

# Делегаты, лямбда-выражения

---

# Делегаты

---

# Делегаты

---

**Делегат** — это тип, который безопасно инкапсулирует метод, схожий с указателем функции в С и С++.

В отличие от указателей функций в С делегаты объектно-ориентированы, типобезопасны и безопасны. Тип делегата задается его именем.

# Делегаты

---

Когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Более того, метод можно вызывать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается

# Делегаты

---

Тип делегата определяет разновидность метода, которую могут вызывать экземпляры делегата. В частности, он определяет возвращаемый тип и типы параметров метода.

# Делегаты (полиморфизм)

---

Один и тот же делегат может быть использован для вызова разных методов **во время выполнения программы**, для чего достаточно изменить метод, на который ссылается делегат.

# Делегаты (полиморфизм)

---

Таким образом, метод, вызываемый делегатом, определяется **во время выполнения**, а не в процессе компиляции. В этом, собственно, и заключается главное преимущество делегата.

# Делегаты

Тип делегата объявляется с помощью ключевого слова *delegate*.

*delegate* возвращаемый\_тип имя(список\_параметров)

Делегат может служить для вызова любого метода с соответствующей сигнатурой и возвращаемым ТИПОМ.



# Делегаты

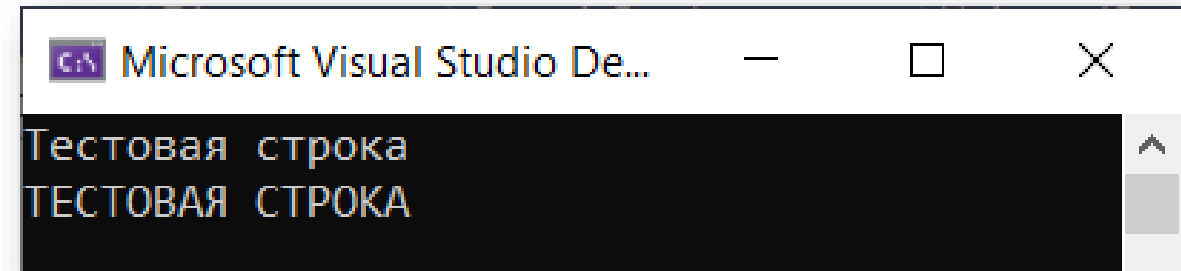
```
void MakeUpperCase(ref string s)
{
    s = s.ToUpper();
}
```

```
void PrintString(ref string s)
{
    Console.WriteLine(s);
}
```

```
delegate void StringServise(ref string s);
```

# Делегаты

```
var testText = "Тестовая строка";  
StringService sService;  
sService = PrintString;  
sService(ref testText);  
sService = MakeUpperCase;  
sService(ref testText);  
sService = PrintString;  
sService(ref testText);
```



# Делегаты

---

```
StringService sService;  
sService = PrintString;
```

или

```
var sService = new StringService(PrintString)
```

# Делегаты

---

Типы делегатов запечатаны (sealed) — они не могут быть производными — и невозможно получить пользовательские классы из делегата

# Делегаты

Когда делегат создается для вызова метода экземпляра класса, делегат ссылается как на экземпляр, так и на метод. Делегат ничего не знает о типе экземпляра, кроме метода, который он обертывает, поэтому делегат может ссылаться на любой тип объекта, если для этого объекта существует метод, соответствующий сигнатуре делегата.

# Делегаты

---

Когда делегат создается для вызова статического метода, он ссылается только на этот метод.

# Делегаты (callback)

**Callback** или **функция обратного вызова** — передача исполняемого кода в качестве одного из параметров другого кода.

Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове. Этот код может быть определён в других контекстах программного кода и быть недоступным для прямого вызова из этой функции.

# Делегаты (callback)

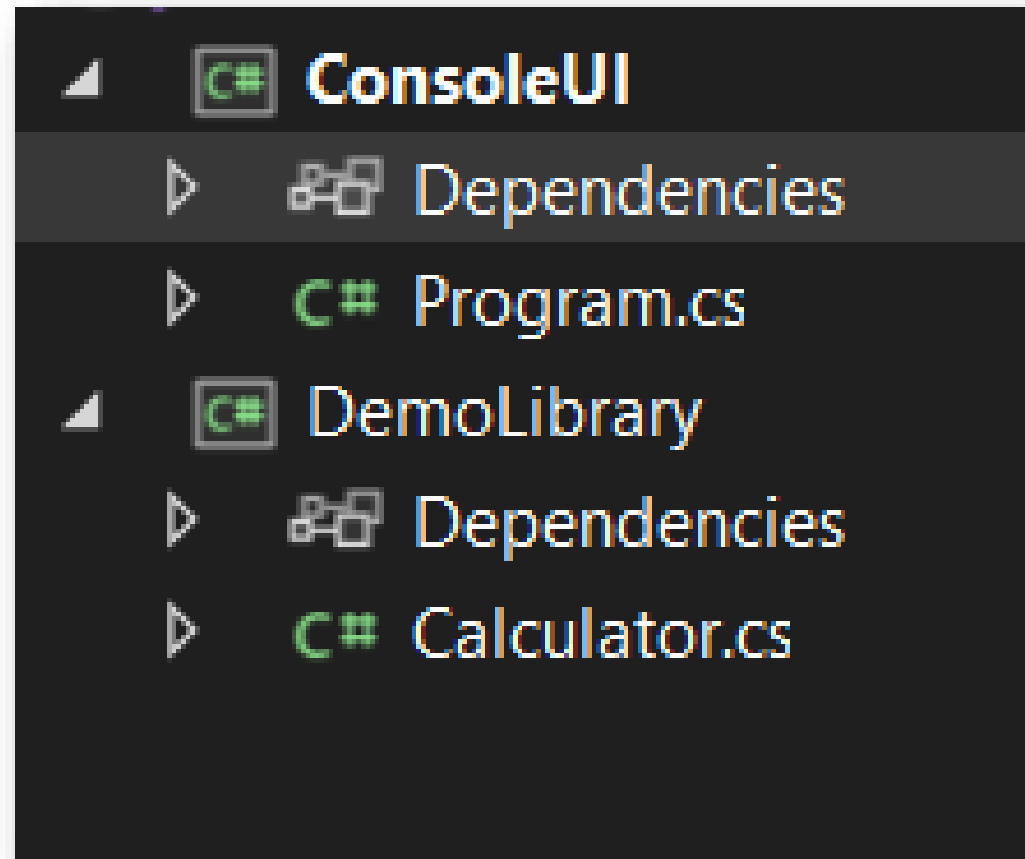
---

Поскольку созданный экземпляр делегата является объектом, его можно передавать как параметр или назначать свойству.

Это позволяет методу принимать делегат в качестве параметра и вызывать делегат в дальнейшем.



# Делегаты (callback)



# Делегаты (callback)

```
public class Calculator
{
    public IEnumerable<(double x, double result)>
        GetSin(double x0, double x1, double dx)
    {
        for (double x=x0; x<x1; x+=dx)
        {
            yield return(x, Math.Sin(x));
        }
    }
}
```

# Делегаты (callback)

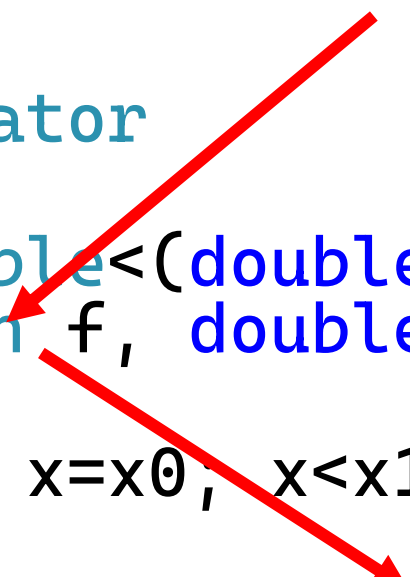
```
var calc = new Calculator();
```

```
foreach(var item in calc.GetSin(0, Math.PI/2, 0.01))  
    Console.WriteLine($"{item.x:F2}: {item.result:F2}");
```

# Делегаты (callback)

```
public delegate double MathFunction(double x);

public class Calculator
{
    public IEnumerable<(double x, double result)>
    GetGrahp(MathFunction f, double x0, double x1, double dx)
    {
        for (double x=x0; x<x1; x+=dx)
        {
            yield return(x, f(x));
        }
    }
}
```

A diagram consisting of two red arrows. The first arrow originates from the 'MathFunction' parameter in the 'GetGrahp' method signature and points to the 'f' parameter in the same signature. The second arrow originates from the 'f' parameter and points to the 'f(x)' expression within the 'yield return' statement in the loop.

# Делегаты (callback)

```
var calc = new Calculator();  
  
foreach(var item in calc.GetGraph(Math.Cos,  
                                   0, Math.PI/2, 0.01))  
    Console.WriteLine($"{item.x:F2}:{item.result:F2}");
```

# Делегаты

```
14 var testText = "Тестовая строка";  
15 StringService sService = null; ►  
16 sService(ref testText);
```

```
17  
18  
19 void MakeUpperCase(ref s  
20 {  
21     s = s.ToUpper();  
22 }  
23  
24 void PrintString(ref str  
25 {
```

## Exception Thrown

**System.NullReferenceException:** 'Object reference not set to an instance of an object.'

**sService** was null.

[View Details](#) | [Copy Details](#) | [Start Live Share session...](#)

# Делегаты

```
var testText = "Тестовая строка";  
StringService sService = null;  
  
if (sService!=null)  
    sService(ref testText);
```

# Делегаты

---

```
var testText = "Тестовая строка";  
StringService sService = null;  
  
sService?.Invoke(ref testText);
```



# Делегаты

---

## КОВАРИАНТНОСТЬ И КОНТРВАРИАНТНОСТЬ

# Делегаты (ковариантность и контрвариантность)

---

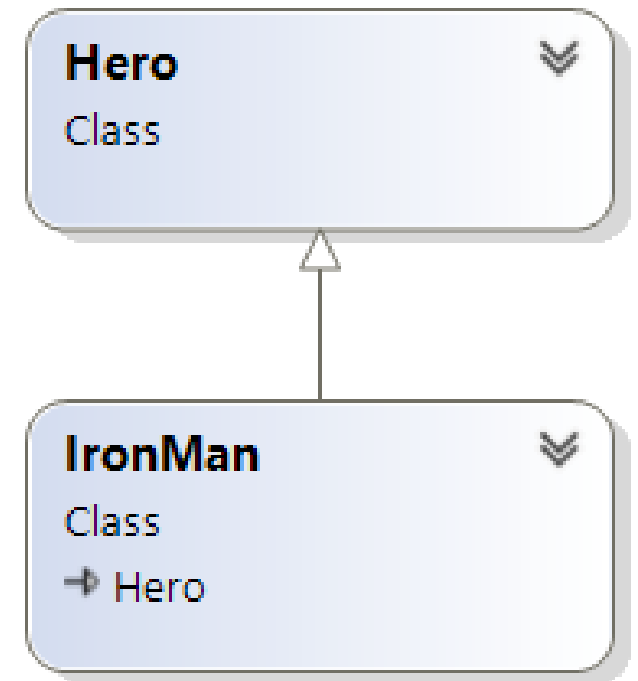
**Ковариантность** делегатов позволяет возвращать из метода объект, тип которого является производным от типа, возвращаемого делегатом.

# Делегаты (ковариантность и контрвариантность)

```
HeroAction unitAction = AddIronMan;
```

```
IronMan AddIronMan(string skill)  
{  
    . . .  
}
```

```
delegate Hero HeroAction(string name);
```



# Делегаты (ковариантность и контрвариантность)

---

**Контрвариантность** предполагает возможность передавать в метод объект, тип которого является более универсальным по отношению к типу параметра делегата.

# Делегаты (ковариантность и контрвариантность)

```
UpgradeIronMan unitAction = UpgradeHero;
```

```
unitAction(new IronMan());
```

```
void UpgradeHero(Hero hero)  
{  
  
}
```

```
delegate void UpgradeIronMan(IronMan unit);
```

# Делегаты

---

ГРУППОВЫЕ ДЕЛЕГАТЫ  
(ГРУППОВАЯ АДРЕСАЦИЯ)

# Делегаты (групповые делегаты)

---

**Групповая адресация** — это возможность создать список, или цепочку вызовов, для методов, которые вызываются автоматически при обращении к делегату.

# Делегаты (групповые делегаты)

Для групповой адресации необходимо получить экземпляр делегата, а затем добавить методы в цепочку с помощью оператора `+` или `+=`.

Для удаления метода из цепочки служит оператор `-` или `-=`.

Делегат, в котором используется групповая адресация, обычно имеет возвращаемый тип **`void`**.



# Делегаты (групповые делегаты)

```
var test = "Тестовая строка";  
StringService servise;  
servise = PrintString + MakeUpperCase;  
servise += PrintString;  
servise(ref test);
```

# Делегаты

---

## ОБОБЩЕННЫЕ ДЕЛЕГАТЫ

# Делегаты (обобщенные делегаты)

---

```
delegate T DoSometh<T, V>(V item);
```

# Делегаты (обобщенные делегаты)

31  
32  
33  
34  
35  
36  
37

```
Book[] books = new Book[10];
```

```
Array.Sort(books,|
```

▲ 3 of 17 ▼ **void Array.Sort(Array array, System.Collections.IComparer? comparer)**

Sorts the elements in a one-dimensional [Array](#) using the specified [System.Collections.IComparer](#).

**comparer:** The implementation to use when comparing elements. -or- null to use the [IComparable](#) implementation of each element.

# Делегаты (обобщенные делегаты)

```
public class GenericComparer<T> : IComparer<T> where T : class
{
    public delegate int ComparerFunc<T>(T x1, T x2);
    private ComparerFunc<T> comparerFunction;

    public GenericComparer(ComparerFunc<T> comparer)
    {
        comparerFunction = comparer;
    }

    public int Compare(T x, T y)
    {
        return comparerFunction(x,y);
    }
}
```

# Делегаты (обобщенные делегаты)

```
int BooksComparerByPages(Book x, Book y)
{
    return x.Pages.CompareTo(y.Pages);
}
```

```
Array.Sort(books,
    new GenericComparer<Book>(BooksComparerByPages));
```

# Делегаты

---

## СТАНДАРТНЫЕ ДЕЛЕГАТЫ

# Делегаты (стандартные делегаты)

---

В пространстве имен `System` определены стандартные обобщенные делегаты

## **`Func<T>` и `Action<T>`**

Их используют вместо определения нового типа делегата с каждым типом параметра и возврата



# Делегаты (стандартные делегаты)

---

Обобщенный делегат **Action<T>** предназначен для ссылки на метод, возвращающий **void**.

Этот класс делегата существует в различных вариантах, так что ему можно передавать до **16** параметров.

# Делегаты (стандартные делегаты)

---

Обобщенный делегат **Func<T>** позволяет вызывать методы с типом возврата.

Подобно **Action<T>**, **Func<T>** определен в разных вариантах для передачи до **16** типов параметров и типа возврата.

# Делегаты (стандартные делегаты)

```
public class Calculator
{
    public IEnumerable<(double x, double result)>
        GetGraph(Func<double, double> f,
                double x0, double x1, double dx)
    {
        for (double x=x0; x<x1; x+=dx)
        {
            yield return(x, f(x));
        }
    }
}
```

# Делегаты (стандартные делегаты)

```
public class GenericComparer<T> : IComparer<T> where T : class
{
    //public delegate int ComparerFunc<T>(T x1, T x2);

    private Func<T, T, int> comparerFunction;

    public GenericComparer(Func<T, T, int> comparer)
    {
        comparerFunction = comparer;
    }

    public int Compare(T x, T y)
    {
        return comparerFunction(x,y);
    }
}
```

# Анонимные методы

---

# Анонимные методы

---

Метод, на который ссылается делегат, нередко больше нигде не используется.

Иными словами, единственным основанием для существования метода служит то обстоятельство, что он может быть вызван посредством делегата, но сам он не вызывается вообще.

# Анонимные методы

---

В подобных случаях можно воспользоваться анонимной функцией, чтобы не создавать отдельный метод.

Анонимная функция, по существу, представляет собой безымянный кодовый блок, передаваемый конструктору делегата.

# Анонимные методы

---

Определение анонимных методов начинается с ключевого слова **delegate**, после которого идет в скобках список параметров и тело метода в фигурных скобках



# Анонимные методы

```
var test = "Тестовая строка";  
StringService servise = PrintString;  
servise += delegate (ref string s)  
{  
    s = s.ToUpper();  
};
```

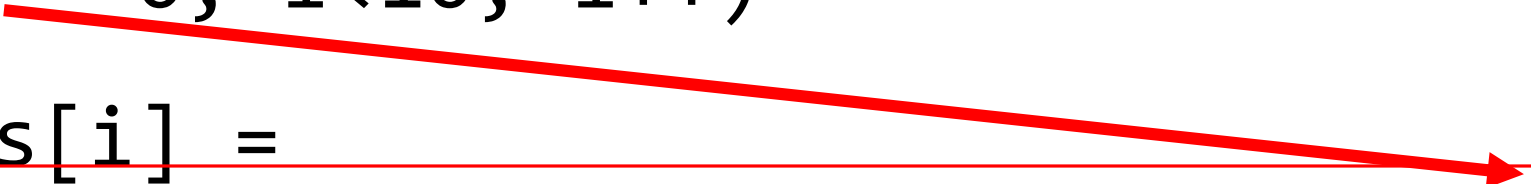
```
servise += PrintString;  
servise(ref test);
```

# Анонимные методы

Анонимные методы могут ссылаться *на внешние переменные*. Это переменные в области метода, в котором определен анонимный метод, или области типа, который содержит анонимный метод. Переменные, полученные таким способом, сохраняются для использования в анонимном методе, даже если бы в ином случае они оказались за границами области видимости и уничтожились сборщиком мусора.

# Анонимные методы

```
Action[] actions = new Action[10];  
for (int i = 0; i < 10; i++)  
{  
    actions[i] =  
        delegate () { Console.WriteLine(i); };  
}  
  
foreach (var a in actions)  
{  
    a();  
}
```



```
10  
10  
10  
10  
10  
10  
10  
10  
10  
10
```

# Замыкания

Замыкание (closure) представляет объект функции, который запоминает свое лексическое окружение даже в том случае, когда она выполняется вне своей области видимости. (<https://metanit.com/sharp/tutorial/3.54.php> )

Замыкания соединяют функцию с ее окружением, позволяя функции получить доступ к нелокальным переменным. В C# замыкания поддерживаются с помощью анонимных методов, лямбда-выражений и делегатов.

(<https://www.infoworld.com/article/3620248/how-to-use-closures-in-csharp.html> )

# Анонимные методы

```
for (int i = 0; i < 10; i++)  
{  
    var index = i;  
    actions[i] =  
        delegate () { Console.WriteLine(index); };  
};
```



0  
1  
2  
3  
4  
5  
6  
7  
8  
9

# Лямбда - выражения

---

# Лямбда - выражения

---

***Лямбда-выражение*** - упрощенный синтаксис для создания анонимной функции.

# Лямбда - выражения

---

## **Синтаксис лямбда - выражения**

(входные параметры) => выражение

**или**

(входные параметры) => { кодовый блок }



# Лямбда - выражения

```
var test = "Тестовая строка";  
StringService servise = PrintString;  
  
servise += (ref string s) => s = s.ToUpper();  
  
servise += PrintString;  
servise(ref test);
```

```
Func<int> f1 = () => 10;  
Func<int, int> f2 = x => x * 2;
```

```
// C# ver. 10 и выше  
var f3 = () => 10;  
var f4 = (int x) => x * 2;
```

```
Func<int, IComparable> f5 = (int x) => x>10  
    ? 10  
    : "x less than 10";
```

```
var f6 = IComparable (int x) => x > 10  
    ? 10  
    : "x less than 10";
```

# Лямбда - выражения

```
Book[] books = new Book[10];
```

```
Array.Sort(books, new GenericComparer<Book>(
    (b1, b2) => b1.Pages.CompareTo(b2.Pages)
));
```

или

```
Array.Sort(books, (b1, b2) => b1.Pages.CompareTo(b2.Pages));
```

# Лямбда - выражения

```
foreach(var item in calc.GetGraph(x=>x*x,  
                                   0, Math.PI/2, 0.01))  
    Console.WriteLine($"{item.x:F2}: {item.result:F2}");
```

# Лямбда - выражения

## Кортежи в лямбда-выражениях

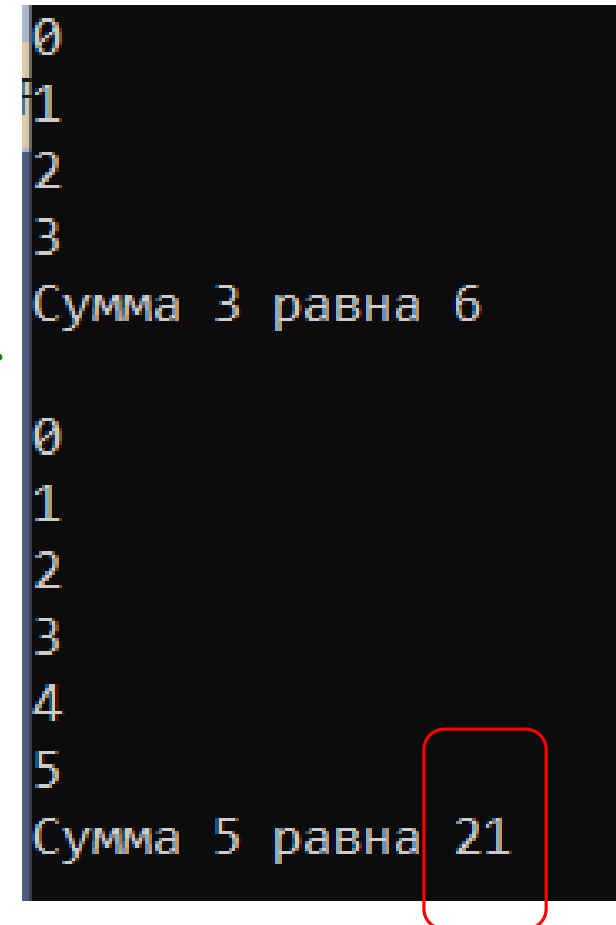
```
Func<(int, int, int), (double, double, double)> mathMeth =  
    (nums) => (Math.Sin(nums.Item1), Math.Sin(nums.Item2), Math.Sin(nums.Item3));
```

```
Console.WriteLine(mathMeth((30, 40, 60)));
```

# Лямбда – выражения (замыкание, closure)

```
Func<int,int> Counter()
{
    int sum = 0;
    Func<int, int> ctObj = (end)=>
    {
        for (int i = 0; i <= end; i++)
        {
            Console.WriteLine(i);
            sum += i; // Здесь подсчитанная сумма сохраняется в переменной sum.
        }
        return sum;
    };
    return ctObj;
}
```

```
Func<int, int> count = Counter();
Console.WriteLine($"Сумма 3 равна {count(3)}");
Console.WriteLine();
Console.WriteLine($"Сумма 5 равна {count(5)}");
```



```
0
1
2
3
Сумма 3 равна 6
0
1
2
3
4
5
Сумма 5 равна 21
```

# Мемоизация

Мемоизация (memoization) — это метод повышения производительности за счет кэширования возвращаемых значений дорогостоящих вызовов функций.

(<https://trenki2.github.io/blog/2018/12/31/memoization-in-csharp/> )



# Мемоизация

```
public static Func<T, V> Memoize<T, V>(this Func<T, V> f)
{
    var map = new Dictionary<T, V>();
    return a =>
    {
        V value;
        if (map.TryGetValue(a, out value))
            return value;
        value = f(a);
        map.Add(a, value);
        return value;
    };
}
```

# Мемоизация

```
Func<int, int> fib = null;
```

```
fib = n => n > 1  
      ? fib(n - 1) + fib(n - 2)  
      : n;
```

```
fib = fib.Memoize();
```