SOLID Principles

Don't dare to violate them!!

Single Responsibility Principle

A Class should have

one and only one

reason

to change

Single Responsibility Principle - Example 1

```
public class Task
{
    public void downloadFile(String location)

    public void parseTheFile(File file)

    public void persistTheData(Data data)
}
```

Single Responsibility Principle - Example 2

```
public class Employee {
  public Money calculatePay() ...
  public String reportHours() ...
  public void save() ...
}
```

Single Responsibility Principle - for methods

- Method should do related things
- A method should be at a high level or a low level

Open/Closed Principle (OCP)

Software entities
should be
open for extension
but
closed for modification

Open/Closed Principle (OCP) - Example 1

```
class AreaCalculator

    public double Area(Rectangle[] shapes)
        double area = 0;
        foreach (var shape in shapes)
            area += shape.Width*shape.Height;
        return area;
```

Open/Closed Principle (OCP) - Example 1 - Add Circle

```
public double Area(object[] shapes)

    double area = 0;

    foreach (var shape in shapes)
        if (shape is Rectangle) {
                Rectangle rectangle = (Rectangle) shape;
                area += rectangle.Width*rectangle.Height;
        } else {
                Circle circle = (Circle)shape;
                area += circle.Radius * circle.Radius * Math.PI;
        }
    return area;
```

Open/Closed Principle (OCP) - Example 1 - Solution

```
public abstract class Shape
{
    public abstract double Area();
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}
```

Open/Closed Principle (OCP) - Example 1 - Solution

```
public class Circle : Shape
{
	public double Radius { get; set; }
	public override double Area()
	{
		return Radius*Radius*Math.PI;
	}
}
public double Area(Shape[] shapes)
{
	double area = 0;
	foreach (var shape in shapes)
	{
		area += shape.Area();
	}
	return area;
}
```

Liskov substitution principle (LSP)

Subtypes must be
substitutable
for their base types.

Liskov substitution principle (LSP)

A subclass may override a parent method only under certain conditions

- Preconditions can only be weaker.
- Postconditions can only be stronger.

Liskov substitution principle (LSP) - Example 1

```
class Rectangle
{
    void setWidth(double w)
    void setHeight(double h)
    double getHeight()
    double getWidth()
}
class Square
{
    void setWidth(double w)  //Set both height and width to w
    void setHeight(double h) //Set height and width values to h
    double getHeight()
    double getWidth()
}
```

Liskov substitution principle (LSP) - Example 1

```
void test(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    assertEquals(5 * 4, r.setWidth() * r.setHeight());
}
```

Interface Segregation Principle (ISP)

The dependency of one class to another one should depend on the smallest possible interface.

- Clients should not be forced to implement interfaces they don't use.
- Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one submodule.

Interface Segregation Principle (ISP) - Example 1

Animal
    void feed(); //abstract

Dog implements Animal
    void feed() //implementation

Tiger implements Animal
     void feed() //implementation

Interface Segregation Principle (ISP) - Example 1 - Enhanced to groom

Animal
    void feed(); //abstract
    void groom(); //abstract

Dog extends Animal
    void feed() //implementation
    void groom(); //implementation

Tiger extends Animal
      void feed() //implementation
      void groom(); //DUMMY implementation - to keep compiler happy

Interface Segregation Principle (ISP) - Example 1 - Better solution

Animal
    void feed(); //abstract

Pet extends Animal
    void groom(); //abstract

Dog extends Pet
    void feed() //implementation
    void groom(); //implementation

Tiger extends Animal
      void feed() //implementation

Dependency Inversion Principle (DIP)

Depend upon

abstractions (interfaces)

not upon concrete classes.

Dependency Inversion Principle (DIP) - Example 1

```
enum OutputDevice {printer, disk};
void copy(OutputDevice dev)
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
    {
      if (dev == printer)
            writePrinter(c);
      else
            writeDisk(c);
    }
}
```

Dependency Inversion Principle (DIP) - Example 1

```
interface Reader
    char read();

interface Writer
    char write(char ch);

void copy(Reader r, Writer w)
{
    char ch;
    while((ch = r.read())!=EOF) {
      w.write(ch);
    }
}
```

THANK YOU