**№ 5 Наследование, полиморфизм, абстрактные классы и интерфейсы**

**Курносенко Софья**

|  |  |
| --- | --- |
| **Вариант 8** | Телевизионная программа, Фильм, Новости, Худ. фильм, Мультфильм, Реклама, Режиссер. |

## ► Наследование

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

class Person

{

private string \_name;

public string Name

{

get { return \_name; }

set { \_name = value; }

}

public void Display()

{

Console.WriteLine(Name);

}

}

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

class Employee : Person

{

}

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

static void Main(string[] args)

{

Person p = new Person { Name = "Tom" };

p.Display();

p = new Employee { Name = "Sam" };

p.Display();

Console.Read();

}

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому вышеопределенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

Все классы по умолчанию могут наследоваться. Однако есть правила наследования:

* **Не поддерживается** **множественное наследование** (но не интерфейсов), класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать **тип доступа** к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производный класс может наследовать только от класса, который имеет модификатор public.

* Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

sealed class Admin

{

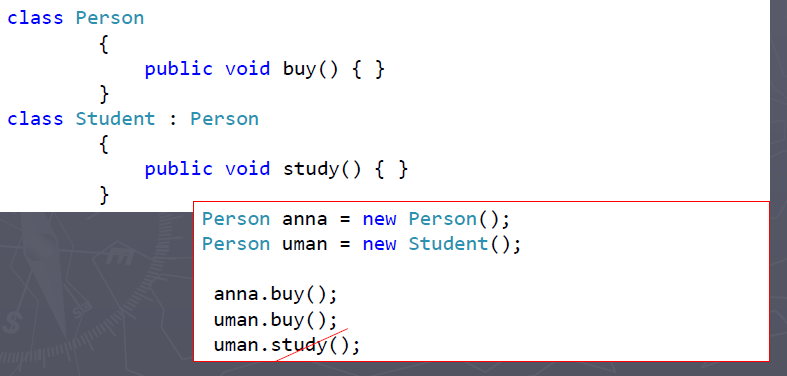
}

* **Нельзя унаследовать класс от статического класса**.
* **Наследуются** все свойства, методы, поля и т.д., которые есть в базовом классе

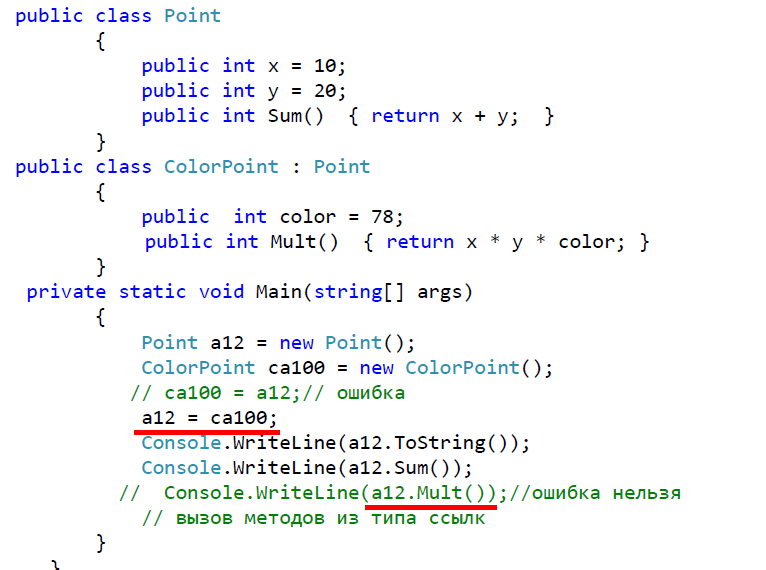
**Но**

**Не наследуются** конструкторы базового класса

* Производному классу доступны public internal, protected и protected internal члены базового класса (**private недоступны**)
* **Ссылке на объект базового класса** можно присвоить объект производного класса (*но доступны для такого объекта только методы и свойства, определенные в базовом классе*)



**Ссылки базового и производного классов**

****

### Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

class Employee : Person

{

public void Display()

{

Console.WriteLine(\_name);

}

}

Этот код не сработает и выдаст ошибку, так как переменная \_name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

class Employee : Person

{

public void Display()

{

Console.WriteLine(Name);

}

}

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

### Ключевое слово base

Теперь добавим в наши классы конструкторы:

class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public void Display()

{

Console.WriteLine(Name);

}

}

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

}

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

static void Main(string[] args)

{

Person p = new Person("Bill");

p.Display();

Employee emp = new Employee("Tom", "Microsoft");

emp.Display();

Console.Read();

}

Конструкторы не передаются производному классу при наследовании. И **если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами** (как в случае с базовым классом Person), **то в производном классе** **мы** **обязательно должны вызвать один из этих конструкторов** через ключевое слово base.

Например, из класса Employee уберем определение конструктора:

class Employee : Person

{

public string Company { get; set; }

}

В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

public Employee(string name, string company)

{

Name = name;

Company = company;

}

То есть в классе Employee через ключевое слово **base** надо явным образом вызвать конструктор класса Person:

public Employee(string name, string company)

: base(name)

{

Company = company;

}

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

class Person

{

// остальной код класса

// конструктор по умолчанию

public Person()

{

FirstName = "Tom";

Console.WriteLine("Вызов конструктора без параметров");

}

}

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно **неявно вызывался бы этот конструктор по умолчанию**. Например, следующий конструктор:

public Employee(string company)

{

Company = company;

}

Фактически был бы эквивалентен следующему конструктору:

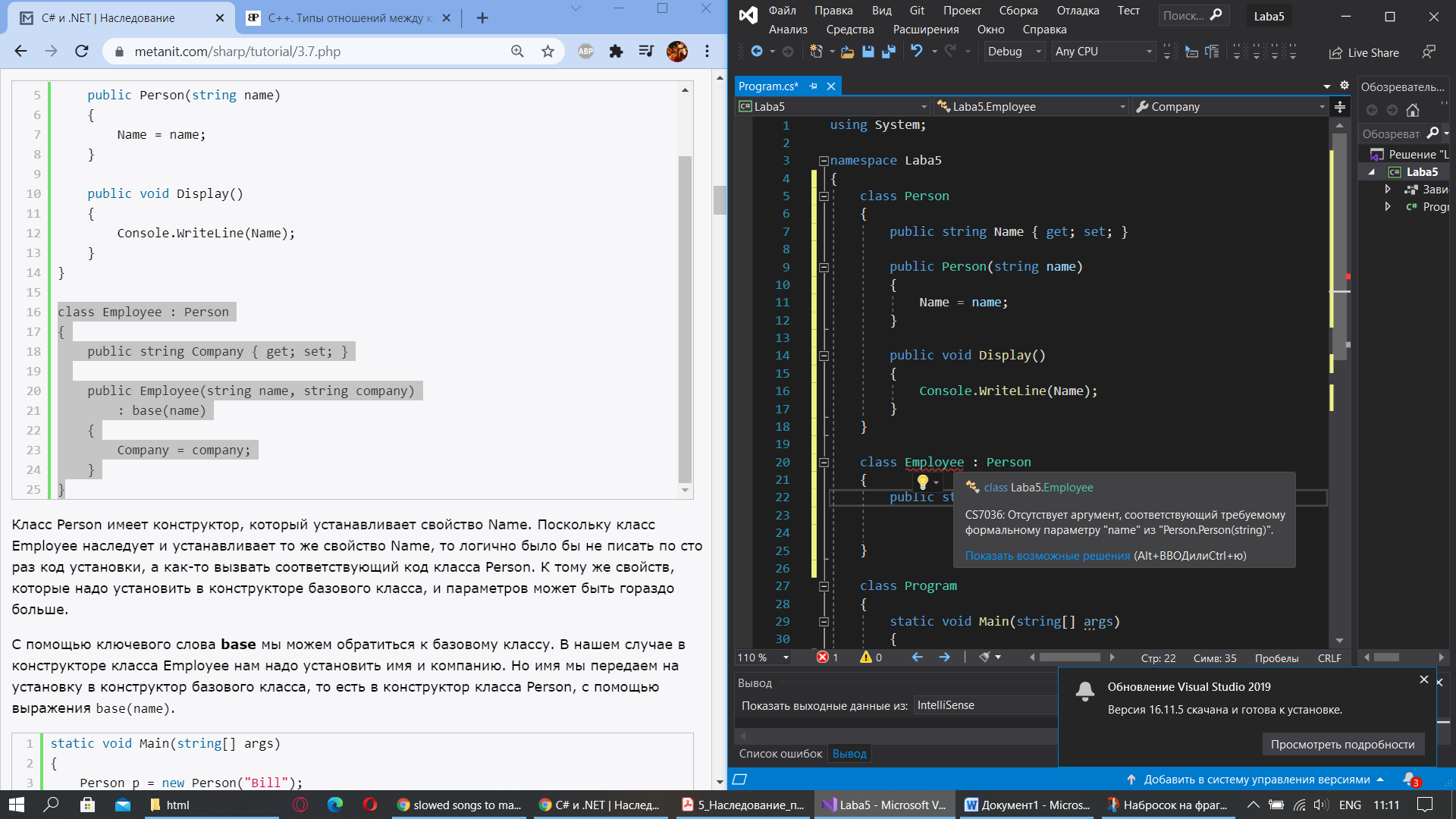
public Employee(string company)

:base()

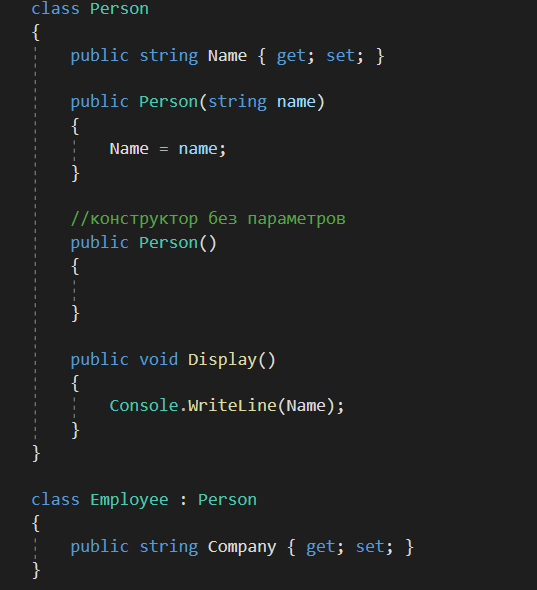
{

Company = company;

}



Конструктор базового класса без параметров в классе наследнике вызывается неявно по умолчанию:



### Порядок вызова конструкторов

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

class Person

{

string name;

int age;

public Person(string name)

{

this.name = name;

Console.WriteLine("Person(string name)");

}

public Person(string name, int age) : this(name)

{

this.age = age;

Console.WriteLine("Person(string name, int age)");

}

}

class Employee : Person

{

string company;

public Employee(string name, int age, string company) : base(name, age)

{

this.company = company;

Console.WriteLine("Employee(string name, int age, string company)");

}

}

При создании объекта Employee:

Employee tom = new Employee("Tom", 22, "Microsoft");

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)
6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

►**Между классами возможны два типа отношений:**

1. Отношение типа **is-a** (есть, является), при котором один класс является подвидом другого класса. При таком отношении один класс расширяет (детализирует) возможности другого класса. Расширение возможностей класса осуществляется благодаря использованию наследования.

2. Отношение, при котором существует взаимосвязь между двумя классами. Здесь выделяют два подвида взаимосвязи между классами:

2.1. Отношение типа **has-a** (класс содержит объект другого класса). В этом случае в классе объявляется один или несколько экземпляров другого класса. При данном отношении возможны два случая взаимодействия:

■ Первый случай, это когда объект (экземпляр), который объявлен в классе, не является составной частью класса (**агрегация**) и его использование не влияет на функциональную работу класса.

**Пример агрегации**

Класс Figures может содержать разное количество разных фигур (даже 0). По возможности к классу Figures могут быть добавлены массивы других фигур, например, Rectangle (прямоугольник). В любом случае, класс Figures будет полноценным функционально, значит, это есть агрегация.

// Класс треугольник

class Triangle

{

// Методы и поля класса Triangle

// ...

};

// Класс, реализующий окружность

class Circle

{

// Методы и поля класса Circle

// ...

};

// Класс, реализующий различные геометрические фигуры.

// Используется тип отношения - агрегация.

class Figures

{

Triangle tr[10]; // массив треугольников

unsigned int n\_tr; // количество треугольников в массиве tr

Circle cr[10]; // массив окружностей

unsigned int n\_cr; // количество окружностей в массиве cr

// Другие поля и методы класса

// ...

};

■ Второй случай, когда объект, который объявлен в классе, является составной частью этого класса (**композиция**).

**Пример композиции**

Класс Bike (Велосипед) содержит экземпляры классов Wheel (Колесо) и Saddle (седло), которые являются его составной частью. Велосипед не может быть без колес или седла, поэтому это есть композиция.

// Класс Колесо

class Wheel

{

// Поля и методы класса

// ...

};

// Класс Велосипед, содержит колеса, седло

// Класс седло

class Saddle

{

// Поля и методы класса

// ...

};

// Класс Велосипед, содержит объекты классов,

// которые есть частью Велосипеда (седло, колеса) - это композиция

class Bike

{

Saddle sd; // одно седло - часть, дополняющая велосипед

Wheel whl[2]; // два колеса - часть велосипеда

// Поля и методы класса Bike

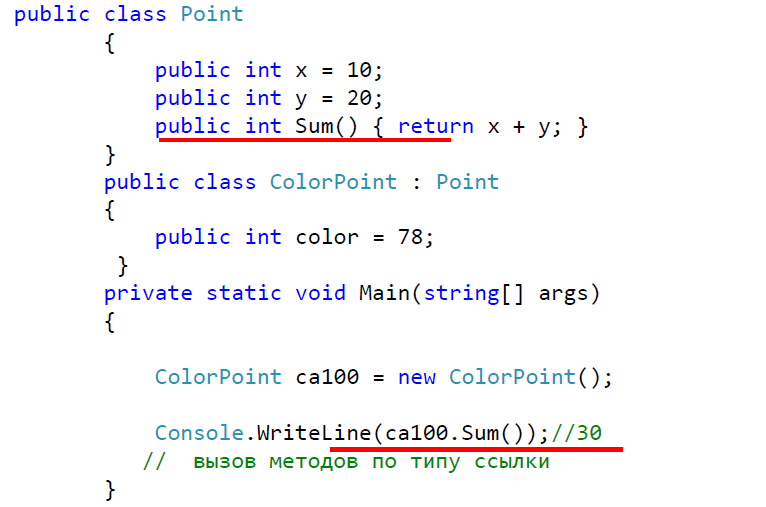
// ...

};

2.2. Отношение типа **uses** (класс «использует другой класс). В этом случае класс содержит программный код другого вложенного класса, к которому он имеет доступ.

**Стратегии наследования**

►Обычное наследование всех членов базового класса в классе-наследнике



►**Переопределение** членов базового класса в классе-наследнике (полиморфизм)

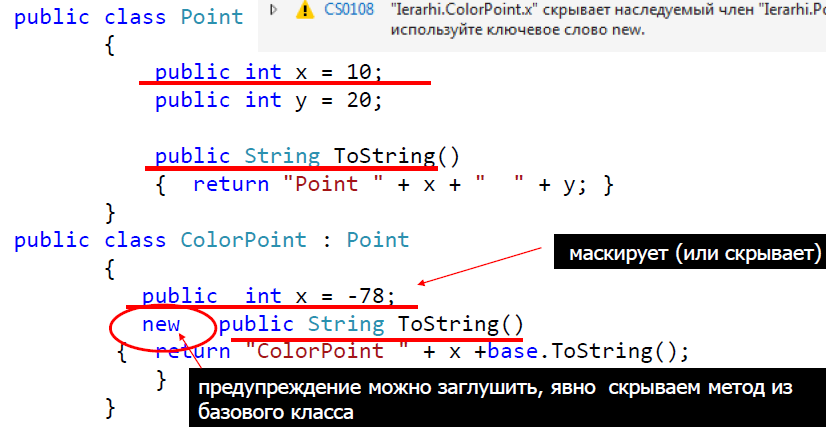
Полиморфный интерфейс в базовом классе – набор членов класса, которые могут быть переопределены в классе наследнике (то есть с модификатором virtual).



переопределение виртуального метода в производном классе:



►**Сокрытие** членов базового класса в классе-наследнике

****

**Мы все ещё можем обратиться к методу базового класса при помощи слова base: base.method\_name().**

**Задания**

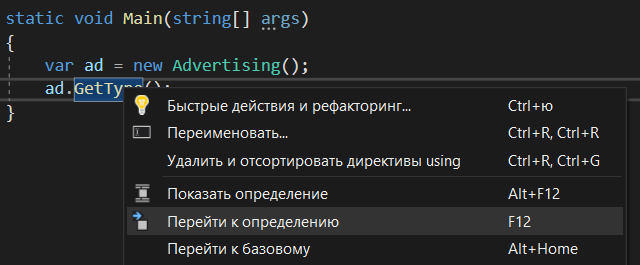
**1) Определить иерархию и композицию классов (в соответствии с вариантом), реализовать классы. Если необходимо расширьте по своему усмотрению иерархию для выполнения всех пунктов л.р.**

**Каждый класс должен иметь отражающее смысл название и информативный состав. При кодировании должны быть использованы соглашения об оформлении кода code convention.**

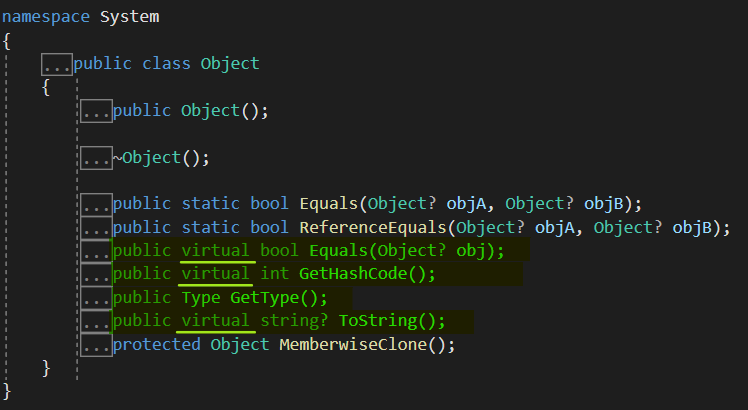
**В одном из классов переопределите все методы, унаследованные от Object.**

Переопределить метод GetType не получится, так как он объявлен *без модификатора virtual*, который и создает возможность переопределения методов.

Можно создать объект любого типа, вызвать у него один из унаследованных методов, далее правой кнопкой мыши по этому методу и “Перейти к определению”:



Видно, что GetType и ещё некоторые методы класса Object не virtual:



**КОД**

**класс:**

partial class Advertising

{

//конструктор

public Advertising(string Title,

string CustomerCompany,

string AdvertisingAgency,

float DurationMinutes,

DateTime PremiereDate)

{

this.Title = Title;

this.CustomerCompany = CustomerCompany;

this.AdvertisingAgency = AdvertisingAgency;

this.DurationMinutes = DurationMinutes;

this.PremiereDate = PremiereDate;

}

//методы

public string Title { get; set; }

public string CustomerCompany { get; set; }

public string AdvertisingAgency { get; set; }

public float DurationMinutes { get; set; }

public DateTime PremiereDate { get; set; }

}

**методы:**

partial class Advertising

{

//вывод информации о рекламе

public override string ToString()

{

return $"\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" +

$"Название : {Title}\n" +

$"Компания заказчик : {CustomerCompany}\n" +

$"Рекламное агентство : {AdvertisingAgency}\n" +

$"Продолжительность : {DurationMinutes}\n" +

$"Дата премьеры : {PremiereDate}\n" +

$"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

}

//---------------------------------------

//хеш-код

public override int GetHashCode()

{

//полученное значение типа DateTime при помощи метода ToString() приводим к строке вида

//"день.месяц.год часы:минуты:секунды" и далеее к массиву [день, месяц, ... , секунды]

string[] bday = (this.PremiereDate.ToString()).Split(' ', ':', '.');

int sum = 0;

//конвертируем значения из массива в тип int и складываем

foreach (string i in bday)

{

sum += Convert.ToInt32(i);

}

//теперь к полученной ранее сумме добавлем ASCII-коды символов компании заказчика

foreach (char i in this.CustomerCompany)

{

sum += (int)i;

}

//и символов рекламного агентства

foreach (char i in this.AdvertisingAgency)

{

sum += (int)i;

}

//полученная сумма, умноженная на ДЕНЬ премьеры, и есть хэш-код

return sum \* Convert.ToInt32(bday[0]);

}

//---------------------------------------

//сравнение объектов

public override bool Equals(object obj)

{

//типы сравниваемых объектов должны быть одинаковыми

if (obj.GetType() != this.GetType())

return false;

//obj приводим к виду объекта типа Advertising

var st = (Advertising)obj;

return (this.ToString() == obj.ToString());

}

}

**2) В проекте должен быть минимум один интерфейс и абстрактный класс. Использовать виртуальные методы и переопределение.**

## ► Виртуальные методы и свойства

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечаются модификатором **virtual**. Такие методы и свойства называют **виртуальными**.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь *тот же набор параметров и имя*, что и виртуальный метод в базовом классе.

Например, рассмотрим следующие классы:

class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public virtual void Display()

{

Console.WriteLine(Name);

}

}

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company) : base(name)

{

Company = company;

}

}

Здесь класс Person представляет человека. Класс Employee наследуется от Person и представляет сотрудника предприятия. Этот класс кроме унаследованного свойства Name имеет еще одно свойство - Company.

Чтобы сделать метод Display доступным для переопределения, этот метод определен с модификатором **virtual**. Поэтому мы можем переопределить этот метод, но можем и не переопределять. Допустим, нас устраивает реализация метода из базового класса. В этом случае объекты Employee будут использовать реализацию метода Display из класса Person:

static void Main(string[] args)

{

Person p1 = new Person("Bill");

p1.Display(); // вызов метода Display из класса Person

Employee p2 = new Employee("Tom", "Microsoft");

p2.Display(); // вызов метода Display из класса Person

Console.ReadKey();

}

Консольный вывод:

Bill

Tom

Но также можем переопределить виртуальный метод. Для этого в классе-наследнике определяется метод с модификатором **override**, который имеет то же самое имя и набор параметров:

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

public override void Display()

{

Console.WriteLine($"{Name} работает в {Company}");

}

}

Возьмем те же самые объекты:

static void Main(string[] args)

{

Person p1 = new Person("Bill");

p1.Display(); // вызов метода Display из класса Person

Employee p2 = new Employee("Tom", "Microsoft");

p2.Display(); // вызов метода Display из класса Employee

Console.ReadKey();

}

Консольный вывод:

Bill

Tom работает в Microsoft

Виртуальные методы базового класса определяют интерфейс всей иерархии, то есть в любом производном классе, который не является прямым наследником от базового класса, можно переопределить виртуальные методы. Например, мы можем определить класс Manager, который будет производным от Employee, и в нем также переопределить метод Display.

При переопределении виртуальных методов следует учитывать **ряд ограничений**:

* Виртуальный и переопределенный методы должны иметь один и тот же **модификатор доступа**. То есть если виртуальный метод определен с помощью модификатора public, то и переопределенный метод также должен иметь модификатор public.
* **Нельзя переопределить** **или объявить виртуальным *статический метод*.**

### Переопределение свойств

Также как и методы, можно переопределять свойства:

class Credit

{

**public virtual decimal Sum { get; set; }**

}

class LongCredit : Credit

{

private decimal sum;

**public override decimal Sum**

**{**

**get**

**{**

**return sum;**

**}**

**set**

**{**

**if (value > 1000)**

**{**

**sum = value;**

**}**

**}**

**}**

}

class Program

{

static void Main(string[] args)

{

LongCredit credit = new LongCredit { Sum = 6000 };

credit.Sum = 490;

Console.WriteLine(credit.Sum);

Console.ReadKey();

}

}

### Ключевое слово base

Кроме конструкторов, мы можем обратиться с помощью ключевого слова **base** к другим членам базового класса. В нашем случае вызов base.Display(); будет обращением к методу Display() в классе Person:

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

public override void Display()

{

**base**.Display();

Console.WriteLine($"работает в {Company}");

}

}

### Запрет переопределения методов

Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором **sealed**:

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

public override sealed void Display()

{

Console.WriteLine($"{Name} работает в {Company}");

}

}

При создании методов с модификатором sealed надо учитывать, что **sealed применяется в паре с override**, то есть только в переопределенных методах.

И в этом случае мы не сможем переопределить метод Display в классе, унаследованном от Employee.

Чтобы к методу класса применить ключевое слово sealed, нужно чтобы выполнялись следующие условия:

* класс должен быть унаследован от другого (базового) класса;
* в базовом классе должен быть объявлен метод, который переопределяется в данном (производном) классе. Этот метод базового класса также должен быть объявлен с ключевыми словами virtual или override (если базовый класс также унаследован).

Общая форма использования ключевого слова sealed для метода следующая:

class ClassName : BaseClassName

{

// ...

access\_modifier sealed override MethodName(parameters)

{

// ...

}

}

где

* ClassName – имя класса, в котором объявлен sealed-метод с именем MethodName;
* BaseClassName – имя класса, который есть базовым для класса ClassName. Запрет переопределения метода в классе предусматривает, что этот класс унаследован от другого базового класса, в котором объявлен одноименный переопределяемый метод. Поэтому, если для некоторого метода в классе нужно применить ключевое слово sealed, этот класс обязательно должен быть унаследован от другого класса;
* access\_modifier – один из модификаторов доступа public, protected, internal, protected internal. Модификатор доступа private не может быть применен к sealed-методу;
* MethodName – имя метода, который объявляется как sealed. Данный метод должен быть объявлен с ключевым словом override. Это значит, что данный метод переопределяет одноименный метод базового класса BaseClassName.

## ► Абстрактные классы и члены классов

## Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. При определении абстрактных классов используется ключевое слово abstract:

abstract class Human

{

public int Length { get; set; }

public double Weight { get; set; }

## }

Согласно синтаксису C# перед ключевым словом abstract может использоваться модификатор доступа public или internal.

## Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта. Например, следующим образом:

Human h = new Human();

Зачем нужны абстрактные классы? Допустим, в нашей программе для банковского сектора мы можем определить две основных сущности: клиента банка и сотрудника банка. Каждая из этих сущностей будет отличаться, например, для сотрудника надо определить его должность, а для клиента - сумму на счете. Соответственно клиент и сотрудник будут составлять отдельные классы Client и Employee. В то же время обе эти сущности могут иметь что-то общее, например, имя и фамилию, какую-то другую общую функциональность. И эту общую функциональность лучше вынести в какой-то отдельный класс, например, Person, который описывает человека. То есть классы Employee (сотрудник) и Client (клиент банка) будут производными от класса Person. И так как все объекты в нашей системе будут представлять либо сотрудника банка, либо клиента, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным:

abstract class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public void Display()

{

Console.WriteLine(Name);

}

}

class Client : Person

{

public int Sum { get; set; } // сумма на счету

public Client(string name, int sum)

: base(name)

{

Sum = sum;

}

}

class Employee : Person

{

public string Position { get; set; } // должность

public Employee(string name, string position)

: base(name)

{

Position = position;

}

}

Затем мы сможем использовать эти классы:

Client client = new Client("Tom", 500);

Employee employee = new Employee("Bob", "Apple");

client.Display();

employee.Display();

Или даже так:

Person client = new Client("Tom", 500);

Person employee = new Employee("Bob", "Операционист");

Но мы НЕ можем создать объект Person, используя конструктор класса Person:

Person person = new Person("Bill");

Однако, несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, конструктор в абстрактных классах тоже может играть важную роль, в частности, инициализировать некоторые общие для производных классов переменные и свойства, как в случае со свойством Name.

Таким образом, **абстрактный класс служит только для** порождения потомков, предоставляя базовый функционал для классов-наследников.

##### *Можно ли использовать конструкторы в абстрактном классе?*

В абстрактном классе допускается использование конструкторов. Хотя экземпляр абстрактного класса создать не удастся. Конструкторы в абстрактном классе необходимы для реализации начальной инициализации внутренних полей абстрактного класса (если такие есть). Эти конструкторы вызываются с помощью ключевого слова base в производных классах.

##### Можно ли в абстрактном классе объявлять неабстрактные элементы?

Да, можно. В абстрактном классе можно объявлять любые не абстрактные элементы, которые обычно объявляются в классах (поля данных, методы, свойства, индексаторы и т.д.).

##### Может ли абстрактный класс не содержать объявления абстрактных элементов?

(про абстрактные члены класса ниже!)

Да, может. Компилятор C# допускает объявление класса с ключевым словом abstract,в котором нет абстрактных элементов. Такое свойство полезно, когда на начальном этапе разработки программы еще до конца неизвестен перечень всех элементов класса, находящегося в вершине иерархии. Однако уже понятно, что при дальнейшей разработке (развитии) программы, этот класс будет содержать абстрактные элементы, а, значит, должен быть абстрактным.

**Например.** В начале разработки иерархии классов вводится абстрактный класс-заглушка. В данном примере объявляется класс с именем BaseAbstractClass

// Базовый класс в иерархии, еще пока что

// не содержит абстрактных элементов

abstract class BaseAbstractClass

{

// Здесь в будущем могут быть

// добавлены абстрактные элементы

}

##### Может ли абстрактный класс наследовать другой абстрактный класс? Может ли абстрактный класс наследовать другой не абстрактный класс?

Да. Абстрактный класс может наследовать как абстрактный, так и не абстрактный класс. В этом случае действуют правила наследования такие же, как при наследовании не абстрактных классов. Во всей цепи иерархии абстрактные и неабстрактные классы могут поочередно наследовать друг друга.

**Пример.** Нижеследующий пример демонстрирует свободу в использовании абстрактных классов в сочетании с неабстрактными.

namespace ConsoleApp5

{

// Абстрактный класс A\_Abstract - вершина иерархии

abstract class A\_Abstract

{

// некоторое внутреннее поле класса

public int a;

}

// Не абстрактный класс B - производный от A\_Abstract

class B : A\_Abstract

{

public int b;

}

// Абстрактный класс C\_Abstract - производный от неабстрактного класса B

abstract class C\_Abstract : B

{

public int c;

}

// Не абстрактный класс D

class D : C\_Abstract

{

public int d;

}

class Program

{

static void Main(string[] args)

{

// 1. Объявить экземпляры классов B, D.

// Экземпляры классов A\_Abstract, C\_Abstract запрещено объявлять

B objB = new B();

D objD = new D();

// A\_Abstract obj = new A\_Abstract(); - ошибка

// 2. Доступ к элементам из экземпляра objB

objB.a = 23;

objB.b = 33;

// 3. Доступ к элементам из экземпляра objD

objD.a = 77;

objD.b = 777;

objD.c = 7777;

objD.d = 7777;

}

}

}

### Абстрактные члены классов

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова **abstract**. Абстрактный метод – это метод, в котором отсутствует реализация. Абстрактный метод не содержит тела метода. Для абстрактного метода задается только его сигнатура: возвращаемый тип, имя и список параметров. В частности, абстрактными могут быть:

* Методы
* Свойства
* Индексаторы
* События

**Нельзя** объявить абстрактное поле (переменную) класса.

Общая форма объявления абстрактного метода:

*access\_modifier* abstract *return\_type* MethodName(*parameters*);

здесь

* *MethodName* – имя метода.
* *access\_modifier* – один из модификаторов доступа protected, public, internal, protected internal, **private использовать нельзя!**;
* *return\_type* – тип, возвращаемый методом;
* *parameters* – список параметров метода.

// Объявить тип делегата

delegate int MyDelegate(int a, int b);

abstract class A

{

// Абстрактный метод

public abstract void Print();

// Абстрактное свойство

public abstract int IntProp

{

get;

set;

}

// Абстрактный индексатор

public abstract int this[int index]

{

get;

set;

}

// Абстрактное событие

public abstract event MyDelegate MyEvent;

}

Абстрактные члены классов **не должны иметь модификатор private. При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе**. При переопределении в производном классе такой метод или свойство также объявляются с модификатором **override** (как и при обычном переопределении виртуальных методов и свойств). Также следует учесть, что **если класс имеет хотя бы один абстрактный метод** (или абстрактное свойство, индексатор, событие), **то этот класс должен быть определен как абстрактный**.

Абстрактные члены так же, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами мы говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

#### Абстрактные методы

**Например.**Пусть нужно разработать базовый класс Figure, что описывает некоторую фигуру. Из этого класса должны быть унаследованы более конкретные реализации фигур: треугольник (Triangle), круг (Circle), прямоугольник (Rectangle) как изображено на рисунке 1.

Пусть в иерархию классов нужно ввести метод Area(), определяющий площадь фигуры. В базовом классе Figure и производных классах, метод Area() использует механизм полиморфизма при его вызове.

Для любого из производных классов существует свой способ определения площади. Однако невозможно реализовать метод Area() базового класса Figure, поскольку, на этом этапе еще не известно для какой фигуры нужно вычислять площадь. Поэтому, реализация (тело) метода Area() не имеет смысла. Такой метод целесообразно объявлять как абстрактный с ключевым словом abstract. Автоматически, класс Figure также становится абстрактным.

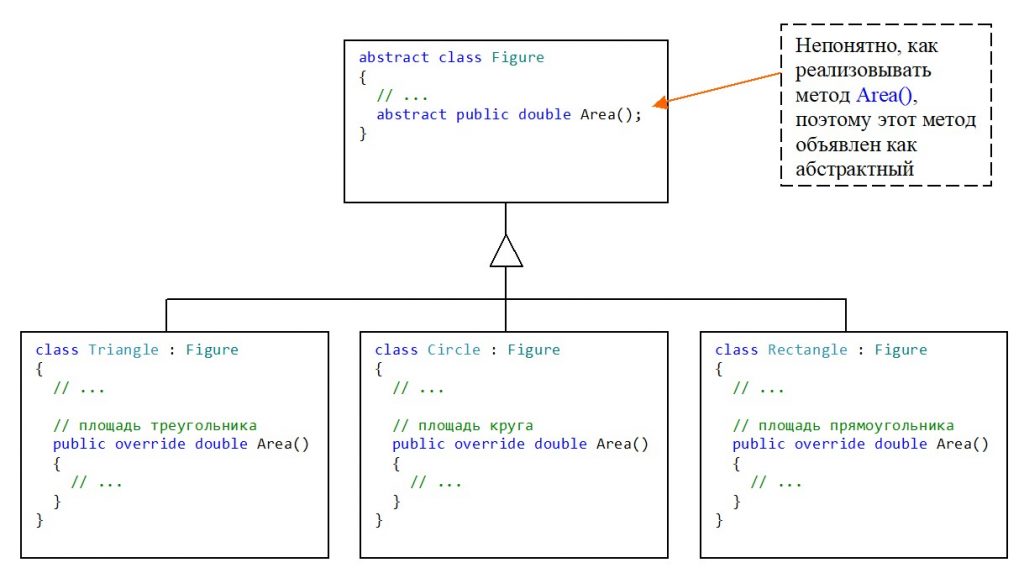
[](https://www.bestprog.net/wp-content/uploads/2020/04/02_02_02_11_08_01r.jpg)

Рисунок 1. Необходимость использования абстрактных классов.  
Метод Area() в классе Figure есть абстрактным

##### Какие ограничения накладываются на абстрактные элементы класса (методы, свойства)?

**Абстрактные элементы класса** – это элементы, которые объявлены с ключевым словом abstract. На абстрактные элементы (методы, свойства, индексаторы, события) накладываются следующие **ограничения**:

* абстрактный элемент не может быть private. Допускается объявлять абстрактный метод с любым другим модификатором доступа: public, protected, internal, protected internal;
* абстрактный элемент не должен иметь реализации;
* абстрактный элемент (элемент, обозначенный как abstract) не может быть обозначен ключевым словом virtual. Это естественно, поскольку для обеспечения использования полиморфизма достаточно ключевого слова abstract.
* абстрактные методы не используются со **static**

#### Отказ от реализации абстрактных членов

**Производный класс обязан реализовать все абстрактные члены базового класса**. Однако мы можем отказаться от реализации, но в этом случае производный класс также должен быть определен как абстрактный:

abstract class Person

{

public abstract string Name { get; set; }

}

abstract class Manager : Person

{

}

# Интерфейсы

## ► Определение интерфейсов

Интерфейс, в отличие от класса, содержит в себе только определение сигнатур метода, без их непосредственной реализации.

В свежих версиях языка уже появились реализации методов по умолчанию в интерфейсах, но применяется это достаточно редко и в очень специфичных случаях. Мы рассмотрим этот механизм позже.

Таким образом, **интерфейс** по своей сути **является** просто **коробкой, которая содержит в себе набор различных сигнатур методов**, и для того, чтобы **класс** смог применить на себя этот интерфейс, он **обязан явно содержать в себе непосредственные реализации всех методов интерфейса**.

За счет того, что класс непосредственно в себе содержит определения методов, CLR способна различать, к какому интерфейсу относится какой метод и избегать конфликтов. Благодаря этому мы можем *реализовывать* несколько интерфейсов, в отличие от множественного *наследования*. Но вот чем похожи наследование и реализация интерфейсов, так это возможностью подставлять экземпляры производного типа на место базовых. То есть наш любимый [полиморфизм](https://shwanoff.ru/oop/). Мы можем объявить переменную интерфейсного типа и в нее поместить экземпляр любого класса, который реализует этот интерфейс.

То есть интерфейс задает поведение для классов, которые будут его наследовать, или, правильнее сказать, реализовывать.

Интерфейсы позволяют определить требования к реализации (контракт).

### Определение интерфейса

Для определения интерфейса используется ключевое слово **interface**. Как правило, названия интерфейсов в C# начинаются с заглавной буквы **I**, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

interface имя

{

возвращаемый\_тип имя\_метода\_1(список\_параметров);

возвращаемый\_тип имя\_метода\_2(список\_параметров);

// ...

возвращаемый\_тип имя\_метода\_N(список\_параметров);

}

где имя — это конкретное имя интерфейса.

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать его имя после двоеточия и имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе:

class имя\_класса : имя\_интерфейса

{

// тело класса

}

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (*начиная с версии C# 8.0*)

**В интерфейсе не может быть конструкторов.**

**Интерфейсы не могут определять нестатические переменные**. Например, простейший интерфейс, который определяет все эти компоненты:

interface IMovable

{

// константа

const int minSpeed = 0; // минимальная скорость

// статическая переменная

static int maxSpeed = 60; // максимальная скорость

// метод

void Move(); // движение

// свойство

string Name { get; set; } // название

delegate void MoveHandler(string message); // определение делегата для события

// событие

event MoveHandler MoveEvent; // событие движения

}

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически **по умолчанию доступ** **public**, так как цель интерфейса – определение функционала для реализации его классом. Это касается также констант и статических переменных, которые в классах и структурах по умолчанию имеют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И, например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

static void Main(string[] args)

{

Console.WriteLine(IMovable.maxSpeed);

Console.WriteLine(IMovable.minSpeed);

}

► Но также, *начиная с версии C# 8.0*, мы можем **явно указывать модификаторы доступа у компонентов интерфейса**:

interface IMovable

{

public const int minSpeed = 0; // минимальная скорость

private static int maxSpeed = 60; // максимальная скорость

public void Move();

protected internal string Name { get; set; } // название

public delegate void MoveHandler(string message); // определение делегата для события

public event MoveHandler MoveEvent; // событие движения

}

► *Начиная с версии C# 8.0*, **интерфейсы поддерживают реализацию своих членов по умолчанию**. Это значит, что мы можем определить в интерфейсах *полноценные* методы и свойства, которые *имеют реализацию* как в обычных классах и структурах. Например, **определим** **реализацию метода** Move **по умолчанию**:

interface IMovable

{

// реализация метода по умолчанию

void Move()

{

Console.WriteLine("Walking");

}

}

При этом во время выполнения используются следующая схема вызова метода интерфейса:

* если реализация интерфейса (то есть класс-наследник) содержит вызываемый метод, то будет выполнен код из данной реализации;
* в противном случае будет выполнен код реализации по умолчанию.

С **реализацией свойств по умолчанию** в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее, реализацию по умолчанию для свойств мы тоже можем определять:

interface IMovable

{

// реализация свойства по умолчанию

// свойство только для чтения

int MaxSpeed { get { return 0; } }

}

Стоит отметить, что **если интерфейс имеет приватные методы и свойства** (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным):

interface IMovable

{

public const int minSpeed = 0; // минимальная скорость

private static int maxSpeed = 60; // максимальная скорость

// находим время, за которое надо пройти расстояние distance со скоростью speed

static double GetTime(double distance, double speed) => distance / speed;

static int MaxSpeed

{

get { return maxSpeed; }

set

{

if (value > 0) maxSpeed = value;

}

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine(IMovable.MaxSpeed);

IMovable.MaxSpeed = 65;

Console.WriteLine(IMovable.MaxSpeed);

double time = IMovable.GetTime(100, 10);

Console.WriteLine(time);

}

}

### Модификаторы доступа интерфейсов

Как и классы, интерфейсы по умолчанию имеют уровень доступа **internal**, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

public interface IMovable

{

void Move();

}

## ► Применение интерфейсов

## Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя

В конечном счете, интерфейс предназначен для реализации в классах и структурах. Например, возьмем следующий интерфейс IMovable:

interface IMovable

{

void Move();

}

Затем какой-нибудь класс или структура могут применить данный интерфейс:

// применение интерфейса в классе

class Person : IMovable

{

public void Move()

{

Console.WriteLine("Человек идет");

}

}

// применение интерфейса в структуре

struct Car : IMovable

{

public void Move()

{

Console.WriteLine("Машина едет");

}

}

При применении интерфейса, как и при наследовании, после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом **класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию**.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

Применение интерфейса в программе:

using System;

namespace HelloApp

{

interface IMovable

{

void Move();

}

class Person : IMovable

{

public void Move()

{

Console.WriteLine("Человек идет");

}

}

struct Car : IMovable

{

public void Move()

{

Console.WriteLine("Машина едет");

}

}

class Program

{

static void Action(IMovable movable)

{

movable.Move();

}

static void Main(string[] args)

{

Person person = new Person();

Car car = new Car();

Action(person);

Action(car);

Console.Read();

}

}

}

**Различия между абстрактными классами и интерфейсами**

Между интерфейсами и абстрактными классами были замечены следующие различия:

* интерфейс не может иметь конструкторов. Абстрактный класс может иметь конструкторы;
* интерфейс не может содержать нестатические поля данных. Абстрактный класс допускает использование нестатических полей данных;
* элементы интерфейса по умолчанию (без модификатора доступа) считаются public. В абстрактных классах элементы по умолчанию считаются private;
* производный класс может наследовать только один абстрактный базовый класс. В случае использования интерфейсов производный класс может наследовать любое количество интерфейсов. Таким образом, интерфейс – это альтернатива абстрактного класса, с помощью которой можно осуществить множественное наследование.

### Реализация интерфейсов по умолчанию

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Зачем это нужно? Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

class Program

{

static void Main(string[] args)

{

IMovable tom = new Person();

Car tesla = new Car();

tom.Move(); // Walking

tesla.Move(); // Driving

}

}

interface IMovable

{

void Move()

{

Console.WriteLine("Walking");

}

}

class Person : IMovable { }

class Car : IMovable

{

public void Move()

{

Console.WriteLine("Driving");

}

}

В данном случае интерфейс IMovable определяет реализацию по умолчанию для метода Move. Класс Person не реализует этот метод, поэтому он применяет реализацию по умолчанию в отличие от класса Car, который определяет свою реализацию для метода Move.

**!** Стоит отметить, что хотя для объекта класса Person мы можем вызвать метод Move - ведь класс Person применяет интерфейс IMovable, тем не менее, мы не можем написать так:

Person tom = new Person();

tom.Move(); // Ошибка - метод Move не определен в классе Person

### Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка С++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

myClass: myInterface1, myInterface2, myInterface3, ...

{ }

Рассмотрим на примере:

using System;

namespace HelloApp

{

interface IAccount

{

int CurrentSum { get; } // Текущая сумма на счету

void Put(int sum); // Положить деньги на счет

void Withdraw(int sum); // Взять со счета

}

interface IClient

{

string Name { get; set; }

}

class Client : IAccount, IClient

{

int \_sum; // Переменная для хранения суммы

public string Name { get; set; }

public Client(string name, int sum)

{

Name = name;

\_sum = sum;

}

public int CurrentSum { get { return \_sum; } }

public void Put(int sum) { \_sum += sum; }

public void Withdraw(int sum)

{

if (\_sum >= sum)

{

\_sum -= sum;

}

}

}

class Program

{

static void Main(string[] args)

{

Client client = new Client("Tom", 200);

client.Put(30);

Console.WriteLine(client.CurrentSum); //230

client.Withdraw(100);

Console.WriteLine(client.CurrentSum); //130

Console.Read();

}

}

}

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обратите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

### Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

// Все объекты Client являются объектами IAccount

IAccount account = new Client("Том", 200);

account.Put(200);

Console.WriteLine(account.CurrentSum); // 400

// Не все объекты IAccount являются объектами Client, необходимо явное приведение

Client client = (Client)account;

// Интерфейс IAccount не имеет свойства Name, необходимо явное приведение

string clientName = ((Client)account).Name;

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

|  |  |
| --- | --- |
| **Свойства** | **Интерфейс** |
| не может содержат | Нестатические поля, операции, конструкторы, деструкторы, типы |
| может содержать | абстрактные методы, обобщения свойства и индексаторы, а также события, делегаты, методы,  Статические поля, методы (с реализацией) и константы (с С# 8.0) |
| Доступность методов | **Publiс по умолчанию (не указывается )**  (при переопределении тоже public)  Могут быть private и protected |
| наследуются | С# поддерживается одиночное наследование для классов и множественное —для интерфейсов  (при реализации интерфейса нужно обеспечить точное совпадение)  Сначала всегда указывается имя базового класса, затем указывается интерфейс |
| Расширение интерфейса | Интерфейс наследуется интерфейсом |
| Имена | с прописной буквы I (не обязательно) |

**3) Сделайте один из классов sealed;**

**4) Добавьте в интерфейсы (интерфейс) и абстрактный класс одноименные методы.**

**Например,**

interface ICloneable

{

bool DoClone();

}

abstract class BaseClone

{

public abstract bool DoClone();

}

class UserClass : BaseClone, ICloneable

{

....

}

**Дайте в наследуемом классе им разную реализацию и вызовите эти методы.**

**► Явная реализация интерфейса**

Единственный класс или структура может реализовать любое количество интерфейсов. Из-за этого всегда существует вероятность реализации интерфейсов с членами, имеющими *идентичные имена*, и, следовательно, возникает необходимость в устранении конфликтов на уровне имен. При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса. В этом случае получается явная реализация члена интерфейса, или просто **явная реализация**.

interface IAction

{

void Move();

}

class BaseAction : IAction

{

void **IAction.Move()**

{

Console.WriteLine("Move in Base Class");

}

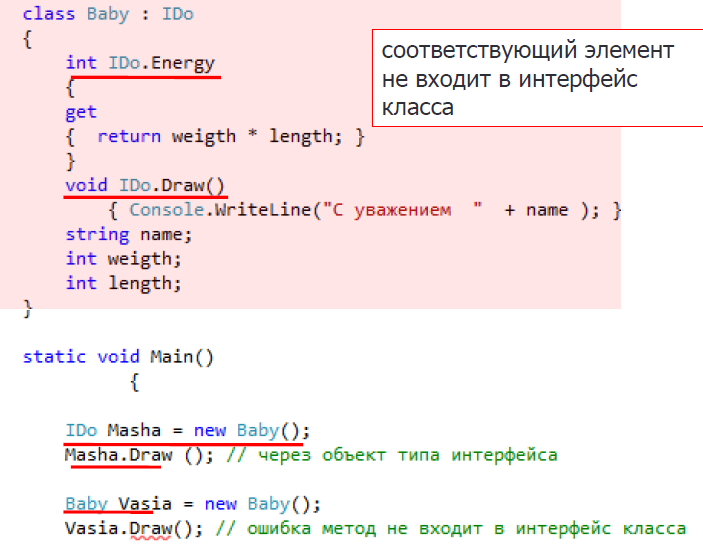
}

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он не стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.

Особенности явной реализации:

◦ метод, реализуемый явно, оказывается доступен не по ссылке типа класса, где метод реализован, а по ссылке типа интерфейса.

Поэтому напрямую через объект класса мы к методам обратиться не сможем:



Приведение к типу интерфейса:

static void Main(string[] args)

{

BaseAction action = new BaseAction();

((IAction)action).Move(); // необходимо приведение к типу IAction

// или так

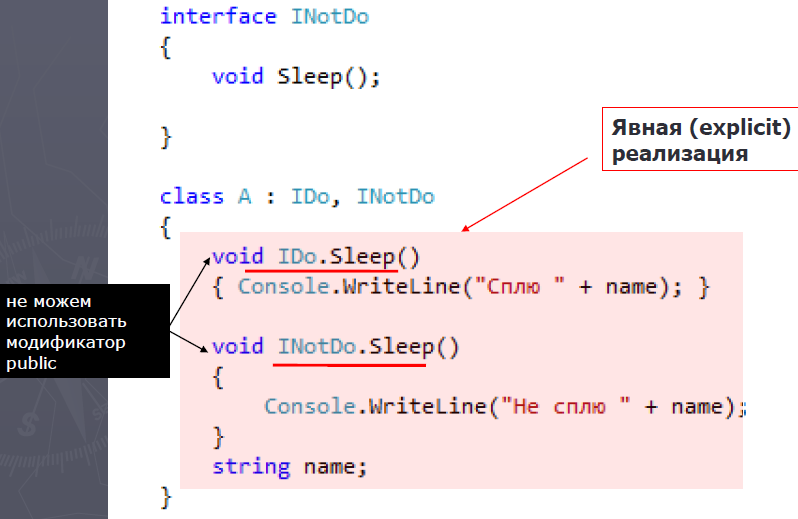
IAction action2 = new BaseAction();

action2.Move();

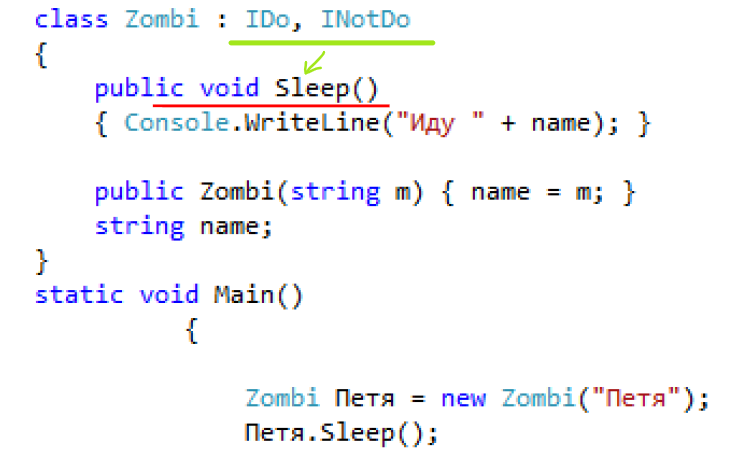
Console.ReadKey();

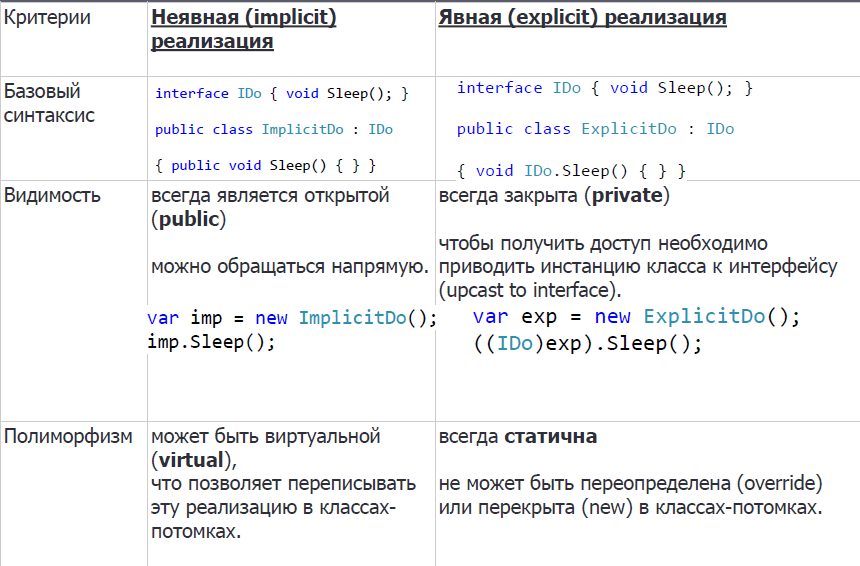
}

◦ с явно реализуемыми методами мы не можем использовать модификатор public



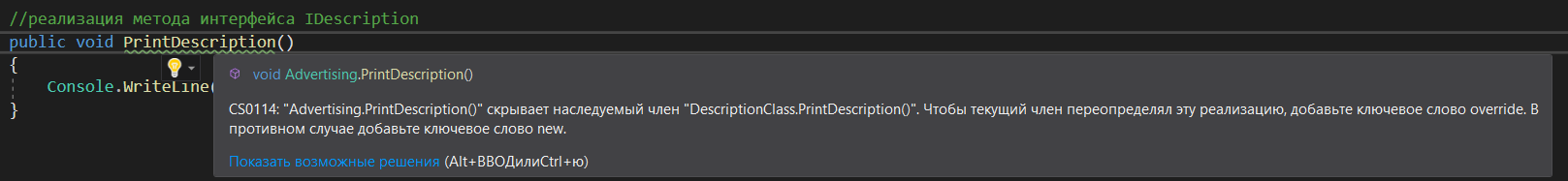
◦ не требуется разное поведение





**Задание:**

Метод по умолчанию определяется как переопределяемый. То есть среда думает, что мы переопределяем метод класса DescriptionClass, а не реализуем метод интерфейса IDescription:



**5) Написать демонстрационную программу, в которой создаются объекты различных классов. Поработать с объектами через ссылки на абстрактные классы и интерфейсы. В этом случае для идентификации типов объектов использовать операторы is или as.**

*Попытка приведения к типу, к которому привести нельзя, вызовет исключение на этапе выполнение:*

using System;

namespace asitis

{

public class Student

{ }

public class Teacher

{ }

class Program

{

static void Main(string[] args)

{

object[] MyObjects = new object[4];

MyObjects[0] = new Student();

MyObjects[1] = new Teacher();

MyObjects[2] = "Student";

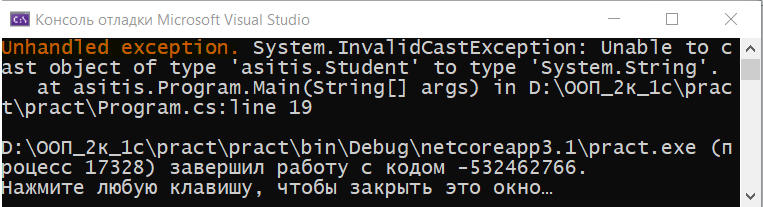
MyObjects[3] = "Teacher";

Console.WriteLine((string)MyObjects[0]);

}

}

}

**

Для проверки совместимости можно использовать try{}catch{} блок. Но синтаксис операторов as, is короче.

**► Операция is** возвращает булево значение, говорящее о том, **совпадает ли тип выражения (слева) с заданным типом данных (справа)**. *Оператор is никогда не генерирует исключение.*

int j = 123;

object boxed = j;

object obj = new Object();

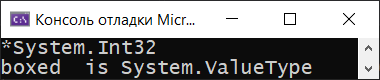
Boolean chekJ = boxed is int ; //true

Console.WriteLine("\*" + boxed.GetType());

Boolean checkObj = obj is int; //false

Console.WriteLine("boxed {0} System.ValueType",

boxed is ValueType ? " is" : "is not");



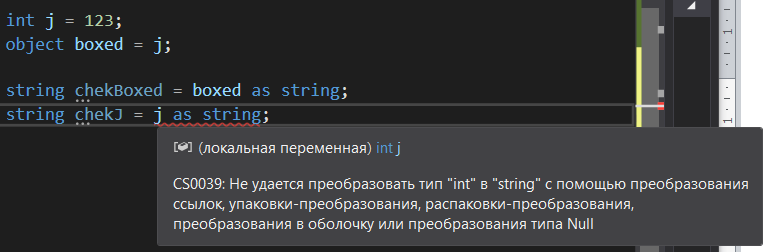
*Для того чтобы использовать оператор is, выражение должно иметь какое-то значение, иначе компилятор выдаст ошибку.*

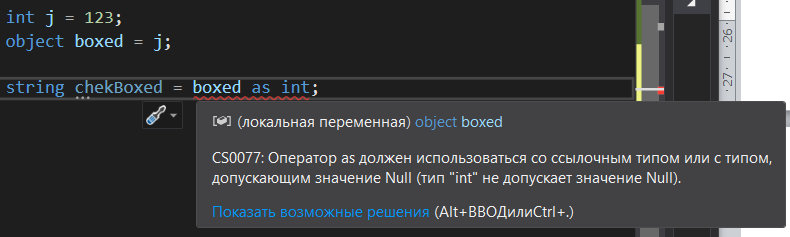
**► Операция as** **позволяет преобразовывать тип** в определенный **ссылочный тип** при помощи следующего синтаксиса:

*операнд as <тип>*

Оператор as должен использоваться со ссылочным типом или с типом, допускающим значение Null.

Тип выражения и тип, к которому оно преобразуется, должны быть ссылочными или допускать значение Null.





Оператор as никогда не генерирует исключение, но если не сравнить полученный результат с null (возвращается в случае невозможности преобразования) и попытаться работать с пустой ссылкой, возникнет исключение System.NullReferenceException.

**Выполняется:**

▪ Если <операнд>имеет тип, заданный в <тип>.

▪ Если <операнд>, может быть неявно преобразован в <тип>.

▪ Если операнд<операнд>, может быть упакован в<тип>.

using System;

namespace asitis

{

public class Student

{ }

public class Teacher

{ }

class Program

{

static void Main(string[] args)

{

object[] MyObjects = new object[4];

MyObjects[0] = new Student();

MyObjects[1] = new Teacher();

MyObjects[2] = "Student";

MyObjects[3] = "Teacher";

for (int i = 0; i < 4; i++)

{

string s = MyObjects[i] as string;

Console.Write($"Inspecting element: {MyObjects[i]}");

if (s == null)

{ Console.Write(" ->> Incompatible type"); }

else

{ Console.Write(" ->> Compatible type"); }

Console.WriteLine(", with string!");

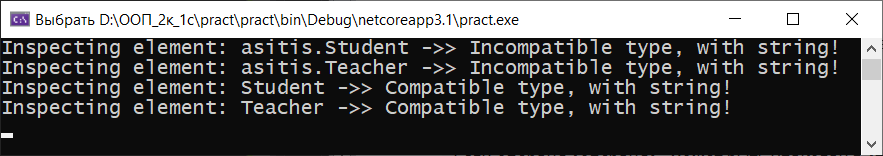
}

Console.ReadKey(); ;

}

}

}



*В случае если операнд не может быть приведен к указанному справа типу, оператор as вернет* ***null****.*

The is operator is used for only reference, boxing, and unboxing conversions, whereas the as operator is used only for nullable, reference, and boxing conversions.

**6) Во всех классах (иерархии) переопределить метод ToString(), который выводит информацию о типе объекта и его текущих значениях.**

**7) Создайте дополнительный класс Printer c полиморфным методом IAmPrinting( SomeAbstractClassOrInterface someobj). Формальным параметром метода должна быть ссылка на абстрактный класс или наиболее общий интерфейс в вашей иерархии классов. В методе iIAmPrinting определите тип объекта и вызовите ToString(). В демонстрационной программе создайте массив, содержащий ссылки на разнотипные объекты ваших классов по иерархии, а также объект класса Printer и последовательно вызовите его метод IAmPrinting со всеми ссылками в качестве аргументов.**

# Dynamic Language Runtime

## ► DLR в C#. Ключевое слово dynamic

Хотя C# относится к статически типизированным языкам, в последних версиях языка были добавлены некоторые динамические возможности. Так, начиная с .NET 4.0 была добавлена новая функциональность под названием DLR (Dynamic Language Runtime). DLR представляет среду выполнения динамических языков, например, таких языков как IronPython и IronRuby.

Чтобы понять значение данного нововведение, нужно осознавать разичие между языками со статической и динамической типизицией. В языках со статической типизацией выявление всех типов и их членов - свойств и методов происходит на этапе компиляции. А в динамических языках системе ничего не известно о свойствах и методах типов вплоть до выполнения.

Благодаря этой среде DLR C# может создавать динамические объекты, члены которых выявляются на этапе выполнения программы, и использовать их вместе с традиционными объектами со статической типизацией.

Ключевым моментом использования DLR в C# является применение типов **dynamic**. Это ключевое слово позволяет опустить проверку типов во время компиляции. Кроме того, объекты, объявленные как dynamic, могут в течение работы программы менять свой тип. Например:

class Program

{

static void Main(string[] args)

{

dynamic x = 3; // здесь x - целочисленное int

Console.WriteLine(x);

x = "Привет мир"; // x - строка

Console.WriteLine(x);

x = new Person() { Name = "Tom", Age = 23 }; // x - объект Person

Console.WriteLine(x);

Console.ReadLine();

}

}

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public override string ToString()

{

return Name + ", " + Age.ToString();

}

## }

Несмотря на то, что переменная x меняет тип своего значения несколько раз, данный код будет нормально работать. В этом использование типов dynamic отличается от применения ключевого слова **var**. Для переменной, объявленной с помощью ключевого слова var, тип выводится во время компиляции и затем во время выполнения больше не меняется.

Также можно найти общее между использованием dynamic и типом object. Если в предыдущем примере мы заменим dynamic на object: object x = 3;, то результат будет тот же. Однако и тут есть различия. Например:

object obj = 24;

dynamic dyn = 24;

obj += 4; // так нельзя

## dyn += 4; // а так можно

На строке obj += 4; мы увидим ошибку, так как операция += не может быть применена к типам object и int. С переменной, объявленной как dynamic, это пройдет, так как ее тип будет известен только во время выполнения.

Еще одна отличительная особенность использования dynamic состоит в том, что это ключевое слово применяется не только к переменным, но и к свойствам и методам. Например:

class Person

{

public string Name { get; set; }

public dynamic Age { get; set; }

// выводим зарплату в зависимости от переданного формата

public **dynamic** getSalary(dynamic value, string format)

{

if (format == "string")

{

return value + " рублей";

}

else if (format == "int")

{

return value;

}

else

{

return 0.0;

}

}

public override string ToString()

{

return Name + ", " + Age.ToString();

}

## }

В классе Person определено динамическое свойство Age, поэтому при задании значения этому свойству мы можем написать и person.Age=33, и person.Age="тридцать три". Оба варианта будут правильными.

Также есть метод getSalary, возвращающий значение dynamic. Например, в зависимости от параметра мы можем вернуть или строковое представление суммы дохода или численное. Также метод принимает dynamic в качестве параметра. Таким образом, мы можем передать в качестве значения дохода как целое, так и дробное число. Посмотрим на конкретное применение:

dynamic person1 = new Person() { Name = "Том", Age = 27 };

Console.WriteLine(person1);

Console.WriteLine(person1.getSalary(28.09, "int"));

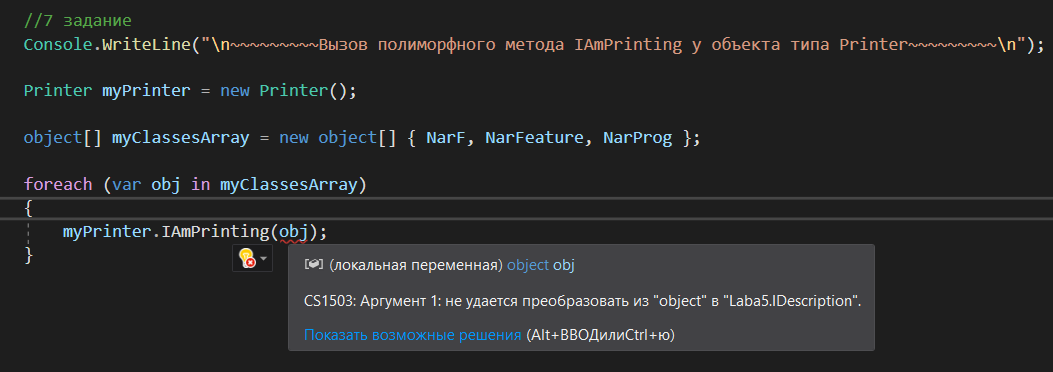
dynamic person2 = new Person() { Name = "Билл", Age = "Двадцать два года" };

Console.WriteLine(person2);

## Console.WriteLine(person2.getSalary(30, "string"));

**Задание:**

Сначала хотела так, но тип переменной obj меняется по ходу выполнения программы, так что var не подходит, ибо типа такой переменной определяется на этапе компиляции.

**

Подходит тип dynamic:

//7 задание

Console.WriteLine("\n~~~~~~~~~Вызов полиморфного метода IAmPrinting у объекта типа Printer~~~~~~~~~\n");

Printer myPrinter = new Printer();

object[] myClassesArray = new object[] { NarF, NarFeature, NarProg };

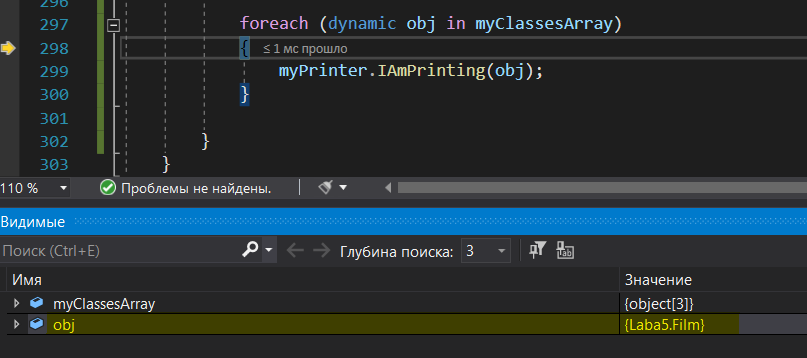
foreach (**dynamic** obj in myClassesArray)

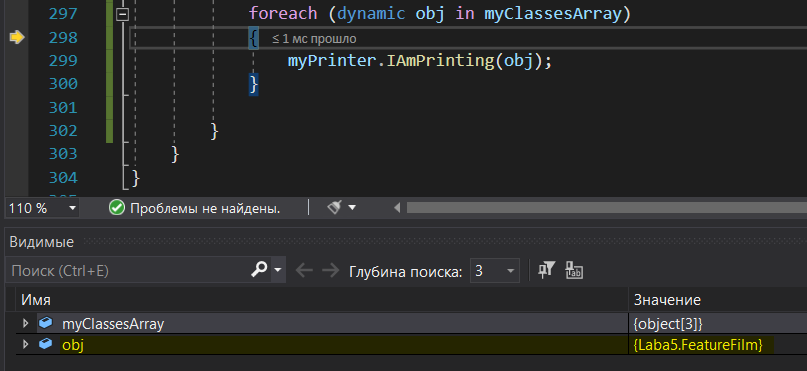
{

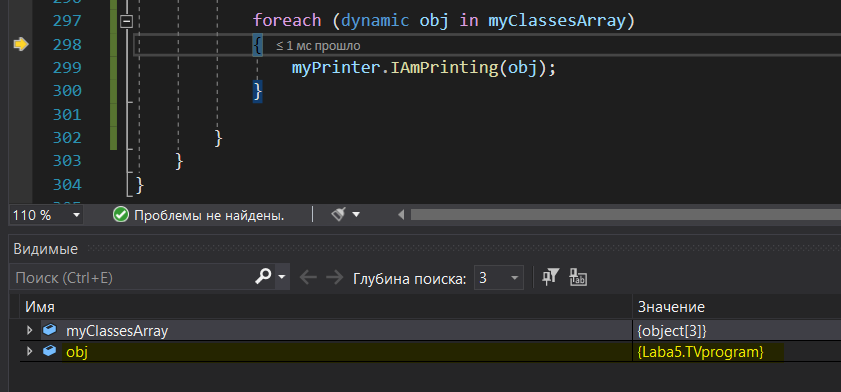
myPrinter.IAmPrinting(obj);

}

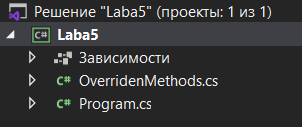
Так как тип переменной **меняется** на протяжении выполнения программы, что для var недопустимо:







**КОД ВСЕЙ ЛАБЫ**

****

**Program.cs**

using System;

namespace Laba5

{

//------------------------------------------------------

//абстрактный класс

abstract class Person

{

public Person(string name, string surname)

{

Name = name;

Surname = surname;

}

public string Name { get; set; }

public string Surname { get; set; }

//абстрактный метод

public abstract void Print();

}

//класс-наследник абстрактного класса Person

class Director : Person

{

public Director(string name, string surname) : base(name, surname) { }

//обязательное переопределение абстрактного метода родительского класса

public override void Print()

{

Console.WriteLine("Имя режиссёра : " + this.Name + ", фамилия : " + this.Surname);

}

public void SaySmth()

{

Console.WriteLine("Я режиссёр!");

}

}

//класс-наследник абстрактного класса Person

class Presenter : Person

{

public Presenter(string name, string surname) : base(name, surname) { }

public override void Print()

{

Console.WriteLine("Имя ведущего :" + this.Name + ", фамилия : " + this.Surname);

}

}

//------------------------------------------------------

//------------------------------------------------------

//интерфейс для вывода описания произведения

interface IDescription

{

void PrintDescription();

}

//------------------------------------------------------

//абстрактный класс к заданию 4, который содержит метод с тем же названием, что и метод интерфейса IDescription

abstract class DescriptionClass

{

public abstract void PrintDescription();

}

//------------------------------------------------------

//------------------------------------------------------

//Фильм может быть игровым и неигровым.

//Неигровое (документальное) кино - съёмки подлинных событий и лиц.

//Игровое - произведение, имеющее в основе сюжет, воплощённый в сценарии и интерпретируемый режиссёром, который создаётся с помощью

//актёрской игры, операторского и прочих искусств.

class Film : IDescription

{

//свойства

public string Title { get; set; }

public string Genres { get; set; }

public Director Director { get; set; }

public int DurationMinutes { get; set; }

public DateTime PremiereDate { get; set; }

public string Description { get; set; }

//конструктор

public Film(string title)

{

Title = title;

}

//реализация метода интерфейса

public void PrintDescription()

{

Console.WriteLine($"Описание \"{Title}\" : {Description}");

}

public override string ToString()

{

return $"-> Тип : {GetType()}, " +

$"Название: {Title}// Жанры: {Genres}// Режиссёр: {Director}" +

$" //Продолжительность: {DurationMinutes}// Дата премьеры: {PremiereDate}// Описание: {Description}";

}

}

//------------------------------------------------------

//------------------------------------------------------

//Художественный фильм - то же, что игровой фильм.

class FeatureFilm : Film

{

public FeatureFilm(string title) : base(title) { }

//принимает строку со списком "Имя Фамилия" актеров

public string MainActors { get; set; }

public override string ToString()

{

return $"-> Тип : {GetType()}, " +

$"Название: {Title}// В главных ролях: {MainActors}";

}

}

//Мультфильм

class Cartoon : Film

{

public Cartoon(string title) : base(title) { }

//принцип создания мультфильма: Пластилиновый, Рисованный, Кукольный, Компьютерный, Художественный

public string WayToCreate { get; set; }

public override string ToString()

{

return $"-> Тип : {GetType()}, " +

$"Название: {Title}// Принцип создания мультфильма: {WayToCreate}";

}

}

//Телевизионная программа

class TVprogram : Film

{

public TVprogram(string title) : base(title) { }

//ведущий

public string Presenter { get; set; }

public override string ToString()

{

return $"-> Тип : {GetType()}, " +

$"Название: {Title}// Ведущий: {Presenter}";

}

}

//------------------------------------------------------

//------------------------------------------------------

//sealed - значит от класса News нельзя наследовать

sealed class News

{

//название ТВ канала

public string ChannelName{ get; set; }

public string Presenter { get; set; }

}

//------------------------------------------------------

//------------------------------------------------------

partial class Advertising : DescriptionClass, IDescription

{

//конструктор

public Advertising(string title)

{

Title = title;

}

//методы

public string Title { get; set; }

public string CustomerCompany { get; set; }

public string AdvertisingAgency { get; set; }

public float DurationMinutes { get; set; }

public DateTime PremiereDate { get; set; }

public string Description { get; set; }

//реализация метода интерфейса IDescription

void IDescription.PrintDescription()

{

Console.WriteLine($"(интерфейс) Описание \"{Title}\" : {Description}");

}

public override void PrintDescription()

{

Console.WriteLine($"(абстрактный класс) Описание \"{Title}\" : {Description}");

}

}

//------------------------------------------------------

class Printer

{

public void IAmPrinting(IDescription obj)

{

Console.WriteLine(obj.ToString());

}

}

class Program

{

static void Main(string[] args)

{

//задание 4, вызов одноименного метода

Advertising adv1 = new Advertising("adv1");

adv1.PrintDescription(); //вызовется переопределение абстрактного метода

((IDescription)adv1).PrintDescription(); //вызовется реализация метода интерфейса

Console.WriteLine("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

//задание 5

//~~~~~~~~~~~~~~~~~~~~Ссылка типа абстрактного класса~~~~~~~~~~~~~~~~~~~~

//Создаем объект типа Director, ссылку на объект передаем в переменную типа Person

//Тип Person является абстрактным классом, от которого Director наследуется

//Таким образом у нас ссылка типа Person, которая хранит данные типа Director

Person P\_NarDir = new Director("Наруто", "Узумаки");

//Получение данных

//P\_NarDir.SaySmth(); //класс Director содержит метод SaySmth, ссылка типа Person не имеет доступа к этому методу (ошибка на этапе компиляции)

//явное преобразование

Director D\_NarDir = P\_NarDir as Director;

//Получение данных

D\_NarDir.SaySmth(); //таким образом тип ссылки ограничивает область доступных данных и методов

if (D\_NarDir != null)

{

//Переменная, типа Director, записанная по ссылке Person, может быть записана по ссылке типа Director при явном преобразовании к этому типу

Console.WriteLine("\n+ Преобразование ссылки типа абстрактного класса в ссылку типа класса, " +

"объект которого хранила ссылка типа абстрактного класса, удалось!");

}

else

{

Console.WriteLine("\n- Преобразование ссылки типа абстрактного класса в ссылку типа класса, " +

"объект которого хранила ссылка типа абстрактного класса, НЕ удалось!");

}

bool P\_NarDir\_is\_Dir = P\_NarDir is Director;

if (P\_NarDir\_is\_Dir)

{

Console.WriteLine("\nPerson P\_NarDir = new Director(\"Наруто\", \"Узумаки\"); =>" +

" P\_NarDir типа Director, Person - абстрактный класс");

}

else

{

Console.WriteLine("\nPerson P\_NarDir = new Director(\"Наруто\", \"Узумаки\"); =>" +

" P\_NarDir НЕ типа Director, Person - абстрактный класс");

}

//~~~~~~~~~~~~~~~~~~~~Ссылка типа интерфейса~~~~~~~~~~~~~~~~~~~~

IDescription Inter\_NarAdv = new Advertising("Йоу собаки");

//Опять же, доступны только методы реализованного интерфейса (в нашем случае метод PrintDescription интерфейса IDescription)

//Inter\_NarAdv.Title; //ошибка

Advertising Adv\_NarAdv = Inter\_NarAdv as Advertising;

if (Adv\_NarAdv != null)

{

Console.WriteLine("\n+ Преобразование ссылки типа интерфейса в ссылку типа класса, " +

"объект которого хранила интерфейсная ссылка, удалось!");

}

else

{

Console.WriteLine("\n- Преобразование ссылки типа интерфейса в ссылку типа класса, " +

"объект которого хранила интерфейсная ссылка, НЕ удалось!");

}

bool Inter\_NarAdv\_is\_Adv = Inter\_NarAdv is Advertising;

if (Inter\_NarAdv\_is\_Adv)

{

Console.WriteLine("\nIDescription Inter\_NarAdv = new Advertising(\"Йоу собаки\"); =>" +

" Inter\_NarAdv типа Advertising, nIDescription - интерфейс");

}

else

{

Console.WriteLine("\nPerson P\_NarDir = new Director(\"Наруто\", \"Узумаки\"); =>" +

" Inter\_NarAdv НЕ типа Advertising, nIDescription - интерфейс");

}

//задание 6

Film NarF = new Film("Гав!");

FeatureFilm NarFeature = new FeatureFilm("Анимационные псы");

TVprogram NarProg = new TVprogram("Поиск покинувших деревню");

Console.WriteLine("\n~~~~~~~~~Вызов переопределенного метода ToString~~~~~~~~~\n");

Console.WriteLine(NarF.ToString());

Console.WriteLine(NarFeature.ToString());

Console.WriteLine(NarProg.ToString());

//7 задание

Console.WriteLine("\n~~~~~~~~~Вызов полиморфного метода IAmPrinting у объекта типа Printer~~~~~~~~~\n");

Printer myPrinter = new Printer();

object[] myClassesArray = new object[] { NarF, NarFeature, NarProg };

foreach (dynamic obj in myClassesArray)

{

myPrinter.IAmPrinting(obj);

}

}

}

}

**OverridenMethods.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace Laba5

{

partial class Advertising

{

//вывод информации о рекламе

public override string ToString()

{

return $"\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" +

$"Название : {Title}\n" +

$"Компания заказчик : {CustomerCompany}\n" +

$"Рекламное агентство : {AdvertisingAgency}\n" +

$"Продолжительность : {DurationMinutes}\n" +

$"Дата премьеры : {PremiereDate}\n" +

$"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n";

}

//---------------------------------------

//хэш-код

public override int GetHashCode()

{

//полученное значение типа DateTime при помощи метода ToString() приводим к строке вида

//"день.месяц.год часы:минуты:секунды" и далеее к массиву [день, месяц, ... , секунды]

string[] bday = (this.PremiereDate.ToString()).Split(' ', ':', '.');

int sum = 0;

//конвертируем значения из массива в тип int и складываем

foreach (string i in bday)

{

sum += Convert.ToInt32(i);

}

//теперь к полученной ранее сумме добавлем ASCII-коды символов компании заказчика

foreach (char i in this.CustomerCompany)

{

sum += (int)i;

}

//и символов рекламного агентства

foreach (char i in this.AdvertisingAgency)

{

sum += (int)i;

}

//полученная сумма, умноженная на ДЕНЬ премьеры, и есть хэш-код

return sum \* Convert.ToInt32(bday[0]);

}

//---------------------------------------

//сравнение объектов

public override bool Equals(object obj)

{

//типы сравниваемых объектов должны быть одинаковыми

if (obj.GetType() != this.GetType())

return false;

//obj приводим к виду объекта типа Advertising

var st = (Advertising)obj;

return (this.ToString() == obj.ToString());

}

}

}

**Вопросы**

1. Для чего используют статические классы?

[Статический](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/static) класс в основном такой же, как и нестатический класс, но имеется одно отличие: нельзя создавать экземпляры статического класса. Другими словами, нельзя использовать оператор [new](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/new-operator) для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса. Например, если есть статический класс, называемый UtilityClass, имеющий открытый статический метод с именем MethodA, вызов метода выполняется, как показано в следующем примере:

UtilityClass.MethodA();

Статический класс может использоваться как обычный контейнер для наборов методов, работающих на входных параметрах, и не должен возвращать или устанавливать каких-либо внутренних полей экземпляра. Например, в библиотеке классов .NET статический класс [System.Math](https://docs.microsoft.com/ru-ru/dotnet/api/system.math) содержит методы, выполняющие математические операции, без требования сохранять или извлекать данные, уникальные для конкретного экземпляра класса [Math](https://docs.microsoft.com/ru-ru/dotnet/api/system.math). Это значит, что члены класса применяются путем задания имени класса и имени метода, как показано в следующем примере.

double dub = -3.14;

Console.WriteLine(Math.Abs(dub));

Console.WriteLine(Math.Floor(dub));

Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:

// 3.14

// -4

// 3

2. Что может содержать статический класс?

Статические классы могут содержать только статические поля, свойства и методы.

3. Что такое производный и базовый классы?

(где-то в заданиях--)

4. Как используют ключевое слово base?

Слово используют в классе наследнике для обращения к методу, полю и т.д. базового класса. Необходимость в таком возникает, когда нам потребовался определенный метод, поле и т.д. базового класса, но в классе наследнике он был переопределен.

Так же base используют при определении конструктора в классе наследнике.

(где-то в заданиях--)

5. В чем заключена основная задача наследования?

Главная задача наследования - обеспечить повторное использование кода.

6. Пусть базовый класс содержит метод basefunc(), а производный класс не имеет метода с таким именем. Может ли объект производного класса иметь доступ к методу basefunc()? Если да, то при каких условиях?

* **Наследуются** все свойства, методы, поля и т.д., которые есть в базовом классе

**Но**

**Не наследуются** конструкторы базового класса

* Производному классу доступны public internal, protected и protected internal члены базового класса (**private недоступны**)

(где-то в заданиях--)

7. Напишите объявление конструктора без аргументов для производного класса B, который будет вызывать конструктор без аргументов базового класса A.

class A

{

}

class B : A

{

public B() : base()

{

}

}

8. Что такое полиморфизм? Приведите пример.

Сам термин полиморфизм можно перевести как **«много форм»**. А если говорить проcтыми словами, **полиморфизм** – это различная реализация однотипных действий. Классическая фраза, которая коротко объясняет полиморфизм – «Один интерфейс, множество реализаций». Приведу примеры из жизни. В автомобилях есть рулевое колесо. Это колесо является интерфейсом между водителем и автомобилем, который позволяет поворачивать автомобиль. Механическая реализация руля у автомобилей может быть разная, но при этом результат получается одинаковым – колесо вправо – автомобиль вправо, и наоборот. Еще один пример. Клавиатура является интерфейсом ввода между пользователем и ПК. При нажатии одной и той же клавиши на различных клавиатурах результат получаем одинаковый, но при этом сама реализация нажатия клавиши может быть разная (емкостная, контактная и тд.).

Переходим к программированию  
С полиморфизмом к нам прибавляются еще несколько понятий: виртуальный/абстрактный метод, переопределение метода.  
**Виртуальный метод** – это метод, который МОЖЕТ быть переопределен в классе-наследнике. Такой метод может иметь стандартную реализацию в базовом классе.  
**Абстрактный метод** – это метод, который ДОЛЖЕН быть реализован в классе-наследнике. При этом, абстрактный метод не может иметь своей реализации в базовом классе (тело пустое), в отличии от виртуального.  
**Переопределение метода** – это изменение реализации метода, установленного как виртуальный (в классе наследнике метод будет работать отлично от базового класса).

Например, есть класс Геометрическая Фигура, и в нем объявлен метод Draw(), который будет рисовать фигуру. От этого класса наследуются классы Треугольник, Прямоугольник, Окружность. В них реализуется метод для рисования (понятно, что реализация рисования каждой фигуры разная). В итоге мы можем создать объекты этих классов, и у всех будет метод Draw(), который будет рисовать соответствующую фигуру.

При наследовании нередко возникает необходимость изменить в классе-наследнике функционал метода, который был унаследован от базового класса. В этом случае класс-наследник может переопределять методы и свойства базового класса.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором **virtual**. Такие методы и свойства называют виртуальными.

А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором **override**. Переопределенный метод в классе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

9. Определите назначение виртуальных функций.

Может методов? Тогда

(где-то в заданиях--)

10. Кому доступны переменные с модификатором protected?

**protected**: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.

11. Наследуются ли переменные с модификатором private?

Да, но они не являются доступными.

12. As, is – что это, как применяется? В чем между ними отличие?

(где-то в заданиях--)

13. Поддерживает ли C# множественное наследование?

Нет, но поддерживает множественную реализацию интерфейсов.

14. Можно ли запретить наследование от класса?

Да, модификатором sealed.

sealed class Admin

{

}

15. Можно ли разрешить наследование класса, но запретить перекрытие метода?

Если метод не имеет модификатора virtual, то его нельзя переопределить в производном классе.

Если случай такой:

class A

{

public virtual void Move()

{

Console.WriteLine("A moooooove");

}

}

class B : A

{

public override void Move()

{

Console.WriteLine("B moooooove");

}

}

То есть мы говорим о переопределении уже переопределенных методов.

Метод Move переопределен в классе наследнике B, но от класса B тоже можно наследоваться, чтобы запретить переопределение для наследников класса B нужно использовать модификатор sealed:

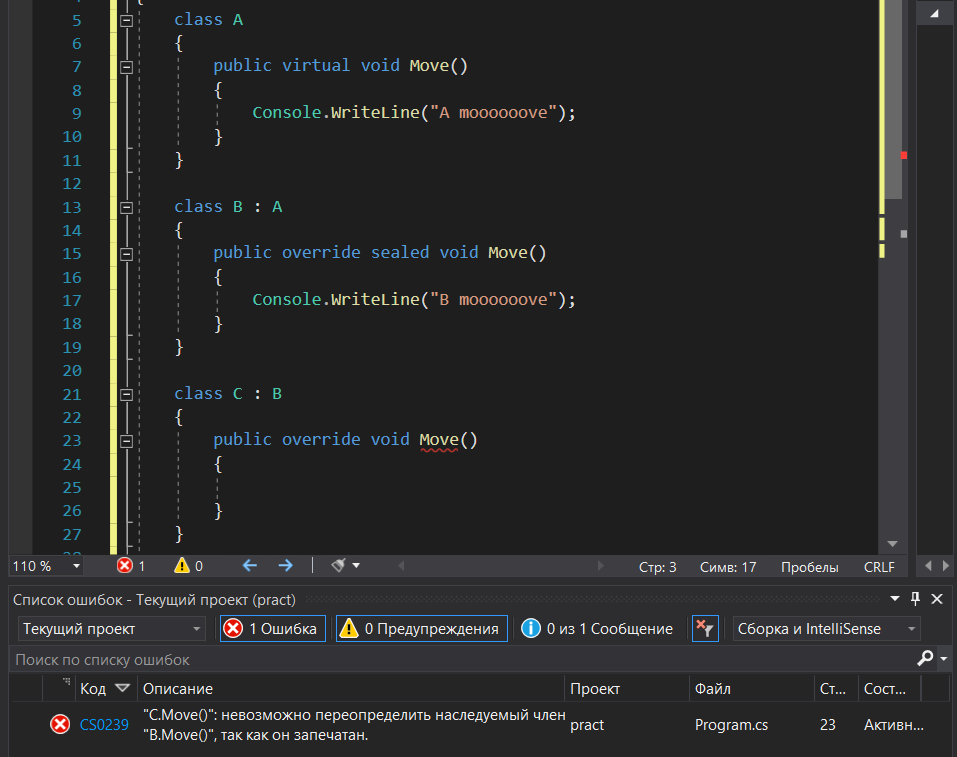
public override **sealed** void Move()

{

Console.WriteLine("Bmoooooove");

}

Вот что из этого выйдет:



16. Что такое абстрактный класс?

(где-то в заданиях--)

17. В каком случае вы обязаны объявить класс абстрактным?

◦ Класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный.

◦ Когда производный класс наследует абстрактный класс, в нем должны быть реализованы все абстрактные методы базового класса. В противном случае производный класс должен быть также определен как abstract.

18. В чем разница между абстрактными и виртуальными классами? Между виртуальными и абстрактными методами?

Абстрактный класс это класс, содержащий хотя бы один метод (abstract).

Понятия “виртуальный класс” в языке C# вообще нет.

Виртуальный метод имеет реализацию и **может быть** переопределен в производном классе. Абстрактный метод не имеет реализацию, только описание метода, который **должен быть** реализован в производных классах.

19. Какие компоненты класса могут быть виртуальными?

Виртуальными могут быть:

* Методы
* Свойства
* Индексаторы
* События

20. Что такое интерфейс?

(где-то в заданиях--)

21. Что может содержать интерфейс?

(где-то в заданиях--)

22. Как работать с объектом через унаследованный интерфейс?

Допустим, мы оформили какие-то действия с объектами в виде метода с параметром типа object. Прежде чем использовать этот параметр внутри метода для обращения к методам, описанным в производных классах, требуется выполнить преобразование к производному классу. Для безопасного преобразования следует проверить, возможно ли оно, например, так:

static void Act( object A )

{

if ( A is IAction )

{

IAction Actor = (IAction) A;

Actor.Draw();

}

}

В метод Act можно передавать любые объекты, но на экран будут выведены только те, которые поддерживают интерфейс IAction, .

Недостатком использования операции is является то, что преобразование фактически выполняется дважды: при проверке и при собственно преобразовании. Более эффективной является другая операция — as. Она выполняет преобразование к заданному типу, а если это невозможно, формирует результат null, например:

static void Act( object A )

{

IAction Actor = A as IAction;

if ( Actor != null ) Actor.Draw();

}

Обе рассмотренные операции применяются как к интерфейсам, так и к классам.

23. Приведите пример явной реализации интерфейса.

(где-то в заданиях--)

24. Почему нельзя указать модификатор видимости для методов интерфейса?

С 8 версии .NET можно.

Интерфейсы, объявляемые непосредственно в пространстве имен, могут быть public или internal. Как и в случае с классами и структурами, для интерфейсов по умолчанию задается доступ internal. **Члены интерфейса** по умолчанию являются public, так как интерфейс как раз и создан для того, чтобы обеспечить другим типам доступ к классу или структуре. Объявления членов интерфейса могут включать любой модификатор доступа. Это удобнее всего использовать для статических методов, чтобы предоставить общие реализации, необходимые всем объектам, реализующим класс.

25. Можно ли наследовать от нескольких интерфейсов?

Можно.

26. Назовите отличия между интерфейсом и абстрактным классом.

Между интерфейсами и абстрактными классами были замечены следующие различия:

* интерфейс не может иметь конструкторов. Абстрактный класс может иметь конструкторы;
* интерфейс не может содержать нестатические поля данных. Абстрактный класс допускает использование статических полей данных;
* элементы интерфейса по умолчанию (без модификатора доступа) считаются public. В абстрактных классах элементы по умолчанию считаются private;
* производный класс может наследовать только один абстрактный базовый класс. В случае использования интерфейсов производный класс может наследовать любое количество интерфейсов. Таким образом, интерфейс – это альтернатива абстрактного класса, с помощью которой можно осуществить множественное наследование.

27. Для чего используются стандартные интерфейсы ICloneable, IComparable, IComparer, lEnumerable?

## ► Копирование объектов. Интерфейс ICloneable

## Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности:

class Program

{

static void Main(string[] args)

{

Person p1 = new Person { Name = "Tom", Age = 23 };

Person p2 = p1;

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Alice

Console.Read();

}

}

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

В данном случае объекты p1 и p2 будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной p2 затронут также и переменную p1.

Чтобы переменная p2 указывала на новый объект, но со значениями из p1, мы можем применить клонирование с помощью реализации интерфейса **ICloneable**:

public interface ICloneable

{

object Clone();

}

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public object Clone()

{

return new Person { Name = this.Name, Age = this.Age };

}

}

Использование:

class Program

{

static void Main(string[] args)

{

Person p1 = new Person { Name = "Tom", Age = 23 };

Person p2 = (Person)p1.Clone();

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Tom

Console.Read();

}

}

Теперь все нормально копируется, изменения в свойствах p2 не сказываются на свойствах в p1.

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public object Clone()

{

return this.MemberwiseClone();

}

}

Этот метод реализует **поверхностное (неглубокое) копирование**. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public Company Work { get; set; }

public object Clone()

{

return this.MemberwiseClone();

}

}

class Company

{

public string Name { get; set; }

}

В этом случае при копировании новая копия будет указывать на тот же объект Company:

Person p1 = new Person { Name = "Tom", Age = 23, Work = new Company { Name = "Microsoft" } };

Person p2 = (Person)p1.Clone();

p2.Work.Name = "Google";

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Tom

Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

{

public string Name { get; set; }

public int Age { get; set; }

public Company Work { get; set; }

public object Clone()

{

Company company = new Company { Name = this.Work.Name };

return new Person

{

Name = this.Name,

Age = this.Age,

Work = company

};

}

}

class Company

{

public string Name { get; set; }

}

## ► Сортировка объектов. Интерфейс IComparable

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:

int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };

Array.Sort(numbers);

foreach (int n in numbers)

Console.WriteLine(n);

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс **IComparable**. Он имеет всего один метод:

public interface IComparable

{

int CompareTo(object o);

}

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
* Равен нулю. Значит, оба объекта равны
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

class Person : IComparable

{

public string Name { get; set; }

public int Age { get; set; }

public int CompareTo(object o)

{

Person p = o as Person;

if (p != null)

return this.Name.CompareTo(p.Name);

else

throw new Exception("Невозможно сравнить два объекта");

}

}

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

Person p1 = new Person { Name = "Bill", Age = 34 };

Person p2 = new Person { Name = "Tom", Age = 23 };

Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };

Array.Sort(people);

foreach (Person p in people)

{

Console.WriteLine($"{p.Name} - {p.Age}");

}

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

class Person : IComparable<Person>

{

public string Name { get; set; }

public int Age { get; set; }

public int CompareTo(Person p)

{

return this.Name.CompareTo(p.Name);

}

}

**Я такое написала:**

using System;

namespace asitis

{

class Num : IComparable

{

public int n;

public Num(int n)

{

this.n = n;

}

public int CompareTo(object obj)

{

Num number = obj as Num;

if (number != null)

{

return this.n.CompareTo(number.n);

}

else

{

throw new Exception("Невозможно сравнить два объекта");

}

}

}

class Program

{

static void Main(string[] args)

{

Num a = new Num(34);

Num b = new Num(3);

Num c = new Num(-7);

Num d = new Num(50);

Num e = new Num(3);

Num[] numbers = new Num[] { a, b, c, d, e };

Array.Sort(numbers);

foreach(Num number in numbers)

{

Console.WriteLine(number.n);

}

}

}

}

## ► IComparer.Применение компаратора

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

interface IComparer

{

int Compare(object o1, object o2);

}

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

// Применение необобщеннкого варианта интерфейса IComparer

using System;

using System.Collections;

// Создать объект типа IComparer для объектов класса Inventory,

class CompInv : IComparer

{

// Реализовать интерфейс IComparer.

public int Compare(object x, object y)

{

Inventory a, b;

a = (Inventory)x;

b = (Inventory)y;

return string.Compare(a.name, b.name);

}

}

class Inventory

{

public string name;

double cost;

int onhand;

public Inventory(string n, double c, int h)

{

name = n;

cost = c;

onhand = h;

}

public override string ToString()

{

return

String.Format($"{name}\t Цена: {cost}\t В наличии: {onhand}");

}

}

class IComparerDemo

{

static void Main()

{

CompInv comp = new CompInv();

ArrayList inv = new ArrayList();

// Добавить элементы в список.

inv.Add(new Inventory("Кусачки", 5.95, 3));

inv.Add(new Inventory("Отвертки", 8.29, 2));

inv.Add(new Inventory("Молотки", 3.50, 4));

inv.Add(new Inventory("Дрели", 19.88, 8));

Console.WriteLine("Перечень товарных запасов до сортировки:");

foreach (Inventory i in inv)

{

Console.WriteLine(" " + i);

}

Console.WriteLine();

// Отсортировать список, используя интерфейс IComparer.

inv.Sort(comp);

Console.WriteLine("Перечень товарных запасов после сортировки:");

foreach (Inventory i in inv)

{

Console.WriteLine(" " + i);

}

}

}

## ► Интерфейсы IEnumerable и IEnumerator

## Как мы увидели, основной для большинства коллекций является реализация интерфейсов IEnumerable и IEnumerator. Благодаря такой реализации мы можем перебирать объекты в цикле foreach:

foreach (var item in перечислимый\_объект)

{

}

Перебираемая коллекция должна реализовать интерфейс IEnumerable.

Интерфейс IEnumerable имеет метод, возвращающий ссылку на другой интерфейс - перечислитель:

public interface IEnumerable

{

IEnumerator GetEnumerator();

}

А интерфейс IEnumerator определяет функционал для перебора внутренних объектов в контейнере:

public interface IEnumerator

{

bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов

object Current { get; } // текущий элемент в контейнере

void Reset(); // перемещение в начало контейнера

}

Метод **MoveNext()** перемещает указатель с текущего элемента на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство **Current** возвращает объект в последовательности, на который указывает указатель.

Метод **Reset()** сбрасывает указатель позиции в начальное положение.

Каким именно образом будет осуществляться перемещение указателя и получение элементов зависит от реализации интерфейса. В различных реализациях логика может быть построена различным образом.

Например, без использования цикла foreach перебирем коллекцию с помощью интерфейса IEnumerator:

### Реализация IEnumerable и IEnumerator

Рассмотрим простейшую реализацию **IEnumerable** на примере:

using System;

using System.Collections;

namespace HelloApp

{

class Week : IEnumerable

{

string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",

"Friday", "Saturday", "Sunday" };

public IEnumerator GetEnumerator()

{

return days.GetEnumerator();

}

}

class Program

{

static void Main(string[] args)

{

Week week = new Week();

foreach (var day in week)

{

Console.WriteLine(day);

}

Console.Read();

}

}

}

В данном случае класс Week, который представляет неделю и хранит все дни недели, реализует интерфейс IEnumerable. Однако в данном случае мы поступили очень просто - вместо реализации IEnumerator мы просто возвращаем в методе GetEnumerator объект IEnumerator для массива.

public IEnumerator GetEnumerator()

{

return days.GetEnumerator();

}

Благодаря этому мы можем перебрать все дни недели в цикле foreach.

В то же время стоит отметить, что для перебора коллекции через foreach в принципе необязательно реализовать интерфейс IEnumerable. Достаточно в классе определить публичный метод **GetEnumerator**, который бы возвращал объект IEnumerator. Например:

class Week

{

string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",

"Friday", "Saturday", "Sunday" };

public IEnumerator GetEnumerator()

{

return days.GetEnumerator();

}

}

Однако это было довольно просто - мы просто используем уже готовый перчислитель массива. Но, возможно, потребуется задать свою собственную логику перебора объектов. Для этого реализуем интерфейс **IEnumerator**:

using System;

using System.Collections;

namespace HelloApp

{

class WeekEnumerator : IEnumerator

{

string[] days;

int position = -1;

public WeekEnumerator(string[] days)

{

this.days = days;

}

public object Current

{

get

{

if (position == -1 || position >= days.Length)

throw new InvalidOperationException();

return days[position];

}

}

public bool MoveNext()

{

if (position < days.Length - 1)

{

position++;

return true;

}

else

return false;

}

public void Reset()

{

position = -1;

}

}

class Week

{

string[] days = { "Monday", "Tuesday", "Wednesday", "Thursday",

"Friday", "Saturday", "Sunday" };

public IEnumerator GetEnumerator()

{

return new WeekEnumerator(days);

}

}

class Program

{

static void Main(string[] args)

{

Week week = new Week();

foreach (var day in week)

{

Console.WriteLine(day);

}

Console.Read();

}

}

}

Здесь теперь класс Week использует не встроенный перечислитель, а WeekEnumerator, который реализует IEnumerator.

Ключевой момент при реализации перечислителя - перемещения указателя на элемент. В классе WeekEnumerator для хранения текущей позиции определена переменная position. Следует учитывать, что в самом начале (в исходном состоянии) указатель должен указывать на позицию условно перед первым элементом. Когда будет производиться цикл foreach, то данный цикл вначале вызывает метод MoveNext и фактически перемещает указатель на одну позицию в перед и только затем обращается к свойству Current для получения элемента в текущей позиции.

28. В какой строке приведенного ниже фрагмента листинга не содержится ошибки?

class A

{

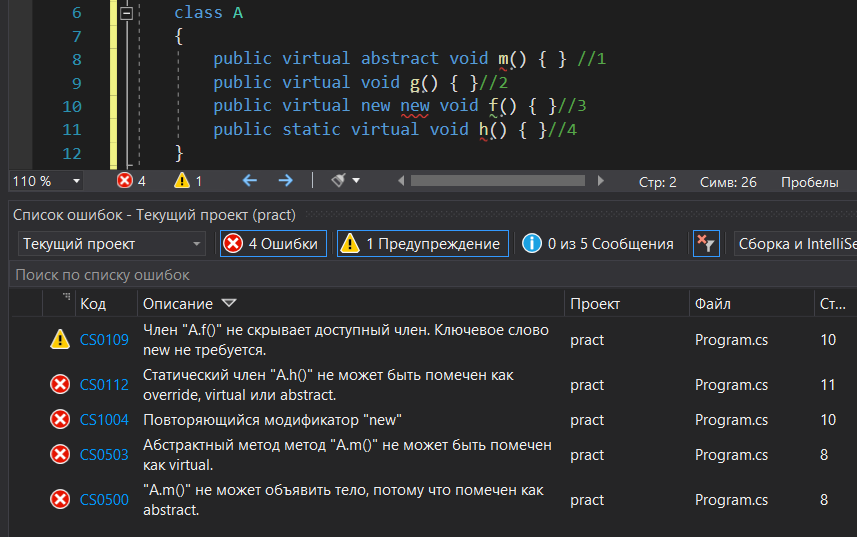
public virtual abstract void m() { } //1

public virtual void g() { }//2

public virtual new new void f() { }//3

public static virtual void h() { }//4

}



29. Что будет выведено на консоль в результате выполнения следующего фрагмента?

|  |
| --- |
| class A  {  public int x = 1;  }  class B : A  {  public new int x = 2;  public void m(int a, int b)  {  x = a;  base.x = b;  Console.Write(x + " " + base.x);  }  }  class Test  {  static void Main(string[] args)  {  A a = new A();  B b = new B();  b.m(3, 4);  }  } |

****

30. Что будет выведено на консоль в результате выполнения следующего фрагмента кода?

class A

{

public class B : A

{

public override void mA()

{

Console.WriteLine("B ");

}

}

public virtual void mA()

{

Console.WriteLine("A ");

}

}

class Prоgrаm

{

static void Main(string[] args)

{

A a = new A();

A.B b = new A.B();

a.mA();

b.mA();

}

}



31. Чем может быть M4 если дано следующее определение:

|  |  |
| --- | --- |
| **public class** C1 : M1, M2 { } **public struct** S1 : M3, M4 { }; | Варианты ответа:  1) M4 - только интерфейс  2) M4 - интерфейс или класс  3) M4 - только класс  4) M4 - только структура  5) M4 - делегат |

32. Выберите верное присваивание для объектов, определенных в листинге.

class A { }

class B : A { }

class C : B { }

class D { }

class Test

{

static void Main(string[] args)

{

A a = new A();

B b = new B();

C c = new C();

D d = new D();

}

}

Варианты ответа:

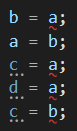
1) b = a;

2) a = b;

3) c = a;

4) d = a;

5) с = b;

****

33. Что будет выведено на консоль в результате выполнения следующего фрагмента, если раскомментировать строчку 1?

|  |  |
| --- | --- |
| public abstract class A  {  public virtual void method()  { Console.Write("A "); }  }  public class B : A  {  public override void method()  { // base.method(); // 1  // this.method(); // 2  Console.Write("B ");  }  }  class Program2  {  static void Main(string[] args)  {  A my = new B();  my.method();  }  } | Варианты ответа:  1) B  2) A  3) A B  4) B A  5) 0 |

Если раскомментировать вторую произойдет переполнение стека.

34. В какой строке приведенного ниже фрагмента листинга содержится ошибка?

|  |
| --- |
| public abstract class A  {  public virtual string m() { return "A"; }//1  }  public class B : A  {  public override new string m() { return "B"; }//2  }  public class C : B  {  public string m() { return "C"; }//3  }  class Prоgrаm  {  static void Main(string[] args)  {  A ac = new C(); //4  Console.WriteLine(ac.m());  }  } |

Во второй 

35. Почему приведенный ниже фрагмент листинга содержит ошибку?

abstract class Student //1

{

public int Age { get; set; } //2

public string Name { get; set; } //3

}

class Program

{

static void Main(string[] args)

{

Student Olga = new Student();//4

}

}

4 вариант:



36. В какой строке может быть ошибка компиляции?

class A { }

class B : A { }

class C : A { } //1

class Program4

{

static void Main()

{

A one = new B(); //2

A two = new C(); //3

one = two; //4

}

}

Ошибок не выдает, но на этапе выполнения, думаю, могут быть проблемы с 4 вариантом.

37. Что будет выведено на консоль в результате выполнения следующего фрагмента листинга:

interface Interface1

{

void f();

void g();

}

class A

{

public void f() { System.Console.WriteLine("F"); }

public void g() { System.Console.WriteLine("G"); }

}

class B : A, Interface1

{

new public void g() { System.Console.WriteLine("new G"); }

}

class Program5

{

static void Main(string[] args)

{

//Interface1 obj = new B();

//obj.g();

B obj = new B();

obj.g();

}

}



А если так:

Interface1 obj = new B();

obj.g();

//B obj = new B();

//obj.g();

то:

