# ООП

otus.ru


Проверить, идет ли запись

# Напишите «+» в чат, если меня слышно и видно

Тема урока

# ООП

***Нилов Павел***

*Преподаватель курса C# Professional, C# Basic*

*Контакты:t.me/@NilovPavel*

# Правила вебинара

Активно
участвуем

Задаем вопрос
в чат

Вопросы вижу в чате,
могу ответить не сразу

# Маршрут вебинара

1. Кратко об ООП

2. Инкапсуляция

3. Наследование

4. Решение задач

5. Ответы на вопросы

# Цели вебинара

| | |
|---|---|
| 1. | Узнать, что такое ООП. |
| 2. | Основные механизмы ООП. |
| 3. | Особенности синтаксиса ООП. |

# Кратко об ООП

# Функциональное программирование

До ООП в разработке использовался другой подход — процедурный. Программа представляется в нем как набор процедур и функций — подпрограмм, которые выполняют определенный блок кода с нужными входящими данными.

Код ассемблера:

```
mov ax, 0x6h                ; заносим в AX число 6
mov cx, 0x8h                ; заносим в CX число 8
mov dx, cx                  ; копируем CX в DX, DX = 6
add dx, ax    ; DX = DX + AX
```

Код C:

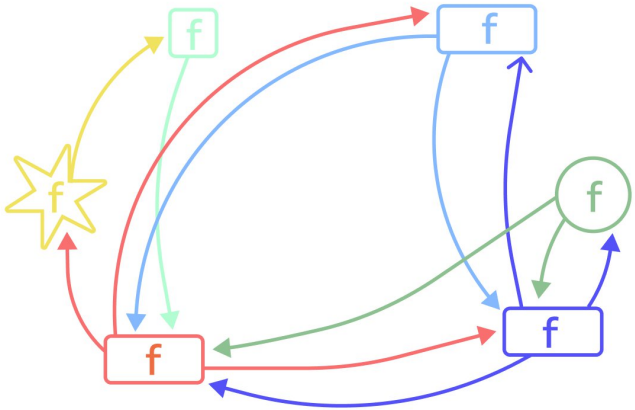
```
int main() {
    int a = 6;
    int b = 8;
    int sum = a + b;
    return 0;
}
```

# Функциональное программирование

Функциональное программирование хорошо подходит для легких программ без сложной структуры. Но если блоки кода большие, а функций сотни, придется редактировать каждую из них, продумывать новую логику. В результате может образоваться много плохо читаемого, перемешанного кода («спагетти-кода).

Минусы использования функционального программирования **для крупных проектов**:

1. Сложно управлять кодом в виду того, что становится сложно декомпозировать код.
2. Отсутствовала инкапсуляция, в виду этого была нарушена связь между кодом и данными.
3. Необходимо было копировать код и, как следствие, его дублировать.



# Что такое ООП?

Объектно-ориентированное программирование (ООП) — это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение.
ООП – это программирование с помощью классов и объектов.

Всё вокруг нас является **объектом.**

У объекта есть **свойства** (еще называют параметры)

У объекта есть **методы** (методы – это действия, то есть что может делать объект)

Например, машина – это объект.

У любой машины есть такие свойства: модель, цвет, размер и т.д.

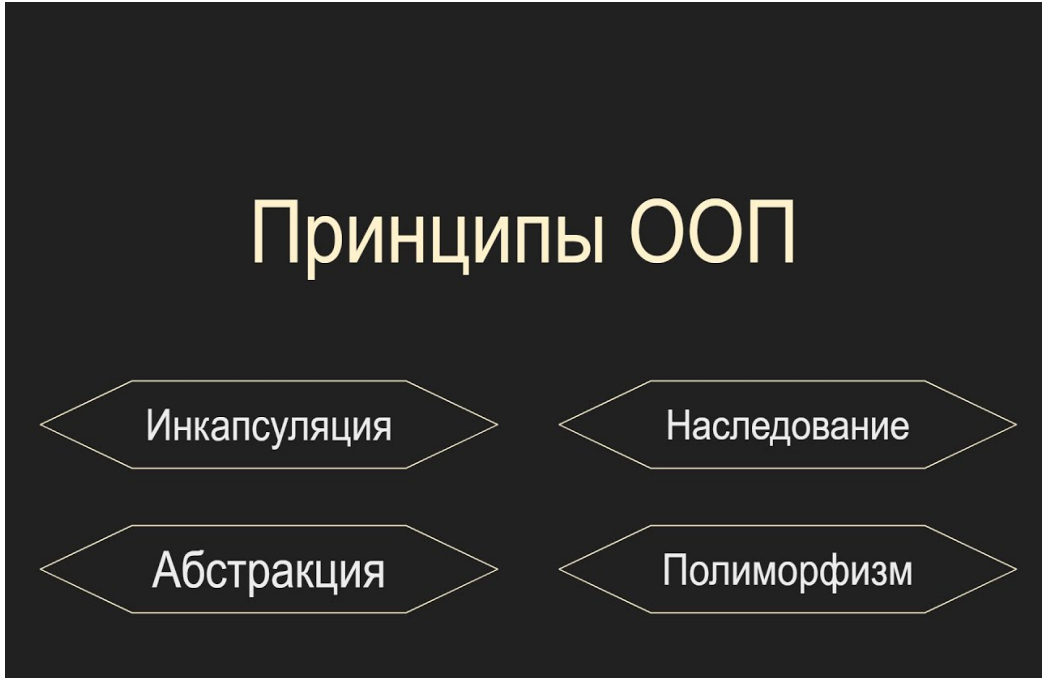
Методы машины:
затормозить ();
нажатьНаГаз ();
открытьДверь ();
закрытьДверь ();
и т.д.

Например, котёнок – это объект.

У любого котенка есть свойства: порода, имя, цвет, длина шерсти, возраст и т.д.

Методы котенка:
спать();
кушать();
играть ();
шкодить ();
и т.д.

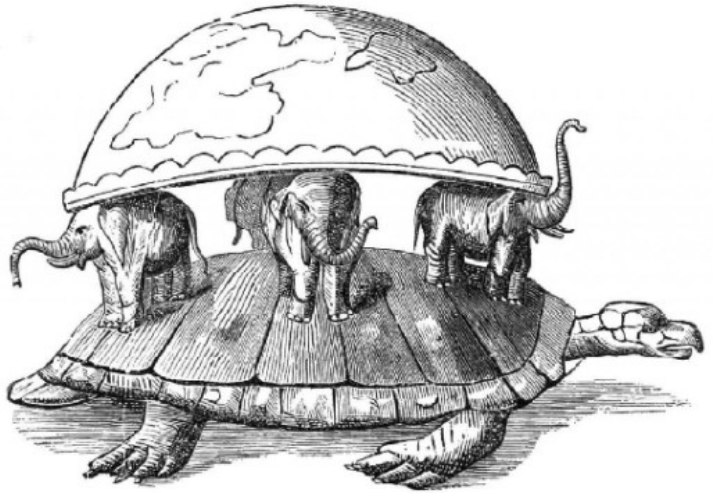
# Что из перечисленного ниже НЕ является принципом ООП?



# Принципы ООП

Объектно-ориентированное программирование основано на следующих принципах:

1. **Инкапсуляция**
2. **Наследование**
3. **Полиморфизм (типов)**
4. **Абстракция**

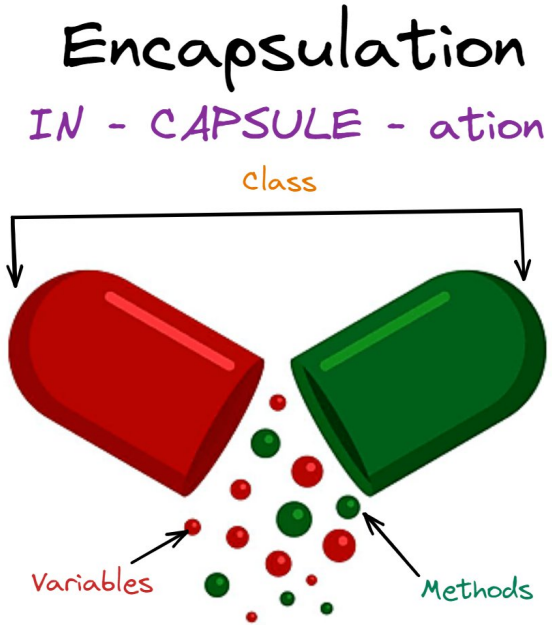


# Инкапсуляция

# Что такое инкапсуляция?

Инкапсуляция заключается в объединении данных и методов, которые с ними работают, внутри одного объекта.

При этом доступ к данным объекта ограничивается (через модификаторы доступа), чтобы скрыть внутреннюю реализацию и предоставить только необходимый интерфейс.



# Содержимое класса

В C# класс может содержать:

1. Поля (Fields)
2. Свойства (Properties)
3. Методы (Methods)
4. Конструкторы (Constructors)

**Класс:**
программист

**Объект:**
разработчик Иван

**Атрибуты:**
зарплата, обязанности

**Методы:**
написание кода

```
class Coder
{
   //Поле(я)
   private string name;
   private string salary;

   //Свойство(а)
   public string Name
   {
      get {return this.name;} }   }

   //Метод(ы)
   public void WriteCode()
   { }

   //Конструктор(ы)
   public Coder(string name)
   {   this.name = name;    }
}
```

# Права доступа

Права доступа к членам класса в программировании нужны для управления видимостью и доступом к данным и методам объекта. Они обеспечивают **инкапсуляцию**, один из ключевых принципов объектно-ориентированного программирования (ООП), помогая скрыть внутреннюю реализацию объекта и предоставляя только необходимый для работы интерфейс.

| Модификатор | Описание |
| --- | --- |
| public | Член доступен из любого места, где виден объект. |
| private | Член доступен только внутри класса, в котором он определён. Это значение по умолчанию. |
| protected | Член доступен в классе и его наследниках. |
| internal | Член доступен внутри текущей сборки (assembly). |

# Модификаторы доступа на бытовых примерах

public предоставляет возможность миру взаимодействовать с созданным объектом.

Пример: завести автомобиль ключом зажигания. *Для того, чтобы запустить двигатель, человек не должен залезать в двигатель, у него есть интерфейс (публичный метод запуска двигателя).*

```
class Car
{
    public void RunEngine()
    { }
}
```

# Модификаторы доступа на бытовых примерах

private не предоставляет миру возможность взаимодействовать с созданным объектом, а служит только для внутреннего использования.

Пример: двигатель автомобиля передает крутящий момент(GetTwist). Все, что происходит внутри двигателя: подача топлива, подача воздуха и пр. – скрыто от мира. Все узлы, агрегаты, характеристики – тоже.

```
class Engine
{
    public void GetTwist()
    { }

    private void GetAir()
    { }

    private int angularVelocity;
}
```

# Модификаторы доступа на бытовых примерах

protected не предоставляет миру возможность взаимодействовать с созданным объектом и служит только для использования в наследовании.

Пример: все (в любом случае большинство) автомобили состоят из одних и тех же элементов. Скажем, что у любого автомобиля есть такой показатель, как скорость, но у разных автомобилей она может быть разной.


```
class Car
{
   protected virtual int GetSpeed()
   {
      return 0;
   }
}

class SuperCar : Car
{

   protected override int GetSpeed()
   {
      return 200;
   } //Переопределение не обязательно
}
```

# Best Practice

1. Всегда обозначайте минимально-необходимый модификатор доступа.
2. Для членов класса действует правила:
2.1 Для полей – модификатор private
2.2 Для свойств – public
2.3 Для методов – зависит от способа использования:
2.3.1 Методы, использующиеся только внутри кода класса – private
2.3.2 Методы, использующие объект извне – public
2.3.3 Методы, которые должны быть унаследованы, но не доступны извне – protected

# Инкапсуляция дает следующую информацию

Итого, инкапсуляция определяет:

1. Как выглядит создаваемый объект во внешнем мире.
2. Какими характеристиками он обладает.
3. Какое поведение он реализует.

# Решение задач

Задачи:
1. Опишите класс Student, который:
1.1 Содержит имя.
1.2 Предоставляет имя, через свойство.
1.3 Содержит возраст.
1.4 Предоставляет возраст через метод.
1.5 Содержит конструктор, принимающий имя студента и возраст.
1.6 Содержит поле, показывающее, является ли студент совершеннолетним.
1.7 При инициализации поля имени в конструкторе вычисляет, является ли студент совершеннолетним в приватном методе.

2. Опишите класс Dog
2.1 Содержит имя.
2.2 Содержит состояние: enum { sit, lie, stand }
2.3 Содержит метод, выполняющий команду "{Имя}, сидеть". Меняет состояние собаки на соответствующее.
2.4 Содержит метод, выполняющий команду "{Имя}, лежать". Меняет состояние собаки на соответствующее.

# Наследование

# Что такое наследование?

**Наследование** – это механизм, который позволяет использовать возможности других классов.

Наследование позволяет определить дочерний класс, который использует (наследует), расширяет или изменяет возможности родительского класса.

Класс, члены которого наследуются, называется *базовым классом*. Класс, который наследует члены базового класса, называется *производным классом*.

# Синтаксис наследования

Наследоваться в рамках языка C# допустимо от:

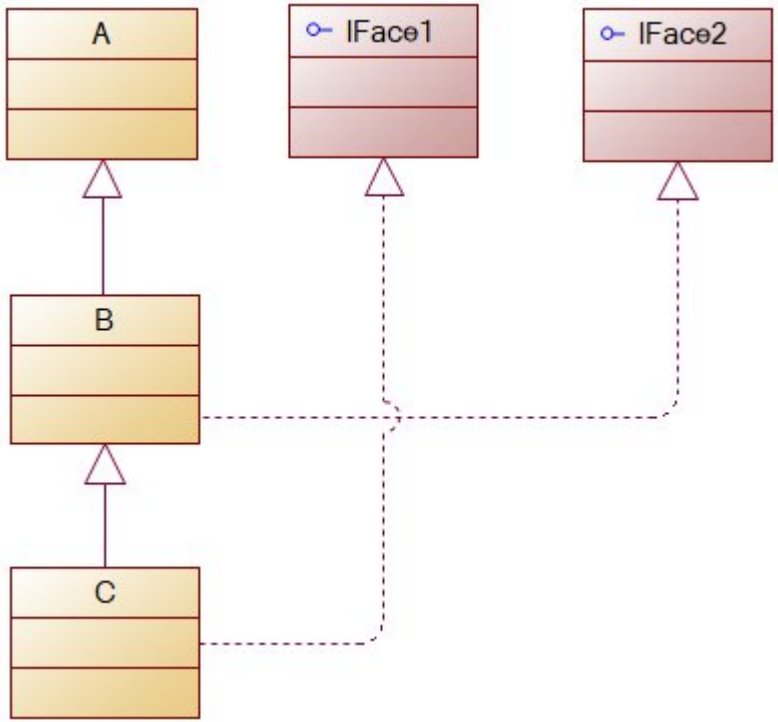
1. Классов.
2. Интерфейсов. В случае интерфейсов, как правило – это *реализация*.

Наследование обозначается символом «**:**», следующего после имени класса с указанием всех типов, перечисленных через «**,**», от которых он наследуется. Стоит отметить, что классы наследуются от классов, а в случае с интерфейсами – реализуют их.

Класс может наследоваться только от одного класса, и *быть реализован* от любого количества интерфейсов.

```
class {className}
            : {abstract classNameParent}
            , {interfaceName1}
            …
            , {interfaceNameN}
{
   //Реализация интерфейсных методов
}
```

# Пример синтаксиса

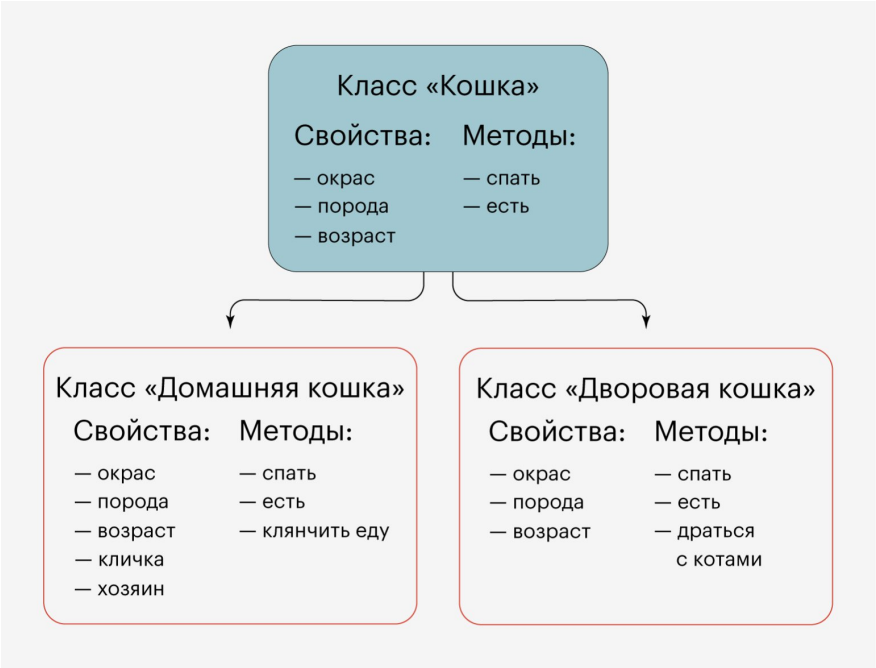



```
abstract class A
{
   //код класса
}
interface IFace1
{
   //код интерфейса
}
interface IFace2
{
   //код интерфейса
}
class B : A, IFace2
{
   //код класса
}
class C : B, IFace1
{
   //код класса
}
```

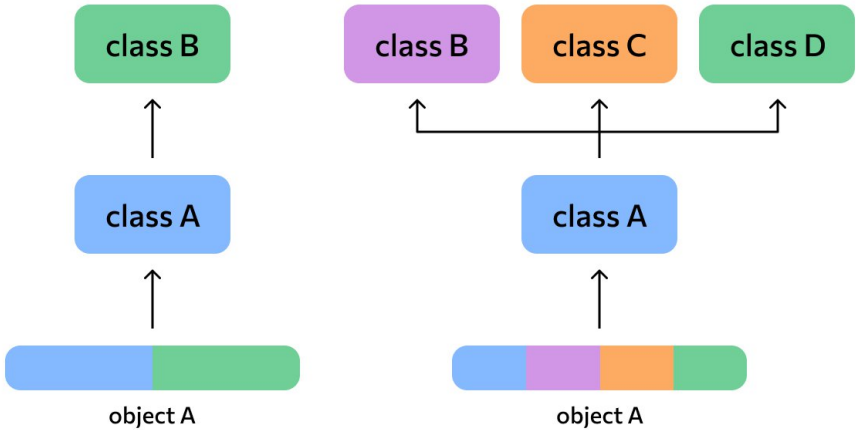
# Что наследуется

В C#, при наследовании у родительского (базового) класса наследуются все **публичные** и **защищенные** (protected) члены класса, включая:

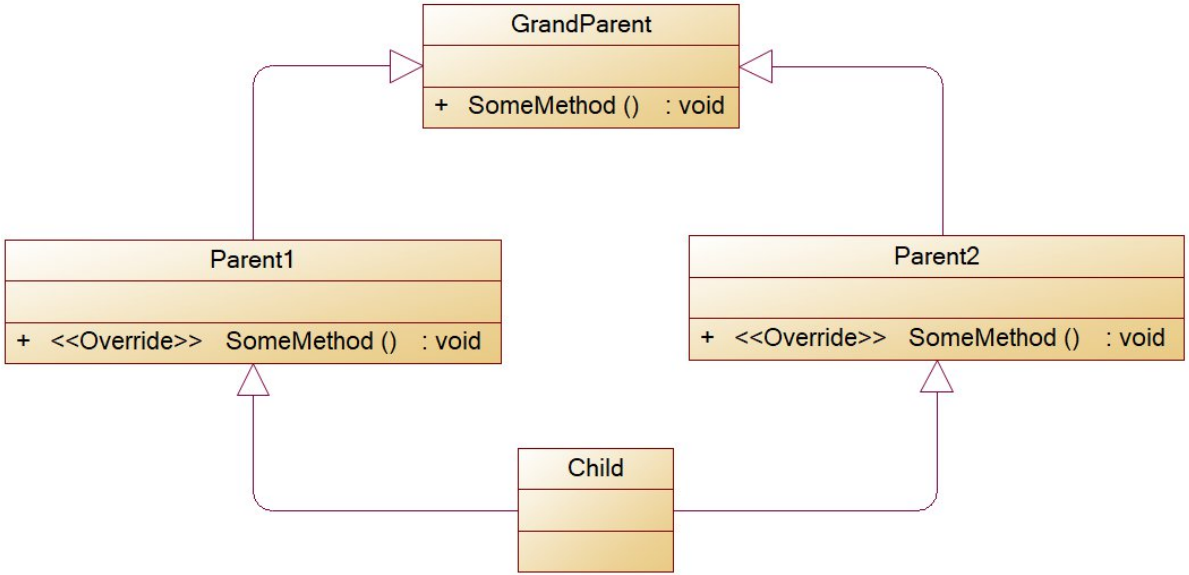
1. **Поля** (если они имеют protected или public модификаторы).
2. **Методы** (с protected или public модификаторами).
3. **Свойства** (с protected или public модификаторами).



# Множественное наследование запреще



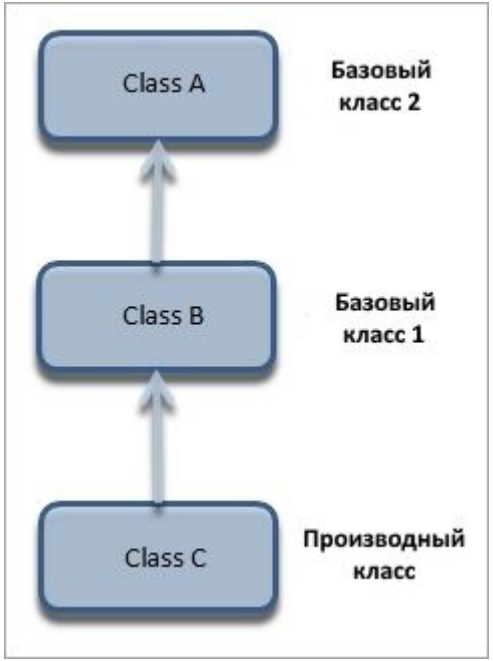
# Почему в C# множественное наследование запрещено




Реализац

# Транзитивное наследование

Транзитивное наследование позволяет определить иерархию наследования для набора типов. Другими словами, тип **C** может наследовать возможности типа **B**, который в свою очередь наследует от типа **A**. Благодаря транзитивности наследования члены типа **A** будут доступны для типа **C**.



# Ключевое слово base

Ключевое слово base используется для доступа к членам базового класса из производного класса.
Используйте его, если вы хотите:

1. Вызвать метод базового класса.
2. Определить конструктор базового класса, который должен вызываться при создании экземпляров производного класса.

```
public class BaseClass
{
   protected void GetInfo() { }

   public BaseClass() {  }
}

public class DerivedClass : BaseClass
{
  private void SomeMethod()
  {
       base.GetInfo();
  }

  public DerivedClass() : base() {  }
}
```

# Подраздел: нюансы наследования

# Некоторые нюансы наследования

В рамках .Net классами **не наследуются**:
**Классы и методы, помеченные ключевым словом** sealed**(запечатанный).**

Также есть нюансы, о которых не стоит забывать

1. **Конструкторы**. Код конструктора класса-потомка будет воспроизводить код класса-предка. При этом сначала будет выполняться код класса-предка, потом – класса-потомка. Конструкторы **не унаследованы** — они **не становятся доступными** в потомке автоматически. Конструктору нужно задать явное наследование от конструктора базового класса через ключевое слово base.
2. **Статичные члены (**static**).** Статичные члены класса будут наследовать поведение класса-предка.
3. **Закрытые члены (**private**)**. К приватным полям и методам класса напрямую обращаться нельзя. При этом допустимо использовать методы для работы с данными членами.

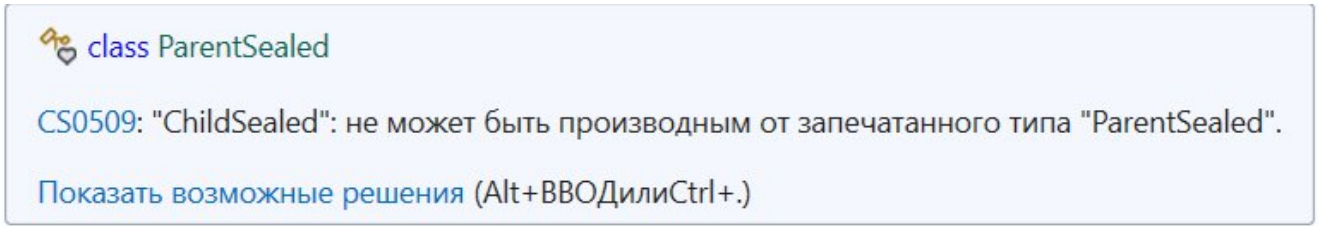
# Запечатанные(sealed) классы

От запечатанного (sealed) класса наследование запрещено.

Будет выдана ошибка компиляции.

```
sealed class ParentSealed
{
}

class ChildSealed : ParentSealed
{
}
```



# Запечатанные(sealed) методы

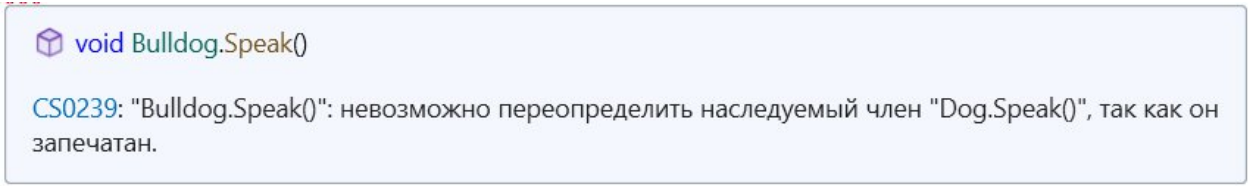
Переопределить sealed-метод нельзя.

Будет выдана ошибка компиляции.

```
class Animal
{
    public virtual void Speak() { }
}

class Dog : Animal
{
    // Метод переопределен и запечатан (sealed)
    public sealed override void Speak() {}
}

// Попытка переопределения метода приведет к ошибке
class Bulldog : Dog
{
    // Ошибка компиляции! Speak запечатан в Dog.
    public override void Speak() { }
}
```



# Конструкторы

Конструкторы **не унаследованы** — они **не становятся доступными** в потомке автоматически. Однако конструктор ОБЯЗАН вызвать конструктор базового класса. Сначала выполняется код конструктора родителя, а затем код конструктора потомка.

Если в базовом классе нет конструктора без параметров — обязателен base.

```
class Animal
{
    public Animal(string name) { }
}
class Dog : Animal
{
    public Dog(string name) : base(name) { }
}
```

Исключением являются конструкторы без параметров.

```
class Animal
{
    public Animal() {}
}
class Cat : Animal
{
    public Cat() { }
}
```

# Статические члены

Для статических членов справедливо следующее:

1. Статические члены доступны в производном классе.
2. Их поведение привязано к типу, а не к объекту.
3. Не переопределяются.

```
Derived.SayHello(); // Hello from Derived
Base.SayHello();    // Hello from Base

class Base
{
   public static void SayHello()
   {
      Console.WriteLine("Hello from Base");
   }
}

class Derived : Base
{
   public static void SayHello()
   {
      Console.WriteLine("Hello from Derived");
   }
}
```

# Приватные члены

Приватные члены в C# **не наследуются в прямом смысле** — к ним **нет доступа из кода в производном классе**, даже если находятся в иерархии.
Но! Они **существуют в объекте** потомка (то есть память выделена), и к ним можно получать **доступ косвенно** через защищённые или публичные методы/свойства базового класса.

❌ Прямой доступ — невозможен

```
class Parent
{
    private int secret = 42;
}

class Child : Parent
{
    public void ShowSecret()
    {
        // ❌ Ошибка компиляции:
        // 'secret' недоступен
        Console.WriteLine(secret);
    }
}
```

✅ Косвенный доступ через protected-метод

```
class Parent
{
    private int secret = 42;
    protected int GetSecret() => secret;
}

class Child : Parent
{
    public void RevealSecret()
    {
        Console.WriteLine(
                $"Inherited secret: {GetSecret()}");
    }
}
```

# Вопросы, на которые отвечает наследование

Итого:

1. Какими свойствами должен обладать объект, созданный из класса-наследника.
2. Какое поведение будет реализовывать объект, созданный из класса наследника.

# Решение задач

1 Описать класса-родителя и класс наследника

1.1 Описать класс Person

1.1.1 Персона содержит поле "имя".

1.1.2 Персона содержит метод, который возвращает значение поля.

1.1.3 Персона содержит конструктор, который принимает 1 параметр - имя.

1.2 Описать класс Employee

1.2.1 Работник содержит поле "Компания".

1.2.4 Работник содержит конструктор, который принимает имя и название компании как параметры. (использовать base)

1.3 Описать класс Coder

1.3.1 Программист содержит конструктор, переопределяемый Employee с теми же параметрами (base)

1.3.2 Программист содержит метод, в котором сообщает свое имя через обращение к методу базового класса (base)

# Вопросы?

Задаем
вопросы в чат

Ставим “–”,
если вопросов нет