

# Делегаты. Анонимные функции



Delegates

# **Делегаты**

**Понятие делегата.**

**Синтаксис объявления делегата.**

**Цели и задачи делегатов.**

**Базовые классы для делегатов:**

**System.Delegate,**

**System.MulticastDelegate.**

**Многоадресный вызов.**

**ДЕЛЕГАТ** - это объект, который может ссылаться на метод.

### Цели и задачи делегатов

- ❖ делегаты обеспечивают поддержку функционирования событий;
- ❖ делегаты позволяют во время выполнения программы выполнить метод, который точно не известен в период компиляции;
- ❖ методы обратного вызова
- ❖ групповые (multicast) операции

## Общая форма объявления делегата :

**delegate** тип\_возврата **имя** (список\_параметров);

**тип\_возврата** - представляет собой тип значений, возвращаемых методами, которые этот делегат будет вызывать.

**имя** - указывается имя делегата .

**список\_параметров** - параметры, принимаемые методами, которые вызываются посредством делегата

**ВАЖНО!** Делегат может вызывать только **такие методы, у которых тип возвращаемого значения и список параметров (т.е. его сигнатура) совпадают с соответствующими элементами объявления делегата.**

```
class Circle
{
    // метод для вывода радиуса окружности
    public void PrintRadius(Double Radius)
    {    Console.WriteLine("Радиус равен: {0,7:N3}", Radius);    }

    // метод для вывода диаметра окружности
    public void PrintDiameter(Double Radius)
    {    Double diameter = 2 * Radius;
        Console.WriteLine("Диаметр равен: {0,7:N3}", diameter);    }

    // метод для вывода длины окружности
    public void PrintLenght(Double Radius)
    {    Double lenght = 2 * Math.PI * Radius;
        Console.WriteLine("Длина окружности равна: {0,7:N3}", lenght);    }

    // метод для вывода площади окружности
    public void PrintSquare(Double Radius)
    {    Double square = Math.PI * Radius * Radius;
        Console.WriteLine("Площадь круга равна: {0,7:N3}", square);    }
}
```

# Пример использования делегата (методы экземпляра)

```
// объявление делегата
private delegate void PrintInfo(double R);

static void Main(string[] args)
{
    string result;
    Console.WriteLine("Простейший делегат!");

    Circle circle = new Circle();

    // 1-ый способ создание объекта делегата на основе PrintInfo
    PrintInfo printInfo1 = new PrintInfo(circle.PrintRadius); //
    Console.WriteLine("Делегат вызван методом" + " PrintRadius()");

    printInfo1(4); // вызов метода через делегат с передачей 4

    // 2-ой способ создание объекта делегата на основе PrintInfo
    PrintInfo printInfo2 = circle.PrintDiameter;
    Console.WriteLine("Делегат вызван методом" + " PrintDiameter()");

    printInfo2(10); // вызов метода через делегат с передачей 10
}
```

Простейший делегат!

Делегат инициализирован методом PrintRadius(<)<br>Радиус равен : 4,000

Делегат инициализирован методом PrintDiameter(<)<br>Диаметр равен : 20,000

```
static class Circle
{
    // метод для вывода радиуса окружности
    public static void PrintRadius(Double Radius)
    {    Console.WriteLine("Радиус равен: {0,7:N3}", Radius);    }

    // метод для вывода диаметра окружности
    public static void PrintDiameter(Double Radius)
    {    Double diameter = 2 * Radius;
        Console.WriteLine("Диаметр равен: {0,7:N3}", diameter);    }

    // метод для вывода длины окружности
    public static void PrintLenght(Double Radius)
    {    Double lenght = 2 * Math.PI * Radius;
        Console.WriteLine("Длина окружности равна: {0,7:N3}", lenght);    }

    // метод для вывода площади окружности
    public static void PrintSquare(Double Radius)
    {    Double square = Math.PI * Radius * Radius;
        Console.WriteLine("Площадь круга равна: {0,7:N3}", square);    }
}
```

# Пример использования делегата (статические методы)

```
// объявление делегата
private delegate void PrintInfo(double R);

static void Main(string[] args)
{
    string result;
    Console.WriteLine("Простейший делегат!");

    // 1-ый способ создание объекта делегата на основе PrintInfo
    PrintInfo printInfo1 = new PrintInfo(Circle.PrintRadius); //
    Console.WriteLine("Делегат вызван методом" + " PrintRadius()");

    printInfo1(4); // вызов метода через делегат с передачей 4

    // 2-ой способ создание объекта делегата на основе PrintInfo
    PrintInfo printInfo2 = Circle.PrintDiameter; // делегата
    Console.WriteLine("Делегат вызван методом" + " PrintDiameter()");

    printInfo2(10); // вызов метода через делегат с передачей 10
}
```

```
Простейший делегат!
Делегат инициализирован методом PrintRadius()
Радиус равен : 4,000
Делегат инициализирован методом PrintDiameter()
Диаметр равен : 20,000
```



# Внутреннее устройство делегата

```
// Этот делегат может указывать на любой метод, который
// принимает два целых и возвращает целое значение
public delegate int BinaryOp(int x, int y);

// Когда компилятор C# обрабатывает тип делегата, он
// автоматически генерирует запечатанный (sealed) класс,
// унаследованный от System.MulticastDelegate

sealed class BinaryOp : System.MulticastDelegate
{
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
                                     AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

## Базовые классы System.MulticastDelegate и System.Delegate

При построении типа **delegate** неявно объявляется тип класса, унаследованного от **System.MulticastDelegate** и **System.Delegate**. От этих классов нельзя наследоваться.

```
public abstract class MulticastDelegate : Delegate
{
    // Возвращает список методов, на которые "указывает" делегат.
    public sealed override Delegate[] GetInvocationList();
    // Перегруженные операции.
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1, MulticastDelegate d2);
    // Используются для управления списком методов, поддерживаемых делегатом.
    private IntPtr _invocationCount;    private object _invocationList;
}
```

```
public abstract class Delegate : ICloneable, ISerializable
{ // Методы для взаимодействия со списком функций.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);
    // Свойства, показывающие цель делегата.
    public MethodInfo Method { get; }
    public object Target { get; }
}
```

**Многоадресатная передача (multicasting)** - это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата.

Для реализации многоадресатной передачи необходимо:

1. **Создать экземпляр делегата.**
2. Добавить методы в цепочку - **используя оператор "+="** .
3. Удалить метода из цепочки - **используя оператор "-="** .

**Делегат с многоадресатной передачей имеет одно ограничение:**

**он должен возвращать тип void т.к. если делегат возвращает значение, то им становится значение, возвращаемое последним методом в списке ВЫЗОВОВ.**

```
class Circle
{
    double radius;
    public Circle(Double Radius)
    {
        radius = Radius;
    }
    // метод для вывода радиуса окружности
    public void PrintRadius()
    {
        Console.WriteLine("Радиус равен: {0,7:N3}", radius);
    }
    // метод для вывода диаметра окружности
    public void PrintDiameter()
    {
        Double diameter = 2 * radius;
        Console.WriteLine("Диаметр равен: {0,7:N3}", diameter);
    }
    // метод для вывода длины окружности
    public void PrintLenght()
    {
        Double lenght = 2 * Math.PI * radius;
        Console.WriteLine("Длина окружности равна: {0,7:N3}", lenght);
    }
    // метод для вывода площади окружности
    public void PrintSquare()
    {
        Double square = Math.PI * radius * radius;
        Console.WriteLine("Площадь круга равна: {0,7:N3}", square);
    }
}
```

# Пример многоадресатной передачи

```
private delegate void PrintInfo();
static void Main(string[] args)
{
    Circle circle = new Circle(4);
    PrintInfo printInfo = circle.PrintRadius;
    printInfo(); // Вызов делегата-вызов одного метода

    printInfo += circle.PrintDiameter; // Добавление метода PrintDiameter
    printInfo += circle.PrintLenght;   // Добавление метода PrintLenght
    printInfo += circle.PrintSquare;   // Добавление метода PrintSquare

    printInfo(); // Вызов делегата-вызов всех методов

    printInfo -= circle.PrintSquare; // Удаление метода PrintSquare

    printInfo(); // Вызов делегата - вызов трех методов

    printInfo = printInfo + circle.PrintSquare - circle.PrintRadius
                - new PrintInfo(circle.PrintDiameter);

    printInfo(); // Вызов делегата - PrintLenght() и PrintSquare()
}
```

## Два делегата равны тогда и только тогда, если:

- они одного и того же типа и при этом инкапсулируют один и тот же метод
- если инкапсулируемый метод является instance методом, то и объект должен быть один и тот же.

```
delegate int D(int x);
static int Metod(int x) { return 1; }
static void Main(string[] args)
{
    D d1 = Metod;
    D d2 = Metod;
    Console.WriteLine(d1==d2); // True
}
```

```
delegate int D(int x);
static int Metod1(int x) { return 1; }
static int Metod2(int x) { return 1; }
static void Main(string[] args)
{
    D d1 = Metod1;
    D d2 = Metod2;
    Console.WriteLine(d1==d2); // False
}
```

```
TestClass testClass = new TestClass();
D d5 = testClass.Metod;
D d6 = testClass.Metod;
Console.WriteLine("d5 == d6 - {0}", d5 == d6); // True

TestClass testClass1 = new TestClass();
TestClass testClass2 = new TestClass();
D d7 = testClass1.Metod;
D d8 = testClass2.Metod;
Console.WriteLine("d7 == d8 - {0}", d7 == d8); // False
```

## Особенности многоадресатного вызова

- сигнатура делегата должна возвращать void – в противном случае будет получен только результат вызова **последнего метода делегата**;
- при вызове группового делегата он **последовательно вызывает все методы по порядку**;
- если один из методов, вызванных делегатом, сгенерирует исключение, то **прекратится вся последовательность вызовов методов**;

# Назначение делегатов

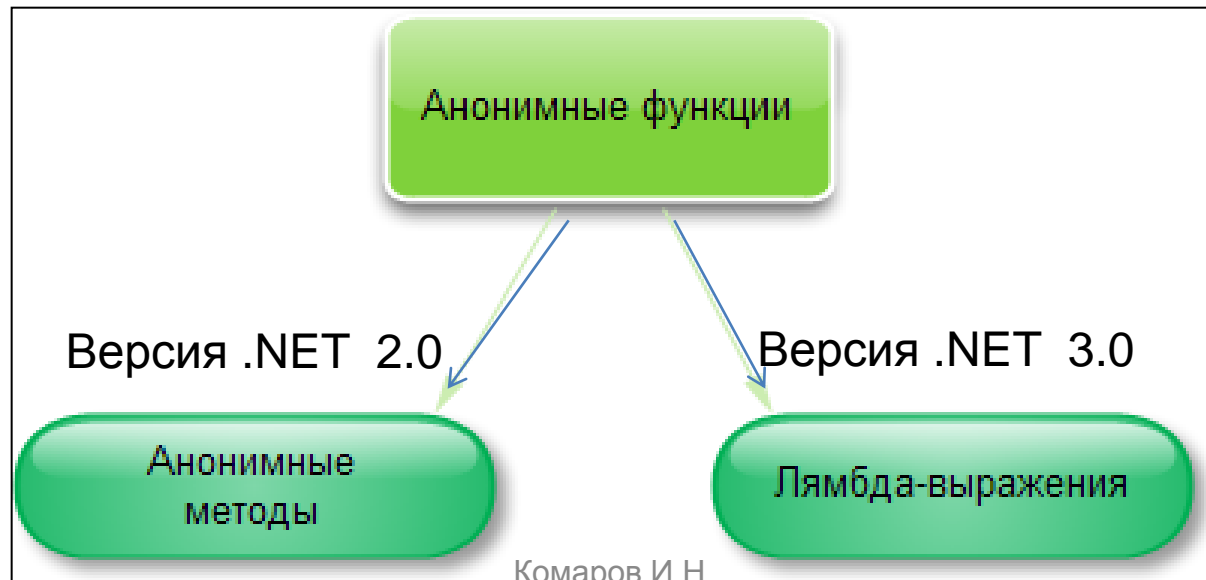
- ❖ делегаты обеспечивают поддержку функционирования событий.
- ❖ делегаты позволяют во время выполнения программы выполнить метод, который точно не известен в период компиляции.



# Анонимные функции.

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

**Преимущество анонимной функции:** состоит в ее простоте - отпадает необходимость объявлять отдельный метод, единственное назначение которого состоит в том, что он передался делегату.



**Анонимный метод** - один из способов создания безымянного блока кода, связанного с конкретным экземпляром делегата.

```
delegate void DelegateCountNums(); // объявление делегата
// . . .
// Анонимный метод для подсчета чисел
DelegateCountNums count = delegate
{
    for (int i = 0; i <= 5; i++)
        Console.WriteLine(i);
};
count(); // вызов анонимного метода через делегат
```

1. Анонимный метод не может **иметь** локальные переменные, **имена** которых совпадают с именами **локальных переменных** объемлющего метода.
2. Анонимный метод **может обращаться** к переменным **экземпляра** (или статическим переменным) из контекста объемлющего класса.

# Анонимный метод с аргументами.

```
delegate String DelegateStrings(String str);

String strMiddle = ", средняя часть,";

// создание анонимного метода и присваивание его объекту делегата
// parameter - входящий параметр!
DelegateStrings anonymousDelegate = delegate(String parameter) {
    parameter += strMiddle;
    parameter += " а эта часть строки добавлена в конец.\n";
    return parameter; // возврат значения
};

// вызов анонимного метода через делегат!
string result = anonymousDelegate("Начало строки");
```

Начало строки, средняя часть, а эта часть строки добавлена в конец.

## Применение внешних переменных в анонимных методах

Локальная переменная, в область действия которой входит анонимный метод, называется **внешней переменной**.

**ВАЖНО!** Локальная переменная, которая обычно прекращает свое существование после выхода из кодового блока, используется в анонимном методе, то она **продолжает существовать до тех пор, пока не будет уничтожен делегат**, ссылающийся на этот метод.

```
// Анонимный метод для подсчета чисел (передаются параметры)
delegate void DelegateCountNums2(int N1, int N2);

int K = 2; // внешняя переменная
DelegateCountNums2 countNums2 = delegate(int N1, int N2)
{
    int Sum = 0;
    for (int i = N1; i <= N2; i++)
    {
        Sum += i;
        Console.WriteLine(Sum);
    }
    Console.WriteLine(Sum * K); // использование внешней переменной
};

countNums2(5, 15);
```

**Лямбда-выражение** - это способ создания анонимной функции.

лямбда-выражения находят применение

- в работе с LINQ
- используются вместе с делегатами и событиями.

## Лямбда-оператор =>

общая форма одиночного лямбда-выражения:

**параметр => выражение**

**n => n % 2 == 0**

общая форма лямбда-выражения со списком параметров:

**(список\_параметров) => выражение**

В левой его части указывается **входной параметр** (или несколько параметров), а в правой части - **тело лямбда-выражения**.

## Одиночное лямбда-выражение,

```
delegate int DelegateSumma2(int N);  
// Простейшее лямбда выражение  
    DelegateSumma2 summa2 = cu => cu + 2;  
    Console.WriteLine(summa2(10));  
  
delegate bool DelegateIsRange(int lower, int upper, int v);  
// Лямбда-выражения для проверки попадания в интервал  
    DelegateIsRange isRange = (Min, Max, N) => N >= Min && N <= Max;  
    Console.WriteLine(isRange(10, 50, 25));
```

## Особенности

- компилятор делает заключение о типе параметра и типе результата вычисления лямбда-выражения по типу делегата.
- если в лямбда-выражении используется несколько параметров, их необходимо заключить в скобки.
- внешние переменные могут использоваться и захватываться в лямбда-выражениях таким же образом, как и в анонимных методах

## Блочные лямбда-выражения

```
delegate void DelegateCountNums2(int N1, int N2);  
// Лямбда-выражения (передаются параметры)  
int K = 2; // внешняя переменная  
DelegateCountNums2 countNums2 = (N1, N2) =>  
{  
    int Sum = 0;  
    for (int i = N1; i <= N2; i++)  
    {  
        Sum += i;  
        Console.WriteLine(Sum);  
    }  
    Console.WriteLine(Sum*K); // использование внешней переменной  
};  
countNums2(5,15);
```

```
delegate String DelegateStrings(String str);  
// Лямбда-выражения (передаются параметры + возвращаемое значение)  
String strMiddle = ", средняя часть,";  
DelegateStrings createStr = (parameter) =>  
{  
    parameter += strMiddle;  
    parameter += " а эта часть строки добавлена в конец.\n";  
    return parameter; // возврат значения  
};  
string rezult = createStr("Начало строки");  
Console.WriteLine(rezult);
```

В лямбда-выражениях получение доступа к переменным вне блока лямбда-выражения называется «**замыканием**».

**Внимание!** если переменная `someVal` позднее изменяется и затем вызывается лямбда-выражение, то будет использоваться новое значение `someVal`.

```
delegate int Invoker(int x);
static void Main(string[] args)
{
    int someVal = 7;
    Invoker f = x => x + someVal;
    Console.WriteLine(f(3));
}
```

Для лямбда-выражения  
`x => x + someVal`  
компилятор создает  
анонимный класс, который  
имеет конструктор,  
принимающий внешнюю  
переменную.

```
public class AnonymousClass
{
    private int someVal;
    public AnonymousClass(int someVal)
    {
        this.someVal = someVal;
    }
    public int AnonymousMethod(int x)
    {
        return x + someVal;
    }
}
```



```
delegate int Transformer(int x);
static void Main(string[] args)
{
    int[] arr = {1, -5, 4, -8};
    Console.WriteLine(String.Join(", ", arr));
    Transform(arr, Invert);
    Console.WriteLine(String.Join(", ", arr));
    Transform(arr, Cube);
    Console.WriteLine(String.Join(", ", arr));
    Console.Read();
}

static int Invert(int x)
{ return x * (-1); }

static int Cube(int x)
{ return x*x*x; }

static void Transform(int [] arr, Transformer t)
{
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = t.Invoke(arr[i]);
    }
}
```

```
int[] arr = { 1, -5, 4, -8 };
Console.WriteLine(String.Join(", ", arr));
// использование методов
TransformOperations.Transform(arr, Operations.Invert);
// использование анонимных методов
TransformOperations.Transform(arr, delegate(int x) { return -x; });
// использование лямбда выражений
TransformOperations.Transform(arr, x => -x);
Console.WriteLine(String.Join(", ", arr));

// использование методов
TransformOperations.Transform(arr, Operations.Cube);
// использование анонимных методов
TransformOperations.Transform(arr, delegate(int x) { return x * x * x; });
// использование лямбда выражений
TransformOperations.Transform(arr, x => x * x * x);
Console.WriteLine(String.Join(", ", arr));
Console.Read();
```