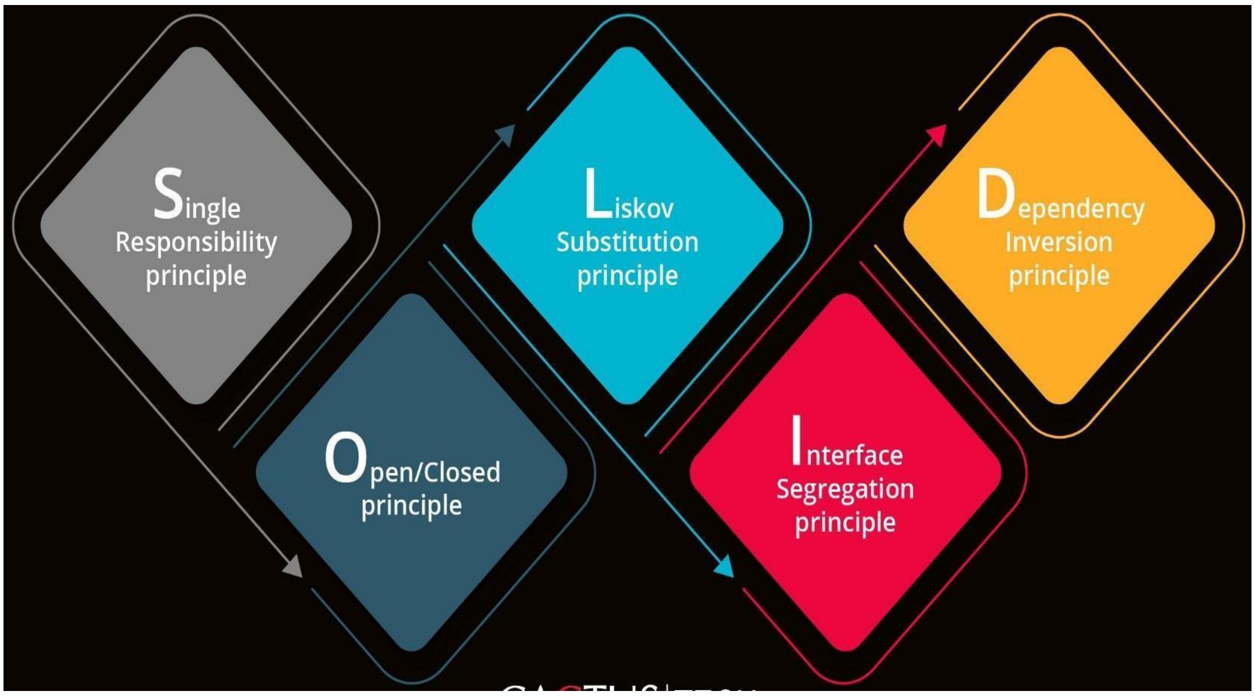
Code and Blog to showcase the understanding of the SOLID principles With Functional Programming


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

INTRODUCTION-

- The SOLID principles are basically the Five Principles used in the Object –Oriented Class Design and these are also the set of rules and best practices to follow while designing an any class structure.

- SOLID Principles were first introduced by Computer Scientist Robert J. Martin in 2000, but the acronym was introduced later by Michael Feathers.

- These SOLID principles were developed in paper named "Design Principles and Design Patterns".

WHY SOLID PRINCIPLES-

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

- It also helps us in growing our application according to the size, and can reduce the complexity of the code.

SOLID PRINCIPLES-

- This principle is an acronym of the five principles of Object Oriented Designs. These are as –

  - Single Responsibility Principle (SRP)
  - Open/Closed Principle (OCP)
  - Liskov’s Substitution Principle (LSP)
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)

- These are helps us in reducing Tight Coupling. Tight Coupling simply means a group of classes that are highly dependent on one another and SOLID Principles helps you to avoid that in your code.

- Loosely Coupled Classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable.

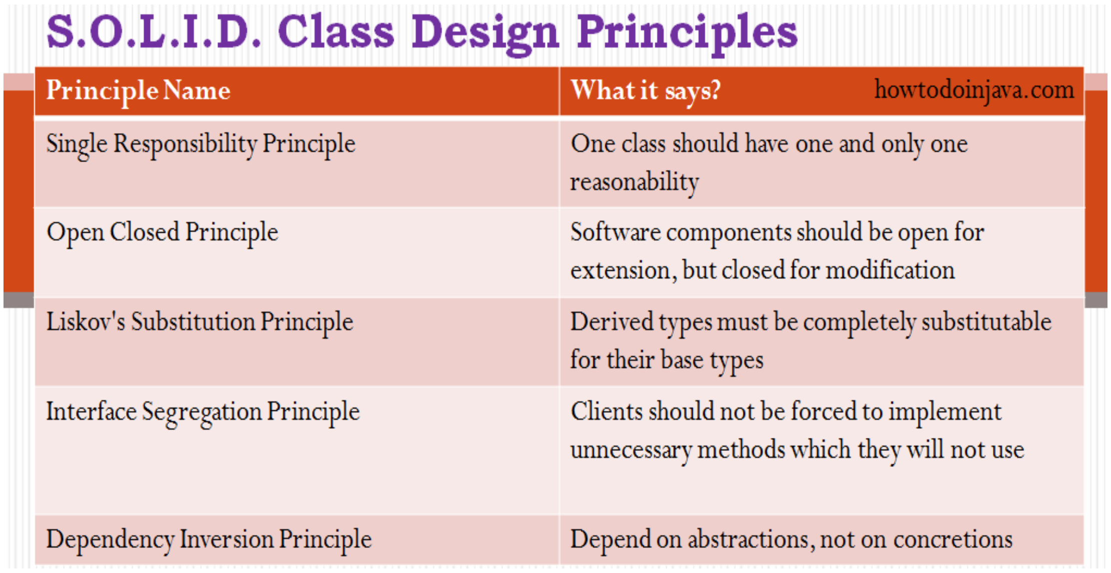
BENEFITS OF SOLID PRINCIPLES-

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

- Loose Coupling
- Code Maintainability
- Dependency Management

EXPLANATION-

Let’s see the SOLID Principles in brief:-



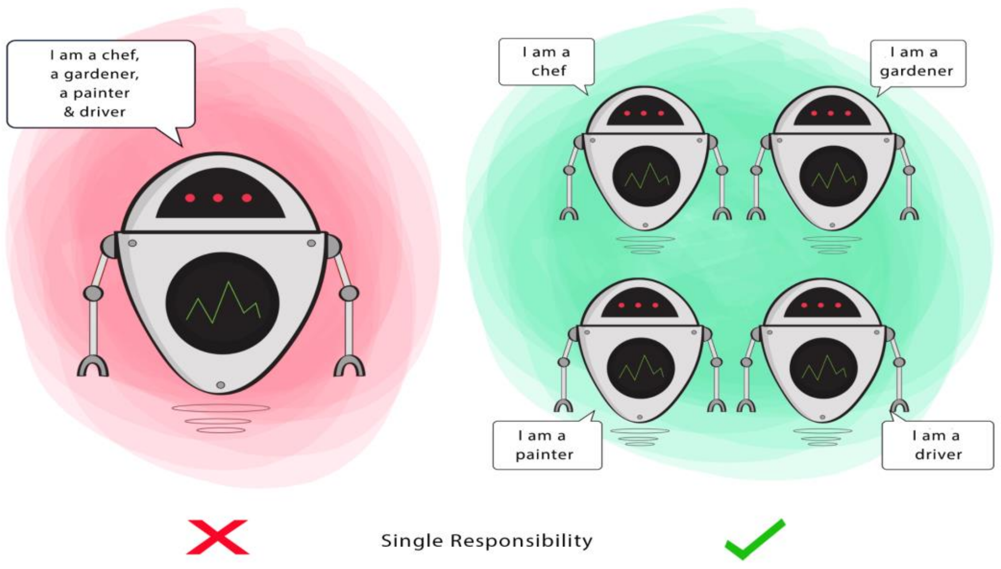
S.O.L.I.D. Class Design Principles

Principle Name	What it says? howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability
Open Closed Principle	Software components should be open for extension, but closed for modification
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use
Dependency Inversion Principle	Depend on abstractions, not on concretions

➢ Single Responsibility Principle (SRP) –

- This principle states that “a class should do one thing and therefore it should have only single reason to change”. It means that every class should have a single responsibility or single job or a single purpose.

- 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.



Benefits –

- The code quality of the application is much better.
- Testing and Writing test cases is much simpler.
- New members can be on boarded easily.

Example-

Books.java

```
package SingleResponsibility;

import java.util.*;
public class Books {
    private int id;
    private String name;
    private 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 Books(){
        setId(161);
        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());
    }
}
```

Book_Issued.java

```
package SingleResponsibility;

public class Book_Issued {
    String date= "25/10/2021";
    void issuedBook(String name,String n){

        System.out.println("The book "+name+" issued to "+n+" on 
"+date);
    }

    public static void main(String[] args) {
        Books B1 =new Books();
        User u1 = new User();
        Book_Issued i1 =new Book_Issued();
        i1.issuedBook(B1.getName(),u1.getName());
    }
}
```

User.java

```
package SingleResponsibility;

public class User {
    int userId;
    String name;
    String address;
    int contactNumber;
    public int getUserId()
    {return this.userId;}

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

    public String getAddress()
    {return this.address;}

    public int getContactNumber()
    {return this.contactNumber;}
    public void setUserId(int userId)
    {this.userId=userId;}
    public void setName(String name)
    {this.name=name;}

    public void setAddress(String address)
    {this.address=address;}

    public void setContactNumber(int contactNumber)
    {this.contactNumber=contactNumber;}

    User(){
        setUserId(1234);
        setName("Deeksha Tripathi");
        setAddress("Dayanand Vihar");
        setContactNumber(987654321);
    }
}
```

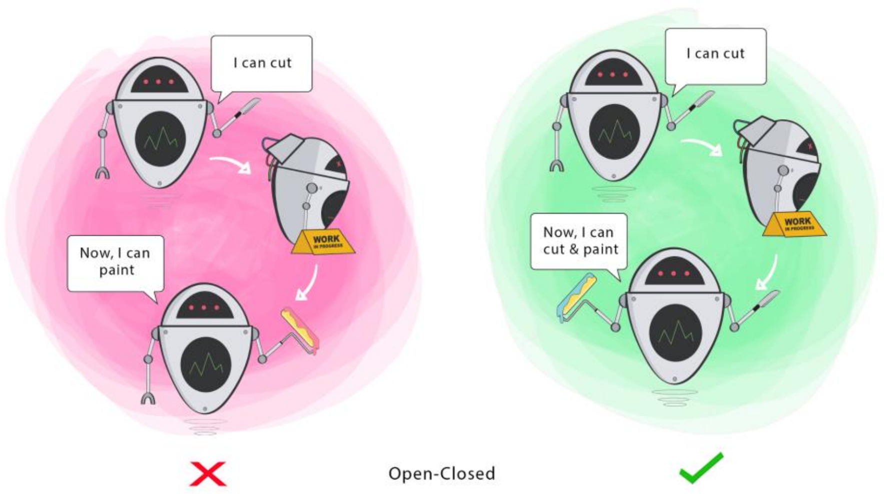
Goal-

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

➢Open/Closed Principle-

- This principle states that the “software entities like (classes, modules, functions and many more) should be open for extension, but closed for modification” which means that one should be able to extend a class behaviour, with modifying it.
- 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 run time.



Benefits-

- Helps to write flexible, scalable and reusable code.
- It also minimizes the changes happened in the code.

Example-

Book.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");
    }
}
```

Book_Issued.java

```
package OpenClose;

public class Book_Issued {
    String date= "25/10/2021";
    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);
    }
}
```

BooksOpenClosed.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("THE NOTEBOOK");
        setAuthor("NICHOLAS SPARKS");
        setCategory("ROMANTIC");

    }
}
```

User.java

```
package OpenClose;

public class User {
    int userId;
    String name;
    String address;
    int contactNumber;
    public int getUserId()
    {return this.userId;}

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

    public String getAddress()
    {return this.address;}

    public int getContactNumber()
    {return this.contactNumber;}
    public void setUserId(int userId)
    {this.userId=userId;}

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

    public void setAddress(String address)
    {this.address=address;}

    public void setContactNumber(int contactNumber)
    {this.contactNumber=contactNumber;}

    User(){
        setUserId(1619);
        setName("DEEKSHA TRIPATHI");
        setAddress("DAYANAND VIHAR ");
        setContactNumber(987654321);
    }
}
```

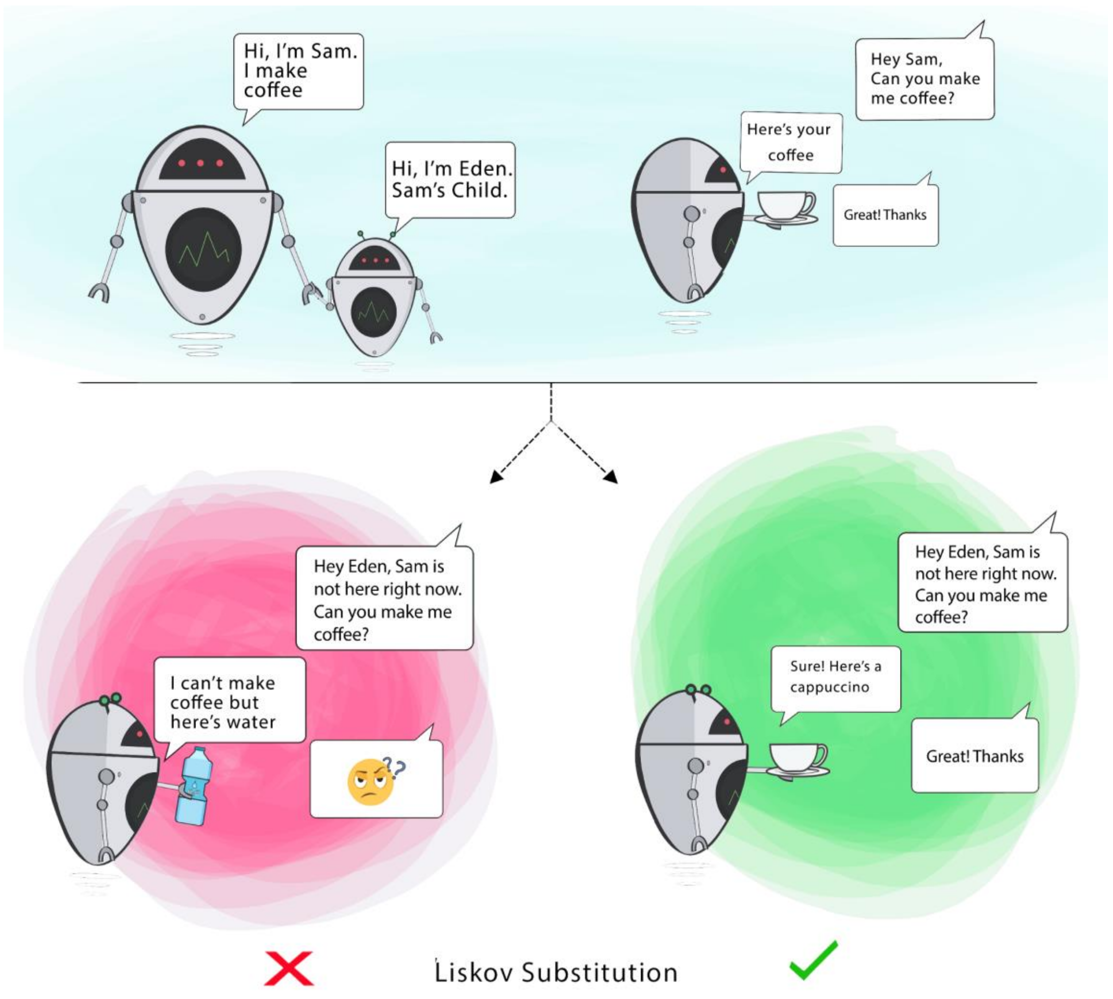
Goals-

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.

➢Liskov’s Substitution Principle (LSP)-

- This principle states that “Derived or child classes must be substitutable for their base or parent classes”. It was introduced by Barbara Liskov in 1987 and it also ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour.
- 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.


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

Benefits-

- It makes code re-usable.
- Reduced Coupling.
- It also helps us in easier maintenance.

Example-

Book.java

```
package LiskovSubstitution;

public interface Book {
}
```

LSP.java

```
package LiskovSubstitution;

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

    }
}
```

nonFictionalBook.java

```
package LiskovSubstitution;

public class nonFictionalBook implements Book{

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

NovelBook.java

```
package LiskovSubstitution;

public class NovelBook extends nonFictionalBook{
    void func1() {
        NovelBook n = new NovelBook();
        System.out.println(" The Notebook is a novel 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 five principles and highlighted their benefits and goals. They are to help you make your code easy to adjust, extend and test with little to no problems.