SOLID

- S — Single responsibility principle.
- O — Open closed principle.
- L — Liskov substitution principle.
- I — Interface segregation principle.
- D — Dependency Inversion principle.

The SOLID principles are a set of design principles for writing clean, maintainable, and flexible object-oriented code. The acronym SOLID stands for:

Single Responsibility Principle (SRP):

Let's begin with the single responsibility principle. As we might expect, this principle states that a class should only have one responsibility. Furthermore, it should only have one reason to change.

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.
For example, let's look at a class to represent a simple book:

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters
}
```

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

Let's now add a couple of methods to query the text:

```
public class BookService {


    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String replacementWord){
        return text.replaceAll(word, replacementWord);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

Now our Book class works well, and we can store as many books as we like in our application.

But what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

```
public class BadBook {
    //...

    void printTextToConsole(){
        // our code for formatting and printing the text
    }
}
```

However, this code violates the single responsibility principle we outlined earlier.

To fix our mess, we should implement a separate class that deals only with printing our texts:

```
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..
    }
}
```

Awesome. Not only have we developed a class that relieves the Book of its printing duties, but we can also leverage our BookPrinter class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

Open/Closed Principle (OCP):

It's now time for the O in SOLID, known as the open-closed principle. Simply put, classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

Of course, the one exception to the rule is when fixing bugs in existing code.

Let's explore the concept with a quick code example. As part of a new project, imagine we've implemented a Guitar class.

It's fully fledged and even has a volume knob:

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

We launch the application, and everyone loves it. But after a few months, we decide the Guitar is a little boring and could use a cool flame pattern to make it look more rock and roll.

At this point, it might be tempting to just open up the Guitar class and add a flame pattern — but who knows what errors that might throw up in our application.

Instead, 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 Principle (LSP):

Next on our list is Liskov substitution, which is arguably the most complex of the five principles. Simply put, if class A is a subtype of class B, we should be able to replace B with A without disrupting the behavior of our program.

Let's jump straight to the code to help us understand this concept:

```
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:

```
public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

As our code describes, we have an engine that we can turn on, and we can increase the power.

But wait — we are now living in the era of electric cars:

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our Car.

```
class Parent { }

class Child extends Parent { }

class Student{}
public class LiskovTest {
    public static void main(String[] args) {
        LiskovTest test = new LiskovTest();
        Parent parent = new Parent();
        test.invite(parent);//ok

        Child child1= new Child();
        test.invite(child1);//yes

        //Student s = new Student();
        ///test.invite(s);//?

    }

    public void invite(Parent parent){

    }
}
```

Interface Segregation Principle (ISP):

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.

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable{}
```

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

As avid zookeepers, we're more than happy to wash and feed our beloved bears. But we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice but to implement the code to pet the bear.

Let's fix this by splitting our large interface into three separate ones:

```
public interface BearCleaner {
    void washTheBear();
}
```

```
public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

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

```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```

And finally, we can leave the dangerous stuff to the reckless people:

```
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

Going further, we could even split our BookPrinter class from our example earlier to use interface segregation in the same way. By implementing a Printer interface with a single print method, we could instantiate separate ConsoleBookPrinter and OtherMediaBookPrinter classes.

Dependency Inversion Principle (DIP):

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.

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

```
class Monitor{}
class HpKeyBoard extends  Keyboard{ }
class ColorMonitor extends Monitor{}
class LenovoKeyBoard extends  Keyboard{}
class BoatKeyBoard extends  Keyboard{}
class Keyboard{}


//lib
public class Window98OSTest {
    private Monitor monitor;
    private Keyboard keyboard;
}
```

```
public class Windows98Machine {}
```

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every Windows98Computer we instantiate comes prepacked with a Monitor and a StandardKeyboard:

```
public class Windows98Machine {
```

```
    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

This code will work, and we'll be able to use the StandardKeyboard and Monitor freely within our Windows98Computer class.

Problem solved? Not quite. By declaring the StandardKeyboard and Monitor with the new keyword, we've tightly coupled these three classes together.

Not only does this make our Windows98Computer hard to test, but we've also lost the ability to switch out our StandardKeyboard class with a different one should the need arise. And we're stuck with our Monitor class too.

Let's decouple our machine from the StandardKeyboard by adding a more general Keyboard interface and using this in our class:

```
public interface Keyboard { }

public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

Here, we're using the dependency injection pattern to facilitate adding the Keyboard dependency into the Windows98Machine class.

Let's also modify our StandardKeyboard class to implement the Keyboard interface so that it's suitable for injecting into the Windows98Machine class:

```
public class StandardKeyboard implements Keyboard { }
```

Now our classes are decoupled and communicate through the Keyboard abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the Monitor class.

Excellent! We've decoupled the dependencies and are free to test our Windows98Machine with whichever testing framework we choose.
By following the SOLID principles, developers can create software that is more modular, flexible, and easier to maintain. These principles contribute to code that is easier to understand, refactor, extend, and test, resulting in higher quality and more maintainable software systems.