S.O.L.I.D Principles

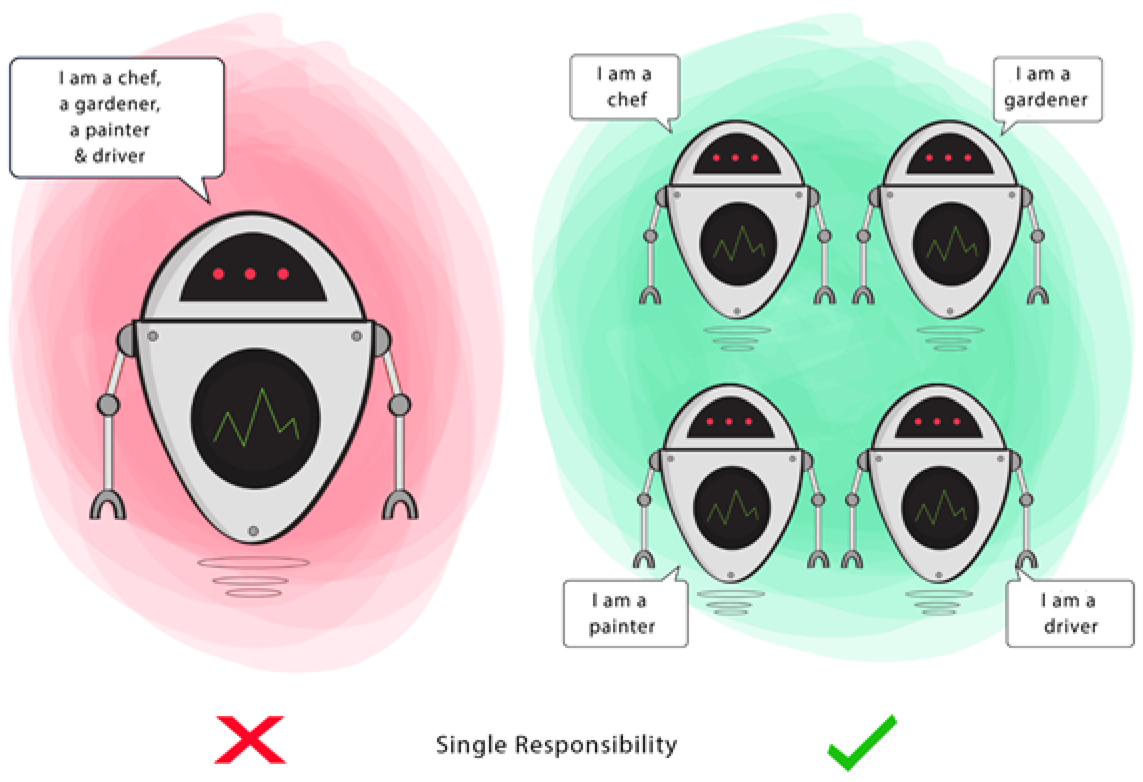
- SOLID principles are object-oriented design concepts relevant to software development.
- SOLID principles are an object-oriented approach that are applied to software structure design.
- The SOLID principle was introduced by Robert C. Martin, also known as Uncle Bob and it is a coding standard in programming.
- SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. When applied properly it makes your code more extendable, logical, and easier to read.
- This principle is an acronym of the five principles which is given below:
  - ❖ Single Responsibility Principle (SRP)
  - ❖ Open-Closed Principle (OCP)
  - ❖ Liskov Substitution Principle (LSP)
  - ❖ Interface Segregation Principle (ISP)
  - ❖ Dependency Inversion Principle (DIP)
- The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure.
- These five principles help us understand the need for certain design patterns and software architecture in general.

First, let us understand through a picture:

What are SOLID Principles?

S — Single Responsibility

A class should have a single responsibility

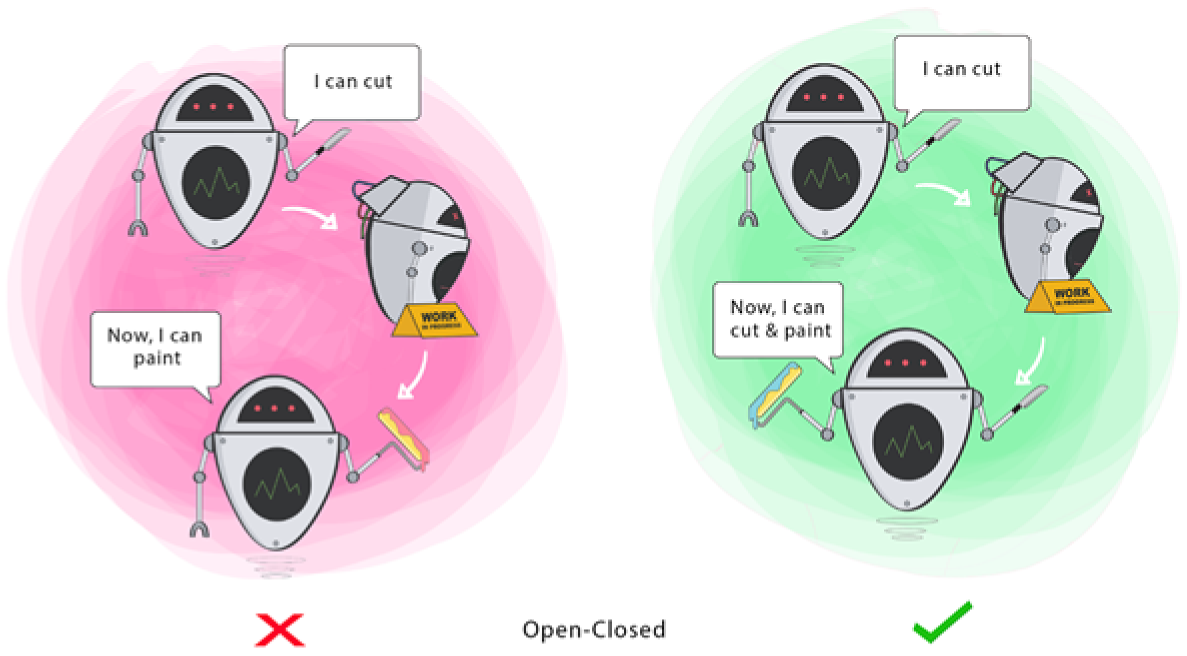


If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities could affect the other ones without you knowing.

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.

O — Open-Closed

Classes should be open for extension, but closed for modification



Open-Closed

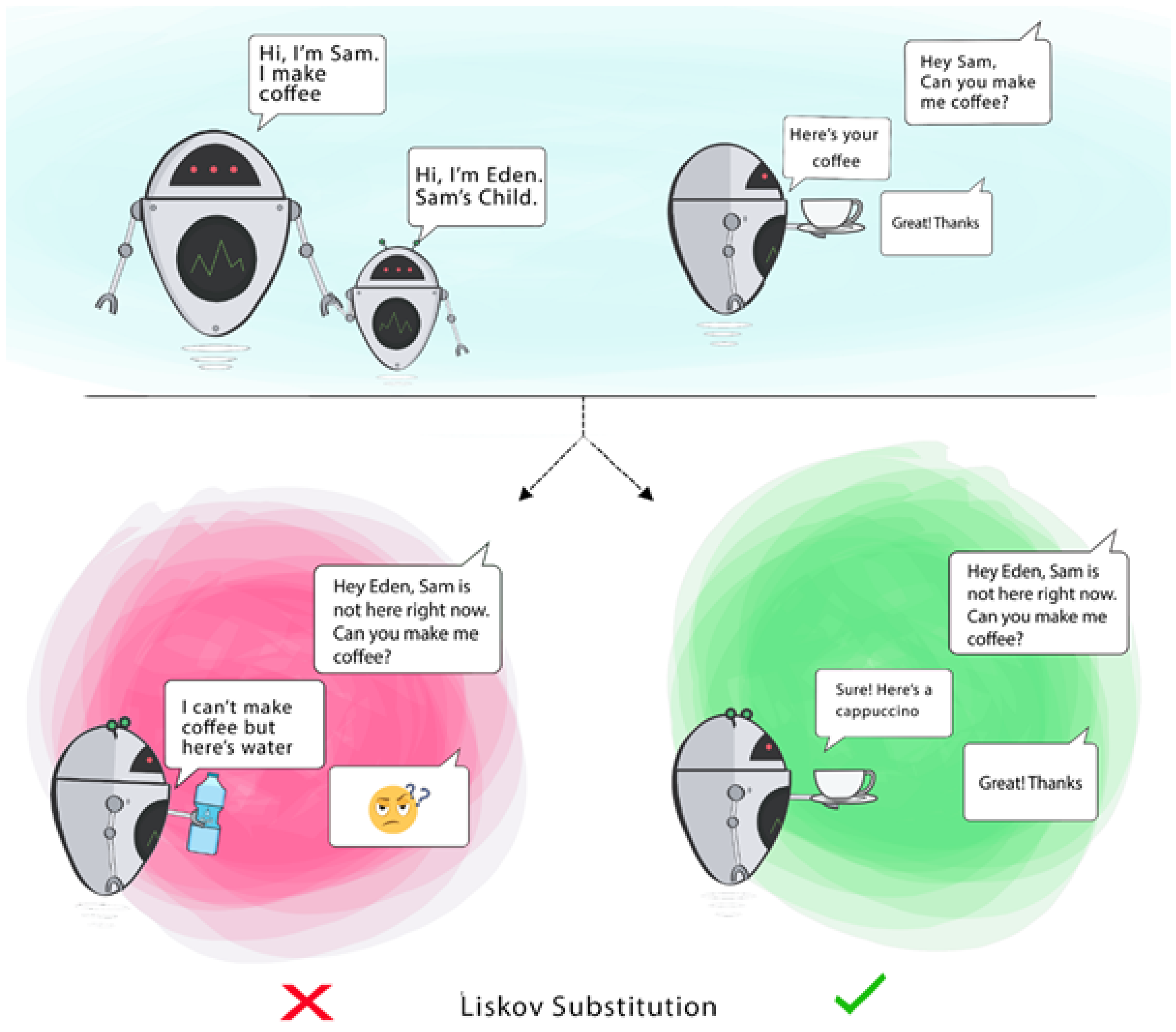
Changing the current behaviour of a Class will affect all the systems using that Class.

If you want the Class to perform more functions, the ideal approach is to add to the functions that already exist NOT change them.

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.

L — Liskov Substitution

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.


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

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.

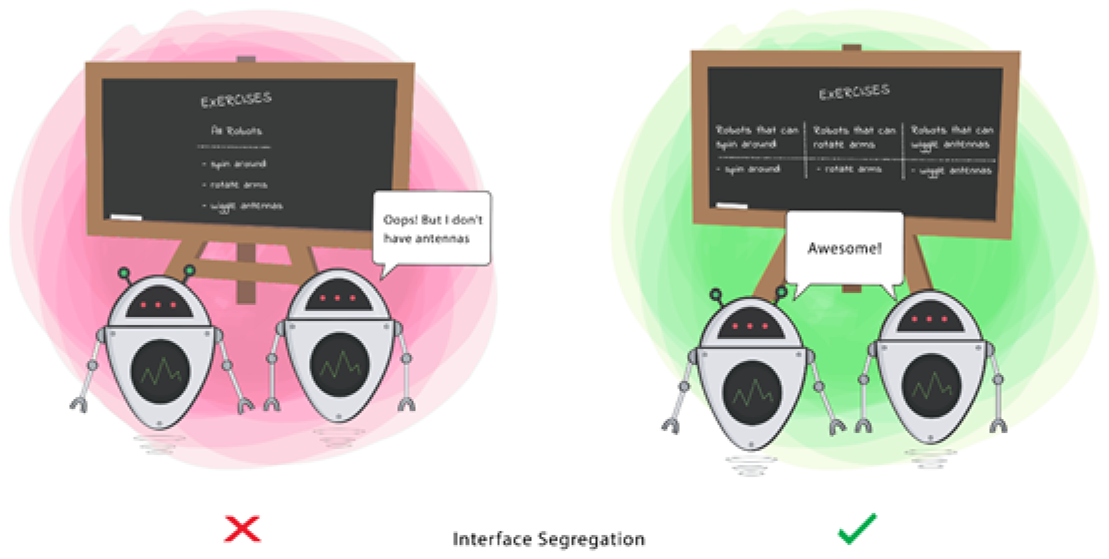
The picture shows that the **parent** Class delivers Coffee (it could be any type of coffee). It is acceptable for the **child** Class to deliver Cappuccino because it is a specific type of Coffee, but it is NOT acceptable to deliver Water.

If the **child** Class doesn’t meet these requirements, it means the **child** Class is changed completely and violates this principle.

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.

I — Interface Segregation

Clients should not be forced to depend on methods that they do not use.



When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

A Class should perform only actions that are needed to fulfil its role. Any other action should be removed completely or moved somewhere else if it might be used by another Class in the future.

Goal: This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires.

D — Dependency Inversion

- High-level modules should not depend on low-level modules. Both should depend on the abstraction.
- Abstractions should not depend on details. Details should depend on abstractions.

Here we will take a deep dive into the three principles and will understand their implementation with the help of code examples.

S.R.P (Single Responsibility Principle):

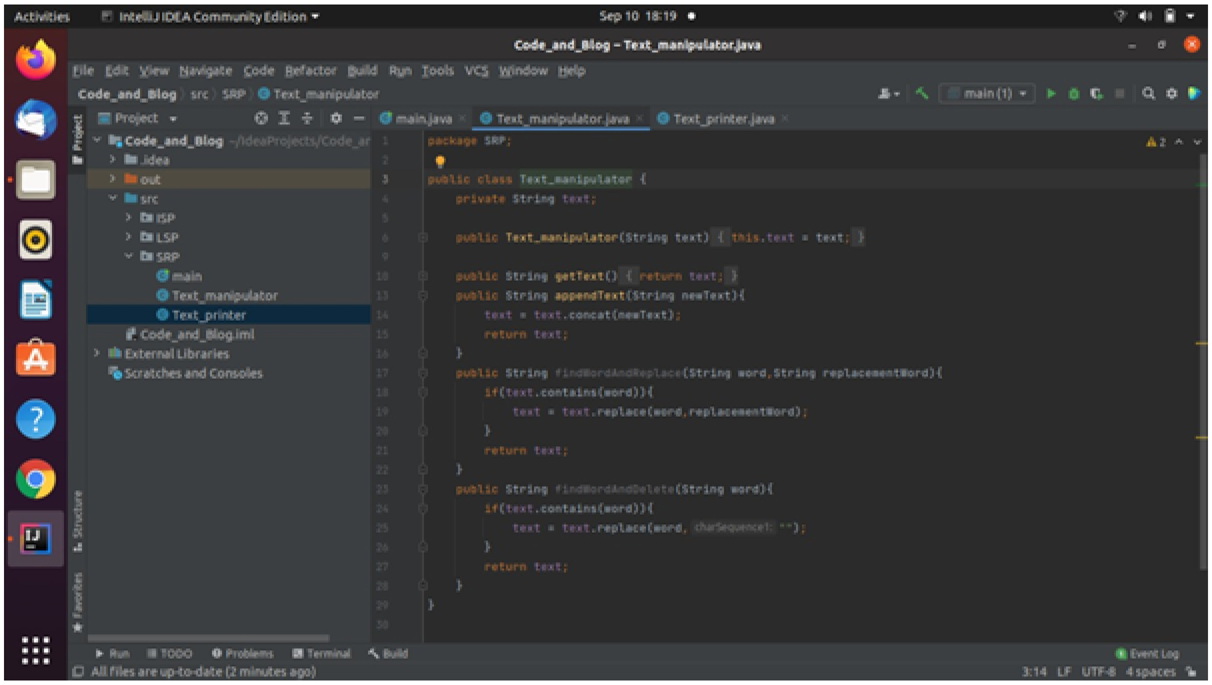
- SRP principle states that “a class should have only one reason to change” which means every class should have a single responsibility or single job or single purpose.

- In other words, the single responsibility principle states that every Java class must perform a single functionality. Implementation of multiple functionalities in a single class mashup the code and if any modification is required may affect the whole class. It is precise and the code can be easily maintained.

- Also, The Single Responsibility Principle (SRP) states that there should never be more than one reason for a class to change. This means that every class, or similar structure, in your code should have only one job to do. Everything in the class should be related to that single purpose. It does **not** mean that your classes should only contain one method or property.

- There can be a lot of members as long as they relate to the single responsibility. It may be that when the one reason to change occurs, multiple members of the class may need modification. It may also be that multiple classes will require updates.

Let us understand through an example:

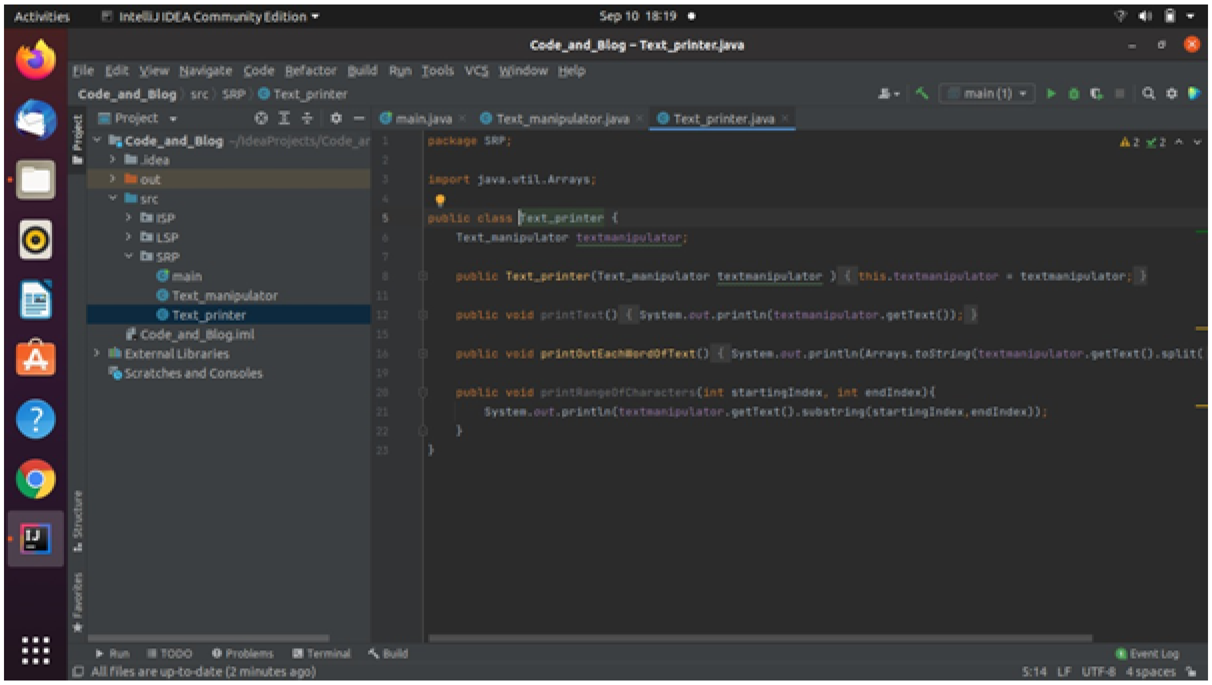


```java
package SRP;

public class Text_manipulator {
    private String text;

    public Text_manipulator(String text) { this.text = text; }

    public String getText() { return text; }
    public String appendText(String newText){
        text = text.concat(newText);
        return text;
    }
    public String findWordAndReplace(String word,String replacementWord){
        if(text.contains(word)){
            text = text.replace(word,replacementWord);
        }
        return text;
    }
    public String findWordAndDelete(String word){
        if(text.contains(word)){
            text = text.replace(word, charSequence1: "");
        }
        return text;
    }
}
```

In the above example we have a Text_Manipulator class that takes text as input and performs some manipulations over the text, but in the Text_Manipulator class we have a method named printText that prints the text on the console.

Here the Text_Manipulator class has all the methods related to a single responsibility that is manipulating the text in some manner except the printText method as it is printing the text to the console and is having a different responsibility of printing.So, when we will be required to change in the functionality of the printing we will have to modify the Text_manipulator class and this is violating the single-responsibility principle as a **class should have only one reason to change.**

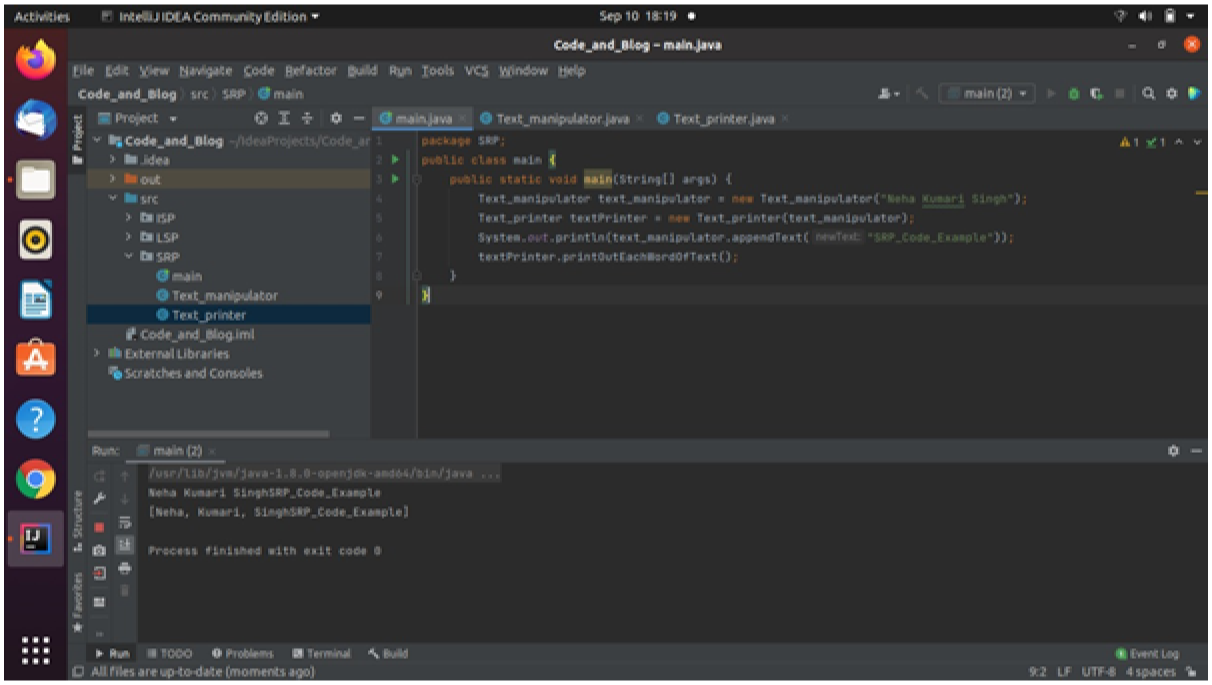
To achieve the SRP in the above example we will have to separate the printing responsibility by making a separate class for printing related tasks.



```java
package SRP;

import java.util.Arrays;

public class Text_printer {
    Text_manipulator textmanipulator;

    public Text_printer(Text_manipulator textmanipulator ) { this.textmanipulator = textmanipulator; }

    public void printText() { System.out.println(textmanipulator.getText()); }

    public void printOutEachWordOfText() { System.out.println(Arrays.toString(textmanipulator.getText().split(

    public void printRangeOfCharacters(int startingIndex, int endIndex){
        System.out.println(textmanipulator.getText().substring(startingIndex,endIndex));
    }
}
```

And the TextPrinter class is handling the responsibilities related to printing of the text. Now both the classes have only one single responsibility and thus helping to achieve SRP.

Now to check whether the above example is following LSP principle or not we require a main class. So let's see our main class ---



```java
package SRP;
public class main {
    public static void main(String[] args) {
        Text_manipulator text_manipulator = new Text_manipulator("Neha Kumari Singh");
        Text_printer textPrinter = new Text_printer(text_manipulator);
        System.out.println(text_manipulator.appendText("SRP_Code_Example"));
        textPrinter.printOutEachWordOfText();
    }
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Neha Kumari SinghSRP_Code_Example
[Neha, Kumari, SinghSRP_Code_Example]

Process finished with exit code 0
```

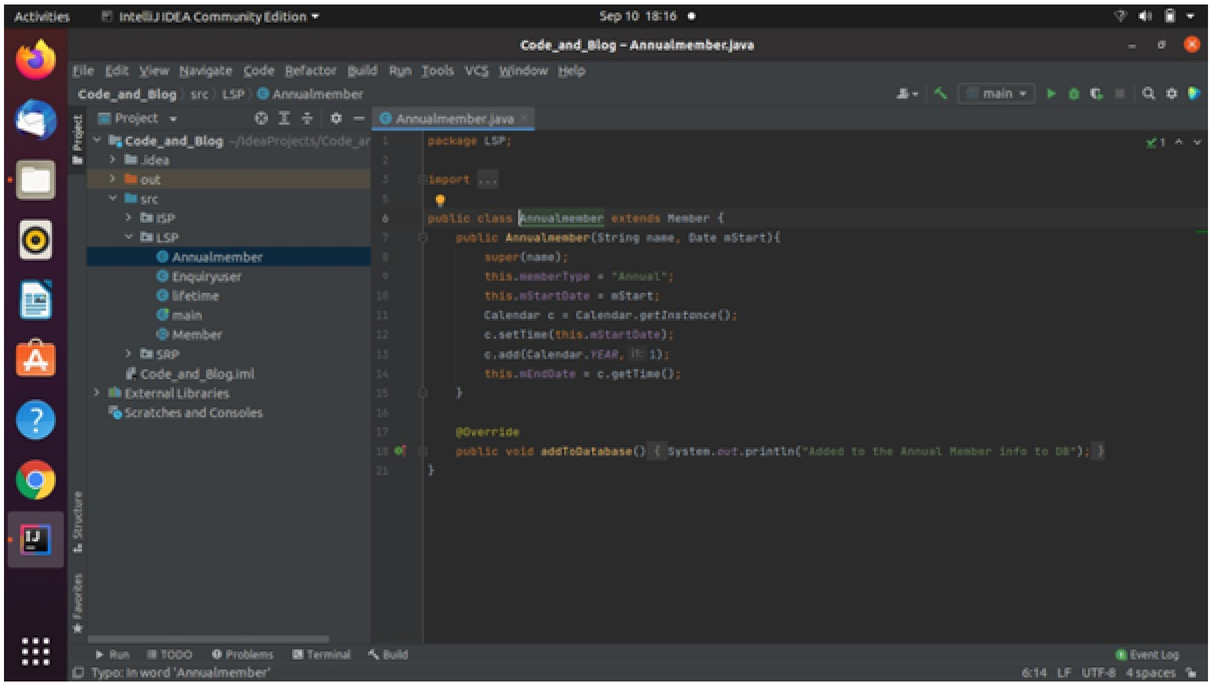
ISP (Interface Segregation Principle):

- Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces. The ISP principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need.

- ISP is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “do not force any client to implement an interface which is irrelevant to them “. Here our main goal is to focus on avoiding fat interfaces and give preference to many small client-specific interfaces. We should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

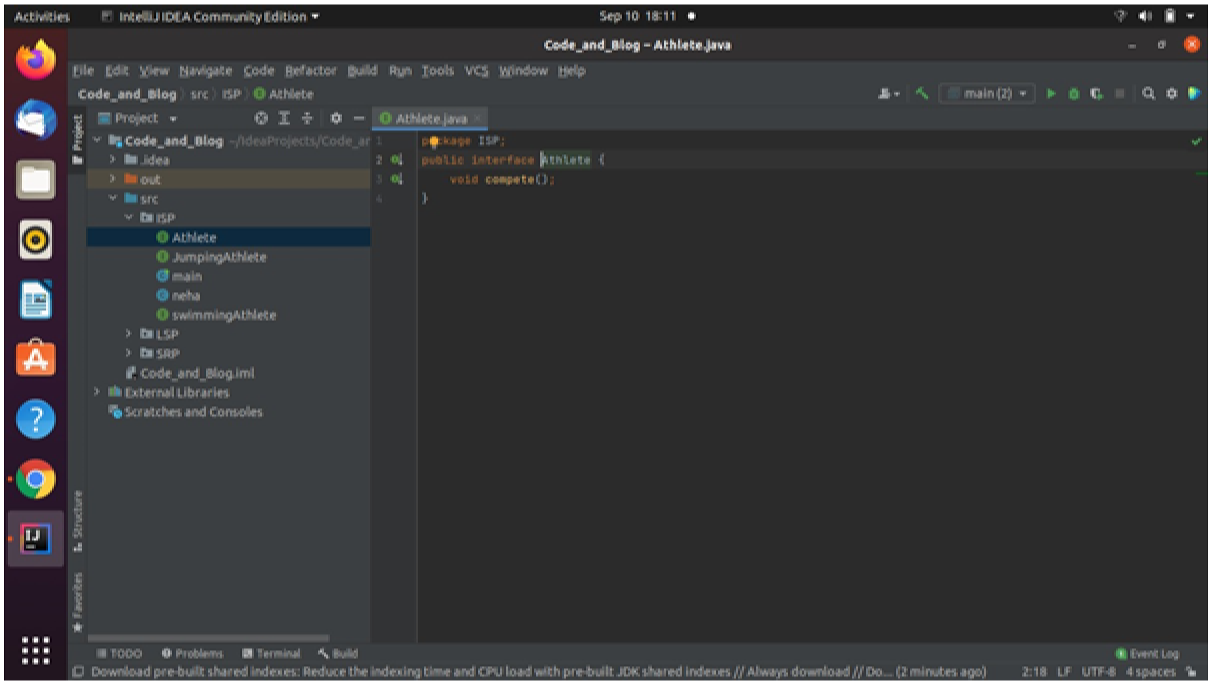
- Also, the principle states that the larger interfaces split into smaller ones. Because the implementation classes use only the methods that are required. We should not force the client to use the methods that they do not want to use. The goal of the interface segregation principle is similar to the single responsibility principle.

Let us understand through an example:



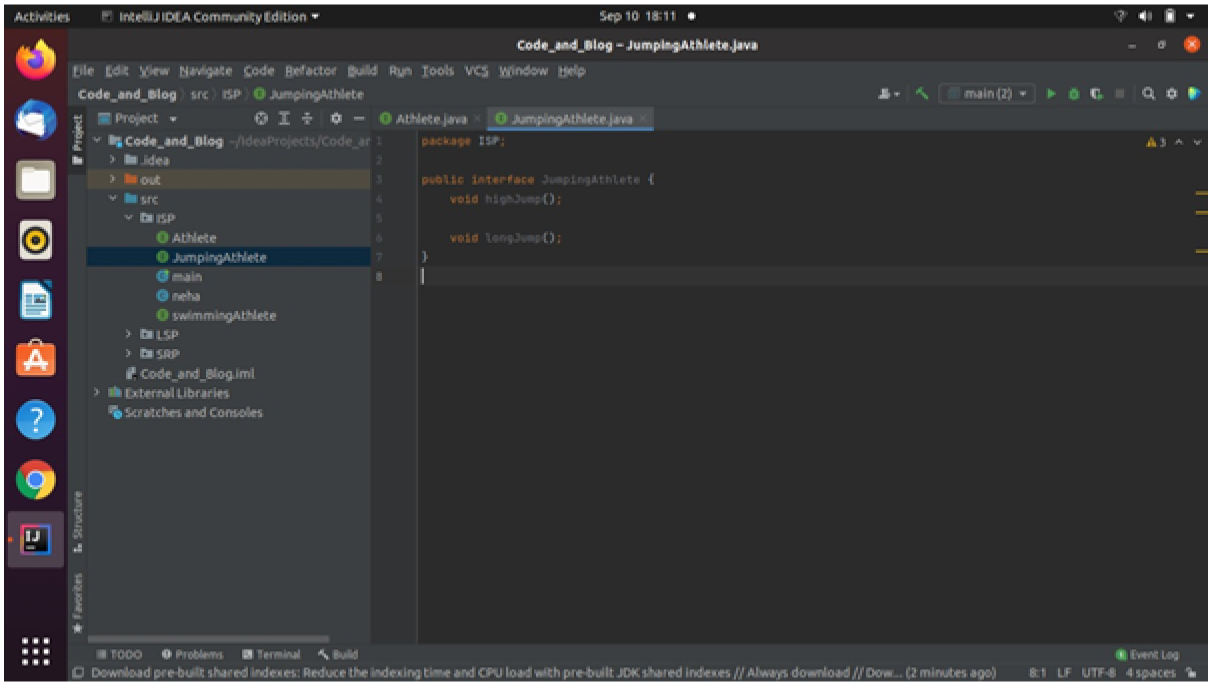
Activities
IntelliJ IDEA Community Edition
Sep 10 18:16
Code_and_Blog – Annualmember.java
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Code_and_Blog src LSP Annualmember
main
Project
Annualmember.java
Code_and_Blog ~/IdeaProjects/Code_ar
.idea
out
src
ISP
LSP
Annualmember
Enquiryuser
lifetime
main
Member
SRP
Code_and_Blog.iml
External Libraries
Scratches and Consoles
Run
TODO
Problems
Terminal
Build
Event Log
Typo: In word 'Annualmember'
6:14 LF UTF-8 4 spaces

```java
package LSP;

import ...

public class Annualmember extends Member {
    public Annualmember(String name, Date mStart){
        super(name);
        this.memberType = "Annual";
        this.mStartDate = mStart;
        Calendar c = Calendar.getInstance();
        c.setTime(this.mStartDate);
        c.add(Calendar.YEAR, i: 1);
        this.mEndDate = c.getTime();
    }

    @Override
    public void addToDatabase() { System.out.println("Added to the Annual Member info to DB"); }
}
```


Code_and_Blog – Athlete.java
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Code_and_Blog src ISP Athlete
Athlete.java
Code_and_Blog ~/IdeaProjects/Code_ar
.idea
out
src
ISP
Athlete
JumpingAthlete
main
neha
swimmingAthlete
LSP
SRP
Code_and_Blog.iml
External Libraries
Scratches and Consoles

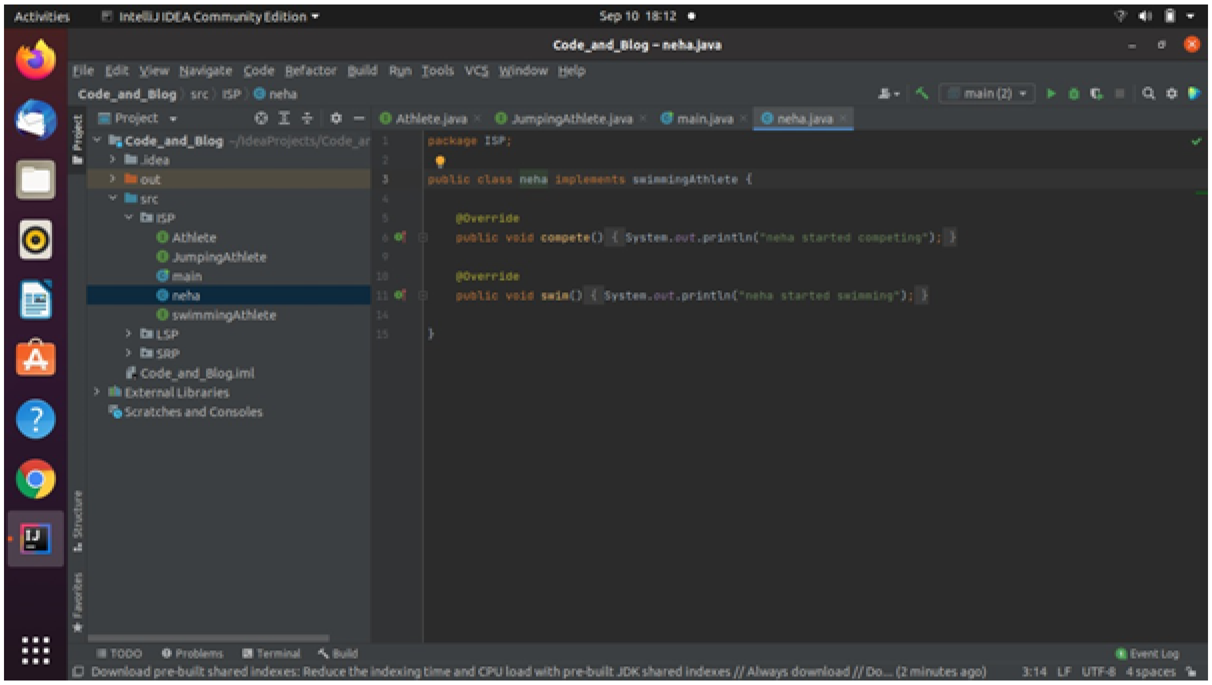
```java
package ISP;
public interface Athlete {
    void compete();
}
```

Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built JDK shared indexes // Always download // Do... (2 minutes ago)

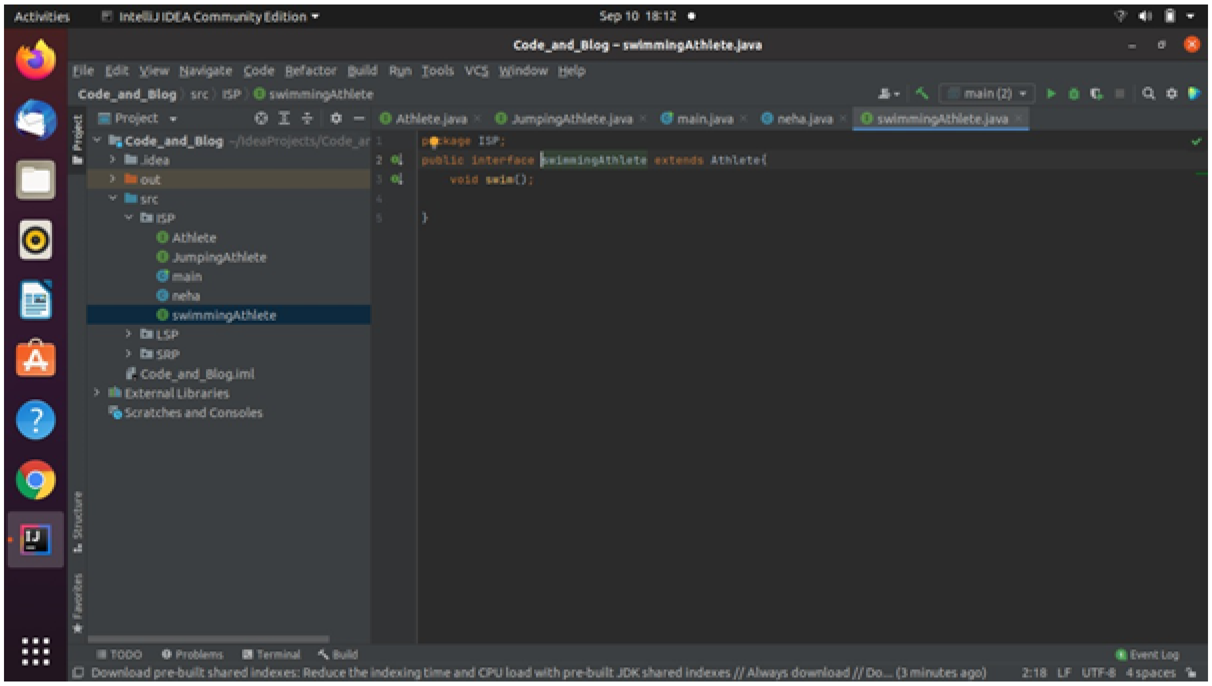

Code_and_Blog – JumpingAthlete.java
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Code_and_Blog src ISP JumpingAthlete
Athlete.java
JumpingAthlete.java
Code_and_Blog ~/IdeaProjects/Code_ar
.idea
out
src
ISP
Athlete
JumpingAthlete
main
neha
swimmingAthlete
LSP
SRP
Code_and_Blog.iml
External Libraries
Scratches and Consoles

```java
package ISP;

public interface JumpingAthlete {
    void highJump();

    void longJump();
}

```

Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built JDK shared indexes // Always download // Dow... (2 minutes ago)

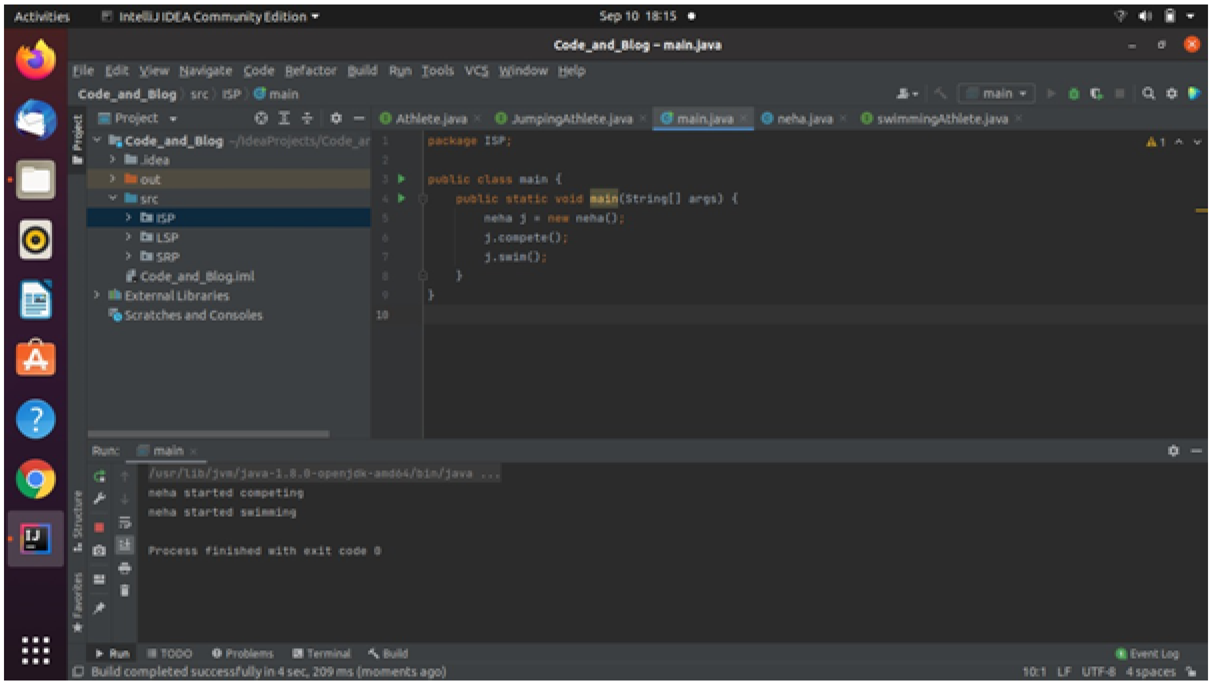


```java
package ISP;

public class neha implements swimmingAthlete {

    @Override
    public void compete() { System.out.println("neha started competing"); }

    @Override
    public void swim() { System.out.println("neha started swimming"); }

}
```



```java
package ISP;
public interface swimmingAthlete extends Athlete{
    void swim();

}
```

Now to check whether the above example is following ISP principle or not we require a main class. So let’s see our main class ---



```java
package ISP;

public class main {
    public static void main(String[] args) {
        neha j = new neha();
        j.compete();
        j.swim();
    }
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
neha started competing
neha started swimming

Process finished with exit code 0
```

LSP (Liskov Substitution Principle):

- The LSP principle was introduced by Barbara Liskov in 1987 and according to this principle “Derived or child classes must be substitutable for their base or parent classes “. Liskov's principle is easy to understand but hard to detect in code.

- The LSP states that subclasses should be substitutable for their base classes. This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

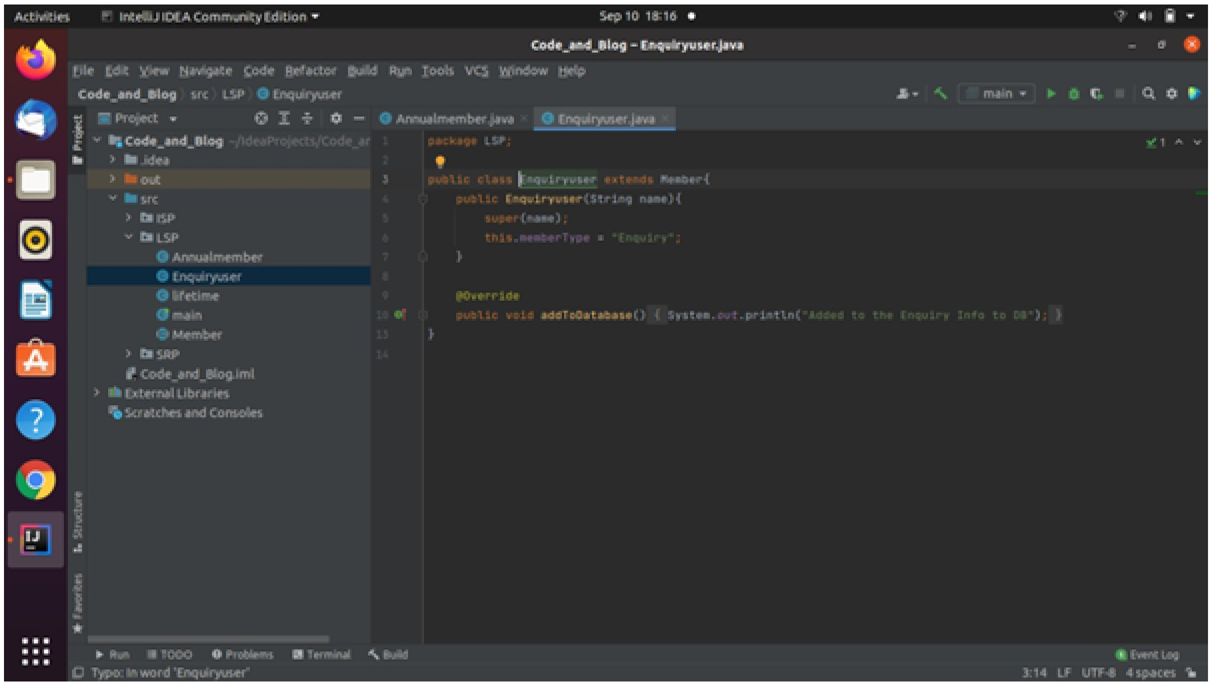
- This is the expected behaviour, because when we use inheritance, we assume that the child class inherits everything that the superclass has. The child class extends the behaviour but never narrows it down. Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect.

- Also, it applies to inheritance in such a way that the derived classes must be completely substitutable for their base classes. In other words, if class A is a subtype of class B, then we should be able to replace B with A without interrupting the behaviour of the program.

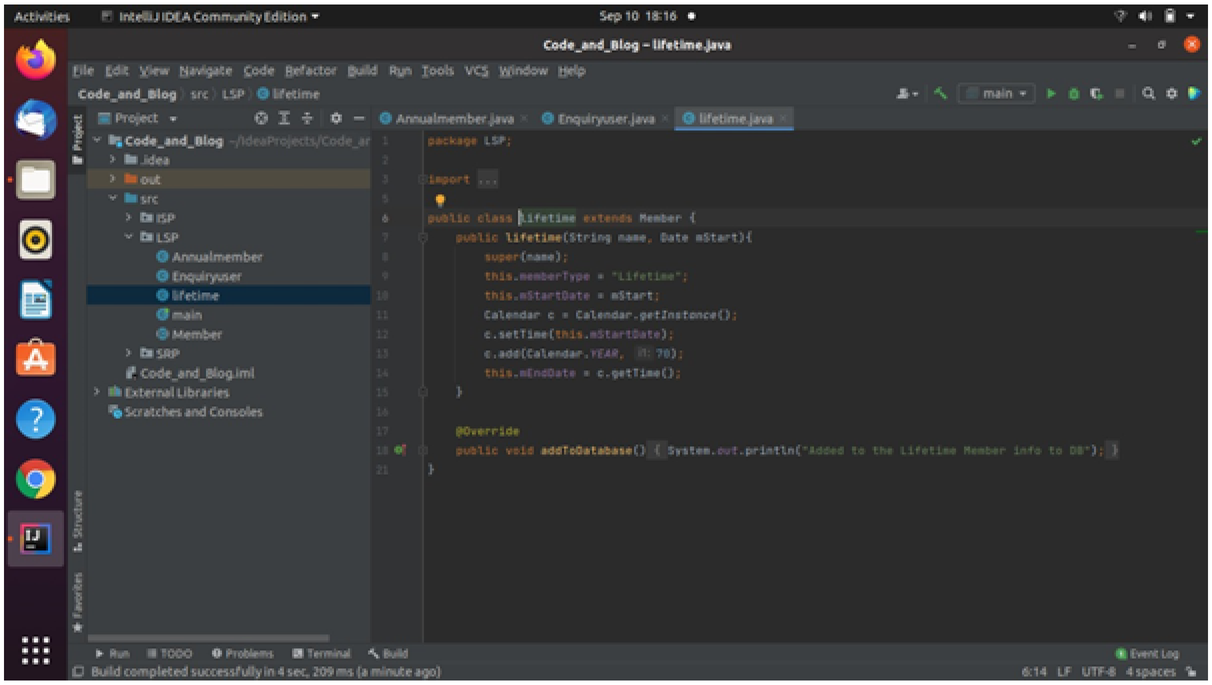
- It extends the open-close principle and also focuses on the behaviour of a superclass and its subtypes. We should design the classes to preserve the property unless we have a strong reason to do otherwise.

Let us understand through an example:

Below in the code we have an abstract class Member that takes multiple string input with an abstract method **“addToDatabase”.**


Activities
IntelliJ IDEA Community Edition
Sep 10 18:16
Code_and_Blog – Enquiryuser.java
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Code_and_Blog src LSP Enquiryuser
Project
Code_and_Blog ~/IdeaProjects/Code_ar
.idea
out
src
ISP
LSP
Annualmember
Enquiryuser
lifetime
main
Member
SRP
Code_and_Blog.iml
External Libraries
Scratches and Consoles
Annualmember.java
Enquiryuser.java
main
Run TODO Problems Terminal Build
Event Log
Typo: In word 'Enquiryuser'
3:14 LF UTF-8 4 spaces

```java
package LSP;

public class Enquiryuser extends Member{
    public Enquiryuser(String name){
        super(name);
        this.memberType = "Enquiry";
    }

    @Override
    public void addToDatabase() { System.out.println("Added to the Enquiry Info to DB"); }
}
```

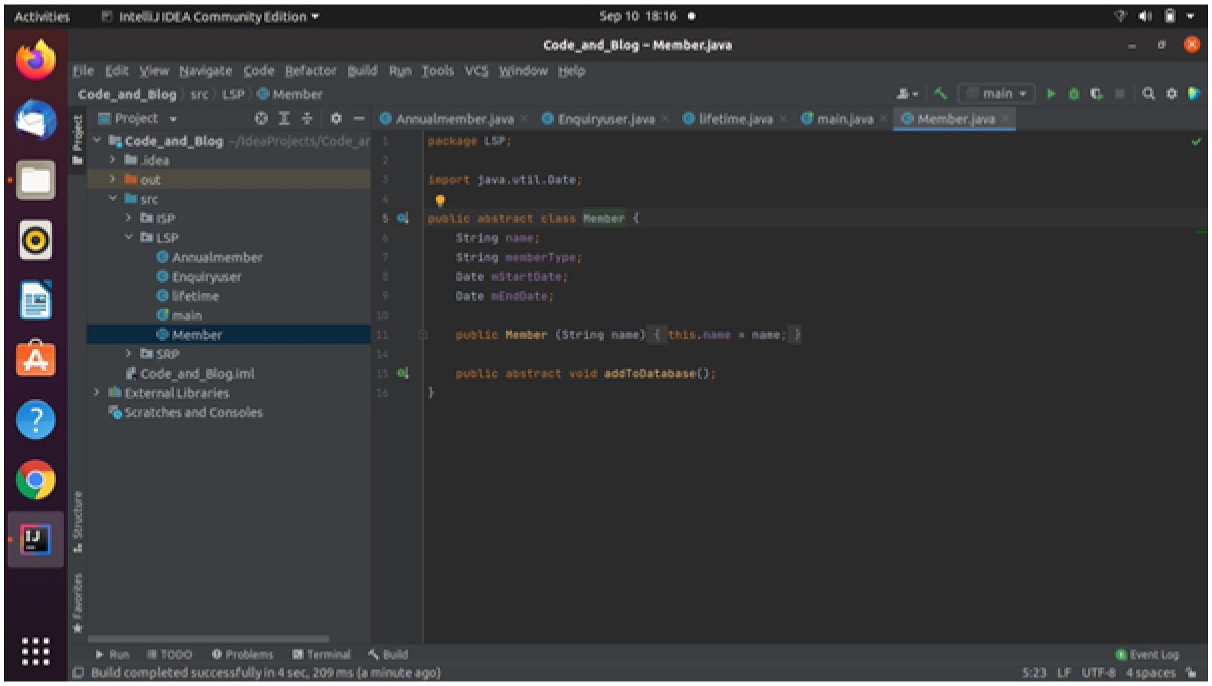


```java
package LSP;

import ...

public class Lifetime extends Member {
    public lifetime(String name, Date mStart){
        super(name);
        this.memberType = "Lifetime";
        this.mStartDate = mStart;
        Calendar c = Calendar.getInstance();
        c.setTime(this.mStartDate);
        c.add(Calendar.YEAR,  n: 70);
        this.mEndDate = c.getTime();
    }

    @Override
    public void addToDatabase() { System.out.println("Added to the Lifetime Member info to DB"); }
}
```

```
Build completed successfully in 4 sec, 209 ms (a minute ago)
```

LifetimeMember Class is a subtype of class Member that is overriding the parent class addToDatabse method .

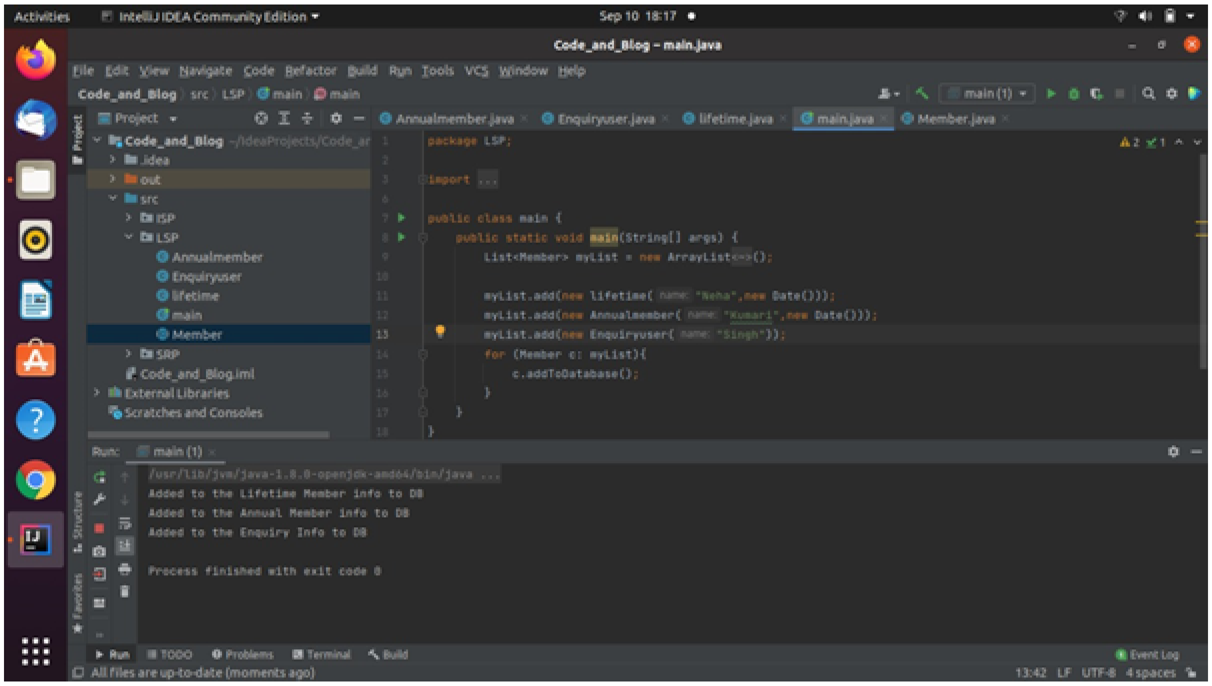
Two more classes of AnnualMember and Enquiry user are there that extend the Member Class.



```java
package LSP;

import java.util.Date;

public abstract class Member {
    String name;
    String memberType;
    Date mStartDate;
    Date mEndDate;

    public Member (String name) { this.name = name; }

    public abstract void addToDatabase();
}
```

```
Build completed successfully in 4 sec, 209 ms (a minute ago)
```

Now to check whether the above example is following LSP principle or not we require a main class. So let's see our main class ---

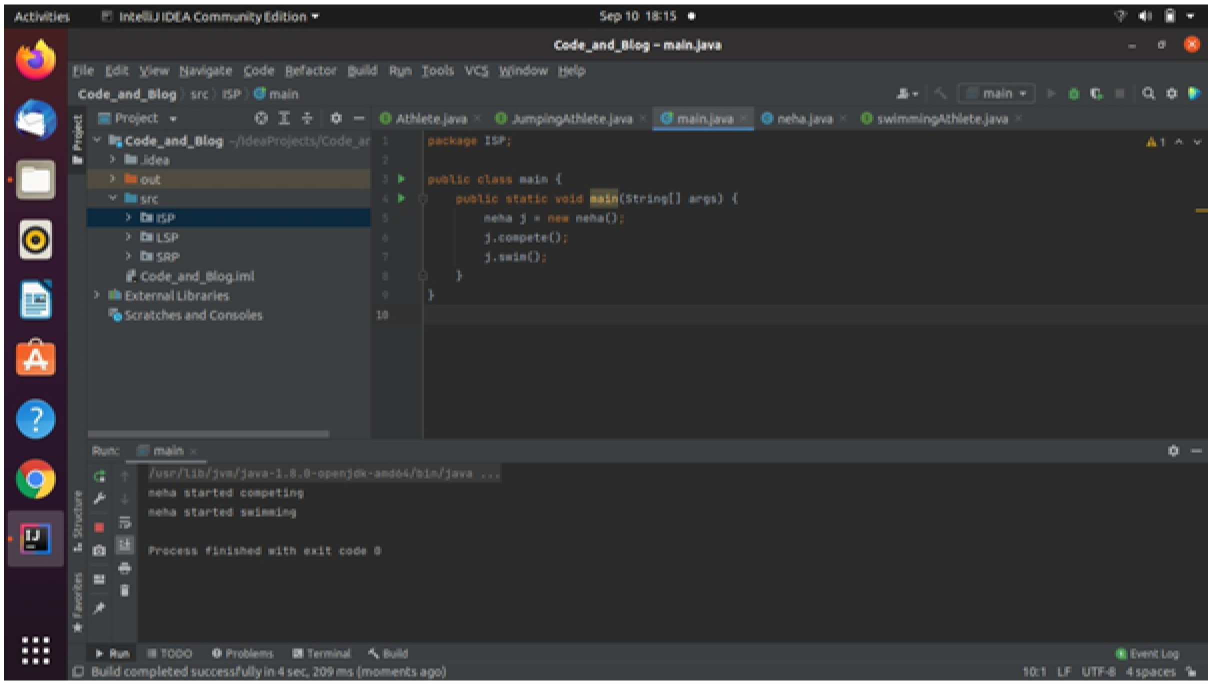


```java
package LSP;

import ...

public class main {
    public static void main(String[] args) {
        List<Member> myList = new ArrayList<>();

        myList.add(new lifetime( name: "Neha",new Date()));
        myList.add(new Annualmember( name: "Kumari",new Date()));
        myList.add(new Enquiryuser( name: "Singh"));
        for (Member c: myList){
            c.addToDatabase();
        }
    }
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
Added to the Lifetime Member info to DB
Added to the Annual Member info to DB
Added to the Enquiry Info to DB

Process finished with exit code 0
```

Here in the main class we have made a list of parent type i.e. "**Member"** class but at the run time we are assigning it with the instance of children classes.
In the loop we are running the "**addToDatabase "** method over each member of the list to check for the LSP and the code worked flowlessly, hence satisfying the **Liskov Substitution Principle.**

Now if we add a **"addBooking"** method in the "**Member"** class then the subclass "**EnquiryUser"** would not be able to substitute this method completely as the Enquiry User does not make booking and overriding that method will have different behavior of that method in the subclass and this will violate the **Liskov Substitution Principle** that says a "**child class should completely substitute its parent class**".


Activities
IntelliJ IDEA Community Edition
Sep 10 18:15
Code_and_Blog – main.java
File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Code_and_Blog 〉 src 〉 ISP 〉 main
Project
Code_and_Blog ~/IdeaProjects/Code_ar
.idea
out
src
ISP
LSP
SRP
Code_and_Blog.iml
External Libraries
Scratches and Consoles
Athlete.java
JumpingAthlete.java
main.java
neha.java
swimmingAthlete.java
Run: main
Run
TODO
Problems
Terminal
Build
Event Log
Build completed successfully in 4 sec, 209 ms (moments ago)
10:1 LF UTF-8 4 spaces

```java
package ISP;

public class main {
    public static void main(String[] args) {
        neha j = new neha();
        j.compete();
        j.swim();
    }
}
```

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
neha started competing
neha started swimming

Process finished with exit code 0
```