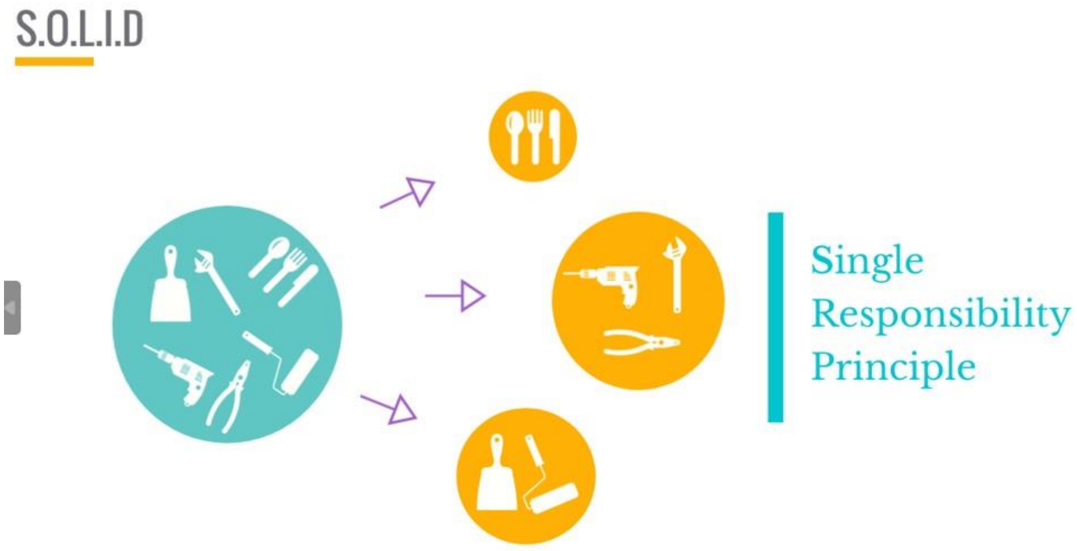
[S] The Single Responsibility Principle

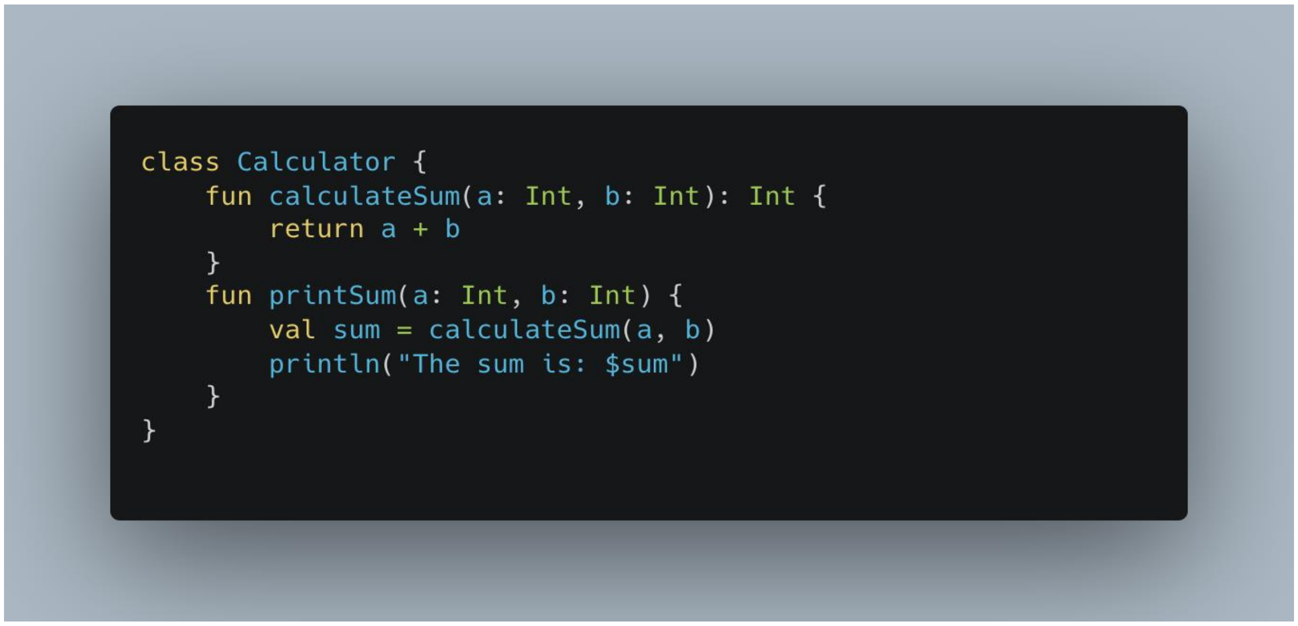


(SRP) is the first principle of the SOLID design principles. It states that a class should have one and only one reason to change, meaning that a class should have only one job.

Each class, method, or function should serve a single, well-defined purpose, with all elements within it supporting that purpose.

A class that has multiple responsibilities is often referred to as a "Blob class" and can be difficult to understand, modify and test.

An example of a bad practice that violates the SRP is a class that is responsible for both calculating and printing the results of a calculation.

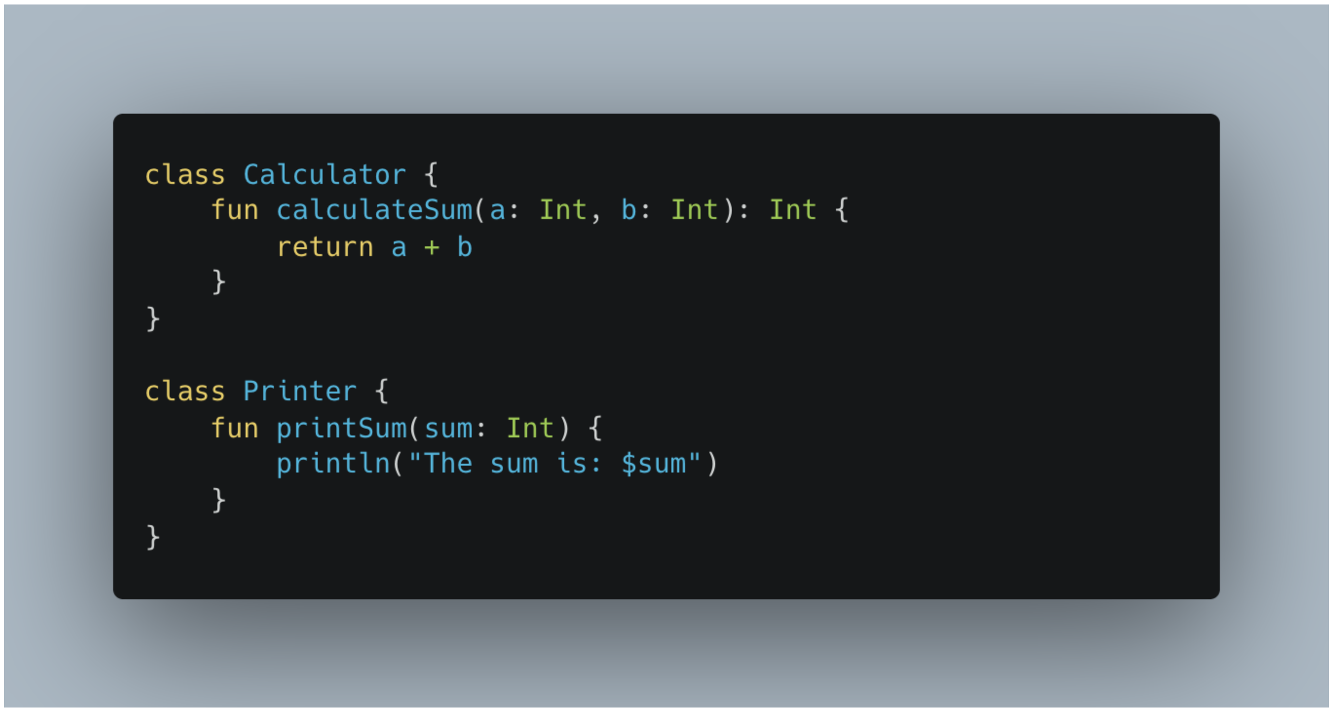


```kotlin
class Calculator {
    fun calculateSum(a: Int, b: Int): Int {
        return a + b
    }
    fun printSum(a: Int, b: Int) {
        val sum = calculateSum(a, b)
        println("The sum is: $sum")
    }
}
```

In this example, the **Calculator** class is responsible for both the calculation of the sum and the printing of the results.

This violates the SRP because the class has two reasons to change: if the calculation needs to be changed, or if the way the results are printed needs to be changed.

A better practice would be to separate the responsibilities of the class into two different classes, one for calculation and one for printing.



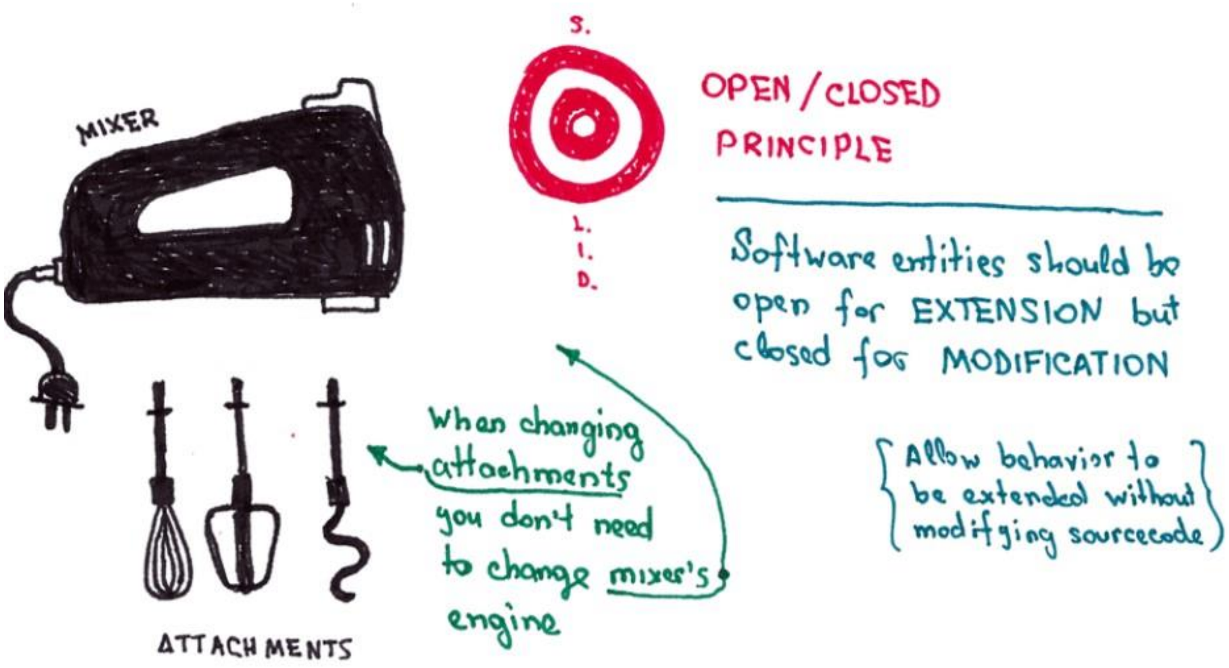
```kotlin
class Calculator {
    fun calculateSum(a: Int, b: Int): Int {
        return a + b
    }
}

class Printer {
    fun printSum(sum: Int) {
        println("The sum is: $sum")
    }
}
```

SRP good practice

In this example, the **Calculator** class is responsible only for the calculation of the sum, and the **Printer** class is responsible only for printing the results. This adheres to the SRP because each class has only one reason to change, making the code more maintainable.

[O] The Open-Closed Principle

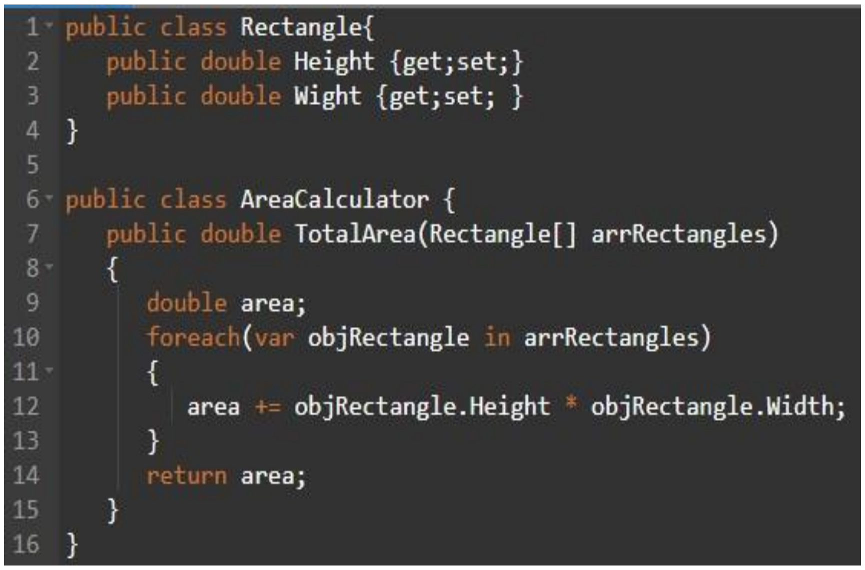
(OCP) is the second principle of the SOLID design principles It suggests that classes should be open for extension and closed to modification.


MIXER
S.
OPEN / CLOSED PRINCIPLE
L.
I.
D.
Software entities should be open for EXTENSION but closed for MODIFICATION
When changing attachments you don't need to change mixer's engine
Allow behavior to be extended without modifying sourcecode
ATTACHMENTS

Meaning that a class should be designed in such a way that new behavior can be added through inheritance or composition, but existing behavior should not be modified. Sometimes, we need to add certain functions to the existing class to perform additional tasks. So according to the Open-Closed Principle We should add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we risk creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible. In simple words, this principle aims to extend a Class's behavior without changing the existing behavior of that Class.

Suppose we have a Rectangle class with the properties Height and Width.

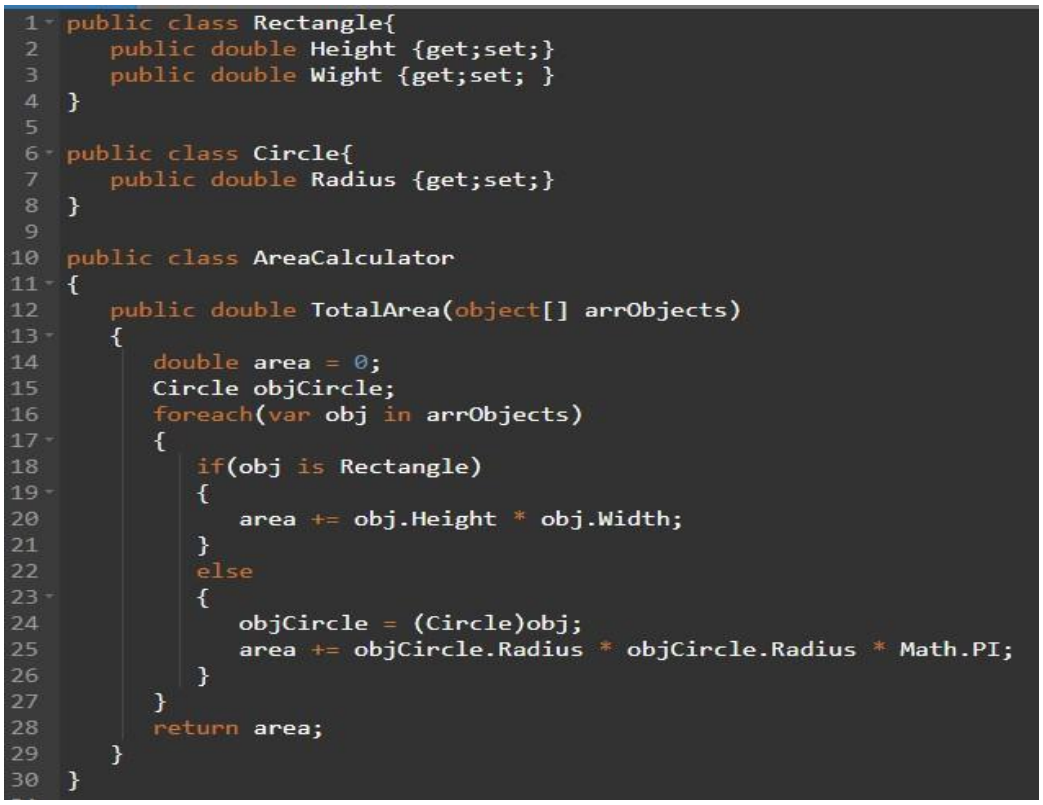
Our app needs to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here, I created another class for area calculation.



```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}

public class AreaCalculator {
   public double TotalArea(Rectangle[] arrRectangles)
   {
      double area;
      foreach(var objRectangle in arrRectangles)
      {
         area += objRectangle.Height * objRectangle.Width;
      }
      return area;
   }
}
```

Hey, we did it. We made our app without violating SRP. No issues for now. But can we extend our app so that it can calculate the area of not only Rectangles but also the area of Circles? Now we have an issue with the area calculation issue because the way to calculate the circle area is different.

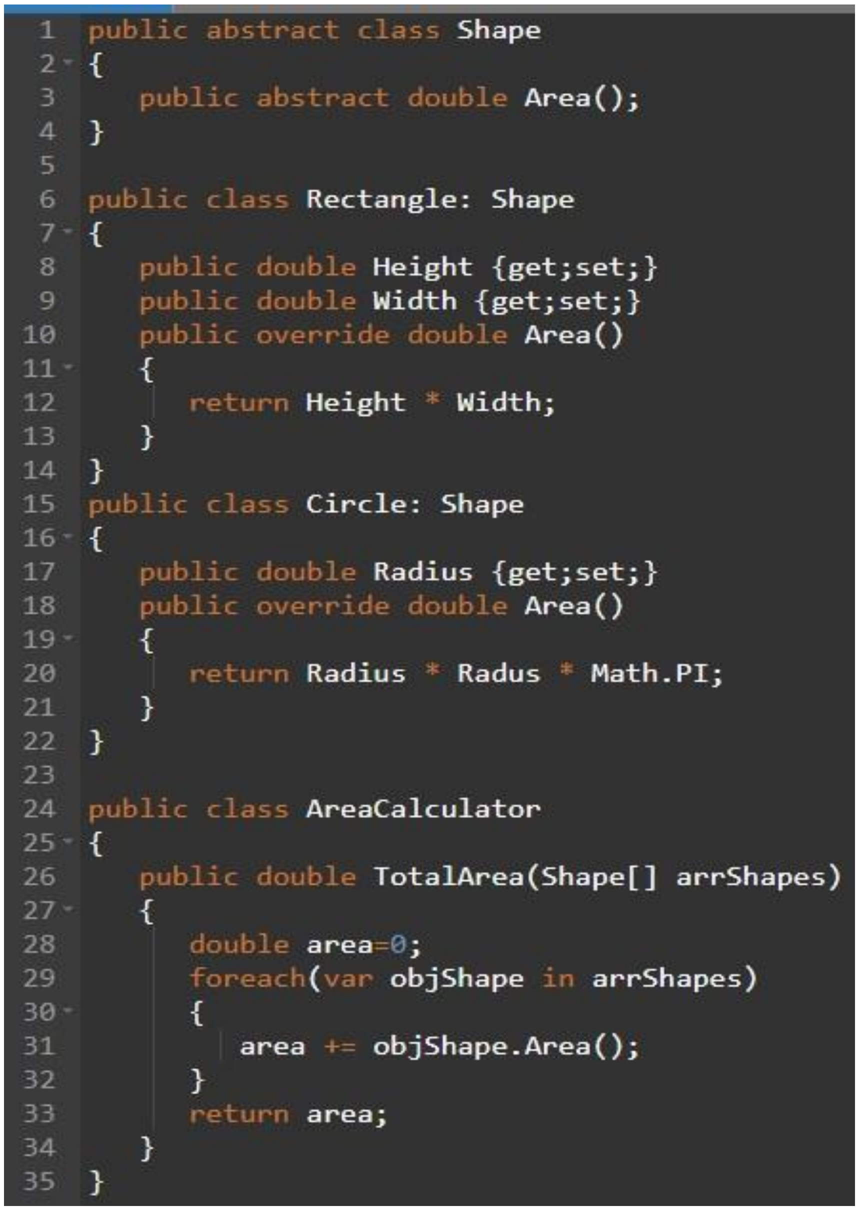
Hmm. Not a big deal. We can change the TotalArea method to accept an array of objects as an argument. We check the object type in the loop and do area calculations based on the object type.



```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}

public class Circle{
   public double Radius {get;set;}
}

public class AreaCalculator
{
   public double TotalArea(object[] arrObjects)
   {
      double area = 0;
      Circle objCircle;
      foreach(var obj in arrObjects)
      {
         if(obj is Rectangle)
         {
            area += obj.Height * obj.Width;
         }
         else
         {
            objCircle = (Circle)obj;
            area += objCircle.Radius * objCircle.Radius * Math.PI;
         }
      }
      return area;
   }
}
```

We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate its area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape, we must alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation?

Generally, we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.



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

public class Rectangle: Shape
{
    public double Height {get;set;}
    public double Width {get;set;}
    public override double Area()
    {
        return Height * Width;
    }
}
public class Circle: Shape
{
    public double Radius {get;set;}
    public override double Area()
    {
        return Radius * Radus * Math.PI;
    }
}

public class AreaCalculator
{
    public double TotalArea(Shape[] arrShapes)
    {
        double area=0;
        foreach(var objShape in arrShapes)
        {
            area += objShape.Area();
        }
        return area;
    }
}
```

Inheriting from Shape, the Rectangle and Circle classes now look like this:

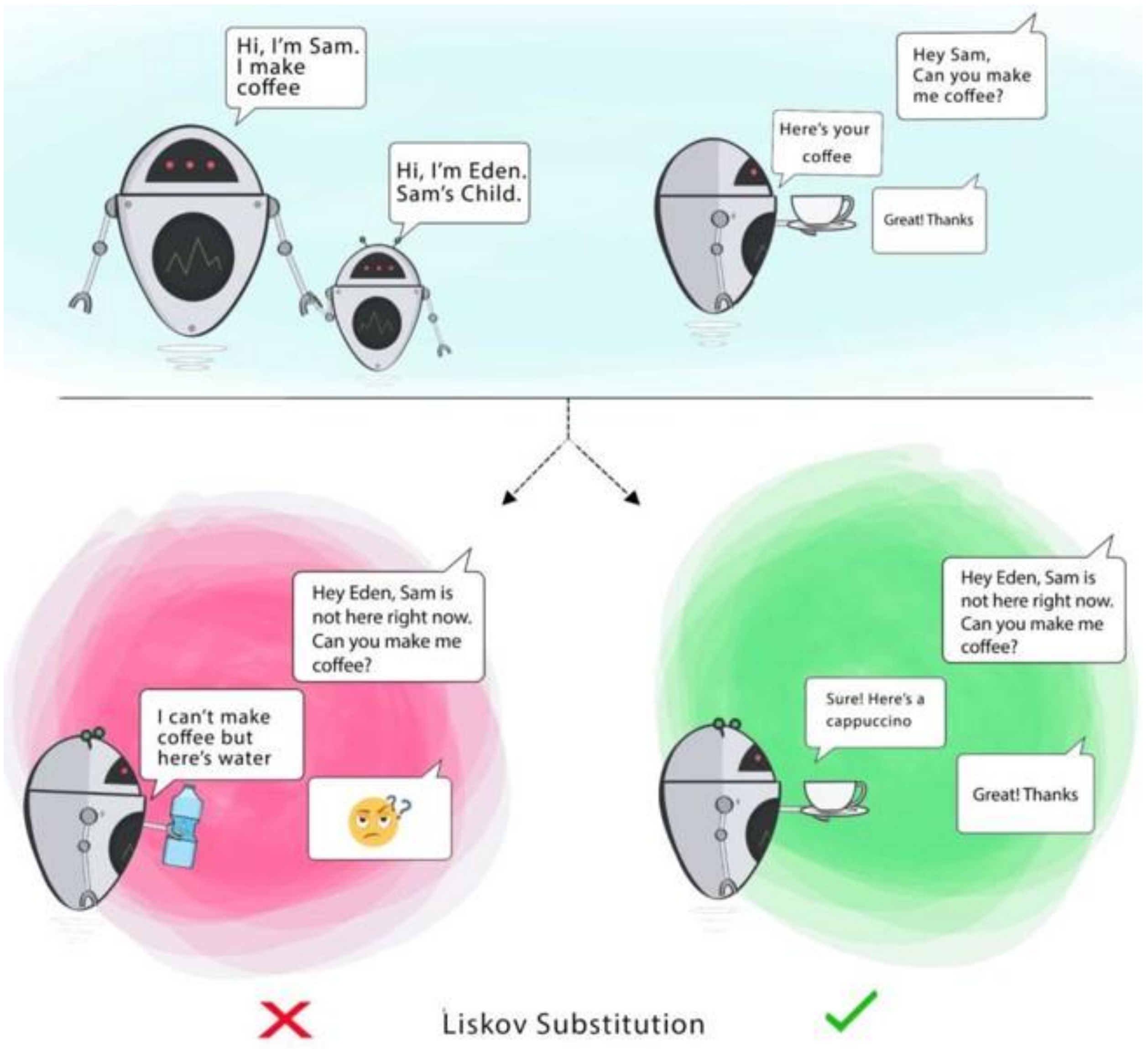
Every shape contains its area with its way of calculation functionality, and our AreaCalculator class will become simpler than before

Now our code is following SRP and OCP both. Whenever you introduce a new shape by deriving from the "Shape" abstract class, you need not change the "AreaCalculator" class.

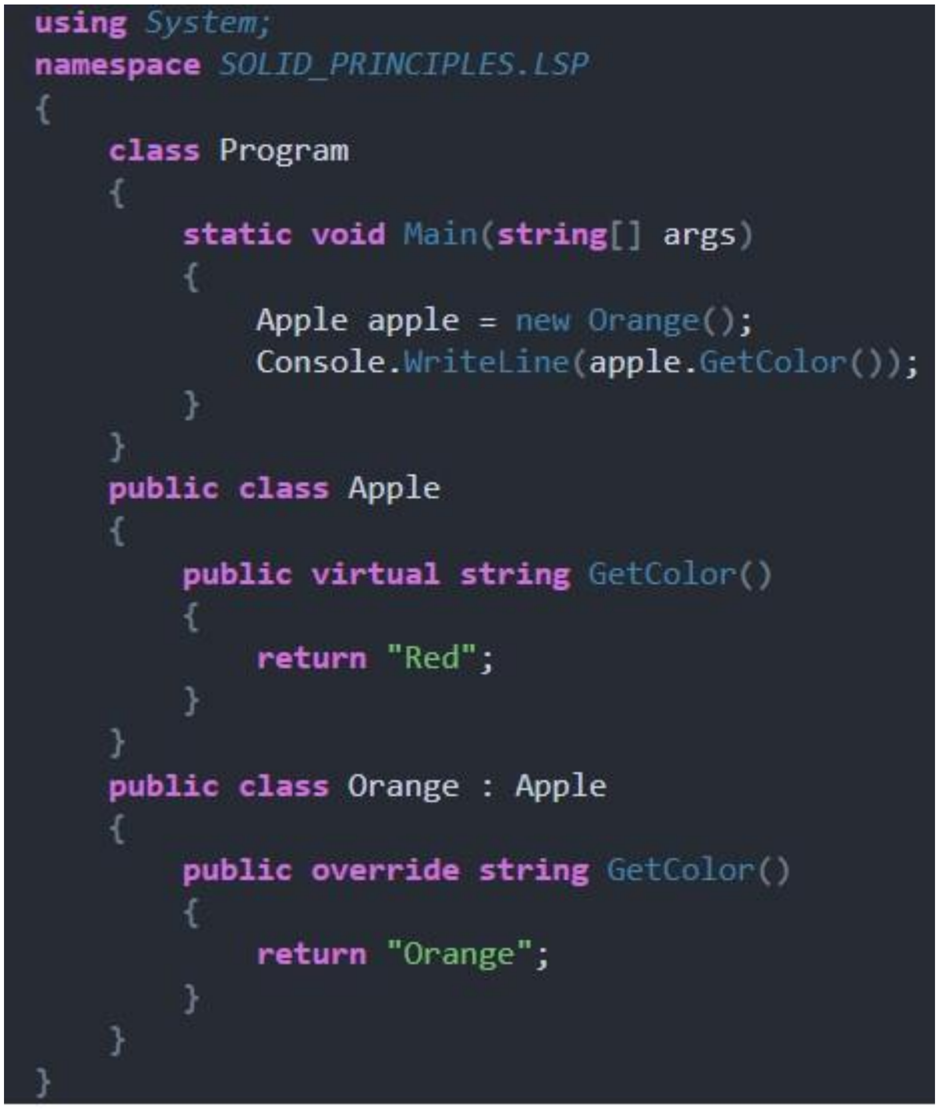
[L] The Liskov Substitution Principle

(LSP) is the third principle of the SOLID design principles. It states that subtypes should be substitutable for their base types, meaning that objects of a superclass should be able to be replaced with objects of a subclass without altering the correctness of the program. This principle helps to ensure that objects of a subclass can be used interchangeably with objects of the superclass, without introducing any unexpected behavior.

An example of a bad practice that violates the LSP is a class hierarchy where a subclass overrides a method in a way that changes the method's contract.


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

Let us first understand one example without using the Liskov Substitution Principle in C#. We will see the problem if we are not following the Liskov Substitution Principle, and then we will see how we can overcome such problems using the Liskov Substitution Principle. In the following example, first, we create the Apple class with the method GetColor. Then, we create the Orange class, which inherits the Apple class and overrides the GetColor method of the Apple class. The point is that an Orange cannot be replaced by an Apple, which results in printing the color of the apple as Orange, as shown in the example below.

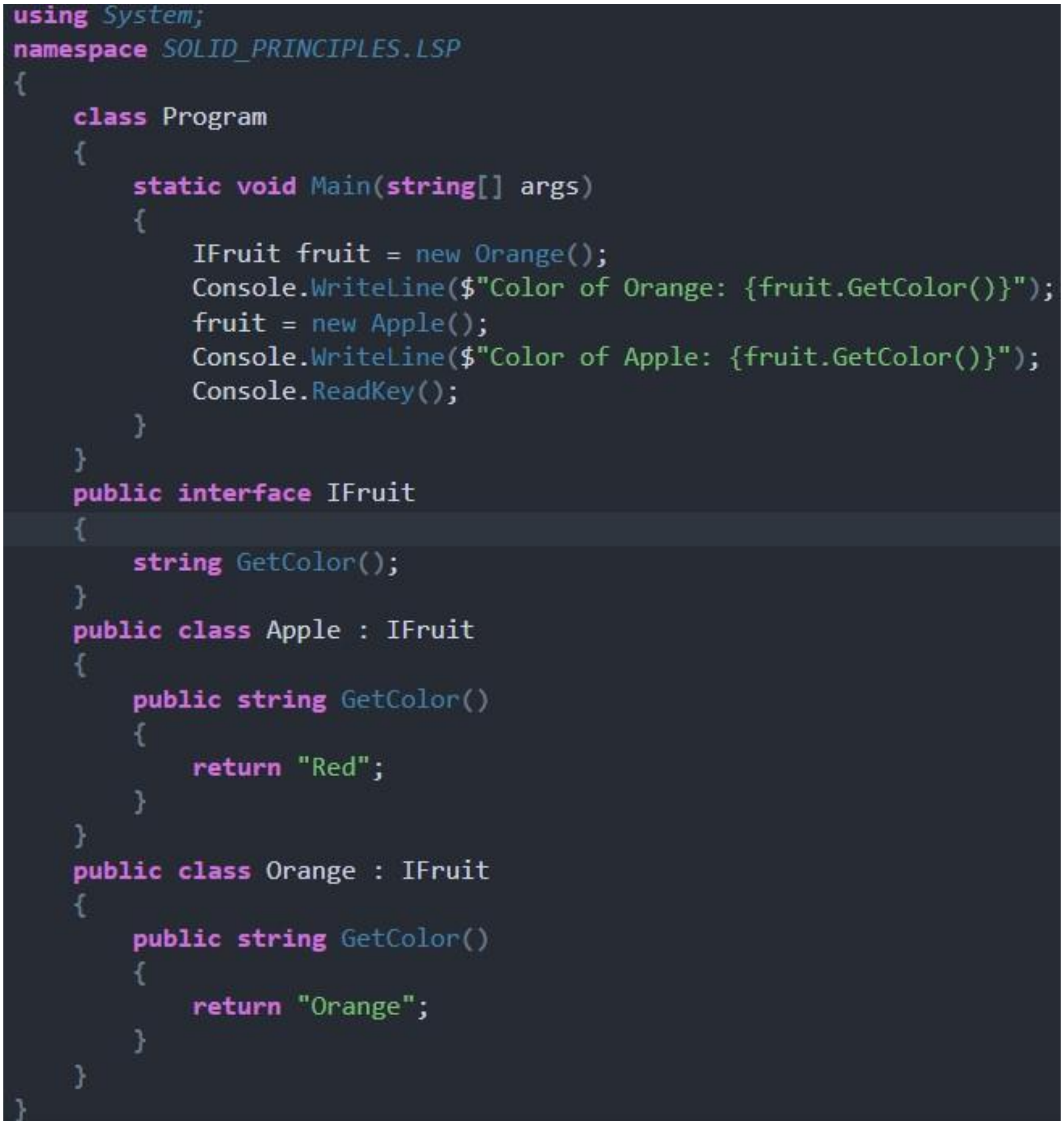


```
using System;
namespace SOLID_PRINCIPLES.LSP
{
    class Program
    {
        static void Main(string[] args)
        {
            Apple apple = new Orange();
            Console.WriteLine(apple.GetColor());
        }
    }
    public class Apple
    {
        public virtual string GetColor()
        {
            return "Red";
        }
    }
    public class Orange : Apple
    {
        public override string GetColor()
        {
            return "Orange";
        }
    }
}
```

As you can see in the above example, Apple is the base class, and Orange is the child class, i.e., there is a Parent-Child relationship. So, we can store the child class object in the Parent class Reference variable, i.e., **Apple apple = new Orange();** and when we call the GetColor, i.e., apple.GetColor(), then we are getting the color Orange, not the color of an Apple. That means the behavior changes once the child object is replaced, i.e., Apple stores the Orange object. This is against the LSP Principle.

The Liskov Substitution Principle states that even if the child object is replaced with the parent, the behavior should not be changed. So, in this case, if we are getting the color Apple instead of Orange, it follows the Liskov Substitution Principle. That means there is some issue with our software design. Let us see how to overcome the design issue and make the application follow the Liskov Substitution Principle using C# Langauge.

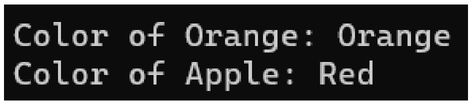
Example Using the Liskov Substitution Principle in C#



```
using System;
namespace SOLID_PRINCIPLES.LSP
{
    class Program
    {
        static void Main(string[] args)
        {
            IFruit fruit = new Orange();
            Console.WriteLine($"Color of Orange: {fruit.GetColor()}");
            fruit = new Apple();
            Console.WriteLine($"Color of Apple: {fruit.GetColor()}");
            Console.ReadKey();
        }
    }
    public interface IFruit
    {
        string GetColor();
    }
    public class Apple : IFruit
    {
        public string GetColor()
        {
            return "Red";
        }
    }
    public class Orange : IFruit
    {
        public string GetColor()
        {
            return "Orange";
        }
    }
}
```

Let’s modify the previous example to follow the Liskov Substitution Principle using C# Language. First, we need a generic base Interface, i.e., IFruit, which will be the base class for both Apple and Orange. Now, you can replace the IFruit variable can be replaced with its subtypes, either Apple or Orage, and it will behave correctly. In the code below, we created the super IFruit as an interface with the GetColor method. Then, the Apple and Orange classes were inherited from the Fruit class and implemented the GetColor method.

Now, run the application, and it should give the expected output, as shown in the image below. Here, we follow the LSP as we can change the object with its subtype without affecting the behavior.



```
Color of Orange: Orange
Color of Apple: Red
```

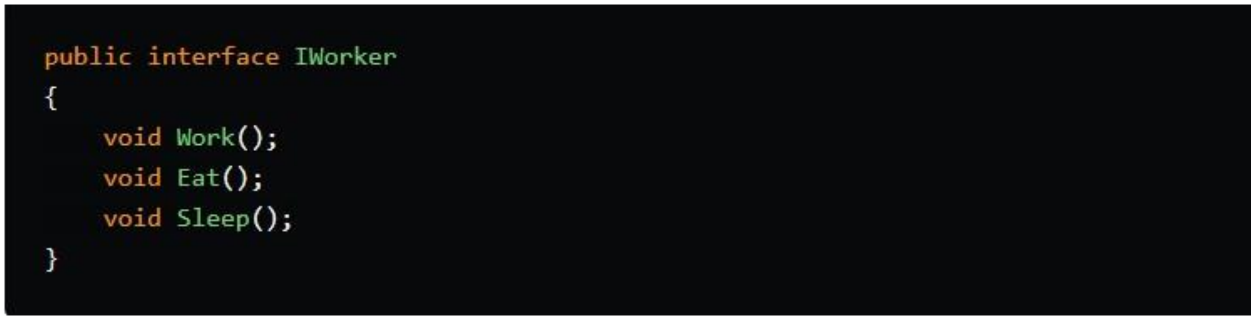
So, now Fruit can be any type and any color, but orange cannot be the color red. An apple cannot be of the color orange, meaning we cannot replace an orange with an apple, but fruit can be replaced with an orange or an apple because they are both Fruits; an apple is not an orange, and an orange is not an apple.

[I] The Interface Segregation Principle

(ISP) is the fourth principle of the SOLID design principles. It states that a class should not be forced to implement interfaces it does not use, meaning that a class should not be forced to implement methods it does not need. This principle encourages creating small, specific interfaces that are tailored to the needs of specific classes, rather than creating large, general interfaces that require classes to implement many methods they do not need.

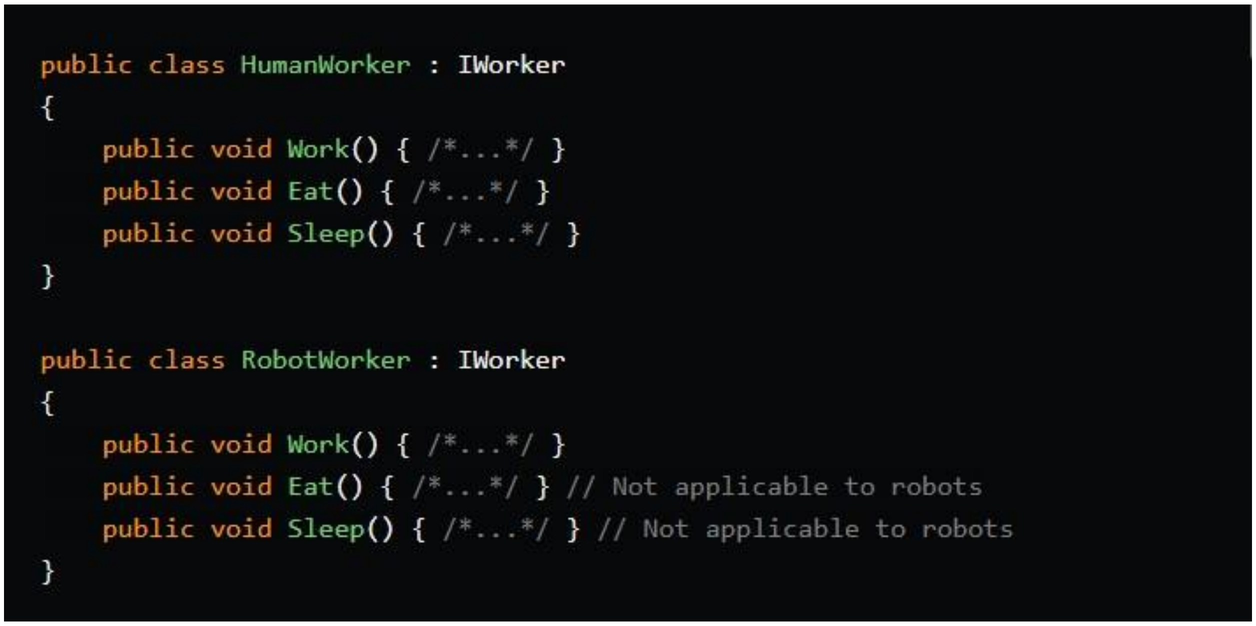
An example of a bad practice that violates the ISP is a class that implements a large, general interface that contains many methods that the class does not need.

Let's consider an example to illustrate the ISP in C#. Suppose we have an IWorker interface that defines methods for various types of work, such as Work, Eat, and Sleep.



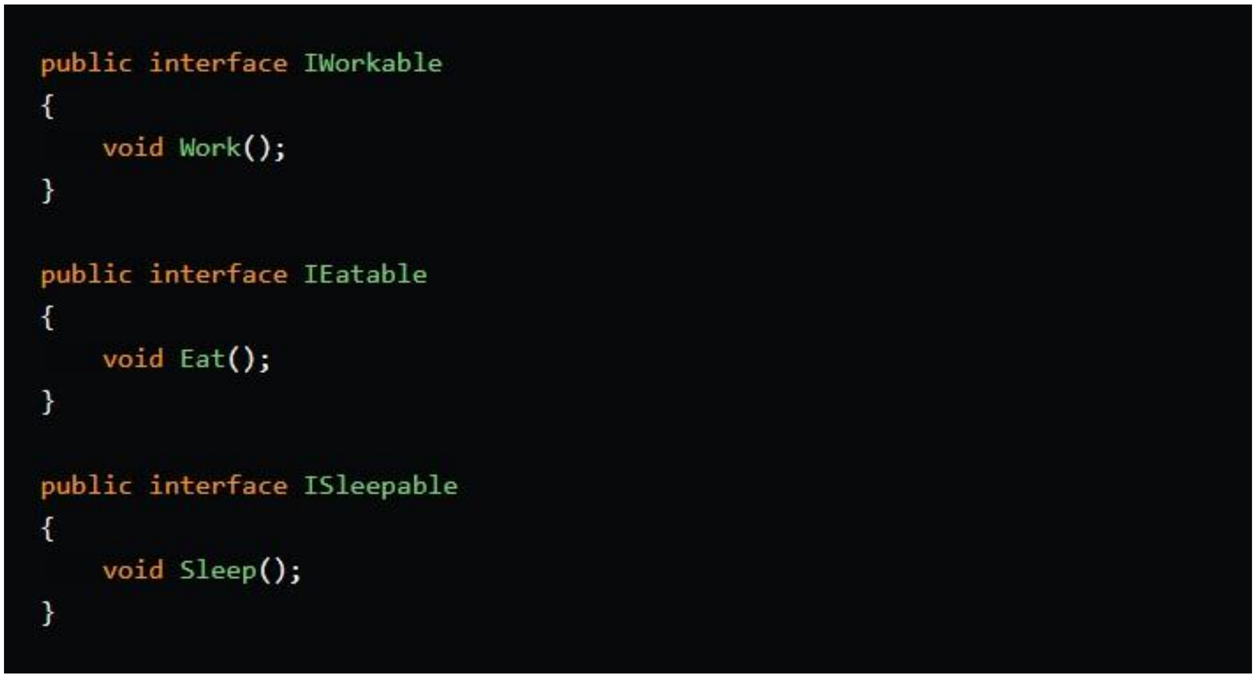
```
public interface IWorker
{
    void Work();
    void Eat();
    void Sleep();
}
```

Now, imagine we have two classes, HumanWorker and RobotWorker, that implement this interface:



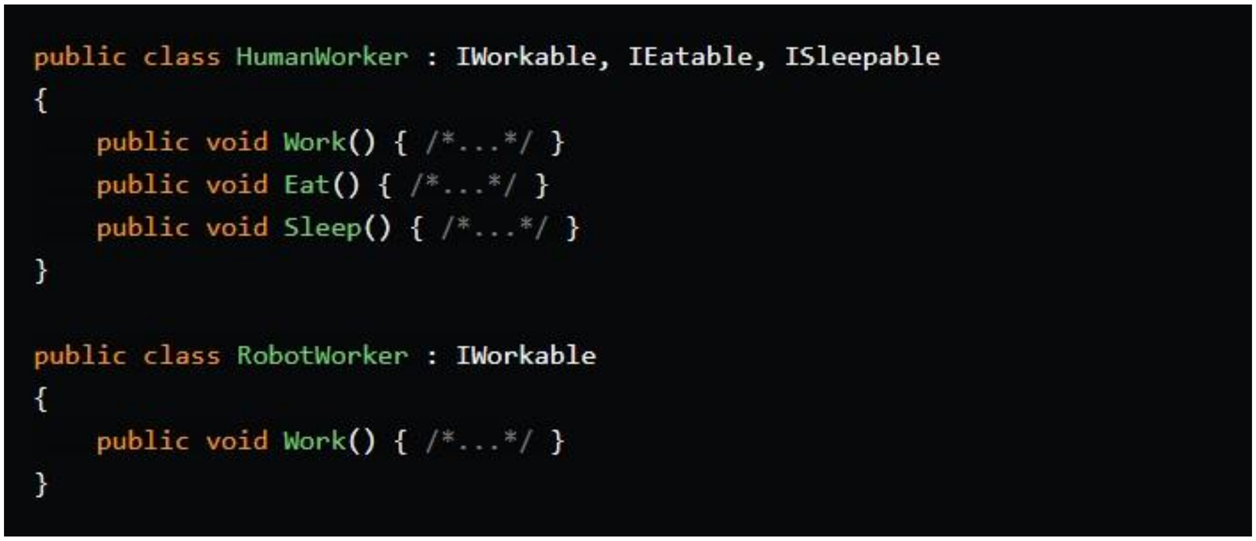
```
public class HumanWorker : IWorker
{
    public void Work() { /*...*/ }
    public void Eat() { /*...*/ }
    public void Sleep() { /*...*/ }
}

public class RobotWorker : IWorker
{
    public void Work() { /*...*/ }
    public void Eat() { /*...*/ } // Not applicable to robots
    public void Sleep() { /*...*/ } // Not applicable to robots
}
```

In this example, the IWorker interface violates the ISP, as it forces the RobotWorker class to implement the Eat and Sleep methods, which are irrelevant for robots. To adhere to the ISP, we can break down the IWorker interface into smaller, more specific interfaces:



```
public interface IWorkable
{
    void Work();
}

public interface IEatable
{
    void Eat();
}

public interface ISleepable
{
    void Sleep();
}
```

Now, the HumanWorker class can implement all three interfaces, while the RobotWorker class only needs to implement the IWorkable interface:



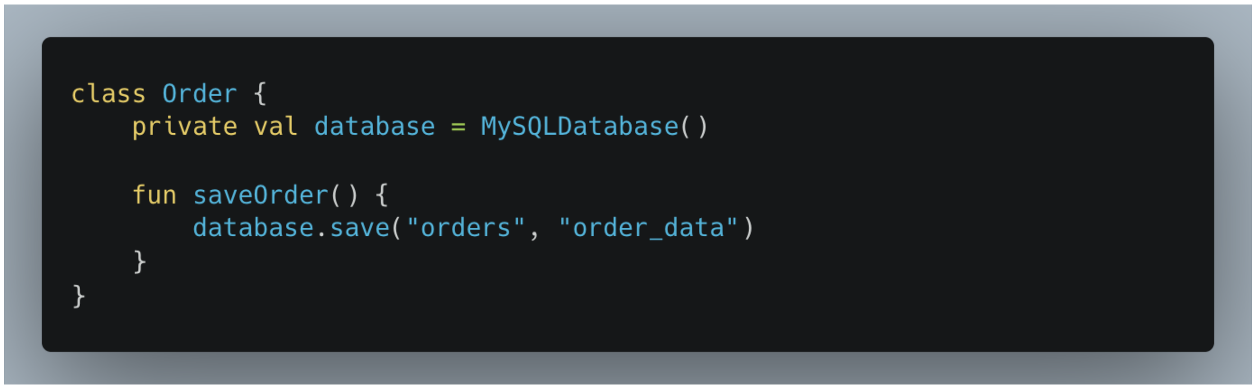
```
public class HumanWorker : IWorkable, IEatable, ISleepable
{
    public void Work() { /*...*/ }
    public void Eat() { /*...*/ }
    public void Sleep() { /*...*/ }
}

public class RobotWorker : IWorkable
{
    public void Work() { /*...*/ }
}
```

By segregating the interfaces, we have ensured that clients only implement the methods they actually need, adhering to the ISP.

[D] The Dependency Inversion Principle

(DIP) is the fifth principle of the SOLID design principles. It states that high-level modules should not depend on low-level modules, but both should depend on abstractions, meaning that a class should depend on abstractions rather than concretions. This principle promotes a design where the high-level modules (such as the business logic) are not tightly coupled to the low-level modules (such as the data access layer), making the code more flexible and maintainable.

An example of a bad practice that violates the DIP is a class that depends on a specific implementation of a low-level module.

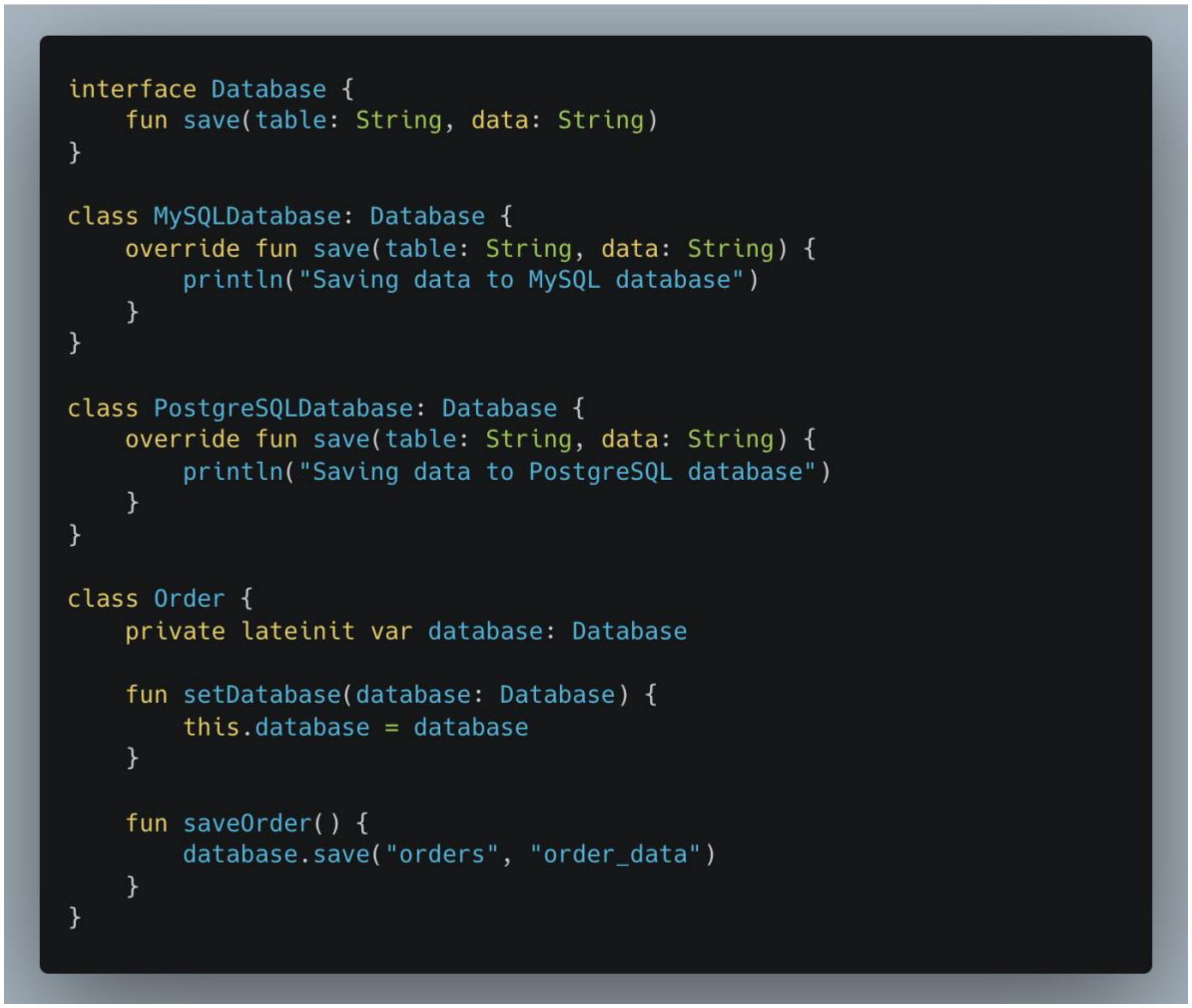


```
class Order {
    private val database = MySQLDatabase()

    fun saveOrder() {
        database.save("orders", "order_data")
    }
}
```

Dip bad practice

In this example, the **Order** class depends on a specific implementation of a low-level module, the **MySQLDatabase** class. This violates the DIP principle because the **Order** class is tightly coupled to the specific implementation of the **MySQLDatabase** class. If we want to change the database to PostgreSQL or any other database, we need to change the **Order** class as well.

A better practice would be to create an abstraction for the low-level module and have the high-level module depend on the abstraction.



```kotlin
interface Database {
    fun save(table: String, data: String)
}

class MySQLDatabase: Database {
    override fun save(table: String, data: String) {
        println("Saving data to MySQL database")
    }
}

class PostgreSQLDatabase: Database {
    override fun save(table: String, data: String) {
        println("Saving data to PostgreSQL database")
    }
}

class Order {
    private lateinit var database: Database

    fun setDatabase(database: Database) {
        this.database = database
    }

    fun saveOrder() {
        database.save("orders", "order_data")
    }
}
```

Dip good practice

In this example, the **Order** class depends on an abstraction, the **Database** interface, rather than a specific implementation of a low-level module. This adheres to the DIP principle, making the code more flexible and maintainable. Now we can change the database to any other database by just creating a new implementation of the **Database** interface and injecting it into the **Order** class.