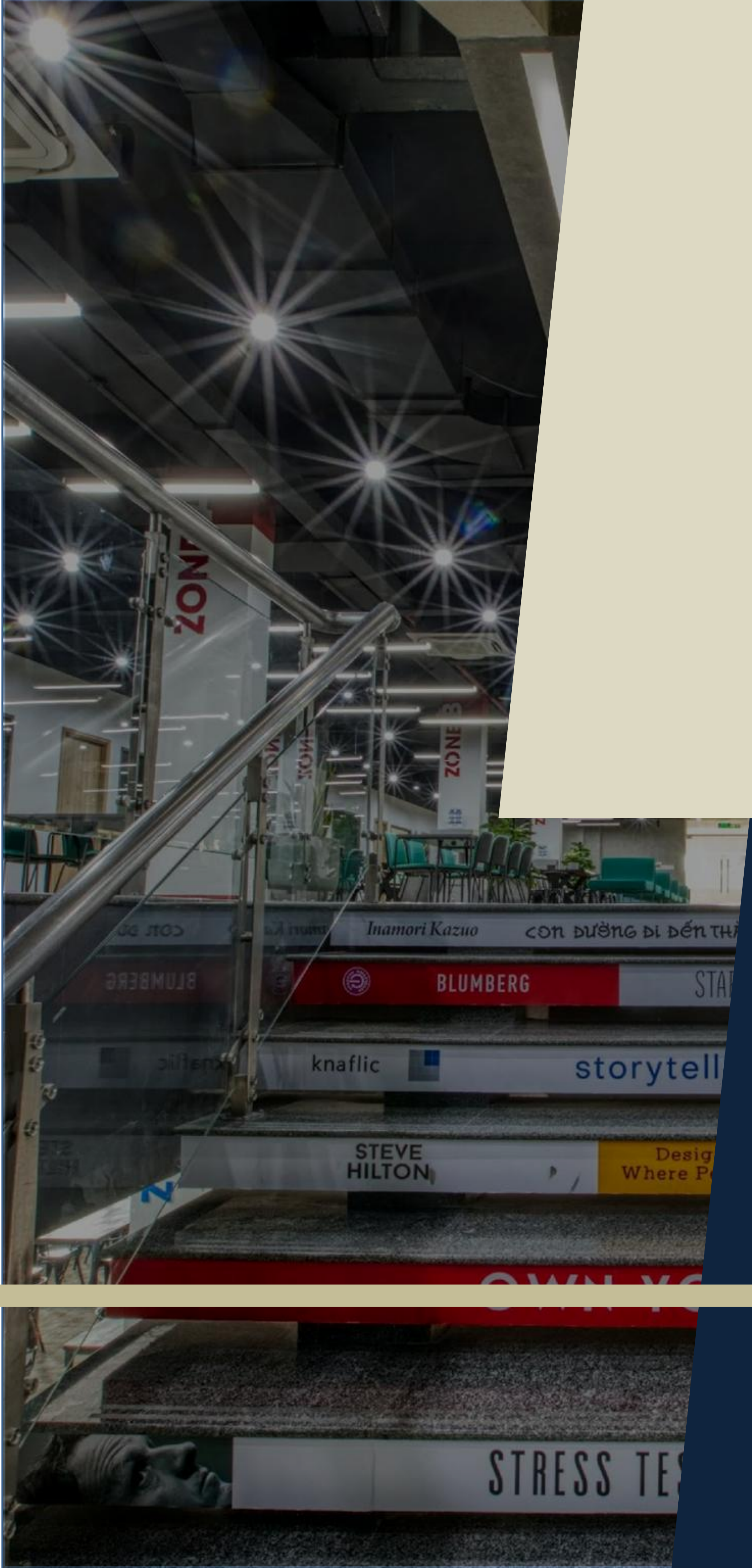
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY



Coding Practice

SOLID Principles

SOLID Principles

Contents

- What is SOLID
- Single responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

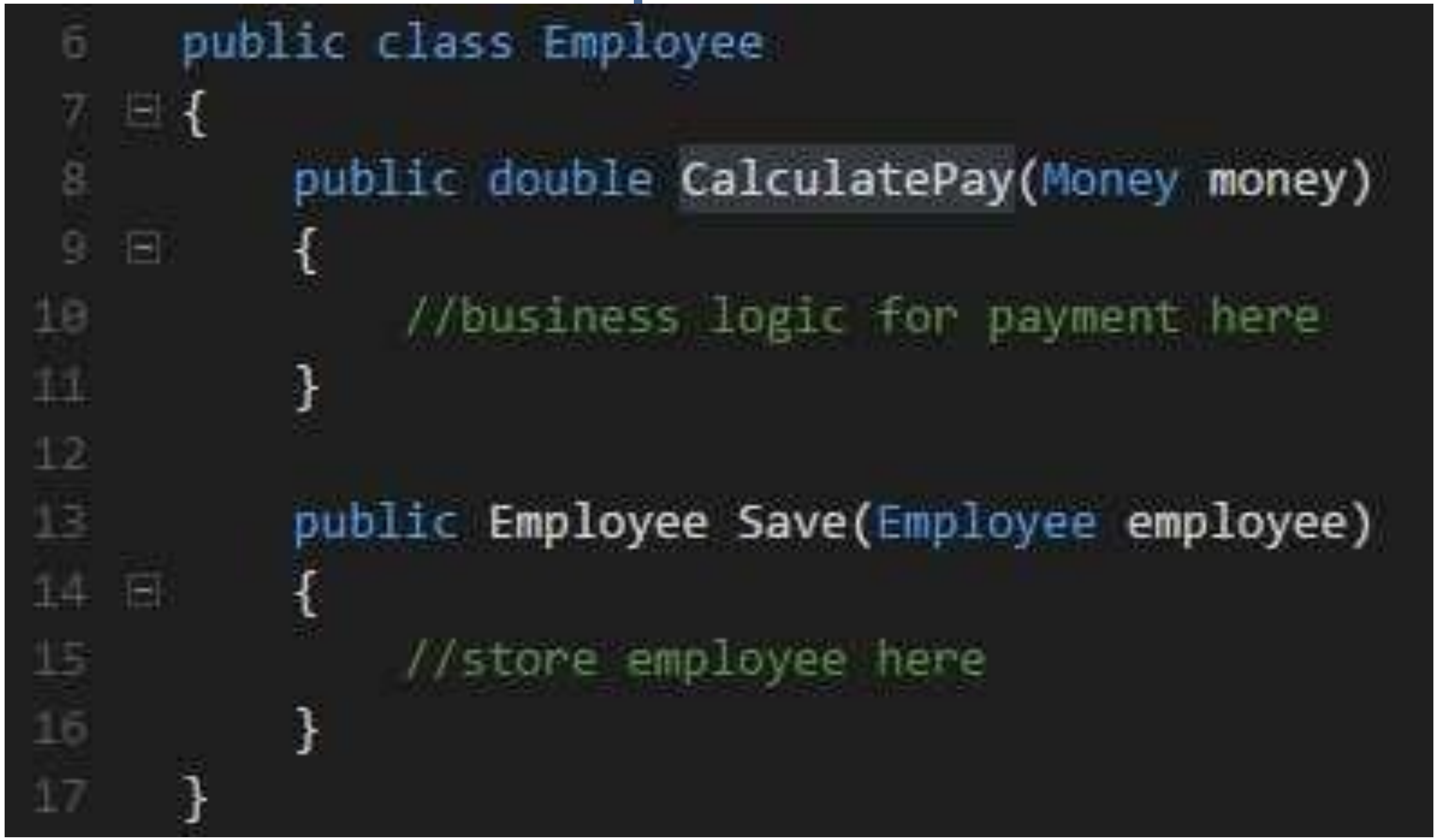
What is SOLID?

S	O	L	I	D
SRP	OCP	LSP	ISP	DIP
Single Responsibility Principle	Open/Closed Principle	Liskovs Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Single Responsibility Principle

“A class should have one and only one reason to change”

Single Responsibility Principle



```
public class Employee
{
    public double CalculatePay(Money money)
    {
        //business logic for payment here
    }

    public Employee Save(Employee employee)
    {
        //store employee here
    }
}
```

Business logic

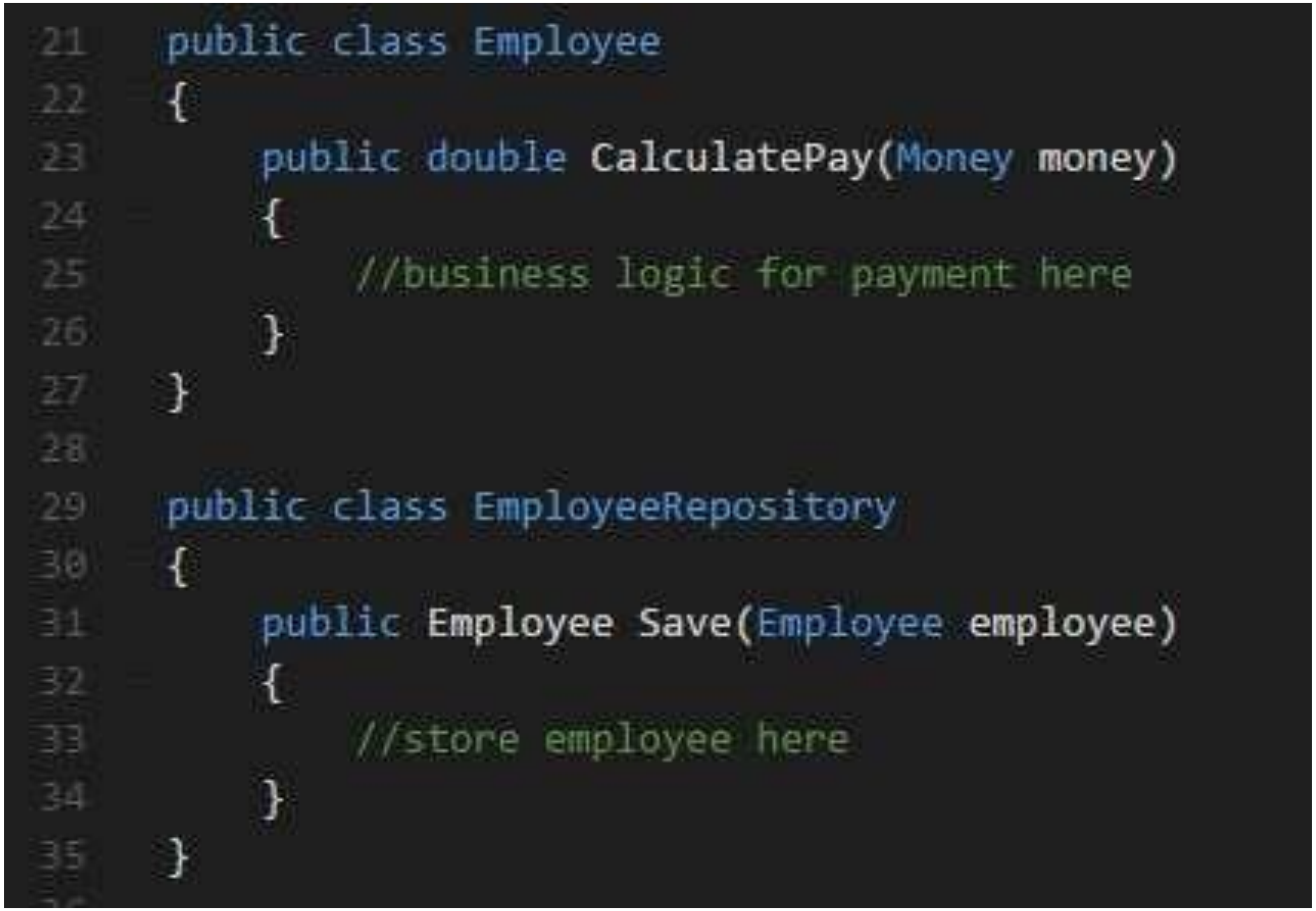
Persistence

There are two responsibilities

Single Responsibility Principle

How to solve this?

Single Responsibility Principle



```
public class Employee
{
    public double CalculatePay(Money money)
    {
        //business logic for payment here
    }
}

public class EmployeeRepository
{
    public Employee Save(Employee employee)
    {
        //store employee here
    }
}
```

Just create two different classes

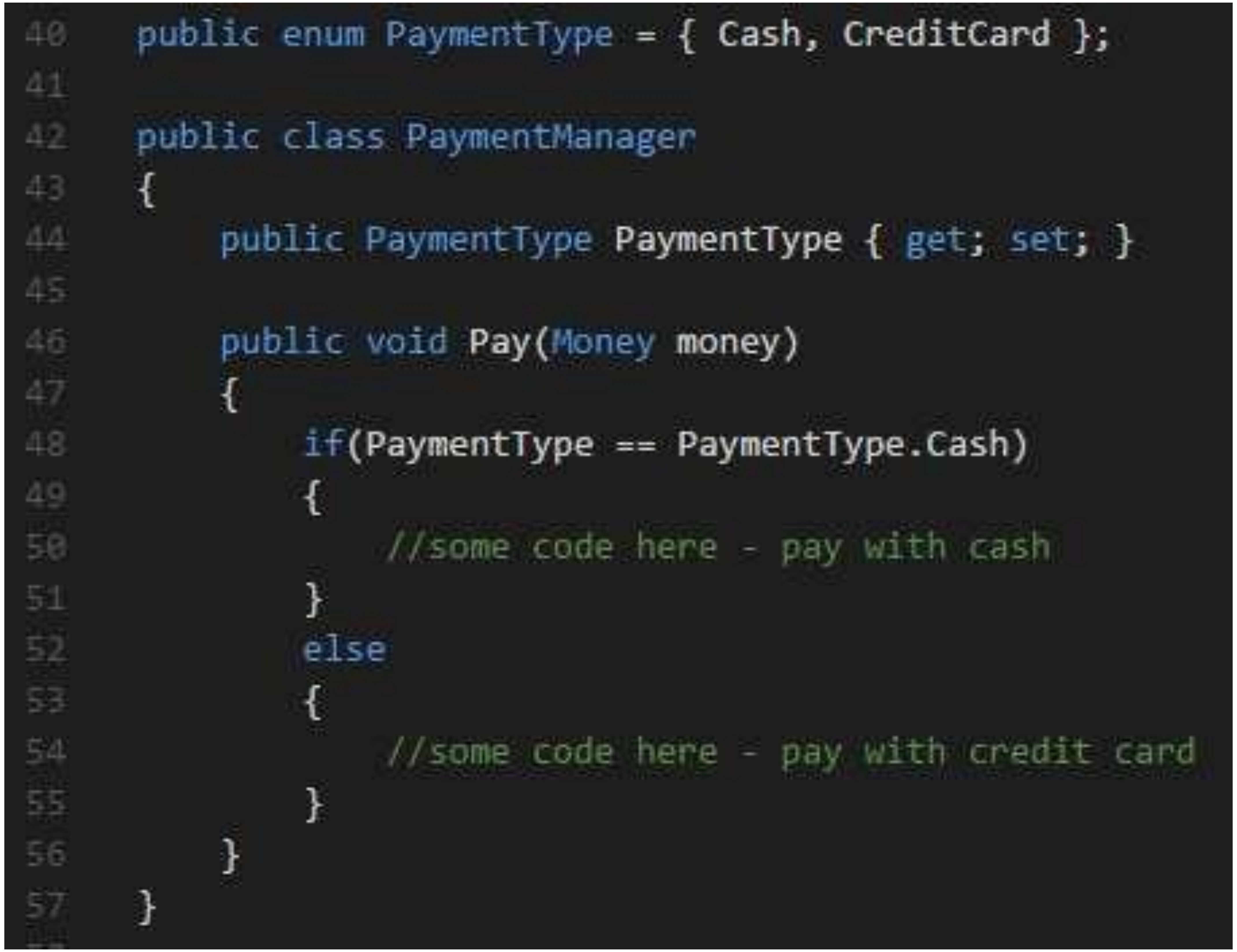
Open/Closed Principle

“Software entities should be ***open*** *for extension, but* ***closed*** *for modification.”*

Open/Closed Principle

- Increased stability – existing code (almost) never changes
- Increased modularity, but many small classes

Open/Closed Principle



```
public enum PaymentType = { Cash, CreditCard };

public class PaymentManager
{
    public PaymentType PaymentType { get; set; }

    public void Pay(Money money)
    {
        if(PaymentType == PaymentType.Cash)
        {
            //some code here - pay with cash
        }
        else
        {
            //some code here - pay with credit card
        }
    }
}
```

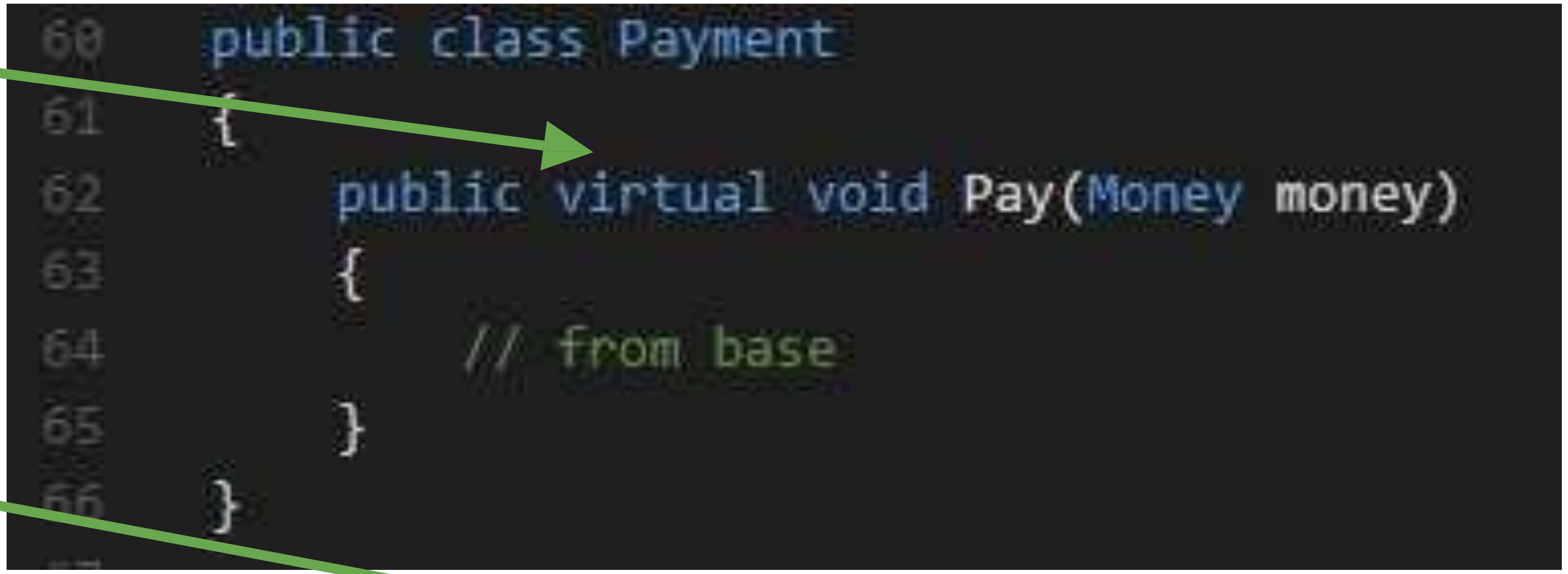
Humm...and if I need to add a new payment type?

You need to modificate this class.

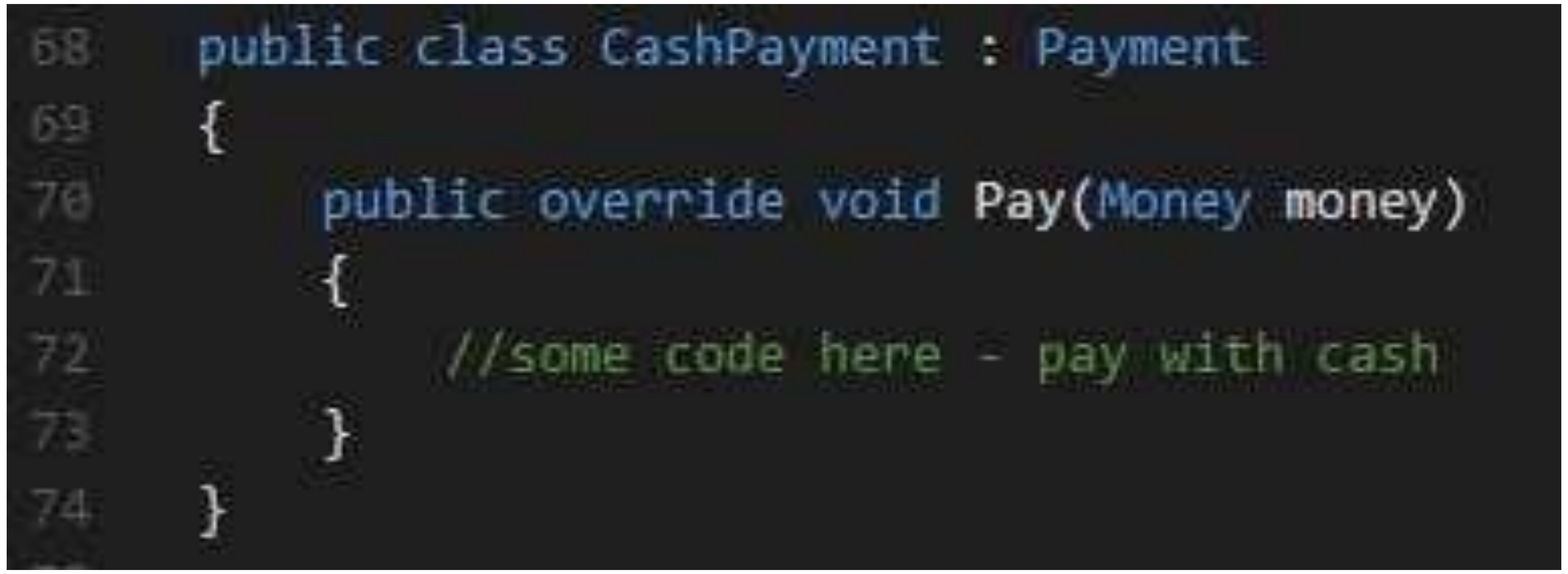
Open/Closed Principle

open for extension

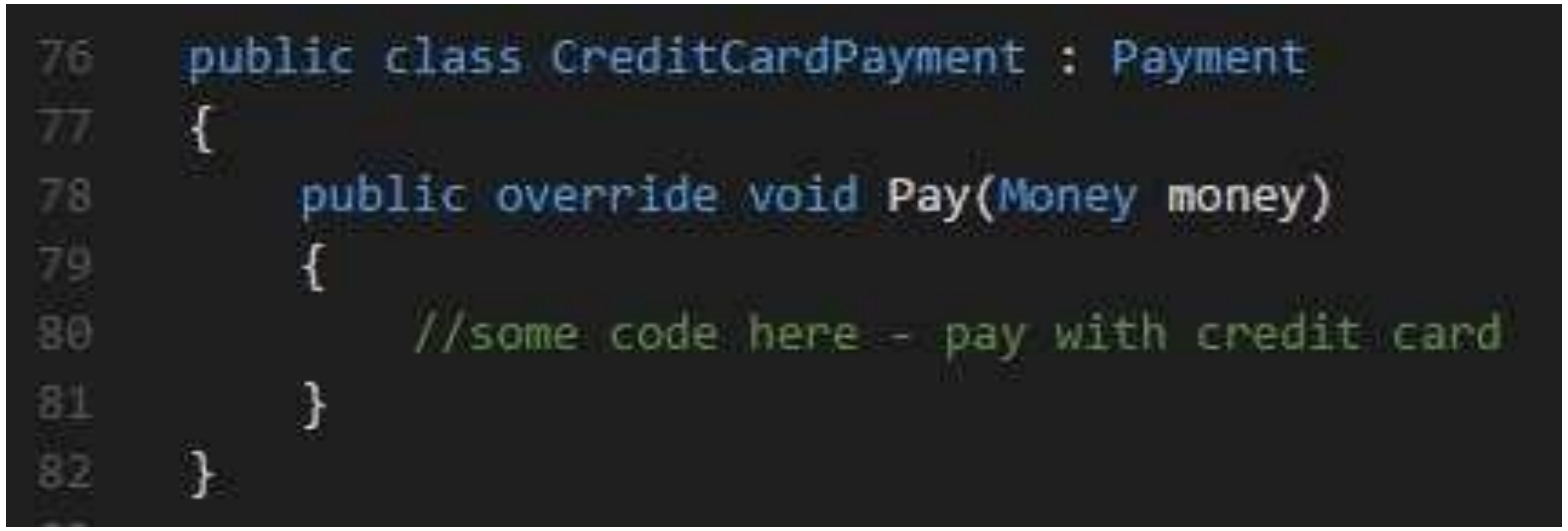
close for modification



```
public class Payment
{
    public virtual void Pay(Money money)
    {
        // from base
    }
}
```



```
public class CashPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with cash
    }
}
```



```
public class CreditCardPayment : Payment
{
    public override void Pay(Money money)
    {
        //some code here - pay with credit card
    }
}
```

Liskov Substitution Principle

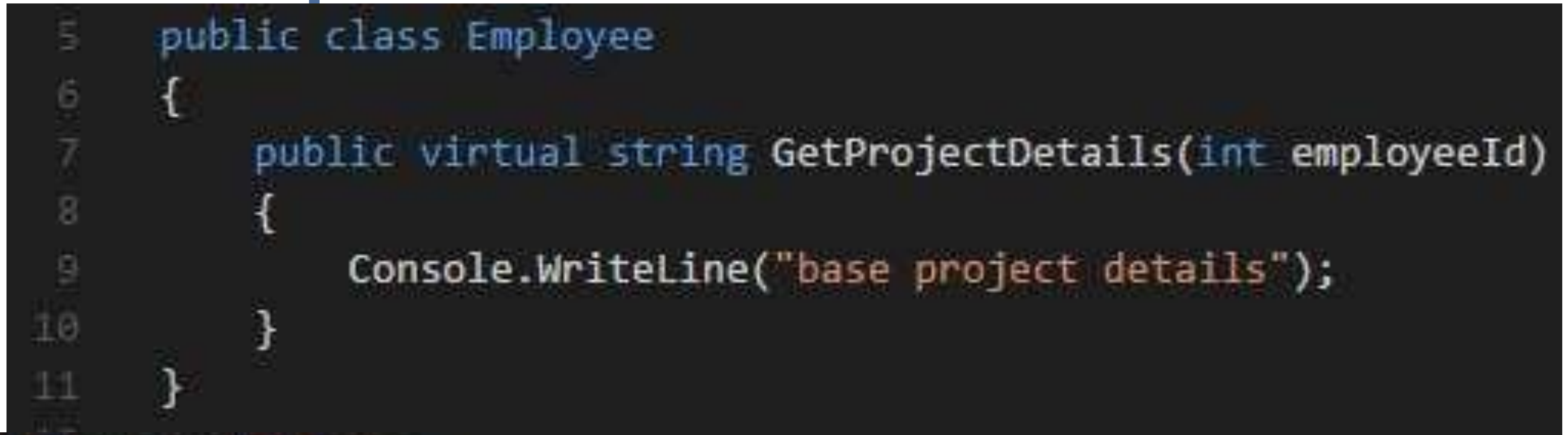
“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T”

What do you say?

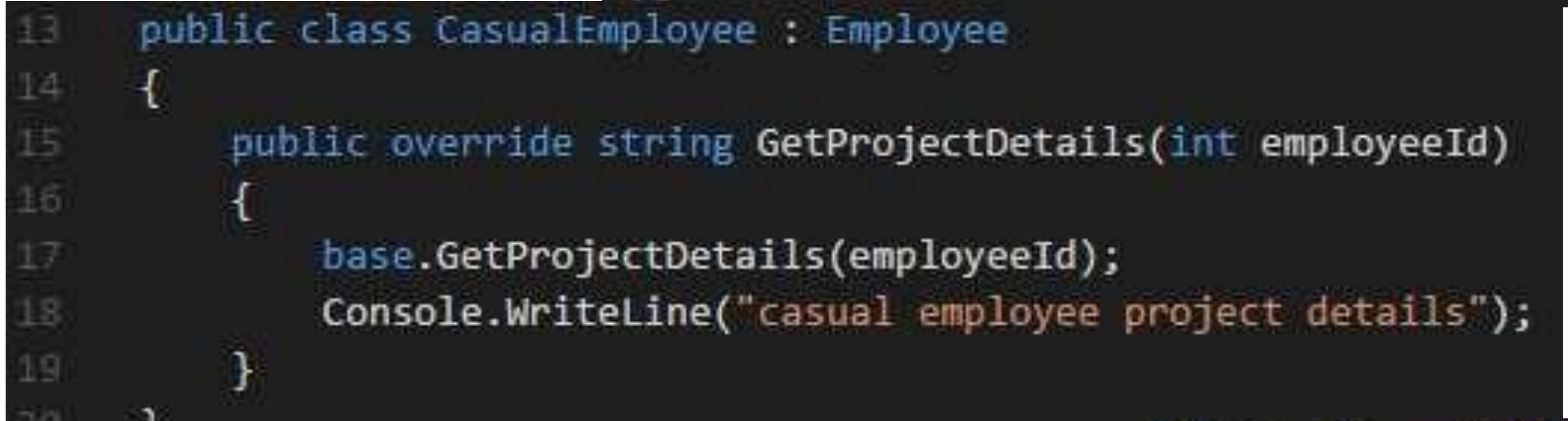
Liskov Substitution Principle

“A subclass should behave in such a way that it will not cause problems when used instead of the superclass.”

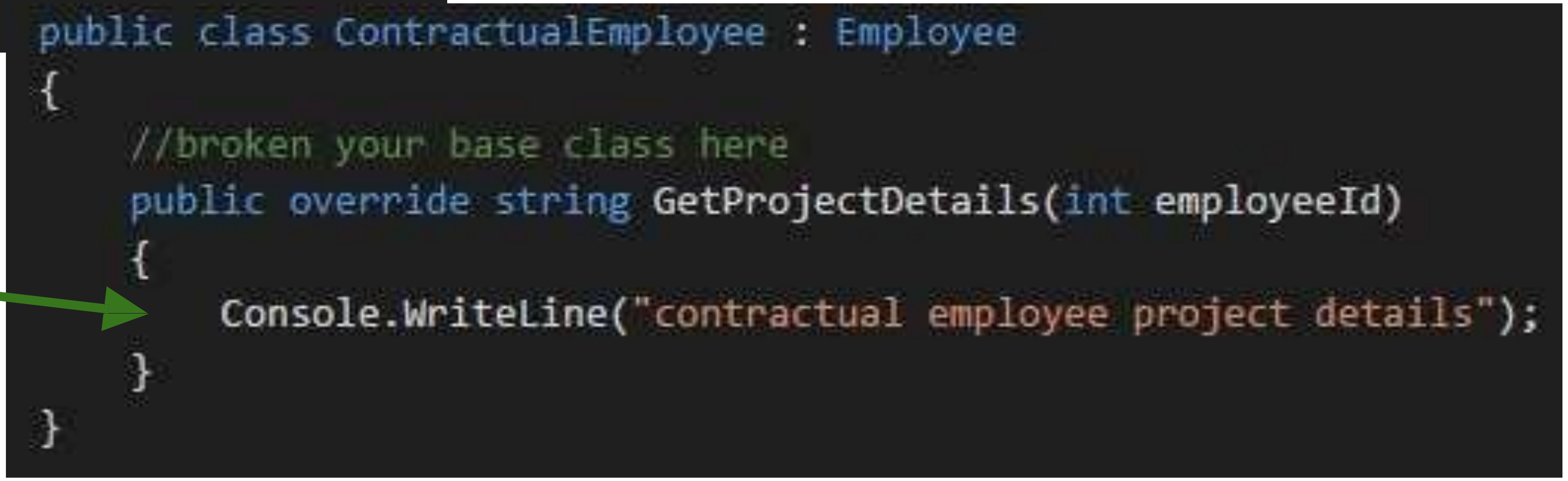
Liskov Substitution Principle



```
public class Employee
{
    public virtual string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("base project details");
    }
}
```

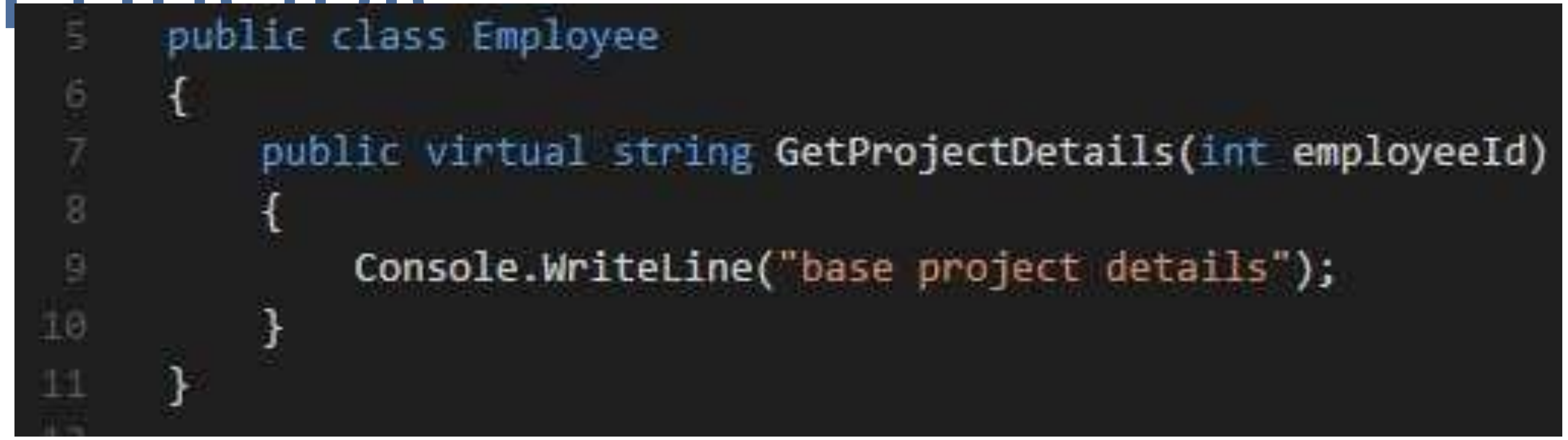


```
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("casual employee project details");
    }
}
```

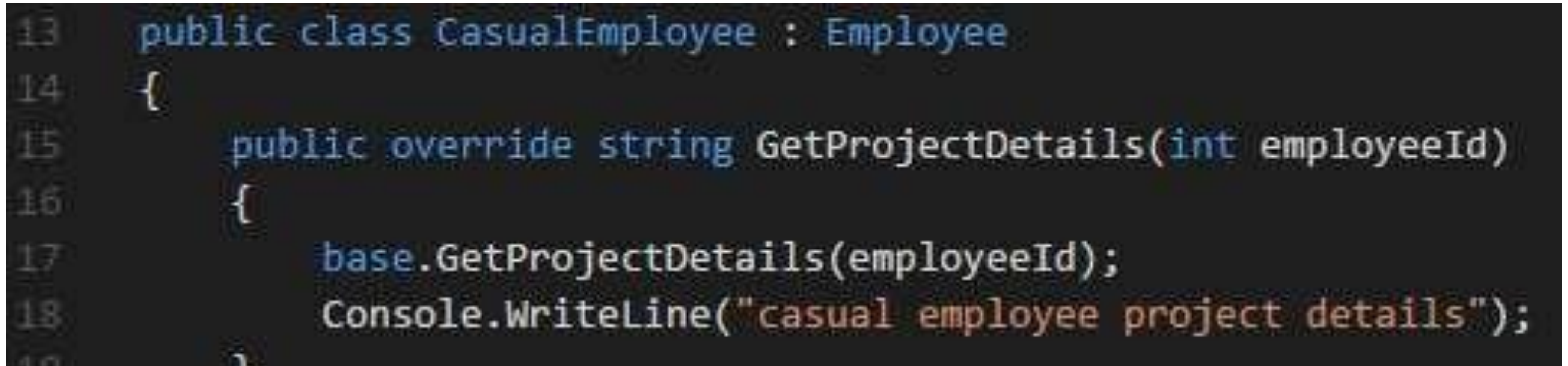


```
public class ContractualEmployee : Employee
{
    //broken your base class here
    public override string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("contractual employee project details");
    }
}
```

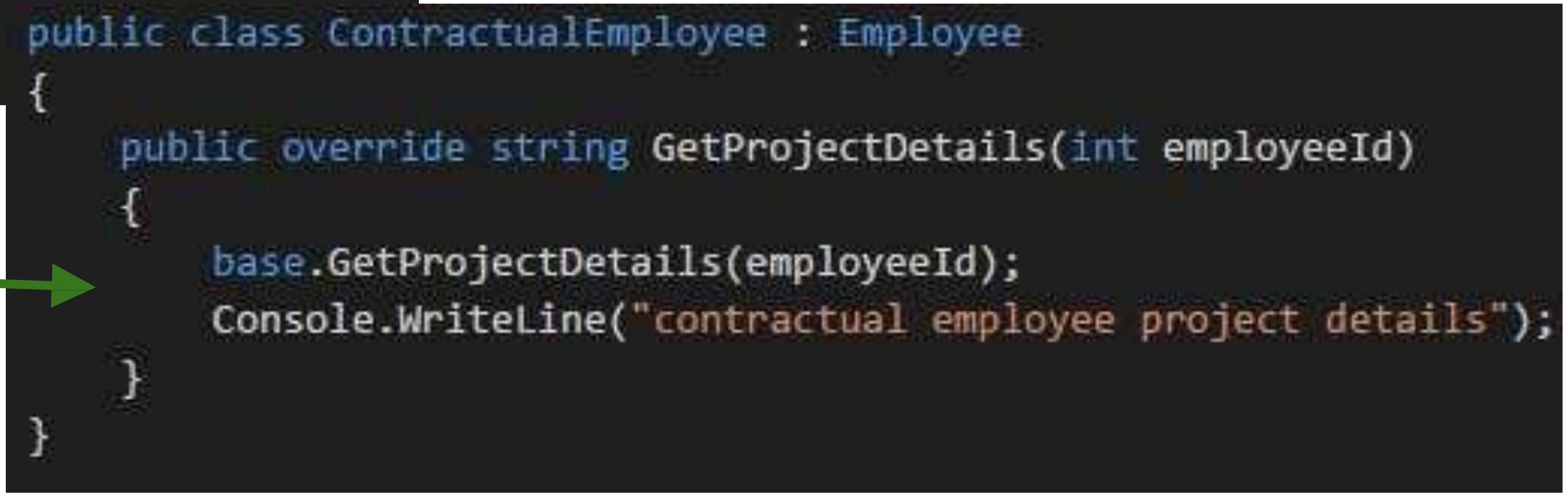
Liskov Substitution Principle



```
public class Employee
{
    public virtual string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("base project details");
    }
}
```



```
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("casual employee project details");
    }
}
```



```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("contractual employee project details");
    }
}
```

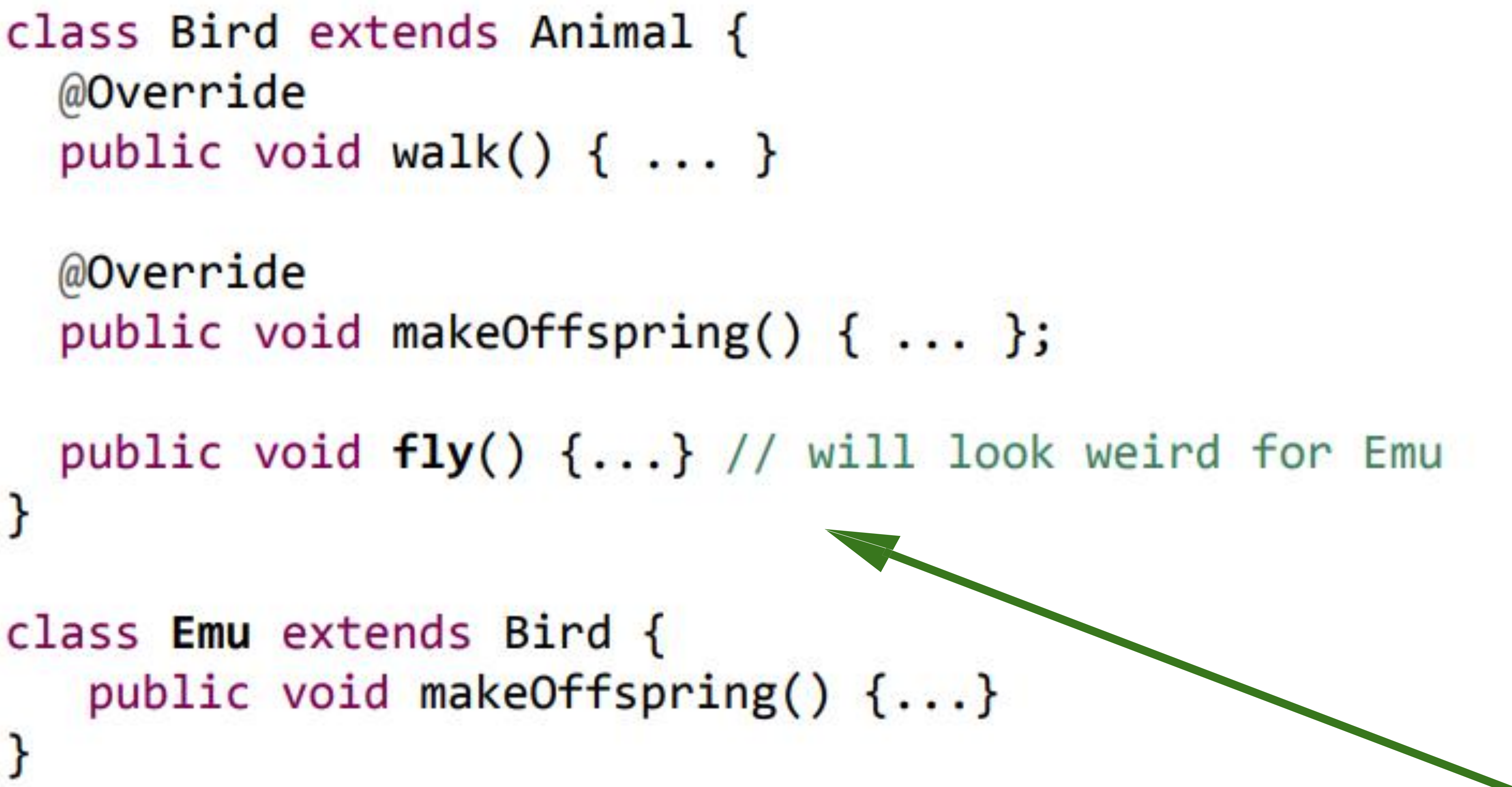
Much better

Liskov Substitution Principle

Major examples of LSP violation:

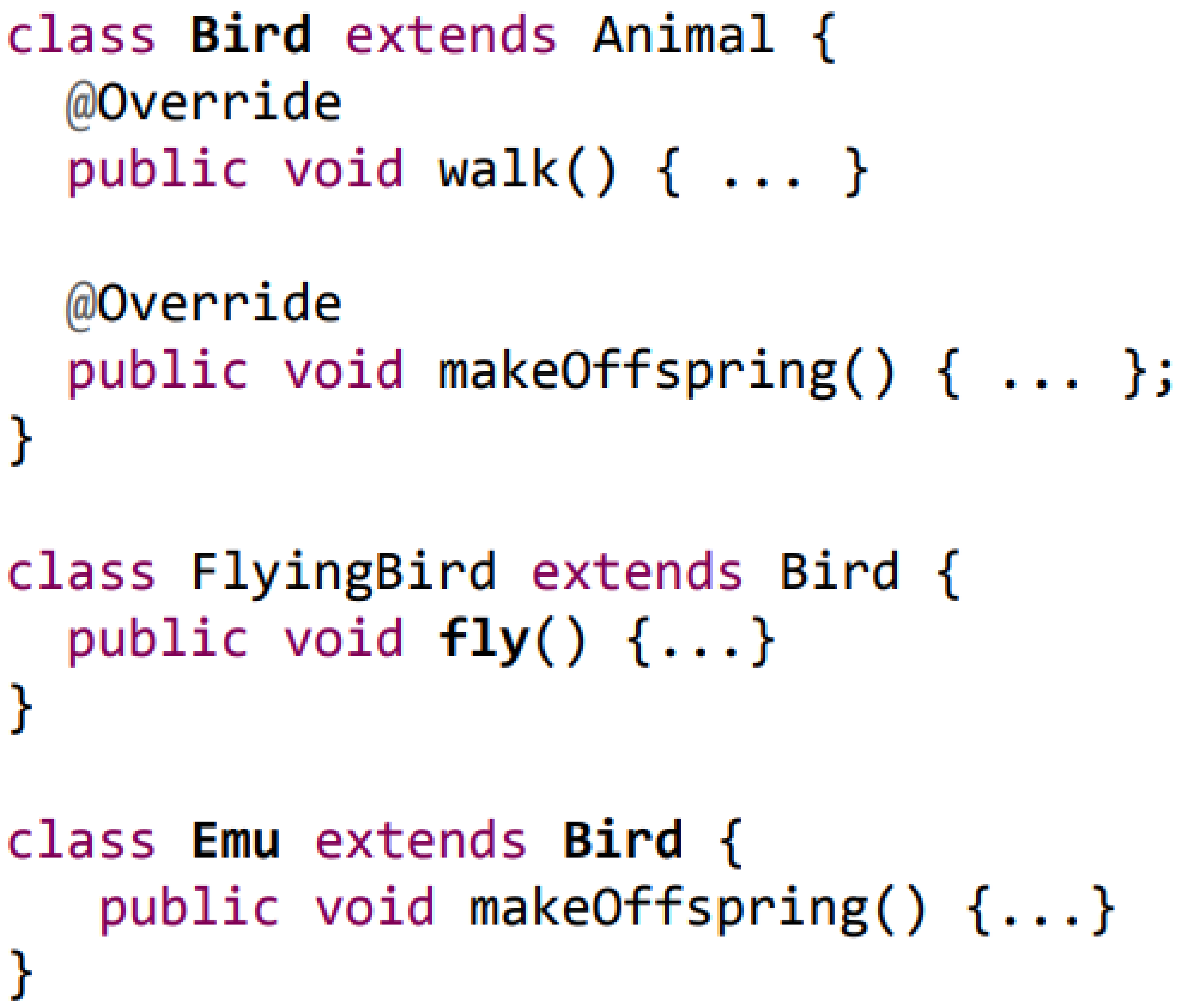
1. Sub-class implements only some methods, other look redundant and … weird
2. Some methods behavior violates contract
3. equals() method symmetry requirement is violated
4. Subclass throws exception which are not declared by parent class/interface (java prevents from introducing checked exceptions)

Liskov Substitution Principle



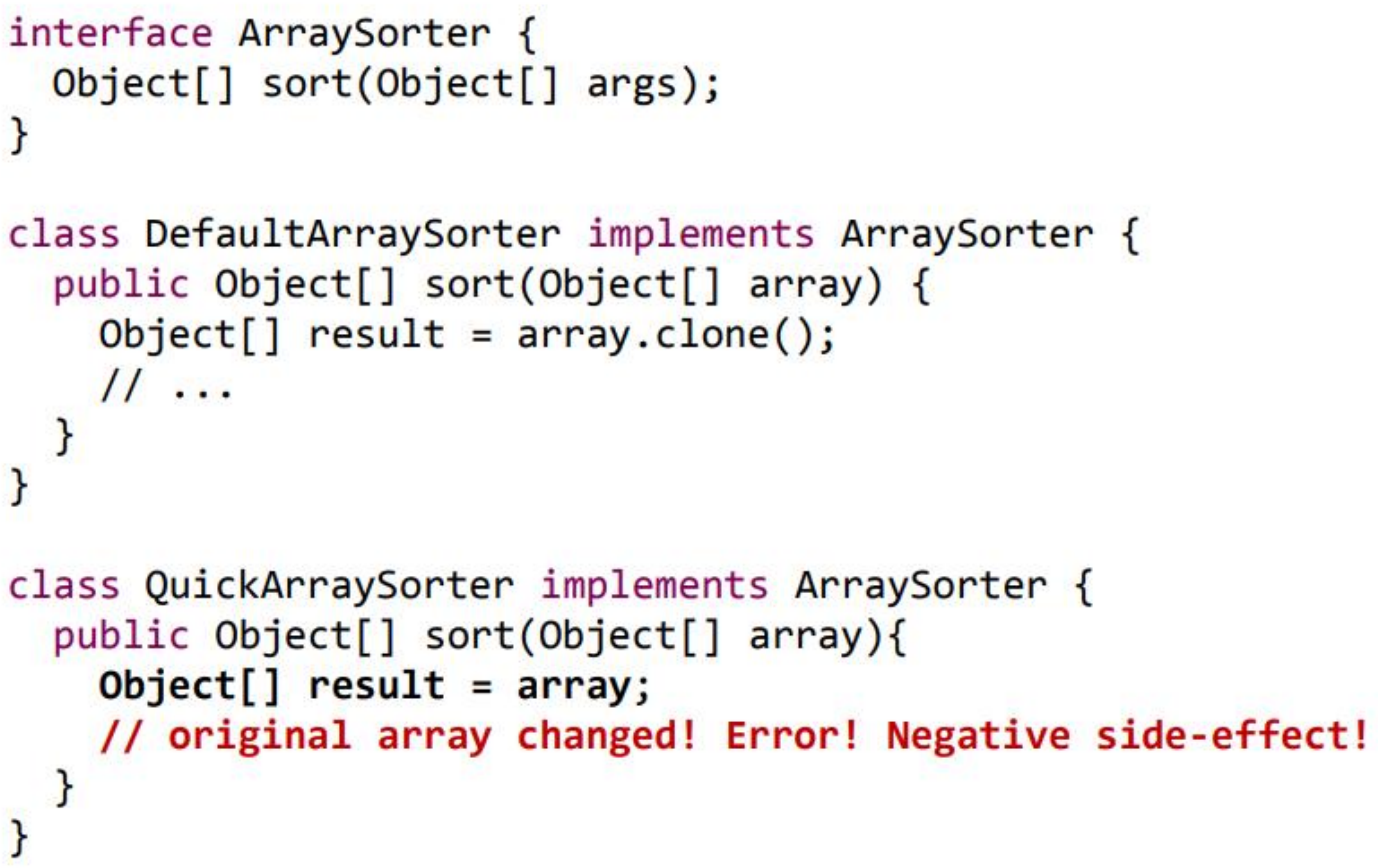
```
class Bird extends Animal {
  @Override
  public void walk() { ... }

  @Override
  public void makeOffspring() { ... };

  public void fly() {...} // will look weird for Emu
}

class Emu extends Bird {
   public void makeOffspring() {...}
}
```

Liskov Substitution Principle



```
class Bird extends Animal {
  @Override
  public void walk() { ... }

  @Override
  public void makeOffspring() { ... };
}

class FlyingBird extends Bird {
  public void fly() {...}
}

class Emu extends Bird {
   public void makeOffspring() {...}
}
```

Liskov Substitution Principle

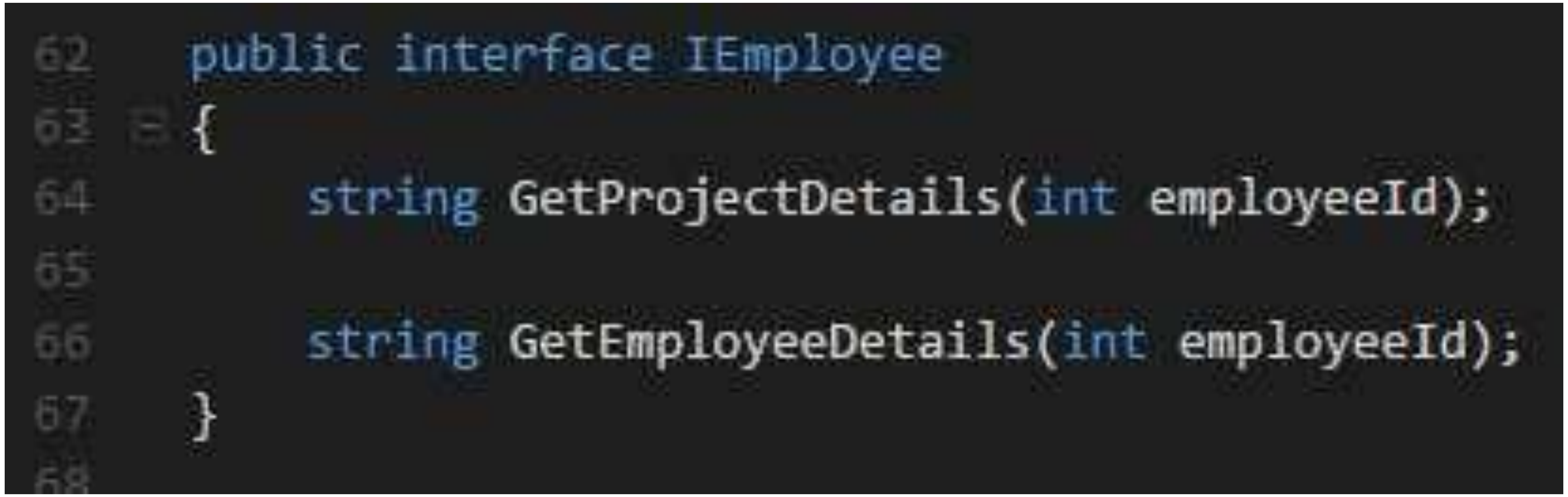


```
interface ArraySorter {
  Object[] sort(Object[] args);
}

class DefaultArraySorter implements ArraySorter {
  public Object[] sort(Object[] array) {
    Object[] result = array.clone();
    // ...
  }
}

class QuickArraySorter implements ArraySorter {
  public Object[] sort(Object[] array){
    Object[] result = array;
    // original array changed! Error! Negative side-effect!
  }
}
```

Interface Segregation Principle

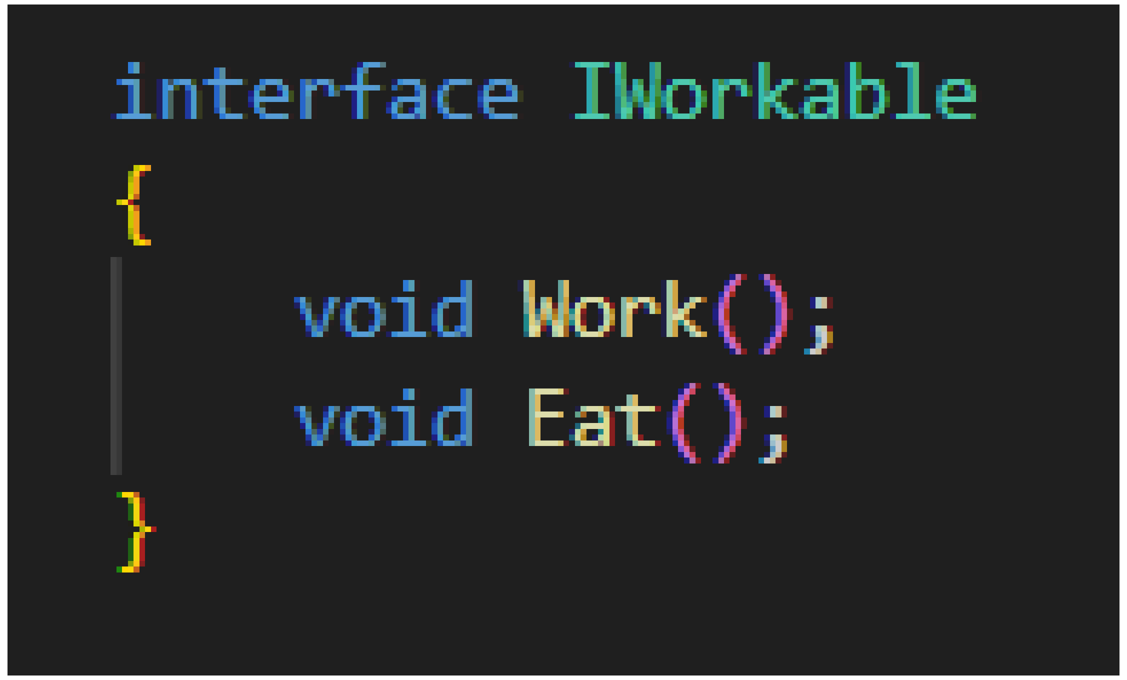
“Clients should not be forced to depend upon interfaces that they don't use”

Interface Segregation Principle

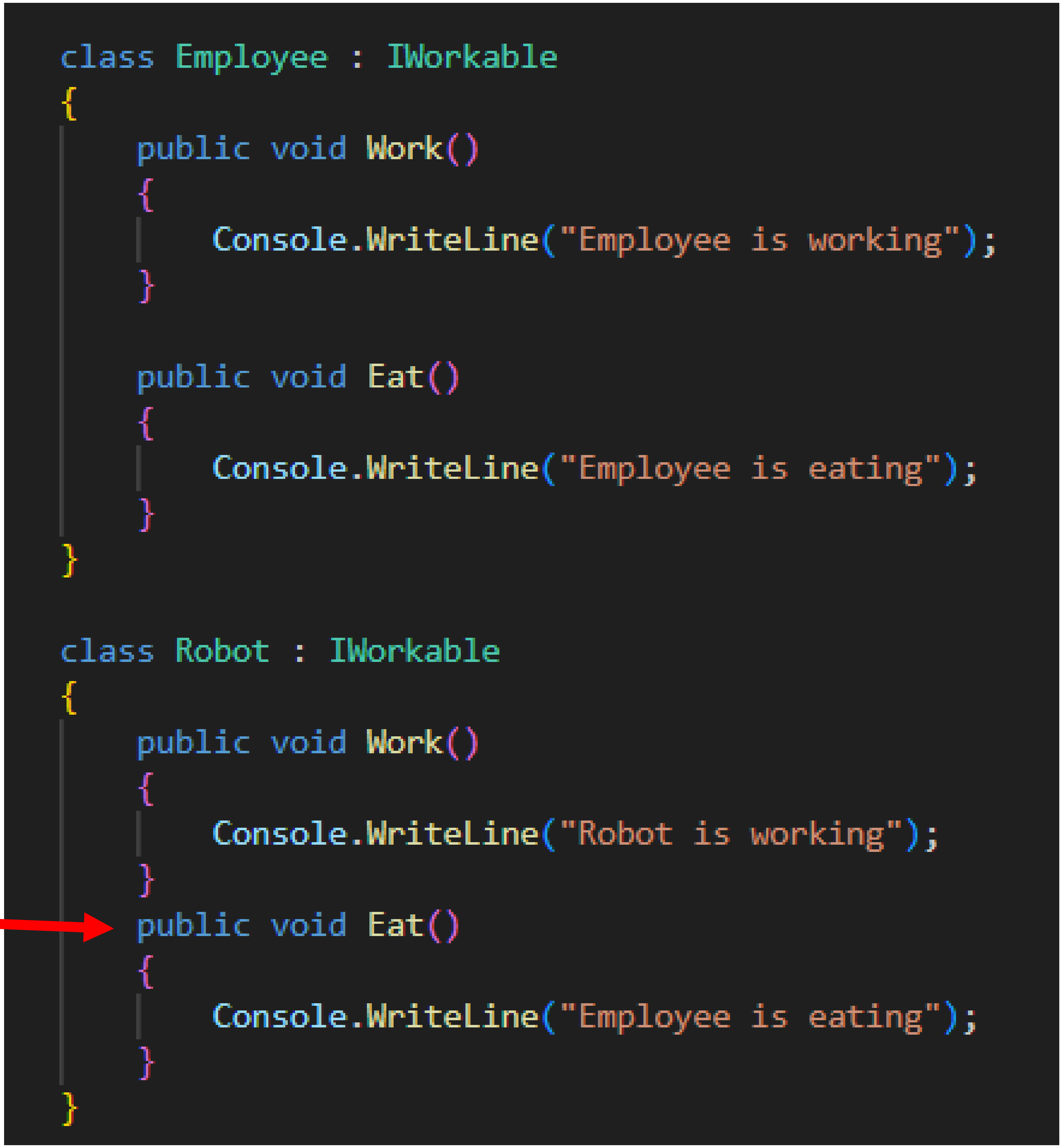


```
public interface IEmployee
{
    string GetProjectDetails(int employeeId);

    string GetEmployeeDetails(int employeeId);
}
```

Interface Segregation Principle



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



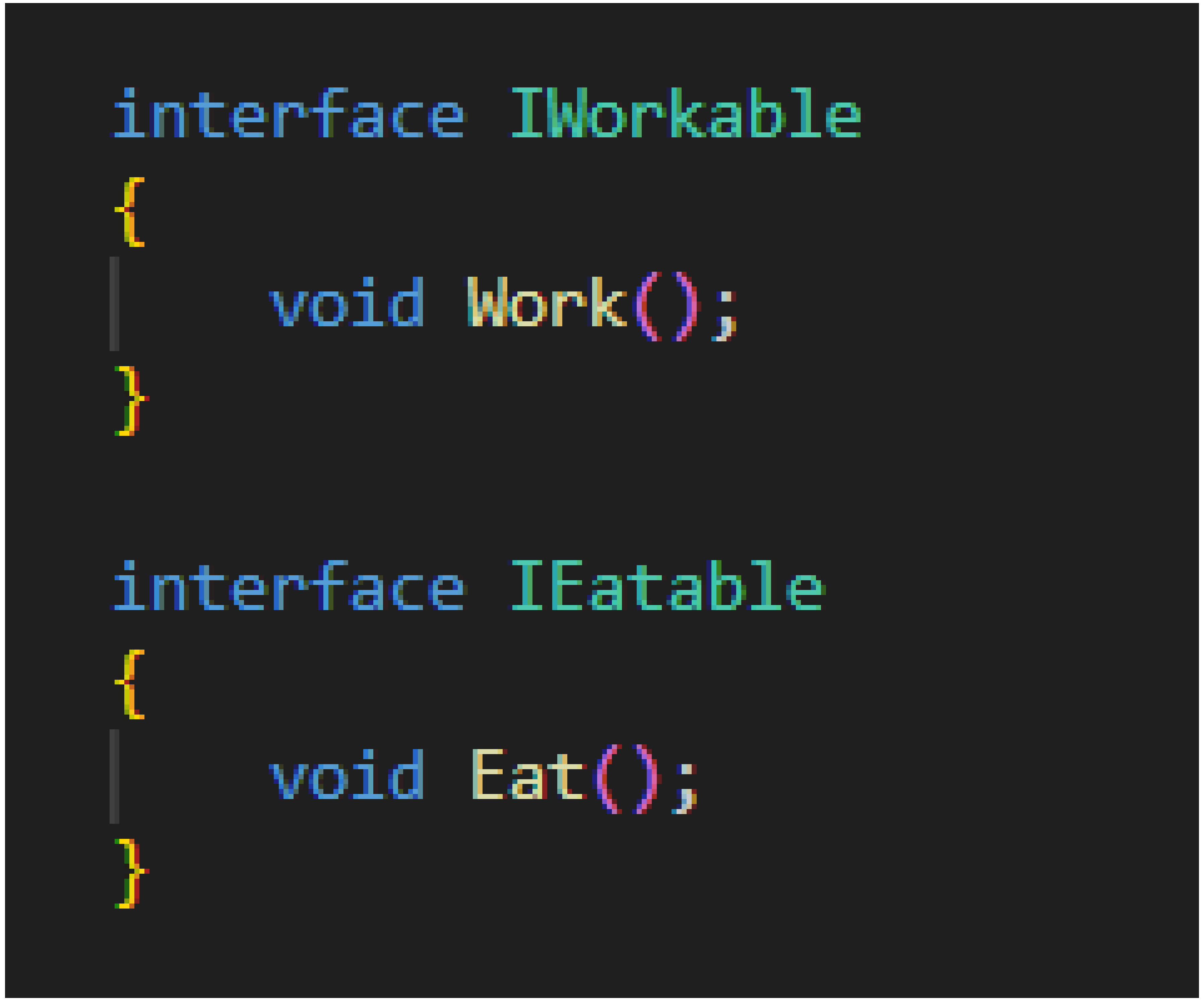
```
class Employee : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Employee is working");
    }

    public void Eat()
    {
        Console.WriteLine("Employee is eating");
    }
}

class Robot : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Robot is working");
    }
    public void Eat()
    {
        Console.WriteLine("Employee is eating");
    }
}
```

WHY?????
I don’t need you!!

Interface Segregation Principle

How to solve this?

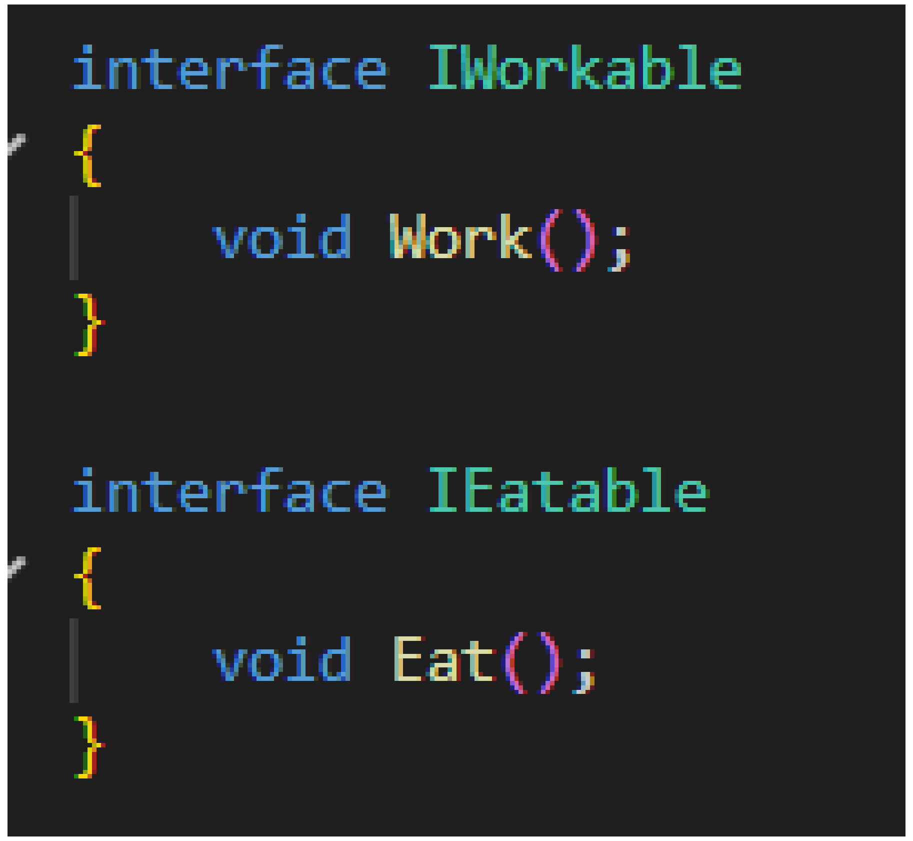
Interface Segregation Principle



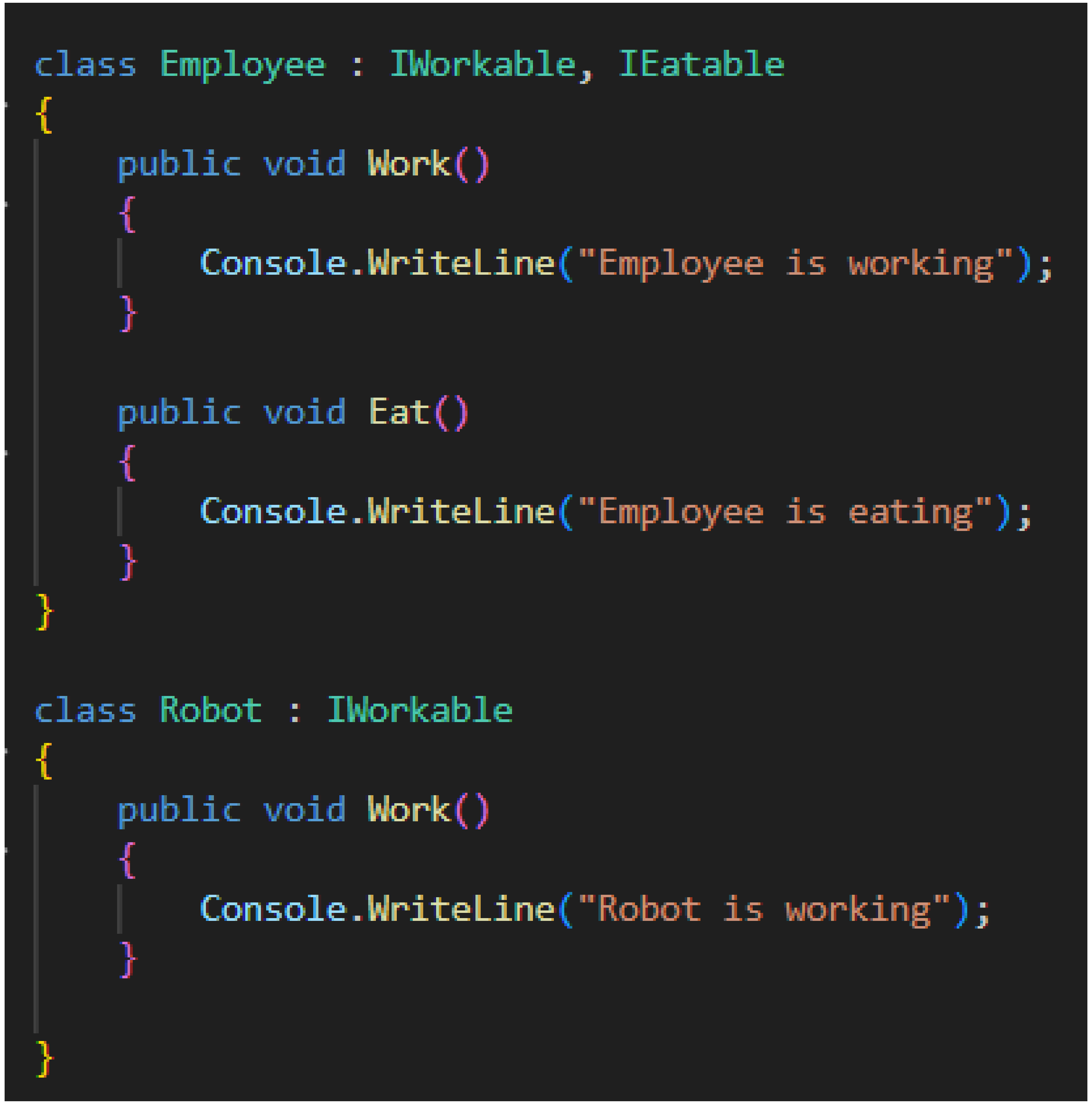
```
interface IWorkable
{
    void Work();
}

interface IEatable
{
    void Eat();
}
```

You need to create two interfaces

Interface Segregation Principle



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

interface IEatable
{
    void Eat();
}
```



```
class Employee : IWorkable, IEatable
{
    public void Work()
    {
        Console.WriteLine("Employee is working");
    }

    public void Eat()
    {
        Console.WriteLine("Employee is eating");
    }
}

class Robot : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Robot is working");
    }

}
```

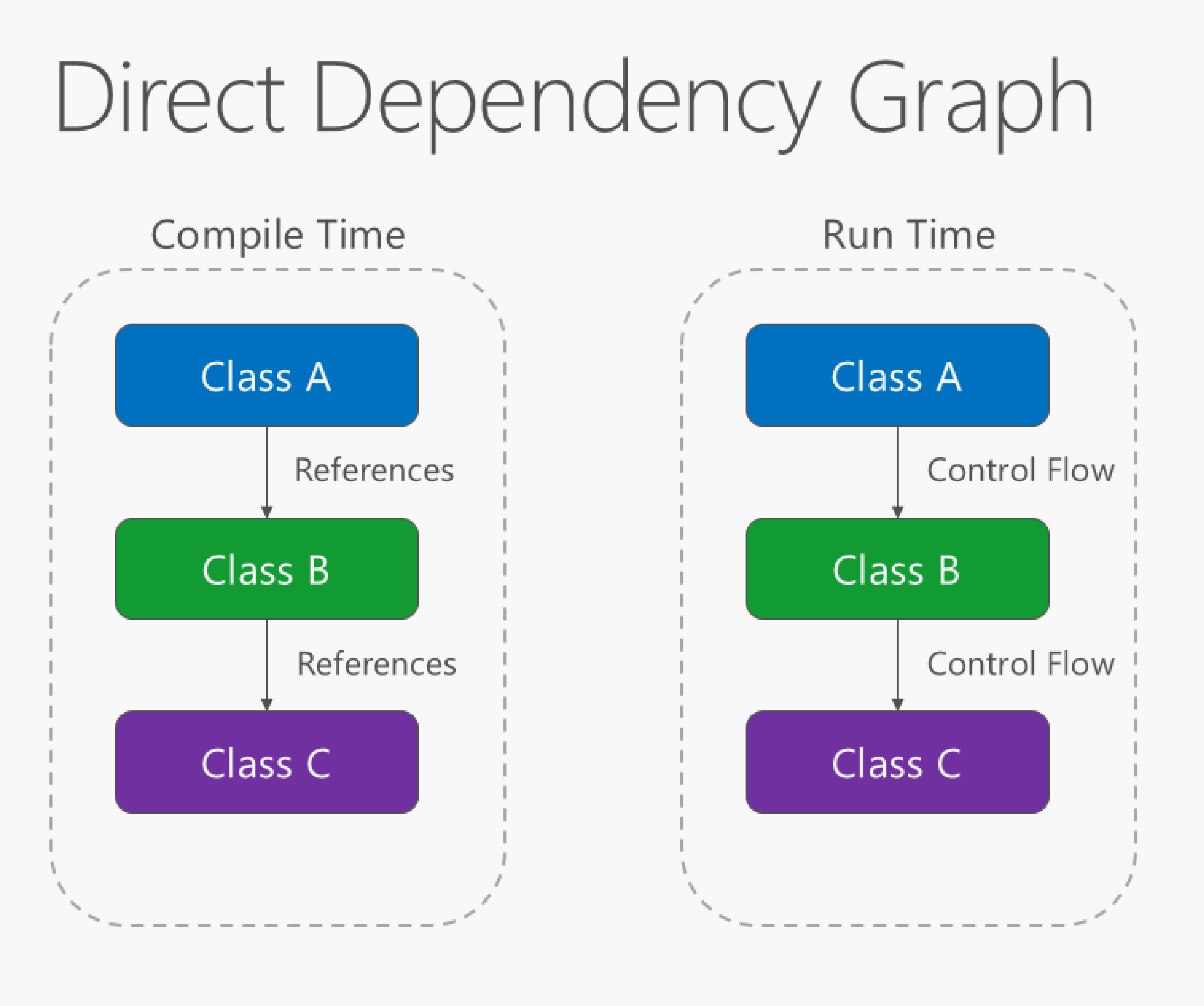
Dependency Inversion Principle

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Direct dependency

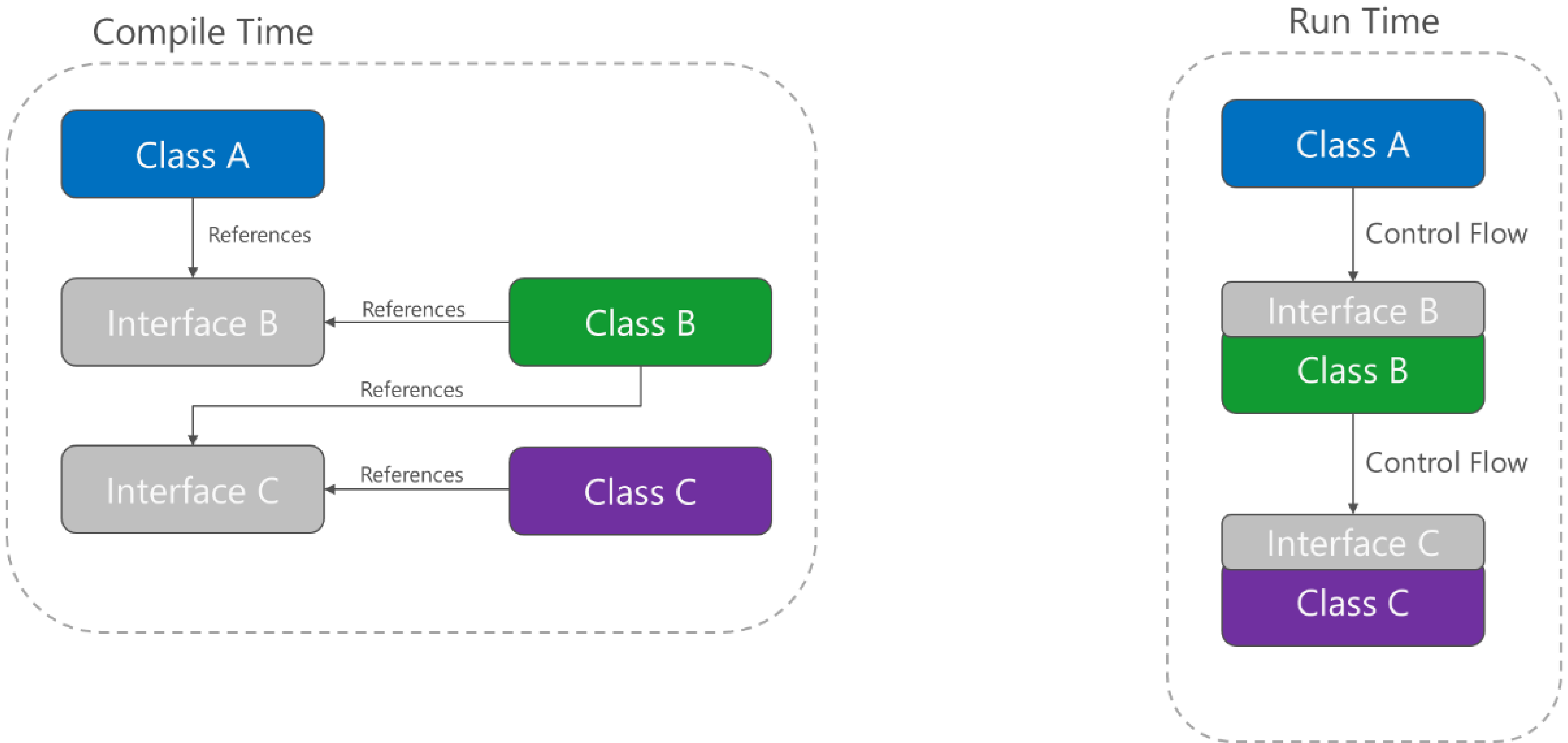
If class A calls a method of class B and class B calls a method of class C, then at compile time class A will depend on class and class B will depend on class C



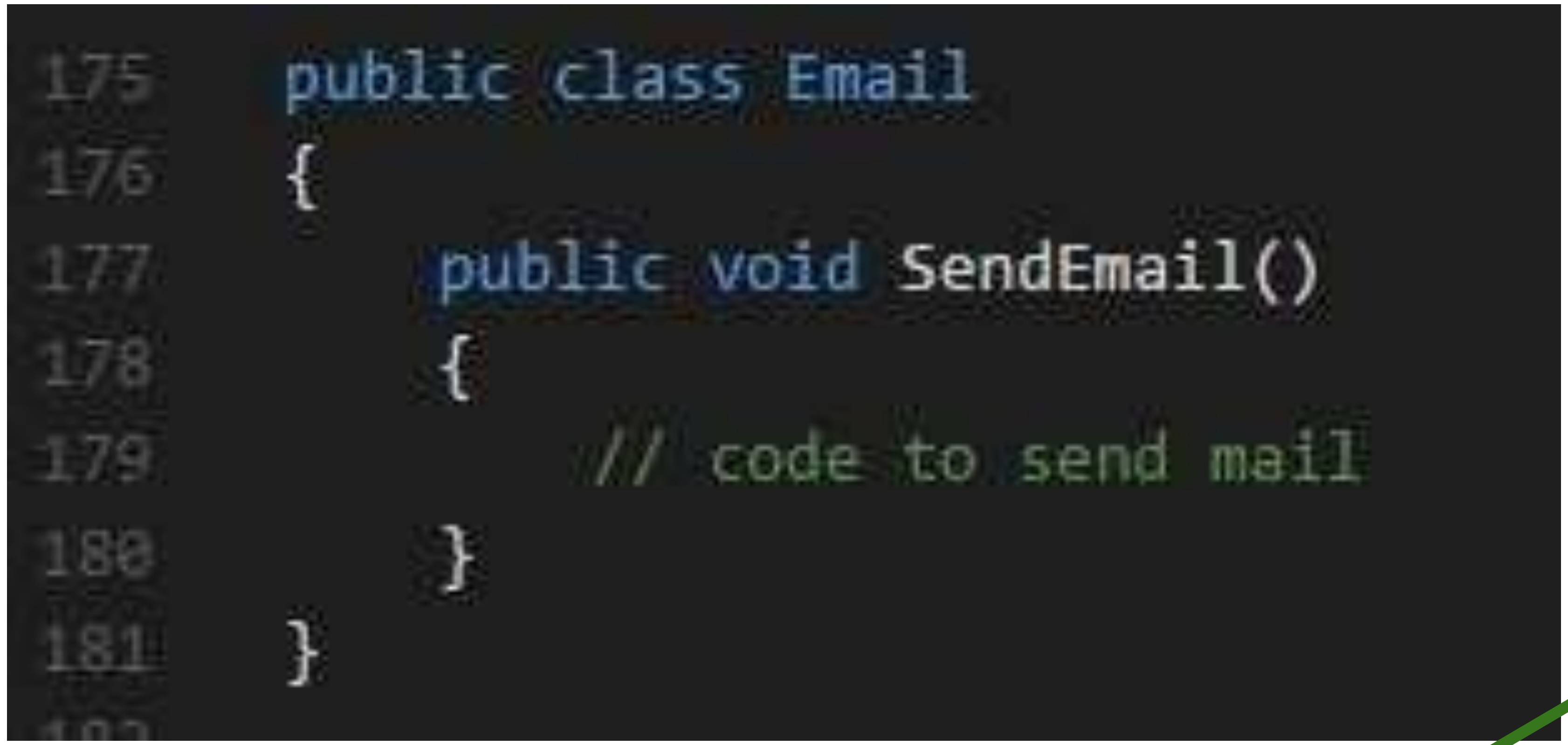
Dependency Inversion Principle

Applying the dependency inversion principle allows A to call methods on an abstraction that B implements, making it possible for A to call B at run time, but for B to depend on an interface controlled by A at compile time (thus, inverting the typical compile-time dependency). **Different implementations of these interfaces can easily be plugged in.**

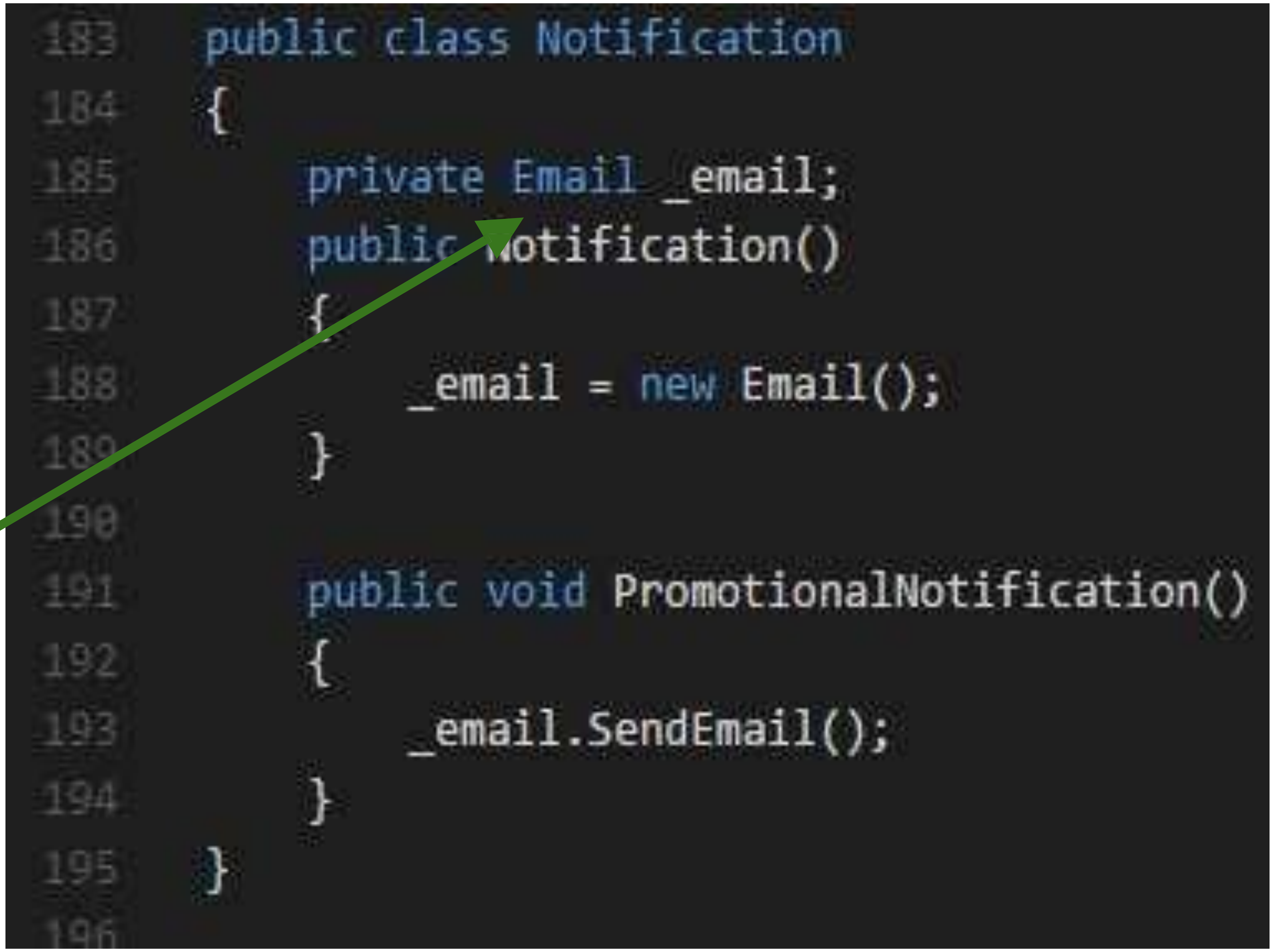
Inverted Dependency Graph



Dependency Inversion Principle



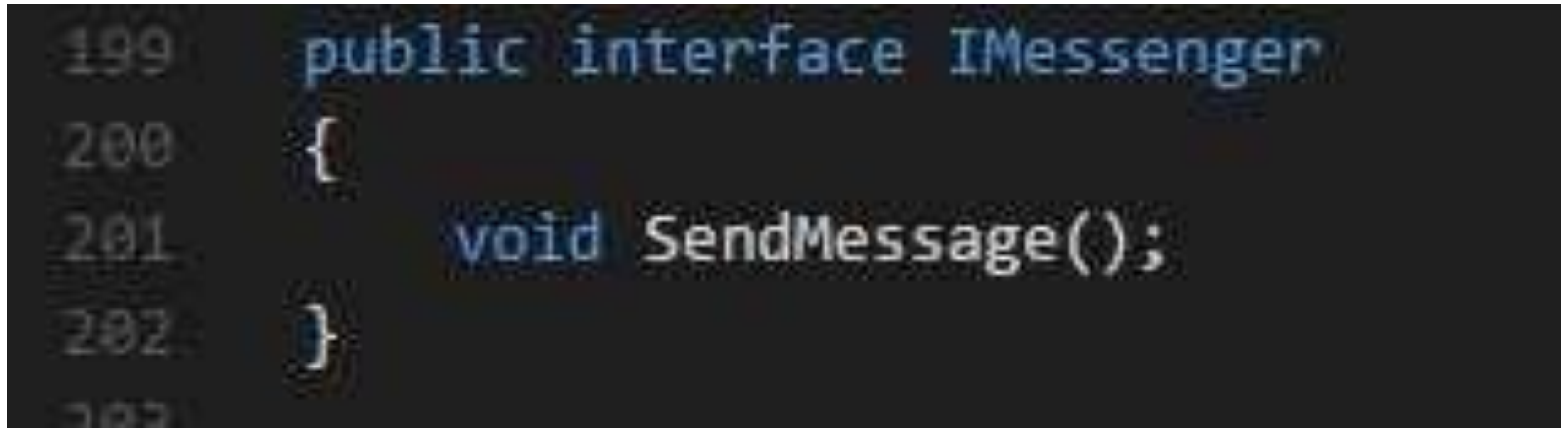
```
public class Email
{
    public void SendEmail()
    {
        // code to send mail
    }
}
```



```
public class Notification
{
    private Email _email;
    public Notification()
    {
        _email = new Email();
    }

    public void PromotionalNotification()
    {
        _email.SendEmail();
    }
}
```

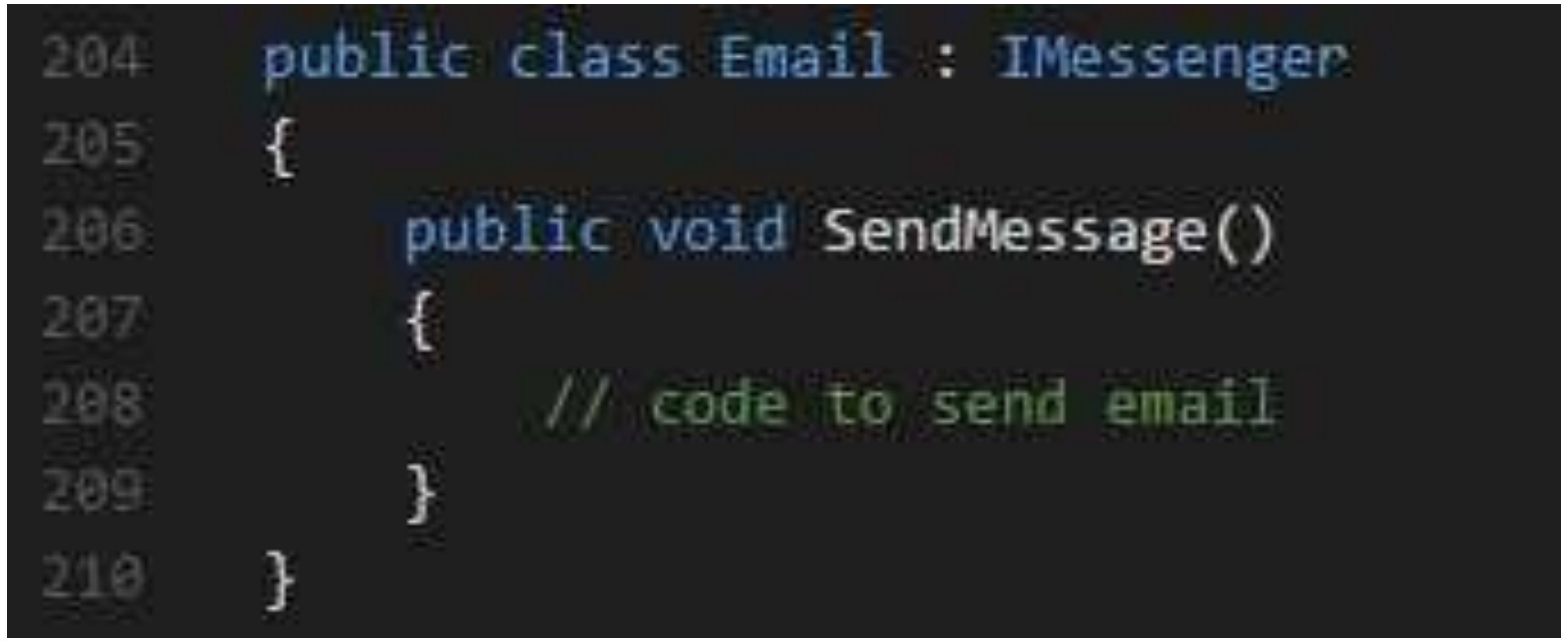
And if I need to send a
notification by SMS?
You need to change this.

Dependency Inversion Principle

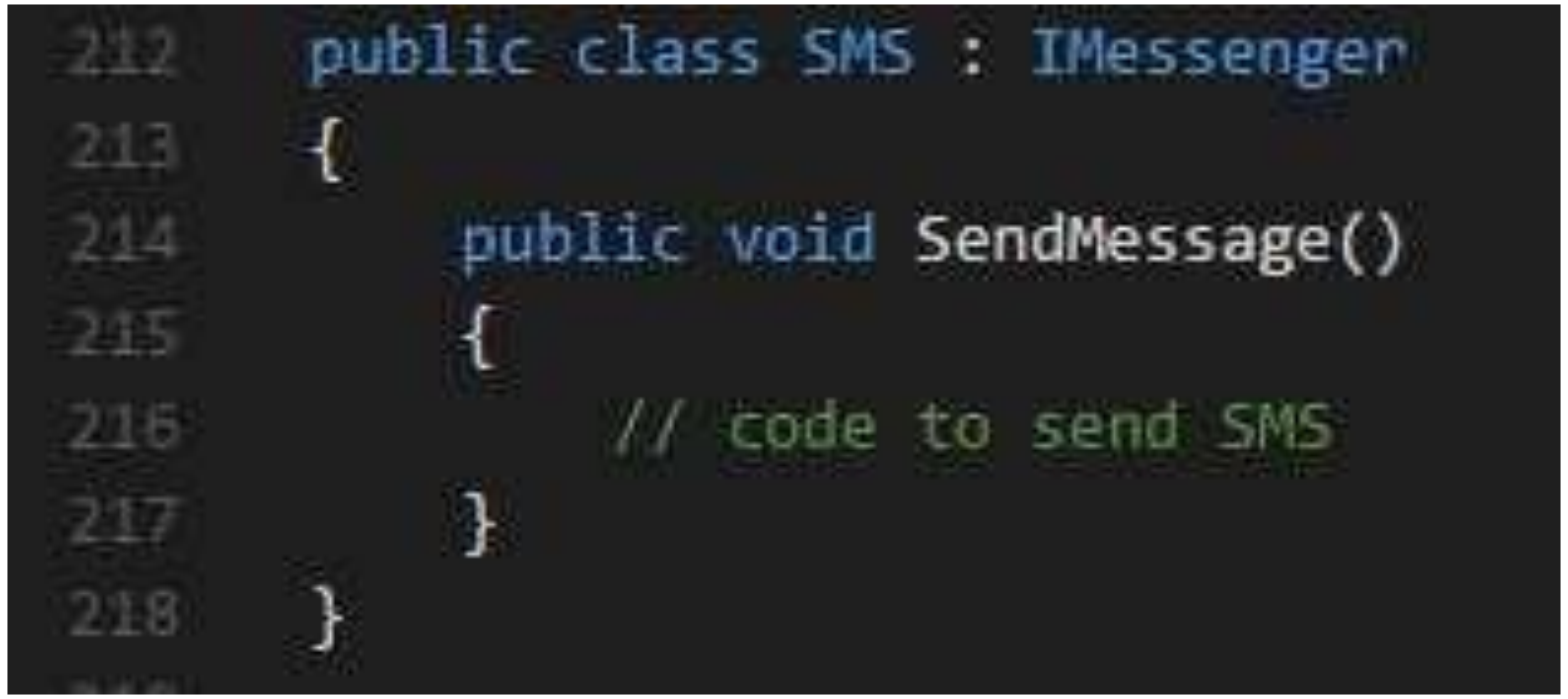


```
public interface IMessenger
{
    void SendMessage();
}
```

So, I create an interface and now?

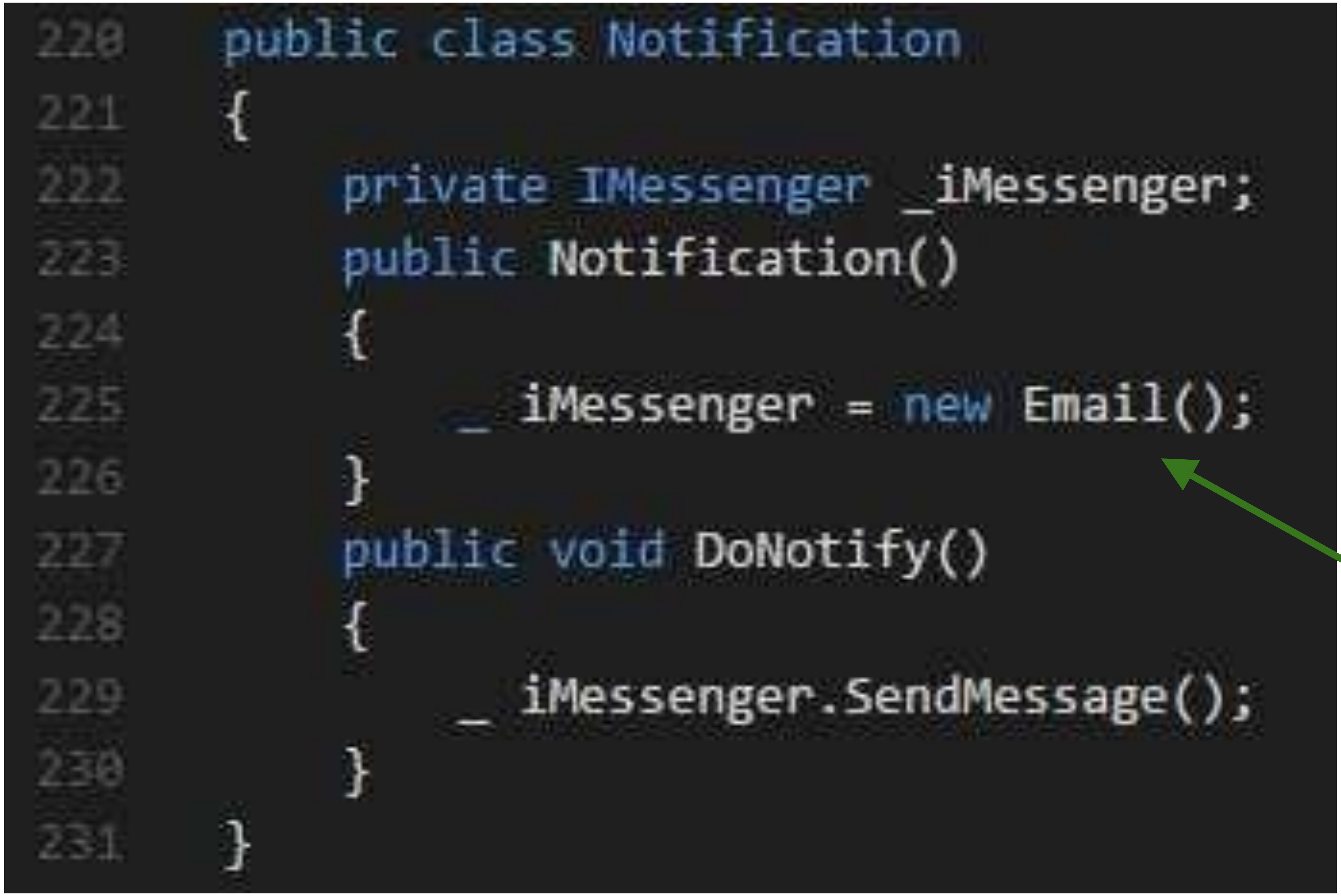


```
public class Email : IMessenger
{
    public void SendMessage()
    {
        // code to send email
    }
}
```



```
public class SMS : IMessenger
{
    public void SendMessage()
    {
        // code to send SMS
    }
}
```

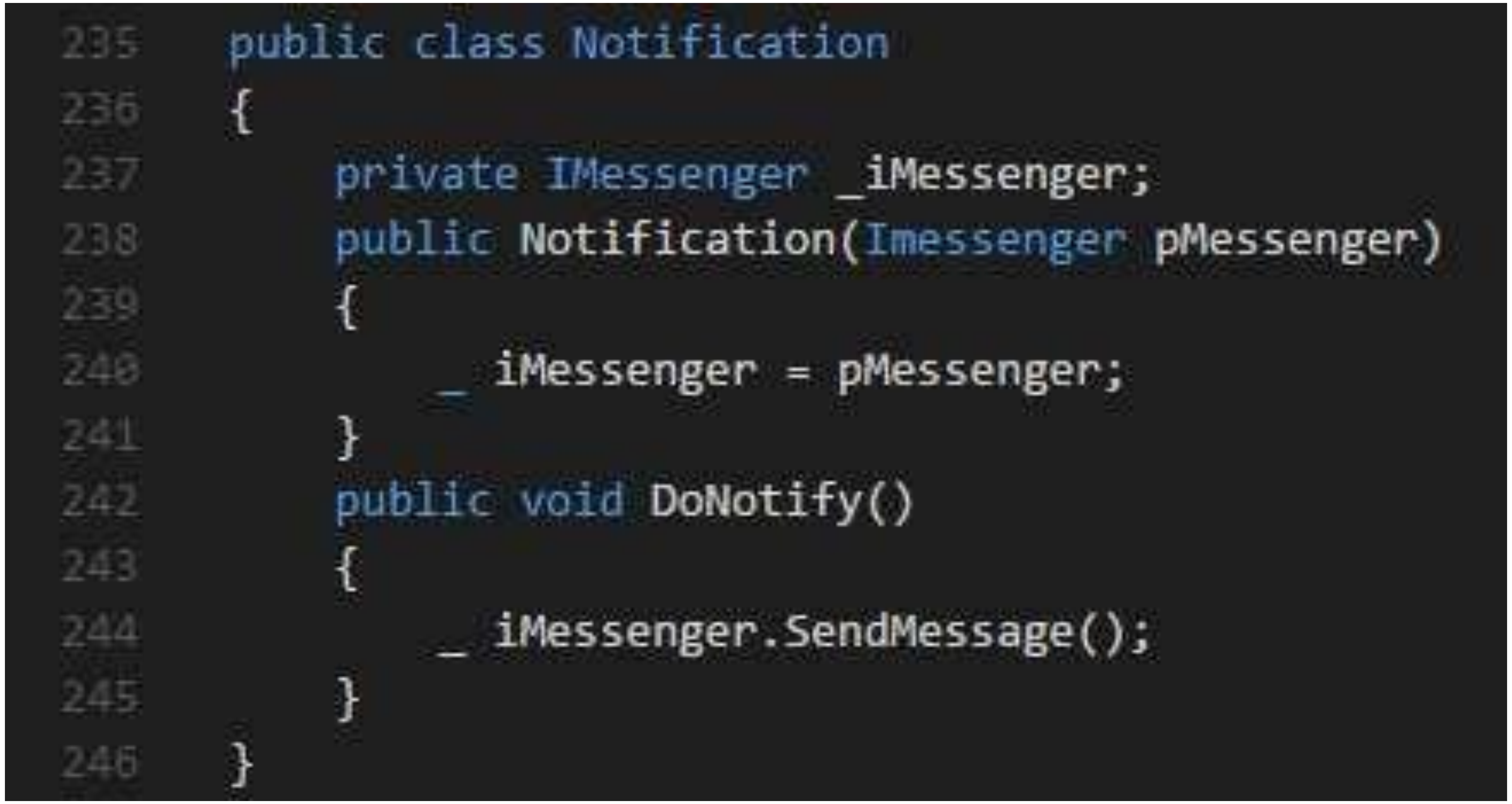
Dependency Inversion Principle



```
public class Notification
{
    private IMessenger _iMessenger;
    public Notification()
    {
        _ iMessenger = new Email();
    }
    public void DoNotify()
    {
        _ iMessenger.SendMessage();
    }
}
```

Dependency Inversion Principle

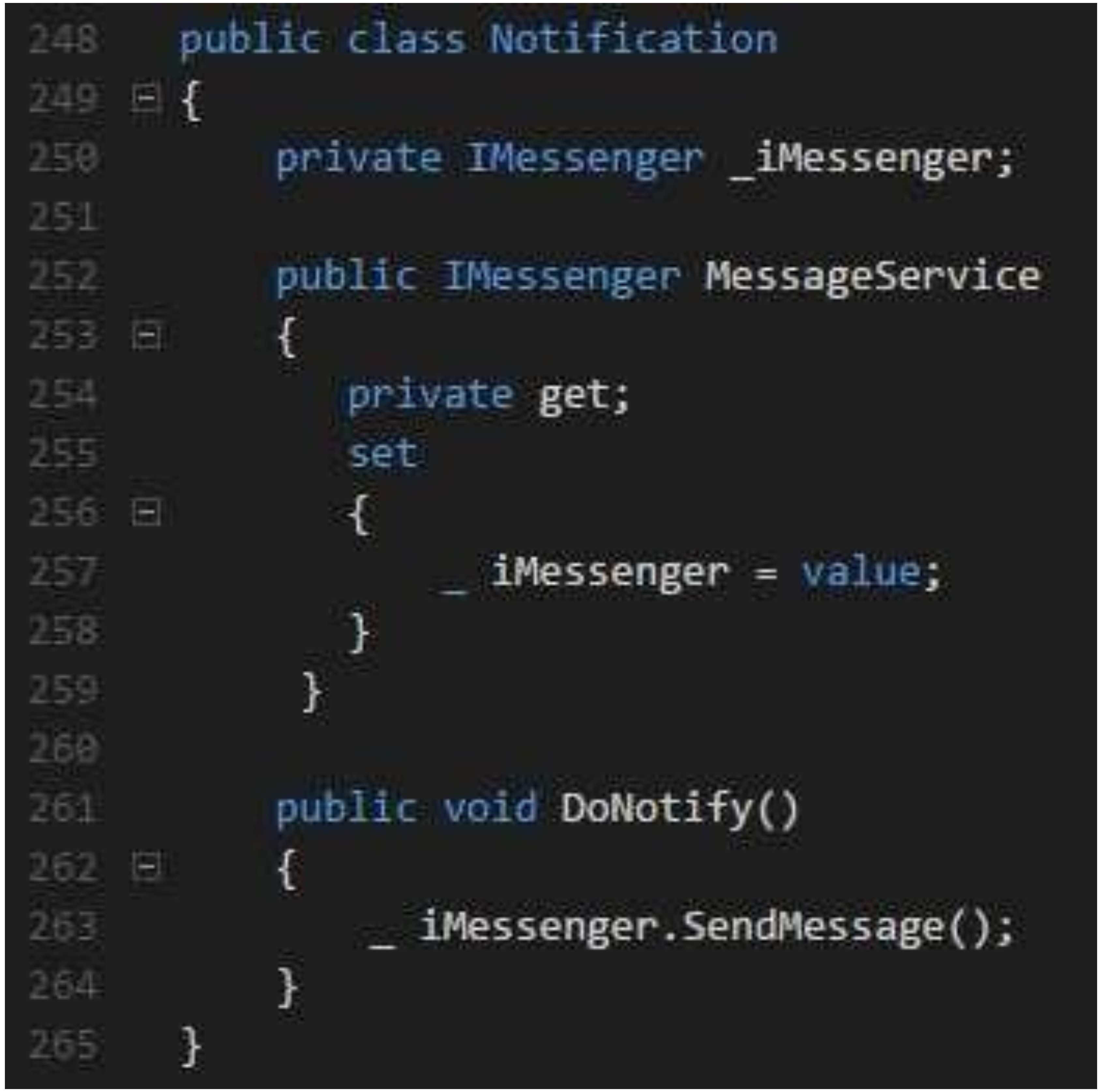
Constructor injection:



```
public class Notification
{
    private IMessenger _iMessenger;
    public Notification(Imessenger pMessenger)
    {
        _iMessenger = pMessenger;
    }
    public void DoNotify()
    {
        _iMessenger.SendMessage();
    }
}
```

Dependency Inversion Principle

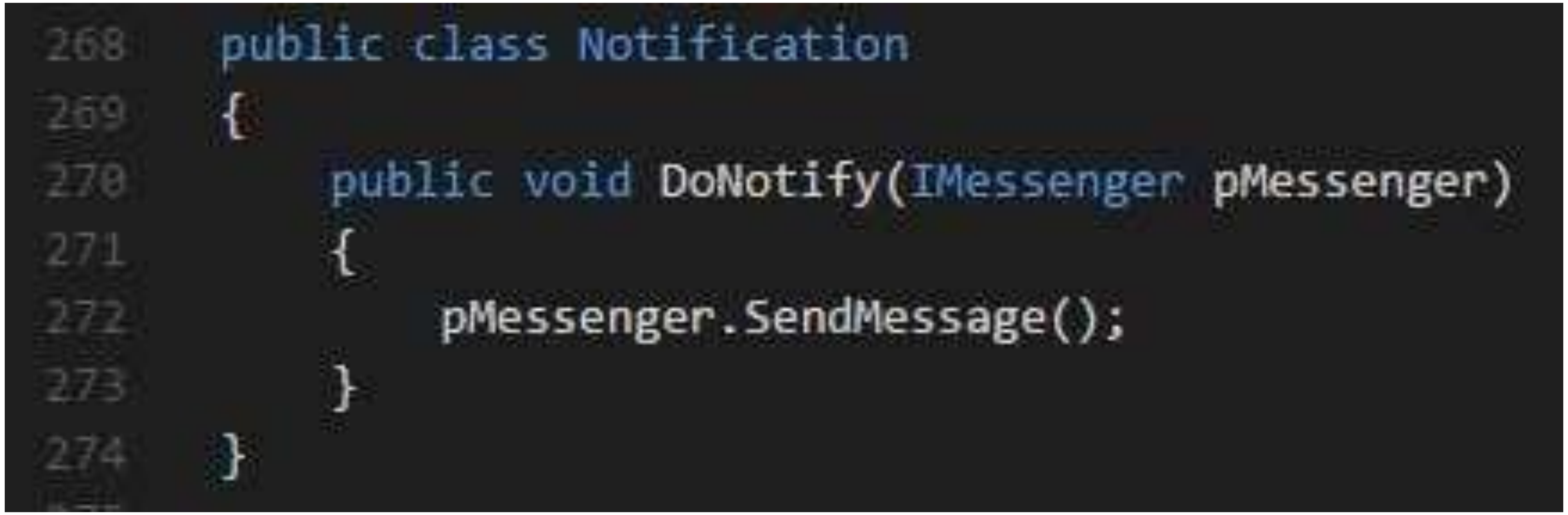
Property injection:



```
public class Notification
{
    private IMessenger _iMessenger;

    public IMessenger MessageService
    {
       private get;
       set
       {
           _ iMessenger = value;
       }
     }

    public void DoNotify()
    {
        _ iMessenger.SendMessage();
    }
}
```

Dependency Inversion Principle

Method injection:



```
public class Notification
{
    public void DoNotify(IMessenger pMessenger)
    {
        pMessenger.SendMessage();
    }
}
```

Keep in mind

DRY - Don’t repeat yourself

+

SLAP - Single layer abstraction principle

+

SOLID

BEST DEVELOPER

TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

Q&A

EIU
TRƯỜNG ĐẠI HỌC
QUỐC TẾ
MIỀN ĐÔNG
EASTERN
INTERNATIONAL
UNIVERSITY

Start your future at EIU

THANK YOU

Introduction

Sample text