



ENCAPSULATION. INHERITANCE. POLYMORPHISM.

.NET & JS LAB

Anzhelika Kravchuk

Инкапсуляция

Способность типа, позволяющее объединить данные и методы, работающие с ними, скрывая при этом внутренние данные и детали реализации, делая доступными для приложений только определенные части типа

Единственный способ взаимодействия внешнего кода с объектом или классом это осуществление доступа через четко определенный набор членов типа (type contract)

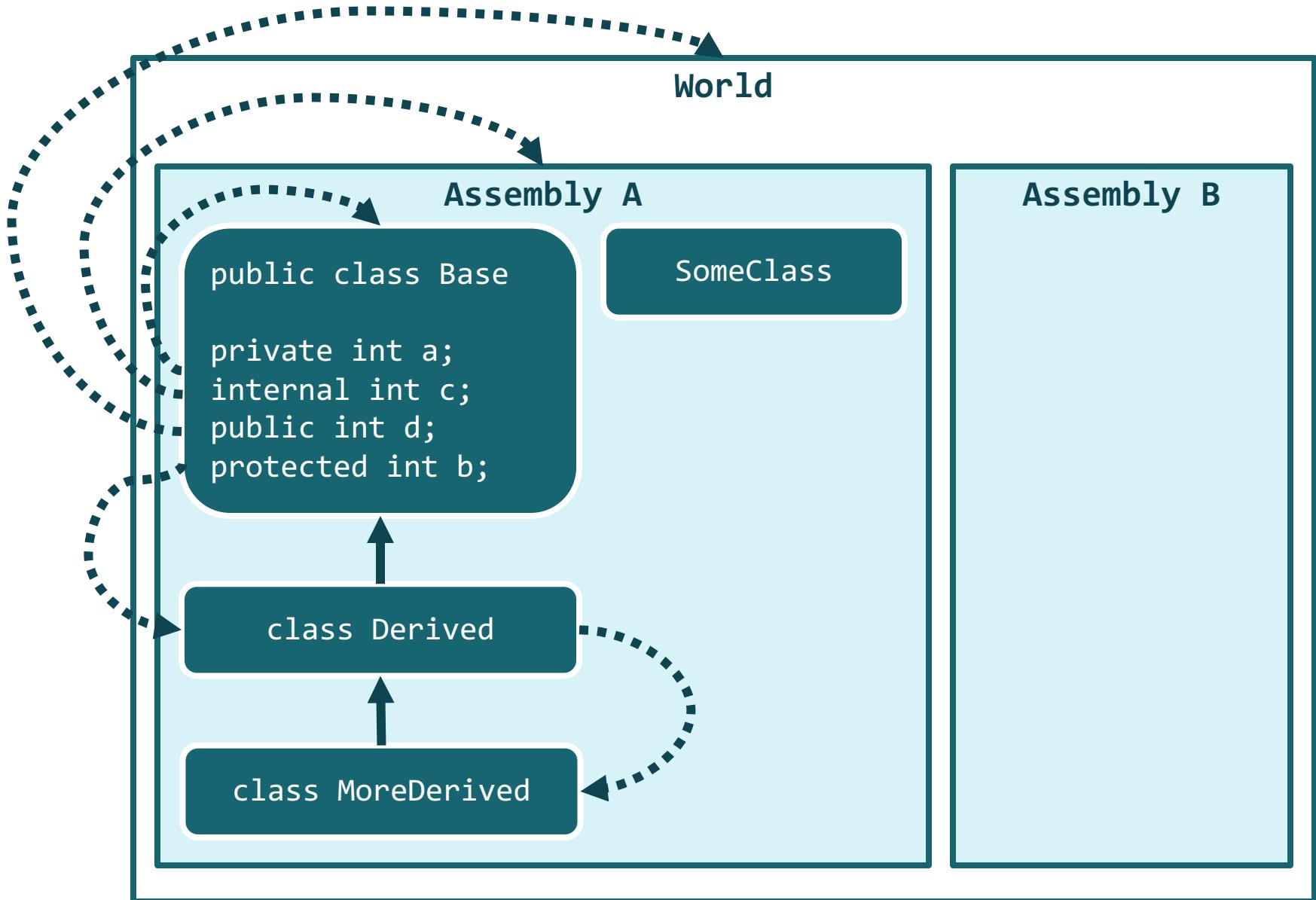
Преимущества

Возможность легко изменить детали реализации типа без необходимости переписывать приложения, использующие тип

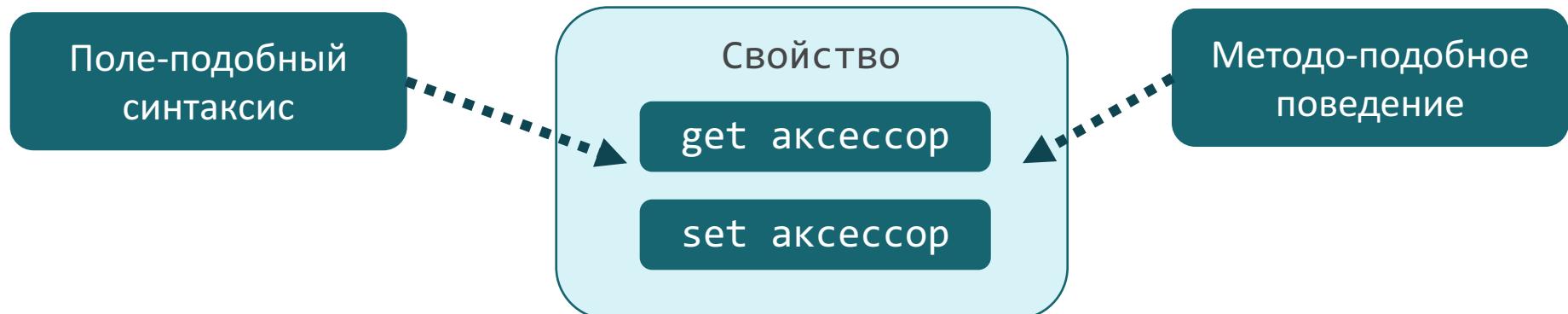
Клиентские приложения не могут исказить состояние типа, выполняя изменения, вызывающие сбои в работе типа и приводящие к непредсказуемым результатам

Внешний код сосредоточен только на полезных свойствах объекта

Модификаторы доступа



Определение свойства



Свойства позволяют осуществлять

контролируемый доступ к полями

проверку данных

контроль чтения/записи

Определение свойства

Модификатор доступа свойства

Спецификатор типа свойства

Имя свойства

Можно изменить модификатор доступа либо get, либо set аксессора свойства

Логика свойства определяется get и set аксессорами

```
private string someString;  
  
public string SomeString  
{  
    get  
    {  
        return this.someString;  
    }  
    private set  
    {  
        someString = value;  
    }  
}
```

Set-аксессор всегда имеет один параметр типа, предоставляемый свойством

Модификаторы свойства

Статический модификатор	static
Модификатор доступа	public internal private protected
Модификатор наследования	new virtual abstract override sealed
Модификатор небезопасного кода	unsafe extern

Автоматические свойства (Automatically Implemented Properties)

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

При использовании автоматического свойства, компилятор создает private поля и автоматически генерирует код для чтения и записи этого поля

```
private string <Name>k__BackingField;  
public string Name  
{  
    get  
    {  
        return <Name>k__BackingField;  
    }  
    set  
    {  
        this._name = value;  
    }  
}
```

Везде, где необходимо добавить поле и можно его сделать public, а не писать свойство для получения и установки его значения, можно использовать автоматические свойства

Полезны, когда не требуется дополнительной обработки или проверки значений полей

Не возникает никаких последствий при переходе от автоматических свойств на определенные

Инициализация объекта

```
class Employee
{
    private string name;
    private string department;
    // Initialize both fields
    public Employee(string name, string department)
    {
        this.name = name;
        this.department = department;
    }
    // Initialize name only
    public Employee(string name)
    {
        this.name = name;
    }
    // Initialize department only
    public Employee(string department)
    {
        this.department = department
    }
    ...
    // Is "Fred" the name of an employee or a department?
    Employee employee = new Employee("Fred");
}
```

Компилятор не может различить
два конструктора, принимающих
один параметр

СТЕ

Инициализация объекта

```
class Employee
{
    public Employee ()
    {
        ...
    }
    public Employee (int grade)
    {
        ...
    }
    public string Name { get; set; }
    public string Department { get; set; }
    ...
}
```

Нужно стараться определять только конструкторы, устанавливающие все необходимые значения свойств по умолчанию

```
Employee louisa = new Employee() { Department = "Technical" };
Employee john = new Employee { Name = "John" };
Employee mike = new Employee
{
    Name = "Mike",
    Department = "Technical"
};
```

Рекомендации по определению и использованию свойств

Свойства могут быть «только для чтения» или «только для записи», а поля всегда доступны и для чтения, и для записи.

Метод свойства может привести к исключению, а при доступе к полям исключений не бывает

Свойства нельзя передавать в метод как параметры с ключевым словом `out` или `ref`

Свойство-метод может выполняться довольно долго, а доступ к полям выполняется моментально

При вызове несколько раз подряд метод свойства может возвращать разные значения (`System.DateTime.Now`), а поле возвращает одно и то же значение

Метод свойства может создавать наблюдаемые сторонние эффекты, а при доступе к полю это невозможно

Рекомендации по определению и использованию свойств

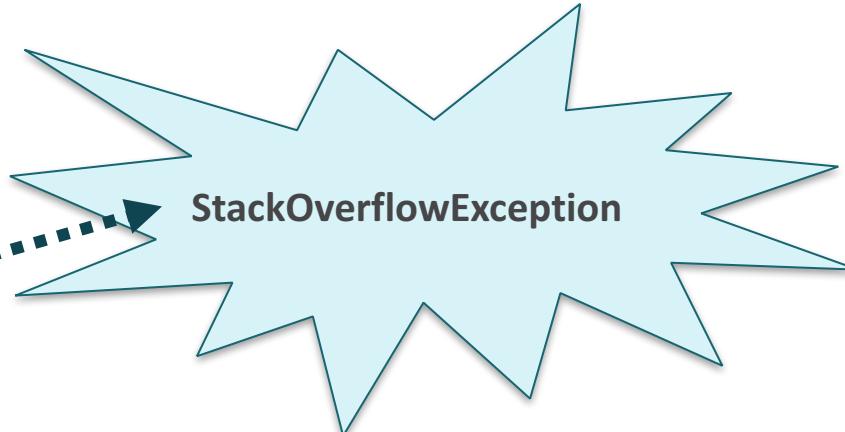
Свойства следует использовать надлежащим образом

BankAccount

- X double Balance (get, set)
- ✓ WithdrawMoney(double Amount)
- ✓ DepositMoney (double Amount)

Не следует представлять каждое поле как
свойство, если для этого нет веских
оснований

```
int data;  
public int Data  
{  
    get  
    {  
        return Data;  
    }  
    ...  
}
```



Определение индексатора

Индексатор обеспечивает механизм инкапсуляции множества значений, так же, как свойство инкапсулирует одно значение

get и set аксессоры используется для управления тем, как значения извлекается или устанавливается на основе индекса передаваемого в качестве параметра для индексации

get и set аксессоры используют свойство-подобный синтаксис

Индексатор использует массив-подобный при доступе к элементам множества

При индексации можно использовать нецелый тип индекса

```
CustomerAddressBook addressBook = ...;  
Address customerAddress = addressBook["a2332"];  
...  
Address customerAddress = addressBook[99];
```

Можно определить перегруженные индексаторы

Определение индексатора

Модификатор доступа

Тип возвращаемого значения

Имя индексатора всегда this

Типы и имена параметров

```
public Address this[string customerID]
{
    get
    {
        return database.FindCustomer(customerID);
    }
    set
    {
        database.UpdateCustomer(customerID, value);
    }
}
```

Параметры индексатора могут быть описаны как параметры-значения или как параметр-список

При написании индексатора следует убедиться, что он содержит логику обработки ошибки в случае, когда код принимает недопустимое значение индекса

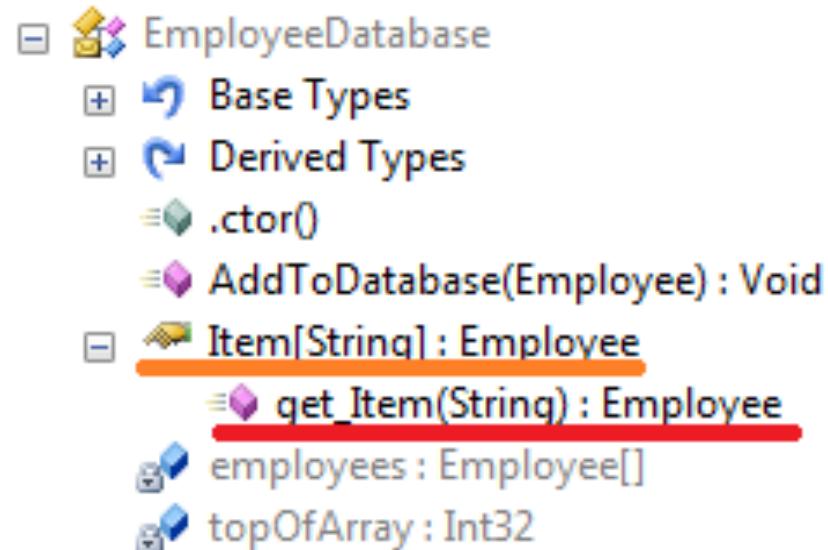
Нельзя определить статические индексаторы

Определение индексатора

```
class EmployeeDatabase
{
    employees [] Employee;
    int topOfArray;

    public EmployeeDatabase(){...}
    public void AddToDatabase(Employee employee){...}

    public Employee this[string name]{...}
}
```

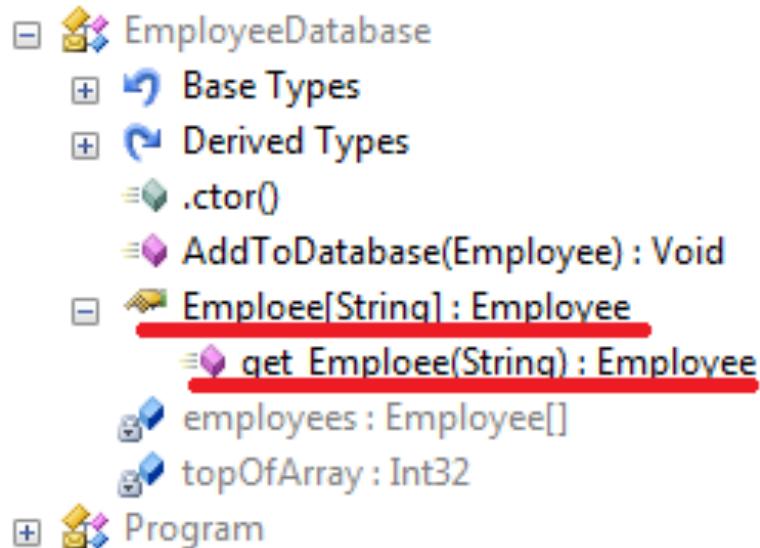


Создание индексатора

```
using System.CompilerServices.Runtime;

class EmployeeDatabase
{
    employees [] Employee;
    int topOfArray;

    public EmployeeDatabase(){...}
    public void AddToDatabase(Employee employee){...}
    [IndexerName("Employee")]
    public Employee this[string name]
}
```



Сравнение индексаторов и массивов

При использовании индексатора используется массиво-подобный синтаксис, однако между индексаторами и массивами существует несколько важных различий

Индексы

При индексировании элементов в массиве можно использовать только числовые индексы

Индексаторы предоставляют возможность использования не числовых индексов

Перегрузка

Операцию индексирования нельзя перегрузить

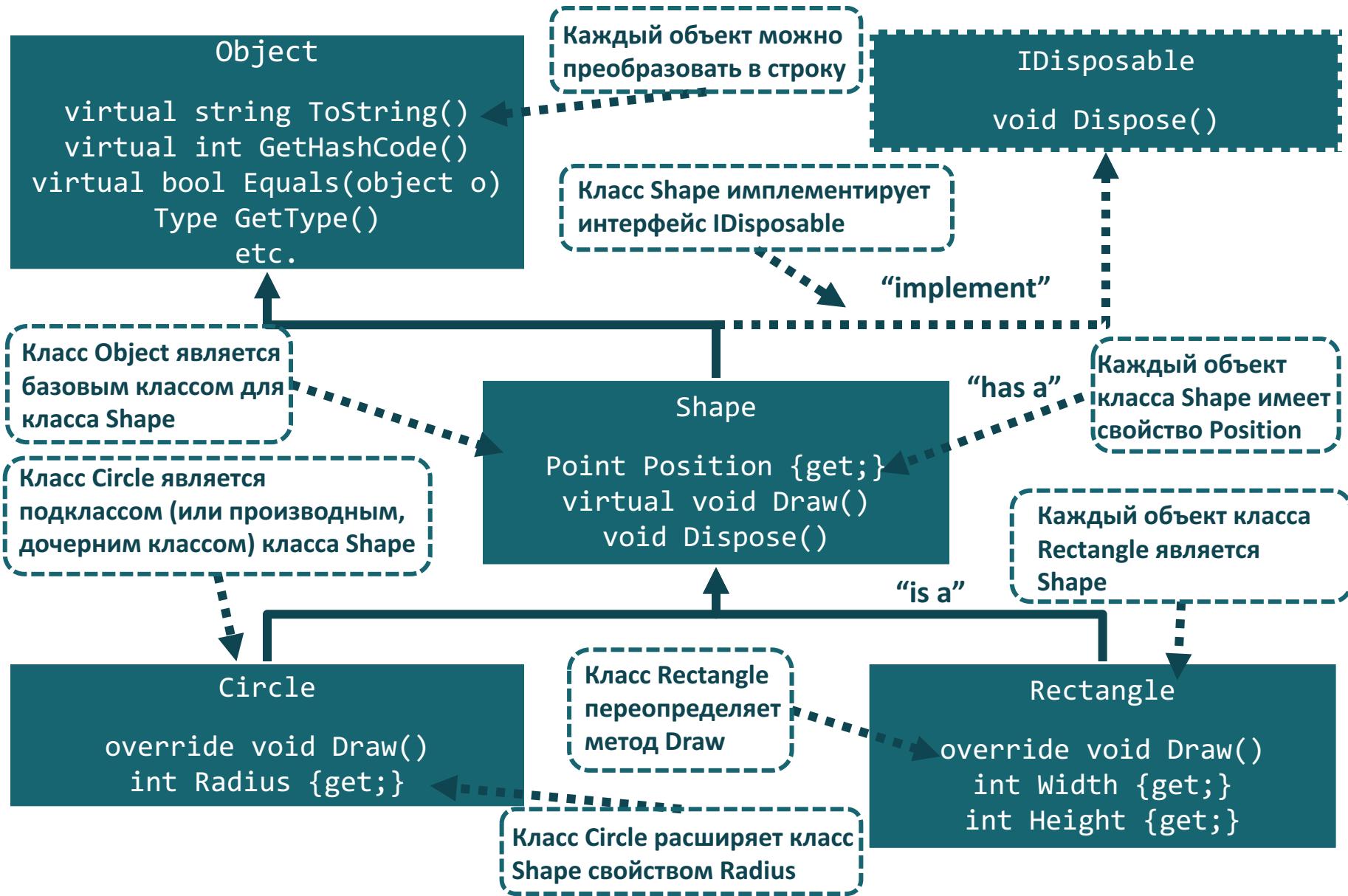
Возможно переопределять и перегружать индексаторы

Использование в качестве параметра

Индексированные элементы массива могут использоваться как при передаче параметров по значению, так и как `ref` и `out` параметры

Индексаторы нельзя использовать в качестве `ref` или `out` параметров, но можно при передаче параметров по значению

Наследование в C#



Наследование «is a»

Наследование реализации это свойство системы, позволяющее описать новый класс на основе существующего с целью повторного использования, расширения и изменения функциональности базового класса

```
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}
```

```
// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```

C# поддерживает только единичное наследование реализации

Наследование является транзитивным



Наследование «is a»

Object

Equals()

Finalize()

GetHashCode()

GetType()

MemberwiseClone()

ReferenceEquals()

ToString()

Наследуемый член

WorkItem:Object

Equals()

Finalize()

GetHashCode()

GetType()

MemberwiseClone()

ReferenceEquals()

ToString()
(переопределен)

int ID

string Title

TimeSpan jobLength

Update()

WorkItem()

ChangeRequest:WorkItem

Equals()

Finalize()

GetHashCode()

GetType()

MemberwiseClone()

ReferenceEquals()

ToString()
(переопределен)

int ID

string Title

TimeSpan jobLength

Update()

int originalItemID

ChangeRequest()

Иерархии наследования

```
class Employee  
{  
...  
}
```

```
class Manager : Employee  
{  
...  
}
```

```
class ManualWorker : Employee  
{  
...  
}
```

```
Manager manager = new Manager("Fred", "VP");  
Employee employee = manager;  
object obj = manager;
```

Ссылка на объект одного типа может быть инициализирована ссылкой на объект другого типа только пока этот тип является классом, находящимся выше в иерархии наследования

Доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается



Иерархии наследования

```
Manager manager = new Manager("Fred", "VP");
Employee employee = manager; // employee refers to a Manager
...
Manager managerAgain = employee as Manager;
// OK - employee is a Manager
...
ManualWorker worker = new ManualWorker("Bert");
employee = worker; // employee now refers to a ManualWorker
...
bool ok = employee is Manager;
// returns false - employee is a ManualWorker
```

Операция **as** проверяет, является ли объект ссылкой на указанный тип и, если это так, возвращает новую ссылку, используя этот тип, в противном случае возвращает null

Операция **is** проверяет, является ли объект ссылкой на указанный тип и возвращает true, если это так и false в противном случае

Вызов методов и конструкторов базового класса

```
class Employee
{
    protected string empName;
    public Employee(string name)
    {
        this.empName = name;
    }
    ...
}

class Manager : Employee
{
    protected string empGrade;
    public Manager(string name, string grade) : base(name)
    {
        this.empGrade = grade;
    }
    ...
}
```

Хорошей практикой для конструктора производного класса является вызов конструктора базового класса как части инициализации объекта производного класса

Вызов методов и конструкторов базового класса

```
class Manager : Employee
{
    public Manager(string name, string grade)
    {
        ...
    }
    ...
}
```

Компилятор C#

```
class Manager : Employee
{
    public Manager(string name, string grade) : base()
    {
        ...
    }
    ...
}
```

Если в конструкторе производного класса нет явного вызова конструктора базового класса, перед выполнением кода компилятор пытается вставить в конструктор производного класса вызов конструктора по умолчанию базового класса

Скрытие методов базового класса

```
class Employee
{
    protected void DoWork()
    {
        ...
    }
}

class Manager : Employee
{
    public new void DoWork()
    {
        // Hide the DoWork method in the base class
        ...
    }
    ...
}
```

Скрытие :
замена функциональности базового
класса новым поведением

Для указания намеренного действия
используется ключевое слово **new**

Переопределение виртуальных методов базового класса

```
class Object
{
    public virtual string ToString()
    {
        ...
    }
}
```

```
class Employee
{
    protected string empName;
    ...
    public override string ToString()
    {
        return string.Format("Employee: {0}", empName);
    }
}
```

Переопределение:
намеренное изменение
или расширение абстрактной
или виртуальной реализации
унаследованного метода,
свойства, индексатора или
события базового класса

Для переопределения в наследуемом классе
используется ключевое слово **override**

Переопределить можно только члены класса, которые помечены в базовом классе
как **virtual**, **override** или **abstract**

Вызов методов базового класса

```
class Employee
{
    protected virtual void DoWork()
    {
        ...
    }
}

class Manager : Employee
{
    protected override void DoWork()
    {
        // Do processing specific to Managers
        ...
        // Call the DoWork method in the base class
        base.DoWork();
    }
    ...
}
```

Позволяет создавать собственную функциональность в дополнение к существующей, определяемой базовым классом

Производный класс с замещенным или переопределенным методом или свойством сохраняет доступ к методу или свойству базового класса с помощью ключевого слова **base**

Определение запечатанных классов и методов

Object

Employee

Manager

```
sealed class Manager : Employee { . . . }
```

В запечатанном классе не могут
объявляться виртуальные методы

```
class Manager : Employee
{
    ...
    protected sealed override void DoWork()
    {
        ...
    }
}
```

Производный класс не может
переопределить запечатанный метод

Можно запечатать только **override** методы,
и следует объявлять их как **sealed override**

В .NET Framework все значимые типы (структуры и перечисления) неявно запечатаны

Полиморфизм

```
class Employee
{
    public virtual string GetTypeName()
    {
        return "This is an Employee";
    }
}
```

```
class Manager : Employee
{
    public override string GetTypeName()
    {
        return "This is a Manager";
    }
}
```

```
Employee employee;
Manager manager = new Manager();
ManualWorker worker = new ManualWorker();
employee = manager;
Console.WriteLine(employee.GetTypeName());
employee = worker;
Console.WriteLine(employee.GetTypeName());
```

Виртуальные методы, определенные в классах, разделяющих иерархию наследования, позволяют вызывать различные версии одного и того же метода в зависимости от типа объекта, который определяется динамически во время выполнения

```
class ManualWorker : Employee
{
    // Does not override GetTypeName
}
```



Полиморфизм

Ссылка на объект одного типа может быть инициализирована ссылкой на объект другого типа только пока он является типом, находящимся выше в иерархии наследования



Доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается



При вызове виртуальных методов по ссылке на базовый класс определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем тип объекта определяется во время выполнения





Понятие полиморфизма имеет два аспекта (msdn)

- ✓ Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в коллекциях и в качестве параметров методов, при этом объявленный тип объекта больше не идентичен его типу времени выполнения
- ✓ Базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их, предоставляя свои собственные определение и реализацию. Во время выполнения метода среда CLR ищет тип времени выполнения объекта и вызывает это переопределение виртуального метода. Таким образом, в коде можно вызвать метод базового класса и вызвать выполнение метода с версией производного класса

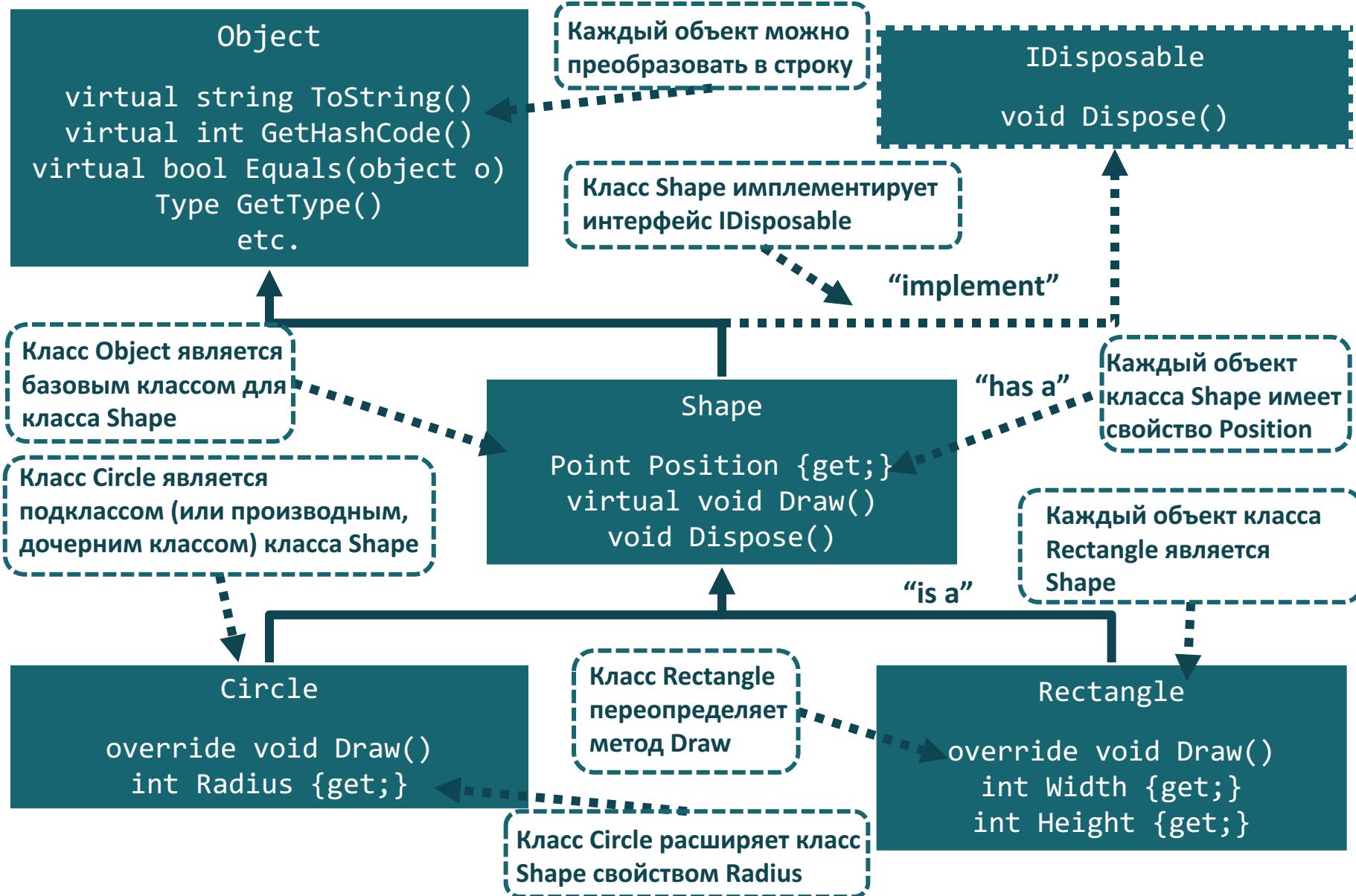
Переопределение и скрытие методов

```
class A
{
    public virtual void M() { Console.WriteLine("A"); }
}
class B: A
{
    public override void M() { Console.WriteLine("B"); }
}
class C: B
{
    new public virtual void M() { Console.WriteLine("C"); }
}
class D: C
{
    public override void M() { Console.WriteLine("D"); }
}
static void Main()
{
    D d = new D(); C c = d; B b = c; A a = b;
    d.M(); c.M(); b.M(); a.M();
}
```

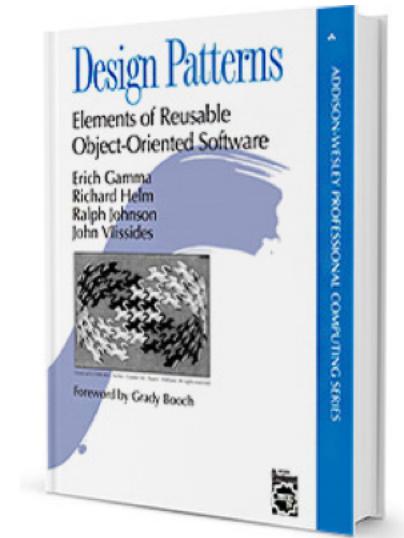
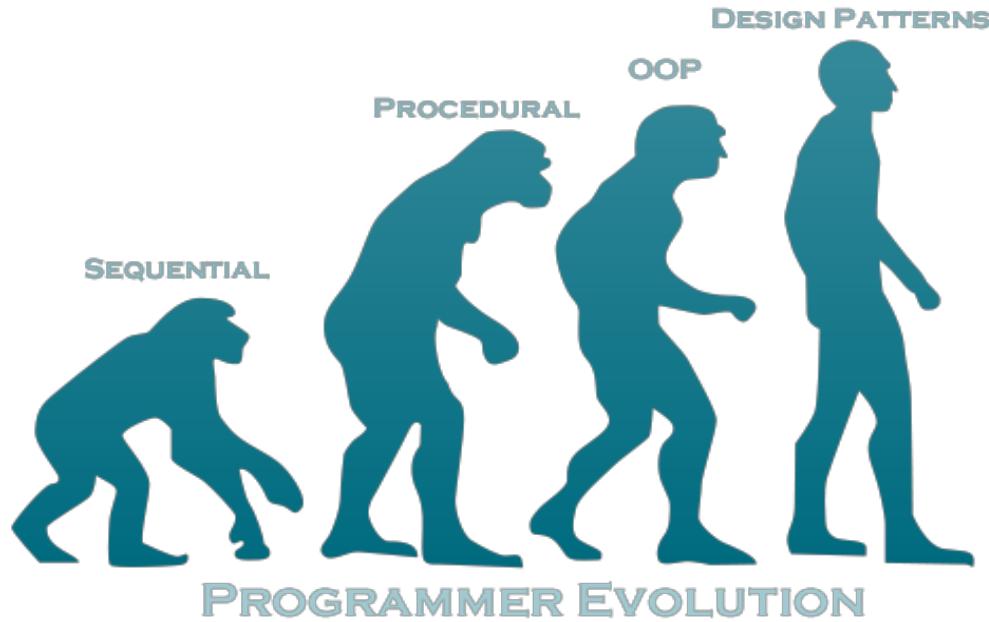


DDBB

Интерфейсы в C#



Моделирование абстракций и слабо связанный код



Программируй, основываясь на интерфейсе, а не на классах!

"Банда четырех"

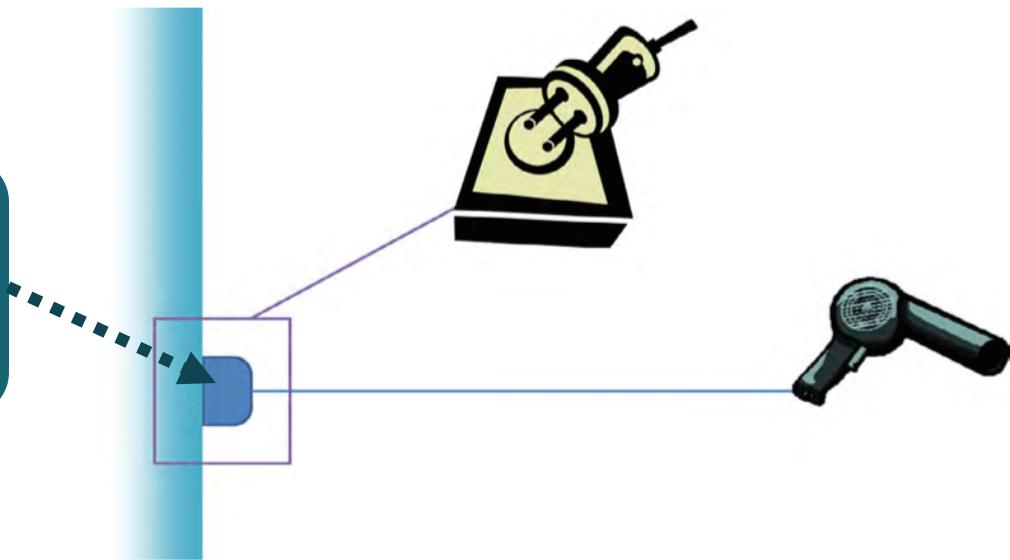
(Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидс)

Электрическая проводка или как написать слабо связанный код



Эквивалент использованию
универсальной практики
написания сильно связанного кода

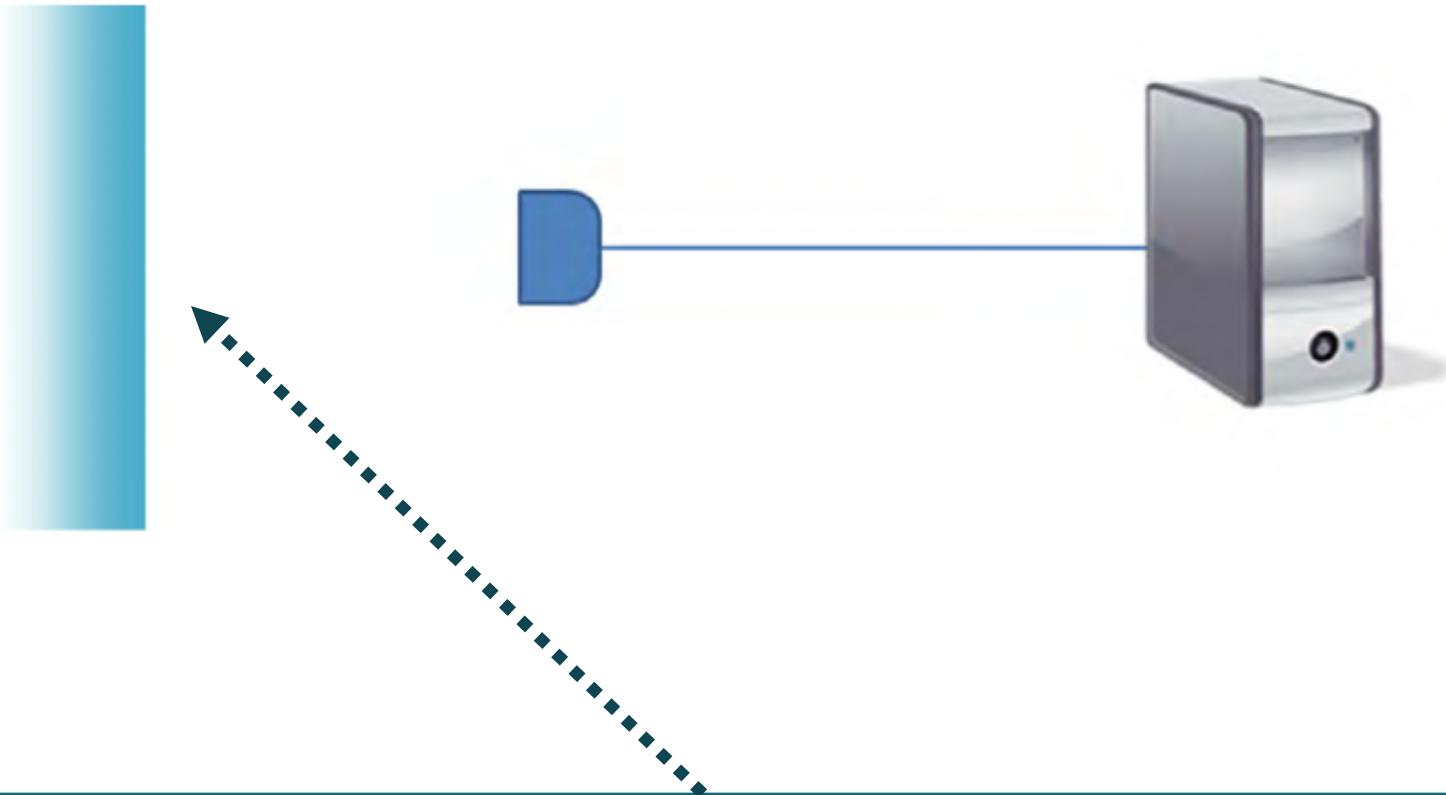
Посредством использования
розеток и вилок фен можно слабо
связать со стенной розеткой



Электрическая проводка или как написать слабо связанный код

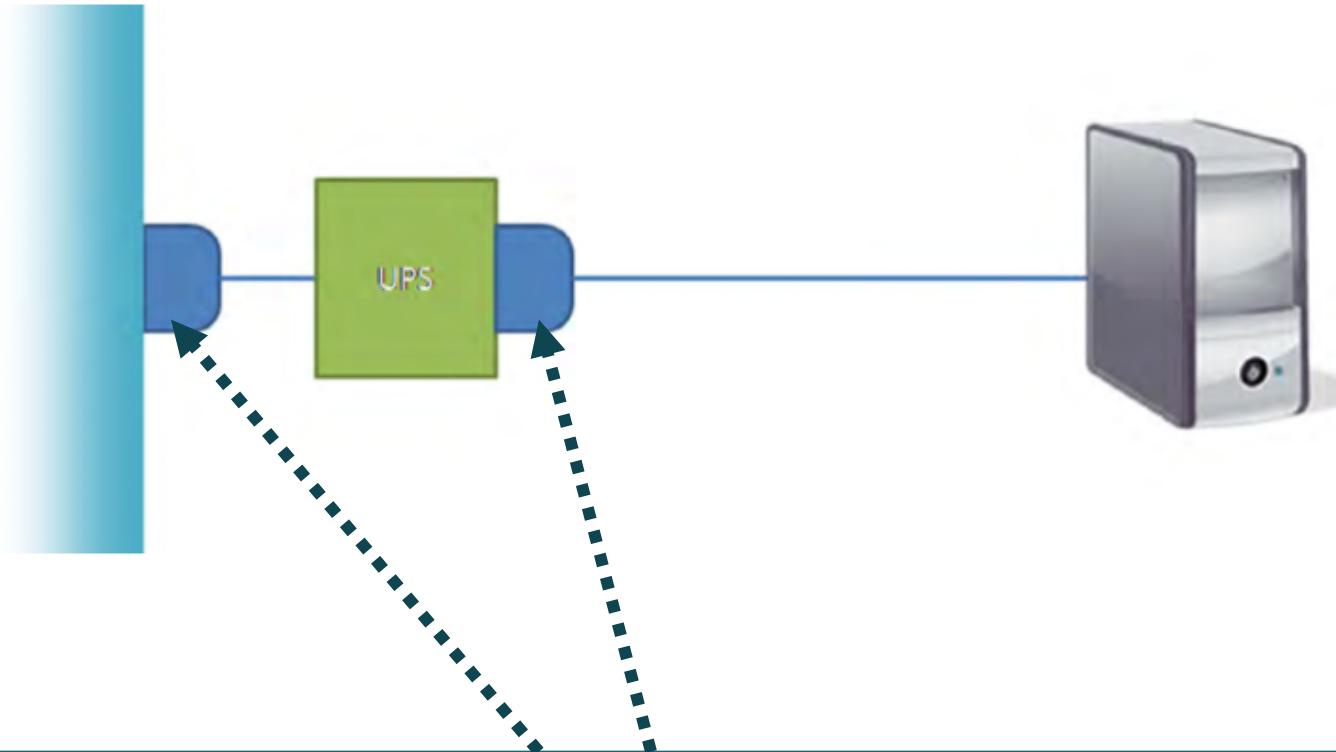


Электрическая проводка или как написать слабо связанный код



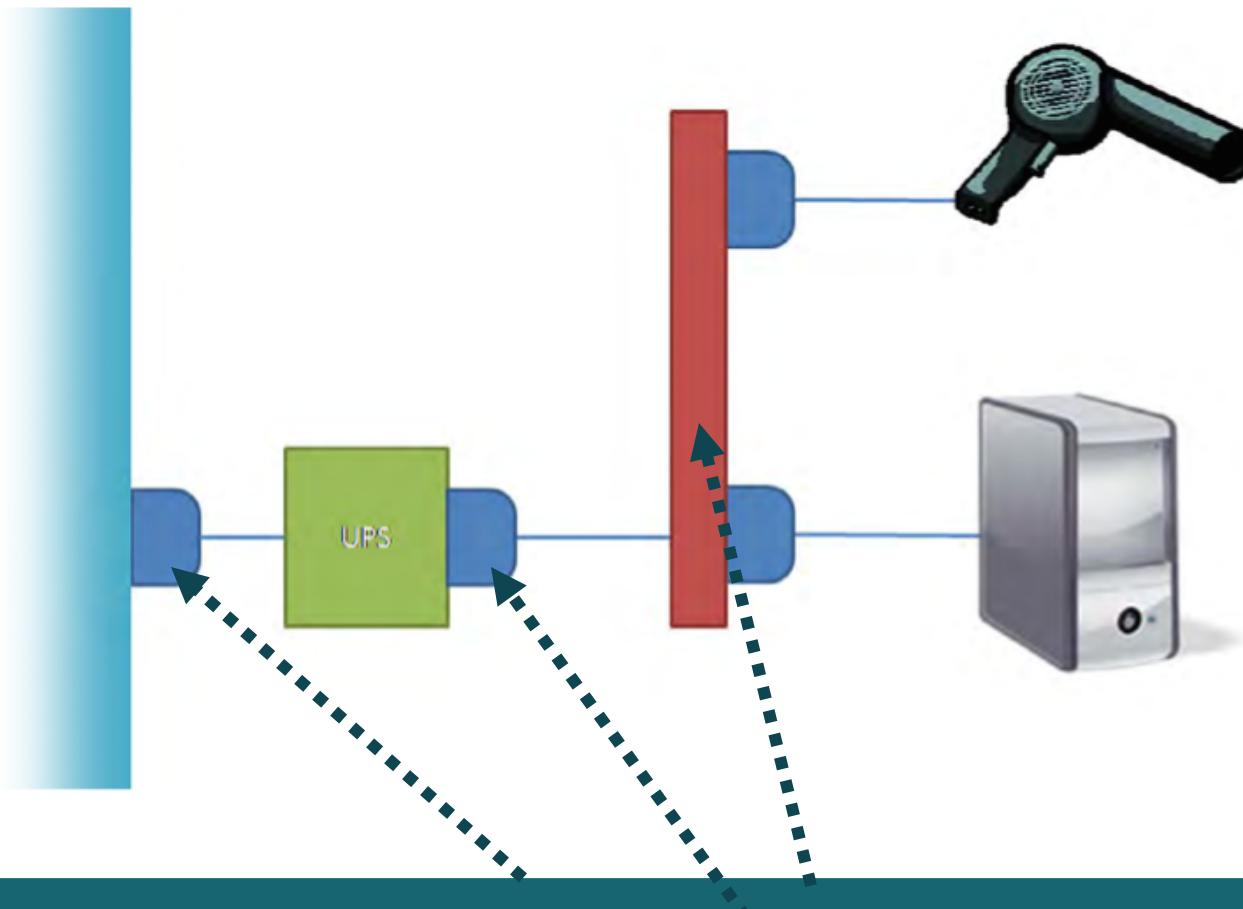
Отключение компьютера не приводит ни к взрыву стены, ни к взрыву компьютера. Это можно приблизенно сравнить с паттерном Null Object

Электрическая проводка или как написать слабо связанный код



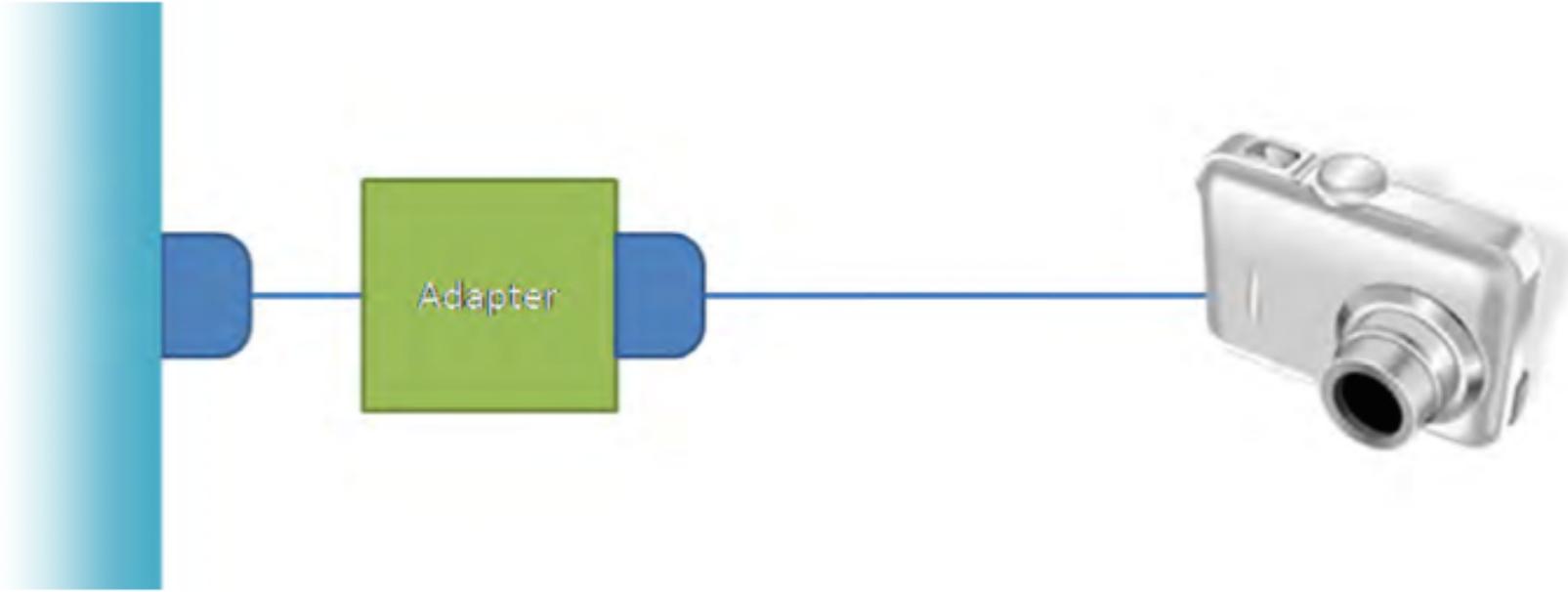
Можно воспользоваться блоком бесперебойного питания для того, чтобы компьютер продолжал работать при отключении электричества. Это соответствует паттерну проектирования Decorator

Электрическая проводка или как написать слабо связанный код



Удлинитель дает возможность подключать несколько устройств к одной стенной розетке. Это соответствует паттерну проектирования Composite

Электрическая проводка или как написать слабо связанный код



Во время путешествия нам часто нужно использовать адаптер для того, чтобы подключить устройство к иностранной розетке (например, чтобы перезарядить фотоаппарат). Это соответствует паттерну проектирования Adapter

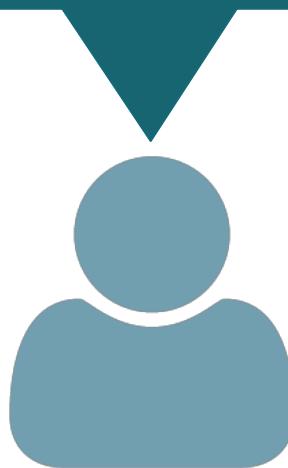
Понятие интерфейса

- В чем сила интерфейсов?
- Сила интерфейсов в слабости системы типов!

Если ты **выполнишь** контракт,
я смогу **воспользоваться**
твоим сервисом

Интерфейс
это контракт

Я согласен **выполнить**
контракт для того, чтобы ты
мог **воспользоваться** моим
сервисом



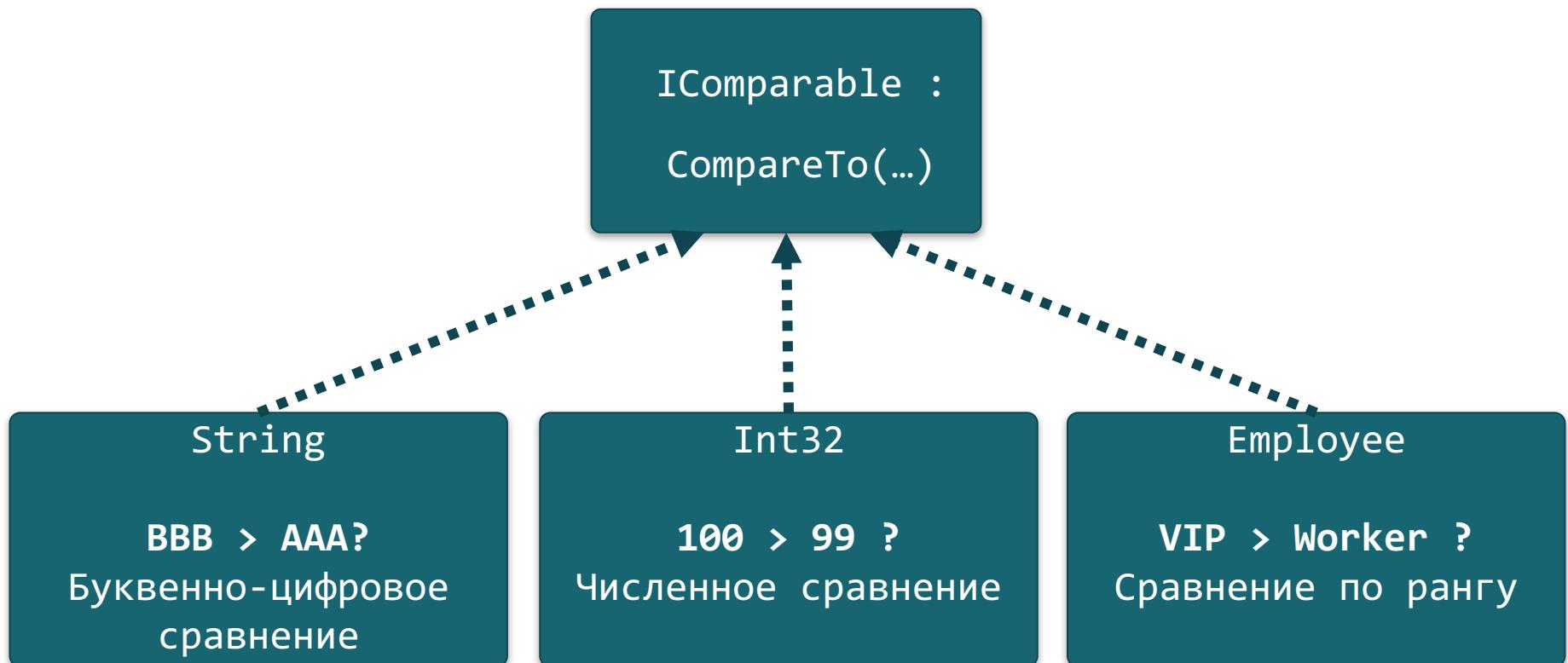
Потребитель

Тип реализует
интерфейс



Исполнитель

Понятие интерфейса



Различные классы могут реализовывать один и тот же интерфейс по-разному, при условии, что они представляют набор методов, определяемых интерфейсом

Когда класс должен реализовывать интерфейс?

1. Класс реализует стратегию или является частью семейства объектов: IRepository, IFormatter, etc.
2. Класс реализует ролевой интерфейс (следствие ISP): ICloneable, IComparable etc.
3. Класс реализует интерфейс, требуемый для связи с другими классами. Класс является адаптером; необходимость интерфейса обусловлено принципом DIP.
4. Класс реализует интерфейс, поскольку его реализация завязана на внешнее окружение. Обеспечивает тестируемость клиентов данного класса. **Это не должно быть единственной причиной.**

Выделять интерфейсы «на всякий случай» - не нужно! Если это библиотечный код, то интерфейсы там полное зло: любое их изменение – ломает всех клиентов. А если это продакшн код, то он все равно будет меняться, и наличие лишних интерфейсов лишь сделает этот процесс более сложным.

Определение интерфейса

[Атрибуты интерфейса]

[Модификаторы интерфейса] **interface IInterfaceName** [Параметры обобщенных типов, интерфейсы]

{
}
}

Члены интерфейса – методы, свойства, индексаторы, события,

Интерфейс (interface) именованный набор абстрактных членов

Интерфейс не содержит код или данные, а просто указывает методы и свойства, которые наследуемый от интерфейса класс должен реализовать

Создание и реализация интерфейсов

```
interface ICalculator  
{  
    double Add();  
    double Subtract();  
    double Multiply();  
    double Divide();  
}
```

Для определения интерфейса используется ключевое слово `interface`

```
class Calculator : ICalculator  
{  
    public virtual double Add() { ... }  
    public double Subtract () { ... }  
    public double Multiply () { ... }  
    public double Divide () { ... }  
}
```

Модификатор доступа членов не указывается, тело метода заменяется точкой с запятой

Имена методов и типы возвращаемого значения в точности совпадают

Любые параметры (в том числе модификаторы ключевых слов `ref` и `out`) в точности совпадают

Реализация метода может быть как виртуальной так и не виртуальной

- ✓ Класс может быть более доступным, чем его базовый интерфейс
- ✓ Интерфейс не может быть более доступным, чем его базовый интерфейс

Создание и реализация интерфейсов

```
class Calculator : ICalculator, IComparable
{
    . . .
    public double Add() {. . .}
    public double Subtract () {. . .}
    public double Multiply () {. . .}
    public double Divide () {. . .}
    . . .
    public int CompareTo(Object obj) {. . .}
    . . .
}
```

Класс может иметь только один базовый класс, однако он может реализовывать несколько интерфейсов



Использование интерфейсных ссылок

```
class Calculator : ICalculator  
{  
    ...  
}
```

```
Calculator calculator = new Calculator();  
ICalculator calc = calculator;
```

Как можно ссылаться на объект с помощью ссылочной переменной класса, находящегося выше в иерархии наследования, так можно ссылаться на объект с помощью переменной интерфейса, который реализует этот класс

Нельзя присвоить объект ICalculator переменной Calculator без приведения, предварительно не проверив, является ли он ссылкой на объект Calculator, а не на какой-нибудь другой класс, также реализующий интерфейс ICalculator

```
Calculator calcAgain = calc as Calculator;  
bool isCalc = calc is Calculator;
```

```
ICalculator iCalc = calculator as ICalculator;
```

Использование интерфейсных ссылок

Техника ссылок на объект через интерфейс полезна, поскольку позволяет определить методы, которые могут принимать различные типы в качестве параметров, при условии, что типы реализуют указанный интерфейс

```
int PerformAnalysis(ICalculator calculator)
{
    ...
}
```

При ссылке на объект через интерфейс, можно вызвать только методы, которые видны через интерфейс

Явная и неявная реализация интерфейса

Неявная реализация интерфейса

```
interface ICalculator
{
    double Add();
    double Subtract();
    double Multiply();
    double Divide();
}
```

```
class Calculator : ICalculator
{
    public double Add() { . . . }
    public double Subtract () { . . . }
    public double Multiply () { . . . }
    public double Divide () { . . . }
}
```

```
interface ITaxCalculator
{
    double Add();
    double Subtract();
}
```

```
class Calculator : ICalculator, ITaxCalculator
{
    public double Add() { . . . }
    public double Subtract () { . . . }
    public double Multiply () { . . . }
    public double Divide () { . . . }
}
```

Методы каких интерфейсов ICalculator или ITaxCalculator реализуют методы Add и Subtract



Явная и неявная реализация интерфейса

```
class Calculator : ICalculator, ITaxCalculator
{
    . . .
    // This is the Add method for ICalculator
    public double Add() { . . . }
    // This is the Subtract method for ICalculator
    public double Subtract () { . . . }
    public double Multiply () { . . . }
    public double Divide () { . . . }
    // This is the Add method for ITaxCalculator?
    public double Add()
    {
        return calculatedValue + taxAmount;
    }
    // This is the Subtract method for ITaxCalculator?
    public double Subtract()
    {
        return calculatedValue - taxAmount;
    }
}
```



CTE

Явная и неявная реализация интерфейса

Явная реализация интерфейса

```
class Calculator : ICalculator, ITaxCalculator
{
    . . .
    double ICalculator.Add() { . . . }
    double ICalculator.Subtract() { . . . }
    double ICalculator.Multiply { . . . }
    double ICalculator.Divide { . . . }
    double ITaxCalculator.Add()
    {
        return calculatedValue + taxAmount;
    }
    double ITaxCalculator.Subtract()
    {
        return calculatedValue - taxAmount;
    }
}
```

```
Calculator calc= new Calculator();
...
double result = calc.Add();
```



```
Calculator calc= new Calculator();
...
ICalculator calculator = calc;
double result = calculator.Add();
ITaxCalculator taxCalc = calc;
double tax = taxCalc.Add();
```

- ✓ Нельзя указать модификатор доступа методам, являющимся частью явной реализации интерфейса
- ✓ При явной реализации метода интерфейса нельзя объявить его как виртуальный

Явная и неявная реализация интерфейса

```
struct SomeStruct : IComparable
{
    private int x;
    public SomeStruct x)
    {
        this.x = x;
    }
    public int CompareTo(object obj)
    {
        return this.x - ((SomeStruct) obj).x;
    }
}
```

```
SomeStruct s = new SomeStruct();
object o = new object();
int v = s.CompareTo(s); //boxing
v = s.CompareTo(o); //InvalidCastException
```

```
struct SomeStruct : IComparable
{
    private int x;
    public SomeStruct x)
    {
        this.x = x;
    }
    public int CompareTo(SomeStruct other)
    {
        return this.x - other.x;
    }
    int IComparable.CompareTo(object obj)
    {
        return CompareTo((SomeStruct) obj);
    }
}
```

```
SomeStruct s = new SomeStruct();
object o = new object();
int v = s.CompareTo(s);
v = s.CompareTo(o); //CTE
```

Явная и неявная реализация интерфейса

```
public sealed class SimpleType : IDisposable
{
    public void Dispose() { Console.WriteLine("Dispose"); }
}
```

EIMI-метод не является частью объектной модели типа

```
public sealed class SimpleType : IDisposable
{
    public void Dispose() { Console.WriteLine("public Dispose"); }
    void IDisposable.Dispose() { Console.WriteLine("IDisposable Dispose"); }
}
```

```
SimpleType st = new SimpleType();
// This calls the public Dispose method implementation
st.Dispose();
// This calls IDisposable's Dispose method implementation
IDisposable d = st;
d.Dispose();
```

Dispose
Dispose

```
SimpleType st = new SimpleType();
// This calls the public Dispose method implementation
st.Dispose();
// This calls IDisposable's Dispose method implementation
IDisposable d = st;
d.Dispose();
```

public Dispose
IDisposable Dispose

Определение свойств в интерфейсе

Свойства могут быть определены в интерфейсе

```
interface IPerson
{
    string Name { get; set; }
    int Age { get; }
    DateTime DateOfBirth { set; }
}
```

Можно использовать тот же синтаксис автоматических свойств

Не обязательно указывать оба аксессора

Нельзя указать модификатор доступа

Определение свойств в интерфейсе

```
interface IPerson
{
    string Name { get; set; }
    int Age { get; }
    DateTime DateOfBirth { set; }
}
```

```
class Person : IPerson
{
    string IPerson.Name
    {
        get {. . .}
        set {. . .}
    }
    int IPerson.Age
    {
        get {. . .}
    }
    DateTime IPerson.DateOfBirth
    {
        set {. . .}
    }
}
```

Неявная реализация интерфейса

```
class Person : IPerson
{
    public string Name
    {
        get {. . .}
        set {. . .}
    }
    public int Age
    {
        get {. . .}
    }
    public DateTime DateOfBirth
    {
        set {. . .}
    }
}
```

Явная реализация интерфейса

Определение индексаторов в интерфейсе

В интерфейсе можно указать индексатор, тогда любой реализующий интерфейс класс должен реализовать и этот индексатор

```
interface IEmployeeDatabase
{
    Employee this[string Name] { get; set; }
}
```

Реализовать индексатор в реализующем интерфейс классе можно явно или неявно

```
class EmployeeDatabase : IEmployeedatabase
{
    public Employee this[string Name]
    {
        get { ... return employee; }
        set { ... }
    }
}
```

Абстрактный класс

Абстрактный класс обеспечивает механизм вынесения в один класс общего кода, который несколько связанных иерархией наследования классов используют совместно

```
abstract class SalariedEmployee : Employee, ISalaried
{
    ...
    void PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}
```

Может содержать поля, методы, свойства и другие члены класса

```
SalariedEmployee employee = new SalariedEmployee();
```



CTE

Абстрактный класс vs интерфейс

Абстрактные классы	Интерфейсы
Могут содержать конструктор, который вызывается в классе-наследнике	Не могут содержать конструктор
Может быть так дополнен элементами, что это не влияет на его классы-наследники	Если в интерфейс помещаются дополнительные элементы, все классы, которые его реализуют, должны быть дополнены
Может хранить данные в полях	Не может хранить данных
Виртуальные элементы могут содержать базовую реализацию. Допустимы невиртуальные элементы	Все элементы являются чисто виртуальными и не включают реализацию
Любой класс может наследоваться от единственного абстрактного класса	Любой класс может реализовывать несколько интерфейсов
Класс-наследник может переопределить только некоторые элементы абстрактного класса	Класс, который реализует интерфейс, должен реализовать все элементы интерфейса
Наследование поддерживается только для классов	Интерфейс может быть реализован структурой
Может наследоваться от класса как абстрактного, так и не абстрактного, и от интерфейсов	Может наследоваться только от интерфейсов
Может объявлять не public члены	Не может объявлять не public члены

Абстрактный метод

Абстрактный метод является полезным, если реализация по умолчанию в абстрактном классе не имеет смысла и необходимо, чтобы производный класс обеспечивал свою собственную реализацию метода

```
abstract class SalariedEmployee : Employee, ISalariedEmployed
{
    abstract void PayBonus();
```

Абстрактный метод объявляется с помощью ключевого слова **abstract**

Абстрактный метод не содержит тела метода

```
interface IToken
{
    abstract string Name();
}
class CommentToken
{
    abstract string Name();
}
```



Только абстрактные классы могут содержать абстрактные члены

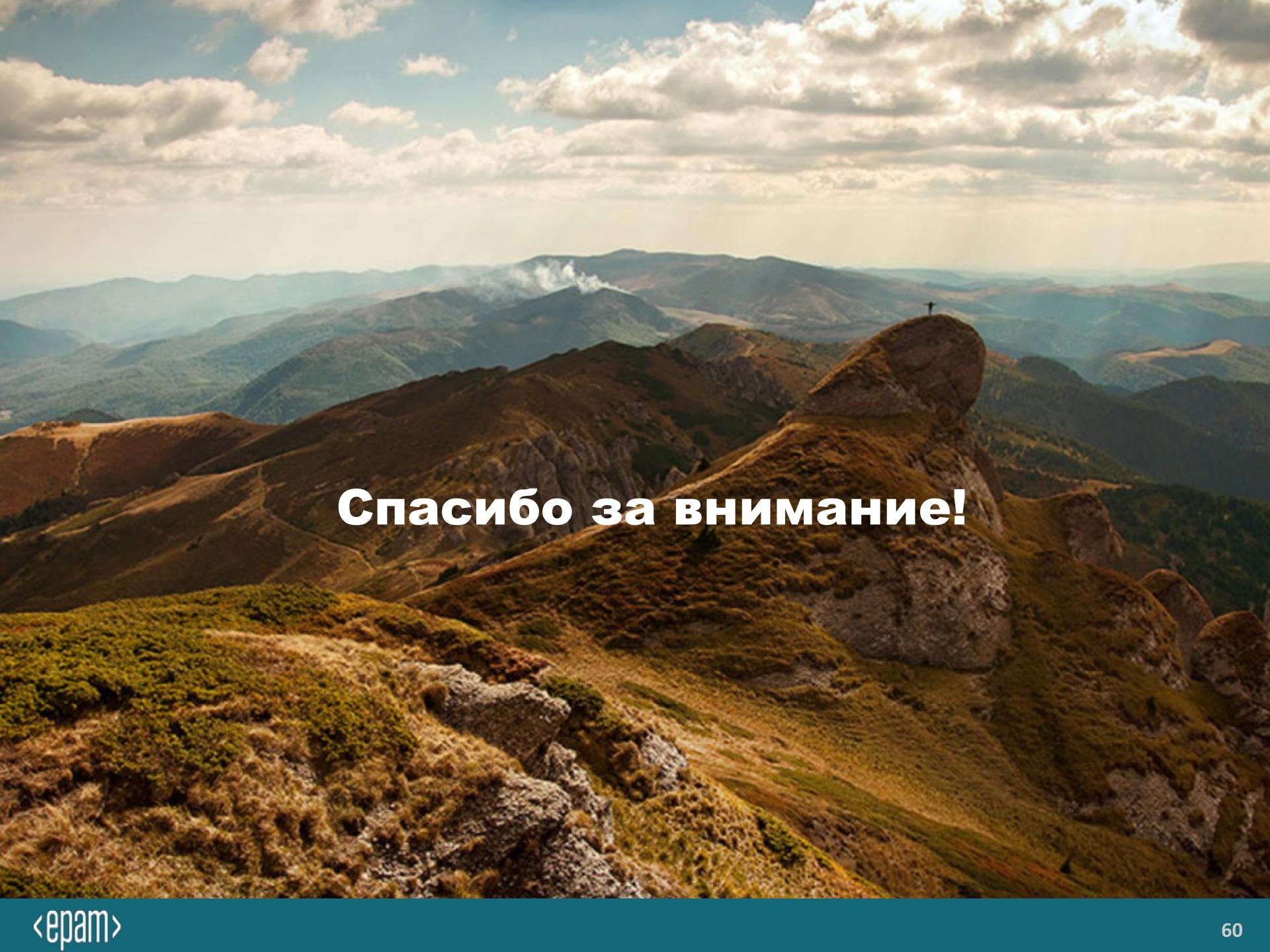
Абстрактный метод

Абстрактный метод необходимо переопределить (override) в производном классе

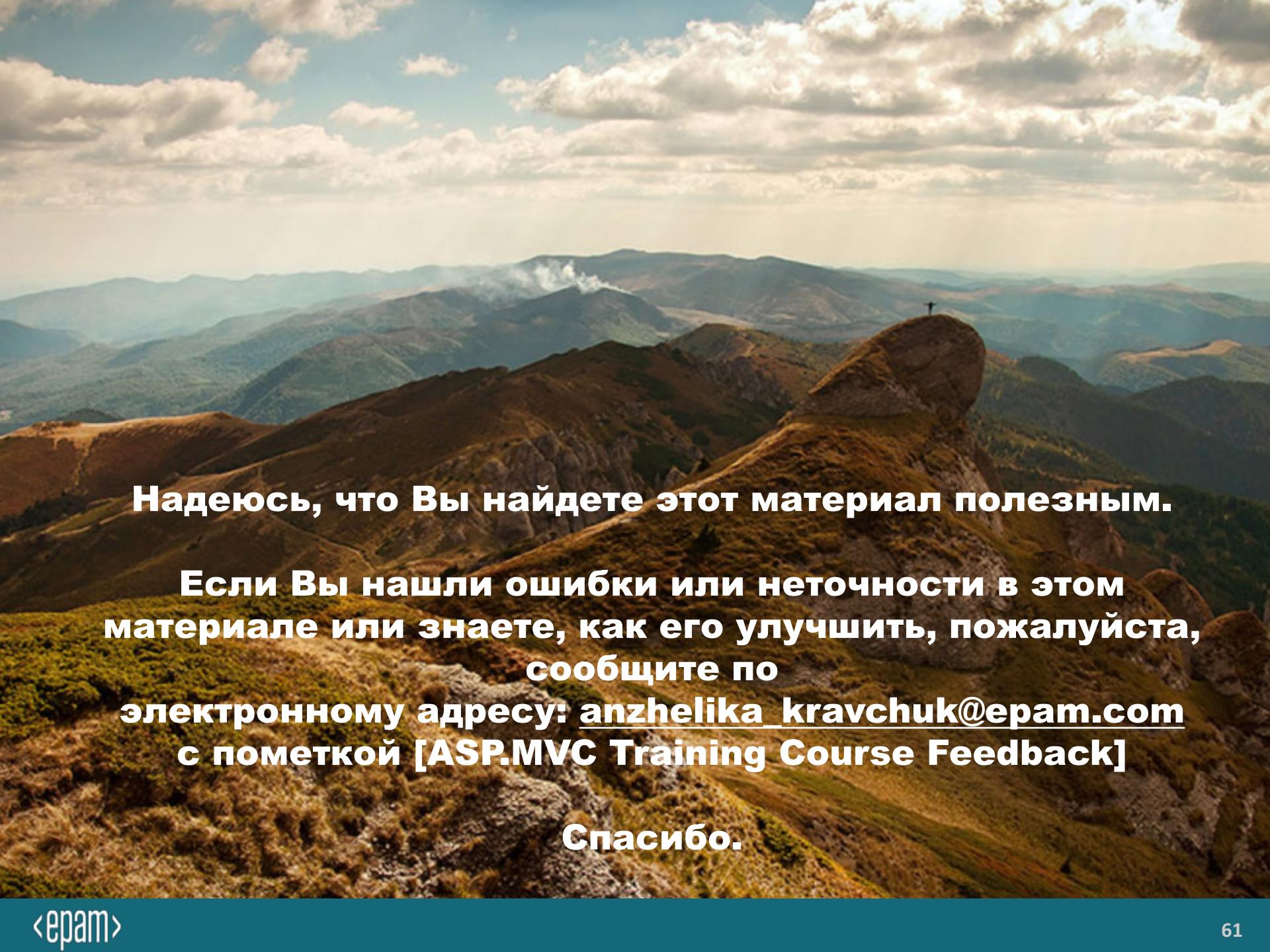
```
class CommentToken: Token
{
    public override string Name( ) { ... }
}
```

Абстрактный метод может переопределять виртуальный метод базового класса

```
class Token
{
    public virtual string Name( ) { ... }
}
abstract class Force: Token
{
    public abstract override string Name( );
}
```

A wide-angle photograph of a mountain range under a dramatic sky. In the foreground, rocky terrain and green slopes are visible. In the middle ground, a person stands on a prominent, rounded rock formation on a ridge. The background features multiple layers of mountains, with one emitting a small plume of white smoke or steam from its peak. The sky is filled with large, fluffy clouds.

Спасибо за внимание!



Надеюсь, что Вы найдете этот материал полезным.

**Если Вы нашли ошибки или неточности в этом
материале или знаете, как его улучшить, пожалуйста,
сообщите по**

**электронному адресу: anzhelika_kravchuk@epam.com
с пометкой [ASP.MVC Training Course Feedback]**

Спасибо.