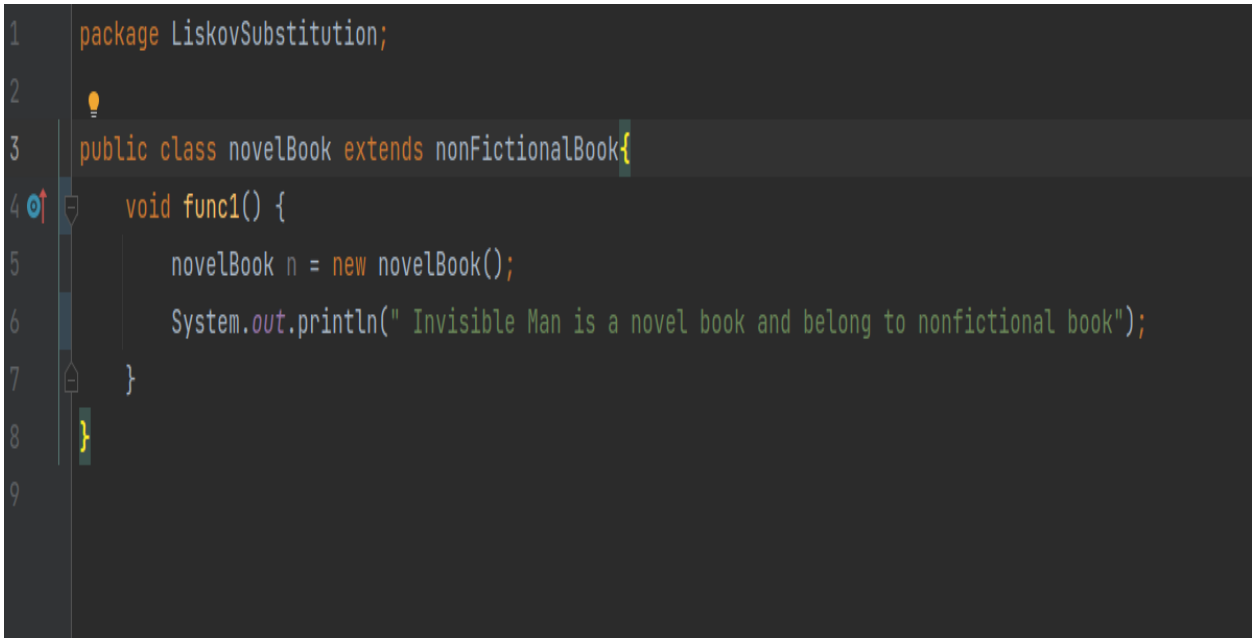
variety of types, they all hold a common truth that makes them interchangeable.

- This pattern is super common in functional programming, where you create functions that embrace polymorphic types (aka generics) to ensure that one set of inputs can seamlessly be substituted for another without any changes to the underlying code.

Code Snippet:-

```java
package LiskovSubstitution;

public class LSP {
    public static void main(String[] args) {
        nonFictionalBook f1 = new novelBook();
        f1.func1();

    }
}

```

```java
package LiskovSubstitution;

public class nonFictionalBook implements Book{

  void func1(){
        nonFictionalBook d = new nonFictionalBook();
        System.out.println("The Beauty Myth  is a nonFictional book");
   }
}

```



```java
package LiskovSubstitution;

public class novelBook extends nonFictionalBook{
    void func1() {
        novelBook n = new novelBook();
        System.out.println(" Invisible Man is a novel book and belong to nonfictional book");
    }
}
```

Goal:-

This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

Summary:-

So far, we have discussed these three principles and highlighted their goals. They are to help you make your code easy to adjust, extend and test with little to no problems.