ЛЕКЦИЯ 1-2

- Модуль 3
- 10.01.2024
- Типы делегаты

ЦЕЛИ ЛЕКЦИИ

- Понять, как мы пришли к делегатам?
- Познакомиться с типом делегата в языке C#
- Научиться осуществлять вызовы с использованием экземпляров делегатов
- Изучить схему обратного вызова и варианты её реализации

КОНТРОЛЬ, МОДУЛЬ 3

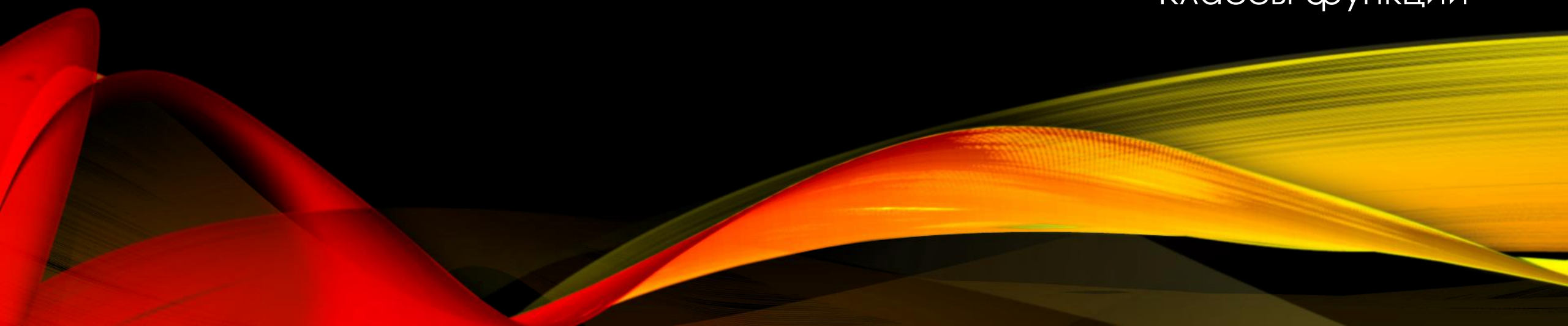
Ориентировочные даты контроля, в зависимости от скорости освоения учебного материала и сложности вариантов даты и продолжительность работы могут быть изменены

- **КДЗ-1** (11.01 – 25.01)
- **КР-1** (31.01-05.02, это диапазон, у каждой группы на своём семинаре)
- **КДЗ-2** (10.02 – 24.02)
- **КР-2** (28.02 – 04.03 , это диапазон, у каждой группы на своём семинаре**)**
- **КДЗ-3** (06.03 – 20.03)

НЕМНОГО ФУНКЦИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ

Побочный эффект

Классы функций



КАК МЫ ПРИШЛИ К ДЕЛЕГАТАМ?

ПОБОЧНЫЙ ЭФФЕКТ

- **Побочный эффект** [side effect] – изменение при выполнении подпрограммы глобальных переменных, не являющихся аргументами подпрограммы

```
public static void Main()
{
    int a = 15;
    void LocalFunction(int x)
    {
        a = x++;
    }
    LocalFunction(12);
    Console.WriteLine(a);
}
```

Здесь изменяется значение переменной из внешнего для LocalFuction() контекста

ФУНКЦИИ ПЕРВОГО КЛАССА

- **Объекты первого класса** [first class objects]
  - Фигурируют в выражениях
  - Могут быть назначены переменной
  - Могут быть использованы как аргумент функции
  - Могут возвращаться из функции

Объекты первого класса – это терминология функционального программирования и относится к языку программирования, т.е. **речь идёт о поддержке языком возможности передачи функций в качестве аргументов других функций**

ФУНКЦИЯ ВЫСШЕГО ПОРЯДКА

- **Функция высшего порядка** [higher-order function, HOF] – функция, принимающая в качестве аргументов или возвращающая в качестве результата другую функцию

Функция высшего порядка

```
public static void Sort<T> (T[] array, Comparison<T> comparison);
```

Способ реализации в языке возможности передачи функции, как параметра

ПОЯСНЕНИЯ ПО КОМПАРАТОРАМ

```
public static void Sort<T> (T[] array, Comparison<T> comparison);
```

Массив / универсальная коллекция

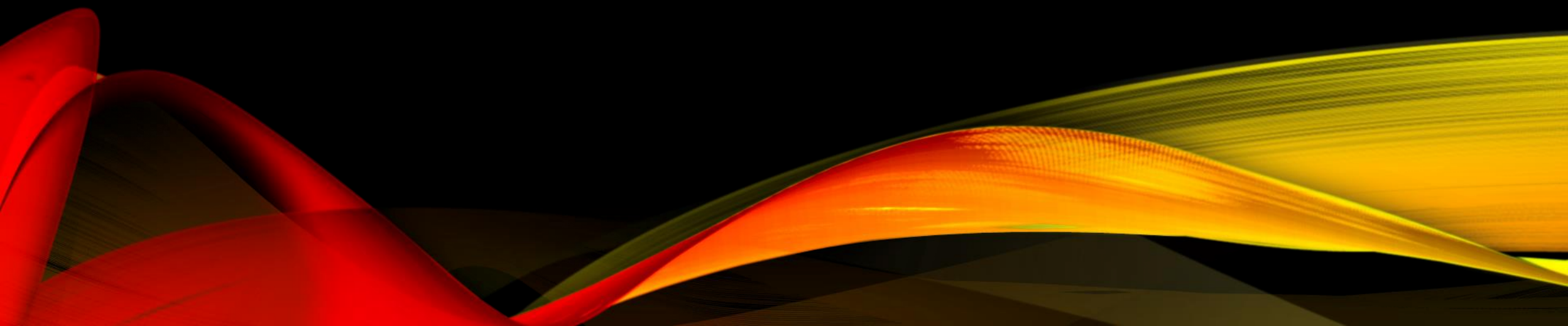
компаратор

Особенности типа T	**Компаратор**
Реализует `System.IComparable<T>`	`IComparable<T>.CompareTo(T)` из `System.IComparable<T>`
Реализует `System.IComparable`	`IComparable.CompareTo(Object)` из `System.IComparable`
Не реализует интерфейсов	Нет компаратора по умолчанию, и ее следует передать явно

К компаратору мы ещё в этой лекции вернёмся

ДЕЛЕГАТ-ТИП

Делегат-тип



ЧТО ТАКОЕ ДЕЛЕГАТЫ?

- **Делегат-тип** – специальный вид класса, экземпляр которого ссылается на методы с определённым типом возвращаемого значения и списком параметров

КАК УСТРОЕНЫ ДЕЛЕГАТЫ

- Делегат-тип является ссылочным типом данных
- Все делегаты пользовательских типов генерируются средой исполнения (CLR)

ОСОБЕННОСТИ ДЕЛЕГАТА-ТИПА

- Делегат-тип является наследником класса `Delegate`
- Пользовательский тип не может быть напрямую унаследован из класса Delegate или `MulticastDelegate`
- Класс `Delegate` не является делегатом-типом
- Делегат-тип опечатан (`sealed`)

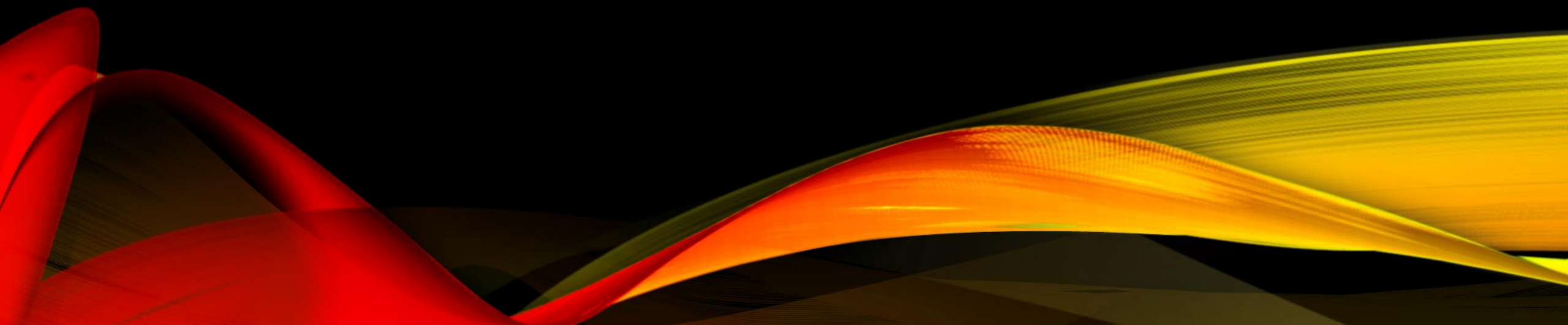
ОБЪЯВЛЕНИЕ ДЕЛЕГАТ-ТИПА

Синтаксиса описание делегата-типа:

<модификатор доступа> delegate <тип_возвр_значения> Идентификатор (<список_форм_парам>);

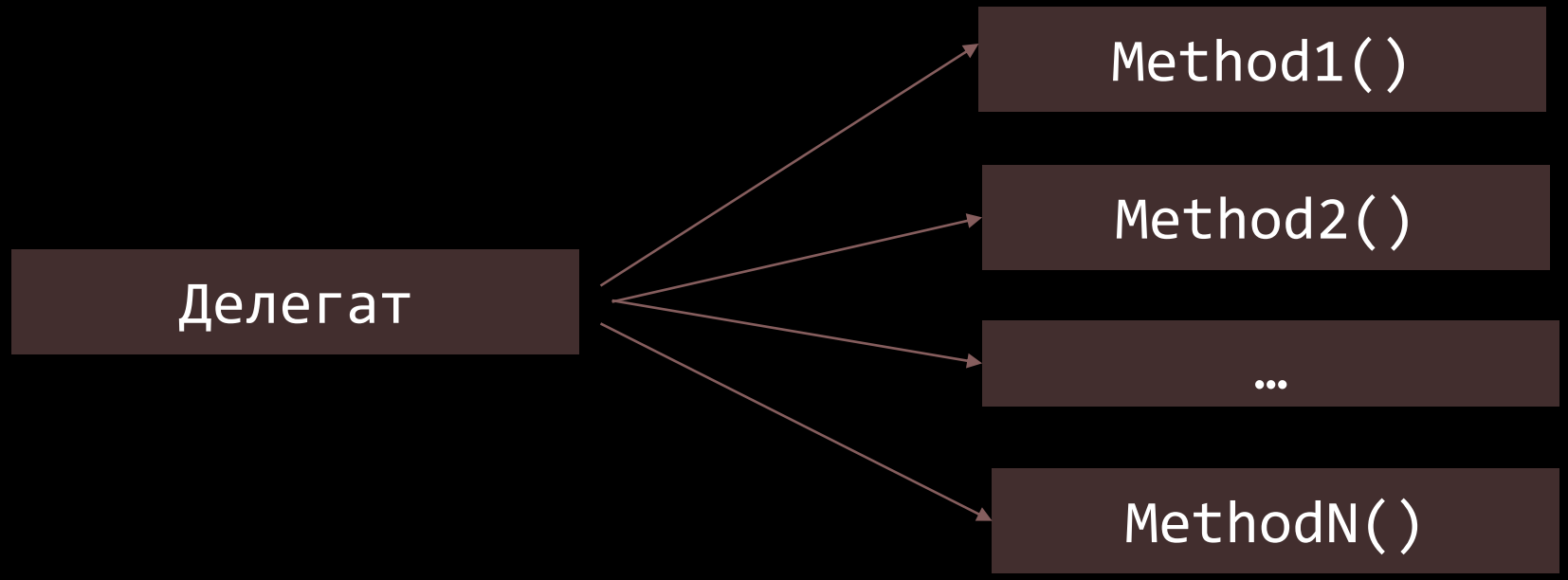
```
namespace NameOfNamespace
{
    /// <summary>
    /// Объявлен делегат-тип MyDel в пространстве имён.
    /// </summary>
    /// <param name="x">Целочисленный параметр</param>
    delegate void MyDel(int x);
    public class A
    {
        /// <summary>
        /// Объявлен делегат-тип MyDelInA в другом типе данных.
        /// </summary>
        /// <param name="x">Целочисленный параметр 1</param>
        /// <param name="y">Целочисленный параметр 2</param>
        /// <returns>Вещественное значение</returns>
        delegate double MyDelInA(int x, int y);
    }
}
```

ЭКЗЕМПЛЯРЫ ДЕЛЕГАТОВ

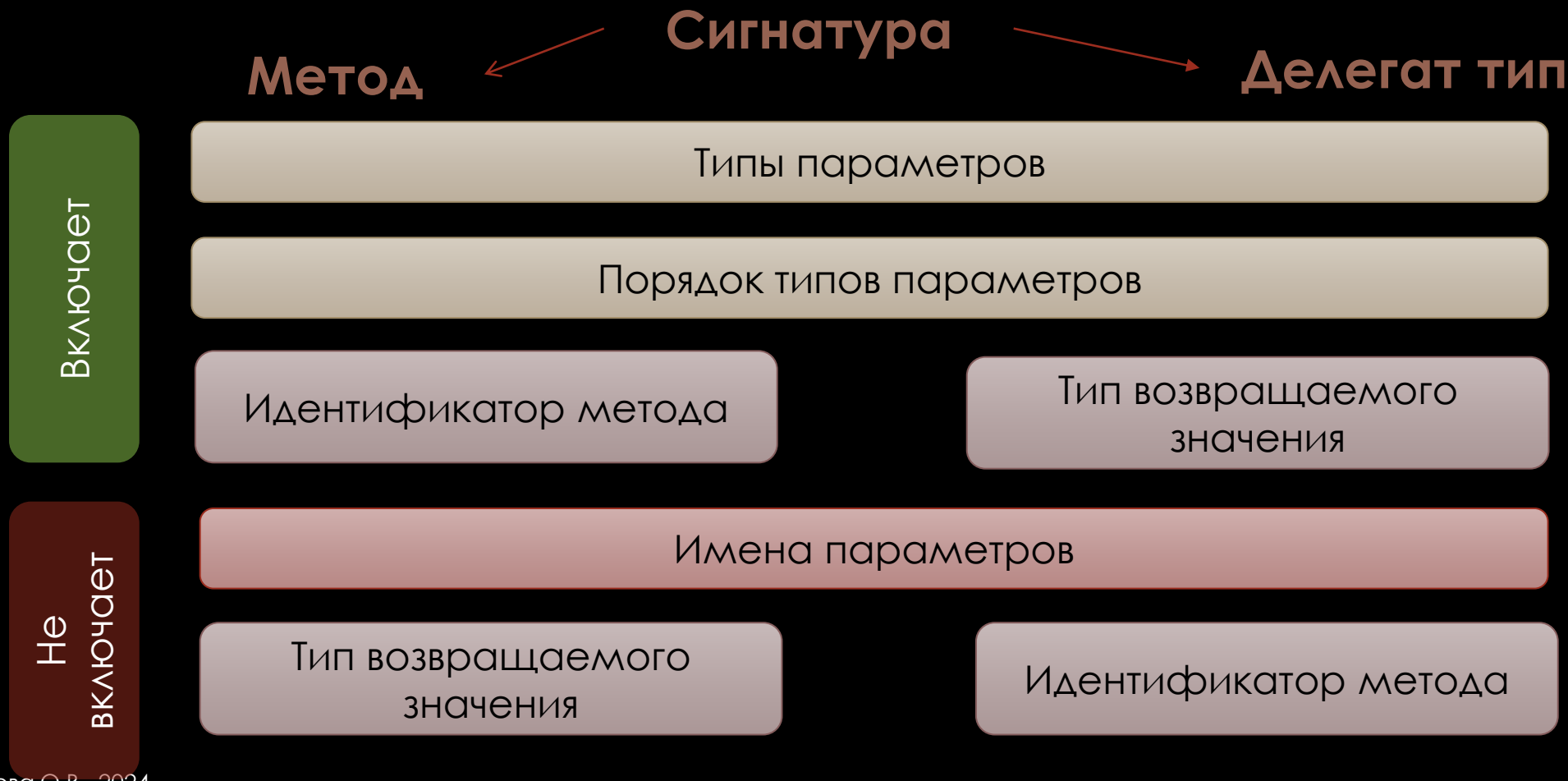


ЭКЗЕМПЛЯР ДЕЛЕГАТА

- **Делегат** или **экземпляр делегата-типа** хранит список ссылок на методы, соответствующие сигнатуре делегата-типа
- Экземпляр делегата является неизменяемым



СИГНАТУРА ДЕЛЕГАТА-ТИПА И СИГНАТУРА МЕТОДА



ОБЪЯВЛЕНИЕ ДЕЛЕГАТ-ТИПА И СОЗДАНИЕ ЭКЗЕМПЛЯРА

Синтаксис создания объектов:

- Стандартный с помощью new
- Укороченный – через присваивание

```
DemoObject myInstObj = new DemoObject();

MyDel delEx1 = new MyDel(myInstObj.Print);
    // Ссылка на метод объекта myInstObj.
MyDel delEx2 = DemoObject.StaticPrint;      // Укороченный синтаксис создания.
```

ВЫЗОВ ЧЕРЕЗ ДЕЛЕГАТ

```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

```
Console.WriteLine(func?.Invoke(4));
```

В этом примере делегат связан с экземплярным методом класса Foo

```
public class Program
{
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        Console.WriteLine(foo.F(5)); // Непосредственный вызов метода.
        Console.WriteLine(func(4)); // Вызов метода через делегат
    }
}
```

вызов делегата, связанного с **null** приведёт к исключению **NullReferenceException**

Это вызов по ссылке, а ссылки бывают null

ДЕЛЕГАТ-ТИП И ЭКЗЕМПЛЯР ДЕЛЕГАТА

Время компиляции

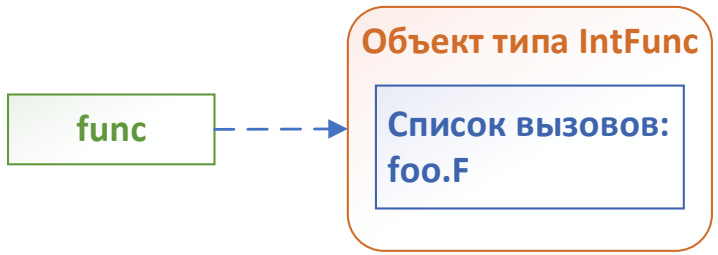
```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

Время исполнения

```
Foo foo = new Foo();
IntFunc func = foo.F;
```

Память



ИНФОРМАЦИЯ ОБ ЭКЗЕМПЛЯРАХ ДЕЛЕГАТ-ТИПОВ

Все делегаты-типы имеют следующие функциональные члены:

- `public Delegate[] GetInvocationList()` – метод, возвращающий массив объектов `Delegate`, содержащих информацию о каждом из вызываемых методов

`public MethodInfo Method { get; }` – информация о последнем в списке вызовов методе

- `public object? Target { get; }` – объект, связанный с последним методом в списке вызовов или null, если метод статический

Поскольку делегаты хранят в себе ссылку на объект, относительно которого вызывается метод, может происходить продление жизни объектов (иногда нежелательное)

ОПРЕДЕЛЕНИЕ ВИДА МЕТОДА

```
delegate int IntFunc(int x);
public class Foo
{
    public int F(int x) => 2 * x;
}
```

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        Console.WriteLine(func.Target);
        Console.WriteLine(sFunc.Target ?? "static");
        Console.WriteLine(func?.Invoke(4)); // Вызов метода через делегат
    }
}
```

Для статических методов свойство Target – null, поэтому для вывода заменим

ЭКЗЕМПЛЯРЫ ДЕЛЕГАТОВ

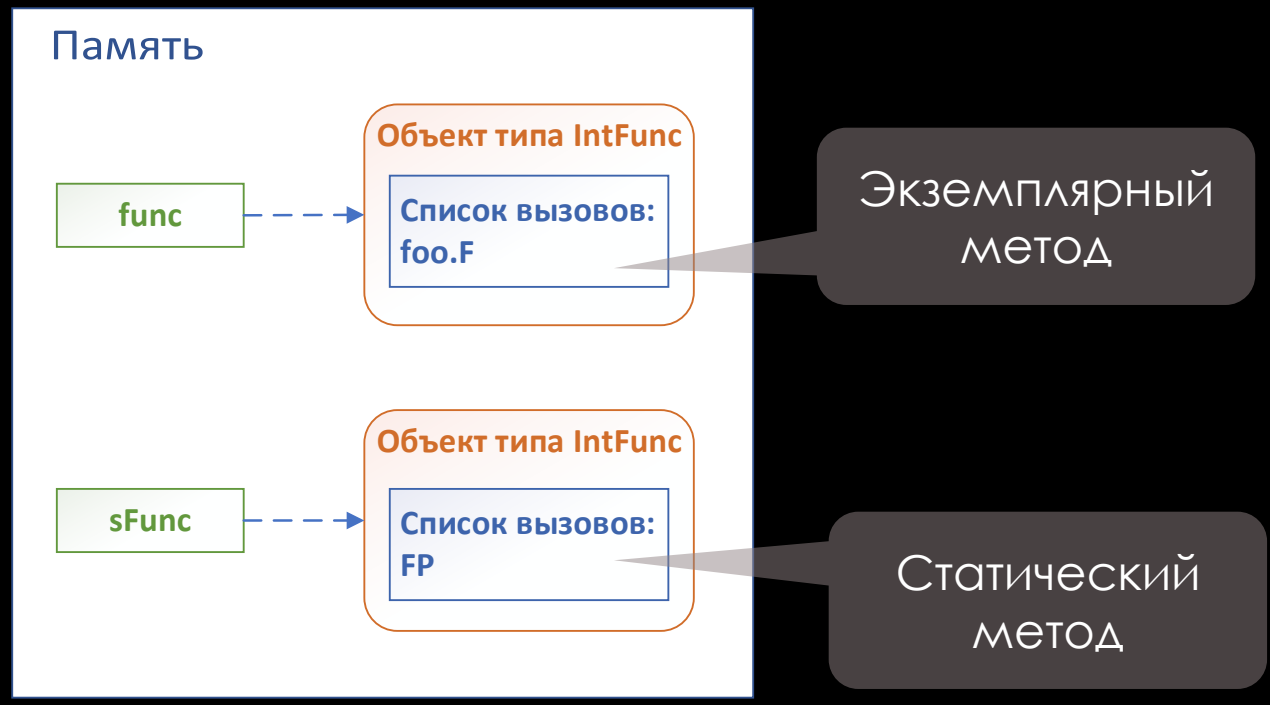
Время компиляции

```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

Время исполнения

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
    }
}
```



СОЗДАНИЕ ЭКЗЕМПЛЯРОВ ДЕЛЕГАТ-ТИПОВ

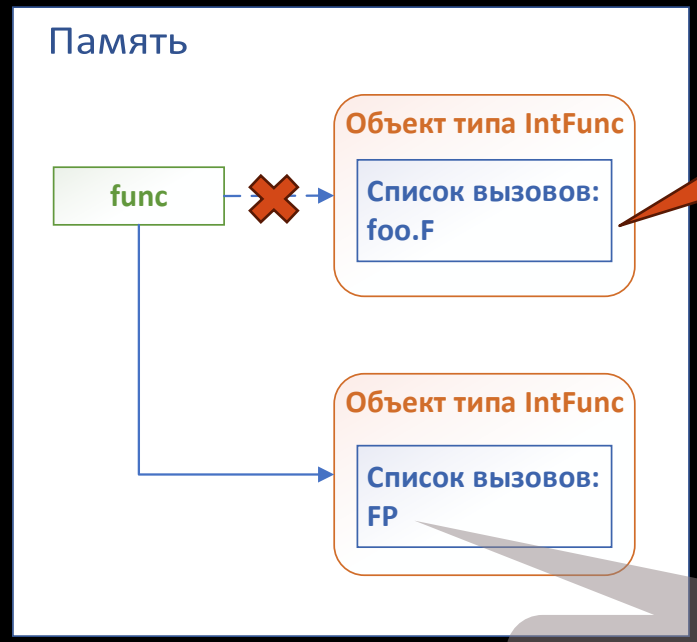
Время компиляции

```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

Время исполнения

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        func = FP;
    }
}
```



Ссылка на этот объект исчезнет при переназначении

Ссылка func теперь связана с другим экземпляром делегата и метод в списке вызовов другой

ПРИМЕР ИСПОЛЬЗОВАНИЯ METHOD И TARGET ДЕЛЕГАТА

```
using System.Collections.Generic;

public delegate int[] Row();     // Делегат-тип и класс в файле DigitSplitter.cs.
public class DigitSplitter {
    private int _value;
    public DigitSplitter(int value) => _value = value;
    public int[] GetDigitsArray() {
        int copy = _value;
        List<int> digits = new List<int>();
        while (copy != 0) {
            digits.Add(copy % 10);
            copy /= 10;
        }
        return digits.ToArray();
    }
}
```

Вывод:

```
The method called is Int32[] GetDigitsArray()
The target object is DigitSplitter
```

```
using System;
// Основная программа – класс Program.cs.
class Program
{
    static void Main()
    {
        DigitSplitter splitter = new(12345);
        Row delRow = splitter.GetDigitsArray;
        Console.WriteLine($"The method called is {delRow.Method}");
        Console.WriteLine($"The target object is {delRow.Target}");
    }
}
```

СПИСОК ВЫЗОВОВ ЭКЗЕМПЛЯРОВ ДЕЛЕГАТ-ТИПОВ

Список вызовов делегата (*Invocation List*) хранит ссылки на методы, связанные с экземпляром и соответствующие сигнатуре делегата-типа

Особенности:

- Допустимы ссылки и на статические, и на экземплярные методы
- Порядок вызова методов в списке вызовов соответствует их порядку добавления
- Допустимо добавить ссылку на метод одного класса / объекта в список вызовов более одного раза
- Попытка вызова экземпляра с пустым списком вызовов приводит к `NullReferenceException`

ОБЪЕДИНЕНИЕ СПИСКОВ ВЫЗОВОВ ОДНОТИПНЫХ ДЕЛЕГАТОВ

```
delegate int IntFunc(int x);
public class Foo
{
    public int F(int x) => 2 * x;
}
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        // Объединение списков вызовов.
        IntFunc combination = (IntFunc)Delegate.Combine(func, sFunc);
    }
}
```

Метод вернёт новый экземпляр делегата с объединенным списком вызовов

ОПЕРАЦИИ НАД ДЕЛЕГАТАМИ

- + позволяет получить новый делегат, состоящий из списков вызовов методов двух других
  - Вы можете также объединять делегаты с методами
- - позволяет убрать методы/списки методов из делегата
  - Если указанного метода в списке вызовов нет, ничего не произойдёт, если их несколько – будет удалено последнее вхождение метода в список вызовов
- += и -= работают интуитивным образом, но копий объекта на создают

```
MyDel myDel = DemoObject.StaticPrint;
DemoObject myInstObj = new DemoObject();
myDel += myInstObj.Print;                    // Добавление в список вызовов.
```

Операндами могут быть как делегаты, так и отдельные методы

ОБЪЕДИНЕНИЕ СПИСКОВ ВЫЗОВА ОДНОТИПНЫХ ДЕЛЕГАТОВ

```
delegate int IntFunc(int x);
public class Foo
{
    public int F(int x) => 2 * x;
}
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        // Объединение списков вызовов.
        func += sFunc;
    }
}
```

Запуск func(5) заставит выполнится оба метода из списка вызовов от аргумента 5… но значение возврата мы получим только из одного метода

Можно ли достучаться до отдельных методов из списка вызовов?

ОТДЕЛЬНЫЕ ВЫЗОВЫ МЕТОДОВ ИЗ СПИСКА ВЫЗОВА ДЕЛЕГАТА

```
Foo foo = new Foo();
IntFunc func = foo.F;
IntFunc sFunc = FP;
// Объединение списков вызовов.
func += sFunc;
```

Допускается объявление делегатов типов с непустым возвращаемым значением (т.е. любым, кроме **void**)

```
Console.WriteLine("One call via func:");
Console.WriteLine(func(5));
```

```
6
```

```
Console.WriteLine("Separately via InvocationList:");
foreach(Delegate d in func.GetInvocationList())
{
    Console.WriteLine(((IntFunc)d).Invoke(5));
}
```

Требуется явно приведение типа

```
10
6
```

ОБЪЕДИНЕНИЕ СПИСКОВ ВЫЗОВОВ: ОПЕРАЦИЯ +=

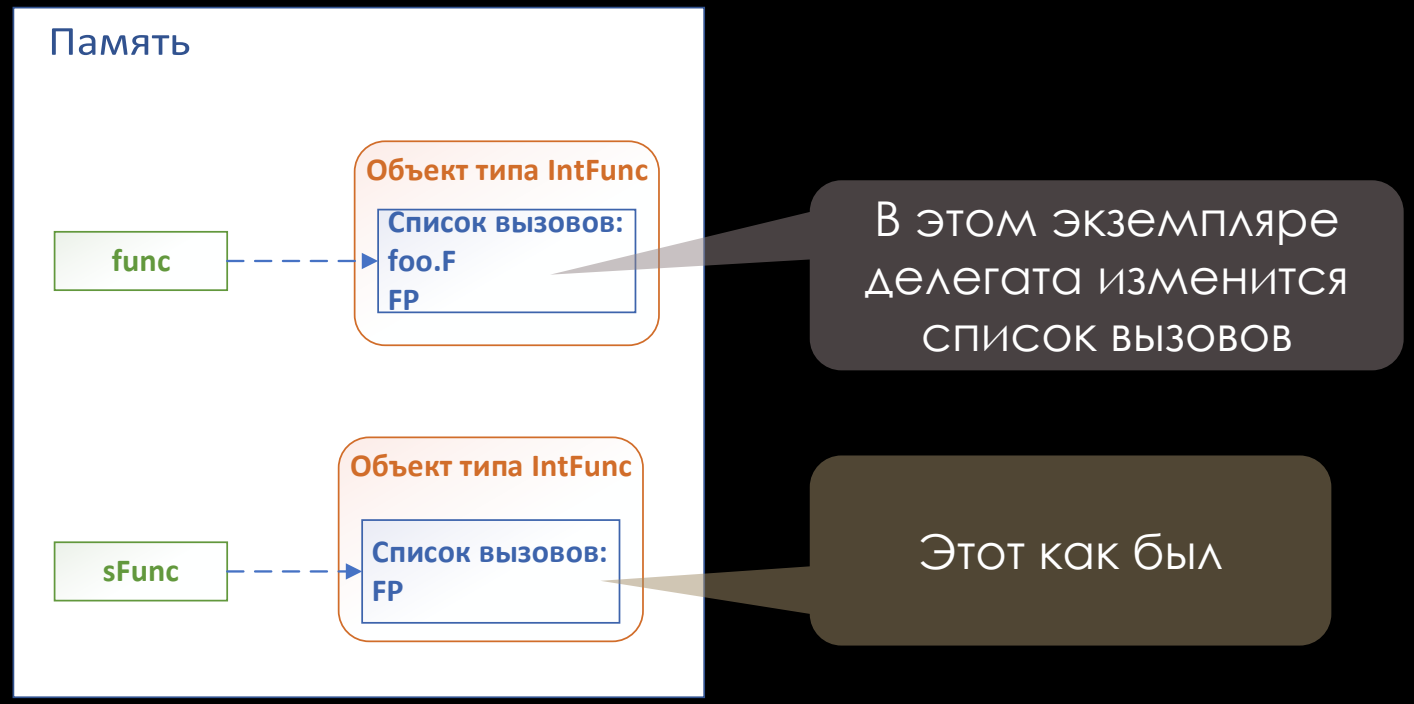
Время компиляции

```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

Время исполнения

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        // Объединение списков вызовов.
        func += sFunc;
    }
}
```



ОБЪЕДИНЕНИЕ СПИСКОВ ВЫЗОВОВ: ОПЕРАЦИЯ +

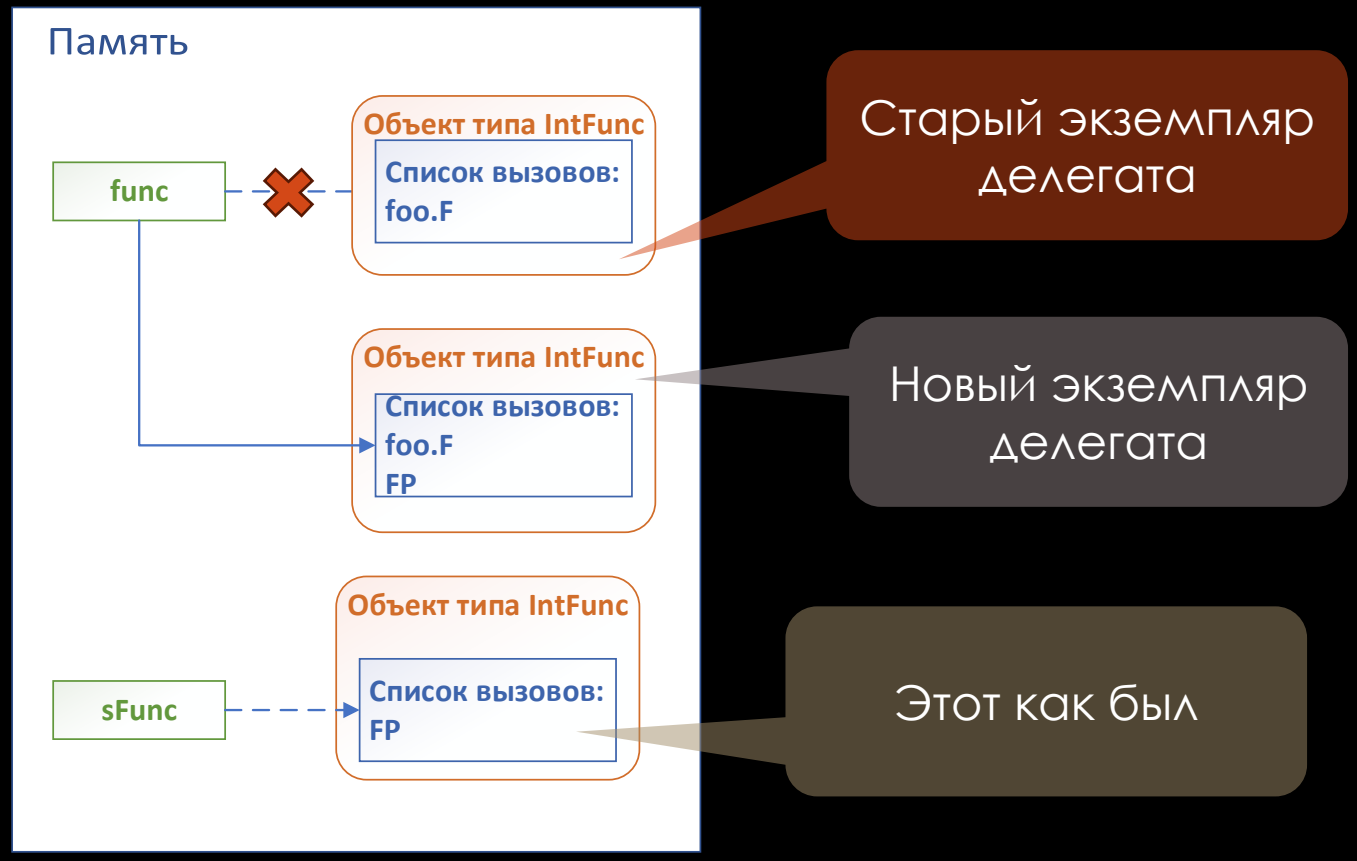
Время компиляции

```
delegate int IntFunc(int x);

public class Foo
{
    public int F(int x) => 2 * x;
}
```

Время исполнения

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        // Объединение списков вызовов.
        func = Func + sFunc;
    }
}
```



УДАЛЕНИЕ МЕТОДОВ ИЗ СПИСКА ВЫЗОВОВ

```
public class Program
{
    public static int FP(int x) => x + 1;
    public static void Main()
    {
        Foo foo = new Foo();
        IntFunc func = foo.F;
        IntFunc sFunc = FP;
        // Объединение списков вызовов.
        func += sFunc;
        func -= sFunc;
        // func -= FP; так тоже можно в этом случае.
        foreach (Delegate d in func.GetInvocationList())
        {
            Console.WriteLine(((IntFunc)d).Invoke(5));
        }
    }
}
```

```
delegate int IntFunc(int x);
public class Foo
{
    public int F(int x) => 2 * x;
}
```

\-

Создаёт копию объекта

\-=

Не создаёт копию объекта

ПРИМЕР ВЫЗОВА ЭКЗЕМПЛЯРНЫХ МЕТОДОВ

```
delegate void PrintDel(int val);
// Файл TargetDelegateDemo.cs – определяет делегат и класс
// с двумя методами, один из которых статический.
public class TargetDelegateDemo
{
    public void Print(int value)
            => Console.WriteLine(value);
    public static void StaticPrint(int value)
            => Console.WriteLine($"static: {value}");
}
```

```
class Program {
    static void Main() {
        TargetDelegateDemo p = new TargetDelegateDemo();
        PrintDel printDel = p.Print;
        printDel += TargetDelegateDemo.StaticPrint;
        printDel += TargetDelegateDemo.StaticPrint;
        printDel += p.Print;
        Delegate[] delegates = printDel.GetInvocationList();
        for (int i = 0; i < delegates.Length; ++i) {
            if (delegates[i].Target != null) {
                ((PrintDel)delegates[i]).Invoke(i);
            }
        }
    }
}
```

Проверка, что метод принадлежит объекту

Вывод:

```
0
3
```

Приведение типов требуется: неизвестно, как делать вызов для базового типа Delegate

ПРИМЕР: КЛАСС И ДЕЛЕГАТ

```
// Файл: Student.cs. Определяет 2 метода с одинаковым набором параметров и типов
// возвращаемого значения для соответствия сигнатуре делегата.
public class Student
{
    private static uint _studentCount = 0;
    private int _group;

    public Student(int group)
    {
        ++_studentCount;
        _group = group;
    }
    public void PrintGroup()
            => Console.WriteLine(_group);
    public static void PrintStudentsCount()
            => Console.WriteLine($"There are {_studentCount} student(s).");
}
```

```
// Файл: Program.cs. Объявляется делегат-тип, используется его
// экземпляр с методами класса/объекта Student.
class Program
{
    static void Main()
    {
        Student student = new(216);
        MyDel studentCaller = student.PrintGroup;
        studentCaller += Student.PrintStudentsCount;
        studentCaller += student.PrintGroup;

        studentCaller();
    }
}
```

Вывод:

```
216
There are 1 student(s).
216
```

ПЕРЕДАЧА МЕТОДОВ В МЕТОДЫ

Обратный вызов

Параметры с типом делегатов

ОБРАТНЫЙ ВЫЗОВ

Обратный вызов [callback, “call-after” function] – любой код, который передаётся как параметр в другой код и ожидается, что этот код выполнит [обратный вызов], переданного в параметре кода в нужное время

В основе обратного вызова лежит идея «функция как параметра»

ВАРИАНТЫ РЕАЛИЗАЦИИ ОБРАТНОГО ВЫЗОВА

делегат

интерфейс

ОБРАТНЫЙ ВЫЗОВ НА ИНТЕРФЕЙСАХ

```
public interface CallbackB
{
    void MethodB(string data);
}
```

```
public class PrinterProxy
{
    public static void Data (int n, CallbackB b)
    {
        for(int i = 0; i < n; i++)
        {
            string data;
            data = $"data: {i} produced by A";
            b.MethodB(data);
        }
    }
}
```

```
public class DataConsolePrinter : CallbackB
{
    string name;
    public DataConsolePrinter(string name)
    {
        this.name = name;
    }
    public void MethodB(string data)
    {
        Console.WriteLine($"{name}
                                processed {data}");
    }
}
```

```
CallbackB printer = new DataConsolePrinter("Console Printer");
PrinterProxy.Data(10, printer);
```

ОБРАТНЫЙ ВЫЗОВ НА ДЕЛЕГАТАХ

```
public delegate void CallbackB(string data);

public class PrinterProxy
{
    public static void Data(int n, CallbackB b)
    {
        for (int i = 0; i < n; i++)
        {
            string data;
            data = $"data: {i} produced by A";
            b(data);
        }
    }
}
```

```
public class DataConsolePrinter
{
    string name;
    public DataConsolePrinter(string name)
    {
        this.name = name;
    }
    public void MethodB(string data)
    {
        Console.WriteLine($"{name}
                                processed {data}");
    }
}
```

```
DataConsolePrinter printer = new DataConsolePrinter("Console Printer");
PrinterProxy.Data(10, printer.MethodB);
```

РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ

делегат

- Используется событийный шаблон проектирования
- Необходимо инкапсулировать статический метод
- Вызывающему коду не нужно обращаться к другим свойствам, методам или интерфейсам объекта, реализующего метод
- Требуется обеспечить лёгко реализуемую композицию
- Классу (в перспективе) может потребоваться более одной реализации метода

интерфейс

- Есть набор связанных методов, которые должна быть возможность вызывать
- Классу требуется только одна реализация метода
- Класс, использующий интерфейс, захочет привести этот интерфейс к другим типам интерфейсов или классов
- Реализуемый метод связан с типом или особенностями структуры класса: например, методы сравнения

КОМПАРАТОР В МЕТОДЕ SORT<T>

Компараторы в Sort<T> предоставляют обе возможности обратного вызова:

Интерфейс

Sort<T>(T[], IComparer<T>)	Sorts the elements in an Array using the specified IComparer<T> generic interface.
Sort<T>(T[], Comparison<T>)	Sorts the elements in an Array using the specified Comparison<T>.

Делегат

```
public static void Sort<T> (T[] array, System.Collections.Generic.IComparer<T>? comparer);

public delegate int Comparison<in T>(T x, T y);
```

РЕАЛИЗАЦИЯ КОМПАРАТОРА С ИНТЕРФЕЙСОМ

```
public class Apple
{
    private double _weigth;
    public double Weight { get => _weigth; }

    public Apple(double weigth = 100)
    {
        if(weigth > 0)
        {
            _weigth = weigth;
        }
    }
}
```

```
public class AppleComparator : IComparer<Apple>
{
    public int Compare(Apple lh, Apple rh)
    {
        if (lh != null && rh != null)
        {
            return lh.Weight == rh.Weight ? 0 : lh.Weight > rh.Weight ? 1 : -1;
        }
        return -2;
    }
}
```

```
Apple[] apples = { new Apple(40.5), new Apple(),
                   new Apple(100.7), new Apple(22), new Apple(34.8)};
AppleComparator ac = new AppleComparator();
Console.WriteLine("Before sorting::");
foreach(Apple cur in apples)
{
    Console.Write(cur.Weight + " ");
}
Array.Sort(apples, ac);
Console.WriteLine($"{Environment.NewLine}After sorting::");
foreach (Apple cur in apples)
{
    Console.Write(cur.Weight + " ");
}
```

РЕАЛИЗАЦИЯ КОМПАРАТОРА ДЕЛЕГАТОМ

```
public class Program
{
    static int ApplesCompare(Apple lh, Apple rh)
    {
        if (lh != null && rh != null)
        {
            return lh.Weight == rh.Weight ? 0 : lh.Weight > rh.Weight ? 1 : -1;
        }
        return -2;
    }
    public static void Main()
    {
        Apple[] apples = { new Apple(40.5), new Apple(),
            new Apple(100.7), new Apple(22), new Apple(34.8)};
        Console.WriteLine("Before sorting::");
        foreach (Apple cur in apples)
        {
            Console.Write(cur.Weight + " ");
        }
        Array.Sort(apples, ApplesCompare);
        Console.WriteLine($"{Environment.NewLine}After sorting::");
        foreach (Apple cur in apples)
        {
            Console.Write(cur.Weight + " ");
        }
    }
}
```

```
public class Apple
{
    private double _weigth;
    public double Weight { get => _weigth; }

    public Apple(double weigth = 100)
    {
        if(weigth > 0)
        {
            _weigth = weigth;
        }
    }
}
```

ОБРАЩЕНИЕ К ДЕЛЕГАТАМ В МЕТОДАХ

Вызов делегата с пустым списком вызовов приведёт к **NullReferenceException**

Для обработки этого случая можно выполнить явную проверку:

- `if (myDel != null) myDel();    // Явная проверка.`
- `myDel?.Invoke();               // Операция ?. + явный вызов.`

```
public delegate int ArrayTransformer(int value);

static void ArrayForEach(int[] array, ArrayTransformer transform)
{
    if (transform == null) { return; }
    for (int i = 0; i < array.Length; ++i)
    {
        array[i] = transform(array[i]);
    }
}
```

ДЕЛЕГАТЫ И ПЕРЕДАЧА ПАРАМЕТРОВ ПО ССЫЛКЕ

При работе с делегатами может возникнуть необходимость изменять переданное значение в процессе вызовов методов по цепочке

Для реализации такого сценария используется **модификатор ref**, который позволяют передавать ссылку последовательно по цепочке

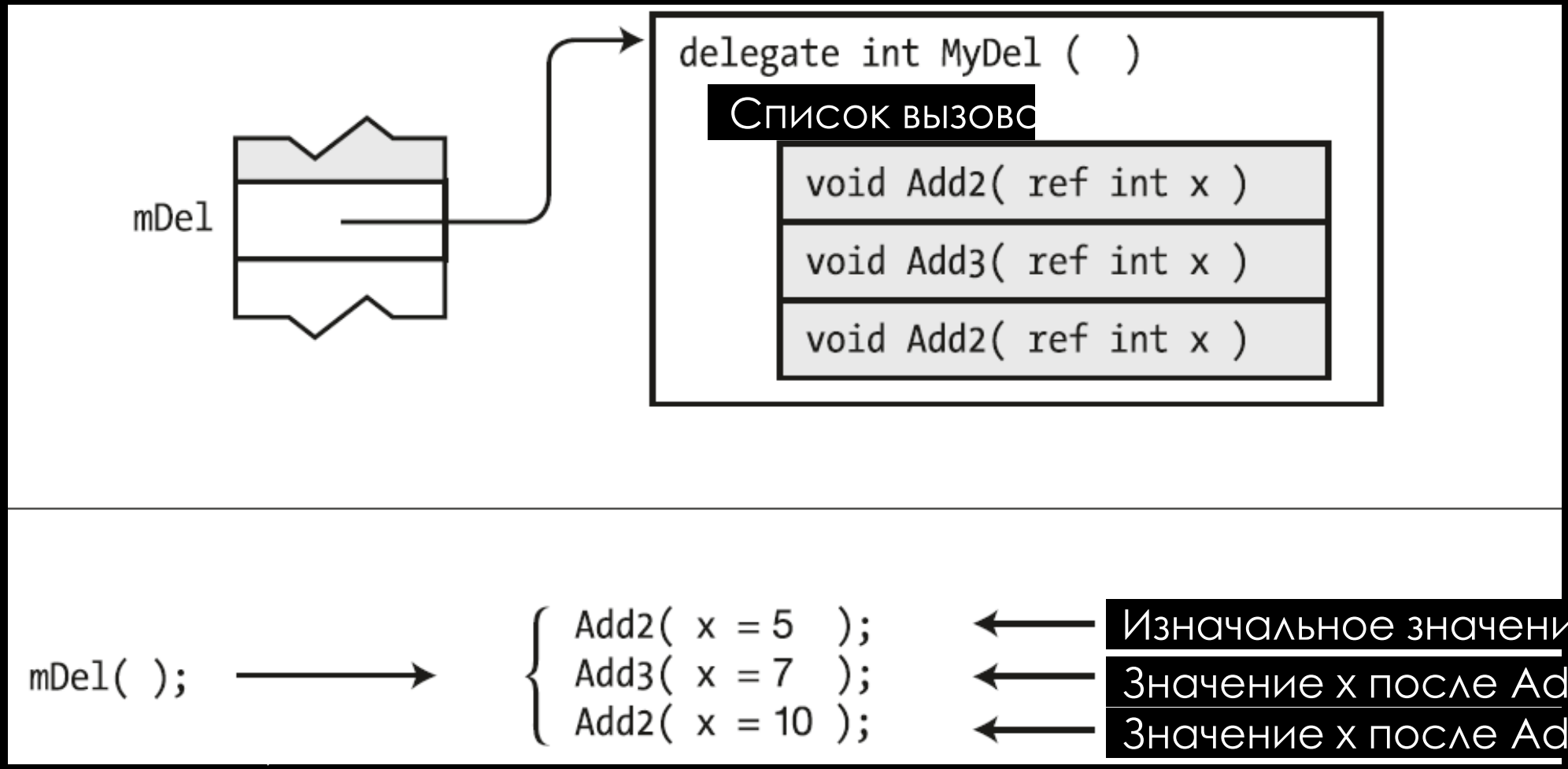
```
MyDel mDel = Add2;
mDel += Add3;
mDel += Add2;
int x = 5;
mDel(ref x);
System.Console.WriteLine($"Value: {x}");

void Add2(ref int x) { x += 2; }
void Add3(ref int x) { x += 3; }

delegate void MyDel(ref int x);
```

Значение **x** передаётся в каждый из методов по ссылке в порядке их добавления

ДЕЛЕГАТЫ И ПЕРЕДАЧА ПАРАМЕТРОВ ПО ССЫЛКЕ: СХЕМА



ПРИМЕР 3: МАССИВЫ ДЕЛЕГАТОВ

```
delegate void Steps(); // Делегат-тип.
class Robot            // Класс для представления робота.
{
    int x, y;          // Положение робота на плоскости.

    public void Right() { ++x; }     // Направо.
    public void Left() { --x; }      // Налево.
    public void Forward() { ++y; }     // Вперёд.
    public void Backward() { --y; }    // Назад.

    public void PrintPosition()           // Печать текущих координат.
            => Console.WriteLine($"The Robot is located at: x={x}, y={y}");
}
```

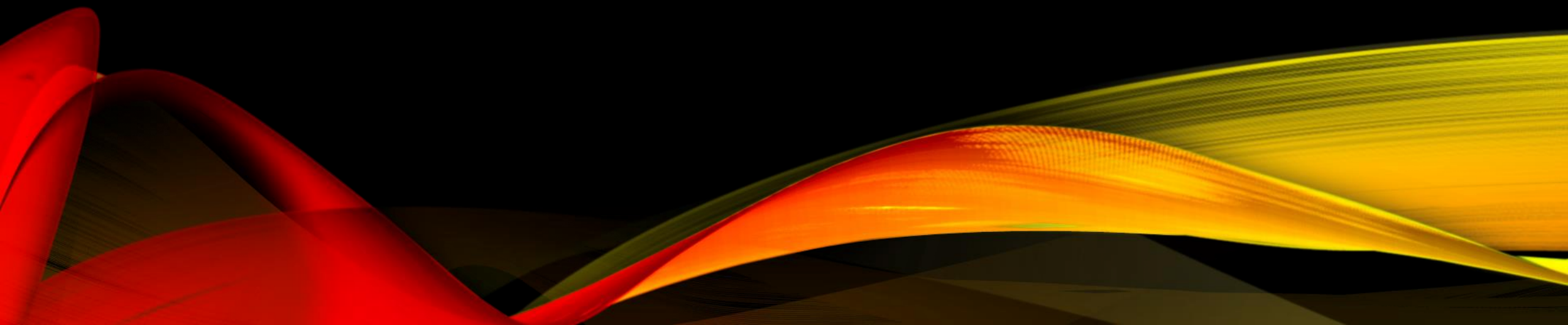
Вывод:

```
Method=Void Backward(), Target=Robot
Method=Void Backward(), Target=Robot
Method=Void Left(), Target=Robot
The Robot is located at: x=-1, y=-2
```

```
class Program {
    static void Main()  {
        Robot rob = new Robot();      // Создание робота.
        Steps[] trace = { rob.Backward, rob.Backward, rob.Left };

        for (int i = 0; i < trace.Length; i++)  {
            Console.WriteLine($"Method={trace[i].Method},
                                        Target={trace[i].Target}");
            trace[i]();
        }
        rob.PrintPosition();          // Вывод итоговых координат.
    }
}
```

ВСТРОЕННЫЕ ДЕЛЕГАТЫ-ТИПЫ



БИБЛИОТЕЧНЫЕ ДЕЛЕГАТЫ В BCL

В пространстве имён `System` определён набор стандартных делегат-типов, который позволяет не создавать собственные делегат-типы без необходимости:

- унификации кода и покрытие большинство сценариев за исключением случаев, когда нужны модификаторы ref / in / out или params

- System.Action – делегат-типы, для которых указывается 0-16 типов входных параметров, а тип возвращаемого значения всегда void
- System.Func – делегат-типы, для которых указывается 0-16 типов входных параметров, а тип возвращаемого значения задаётся последним параметром

ДЕЛЕГАТЫ ACTION (ТИП ВОЗВРАТА VOID)

```
public delegate void Action();

public delegate void Action<in T>(T obj);

public delegate void Action<in T1,in T2>(T1 arg1, T2 arg2);

...

public delegate void Action<in T1,…,in T16>(T1 arg1,…, T16 arg16);
```

T, T<N> – типизирующие параметры

SYSTEM.ACTION<>

- Обобщённый тип делегата используется для создания экземпляров, которые могут быть связаны с методом, не возвращающим значение, но имеющим от одного до шестнадцати параметров разных типов

```
public delegate  void Action<in T>(T obj);
```

Пример из (Asad, A.; Hamza, A., The C# Programmer's Study Guide (MCSD): Exam: 70-483)

```
using System;
class MyClass {
    static void myintMethod(int i) {
        Console.WriteLine("myintMethod: i = {0}", i);
    }
    static void myintStringMethod(int i, string s) {
        Console.WriteLine("myintStringMethod: i = {0} s = {1}", i, s);
    }
    static void Main(string[] args) {
        Action<int> myIntAct = myintMethod;
        Action<int, string> myIntStringAct = myintStringMethod;
        myIntAct(22);
        myIntStringAct(22, "Ali");
    }
}
```

ПРИМЕР. ПЕРЕДАЧА МЕТОДА В МЕТОД ЧЕРЕЗ ACTION<T>

Пример из MSDN

Печать элементов списка **List<T>** при помощи метода **List<T>.ForEach()** с параметром типа **Action<>**

```
class Program {
    static void Main() {
        List<String> names = new List<String>();
        names.Add("Bruce");
        names.Add("Alfred");
        names.Add("Tim");
        names.Add("Richard");
        // Display the contents of the list using the Print method.
        names.ForEach(Print);
        // The following demonstrates the anonymous method feature of C#
        // to display the contents of the list to the console.
        names.ForEach(delegate (String name) {
            Console.WriteLine(name);
        });
    }

    private static void Print(string s) { Console.WriteLine(s); }
}
```

ДЕЛЕГАТЫ FUNC (ТИП ВОЗВРАТА TRESULT)

```
public delegate TResult Func<out TResult>();

public delegate TResult Func<in T,out TResult>(T arg);



public delegate TResult Func<in T1,…,in T16,out TResult>
(T1 arg1,…, T16 arg16);
```

, T<N>, TResult – типизирующие параметры

SYSTEM.FUNC<>

- Документация по C# для ссылки на метод, который имеет один параметр и возвращает значение, рекомендует использовать обобщённый делегат **Func<T,Tresult>**

```
public delegate TResult Func<in T,out TResult>(T arg);
```

Пример из (Asad, A.; Hamza, A., The C# Programmer's Study Guide (MCSD): Exam: 70-483)

```
using System;
class MyClass {
    static int Add(int x, int y) {
        Console.Write("{0} + {1} = ", x, y);
        return (x + y);
    }
    static int Min(int x, int y) {
        Console.Write("{0} - {1} = ", x, y);
        return (x - y);
    }
    static int Mul(int x, int y) {
        Console.Write("{0} * {1} = ", x, y);
        return (x * y);
    }
    static string Name() {
        Console.Write("My name is = ");
        return "Ali Asad";
    }
    static string DynamicName(string name) {
        Console.Write("My name is = ");
        return name;
    }
}
```

```
static void Main(string[] args) {
        //return string value
        Func<string> info = Name;
        Console.WriteLine(info());
        //return string, and take string as parameter
        Func<string, string> dynamicInfo = DynamicName;
        Console.WriteLine(dynamicInfo("Hamza Ali"));
        //return int, and take two int as parameter
        Func<int, int, int> calculate = Add;
        calculate += Min;
        calculate += Mul;
        foreach (Func<int, int, int>
                        item in calculate.GetInvocationList()) {
            Console.WriteLine(item(10, 5));
        }
    }
```

SYSTEM.PREDICATE<>

```
public delegate bool Predicate<in T>(T obj);
```

Используется методами классов Array и List<T>

Немного модифицированный пример из документации

```
public class Example
{
    public static void Main()
    {
        Point[] points = { new Point(100, 200),
                           new Point(150, 250), new Point(250, 375),
                           new Point(275, 395), new Point(295, 450) };

        Predicate<Point> predicate = FindPoints;
        Point first = Array.Find(points, predicate);
        Console.WriteLine($"Found: X = {first.X}, Y = {first.Y}");
    }
    private static bool FindPoints(Point obj) => obj.X * obj.Y > 100000;
}
```

Структура из System.Drawing

Пример из (Asad, A.; Hamza, A., The C# Programmer's Study Guide (MCSD): Exam: 70-483)

```
using System;
class MyClass {
    static bool Even(int i) {
        return (i % 2 == 0);
    }
    static void Main(string[] args) {
        Predicate<int> isEven = Even;
        Console.WriteLine(isEven(7));
    }
}
```

ССЫЛКИ С ИСТОЧНИКАМИ

- Краткий обзор возможностей: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/
- Инструкция по использованию делегатов: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates
- Основы работы с делегатами: https://docs.microsoft.com/en-us/dotnet/csharp/delegate-class
- Класс Delegate: https://docs.microsoft.com/en-us/dotnet/api/system.delegate?view=net-6.0
- Класс MulticastDelegate: https://docs.microsoft.com/en-us/dotnet/api/system.multicastdelegate?view=net-6.0
- Исходный код System.Delegate (partial class): https://github.com/dotnet/runtime/blob/main/src/coreclr/nativeaot/System.Private.CoreLib/src/System/Delegate.cs https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Delegate.cs Исходный код System.MulticastDelegate: https://github.com/dotnet/runtime/blob/main/src/coreclr/nativeaot/System.Private.CoreLib/src/System/MulticastDelegate.cs
- Использование рефлексии для работы с делегатами: https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/how-to-hook-up-a-delegate-using-reflection
- Статья с информацией и разными источниками про внутреннее устройство делегатов: https://mattwarren.org/2017/01/25/How-do-.NET-delegates-work/
- Action: https://docs.microsoft.com/en-us/dotnet/api/system.action?view=net-6.0
- Func: https://docs.microsoft.com/en-us/dotnet/api/system.func-1?view=net-6.0
- Исходный код MulticastDelegate: https://github.com/dotnet/coreclr/blob/01a9eaaa14fc3de8f11eafa6155af8ce4e44e9e9/src/mscorlib/src/System/MulticastDelegate.cs#L622-L627
- Описание, как внутри устроены делегаты и события: https://www.codeproject.com/Articles/26936/Understanding-NET-Delegates-and-Events-By-Practice#Internal
- Статья про внутреннее устройство делегатов с большим количеством ссылок: https://mattwarren.org/2017/01/25/How-do-.NET-delegates-work/