# ЛЕКЦИЯ 3

- Модуль 2
- 08.11.2023
- Наследование

# ЦЕЛИ ЛЕКЦИИ

- Познакомится с реализацией наследования в C#
- Разобраться с модификаторами доступа и их влиянием на наследование членов классов

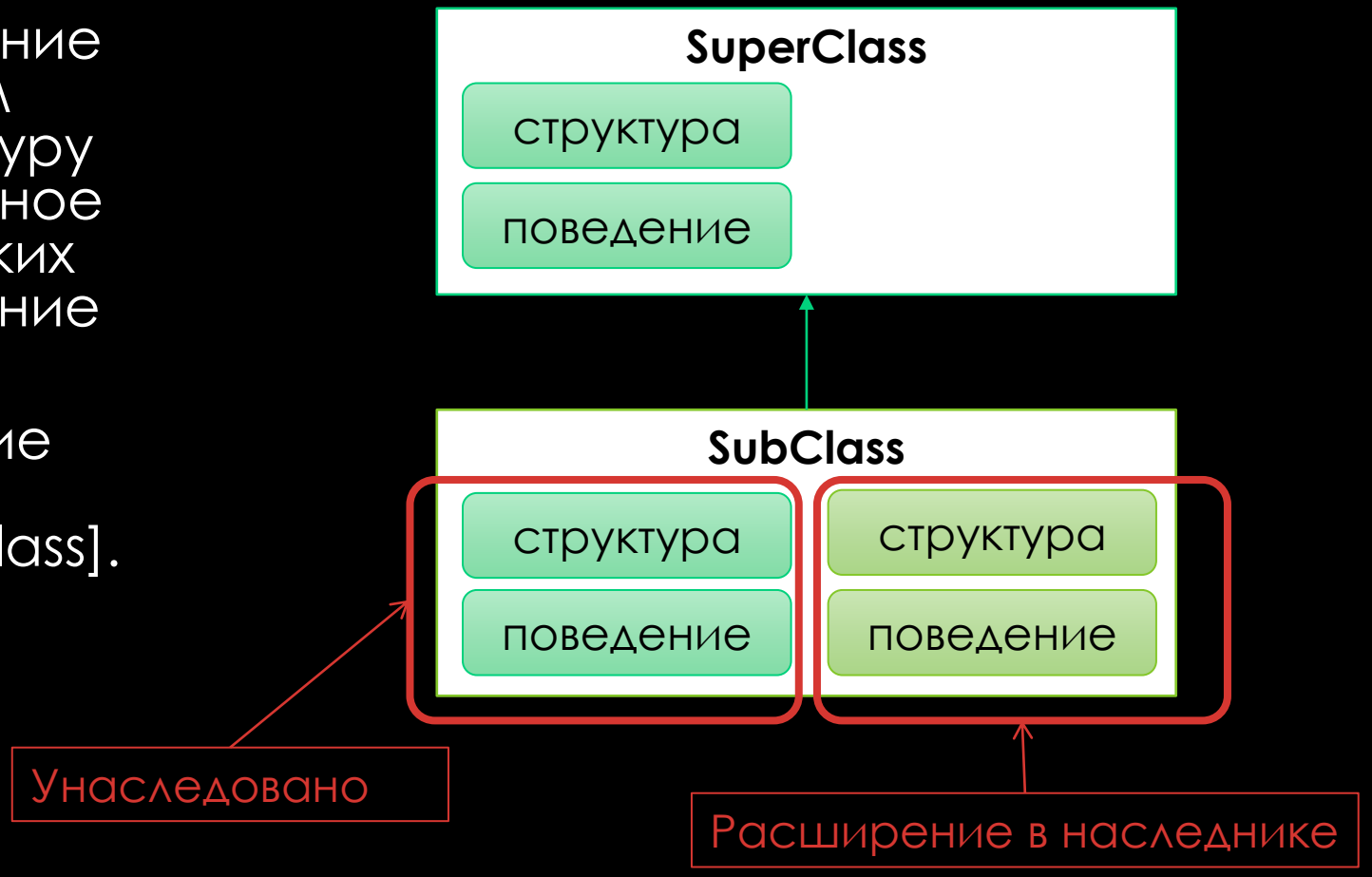
# НАСЛЕДОВАНИЕ

Общие определения наследования

Реализация наследования в C#

# ОТНОШЕНИЕ НАСЛЕДОВАНИЯ

- **Наследование** – это отношение между классами, в котором один класс повторяет структуру и поведение одного (одиночное наследование) или нескольких (множественное наследование классов
- Класс, структура и поведение которого наследуются, называется **базовым** [superclass].
- Класс, производный от суперкласса, называется **производным** [subclass]



# НАСЛЕДОВАНИЕ, КАК ОТНОШЕНИЕ МЕЖДУ КЛАССАМИ

**IS A**

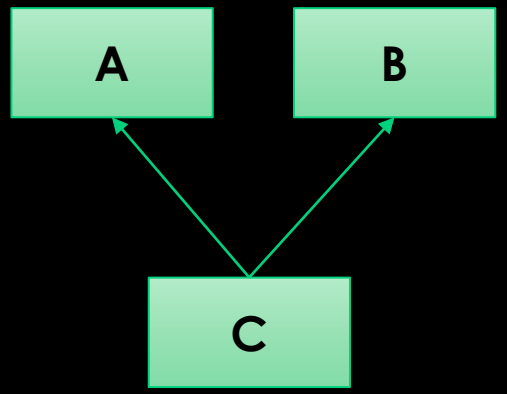
Если класс **A** не является разновидностью класса **B**, то класс **A** не может быть подклассом класса **B**

В диаграммах стрелка направлена от наследника к родителю

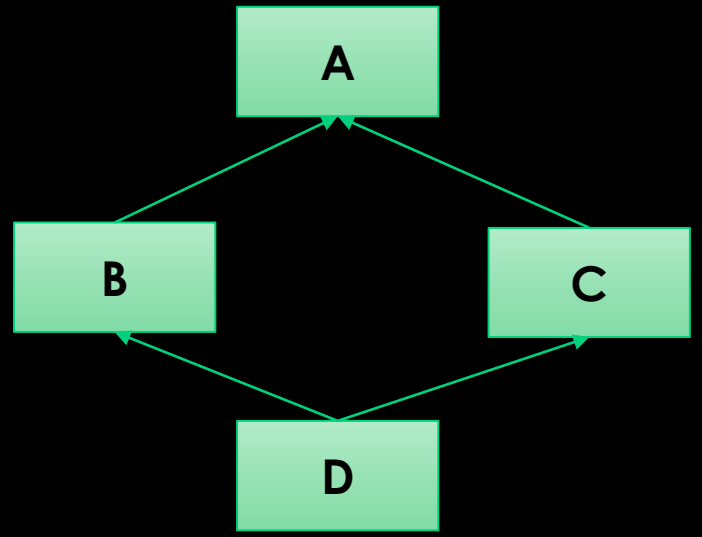
# МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

**Основные проблемы, связанные со множественным наследованием:**

- Как разрешить конфликты имён?
- Что делать с повторным наследованием?




Множественное наследование




Повторное наследование

# ОБЪЯВЛЕНИЕ ПРОИЗВОДНОГО КЛАССА

**Идентификатор производного класса (базы)**

```
class OtherClass  : SomeClass
{
```

**двоеточие** **базовый класс (база)**

```
}
```

```
class Animal
{

}
class Cat : Animal
{

}
```

Множественное наследование классов в C# запрещено

# ПРИМЕР НАСЛЕДОВАНИЯ

```
// Класс родитель.
public class A
{
    int _x;
    public int X { get => _x; }
}
// Наследник A.
public class B : A
{

}
```

```
// Ещё наследник A.
public class C : A
{

}
```

```
A objA = new A();
B objB = new B();

objA = objB;

objB = objA; // Явно так нельзя.
```

В ссылку с родительским типом может быть назначен любой объект-наследник

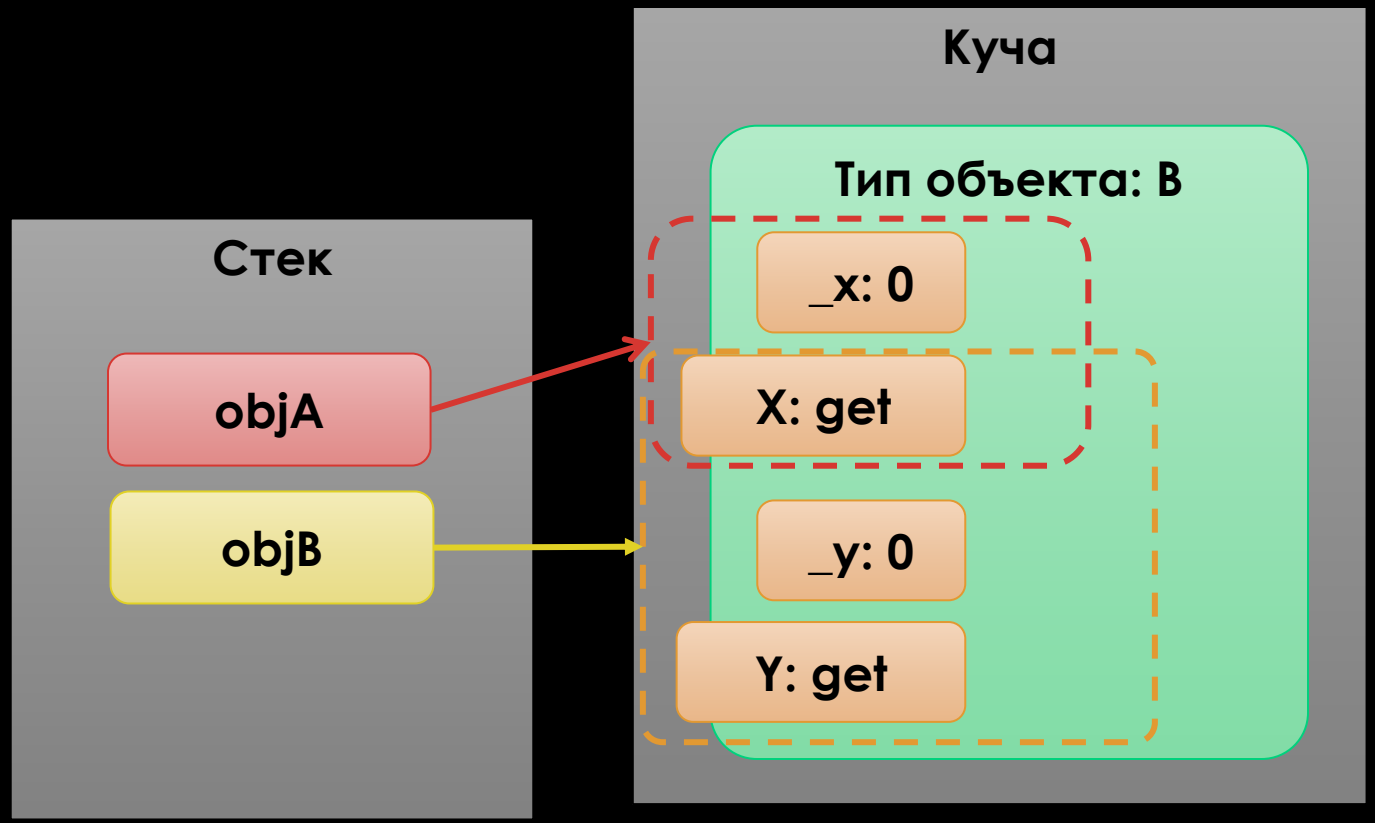
```
objB = (B)objA;
```

```
A[] objs = { new A(), new B(),new C() };
```

# ДОСТУПНОСТЬ ЧЛЕНОВ КЛАССА ПРИ НАСЛЕДОВАНИИ

```
// Класс родитель.
public class A
{
    int _x;
    public int X { get => _x; }
}
// Наследник A.
public class B : A
{
    int _y;
    public int Y { get => _y; }
}
```

```
A objA;
B objB = new B();

objA = objB;
```



# НАСЛЕДОВАНИЕ ИЗ OBJECT

```
// Класс родитель.
public class A
{
    int _x;
    public int X { get => _x; }
    public override string ToString() => this.GetType().Name.ToString();
}
// Наследник A.
public class B : A { }
// Ещё наследник A.
public class C : A { }
```

Переопределяем ToString() для возврата имени класса

```
Object[] objects = { new A(), new B(), new C() };
foreach (Object o in objects)
{
    Console.WriteLine(o.ToString());
}
```

Массив ссылок с типом object

# ПРИМЕР ИСПОЛЬЗОВАНИЯ ОПЕРАЦИИ IS

```
// Класс родитель.
public class A
{
    int _x;
    public int X { get => _x; }
    public override string ToString() => this.GetType().Name.ToString();
}
// Наследник A.
public class B : A { }
// Ещё наследник A.
public class C : A { }
```

```
Object[] objects = { new A(), new B(), new C() };
foreach (Object o in objects)
{
    if (o is C)
    {
        Console.WriteLine(o.ToString());
    }
}
```

Проверяем тип объекта, доступного по ссылке

# НАСЛЕДОВАНИЕ ОТ OBJECT

```
class ExClass
{

}
```

**неявно**

```
class ExClass : Object
{

}
```

**явно**

# ИЕРАРХИЯ КЛАССОВ

В коде (реализация)

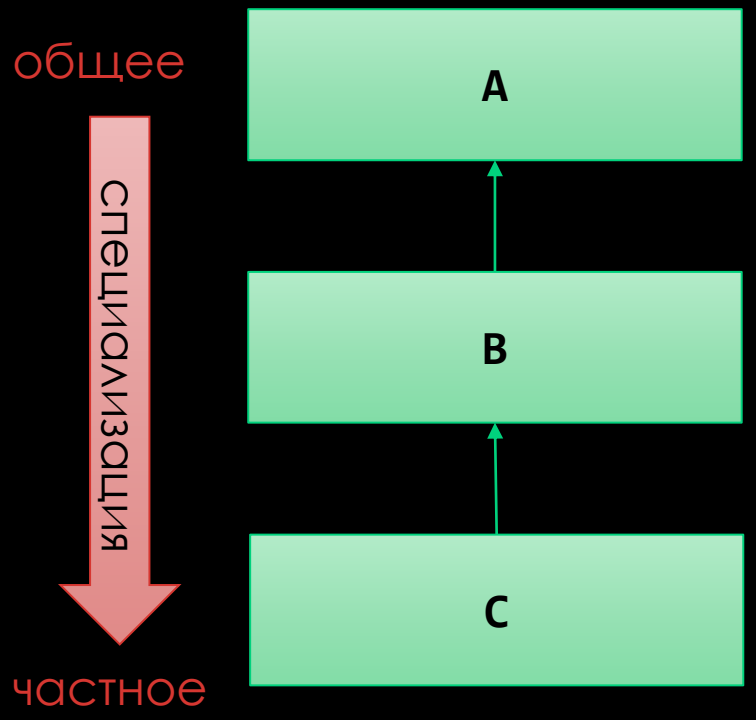
```
class A { }

class B : A { }

class C : B { }
```

общее

специализация

частное

В понимании иерархии типов



# КОНСТРУКТОРЫ КЛАССОВ ПРИ НАСЛЕДОВАНИИ

Поведение конструкторов при наследовании

Работа с конструкторами при наследовании

# КОНСТРУКТОРЫ ПРИ НАСЛЕДОВАНИИ

- Производный класс должен содержать описание собственных конструкторов
- **Конструкторы базового класса** доступны в производном классе, но **автоматически не наследуются**
- Конструкторы базового класса всегда выполняются первыми

# КОНСТРУИРОВАНИЕ ОБЪЕКТА ПРИ НАСЛЕДОВАНИИ

- Инициализируются поля производного класса
- Вычисляются параметры конструктора базового класса
- Инициализируются поля базового класса
- выполняются инструкции конструктора базового класса
- Выполняются инструкции конструктора производного класса

# ЭТАПЫ СОЗДАНИЯ ОБЪЕКТА

```
class MyDerivedClass : MyBaseClass     {
   int MyField1 = 5;                 // 1. инициализация поля
   int MyField2;                      //     инициализация поля
   public MyDerivedClass()           // 3. выполнение тела конструктора
   {   ...    }
}

class MyBaseClass          {
   public MyBaseClass()    // 2. выполнение конструктора базового класса
   {   ...    }
}
```

# КОНСТРУКТОРЫ ПРИ НАСЛЕДОВАНИИ

```
class Bar
{
    public int Field { get; set; }
    public Bar() { Console.WriteLine($"1::{this.GetType().Name.ToString()}");}
}
class Foo : Bar
{
    public int Field { get; set; }
    public Foo() { Console.WriteLine($"2::{this.GetType().Name.ToString()}"); }
}
```

Что произойдёт при конструировании объекта типа Foo?

```
Foo f = new Foo();
```

# КОНСТРУКТОРЫ ПРИ НАСЛЕДОВАНИИ: ПРОВЕРЯЕМ СЕБЯ

В этом коде есть ошибки, найдём их и исправим

```
class Bar
{
    public int Field { get; set; }
    public Bar(int x) => Field = x;
}
class Foo : Bar
{
    public int Field { get; set; }
    public Foo(int x) => Field = x;
}
```

Здесь неявно вызывается конструктор без параметров базового типа

# ИНИЦИАЛИЗАТОР В КОНСТРУКТОРЕ

инициализатор конструктора

↓

```
public MyClass(int x) : this(x, "Using Default String")
{
```

↑

ключевое. слово

==========

инициализатор конструктора

↓

```
public MyDerivedClass( int x, string s ) : base( s, x )
{
```

↑

ключевое слово

Ключевое слово **base**:

- позволяет обращаться к переопределённым членам базового класса из методов производного класса
- применяется для вызова конструктора базового класса

```
class Bar
{
    protected int _field;
    public int Field {
        get => _field;
        set => _field = value;
    }
    public Bar(int x) => Field = x;
}
class Foo : Bar
{
    public int Field
    {
        get => _field;
        set => _field = Math.Abs(value);
    }
    public Foo(int x) : base(x) { }
}
```

Что будет выведено на экран?

```
Foo f = new Foo(5);
Console.WriteLine(f.Field);
```

Добавим еще один уровень наследования

Явный вызов конструктора из базового типа

```
class Bar
{
    protected int _field;
    public int Field {
        get => _field;
        set => _field = value;
    }
    public Bar(int x) => Field = x;
}
class Foo : Bar
{
    public int Field
    {
        get => _field;
        set => _field = Math.Abs(value);
    }
    public Foo(int x) : base(x) { }
}
```

```
class Baz : Foo
{
    public int Field
    {
        get => _field;
        set => _field = base.Field + 1;
    }
    public Baz(int x) : base(x) { }
}
```

```
Baz f = new Baz(-5);
Console.WriteLine(f.Field);
```

Что будет выведено на экран?

Какие есть варианты, чтобы Foo и его наследники не обращались в родительское свойство?

```
class Bar
{
    protected int _field;
    public int Field {
        get => _field;
        set => _field = value;
    }
    public Bar() { }
    public Bar(int x) => Field = x;
}
```

```
class Foo : Bar
{
    public new int Field
    {
        get => _field;
        set => _field = Math.Abs(value);
    }
    public Foo() { }
    public Foo(int x) => Field = x;
}
```

```
class Baz : Foo
{
    public new int Field
    {
        get => _field;
        set => _field = base.Field + 1;
    }
    public Baz() { }
    public Baz(int x) : base(x) { }
}
```

```
Bar[] bars = { new Bar(5), new Foo(-6),
       new Baz(-7) };
foreach(Bar b in bars)
{
    Console.WriteLine(b.Field);
}
```

# УРОВНИ ДОСТУПА И НАСЛЕДОВАНИЕ

Модификаторы доступа к членам классов

Наследование между сборками

# МОДИФИКАТОРЫ ДОСТУПА К КЛАССАМ

ключевое слово

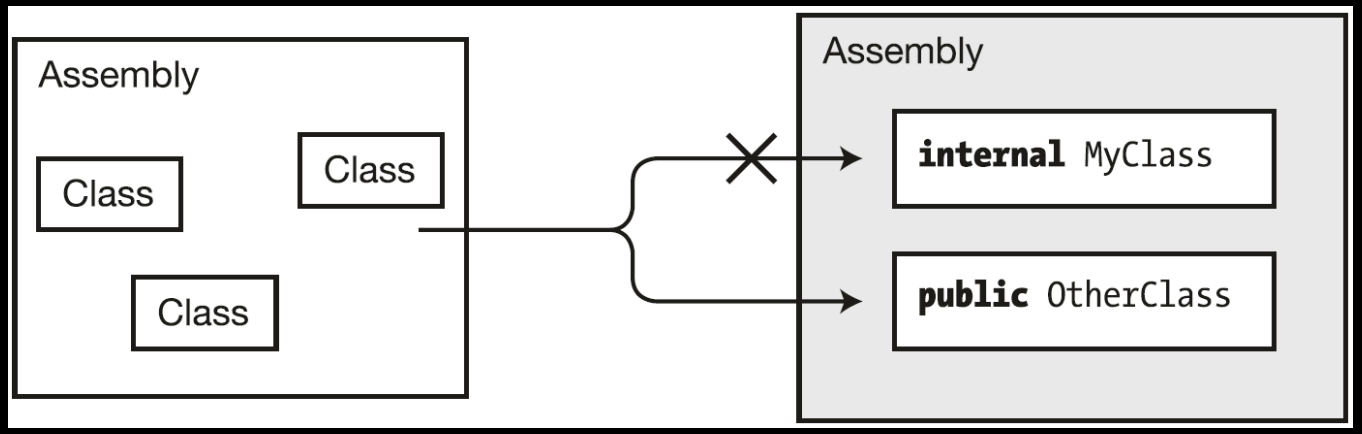
↓

```
public class MyBaseClass
{ ...
```

ключевое слово

↓

```
internal class MyBaseClass
{ ...
```



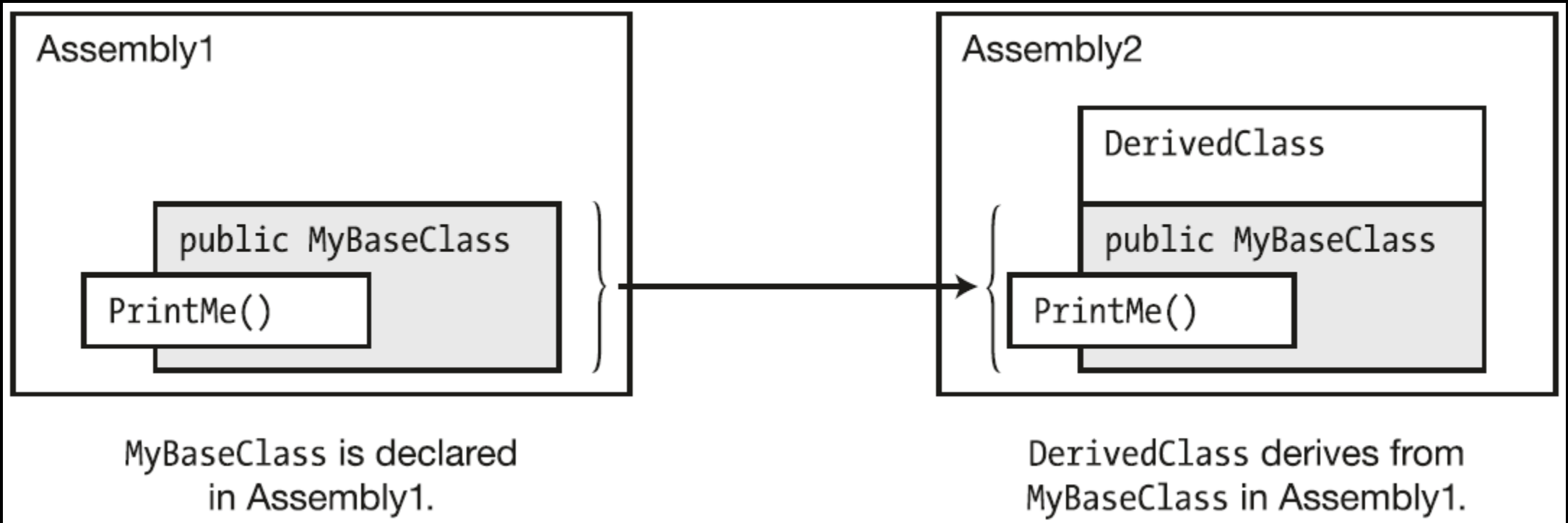
# НАСЛЕДОВАНИЕ МЕЖДУ СБОРКАМИ (INHERITANCE BETWEEN ASSEMBLIES )

```
// Assembly1.cs
namespace BaseClassNS {
    public class MyBaseClass {
        public void PrintMe() {
            Console.WriteLine("I am MyBaseClass");
        }
    }
}
```

```
// Assembly2.cs
using BaseClassNS;
namespace UsesBaseClass  {
    class DerivedClass: MyBaseClass {
            // пустое тело
    }
    class Program {
        static void Main( )        {
            DerivedClass mdc = new DerivedClass();
            mdc.PrintMe();
        }
    }
}
```

Это не сборка наследует сборку, а механизм работы наследования классов между сборками!!! Сборки – не типы

# НАСЛЕДОВАНИЕ МЕЖДУ СБОРКАМИ



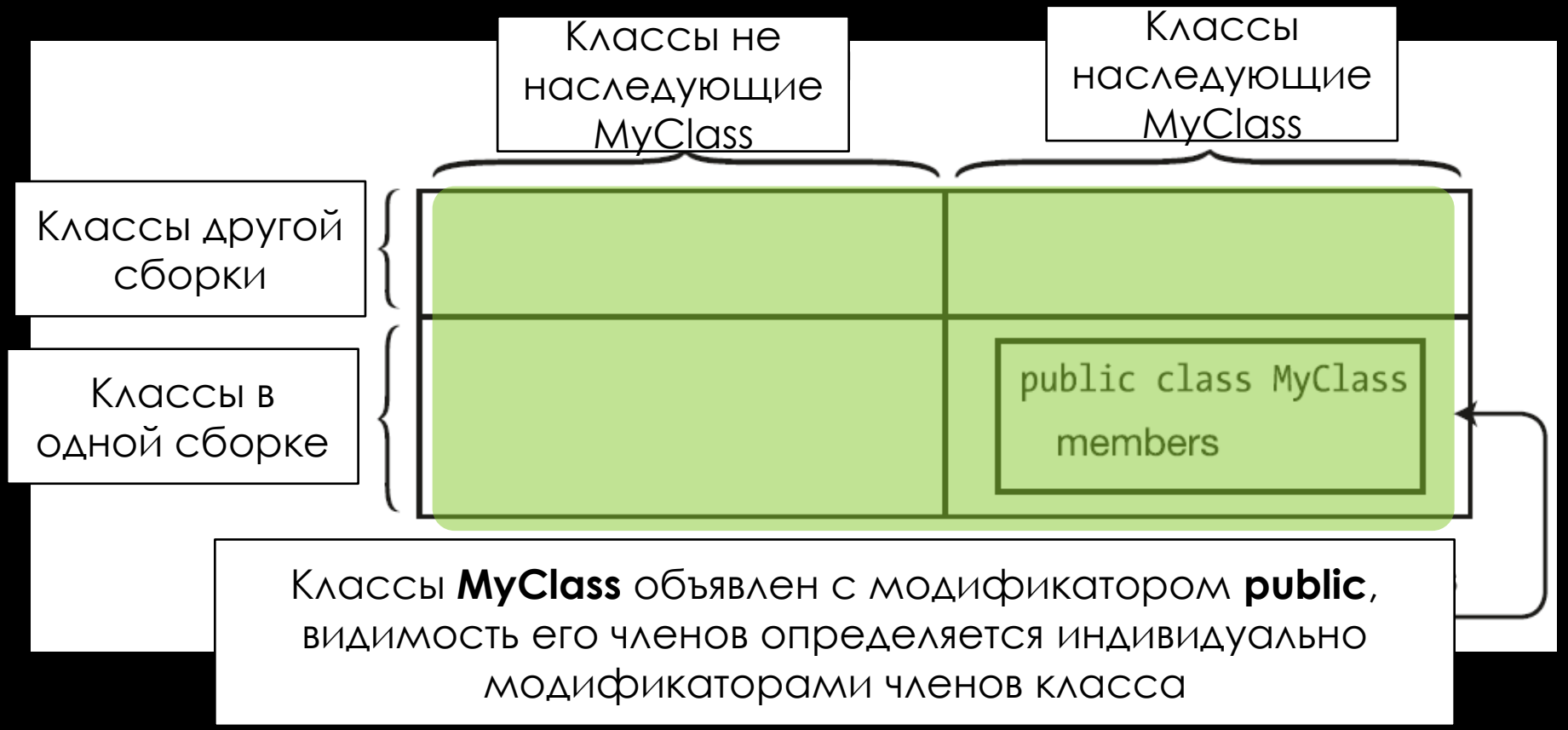
# МОДИФИКАТОРЫ ДОСТУПА К ЧЛЕНАМ ТИПА

**Существует шесть уровней доступа:**

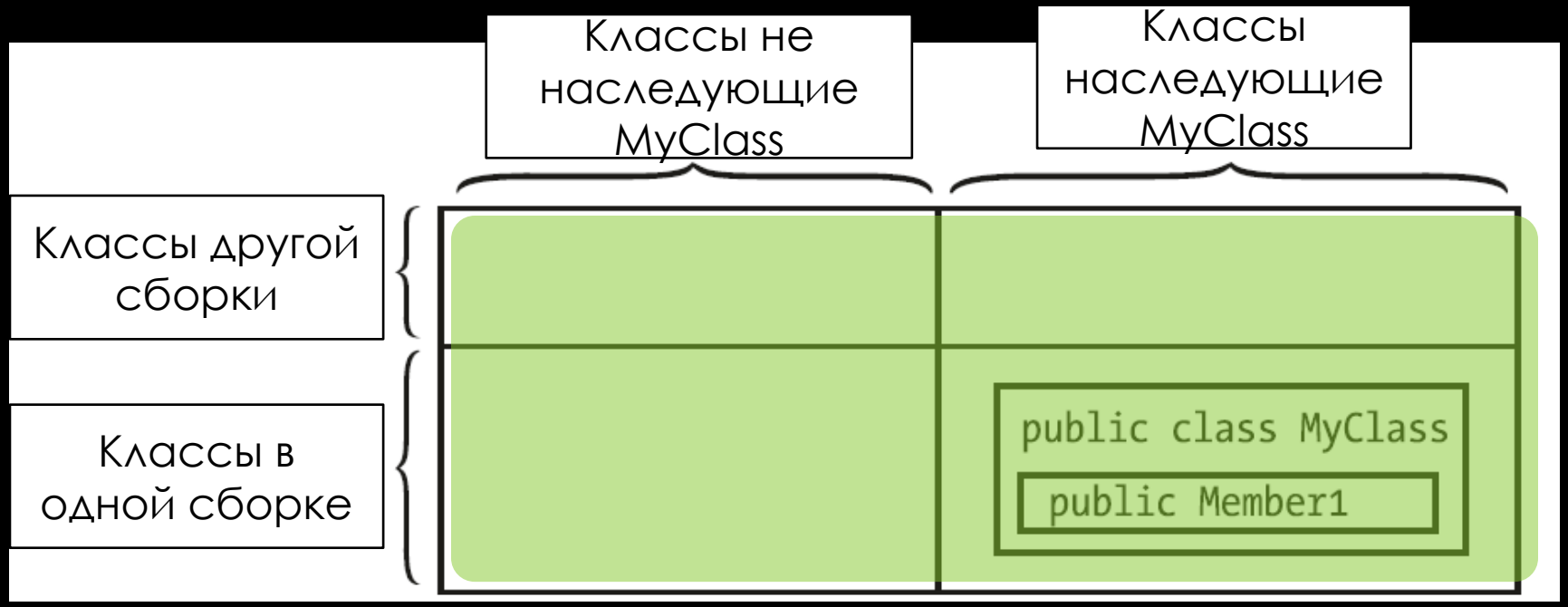
- public
- private
- protected
- internal
- protected internal
- private protected (C# 7.2)

```
public class MyClass    {
  public                int Member1;
  private               int Member2;
  protected             int Member3;
  internal              int Member4;
  protected internal    int Member5;
  private protected     int Member6;
  ...
}
```

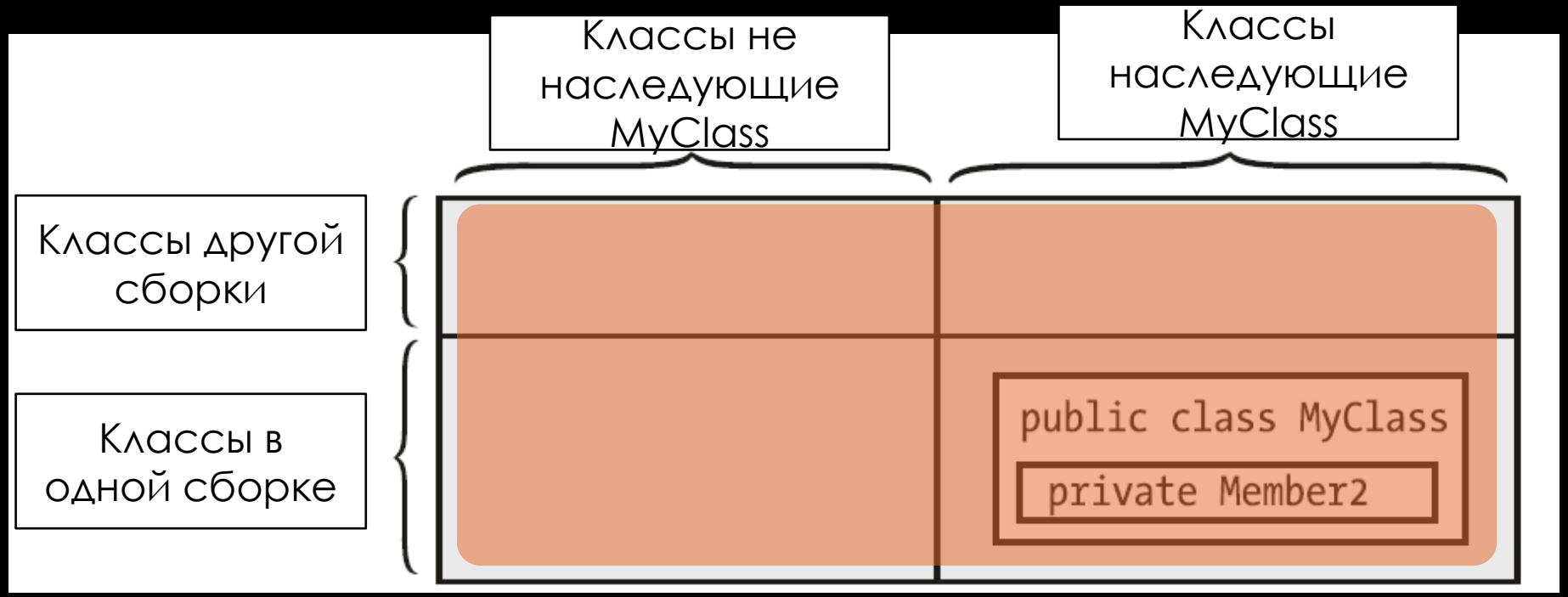
# ДОСТУП: PUBLIC КЛАСС



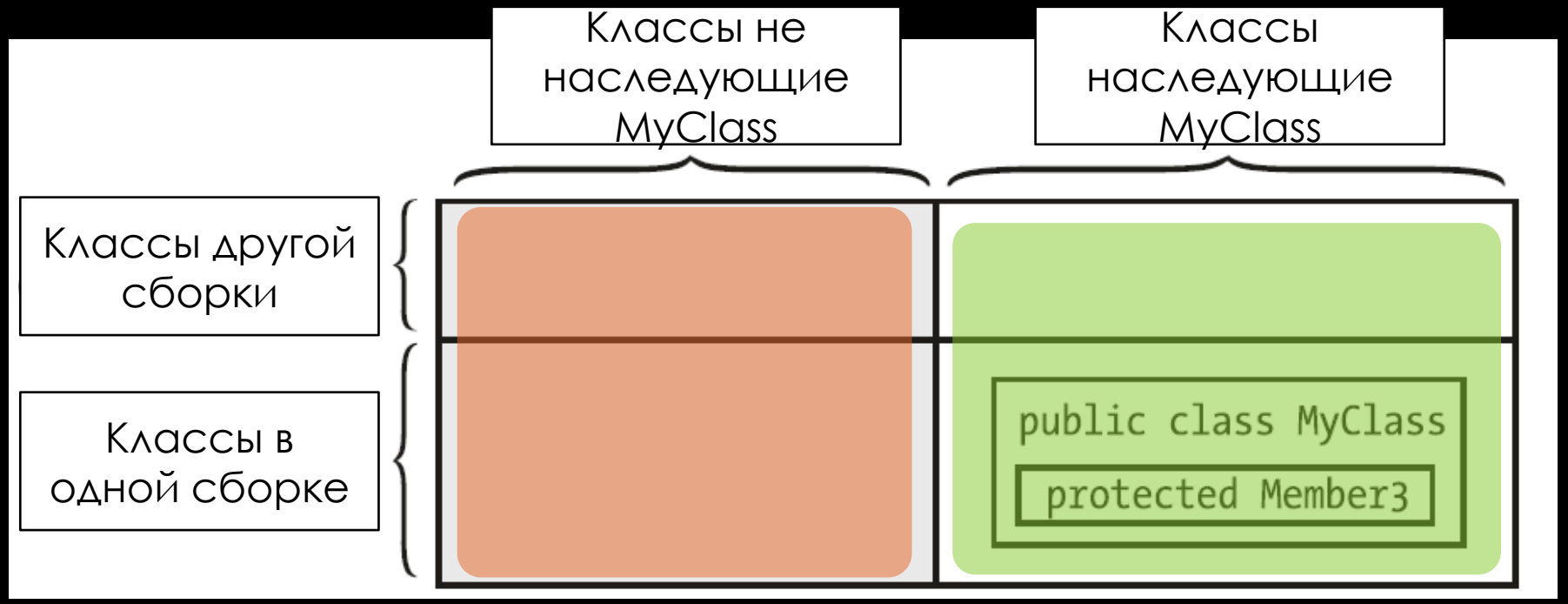
# ДОСТУП: PUBLIC-КЛАСС И PUBLIC-ЧЛЕН КЛАССА



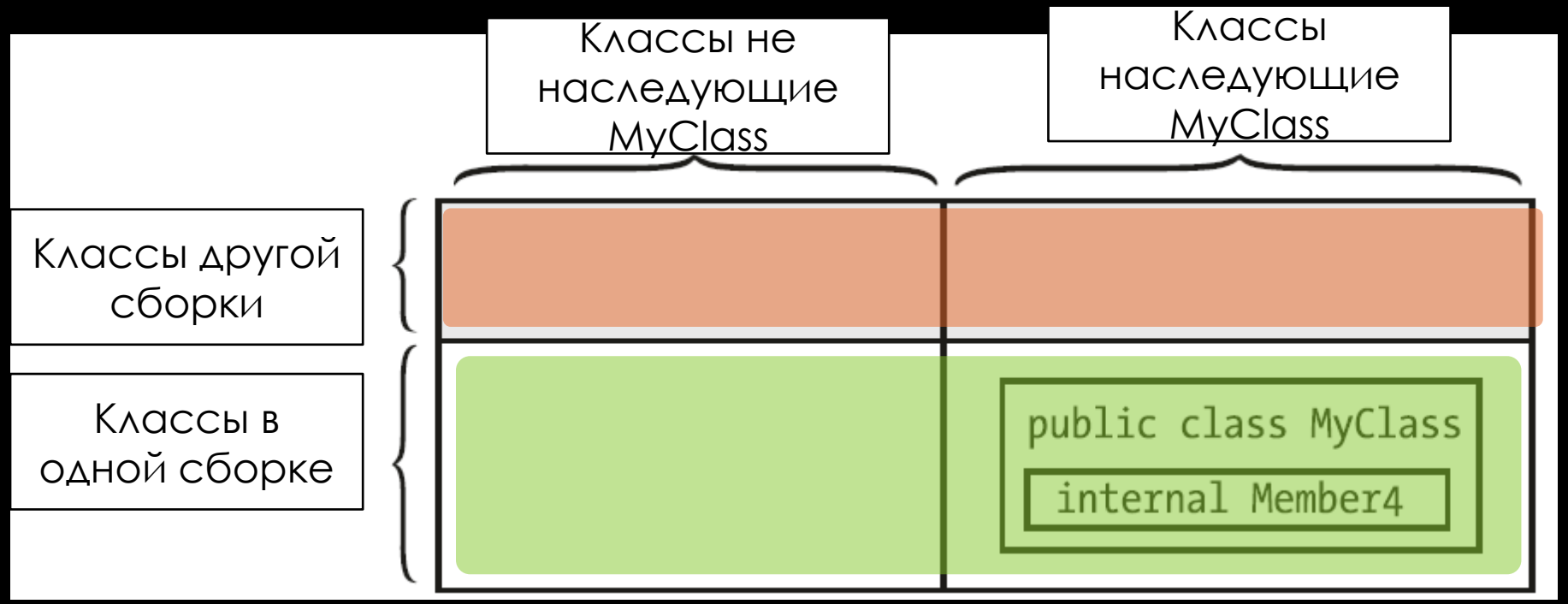
# ДОСТУП: PUBLIC-КЛАСС И PRIVATE-ЧЛЕН КЛАССА



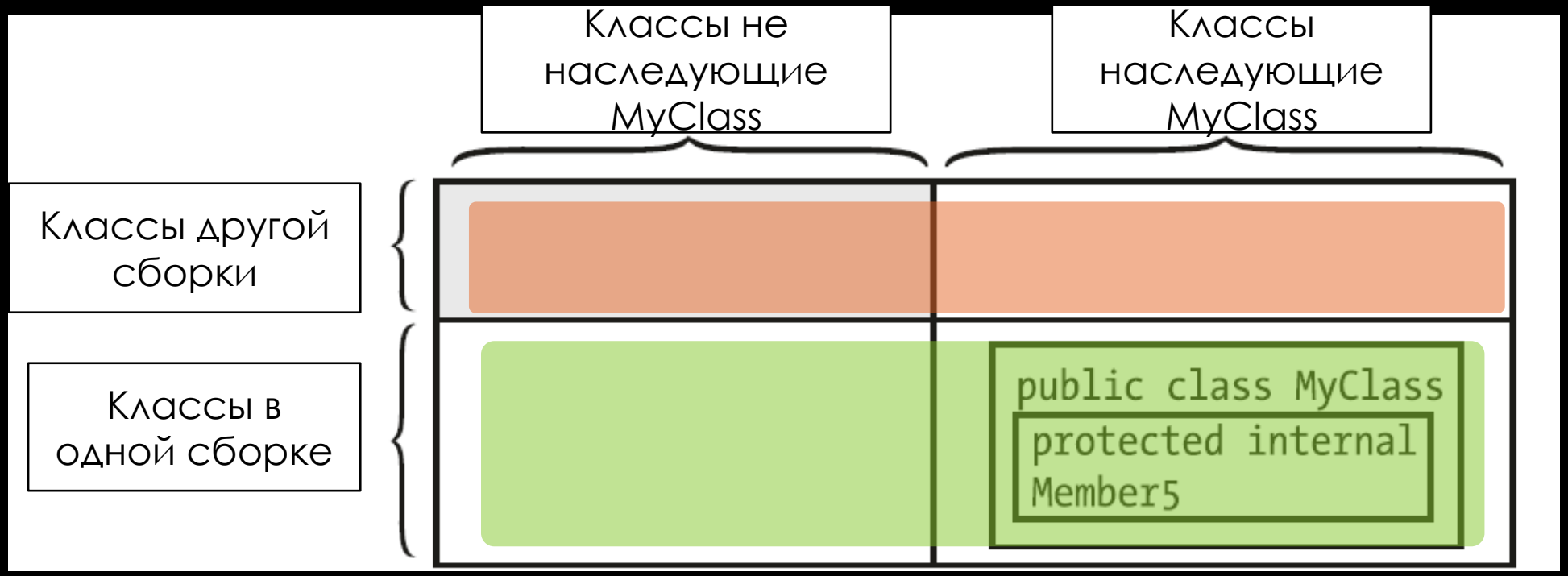
# ДОСТУП: PUBLIC-КЛАСС И PROTECTED-ЧЛЕН КЛАССА



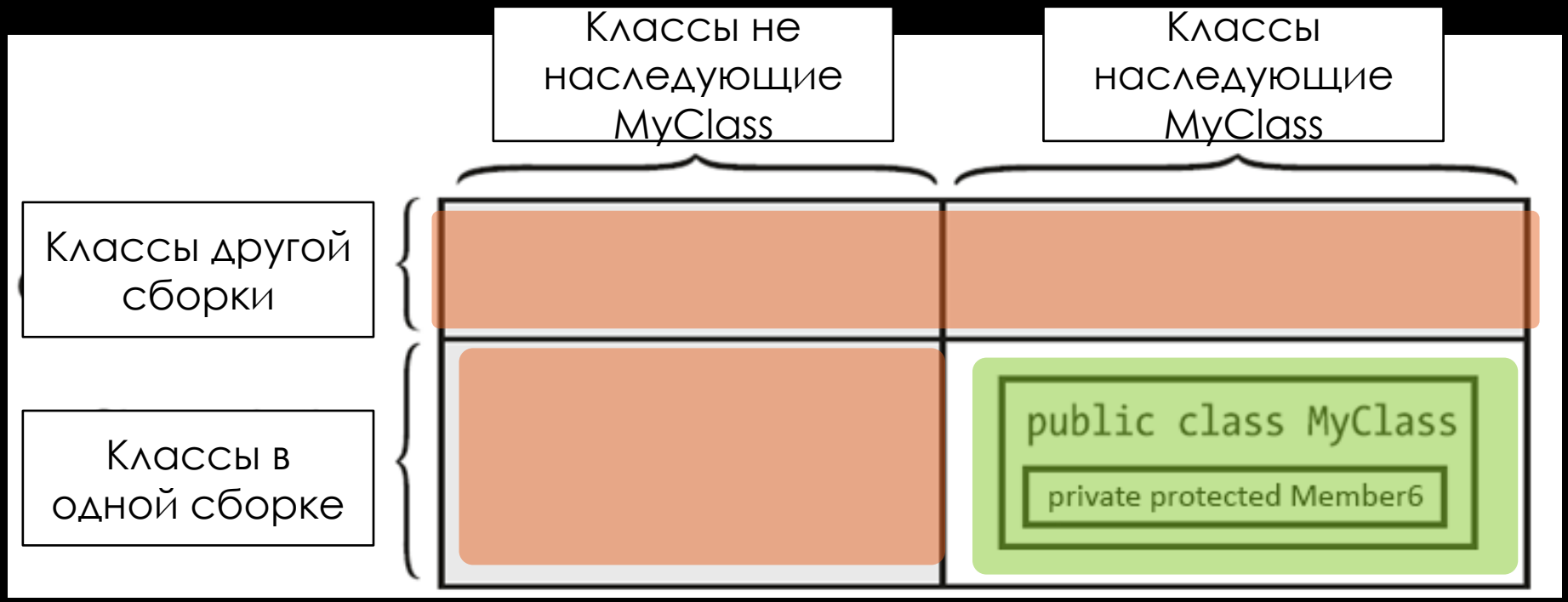
# ДОСТУП: PUBLIC-КЛАСС И INTERNAL-ЧЛЕН КЛАССА



# ДОСТУП: PUBLIC-КЛАСС И PROTECTED INTERNAL-ЧЛЕН КЛАССА



# ДОСТУП: PUBLIC-КЛАСС И PRIVATE PROTECTED-ЧЛЕН КЛАССА



# МОДИФИКАТОРЫ ДОСТУПА К ЧЛЕНАМ ТИПА

| | **Классы в одной сборке** | | **Классы в разных сборках** | |
|---|---|---|---|---|
| | Не наследник | Наследник | Не наследник | Наследник |
| private | | | | |
| internal | | | | |
| protected | | | | |
| protected internal | | | | |
| private protected | | | | |
| public | | | | |

# НЕМНОГО ОБ ИЕРАРХИЯ

## Пример коллизии при «очевидной иерархии»

# КОЛЛИЗИИ ИЕРАРХИЙ

- Моделирование иерархий может приводить к неверному описанию поведений объектов
- Иерархия объектов программной системы не всегда полностью отвечает иерархии объектов реального мира


Математику важно, что квадрат – частный случай прямоугольника
Программисту важно, что у квадрата всегда нужно изменять размер обеих сторон, что произойдёт при обращении по полиморфной ссылке?

# КОЛЛИЗИЯ ИЕРАРХИЙ: ПРИМЕР

```
public class Rectangle
{
    protected double _height;
    protected double _width;
    public double Height { set { _height = value; } }
    public double Width { set { _width = value; } }

    public Rectangle() { }
    public Rectangle (double h, double w)
    {
        Height = h;
        Width = w;
    }
    public double SquareCalc { get
        {
            return _height * _width;
        }
    }
}
```

```
Square s = new Square(10);
Console.WriteLine(s.SquareCalc);
Rectangle r = s;
r.Width = 5;
Console.WriteLine(s.SquareCalc);
```

**Вывод:**
100
50

```
public class Square : Rectangle
{
    public double Height {
       set { _height = value; _width = value; }
}
    public double Width {
       set { _width = value; _height = value; }
}
    public Square() { }
    public Square(double w) : base(w, w) { }
}
```

# ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА


- https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/new-modifier
- https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/tutorials/inheritance