Programming in C#. Fundamentals

Lesson 5
Object Oriented Programming

Object Oriented Programming

Inheritance
Polymorphism
Abstract Classes
Interfaces
Common Interfaces
SOLID, KISS, DRY, YAGNI

Key Concepts

Inheritance

Polymorphism

Encapsulation

Inheritance

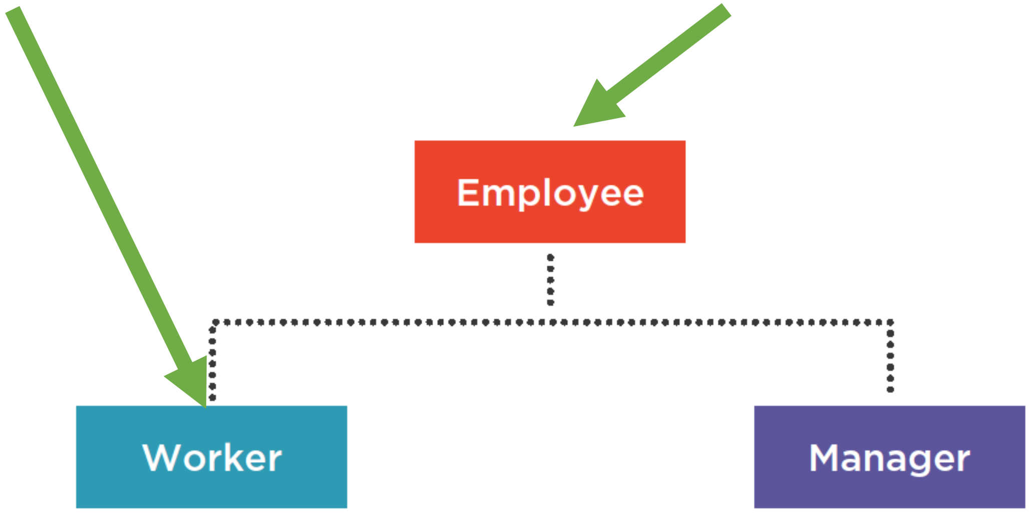
Classes may derive from existing classes

Derive

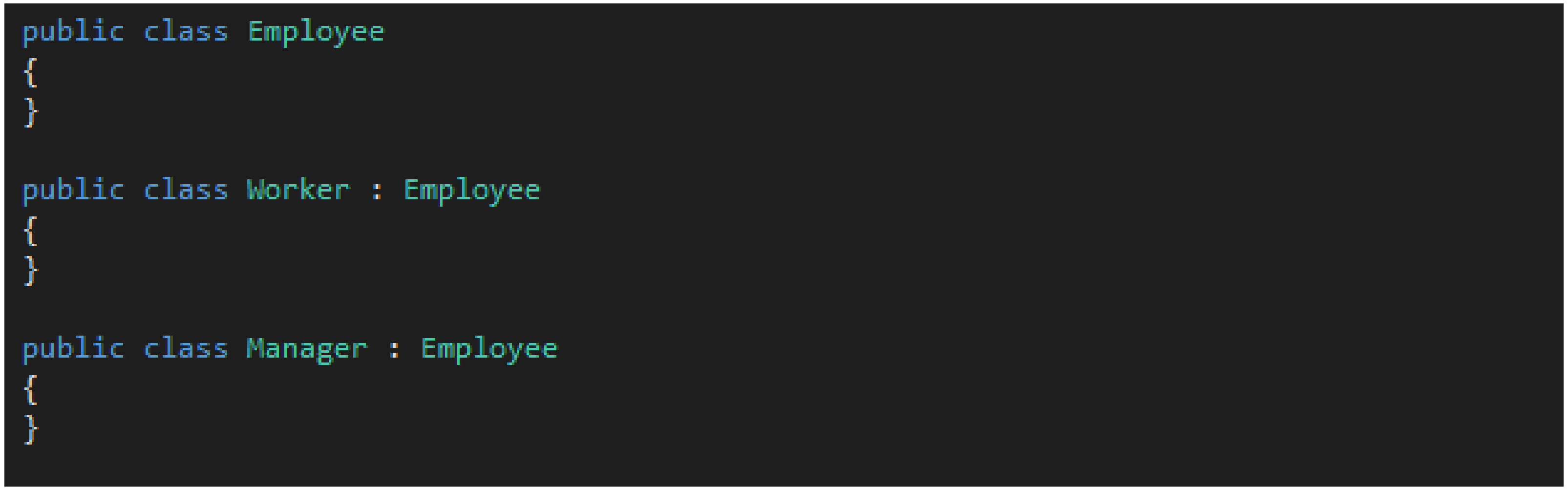
Specialize the “parent” class

Parent Class

Generalization of the derived classes



Inheritance creates an “is –a ” relationship



```
public class Employee
{
}

public class Worker : Employee
{
}

public class Manager : Employee
{
}
```

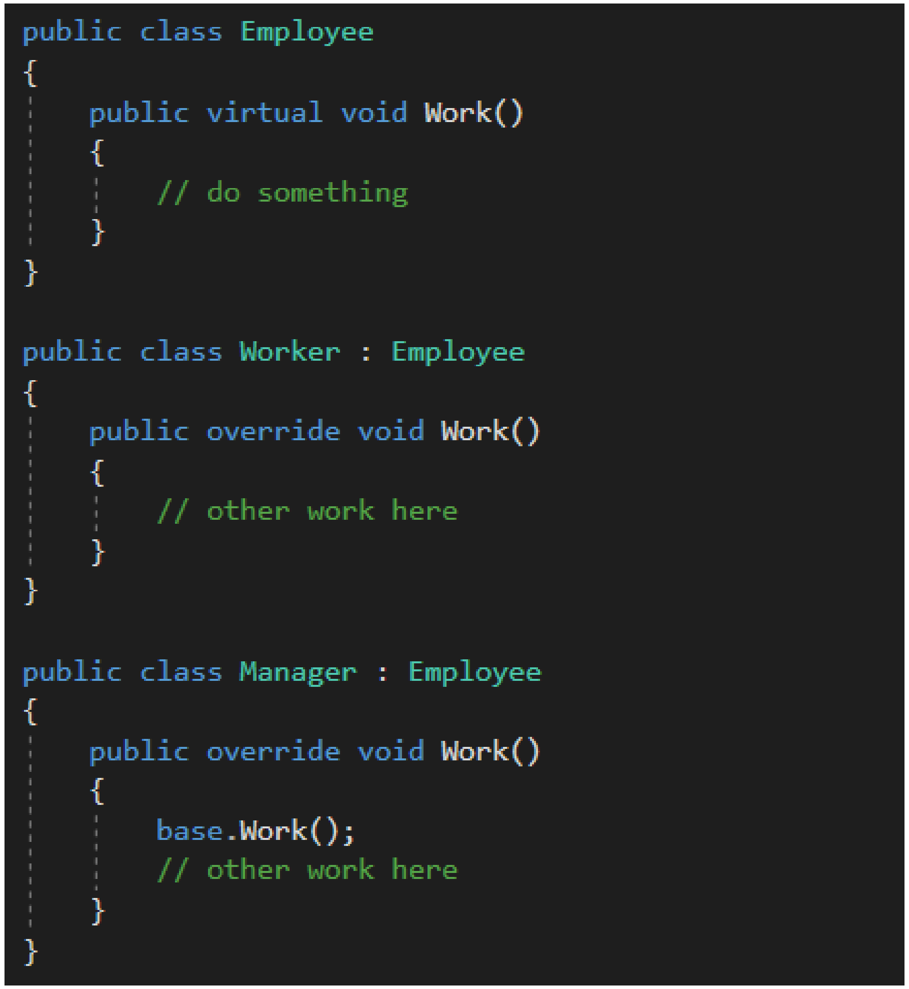
Derived Class Indicates Base Class with Colon

Polymorphism

Taking Many Forms

Method Overriding

Modifying a method in the derived class

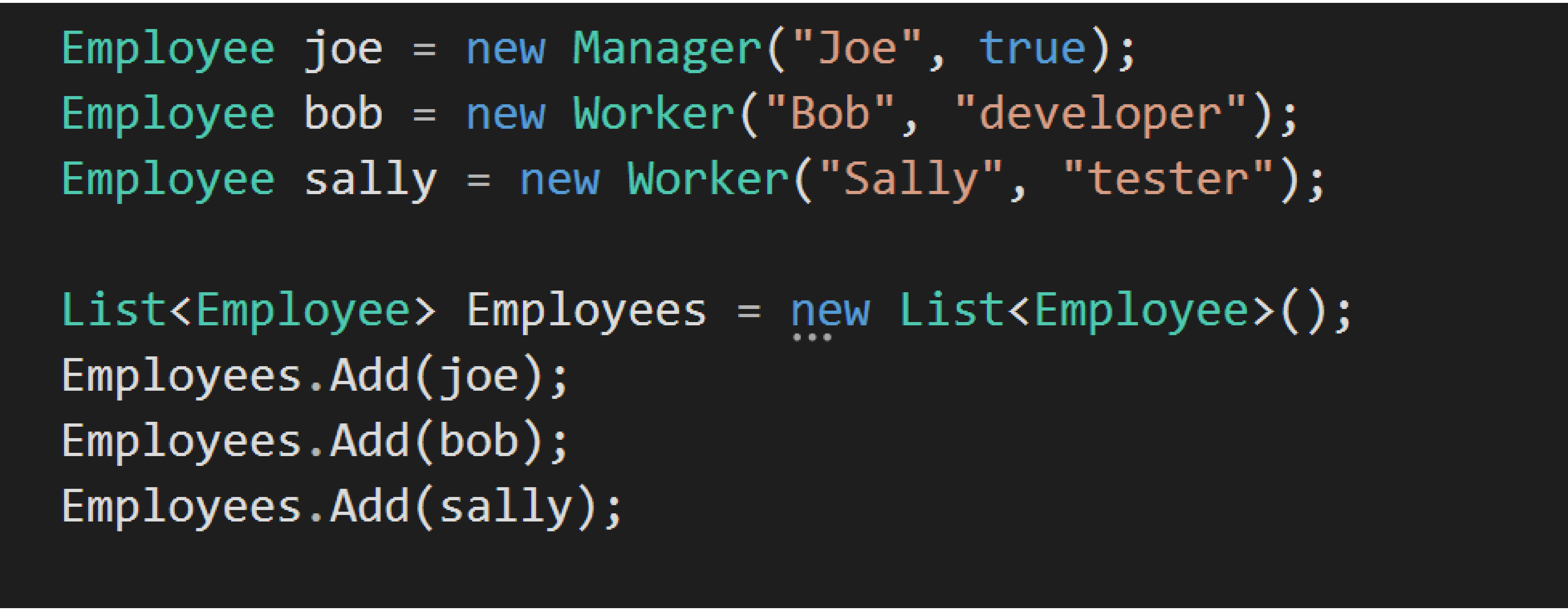


```
public class Employee
{
    public virtual void Work()
    {
        // do something
    }
}

public class Worker : Employee
{
    public override void Work()
    {
        // other work here
    }
}

public class Manager : Employee
{
    public override void Work()
    {
        base.Work();
        // other work here
    }
}
```

Virtual and Overridden Methods

Chaining up to the Parent (base) Class

Derived and base classes can be treated polymorphically



```
Employee joe = new Manager("Joe", true);
Employee bob = new Worker("Bob", "developer");
Employee sally = new Worker("Sally", "tester");

List<Employee> Employees = new List<Employee>();
Employees.Add(joe);
Employees.Add(bob);
Employees.Add(sally);
```

Encapsulation

Most of the internals of the class are private, with a few well-defined properties and methods that are public.

Abstract Classes

Exist to provide a base class, but are never instantiated

Abstract vs. Concrete Classes

Concrete Class

Acts as a base class to other classes
Can be instantiated
Cannot have abstract methods
Child classes may override methods

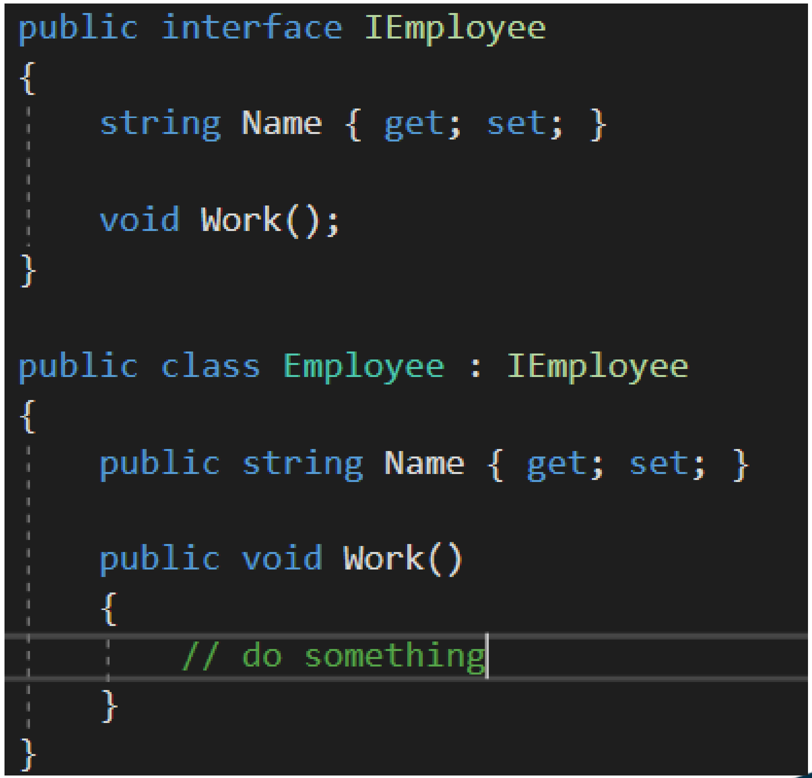
Abstract Class

Acts as a base class to other classes
Can not be instantiated
Has at least one abstract method
Concrete child classes must override all abstract methods

Interfaces

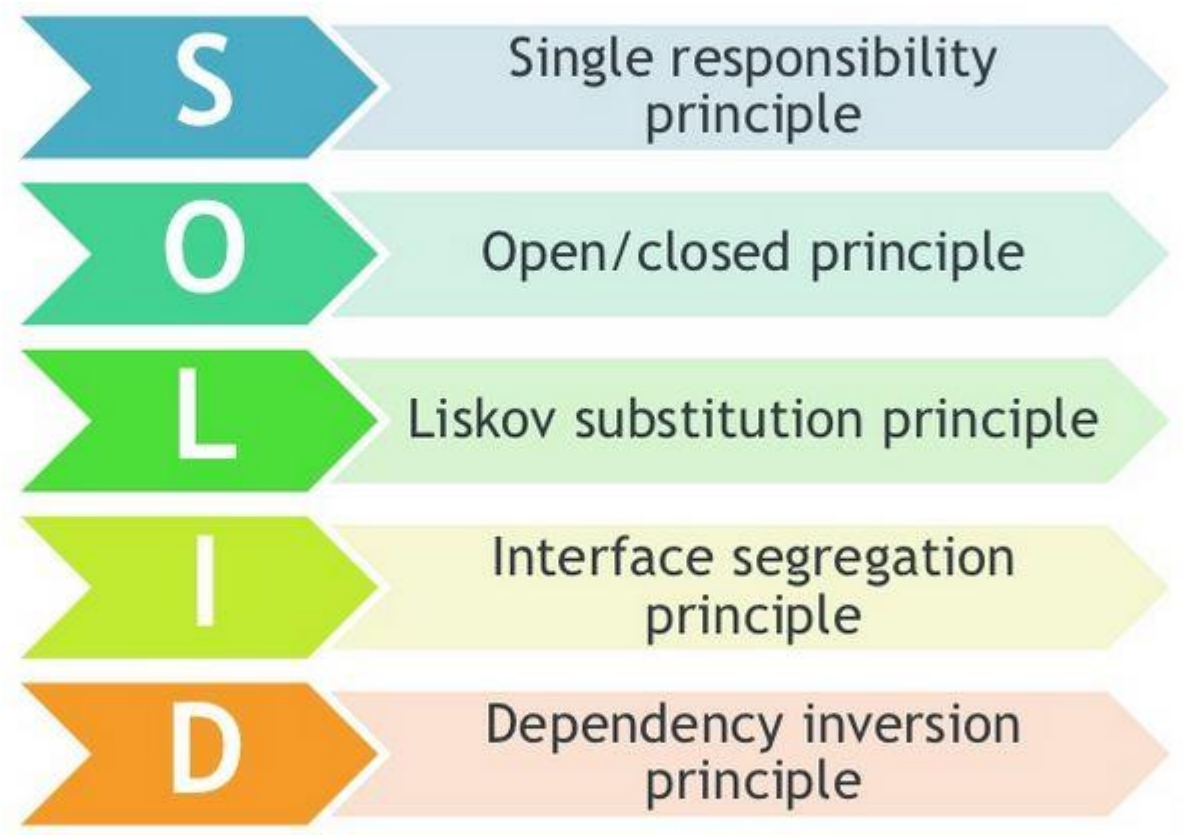
An Interface is a Contract

- Multiple inheritance
- Contain methods, properties, indexers, and events
- Private interface implementations
- An interface is fulfilled by a class
- You Can't Instantiate an Interface

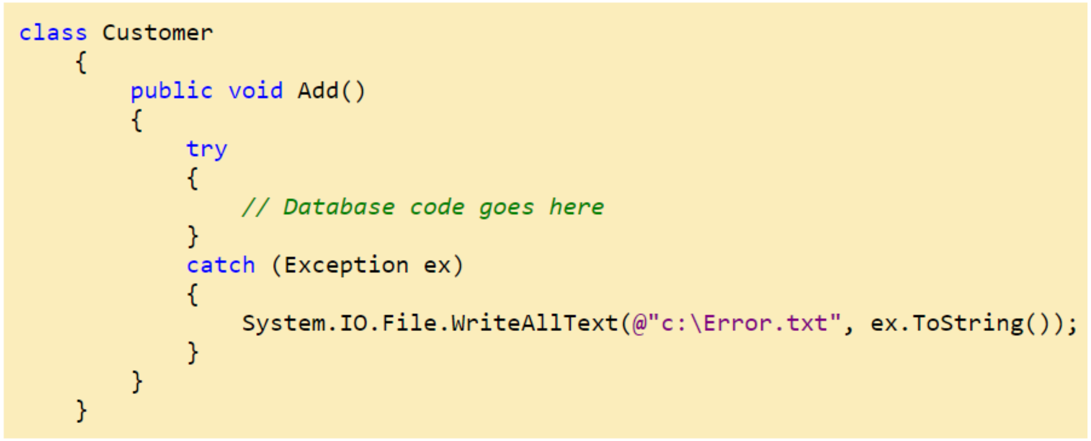


```
public interface IEmployee
{
    string Name { get; set; }

    void Work();
}

public class Employee : IEmployee
{
    public string Name { get; set; }

    public void Work()
    {
        // do something
    }
}
```

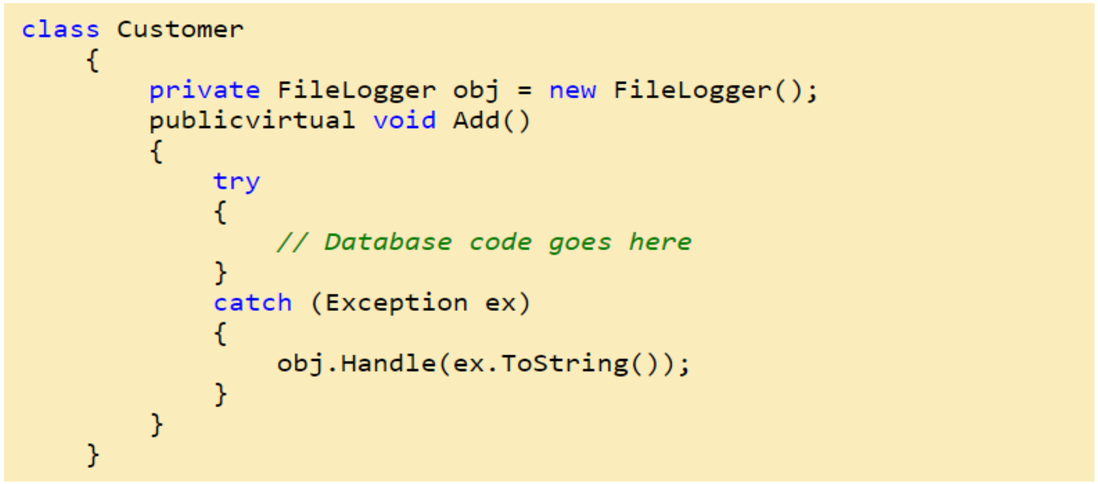
SOLID


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

“S”- Single responsibility principle

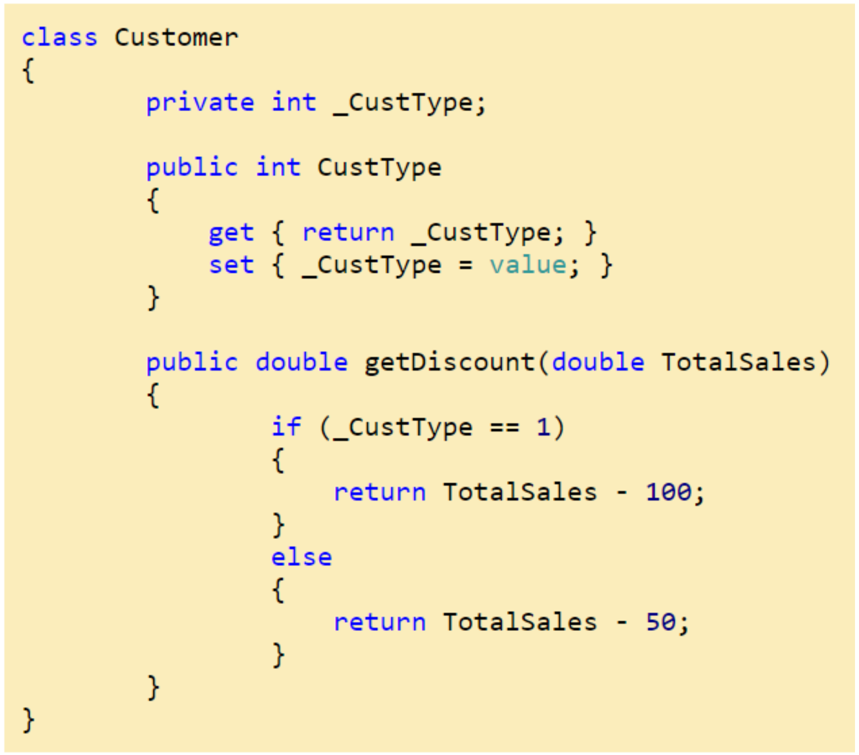


```
class Customer
    {
        public void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
            }
        }
    }
```

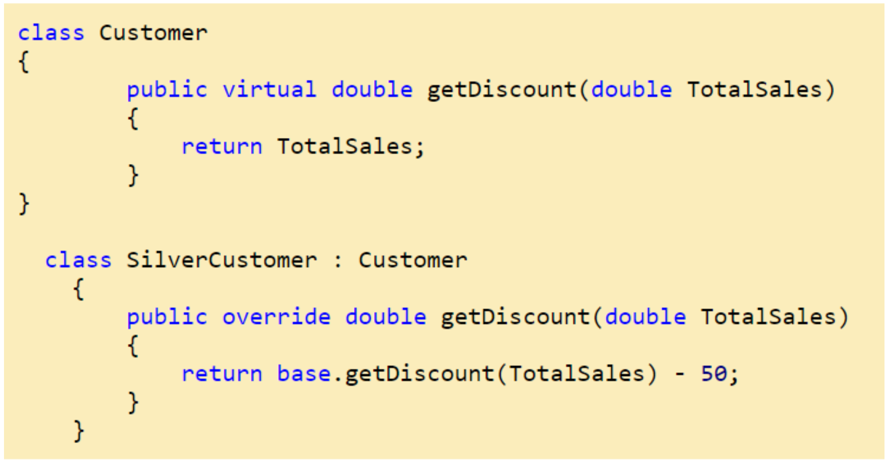


```
class Customer
    {
        private FileLogger obj = new FileLogger();
        publicvirtual void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                obj.Handle(ex.ToString());
            }
        }
    }
```

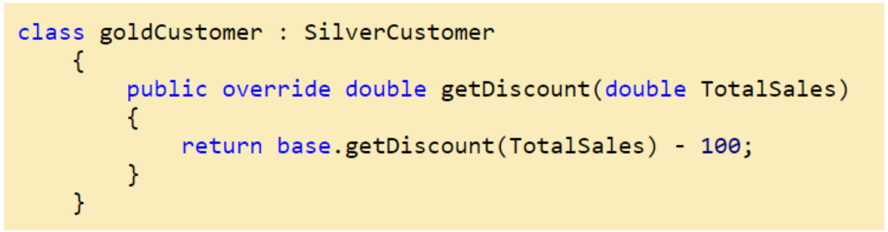
“O” - Open closed principle



```
class Customer
{
        private int _CustType;

        public int CustType
        {
            get { return _CustType; }
            set { _CustType = value; }
        }

        public double getDiscount(double TotalSales)
        {
                if (_CustType == 1)
                {
                    return TotalSales - 100;
                }
                else
                {
                    return TotalSales - 50;
                }
        }
}
```



```
class Customer
{
        public virtual double getDiscount(double TotalSales)
        {
            return TotalSales;
        }
}

  class SilverCustomer : Customer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 50;
        }
    }
```



```
class goldCustomer : SilverCustomer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 100;
        }
    }
```

“L”- Liskov substitution principle

```
class Enquiry : Customer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 5;
        }

        public override void Add()
        {
            throw new Exception("Not allowed");
        }
    }
```

```
List<Customer> Customers = new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

 foreach (Customer o in Customers)
 {
               o.Add();
 }
}
```

“I” - Interface Segregation principle

```
interface IDatabase
{
        void Add(); // old client are happy with these.
voidRead(); // Added for new clients.
}
```

```
interface IDatabaseV1 : IDatabase // Gets the Add method
{
Void Read();
}
```

```
IDatabase i = new Customer(); // 1000 happy old clients not touched
i.Add();

IDatabaseV1 iv1 = new CustomerWithread(); // new clients
Iv1.Read();
```

“D”- Dependency inversion principle

```
class Customer : IDiscount, IDatabase
 {
        private Ilogger obj;
        public Customer(ILogger i)
        {
            obj = i;
        }
}
```

```
IDatabase i = new Customer(new EmailLogger());
```

KISS

THE KISS PRINCIPLE

KEEP
IT
SIMPLE,
STUPID

DRY

Y.A.G.N.I

You ain’t gonna need it

Q & A

Practice

Lesson 5

Home work