В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

Модуль 3. Лекция 1b

Анонимные Методы. Лямбда-выражения.

Благодарность Д.Д. Сагалову за помощь в переработке презентаций

Анонимные Методы

При работе с методами обратных вызовов (callback) периодически возникает необходимость передавать в качестве аргументов **небольшие методы, которые используются разово (только в одном месте)**.

Объявлять для этих целей отдельные методы (и классы) часто нецелесообразно, т.к. это приводит к “раздуванию” исходного кода.

Для решения данной проблемы существует два варианта синтаксиса **анонимных функций**:

- Анонимные методы (C# 2.0);
- Лямбда-выражения (C# 3.0, альтернатива анонимным методам).

Анонимные методы: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator
Лямбда-выражения: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions
Анонимные методы vs. Лямбда-выражения:
https://stackoverflow.com/questions/299703/delegate-keyword-vs-lambda-notation

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

https://docs.microsoft.com/ru-ru/dotnet/api/system.action?view=net-6.0

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, T<N>, TResult – типизирующие параметры

https://docs.microsoft.com/ru-ru/dotnet/api/system.func-1?view=net-6.0

Объявление Анонимных Методов

Для объявления анонимных методов используется специальный синтаксис, указывается:

- Ключевое слово delegate;
- Список параметров (может опускаться);
- Тело.

```
class Program
{
    static void Main()
    {
        System.Func<int, int, int> sumFunc = delegate (int x, int y)
        {
            return x + y;
        };
    }
}
```

ключевое слово ↓ — delegate

список параметров ↓ — (int x, int y)

↑ тело анонимного метода

Анонимные Методы vs. Методы Класса

```
class Program
{
  public static int Add20(int x)
  {
    return x + 20;
  }

  delegate int OtherDel(int InParam);
  static void Main()
  {
    OtherDel del = Add20;

    Console.WriteLine("{0}", del(5));
    Console.WriteLine("{0}", del(6));
  }
}
```

Именованный Метод

```
class Program
{

  delegate int OtherDel(int InParam);
  static void Main()
  {
    OtherDel del = delegate(int x)
                   {
                     return x + 20;
                   };
    Console.WriteLine("{0}", del(5));
    Console.WriteLine("{0}", del(6));
  }
}
```

Анонимный Метод

Пропуск Списка Параметров

При объявлении анонимных методов можно опускать параметры – это позволяет преобразовать анонимный метод к делегату-типу с любым списком параметров. Сами параметры в таком случае будут игнорироваться (отбрасываться). Это единственная возможность, которая доступна для анонимных методов и не доступна для лямбда-выражений.

```
using System;

Action noParams = delegate { Console.WriteLine("Anonymous..."); };
noParams();

Action<int, double> twoParams = delegate { Console.WriteLine("...methods"); };
twoParams(42, 3.14);
```

Область Видимости Переменных и Параметров

Локальные переменные и параметры анонимных методов существуют только внутри охватывающих фигурных скобок.

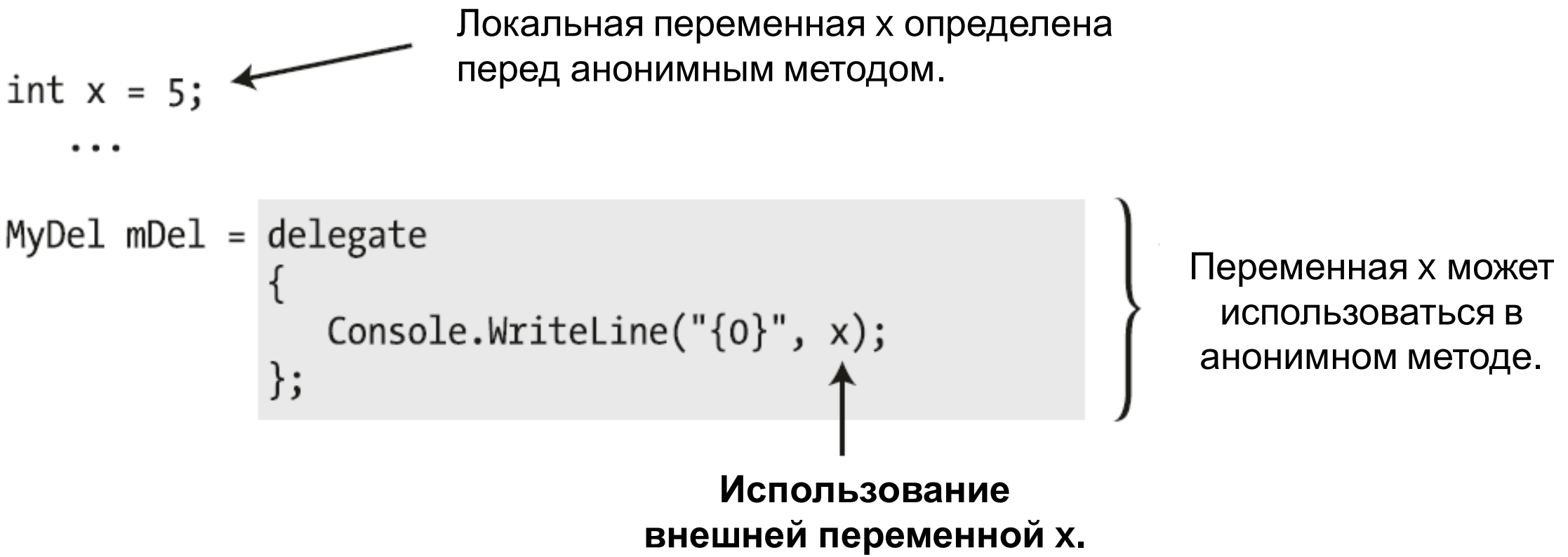
```
delegate void MyDel( int x );
...

MyDel mDel = delegate ( int y )
             {
                int z = 10;
                Console.WriteLine("{0}, {1}", y, z);
             };

Console.WriteLine("{0}, {1}", y, z);     // Ошибка компиляции.
```

Область видимости y и z

y и z уже не существуют

Внешние Переменные и Анонимные Методы

Для нестатических анонимных методов внешние переменные захватываются автоматически и доступны без дополнительных ограничений.



Продление Времени Жизни Локальных Переменных

```
delegate void MyDel( );
static void Main()
{
   MyDel mDel;
   {
      int x = 5;
      mDel = delegate
             {
                Console.WriteLine("Value of x: {0}", x);
             };
   }

   // Console.WriteLine("Value of x: {0}", x);

   if (null != mDel)
      mDel( );
}
```

Переменная x определена во вложенном блоке вне анонимного метода.

Область видимости **x**.

Переменная **x** захватывается анонимным методом.

Т.к. переменная **x** вышла из области видимости, раскомментирование данной строки приводит к ошибке компиляции.

x была захвачена анонимным методом и корректно используется при его вызове.

Объявление Лямбда-Выражений

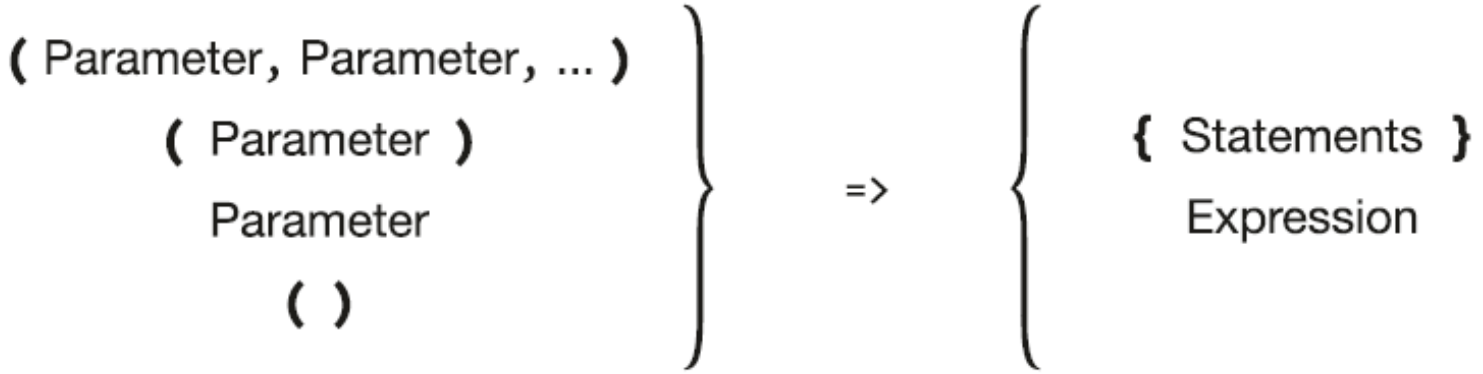
Начиная с C# 3.0 появилась поддержка лямбда-выражений в качестве альтернативы анонимным методам.

Для лямбда-выражений можно выделить два основных варианта синтаксиса:

- С телом, представленным единственным выражением:
  ```
  (входные_параметры) => выражение;
  ```

- С телом, представленным блоком операторов:
  ```
  (входные_параметры) => { <операторы> };
  ```

Примеры Объявлений Лямбда-Выражений




```
using System;

Func<int, int> lambda1 = (int x) => { return x + 1; };
Func<int, int> lambda2 = (x) => { return x + 1; };   // Без явного типа параметра.
Func<int, int> lambda3 = x => { return x + 1; };     // Без скобок.
Func<int, int> lambda4 = x => x + 1;                 // Тело – выражение.
Action lambda5 = () => Console.WriteLine("lambda"); // Без параметров.

// С неявным указанием параметров:
Action<int, int> lambda6 = (x, y) => Console.WriteLine(x * y);
// С явным типом возвращаемого значения (C# 10.0):
ByRefDel lambda7 = ref int (ref int val) => ref val;

delegate ref int ByRefDel(ref int x);
```

Правила Объявления Лямбда-Выражений-1

Для каждого из вариантов синтаксиса предусмотрены определённые правила:

- Между списком параметров и телом лямбда-оператор => указывается всегда;

- Типы всех параметров указываются либо полностью явно, либо полностью неявно:

```
Func<string, int, string> lambda = (string li, int num) => li + num;
Func<string, int, string> lambda = (line, num) => line + num;
```

- Если у лямбда-выражения только один входной параметр, скобки можно опустить:

```
Action<int> lambda = num => Console.WriteLine(x + 1);
```

Правила Объявления Лямбда-Выражений-2

- Для лямбда-выражений без параметров круглые скобки обязательны:

```
Action lambda = () => Console.WriteLine("lambda");
```

- Начиная с C# 9.0, символ _ можно использовать для пропуска двух и более параметров (для одиночного _ считается именем локальной переменной в целях обратной совместимости):

```
Func<int, int, int> lambda = (_, _) => 42;
```

- Начиная с C# 10.0, для лямбда-выражений допускается явное указание типа возвращаемого значения перед списком параметров:

```
Func<bool, dynamic> lambda = dynamic (bool b) => b ? 1 : "two";
```

Пример: Использование Лямбда-Выражений

```
using System;

// Если строка не длиннее 10 символов, вывести её.
string inputStr = Console.ReadLine();
Func<string, bool> condition = line => line.Length <= 10;
if (condition(inputStr)) {
    Console.WriteLine(inputStr);
    return;
}

// Новый предикат: проверка на чётность длины строки.
condition = line => line.Length % 2 == 0;
if (condition(inputStr)) {
    // Вывести все символы, коды которых чётные.
    foreach (char letter in inputStr) {
        Console.Write(letter % 2 != 0 ? letter : "");
    }
}
```

Ввод:
Hello

Вывод:
Hello

Ввод:
abcdefghijkl

Вывод:
acegik

Статические Анонимные Методы

Начиная с C# 9.0 можно указывать модификатор static при объявлении анонимных методов, что меняет их поведение следующим образом:

- Нестатические и не-const переменные и поля не захватываются;
- Нет доступа к ссылкам this/base, даже если такие имелись в охватывающей области видимости.

```
int value = 42;

Func<int> lambda = static () =>
{
    // Ошибка компиляции при попытке раскомментировать следующую строку.
    // return value + 10;
    return 42;
};
```

Анонимные Методы как Возвращаемые Значения

Анонимные методы могут выступать в качестве возвращаемых значений в функциях, возвращающих экземпляры делегат-типов или Expression:

```
static Func<double, double, double, double> GetFunction(int a, int b, int c)
{
    return (a, b, c) => a * a * b + c;
}
```

```
static Expression<Func<string, int>> GetExpression(string line)
{
    return (line) => int.Parse(line);
}
```

Анонимные Методы и Модификатор params

При использовании делегат-типов с модификатором **params** при объявлении анонимных методов ключевое слово **params** опускается:

```
                                приводит к ошибке компиляции
int prod = 1;                   ↓
ParamsDel mydel = (/* params */ int[] numbers) => {
    int sum = 0;
    foreach (int value in numbers)
    {
        sum += value;
        prod *= value;
    }
    System.Console.WriteLine("Sum = " + sum);
    System.Console.WriteLine("Composition = " + prod);
};
mydel(5, 1, 2, 3);

delegate void ParamsDel(params int[] values);      // Делегат-тип с params.
```

Захват Итерационной Переменной в Цикле for

Анонимные методы могут захватывать итерационные переменные цикла for (сохраняется последнее состояние переменной, т.е. копии значений не создаются):

```
using System;

Action[] iterationCapture = new Action[3];
for (int i = 0; i < iterationCapture.Length; ++i)
{
    iterationCapture[i] = () => Console.Write(i + " ");
}
foreach (var lambda in iterationCapture)
{
    lambda();
}
```

При захвате переменная **i** не копируется.

Вывод:
3 3 3

Решение Проблемы Захвата Переменной в Цикле for

Для решения проблемы при захвате достаточно добавить локальную переменную, которая каждый раз будет создаваться заново и захватываться:

```
using System;

Action[] iterationCapture = new Action[3];
for (int i = 0; i < iterationCapture.Length; ++i)
{
    int temp = i;
    iterationCapture[i] = () => Console.Write(temp + " ");
}
foreach (var lambda in iterationCapture)
{
    lambda();
}
```

Теперь каждый раз захватывается отдельная переменная.

Вывод:
0 1 2

Захват Итерационной Переменной в Цикле foreach

Захват итерационной переменной анонимными методами работает предсказуемо: т.к. захватывается конкретный элемент последовательности, пересечений в принципе не возникает:

```
using System;

Action[] iterationCapture = new Action[3];
foreach (int val in new[] { 0, 1, 2 })
{
    iterationCapture[val] = () => Console.Write(val + " ");
}
foreach (var lambda in iterationCapture)
{
    lambda();
}
```

Захватываются никак не связанные друг с другом значения.

Вывод:
0 1 2

Методы Класса Array, Принимающие Делегаты-1

В классе Array определён ряд методов, принимающих параметры делегат-типов. Данные методы позволяют выполнять различные преобразования с массивами и их элементами, осуществлять проверки:

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

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

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

Методы Класса Array, Принимающие Делегаты-2

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

int FindIndex<T> (T[] array, int startIndex, int length, Predicate<T> match) – возвращает индекс первого вхождения элемента, удовлетворяющего предикату match, в диапазоне длины length, начиная с индекса startIndex, или -1, если элемент не найден. Для поиска последнего вхождения существует метод FindLastIndex.

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

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

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

Документация по Array: https://docs.microsoft.com/en-us/dotnet/api/system.array?view=net-6.0

Array Пример 1: Методы ConvertAll и ForEach

```
using System;

int[] ints = { 2, 6, 4, 5, 17, 8, 10, 3, 1, 14 };
// Конвертация всех чисел в double:
double[] doubles = Array.ConvertAll(ints,
    value => value >= 10 ? value + 0.1 : value * 0.1);

// Вывод всех чисел в полученном массиве.
Array.ForEach(doubles, value => Console.Write($"{value:F1} "));
```

Вывод:
0.2 0.6 0.4 0.5 17.1 0.8 10.1 0.3 0.1 14.1

Array Пример 2: Методы Sort и FindAll

```
using System;

double[] doubles = { 0.2, 0.6, 0.4, 0.5, 17.1, 0.8, 10.1, 0.3, 0.1, 14.1 };

// Сортировка всех чисел в порядке убывания и вывод:
Array.Sort(doubles, (value1, value2) => -value1.CompareTo(value2));
Array.ForEach(doubles, value => Console.Write($"{value:F1} "));
Console.WriteLine("\n");

// Собрать и вывести все числа, заканчивающиеся на .1:
double[] endWithPointOne = Array.FindAll(doubles, value => {
        double temp = Math.Round(value - Math.Truncate(value), 1);
        return temp - 0.1 <= 0.00001;
    });
Array.ForEach(endWithPointOne, value => Console.Write($"{value:F1} "));
```

Вывод:

17.1 14.1 10.1 0.8 0.6 0.5 0.4 0.3 0.2 0.1

17.1 14.1 10.1 0.1

Источники

Создание анонимных методов: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator

О лямбда-выражениях: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression

Общая спецификация анонимных функций: https://docs.microsoft.com/en-us/dotnet/csharp/delegate-class

Статические анонимные функции в C# 9.0: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/static-anonymous-functions

Отбрасывание параметров анонимных функций: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/lambda-discard-parameters

Улучшение анонимных функций в C# 10.0: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-10.0/lambda-improvements

Исходный код класса Array, содержащий методы с делегатами: https://github.com/dotnet/runtime/blob/main/src/libraries/System.Private.CoreLib/src/System/Array.cs

Естественные Типы Анонимных Методов

Сами по себе анонимные методы не имеют типа вне контекста. Они преобразуются компилятором либо к делегат-типам, либо к Expression.

Начиная с C# 10.0 появляется такое понятие, как **естественные типы** анонимных методов. Компилятор может выводить тип по следующим правилам:

- Action/Func выводится при наличии подходящей сигнатуры;
- Генерируется автоматически (например, при наличии ref);
- Выведение типа для групп методов возможно только при отсутствии перегрузок/методов расширений;

- Вывод типа работает по ссылкам базового типа (Object/Delegate или Expression/LambdaExpression) по описанным выше правилам.

Важно: Вывод типа может быть невозможен, если компилятор не может однозначно определить типы аргументов.

Пример: Естественные Типы Анонимных Методов

```
using System;
using System.Linq.Expressions;

// Тип выведется как Func<string, int>.
var parseToInt = (string line) => int.Parse(line);
// Тип выведется как Action.
var simplePrint = delegate () { Console.WriteLine("Natural type"); };
// Тип будет сгенерирован компилятором, хранение по ссылке Delegate:
Delegate refReturn = (ref string line) => ref line;
// Явное определение лямбда-выражения как Expression<Func<int, int, int>>:
Expression expression = (int first, int second) => first + second;
// Определение типа группы методов как Func<int> по ссылке Object.
var consoleRead = Console.Read;

// Ошибка: тип параметра не вывести однозначно (string или ReadOnlySpan<char>).
var parseToUnknown = line => int.Parse(line);
// Ошибка: группа методов представляет несколько перегрузок.
var methodGroupError = Console.Write;
```