№3 Проектирование типов. Классы.

|  |  |
| --- | --- |
| **Вариант 3** | Создать класс **Student**: id, Фамилия, Имя, Отчество, Дата рождения, Адрес, Телефон, Факультет, Курс, Группа. Свойства и конструкторы должны обеспечивать проверку корректности. Добавить метод расчет возраста студента  *Создать массив объектов. Вывести:*  *a) список студентов заданного факультета;*  *d) список учебной группы.* |

**1**

Определить класс, указанный в варианте, содержащий:

 Не менее трех конструкторов (с параметрами и без, а также с параметрами по умолчанию);

 **статический конструктор** (конструктор типа);

 определите **закрытый** конструктор; предложите варианты его вызова;

 поле - **только для чтения** (например, для каждого экземпляра сделайте поле только для чтения ID - равно некоторому уникальному номеру (хэшу) вычисляемому автоматически на основе инициализаторов объекта);

 поле - константу;

 **свойства** (get, set) – для всех поле класса (поля класса должны быть закрытыми); Для одного из свойств ограните доступ по set

 в одном из методов класса для работы с аргументами используйте **ref -** и **out**-**параметры**.

 создайте в классе **статическое поле**, хранящее количество созданных объектов (инкрементируется в конструкторе) и **статический метод** вывода информации о классе.

 сделайте класс **partial**

 переопределяете методы класса Object: Equals, для сравнения объектов, GetHashCode; для алгоритма вычисления хэша руководствуйтесь стандартными рекомендациями, ToString – вывода строки – информации об объекте.

► **Классы и объекты**

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является **класс**, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.



По умолчанию проект консольного приложения уже содержит один класс Program, с которого и начинается выполнение программы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова **сlass**:

|  |  |
| --- | --- |
| 1  2  3  4 | class Person  {    } |

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using System;    namespace HelloApp  {      class Person      {        }      class Program      {          static void Main(string[] args)          {            }      }  } |

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями.

Определим класс Person:

class Person

{

public string name; // имя (поле класса)

public int age; // возраст (поле)

public void GetInfo() //метод

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

class Program

{

static void Main(string[] args)

{

Person jojo; // переменная jojo не указывает ни на какой объект и по умолчанию она имеет значение null

Person tom = new Person(); //конструктор

tom.GetInfo(); // Имя: Возраст: 0

tom.name = "Tom";

tom.age = 34;

tom.GetInfo(); // Имя: Tom Возраст: 34

}

}

**Конструкторы**

Кроме обычных методов в классах используются также и специальные методы, которые называются **конструкторами**. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта.

#### Конструктор по умолчанию

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор.

Для создания объекта Person используется выражение new Person(). Оператор **new** выделяет память для объекта Person. И затем вызывается **конструктор** по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение **null** (то есть фактически отсутствие значения).

#### Создание конструкторов

Мы сами можем определить свои конструкторы:

class Person

{

public string name;

public int age;

public Person() { name = "Неизвестно"; age = 18; } // 1 конструктор

public Person(string n) { name = n; age = 18; } // 2 конструктор

public Person(string n, int a) { name = n; age = a; } // 3 конструктор

public void GetInfo()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

static void Main(string[] args)

{

Person tom = new Person(); // вызов 1-ого конструктора без параметров

Person bob = new Person("Bob"); //вызов 2-ого конструктора с одним параметром

Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами

bob.GetInfo(); // Имя: Bob Возраст: 18

tom.GetInfo(); // Имя: Неизвестно Возраст: 18

sam.GetInfo(); // Имя: Sam Возраст: 25

}

Короче, конструктор – это специальный метод, который вызывается при создании объекта с ключевым словом new. Конструктор по умолчанию не вносит изменений в логику класса, но мы можем создать собственный конструктор, где сами решаем, что происходит при создании объекта.

### Ключевое слово this

Ключевое слово **this** представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

class Person

{

    public string name;

    public int age;

    public Person() **: this**("Неизвестно")

    {

    }

    public Person(string name) **: this**(name, 18)

    {

    }

    public Person(string name, int age)

    {

        this.name = name;

        this.age = age;

    }

    public void GetInfo()

    {

        Console.WriteLine($"Имя: {name}  Возраст: {age}");

    }

}

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

|  |  |
| --- | --- |
| 1  2  3 | public Person(string name) : this(name, 18)  {  } |

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Person(string name, int age)  {      this.name = name;      this.age = age;  } |

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

### Инициализаторы объектов

Для инициализации объектов классов можно применять **инициализаторы**. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

|  |  |
| --- | --- |
|  | Person tom = new Person **{ name = "Tom", age=31 }**; |

## ► Модификаторы доступа

Все члены класса - поля, методы, свойства - все они имеют **модификаторы доступа**. Модификаторы доступа позволяют задать допустимую область видимости для членов класса. То есть модификаторы доступа определяют контекст, в котором можно употреблять данную переменную или метод.

В C# применяются следующие модификаторы доступа:

* **public**: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* **private**: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* **protected**: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* **internal**: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* **protected internal**: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* **private protected**: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

Мы можем явно задать модификатор доступа, например:

**private protected** class State

{

    internal int a;

    protected void Print()

    {

        Console.WriteLine($"a = {a}");

    }

}

Либо можем не указывать:

class State

{

    int a;

    void Print()

    {

        Console.WriteLine($"a = {a}");

    }

}

Если для полей и методов не определен модификатор доступа, то по умолчанию для них применяется модификатор **private**.

Классы и структуры, объявленные без модификатора, по умолчанию имеют доступ **internal**.

**! Все классы и структуры**, определенные напрямую в пространствах имен и **не являющиеся вложенными в другие классы**, **могут иметь только модификаторы public или internal**.

// Открытый тип доступен из любой сборки

public class cl1 { }

// Внутренний тип доступен только из собственной сборки

internal class cl2 { }

//Это внутренний тип, так как модификатор доступа не указан явно

class cl3 { }

## ► Свойства

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают простой доступ к полям классов и структур, позволяя узнать их значение или выполнить их установку.

Стандартное описание свойства имеет следующий синтаксис:

|  |  |
| --- | --- |
| 1  2  3  4 | [модификатор\_доступа] возвращаемый\_тип произвольное\_название  {      // код свойства  } |

Например:

class Person

{

private string name;

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

}

Здесь у нас есть закрытое поле name и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. Стандартное определение свойства содержит блоки **get** и **set**. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.

Мы можем использовать данное свойство следующим образом:

Person p = new Person();

// Устанавливаем свойство - срабатывает блок Set

// значение "Tom" и есть передаваемое в свойство value

p.Name = "Tom";

// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get

string personName = p.Name;

Свойства позволяют вложить *дополнительную логику*, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по возрасту:

class Person

{

private int age;

public int Age

{

set

{

if (value < 18)

{

Console.WriteLine("Возраст должен быть больше 17");

}

else

{

age = value;

}

}

get { return age; }

}

}

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют только блок get, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

### Модификаторы доступа у аксессоров (set, get)

class Person

{

private string name;

public string Name

{

get

{

return name;

}

**private** set

{

name = value;

}

}

public Person(string name)

{

Name = name;

}

}

Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе.

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

* Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get)
* Только один блок set или get может иметь модификатор доступа, но не оба сразу
* Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы protected internal, internal, protected, private

### Автоматические свойства

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в фреймворк .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

public string Name { get; set; }

public int Age { get; set; }

В чем преимущество автосвойств, если по сути они просто обращаются к автоматически создаваемой переменной, почему бы напрямую не обратиться к переменной без автосвойств? Дело в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

public string Name { get; set; } = "Tom";

public int Age { get; set; } = 23;

Автосвойства также могут иметь модификаторы доступа.

## ► Статические члены и модификатор static

## Кроме обычных полей, методов, свойств, класс может иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса.

class Account

{

public **static** decimal bonus = 100;

public decimal totalSum;

public Account(decimal sum)

{

totalSum = sum + bonus;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine(Account.bonus); // 100

**Account.bonus** += 200; ← не создается экземпляр

Account account1 = new Account(150);

Console.WriteLine(account1.totalSum); // 450

Account account2 = new Account(1000);

Console.WriteLine(account2.totalSum); // 1300

Console.ReadKey();

}

}

Поле bonus является статическим, поэтому оно хранит состояние класса в целом, а не отдельного объекта. И поэтому мы можем обращаться к этому полю по имени класса:

|  |  |
| --- | --- |
| 1  2 | Console.WriteLine(Account.bonus);  Account.bonus += 200; |

На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса.



При этом память для статических переменных выделяется даже в том случае, если не создано ни одного объекта этого класса.

### Статические свойства и методы

Подобным образом мы можем создавать и использовать статические методы и свойства:

class Account

{

public Account(decimal sum, decimal rate)

{

if (sum < MinSum) throw new Exception("Недопустимая сумма!");

Sum = sum; Rate = rate;

}

private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов

public **static** decimal MinSum

{

get { return minSum; }

set { if (value > 0) minSum = value; }

}

public decimal Sum { get; private set; } // сумма на счете

public decimal Rate { get; private set; } // процентная ставка

// подсчет суммы на счете через определенный период по определенной ставке

public **static** decimal GetSum(decimal sum, decimal rate, int period)

{

decimal result = sum;

for (int i = 1; i <= period; i++)

result = result + result \* rate / 100;

return result;

}

}

Переменная minSum, свойство MinSum, а также метод GetSum здесь определены с ключевым словом **static**, то есть они являются статическими.

Переменная minSum и свойство MinSum представляют минимальную сумму, которая допустима для создания счета. Этот показатель не относится к какому-то конкретному счету, а относится ко всем счетам в целом. Если мы изменим этот показатель для одного счета, то он также должен измениться и для другого счета. То есть в отличии от свойств Sum и Rate, которые хранят состояние объекта, переменная minSum хранит состояние для всех объектов данного класса.

То же самое с методом GetSum - он вычисляет сумму на счете через определенный период по определенной процентной ставке для определенной начальной суммы. *Вызов и результат этого метода не зависит от конкретного объекта или его состояния.*

Таким образом, ***переменные и свойства****, которые хранят состояние, общее для всех объектов класса, следует определять как статические*. И также ***методы****, которые определяют общее для всех объектов поведение, также следует объявлять как статические.*

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса:

|  |  |
| --- | --- |
| 1  2 | Account.MinSum = 560;  decimal result = Account.GetSum(1000, 10, 5); |

Следует учитывать, что статические методы **могут обращаться только к статическим членам класса**. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

### Статический конструктор

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

* Статические конструкторы не должны иметь модификатор доступа и не принимают параметров
* Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса
* Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз.

Определим статический конструктор:

class User

{

**static User()**

{

Console.WriteLine("Создан первый пользователь");

}

}

class Program

{

static void Main(string[] args)

{

User user1 = new User(); // здесь сработает статический конструктор

User user2 = new User();

Console.Read();

}

}

### Статические классы

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы. Например, если бы класс Account имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический:

static class Account

{

private static decimal minSum = 100; // минимальная допустимая сумма для всех счетов

public static decimal MinSum

{

get { return minSum; }

set { if (value > 0) minSum = value; }

}

// подсчет суммы на счете через определенный период по определенной ставке

public static decimal GetSum(decimal sum, decimal rate, int period)

{

decimal result = sum;

for (int i = 1; i <= period; i++)

result = result + result \* rate / 100;

return result;

}

## }

Ниже приведены основные возможности статического класса.

* Содержит только статические члены.
* Создавать его экземпляры нельзя.
* Является запечатанным.
* Не может содержать [конструкторы экземпляров](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/instance-constructors).

По сути, создание статического класса аналогично созданию класса, содержащего только статические члены и закрытый конструктор. Закрытый конструктор (с модификатором private) не допускает создания экземпляров класса. Преимущество применения статических классов заключается в том, что компилятор может проверить отсутствие случайно добавленных членов экземпляров. Таким образом, компилятор гарантирует невозможность создания экземпляров таких классов.

Статические классы запечатаны, поэтому их нельзя наследовать. Они не могут наследовать ни от каких классов, кроме [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object). Статические классы не могут содержать конструктор экземпляров. Однако они могут содержать статический конструктор. Нестатический класс также должен определять статический конструктор, если класс содержит статические члены, для которых нужна нетривиальная инициализация.

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.

## ► Константы, поля и структуры для чтения

**Константы**

Константы характеризуются следующими признаками:

* Константа должна быть проинициализирована при определении
* После определения значение константы не может быть изменено

Константы предназначены для описания таких значений, которые не должны изменяться в программе. Для определения констант используется ключевое слово **const**:

const double PI = 3.14;

const double E = 2.71;

При использовании констант надо помнить, что объявить мы их можем только один раз и что к моменту компиляции они должны быть определены.

class MathLib

{

public const double PI = 3.141;

public const double E = 2.81;

public const double K; // Ошибка, константа не инициализирована

}

class Program

{

static void Main(string[] args)

{

MathLib.E = 3.8; // Ошибка, значение константы нельзя изменить

}

}

Также обратите внимание на синтаксис обращения к константе. Так как **неявно это статическое поле**, для обращения к ней необходимо использовать имя класса.

Но следует учитывать, что мы не можем объявить константу с модификатором static. Но в этом собственно и нет смысла.

Константу можно определить как на уровне класса, так и внутри метода:

class MathLib

{

public double GetCircleArea(double radius)

{

const double PI = 3.141;

return PI \* radius \* radius;

}

}

### Поля для чтения

Поля для чтения можно инициализировать при их объявлении на уровне класса, либо инициализировать и изменять в *конструкторе*. Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом **readonly**:

class MathLib

{

public **readonly** double K = 23; // можно так инициализировать

public MathLib(double \_k)

{

K = \_k; // поле для чтения может быть инициализировано или изменено в конструкторе после компиляции

}

public void ChangeField()

{

// так нельзя

//K = 34;

}

}

class Program

{

static void Main(string[] args)

{

MathLib mathLib = new MathLib(3.8);

Console.WriteLine(mathLib.K); // 3.8

//mathLib.K = 7.6; // поле для чтения нельзя установить вне своего класса

Console.ReadLine();

}

}

#### Сравнение констант и полей для чтения

* Константы должны быть определены во время компиляции, а поля для чтения могут быть определены во время выполнения программы.

Соответственно инициализировать константу можно только при ее определении.

Поле для чтения можно инициализировать либо при его определении, либо в конструкторе класса.

* Константы не могут использовать модификатор static, так как уже неявно являются статическими. Поля для чтения могут быть как статическими, так и не статическими.

## ► Передача параметров по ссылке и значению. Выходные параметры

Существует два способа передачи параметров в метод в языке C#: **по значению** и **по ссылке**.

### Передача параметров по значению

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

class Program

{

static void Main(string[] args)

{

Sum(10, 15); // параметры передаются по значению

Console.ReadKey();

}

static int Sum(int x, int y)

{

return x + y;

}

}

### Передача параметров по ссылке и модификатор ref

При передаче параметров по ссылке перед параметрами используется модификатор **ref**:

static void Main(string[] args)

{

int x = 10;

int y = 15;

Addition(**ref** x, y); // вызов метода

Console.WriteLine(x); // 25

Console.ReadLine();

}

// параметр x передается по ссылке

static void Addition(**ref** int x, int y)

{

x += y;

}

Обратите внимание, что модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

### Сравнение передачи по значению и по ссылке

В чем отличие двух способов передачи параметров? *При передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти.* И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

### Выходные параметры. Модификатор out

Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out:

static void Sum(int x, int y, **out** int a)

{

a = x + y;

}

Здесь результат возвращается не через оператор return, а через выходной параметр. Использование в программе:

static void Main(string[] args)

{

int x = 10;

int z;

Sum(x, 15, **out** z);

Console.WriteLine(z);

Console.ReadKey();

}

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:

static void Sum(int x, int y, out int a)

{

    Console.WriteLine(x+y);

}

Прелесть использования подобных параметров состоит в том, что по сути мы можем **вернуть** из метода не один вариант, а **несколько**.

static void GetData(int x, int y, out int area, out int perim)

{

area = x \* y;

perim = (x + y) \* 2;

}

По сути, как и в случае с ключевым словом **ref**, ключевое слово **out** применяется для передачи аргументов по ссылке. Однако, в отличие от ref, для переменных, которые передаются с ключевым словом out, не требуется инициализация. И, кроме того, вызываемый метод должен обязательно присвоить им значение.

## ► Частичные классы

Классы могут быть частичными. То есть мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно.

Например, определим в проекте два файла с кодом. Не столь важно как эти файлы будут называться. Например, PersonBase.cs и PersonAdditional.cs. В одном из этих файлов (без разницы в каком именно) определим следующий класс:

public **partial** class Person

{

public void Move()

{

Console.WriteLine("I am moving");

}

## }

## А в другом файле определим следующий класс:

public **partial** class Person

{

public void Eat()

{

Console.WriteLine("I am eating");

}

## }

## Таким образом, два файла в проекте содержат определение одного и того же класса Person, которые содержат два разных метода. И оба определенных здесь класса являются частичными. Для этого они определяются с ключевым словом partial.

## 

## Затем мы можем использовать все методы класса Person:

class Program

{

static void Main(string[] args)

{

Person tom = new Person();

tom.Move();

tom.Eat();

Console.ReadKey();

}

## }

### Частичные методы

Частичные классы могут содержать частичные методы. Такие методы также определяются с ключевым словом **partial**. Причем *определение* частичного метода *без тела* находится в одном частичном классе, а *реализация* этого же метода - в другом частичном классе.

Например, изменим вышеопределенные классы Person. Первая часть класса:

public partial class Person

{

**partial void Read();**

    public void DoSomething()

    {

        Read();

    }

}

## Вторая часть:

public partial class Person

{

**partial void Read()**

**{**

**Console.WriteLine("I am reading a book");**

**}**

}

В первом классе определен метод Read(). Причем на момент определения первого класса неизвестно, что представляет собой этот метод, какие действия он будет выполнять. Тем не менее, мы знаем список его параметров и можем вызвать в первом классе.

Второй класс уже непосредственно определяет тело метода Read().

Стоит отметить, что по умолчанию к частичным методам применяется ряд ограничений:

* Они не могут иметь модификаторы доступа
* Они имеют тип **void**
* Они не могут иметь out-параметры
* Они не могут иметь модификаторы virtual, override, sealed, new или extern

## ► Закрытые конструкторы

## Закрытый конструктор - это специальный конструктор экземпляра. Обычно он используется в классах, содержащих только статические члены. Если у класса есть один или несколько закрытых конструкторов и *нет общедоступных конструкторов*, другие классы (кроме вложенных классов) *не могут создавать экземпляры* этого класса.

class NLog

{

// Private Constructor:

**private** NLog() { }

public static double e = Math.E; //2.71828...

## }

## Объявление пустого закрытого конструктора *предотвращает автоматическое создание конструктора без параметров*. Обратите внимание, что если вы не используете модификатор доступа с конструктором, он по-прежнему не будет public по умолчанию. Однако модификатор [private](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/private) обычно используется явно, чтобы прояснить, что класс не может быть создан.

## Пример класса с частным конструктором:

public class Counter

{

private Counter() { }

public static int currentCount;

public static int IncrementCount()

{

return ++currentCount;

}

}

class TestCounter

{

static void Main()

{

// If you uncomment the following statement, it will generate

// an error because the constructor is inaccessible:

// Counter aCounter = new Counter(); // Error

Counter.currentCount = 100;

Counter.IncrementCount();

Console.WriteLine("New count: {0}", Counter.currentCount);

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

## // Output: New count: 101

# ► Работа с датами и временем

## Структура DateTime

Для работы с датами и временем в .NET предназначена структура **DateTime**. Она представляет дату и время от 00:00:00 1 января 0001 года до 23:59:59 31 декабря 9999 года.

Для создания нового объекта DateTime также можно использовать конструктор. Пустой конструктор создает начальную дату:

DateTime date1 = new DateTime();

Console.WriteLine(date1); // 01.01.0001 0:00:00

## То есть мы получим минимально возможное значение.

## Чтобы задать конкретную дату, нужно использовать один из конструкторов, принимающих параметры:

DateTime date1 = new DateTime(2015, 7, 20); // год - месяц - день

Console.WriteLine(date1); // 20.07.2015 0:00:00

## Установка времени:

DateTime date1 = new DateTime(2015, 7, 20, 18, 30, 25); // год - месяц - день - час - минута - секунда

Console.WriteLine(date1); // 20.07.2015 18:30:25

## Если необходимо получить текущую время и дату, то можно использовать ряд свойств DateTime:

Console.WriteLine(DateTime.Now);

Console.WriteLine(DateTime.UtcNow);

Console.WriteLine(DateTime.Today);

Консольный вывод:

20.07.2015 11:43:33

20.07.2015 8:43:33

20.07.2015 0:00:00

## Свойство DateTime.Now берет текущую дату и время компьютера, DateTime.UtcNow - дата и время относительно времени по Гринвичу (GMT) и DateTime.Today - только текущая дата.

Для **добавления дат** используется ряд методов:

* Add(TimeSpan value): добавляет к дате значение TimeSpan

## ▪ DateTime.Today – возвращает текущий день, месяц, год и время равное 00:00:00

## date.Today – возвращает день, месяц, год и время равное 00:00:00, для даты date

## ▪ DateTime.Year – возвращает текущий год

## date.Year – возвращает год для даты date

## ► Класс System.Object и его методы

## Все остальные классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object.

### ToString

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int i = 5;  Console.WriteLine(i.ToString()); // выведет число 5    double d = 3.5;  Console.WriteLine(d.ToString()); // выведет число 3,5 |

Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. И мы можем переопределить данный метод. Посмотрим на примере:

using System;

namespace FirstApp

{

class Program

{

private static void Main(string[] args)

{

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

Console.WriteLine(person.ToString()); // выведет название класса Person

Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };

Console.WriteLine(clock.ToString()); // выведет 15:34:53

Console.Read();

}

}

class Clock

{

public int Hours { get; set; }

public int Minutes { get; set; }

public int Seconds { get; set; }

**public override string ToString()**

**{**

**return $"{Hours}:{Minutes}:{Seconds}";**

**}**

}

class Person

{

public string Name { get; set; }

}

## }

## Для переопределения метода ToString в классе Clock, который представляет часы, используется ключевое слово override.

### Метод GetHashCode

Метод **GetHashCode** позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализацию базового типа:

class Person

{

public string Name { get; set; }

public override int GetHashCode()

{

return Name.GetHashCode();

}

## }

## Хеш-код это число соответствующее значению объекта. Это число мы получаем в результате работы некоторого метода, который должен обладать следующими *свойствами*: - он должен возвращать одинаковый хеш-код каждый раз при вызове для одного и того же объекта. - если имеется два равных (эквивалентных) объекта, то хеш-код для них должен быть одинаковым. Только это не означает, что если объекты неравны, то их хеш-коды обязательно будут разными.

### Получение типа объекта и метод GetType

## Метод GetType позволяет получить тип данного объекта:

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

Console.WriteLine(person.GetType());    // Person

### Метод Equals

Метод Equals позволяет сравнить два объекта на равенство:

class Person

{

    public string Name { get; set; }

    public override bool Equals(object obj)

    {

        if (obj.GetType() != this.GetType()) return false;

        Person person = (Person)obj;

        return (this.Name == person.Name);

    }

}

## Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

**2**

Создайте несколько объектов вашего типа. Выполните вызов конструкторов, свойств, методов, сравнение объекты, проверьте тип созданного объекта и т.п.

**3**

Создайте массив объектов вашего типа. И выполните задание, выделенное курсивом в таблице.

**КОД:**

Student[] array = new Student[] { st1, st2, st3 };

Можно и так, тогда в массив можно поместить разные типы:

object[] array = { st1, st2, st3 };

**4**

Создайте и выведите анонимный тип (по образцу вашего класса).

## ► Анонимные типы

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова **var** и инициализатора объектов:

|  |  |
| --- | --- |
| 1  2 | var user = new { Name = "Tom", Age = 34 };  Console.WriteLine(user.Name); |

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие "<>f\_\_AnonymousType0'2".

Для исполняющей среды CLR анонимные типы будут так же, как и классы, представлять ссылочный тип.

Если в программе используются несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | var user = new { Name = "Tom", Age = 34 };  var student = new { Name = "Alice", Age = 21 };  var manager = new { Name = "Bob", Age = 26, Company = "Microsoft" };    Console.WriteLine(user.GetType().Name); // <>f\_\_AnonymousType0'2  Console.WriteLine(student.GetType().Name); // <>f\_\_AnonymousType0'2  Console.WriteLine(manager.GetType().Name); // <>f\_\_AnonymousType1'3 |

Здесь user и student будут иметь одно и то же определение анонимного типа. Однако подобные объекты нельзя преобразовать к какому-нибудь другому типу, например, классу, даже если он имеет подобный набор свойств.

Следует учитывать, что свойства анонимного объекта **доступны для установки только в инициализаторе**. Вне инициализатора присвоить им значение мы не можем. Поэтому, например, в следующем случае мы столкнемся с ошибкой:

|  |  |
| --- | --- |
| 1  2 | var student = new { Name = "Alice", Age = 21 };  student.Age = 32;   // ! Ошибка |

Кроме использованной выше формы инициализации, когда мы присваиваем свойствам некоторые значения, также можно использовать **инициализаторы с проекцией** (projection initializers), когда мы можем передать в инициализатор некоторые идентификаторы, имена которых будут использоваться как названия свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class User  {      public string Name { get; set; }  }  class Program  {      static void Main(string[] args)      {          User tom = new User { Name = "Tom" };          int age = 34;          var student = new { tom.Name, age}; // инициализатор с проекцией          Console.WriteLine(student.Name);          Console.WriteLine(student.age);          Console.Read();      }  } |

В данном случае определение анонимного объекта фактически будет идентично следующему:

|  |  |
| --- | --- |
| 1 | var student = new { Name = tom.Name, age = age}; |

Названия свойств и переменных (Name и age) будут использоваться в качестве названий свойств объекта.

Также можно определять массивы объектов анонимных типов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | var people = new[]  {      new {Name="Tom"},      new {Name="Bob"}  };  foreach(var p in people)  {      Console.WriteLine(p.Name);  } |

**Зачем нужны анонимные типы?** Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если нам захочется добавить свойство, то мы сразу же на месте анонимного объекта это можем сделать. В случае с классом придется изменять еще и класс, который может больше нигде не использоваться. Типичная ситуация - получение результата выборки из базы данных: объекты используются только для получения выборки, часто больше нигде не используются, и классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

**КОД:**

var anon\_type = new

{

id = new Guid(),

surname = "Pastel",

name = "Dark",

middle\_name = "Blue",

bday\_date = new DateTime(2000, 12, 17),

Adress = "Россия Иркутск Ул 5 6",

Phone\_number = "+375294250143",

faculty = "ФИТ",

Course = 1,

Group = 7

};

Console.WriteLine(anon\_type);

**5**

Ответьте на вопросы, приведенные ниже

**Вопросы**

1. Назовите принципы ООП. Поясните каждый из них.

**▪ Инкапсуляция** ─ механизм, связывающий вместе данные и код, обрабатывающий эти данные, и сохраняющий их от внешнего воздействия и ошибочного использования.

В объектно-ориентированном языке данные и код могут быть объединены в совершенно автономный черный ящик. Внутри такого ящика находятся все необходимые данные и код. Когда код и данные связываются вместе подобным образом, создается объект. Иными словами, **объект** ─ это элемент, поддерживающий инкапсуляцию.

Основной единицей инкапсуляции в C# является **класс**, который определяет форму объекта. Он описывает данные, а также код, который будет ими оперировать.

1) Никто не знает что внутри

2) Никто не может менять данные снаружи

**Инкапсуляция в C#**

При помощи инкапсуляции класс в программе можно написать таким образом, чтобы использовать его можно было лишь как вы задумали. То есть использовать правильно.

using System;

namespace HelloApp

{

class Program

{

class Gun

{

private bool isLoaded;

private void Reload()

{

Console.WriteLine("Заряжаю--");

isLoaded = true;

Console.WriteLine("Заряжено!");

}

public void Shoot()

{

if (!isLoaded)

{

Console.WriteLine("Оружие не заряжено!");

Reload();

}

Console.WriteLine("Пыщ - Пыщ");

isLoaded = false;

}

}

static void Main(string[] args)

{

Gun gun = new Gun();

gun.Shoot();

}

}

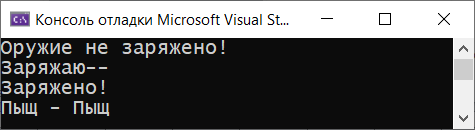
}

Здесь реализован класс Gun, у которого есть приватный метод Reload и публичный Shoot.

Правильная логика программы: зарядить оружие → выстрелить.

Но пользователю доступен лишь шаг “выстрелить”. Таким образом, мы скрыли детали реализации выстрела, ведь контроль зарядки оружия осуществляет сам класс. И возможности использовать оружие неправильно нет.

И всё работает так как мы и хотели:



**▪ Наследование** ─ способность языка позволять создавать иерархические структуры объектов. Используя наследование, можно создать общий класс, который будет определять характеристики и поведение, свойственные какому-то набору связанных объектов. В дальнейшем этот класс может быть унаследован другими, более частными классами, каждый из которых будет добавлять уникальные, свойственные только ему характеристики и дополнять или изменять поведение базового класса. В терминах C# такой общий класс называется **базовым классом** (base class), или **классом-родителем** (parent class), а класс, его наследующий, ─ **дочерним классом** (child class), или **классом-потомком** (derived class).

Пусть, например, классы Employee и Manager представляют собой абстракцию понятий «Сотрудник» и «Менеджер». Каждый менеджер также является сотрудником компании, в которой он работает. Таким образом, с точки зрения наследования, при выстраивании иерархии классов класс Employee будет являться классом-родителем, а класс Manager – дочерним классом.



**Наследование в C#**

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 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 выше определенные 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 здесь есть ряд ограничений:

* Не поддерживается множественное наследование, 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 производый 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:

|  |  |
| --- | --- |
| 1  2  3 | sealed class Admin  {  } |

* Нельзя унаследовать класс от статического класса.

### 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

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 в производных классах

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:

|  |  |
| --- | --- |
| 1  2  3  4 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string name, string company)          : base(name)  {      Company = company;  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | 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

|  |  |
| --- | --- |
| 1  2  3  4 | public Employee(string company)  {      Company = company;  } |

Фактически был бы эквивалентен следующему конструктору:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string company)      :base()  {      Company = company;  } |

**▪ Полиморфизм** ─ семейство различных механизмов, позволяющих использовать один и тот же участок программы с различными типами в различных контекстах (с различными переменными в текущем участке программы).

**Полиморфи́зм** ─ возможность объектов с одинаковой спецификацией иметь различную реализацию.





**Полиморфизм в C#**

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.

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 сотруднника 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 переопредленный 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}");

}

}

Тогда:

Employee em = new Employee("Jojo", "Anime");

em.Display();

И консольный вывод:



### 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 переопределяемых методах.

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

**▪ Абстракция** ─ подразумевает разделение и независимое рассмотрение интерфейса и реализации.

Абстракция ─ уровень описания/представления модели чего либо.

2. Назовите класс .NET, от которого наследуются все классы.

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование.

3. Охарактеризуйте открытые методы System.Object.

Класс System.Object определяет шесть открытых методов:

◦ *public virtual bool Equals(object ob)*

Этот метод определяет равенство вызывающего и передаваемого (ob) экземпляров.

Корректная реализация:

public class Object

{

public virtual Boolean Equals(Object obj)

{

//Сравниваемый объект не может быть равным null

if (obj == null) return false;

//Объекты разных типов не могут быть равны

if (this.GetType() != obj.GetType()) return false;

//Если типы объектов совпадают, возвращаем true при условии,

//что все их поля попарно равны.

//Так как в System.Object не определены поля,

//следует считать, что поля равны

return true;

}

}  
◦ *public static bool Equals(object ob1,object ob2)*  
Этот метод определяет равенство объектов ob1 и ob2, передаваемых в качестве параметров. Метод Equals() перегружен, т.к. для него определено два разных контекста.

int a = 45;

int b = 45;

Console.WriteLine(Equals(a, b)); //True

Если сравниваемые объекты оба раны null, метод возвращает true.

◦ *public virtual int GetHashCode()*Этот метод возвращает хэш-код, соответствующий вызывающему объекту. Применяется в алгоритмах, использующих хэши для доступа к экземплярам.

◦ *public Type GetType()*Этот метод возвращает тип объекта.

using System;

namespace HelloApp

{

class Program

{

class Gun

{

}

static void Main(string[] args)

{

Gun gun = new Gun();

int a = 6;

Console.WriteLine(gun.GetType()); //HelloApp.Program + Gun

Console.WriteLine(a.GetType()); //System.Int32

}

}

}

◦ *public static bool ReferenceEquals(object ob1,object ob2)*Этот метод определяет равенство ссылок ob1 и ob2, т.е. ссылаются ли они на один экземпляр.

Метод возвращает true, если ob1 и ob2 ссылаются на один и тот же экземпляр или оба равны null. Иначе метод возвращает false.

Gun gun1 = new Gun();

Gun gun2 = gun1;

Gun gun3 = new Gun();

Console.WriteLine(ReferenceEquals(gun1, gun2)); //True

Console.WriteLine(ReferenceEquals(gun1, gun3)); //False

◦ *public virtual string ToString()*Этот метод возвращает строку с описанием объекта.

→ cлужит для получения строкового представления объекта

int year = 2017;

Console.WriteLine(year.ToString()); // выведет 2017

Console.WriteLine(3.56.ToString()); //3,56

→Для классов выводит полное название класса с указанием пространства имен, в котором определен этот класс.

4. Охарактеризуйте закрытые методы System.Object.

◦ *protected void Finalize()*  
Позволяет объекту попытаться освободить ресурсы и выполнить другие операции очистки, прежде чем он будет освобожден сборкой мусора.

→ **Деструктор**, вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом

→ Реализация из Object игнорируется сборщиком мусора

→ Переопределяется если объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении

◦ *protected object MemberwiseClone()*

→ создает неглубокую копию объекта и возвращает ссылку на эту копию

→ *неглубокое копирование* ─ копируются все типы значений в классе, копируются только ссылки, а не объекты, на которые они указывают

→ не виртуальный, переопределять его реализацию нельзя

5. Приведите пример определения класса.

Класс определяется с помощью ключевого слова **сlass**:

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предок ]

{

тело класса

}

Пример:

using System;

namespace HelloApp

{

class Program

{

public class Student

{

public string name;

public string secondName;

public int course;

public void Info()

{

Console.WriteLine($"Студент {name} учится на {course} курсе");

}

}

static void Main(string[] args)

{

Student Olga = new Student();

Olga.Info();

}

}

}



6. Какие ключевые слова можно использовать при определении класса?

7. В чем отличие между объектом и классом?

(где-то в заданиях--)

8. Что такое конструктор? Когда вызывается конструктор?

(где-то в заданиях--)

9. Перечислите свойства конструктора?

Свойства конструкторов:

1) не имеет возвращаемого значения

2) имя такое же, как и у типа (класса)

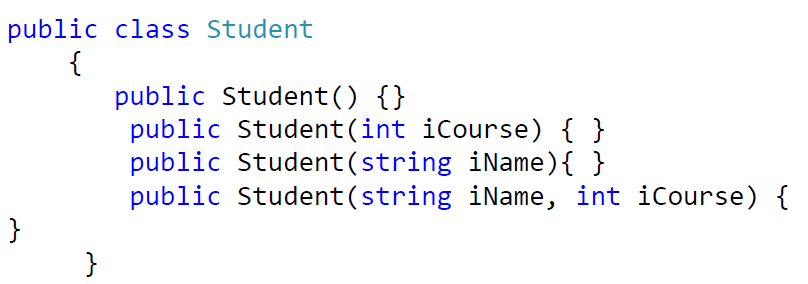
3) не наследуются

4) нельзя применять модификаторы virtual, new, override, sealed и abstract

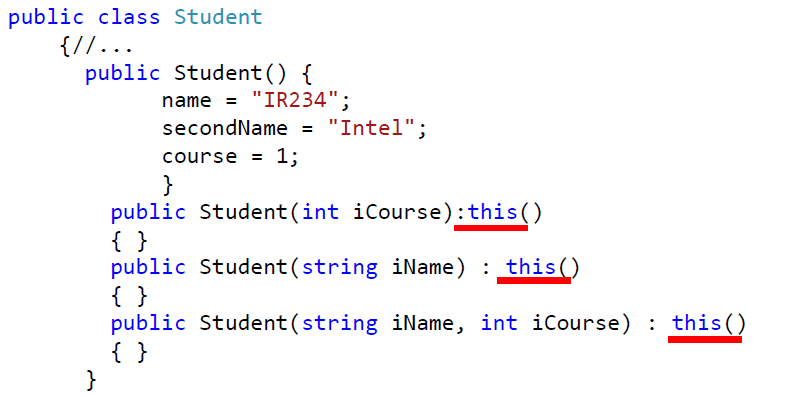
5) для класса без явно заданных конструкторов компилятор создает конструктор по умолчанию (без параметров)

6) для статических классов (sealed и abstract) компилятор не создает конструктор по умолчанию

7) может определяться несколько конструкторов, сигнатуры и уровни доступа к конструкторам обязательно должны отличаться



8) можно явно заставлять один конструктор вызывать другой конструктор посредством зарезервированного слова this



10. Что такое деструктор (destructor)?

Большинство объектов, используемых в программах на C#, относятся к управляемым или managed-коду, и легко очищаются сборщиком мусора. Однако вместе с тем встречаются также и такие объекты, которые задействуют неуправляемые объекты (низкоуровневые файловые дескрипторы, сетевые подключения и т.д.). Такие неуправляемые объекты обращаются к API операционной системы через службы PInvoke. Сборщик мусора может справиться с управляемыми объектами, однако он не знает, как удалять неуправляемые объекты. В этом случае разработчик должен сам реализовывать механизмы очистки на уровне программного кода.

Освобождение неуправляемых ресурсов подразумевает реализацию одного из двух механизмов:

* Создание деструктора
* Реализация классом интерфейса System.IDisposable

Бывают случаи в программе, когда перед освобождением памяти, выделенной под объект нужно выполнить какие то действия. *Деструктор*, это метод который вызывается для гарантированного освобождения памяти под объект. В деструкторе указываются те действия, которые необходимо выполнить прежде чем уничтожить объект.

Общая форма деструктора имеет вид:

***~имя\_класса***()

{

// код деструктора

}

где *имя\_класса* – имя конкретного класса.

У деструктора отсутствует тип, который возвращается, и список параметров.

В C# явно вызвать деструктор не получится, потому что на это нет специального оператора. Например, в языке C++ для этих целей используется оператор delete, который в языке C# не используется. Но в C++ не реализована система «сборки мусора».

Однако деструктор автоматически вызывается системой «сборки мусора» перед уничтожением памяти под объект его класса.

**Пример**. В примере объявляется класс MyClass, в котором реализован деструктор ~MyClass().

// пример класса, в котором реализован деструктор

class MyClass

{

int[] A; // внутренний массив A

// конструктор класса

public MyClass()

{

// выделение памяти для массива A

A = new int[100];

}

// деструктор класса

~MyClass()

{

// действия, которые необходимо выполнить, если произойдет "сборка мусора",

// и может быть вызван деструктор

// ...

}

}

Однако на деле при очистке сборщик мусора вызывает не деструктор, а метод **Finalize** класса Person. Все потому, что компилятор C# компилирует деструктор в конструкцию, которая эквивалентна следующей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | protected override void Finalize()  {      try      {          // здесь идут инструкции деструктора      }      finally      {          base.Finalize();      }  } |

Метод Finalize уже определен в базовом для всех типов классе Object, однако данный метод нельзя так просто переопределить. И фактическая его реализация происходит через создание деструктора.

**Свойства деструктора**

►Класс может иметь только один деструктор.

► Деструкторы не могут быть унаследованы или перегружены.

► Деструкторы невозможно вызвать. Они запускаются автоматически.

► Деструктор не принимает модификаторы и не имеет параметров.

11. Что такое this?

(где-то в заданиях--)

12. Что будет выведено в результате выполнения

class A

{

private int \_num;

public A(int num) {

Num = num;

}

public int Num {

get { return \_num; }

set { \_num = value; }

}

}

static void Main(string[] args)

{

A a = new A(5);

A b = a;

Console.WriteLine(a.Num + " " + b.Num);

a.Num = 7;

Console.WriteLine(a.Num + " " + b.Num);

}

**a** хранит ссылку на созданный объект, когда мы пишем *A b = a;*, мы создаем вторую ссылку на тот же самый объект. Таким образом, в стеке у нас две ссылки (a, b) на одни и те же данные, хранимые в управляемой куче.

Поэтому вывод в консоль:



13. Какие спецификаторы доступа для класса и методов класса существуют в C#?

Для классов: public, internal.

Для методов класса: public, private, protected, internal.

14. Опишите модификатор protected internal.

Члены с таким модификатором доступны классам, находящимся в той же сборке и унаследованным от данного класса.

15. Зачем и как используются ref и out параметры функции?

(где-то в заданиях--)

16. Приведите пример необязательных и именованных параметров метода.

## Параметры методов

Параметры позволяют передать в метод некоторые входные данные. Например, определим метод, который складывает два числа:

static int Sum(int x, int y)

{

    return x + y;

}

Метод Sum имеет два параметра: x и y. Оба параметра представляют тип int. Поэтому при вызове данного метода нам обязательно надо передать на место этих параметров два числа.

Значения, которые передаются параметрам, называются **аргументами**.

Иногда можно встретить такие определения как **формальные параметры** и **фактические параметры**. Формальные параметры - это собственно параметры метода (в данном случае x и y), а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

### Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров. Но C# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

static int OptionalParam(int x, int y, int z = 5, int s = 4)

{

return x + y + z + s;

}

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

static void Main(string[] args)

{

OptionalParam(2, 3);

OptionalParam(2, 3, 10);

}

### Именованные параметры

В предыдущих примерах при вызове методов значения для параметров передавались в порядке объявления этих параметров в методе. Но мы можем нарушить подобный порядок, используя именованные параметры:

static int OptionalParam(int x, int y, int z = 5, int s = 4)

{

return x + y + z + s;

}

static void Main(string[] args)

{

OptionalParam(x: 2, y: 3);

//Необязательный параметр z использует значение по умолчанию

OptionalParam(y: 2, x: 3, s: 10);

}

Мой пример:

using System;

namespace HelloApp

{

class Program

{

static int Sum(int a=0,int b=0,int c = 0)

{

return a + b + c;

}

static void Main(string[] args)

{

Console.WriteLine(Sum(b: 4, c: 8));

}

}

}

17. Приведите пример полей класса – статические, константные, только для чтения.

using System;

namespace HelloApp

{

class Program

{

class WhiteRabbit

{

public const string color = "white";

public static bool mealTime;

//поля для чтения можно задать ДО компиляции или же в конструкторе

public readonly DateTime bday;

public WhiteRabbit(DateTime bday)

{

this.bday = bday;

}

}

static void Main(string[] args)

{

var r1 = new WhiteRabbit(new DateTime(2020, 9, 15));

Console.WriteLine(r1.bday); //15.09.2020 0:00:00

Console.WriteLine(WhiteRabbit.color); //white

//константы это неявно статические значения!

Console.WriteLine(WhiteRabbit.mealTime); //False

}

}

}

18. Приведите пример определения свойств класса. Как свойства связаны с инкапуляцией?

using System;

namespace HelloApp

{

class Program

{

class Rabbit

{

private bool isBig = false;

public bool IsBig

{

set

{

isBig = value;

canJumpHigh = value;

}

get

{

return isBig;

}

}

private bool canJumpHigh;

public bool CanJumpHigh

{

//при получении значения проверяем isBig, если переменная равна true, то кролик может прогать высоко

get

{

return canJumpHigh;

}

}

}

static void Main(string[] args)

{

var rabbit = new Rabbit();

rabbit.IsBig = true;

Console.WriteLine(rabbit.CanJumpHigh); //True

rabbit.IsBig = false;

Console.WriteLine(rabbit.CanJumpHigh); //False

}

}

}

В данном примере я скрыла поля isBig и canJumpHigh, теперь кролик может высоко прыгать только, если он взрослый. Такое поведение настраивается за счет дополнительной логики в свойстве IsBig:

set

{

isBig = value;

canJumpHigh = value;

}

При этом пользователь сам не может настраивать поля isBig и canJumpHigh, а может только обращаться к свойствам IsBig и CanJumpHigh, где для настройки этих полей нужно соблюсти определенные правила.

19. Назовите явное имя параметра, передаваемого в метод set свойства класса?

value. Тип этого параметра определяется типом свойства.

20. Что такое автоматические свойства?

(где-то в заданиях--)

21. Что такое индексаторы класса? Какие ограничения существуют на индексатор?

**Индексаторы** позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

Формальное определение индексатора:

возвращаемый\_тип this [Тип параметр1, ...]

{

    get { ... }

    set { ... }

}

В отличие от свойств индексатор не имеет названия. Вместо него указывается ключевое слово **this**, после которого в квадратных скобках идут параметры. Индексатор должен иметь как минимум один параметр.

Посмотрим на примере.

using System;

using System.Media;

namespace HelloApp

{

class Program

{

class myClass

{

private int[] Arr = new int[] { 2, 54, 68 };

//индексатор

**public int this[int index]**

**{**

**set { Arr[index] = value; } //сохраняем значение value в массив Arr по индексу index**

**get { return Arr[index]; }**

**}**

}

static void Main(string[] args)

{

myClass ex = new myClass();

ex[1] = 666;

for(int i = 0; i < 3; i++)

{

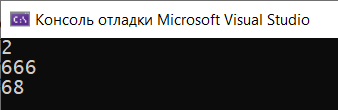
Console.WriteLine(ex[i]);

}

}

}

}



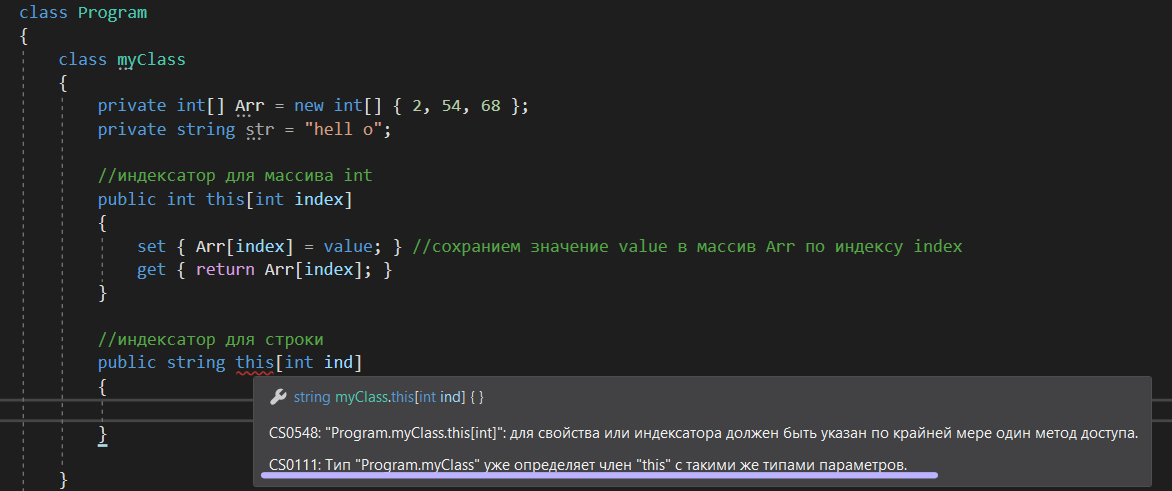
Индексатор, как полагается, получает набор индексов в виде параметров. Однако индексы необязательно должны представлять тип int.

Также индексатор может принимать несколько параметров.

Как и в свойствах, в индексаторах **можно опускать блок get или set**, если в них нет необходимости.

### Перегрузка индексаторов

Подобно методам индексаторы можно перегружать. В этом случае индексаторы **должны** отличаться по количеству, типу или порядку используемых параметров.



using System;

using System.Media;

namespace HelloApp

{

class Program

{

class myClass

{

private int[] Arr = new int[] { 2, 54, 68 };

private string str = "hell o";

//индексатор для массива int

public int this[int index]

{

set { Arr[index] = value; } //сохраняем значение value в массив Arr по индексу index

get { return Arr[index]; }

}

//индексатор для строки

**public string this[string ind]**

**{**

**get { return str; }**

**}**

}

static void Main(string[] args)

{

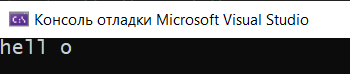
myClass ex = new myClass();

Console.WriteLine( ex["hello!!"] );

}

}

}



22. Что такое перегруженный метод?

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? **Сигнатура** складывается из следующих аспектов:

* Количество параметров
* Типы параметров
* Порядок параметров
* Модификаторы параметров

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

Например, пусть у нас есть следующий класс:

class Calculator

{

public void Add(int a, int b)

{

int result = a + b;

Console.WriteLine($"Result is {result}");

}

public void Add(int a, int b, int c)

{

int result = a + b + c;

Console.WriteLine($"Result is {result}");

}

public int Add(int a, int b, int c, int d)

{

int result = a + b + c + d;

Console.WriteLine($"Result is {result}");

return result;

}

public void Add(double a, double b)

{

double result = a + b;

Console.WriteLine($"Result is {result}");

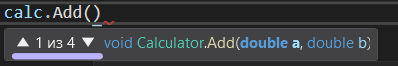
}

}

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу.

Перегрузки:



После определения перегруженных версий мы можем использовать их в программе:

static void Main(string[] args)

{

Calculator calc = new Calculator();

calc.Add(1, 2); // 3

calc.Add(1, 2, 3); // 6

calc.Add(1, 2, 3, 4); // 10

calc.Add(1.4, 2.5); // 3.9

}

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

void Increment(ref int val)

{

val++;

Console.WriteLine(val);

}

void Increment(int val)

{

val++;

Console.WriteLine(val);

}

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

23. Что такое partial класс и какие его преимущества?

(где-то в заданиях--)

Когда использовать: может понадобиться, когда какая-то часть класса написана и не будет изменяться, тогда её можно вынести в отдельный файл. Или же, когда часть класса редактируется не вручную, а, например, скриптами.

24. Что такое анонимный тип в C#?

(где-то в заданиях--)

25. Для чего делают статические классы?

(где-то в заданиях--)

26. В чем отличие статического поля от экземплярного?

Статическое поле меняется на уровне типа (вызов: Имя\_типа.Имя\_метода);  
экземплярное – на уровне объявленного экземпляра типа (вызов: Имя\_переменной. Имя\_метода).

Плюс статическое поле общее для всех экземпляров класса, то есть все объекты данного типа разделяют одну и ту же область памяти.

27. Поясните работу статических конструкторов.

(где-то в заданиях--)

28. Какая разница между поверхностным (shallow) и глубоким (deep) копированием?

При поверхностном копировании копируются значения полей класса, включая значения любых указателей или ссылок. При этом скопированные значения этих указателей и ссылок указывают на одни и те же объекты, что и в оригинальном объекте, что зачастую ведет к ошибкам. Отсюда и название такого метода копирования: мы копируем только указатели/ссылки, вместо того, чтобы делать копии этих внутренних объектов и ссылаться на них, собственно не углубляемся во внутреннюю структуру объекта. При глубоком копировании мы копируем значения полей не только на первом "уровне", но и заходим глубже, копируя все значения.

29. В чем разница между равенством и тождеством (индентичностью) объектов?

* идентичность: переменная содержит тот **же** экземпляр, что и другая переменная.
* равенство: два **различных** объекта могут использоваться взаимозаменяемо. у них часто один и тот же идентификатор.

# Идентичность

Например:

Integer a = new Integer(1);

Integer b = a;

a идентичен b .

В Java идентичность проверяется с помощью == . Например, if( a == b ) .

# Равенство

Integer c = new Integer(1);

Integer d = new Integer(1);

c равно, но не идентично d .

Конечно, две одинаковые переменные всегда равны.

В Java равенство определяется методом equals .

30. Что такое частичные классы и частичные методы?

(где-то в заданиях--)

31. Что будет выведено на консоль результате выполнения следующего кода:

class Program

{

static void Main(string[] args)

{

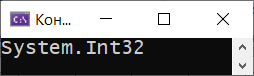
var age = 15;

Type ageType = age.GetType();

Console.WriteLine(ageType);

}

}



32. Что будет выведено на консоль в результате выполнения следующего кода:

static void Main(string[] args)

{

int a = 1, b = 2;

change(ref a, ref b);

Console.WriteLine("a = " + a + " b = " + b);

}

private static void change(ref int a,ref int b)

{

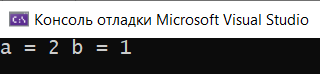
int c = a;

a = b;

b = c;

}

Значения a и b поменяются местами:



33. Пусть задан следующий класс.

internal class A

{

public A() { } //1

public int A { } //2

public A(int someI) { } //3

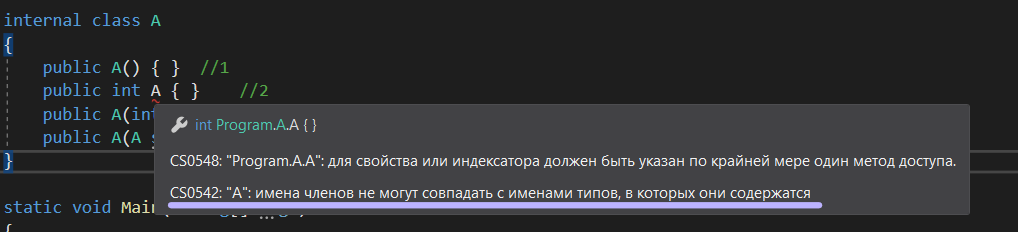
public A(A someA) { } //4

}

Какой из конструкторов задан неверно?

2, так как конструктор не имеет типа.

Компилятор думает, что мы пытаемся задать метод с именем таким же, что и имя класса.



34. Пусть задан следующий класс.

class Motorcycle

{

private int driverIntensity;

private string driverName;

public Motorcycle(int intensity=0,string name = "")

{

if (intensity > 10)

{

intensity = 10;

}

driverIntensity = intensity;

driverName = name;

}

}

Сколько аргументов может быть задано при вызове конструктора данного класса?

0 – параметры настроены по умолчанию, так что, если мы не передадим ни одного аргумента, то будут использоваться значения по умолчанию.

var m1 = new Motorcycle();

1 – мы можем передать один аргумент, в том числе второй, при помощи именованного параметра.

var m2 = new Motorcycle(1);

var m3 = new Motorcycle(name: "Jojo");

2 – передача обоих значений.

var m4 = new Motorcycle(18, "Ito");

35. Почему не удается создать объект класса A?

internal class A

{

A() { } //1

A(String st) { } //2

A(int a) { } //3

public A(int a, int b) { } //4

}

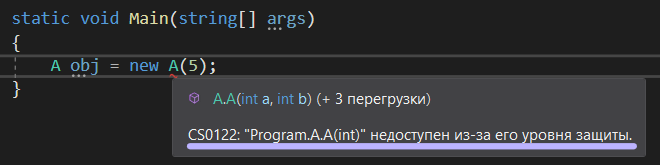
static void Main(string[] args)

{

A obj = new A(5);

}

Всего у нас 4 перегрузки конструктора. И только один, четвертый, будет доступен, так как он public. Остальные же по умолчанию private.



36. Что будет выведено в консоль при выполнении данной программы?

internal class A

{

static A() { Console.WriteLine("A static"); }

public A() { Console.WriteLine("A"); }

}

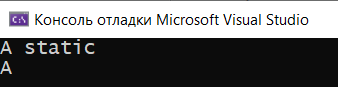
static void Main(string[] args)

{

new A();

}

Статический конструктор выполняется один раз в начале программы при первом взаимодействии с классом (создании экземпляра, обращении к статическому методу и т.д.). Так как создается экземпляр класса, то вызывается и нестатический конструктор.



37. Какая строка приведенного далее класса вызовет ошибку компиляции?

class Points

{

public readonly int a = 10;

public static readonly Int32 b = new Int32();

public static string c = "New";

private int d;

public Points()

{

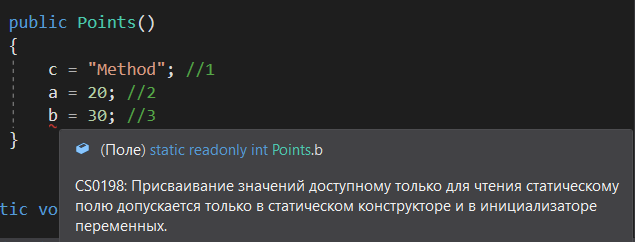
c = "Method"; //1

a = 20; //2

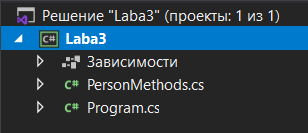
b = 30; //3

}

}

****

**КОД ВСЕЙ ЛАБЫ**



**Program.cs**

using System;

//3 ВАРИАНТ(СТУДЕНТ)

namespace Laba3

{

partial class Student

{

//переменная счетчик

private static int count = 0;

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

//статический конструктор (вызывается один раз при первом использовании класса)

static Student()

{

Console.WriteLine("Начинаем--");

}

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

//Закрытый конструктор (с модификатором private) не позволяет создавать определенного вида объекты

//То есть, например наш конструктор не принимает никаких параметров => мы не сможем создать экземпляр класса

//без передачи параметров

private Student() { }

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

////1) конструктор без параметров (из-за закрытого конструктора больше не действителен)

//public Student()

// {

// id = Guid.NewGuid();

// count++;

// }

//2) конструктор с параметрами

public Student(string surname, DateTime bday\_date, string faculty) //минимальная информация о студенте для его идентификации

{

id = Guid.NewGuid();

this.surname = surname;

this.bday\_date = bday\_date;

this.faculty = faculty;

count++;

}

//3) конструктор с параметрами

public Student(

string surname,

string name,

string middle\_name,

DateTime bday\_date,

string adress,

string phone\_number,

string faculty,

int course,

int group

)

{

id = Guid.NewGuid();

this.surname = surname;

this.name = name;

this.middle\_name = middle\_name;

this.bday\_date = bday\_date;

this.Adress = adress;

this.Phone\_number = phone\_number;

this.faculty = faculty;

this.Course = course;

this.Group = group;

count++;

}

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

//поле-константа (минимальный средний балл для получения стипендии)

public const float min\_scolarship\_score = 5.0f;

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

//поле - только для чтения

public readonly Guid id; //Guid - спец. структура для создания уникальных id

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

public string surname { get; set; }

public string name { get; set; }

public string middle\_name { get; set; }

public DateTime bday\_date { get; set; }

//в качестве адреса принимаем строку, где идет сначала страна, город, название улицы, затем номер дома и квартиры

//проверяем, что 2 последних значения числа

private string adress;

public string Adress {

set

{

string[] country\_city\_street\_house\_flat = value.Split(' ');

if (country\_city\_street\_house\_flat.Length < 5)

{

adress = $"адрес указан неправильно";

return;

}

try

{

Convert.ToUInt32(country\_city\_street\_house\_flat[3]);

Convert.ToUInt32(country\_city\_street\_house\_flat[4]);

}

catch

{

adress = $"адрес указан неправильно";

return;

}

adress = $"Страна {country\_city\_street\_house\_flat[0]}, город {country\_city\_street\_house\_flat[1]}\n" +

$"Улица {country\_city\_street\_house\_flat[2]} дом {country\_city\_street\_house\_flat[3]}, квартира {country\_city\_street\_house\_flat[4]}";

}

}

private string phone\_number;

public string Phone\_number

{

set

{

if (value.Length != 9)

{

phone\_number = "номер был указал неправильно";

}

else

{

phone\_number = "+375" + value;

}

}

get

{

return phone\_number;

}

}

public string faculty { get; set; }

private int course;

public int Course

{

set

{

if (value < 0 || value > 5)

{

course = 0;

}

else

{

course = value;

}

}

get

{

return course;

}

}

private int group;

public int Group {

set {

if (value < 0 || value > 10)

{

group = 0;

}

else

{

group = value;

}

}

get

{

return group;

}

}

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

}

class Program

{

static void Main(string[] args)

{

//Фамилия(строка), Имя(строка), Отчество(строка), дата рождения(объект типа DateTime),

//адрес(строка (Страна Город Улица Дом Квартира)), номер телефона(строка (без +375!)), факультет(строка), курс(число (1-5)), группа(число (1-10))

//создание объектов с помощью разных конструкторов

//Student st0 = new Student(); (невозможно из-за закрытого конструктора)

Student st1 = new Student("Хмельной", new DateTime(2002, 12, 17), "ФИТ");

Student st2 = new Student("Творожный", "Сырок", "Фундук-какао", new DateTime(2021, 9, 22),

"Беларусь Слуцк Тутаринова 45 0", "179555502", "Клецкая крыначка", 1, 8);

Student st3 = new Student("Хмельной", new DateTime(2000, 10, 23), "ФИТ");

Student st4 = new Student("Ито", "Шинджи", "-", new DateTime(1988, 12, 22),

"Япония Токио Улица 14 0", "379455502", "Манга", 1, 7);

Student st5 = new Student("Мицухаши", "Такаши", "-", new DateTime(1988, 9, 22),

"Япония Токио Улица 14 0", "178885502", "Манга", 1, 7);

//

Console.WriteLine("\n\*\*\*Количество созданных объектов типа Student\*\*\*");

Student.amount\_of\_obj();

//

Console.WriteLine("\n\*\*\*Использование метода с ref-параметром (получение адреса)\*\*\*");

string adress = "";

st2.getAdress(ref adress);

Console.WriteLine(adress);

//

Console.WriteLine("\n\*\*\*Сравнение объектов\*\*\*");

Console.WriteLine(st2.Equals(st3));

Console.WriteLine(st1.Equals(st3));

//

Console.WriteLine("\n\*\*\*Хеш-коды\*\*\*");

Console.WriteLine(st1.GetHashCode());

Console.WriteLine("-------------------------------");

Console.WriteLine(st2.GetHashCode());

Console.WriteLine("-------------------------------");

Console.WriteLine(st3.GetHashCode());

//

Console.WriteLine("\n\*\*\*Объекты в виде строк\*\*\*");

Console.WriteLine(st2.ToString());

Console.WriteLine(st3.ToString());

//

Console.WriteLine("\n\*\*\*Функция с out-параметром (переход студента на следующий курс)\*\*\*");

st2.nextCourse();

//

Console.WriteLine("\n\*\*\*Тип созданного объекта\*\*\*");

Console.WriteLine(st2.GetType());

//

Console.WriteLine("\n\*\*\*Получение значений некоторых свойств объекта\*\*\*");

Console.WriteLine($"Имя : {st2.name}, фамилия : {st2.surname}, отчество : {st2.middle\_name}");

//

Console.Write("\n////////////////////// РАБОТА С МАССИВОМ //////////////////////\n");

Student[] array = new Student[] { st1, st2, st3, st4, st5 };

Console.Write("=> Введите факультет, студентов которого необходимо вывести на консоль : ");

string fac = Console.ReadLine();

//если не будет найдено студентов с указанного факультета, то в переменной student\_is сохранится значение false, иначе true (\*)

bool student\_is = false;

foreach(Student st in array)

{

if (st.faculty == fac)

{

Console.WriteLine($"\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" +

$"ID : {st.id}\n" +

$"Surname : {st.surname}\n" +

$"Name : {st.name}\n" +

$"Middle name : {st.middle\_name}\n" +

$"Birthday date : {st.bday\_date}\n" +

$"Phone number : {st.Phone\_number}\n" +

$"Faculty : {st.faculty}\n" +

$"Course : {st.Course}\n" +

$"Group : {st.Group}\n" +

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

student\_is = true; //(\*)

}

}

if (!student\_is)

{

Console.WriteLine("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\nНа указанном факультете нет студентов\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

}

Console.Write("=> Введите номер группы, студентов которой необходимо вывести на консоль : ");

int gr = Convert.ToInt32(Console.ReadLine());

student\_is = false;

foreach (Student st in array)

{

if (st.Group == gr)

{

Console.WriteLine($"\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" +

$"ID : {st.id}\n" +

$"Surname : {st.surname}\n" +

$"Name : {st.name}\n" +

$"Middle name : {st.middle\_name}\n" +

$"Birthday date : {st.bday\_date}\n" +

$"Phone number : {st.Phone\_number}\n" +

$"Faculty : {st.faculty}\n" +

$"Course : {st.Course}\n" +

$"Group : {st.Group}\n" +

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

student\_is = true;

}

}

if (!student\_is)

{

Console.WriteLine("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\nВ указанной группе нет студентов\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");

}

//

Console.WriteLine("\*\*\*Получение возрастов студентов\*\*\*");

Console.WriteLine($"Студент с датой рождения: {st1.bday\_date} => возраст {st1.age()} лет");

Console.WriteLine($"Студент с датой рождения: {st2.bday\_date} => возраст {st2.age()} лет");

//

Console.WriteLine($"\n\*\*\*Анонимный тип\*\*\*");

var anon\_type = new

{

id = new Guid(),

surname = "Pastel",

name = "Dark",

middle\_name = "Blue",

bday\_date = new DateTime(2000, 12, 17),

Adress = "Россия Иркутск Ул 5 6",

Phone\_number = "+375294250143",

faculty = "ФИТ",

Course = 1,

Group = 7

};

Console.WriteLine(anon\_type);

}

}

}

**PersonMethods.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace Laba3

{

partial class Student

{

//вывод количества созданных объектов класса Student

public static void amount\_of\_obj()

{

Console.WriteLine("Количество созданных экземпляров класса Student : " + count);

}

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

//вывод информации о студенте

public override string ToString()

{

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

$"ID : {id}\n" +

$"Surname : {surname}\n" +

$"Name : {name}\n" +

$"Middle name : {middle\_name}\n" +

$"Birthday date : {bday\_date}\n" +

$"Phone number : {Phone\_number}\n" +

$"Faculty : {faculty}\n" +

$"Course : {Course}\n" +

$"Group : {Group}\n" +

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

}

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

//хеш-код студента

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.bday\_date.ToString()).Split(' ', ':', '.');

int sum = 0;

//конвертирует 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 фамилии

foreach(char i in this.surname)

{

sum += (int)i;

}

//и символов факультета

foreach (char i in this.faculty)

{

sum += (int)i;

}

//полученная сумма, умноженная на день рождения и есть хеш-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 Student

Student st = (Student)obj;

return (

this.surname == st.surname &&

this.bday\_date == st.bday\_date &&

this.faculty == st.faculty

);

}

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

//переход студента на следующий курс

public void nextCourse()

{

//вызываем метод, в котором в переменную mes запишется сообщение о переводе

changeCourse(out string mes);

Console.Write(mes);

}

private void changeCourse(out string mes)

{

Course++;

mes = $"\nСтудент переведен на {Course} курс\n";

}

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

//создадим метод для получения адреса студента

public void getAdress(ref string adress)

{

adress = this.adress;

}

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

//расчет возраста студента

public int age()

{

var today = DateTime.Today;

var age = today.Year - this.bday\_date.Year;

//отнимаем полученный возраст от сегодняшней даты

//если в этом году у студента ещё не было дня рождения, то результат будет меньше даты дня рождения студента

//так как age - это возраст, которого достигнет человек в этом году, то отнимаем 1 для получения текущего возраста

if (this.bday\_date.Date > today.AddYears(-age)) age--;

return age;

}

}

}