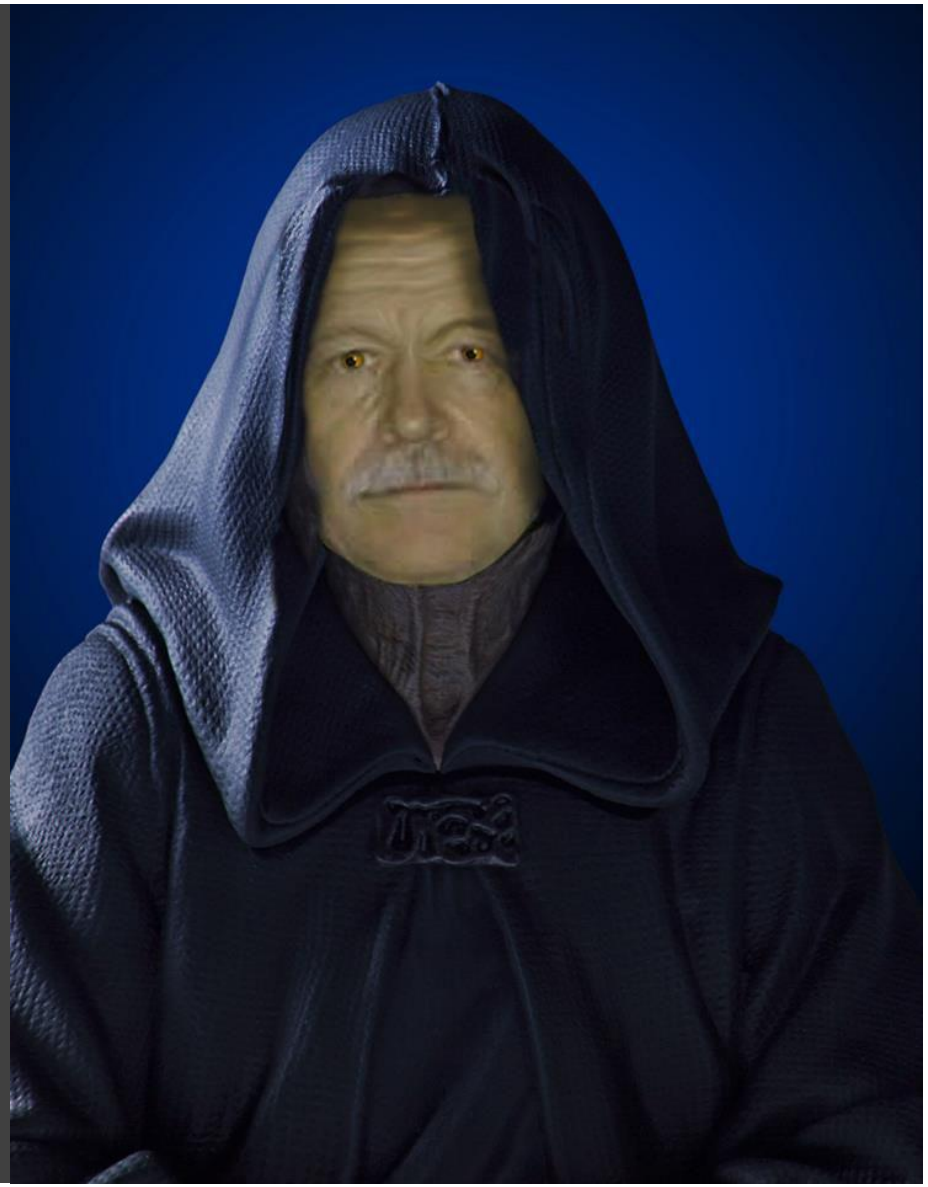
ПИШНИК: BECOME HUMAN

Глава 9: Делегаты. Анонимные Методы. Лямбда-Выражения

Автор презентации – Сагалов Даниил БПИ-196



Делегаты

Делегаты – специальный типы, объекты которых ссылаются на методы с определённым набором параметров и конкретным типом возвращаемого значения. Вы можете воспринимать делегаты как безопасные ссылки, которые указывают на список методов, которые им надо вызвать. По умолчанию при создании делегаты имеют значение null. При компиляции делегаты разворачиваются в спец классы. Все методы, добавленные в делегат вызываются в порядке их добавления. Помните, что вызов пустого делегата вызовет **NullReferenceException**!

При объявлении делегат-типа вы НЕ можете пометить его как **static** (но вы сможете создавать статические поля делегат-типов), **abstract** или **virtual** (для типов данных нет смысла в виртуальности), при использовании **params** в объявлении типа при создании экземпляров params опускается.

Для делегатов вводится собственное понятие сигнатуры, которое отличается от классического определения.

В сигнатуру в контексте работы с делегатами входит:

- **тип возвращаемого значения метода**
- типы параметров и их модификаторы (ref = out = in, params не учитывается).

В сигнатуру делегата не входит:

- **имя метода**
- имена параметров

Создание Делегатов

Типы делегатов объявляются при помощи ключевого слова **delegate**. После объявления делегат-типа вы можете создавать его экземпляры при помощи new и списка вызовов – набора методов, которые делегат будет вызывать.

Помните, что для делегатов также существует упрощённый синтаксис – инициализация без new. Делегаты могут указывать как на экземплярные, так и на статические методы.

Пример:

```
delegate void myDel(int x); // объявляем делегат-тип
myDel delEx1 = new MyDel(myInstObj.MyM1); // ссылка на метод MyM1 объекта myInstObj
myDel delEx2 = myInstObj.MyM1; // укороченный синтаксис создания делегата
```

Делегаты как и все ссылочные типы неизменны, т. е. любые изменения делегата приводят к генерации нового объекта. Ещё одной особенностью делегатов является то, что в один и тот же метод может быть добавлен в список вызовов делегата несколько раз.

Пример Создания Делегат-Типа

```
// Объявление делегат-типа. Обратите внимание - вне класса
public delegate void MyDel(int index);
Ссылок: 0
class Program
{
    Ссылок: 0
    static void Main() {
        // Создадим объект типа Student
        Student s = new Student();
        // Создадим объект типа MeDel и добавим ему экземплярный метод
        MyDel delegate1 = new MyDel(s.GetGroup);
        // Заметьте, аналогично мы можем добавить и статический метод
        MyDel delegate2 = Student.GetOverallId;

        // Вызываем
        delegate1(4);
        delegate2(4);
    }
}
Ссылок: 4
public class Student {
    static uint count = 0;
    ссылка: 1
    public Student() => ++count;
    ссылка: 1
    public void GetGroup(int num) => Console.WriteLine(num);

    ссылка: 1
    public static void GetOverallId(int num) =>
        Console.WriteLine($"Всего {count} студентов в группе {num}");
}
```

Операции с Делегатами

Для делегатов перегружены операции +, -, += и -=.

+ 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 делегате нет, ничего не произойдёт, если их несколько – удалит последнее вхождение метода в список вызовов.

+= и -= работают аналогичным образом

Всегда помните, что вы можете прибавлять/вычитать как делегаты, так и отдельные методы.

Пример:

```
Student s = new Student( );
MyDel delegate1 = new MyDel(s.GetGroup);
MyDel delegate2 = Student.GetOverallId + delegate1;
```

Вызов Делегатов

Вызов делегата аналогичен вызову обычного метода, при вызове делегата будут вызваны все содержащиеся в списке вызовов методы. Важно помнить, что делегат может быть пустым (т. е. иметь значение null) и вызов его в таком виде приведёт к **NullReferenceException**. По этой причине вы можете воспользоваться проверкой на null или оператором **?.** :

```
if (myDel != null) myDel( );
myDel?.Invoke( );
```

Важно понимать, что при вызове делегата с возвращаемым значением (т. е. не void) будет возвращено **значение последнего метода, добавленного в список вызовов** (при этом всё равно отработают все методы).

Пример:

```
class Program {
    static int number = 0;
    static int GetVal(int value) => number += value + 1;
    delegate int Returner(int val);
    static void Main( ) {
        Returner valReturn = GetVal;
        valReturn += GetVal;
        Console.WriteLine(valReturn(2)); // Вернёт только 6
    }
}
```

```
delegate int Summator(ref int val);
Ссылок: 0
class Program {
    // 4 метода, изменяющих число
    Ссылок: 2
    static int IncreaseVal(ref int value) => value += 42;
    ссылка: 1
    static int DecreaseVal(ref int value) => value -= 42;
    ссылка: 1
    static int MultiplyVal(ref int value) => value *= 42;
    ссылка: 1
    static int DevideVal(ref int value) => value /= 42;
    Ссылок: 0
    static void Main() {
        int x = -42;
        Summator summa = IncreaseVal;
        summa += summa + IncreaseVal;
        summa += DevideVal;
        summa += DecreaseVal;
        summa += MultiplyVal;
        Console.WriteLine(summa(ref x));
    }
}
```

ref в Делегатах

Иногда возникает необходимость изменить переданное значение в процессе вызовов методов делегата – на помощь приходит ref, который позволяет по цепочке передавать значение из метода в метод, а в итоге вернуть результат выполнения всей цепочки.

```
delegate int Summator(ref int val);
Ссылок: 4
class DelegatesInvokationList
{
    // 4 метода, изменяющих число
    public int x = -42;
    Ссылок: 2
    int IncreaseVal(ref int value) => value += 42;
    ссылка: 1
    static int DecreaseVal(ref int value) => value -= 42;
    ссылка: 1
    int MultiplyVal(ref int value) => value *= 42;
    ссылка: 1
    static int DevideVal(ref int value) => value /= 42;
    Ссылок: 0
    static void Main()
    {
        DelegatesInvokationList list = new DelegatesInvokationList();
        // Добавляем в список вызовов как статические, так и экземплярные методы
        Summator summa = list.IncreaseVal;
        summa += summa + list.IncreaseVal;
        summa += DevideVal;
        summa += DecreaseVal;
        summa += list.MultiplyVal;
        // Получаем полный список вызовов внутри делегата
        Delegate[] invokationList = summa.GetInvocationList();
        Console.WriteLine($"Длина списка вызовов: {invokationList.Length}");
        foreach (var method in invokationList)
        {
            if (method.Target != null)
                // Приводимтип к нашему типу делегата (в данном случае это всегда возможно)
                summa += method as Summator;
        }
        // Видим, что в список вызовов повторно добавлены только экземплярные методы
        Console.WriteLine($"Длина списка вызовов: {summa.GetInvocationList().Length}");
        Console.WriteLine(summa(ref list.x));
        // MultiplyVal - Экземплярный, получаем ссылку на созданный объект
        DelegatesInvokationList list1 = summa.Target as DelegatesInvokationList;
        // Убедимся, что ссылки в итоге указывают на один и тот же объект
        Console.WriteLine(list1 == list);
    }
}
```

Method и Target. InvocationList

На самом деле, все делегаты наследуются от классов **Delegate → MulticastDelegate**, за счёт чего у каждого создаваемого делегат-типа имеется набор методов и свойств. Только система и компиляторы могут наследоваться от данных классов, мы же используем ключевое слово **delegate**.

Method – свойство делегата, которое возвращает информацию о последнем методе (объект типа **MethodInfo**) в списке вызовов делегата.

Target – свойство, которое возвращает ссылку типа Object на последний объект в списке вызовов делегата. Если последний метод списка является статическим, свойство вернёт null.

Метод **Delegate[ ] GetInvocationList( )** возвращает список вызовов делегата. С ним можно проводить различные манипуляции как в данном примере, в котором мы повторно добавляем в список вызовов делегата summa только экземплярные методы.

Некоторые Делегаты System

В пространстве имён System определён набор стандартных делегатов, который позволяет не создавать собственные делегат-типы без необходимости. Данные делегаты нередко используются в качестве параметров методов.

Action/Action<in T> – делегат-тип, который может принимать от 0 до 16 контрвариантных параметров и при этом ничего не возвращает.

Func<out TResult>/Func<in T, out TResult> – делегат-тип, который работает аналогично Action, однако в конце списка его параметров всегда должен указываться ещё 1 – возвращаемое значение.

Predicate<in T>(T obj) – делегат-тип, принимающий на вход 1 параметр типа T и возвращающий bool.

Converter<in TInput,out TOutput>(TInput input) – делегат-тип, предназначенный для конвертации элемента типа TInput к типу TOutput.

Comparison<in T>(T x, T y) – делегат-тип, возвращающий int. Кодстайл рекомендует возвращать 1, когда объект x «больше» y, 0, когда они равны, и -1, если x «меньше» y.

Задачи

Задание 1

В результате выполнения фрагмента программы:

```
using System;

class Program {
    delegate void MyDel(ref int x);
    static void Main() {
        int x = 5;
        MyDel md = new MyDel(Meth1);
        md += Meth2;
        md.Invoke(ref x);
        Console.Write(x);
    }

    static void Meth1(ref int y) {
        y += 10;
        Console.Write(1);
    }

    static void Meth2(ref int z) {
        z += 20;
        Console.Write(2);
    }
}
```

на экран будет выведено:

Примечание:
Если возникнет ошибка компиляции, введите: ***
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задание 2

Верно, что делегат (*укажите все верные ответы*)**:**

1) Может быть многоадресным.
2) Может содержать в себе статические методы.
3) Можно сложить с другим делегатом.
4) Может быть уменьшен на другой делегат.
5) Содержит в своей сигнатуре тип возвращаемого значения.

Задание 3

Делегаты могут иметь следующие модификаторы (*укажите все верные ответы*)**:**

1) public;
2) static;
3) virtual;
4) abstract;
5) private;

Задание 4

Делегат может находиться в (*укажите все верные ответы*)**:**

1) Пространстве имён.
2) Глобальном пространстве имён.
3) Классе.
4) Методе.
5) Конструкторе.

Задание 5

В результате выполнения фрагмента программы:

```
using System;

class Program {
    delegate void MyDel();
    static void Main() {
        MyDel md = Meth1;
        md += Meth2;
        md += Meth3;
        MyDel md2 = Meth2;
        md -= md2;
        md.Invoke();
    }

    static void Meth1() => Console.WriteLine(1);
    static void Meth2() => Console.WriteLine(2);
    static void Meth3() => Console.WriteLine(3);
}
```

на экран будет выведено:

Примечание:
Если возникнет ошибка компиляции, введите: ***
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 6

В результате выполнения фрагмента программы:

```
using System;

class Program {
        static void Main() {
            Action[] delegates = new Action[5];
            for (int i = 0; i < delegates.Length; i++) {
                int tmp = i << i;
                if (i % 2 != 0)
                    delegates[i] = () => Console.Write(i + " ");
                else
                    delegates[i] = () => Console.Write(tmp + " ");
            }
            foreach (Action actionDelegate in delegates)
                actionDelegate?.Invoke();
        }
    }
```

на экран будет выведено:

Задание 7

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) Делегат-тип в пространстве имён может быть помечен модификатором protected;
2) Разрешается создавать абстрактные делегат-типы;
3) Делегаты могут являться параметрами методов;
4) Делегаты Func всегда имеют какое-либо возвращаемое значение;
5) Статические методы могут содержаться только в статических делегатах;

Задание 8

В результате выполнения фрагмента программы:

```
using System;

class Program {
    delegate MyDel2 MyDel();
    delegate int MyDel2();
    static void Main() {
        MyDel md = DelMeth;
        MyDel2 md2 = Meth2;
        Console.Write(md2.Invoke() + (md2 + md.Invoke()).Invoke() *
2);
    }

    static MyDel2 DelMeth() => new MyDel2(Meth1) + new
MyDel2(Meth3);
    static int Meth1() => 1;
    static int Meth2() => 2;
    static int Meth3() => 3;
}
```

на экран будет выведено:

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Ответы

Задание	Ответ
1	**1235**
2	**12345**
3	**125** *(static может быть при создании поля типа делегата)*
4	**123**
5	**13**
6	**058564**
7	**34**
8	**8**

Анонимные Методы

При работе с делегатами порой приходится добавлять в список вызовов делегата метод, который разово используется только внутри данного делегата. В таких ситуациях гораздо лучшим решением будет применения **анонимных методов** –методов без имени, которые создаются при помощи оператора **delegate**. Анонимные методы не бывают статическими, а при их объявлении без параметров круглые скобки можно опускаться – это единственная функция анонимных методов, которая не поддерживается лямбда-выражениями.

Важно: Вы не сможете удалить анонимный метод из списка вызовов делегата, если примените оператор -= с объявлением такого же анонимного метода.

Сравните такой вариант:

```
public static int GetLollipops(int lollipopCount) {
    return lollipopCount % 10000;
}
Func<int, int> example1 = GetLollipops;
```

И такой:

```
Func<int, int> example1 = delegate(int lollipopCount) {
    return lollipopCount * 10000;
};
```

Объявление Анонимного Метода

И так, для того чтобы создать анонимный метод нужно:

- ключевое слово delegate
- список параметров в скобках (опционально, при отсутствии параметров скобки можно опустить)
- блок операторов в фигурных скобках

```
Func<int, int> example = delegate // скобки отсутствуют
        {
            return 5 * 10000;
        };
```

При работе с анонимными методами стоит помнить, что Target и Method всё также будут возвращать объект и сам метод, однако он будет содержать специальные символы, добавленные компилятором.

Увеличение Времени Жизни Переменной

При работе с анонимными методами следует помнить об областях видимости. Все переменные, созданные внутри анонимного метода, будут существовать только внутри самого метода. Однако, существует такое понятие, как увеличение времени жизни:

```
delegate void MyDel( );
        static void Main( ) {
            MyDel del;
            {
                int x = 5;
                del = delegate {
                    Console.WriteLine("Value of x: {0}", x);
                };
            }
            del?.Invoke( ); // В данном случае ошибки не возникнет, существование x продлено
        }
```

Итерационные Переменные и Делегаты

Важно: при захвате итерационных переменных и продлении времени их жизни захватывается только последнее значение итерационных переменных! По этой причине, если вы хотите сохранить значения итерационных переменных, придётся каждый раз внутри цикла создавать временную переменную для каждого делегата. Данная проблема касается for, но НЕ затрагивает foreach.

Без временной переменной:

```
static void LambdaCaptureForUnsafe()
{
    Action[] delegatesExample1 = new Action[3];
    for (int i = 0; i < delegatesExample1.Length; i++)
        delegatesExample1[i] = () => Console.Write(i + " ");

    // Для всех делегатов выведется 3! Запоминается только последнее значение i
    foreach (Action del in delegatesExample1)
        del?.Invoke();
}
```

С временной переменной:

```
static void LambdaCaptureForSafe()
{
    Action[] delegatesExample1 = new Action[3];
    for (int i = 0; i < delegatesExample1.Length; i++)
    {
        // Каждый раз создаётся новая временная переменная, она связывается с делегатом
        int temp = i;
        delegatesExample1[i] = () => Console.Write(temp + " ");
    }

    // Итог: Выведется 0 1 2
    foreach (Action del in delegatesExample1)
        del?.Invoke();
}
```

Лямбда-Выражения

Аналогично анонимным методам, **лямбда-выражения** позволяют создавать «короткие методы», которые могут использоваться, например, внутри других методов или при работе с делегатами. Помните, что все типы параметров лямбда-метода должны быть либо явными, либо неявными.

Лямбда-выражения имеют следующий синтаксис:

```
( ) => { <Блок Оператор> }
<Параметр> => <Выражение>
(<Тип1> <Параметр1>, <Тип2> <Параметр2>, …) => <Выражение>
```

Примеры:

```
delegate int ExampleDel( int X ); // делегат-тип

ExampleDel lambda1 = (int x) => { return x + 1; } ;
ExampleDel lambda2 = (x) => { return x + 1; } ;
ExampleDel lambda3 = x => { return x + 1; } ;
ExampleDel lambda4 = x => x + 1 ; // самый короткий вариант!
```

Лямбда-Методы в Действии

```
// Делегат-Предикат
static Predicate<string> lambdaBool;
Ссылок: 0
static void Main()
{
    string inputStr = Console.ReadLine();
    // Проверка длины строки, используется лямбда-метод
    lambdaBool = line => line.Length <= 10 ? true : false;
    if (lambdaBool(inputStr))
        Console.WriteLine(inputStr);
    else
    {
        // Проверка на чётность количества символов в строке
        lambdaBool = line => line.Length % 2 == 0 ? true : false;
        if (lambdaBool(inputStr))
            foreach (char latter in inputStr)
                // Проверка кратности кода символа 2
                Console.Write(latter % 2 != 0 ? latter.ToString() : "");
        else
            Console.WriteLine(inputStr.Substring(0, 10));
    }
}
```

Задачи

Задание 1

Про анонимный метод верно (*укажите все верные ответы*)**:**

1) Можно с помощью оператора -= удалить его из делегата, скопировав полностью его определение.
2) Объявляется с ключевым словом delegate.
3) Не может возвращать void.
4) Использует лямбда-оператор для объявления тела метода.
5) Не может быть статическим.

Задание 2

Выберите все лямбда-выражения, являющиеся верными по синтаксису (*укажите все верные ответы*)**:**

1) `(x, y) => Console.WriteLine(x + y);`
2) `(int x, int y) => x + y;`
3) `() => Console.WriteLine(10);`
4) `x, y => Console.WriteLine(x + y);`
5) `=> Console.WriteLine(10);`

Задание 3

Допустимы следующие лямбда-выражения (*укажите все верные ответы*)**:**

1) `() => { return; };`
2) `(int x) => x + x;`
3) `int x => x + x;`
4) `(int x) => return x;`
5) `x => 2 * x;`

Задание 4

В результате выполнения фрагмента программы:

```
using System;

delegate string Sum(int number);
class Program {
    static Sum SomeVar() {
        string s = "";
        Sum del = delegate (int number) {
            for (int i = 0; i < number; i++)
                s += i.ToString();
            return s;
        };
        return del;
    }
    static void Main() {
        Sum del1 = SomeVar();
        for (int i = 0; i <= 3; i++) {
            Console.Write(del1(i));
        }
    }
}
```

на экран будет выведено:

Ответы

Ответ 1: **25**

Ответ 2: **123**

В 4 более 1 параметра вне скобок – так нельзя.

Ответ 3: **125**

Ответ 4: **0001001012**

Методы Array с Делегатами

Среди методов класса Array можно найти сразу несколько методов, принимающих на вход делегаты. Зачастую они позволяют получить более короткую версию записи решения какой-либо задачи:

TOutput Array.ConvertAll(TInput[ ] array, Converter<TInput, TOutput> converter) – статический метод, позволяющий конвертировать все элементы массива с использованием делегата Converter, возвращает массив полученного типа.

bool Exists(T[ ] array, Predicate<T> condition) – возвращает true, если заданный массив содержит элементы, удовлетворяющие переданному условию и false, если нет.

T Find<T>(T[ ] array, Predicate<T> condition) – возвращает первое вхождение элемента в массиве, удовлетворяющее условиям поиска или значение тип T по умолчанию, если элемент не найден. Для поиска с конца существует аналог **FindLast**.

T[ ] FindAll<T> (T[ ] array, Predicate<T> condition) – возвращает все элементы массива, удовлетворяющие переданному условию.

Методы Array с Делегатами

int FindIndex<T> (T[ ] array, int startIndex, int length, Predicate<T> match) – возвращает индекс первого вхождения элемента, удовлетворяющего условию match, на участке длины length, начиная с индекса startIndex, или -1, если элемент не найден. Для последнего вхождения существует аналог **FindLastIndex**.

void ForEach<T> (T[ ] array, Action<T> action) – метод, позволяющий совершить действие action с каждым элементом переданного массива.

void Sort<T> (T[ ] array, Comparison<T> comparer) – сортирует массив с использованием делегата Comparison для сравнения элементов.

bool TrueForAll<T> (T[ ] array, Predicate<T> match) – возвращает true, если для всех элементов выполняется условие предиката и false в противном случае.

```
int[] ints = { 2, 6, 4, 5, 17, 8, 10, 3, 1, 14 };
// Конвертируем весь массив в double
double[] doubles = Array.ConvertAll(ints, delegate (int value)
{
    if (value >= 10) return value + 0.1;
    return value * 0.1;
});
// Выводим результат: 0,2 0,6 0,4 0,5 17,1 0,8 10,1 0,3 0,1 14,1
Array.ForEach(doubles, value => Console.Write(value + " "));
// Проверим, все ли значения можно конвертировать в тип decimal
if (Array.TrueForAll(doubles, doubleVal => {
    decimal tmp;
    return Decimal.TryParse(doubleVal.ToString(), out tmp);
}))
    Console.WriteLine("Все числа данного массива конвертируемы в decimal");

// Отсортируем элементы массива в порядке убывания
Array.Sort(doubles, (double doubleVal1, double doubleVal2) => -doubleVal1.CompareTo(doubleVal2));
// Выведем полученный массив: 17,1 14,1 10,1 0,8 0,6 0,5 0,4 0,3 0,2 0,1
Array.ForEach(doubles, value => Console.Write(value + " "));

// Соберём массив чисел, дробная часть которых оканчивается на 0.1
double[] endWithOne = Array.FindAll(doubles, (double doubleVal) => {
    // Math.Round используется для устранения погрешности
    if (Math.Round(doubleVal - Math.Truncate(doubleVal), 1) == 0.1) return true;
    return false;
});
// Выведем полученный массив: 17,1 14,1 10,1 0,1
Array.ForEach(endWithOne, value => Console.Write(value + " "));
```

Пример Работы Методов Array