Лекция №4. Делегаты

Что такое делегаты?

На предыдущих занятиях мы познакомились с двумя видами типов данных, которые существуют в языке C# — значимые типы, которые создаются как структуры (**struct**) и ссылочные типы, которые создаются как классы (**class**). Как вы помните, отличие между ними заключается в том, что значимые типы хранятся там же где и объявлены, а ссылочные типы хранятся в куче (**heap**) и в месте их объявления хранится только ссылка на объект в куче. Но кроме этих двух разновидностей существует еще один — делегаты. **Делегаты — это тип данных, который содержит ссылку на метод**. Для чего это нужно? Давайте рассмотрим на примере алгоритма сортировки, с прошлого семестра:

```
//Метод Sort делегирует интерфейсу IComparer функциональность
//по сравнению элементов массива
public static void SortStrings(string[] array, IComparer<string> comparer)
{
    for (int i = array.Length - 1; i > 0; i--)
    for (int j = 1; j <= i; j++)
    {
        var element1 = array[j - 1];
        var element2 = array[j];
        if (comparer.Compare(element1, element2) > 0)
        {
            var temporary = array[j];
            array[j] = array[j - 1];
            array[j - 1] = temporary;
        }
    }
}
```

Этот метод сортирует массив строк. Но как правильно сортировать строки? Их можно отсортировать по длине или в алфавитном порядке или по какому-нибудь другому правилу. Для того чтобы этот метод был универсальным, в нем не реализовано конкретное правило для сравнения строк, вместо этого мы передаем в него `comparer` (сравниватель), в котором есть метод `Compare` с конкретным способом сортировки. Это является примером паттерна **"Делегирование"**, так как мы **делегируем** сравнение строк из метода `SortStrings` в `comparer`. Давайте посмотрим реализацию разных способов сравнения:

```
//Интерфейс , в котором описано,
//какие методы должны быть у класса
```

```
interface IComparer<T>
{
    int Compare(string x, string y);
}

//Сравнение строк по длине
class StringLengthComparer : IComparer<string>
{
    public int Compare(string x, string y)
    {
        return x.Length.CompareTo(y.Length);
    }
}

//Дефолтное сравнение строк
class StringComparer : IComparer<string>
{
    public int Compare(string x, string y)
    {
        return x.CompareTo(y);
    }
}
```

Теперь передавая разные классы, можно сортировать массив по разному:

```
var strings = new[] { "A", "B", "AA" };
SortStrings(strings, new StringComparer()); //A AA B
SortStrings(strings, new StringLengthComparer()); // A B AA
```

Однако, вы наверное уже заметили проблему у этого подхода. Для обеспечения делегирования приходится писать слишком много инфраструктурного кода: реализовывать интерфейс и объявлять классы. При том, что нам по сути нужны только правила для сравнения строк `x.Length.CompareTo(y.Length)` и `x.CompareTo(y)`. Только эти две строчки являются важными, весь остальной код нужен, только для того чтобы всё работало. Благодаря делегатам, можно значительно сократить количество инфраструктурного кода.

Давайте сделаем это. Сначала нужно объявить делегат для метода сравнения:

```
//Здесь объявляется тип указателей на функции, принимающие
//две строки и возвращающие int.
//Помните! Делегат - это не член класса! Это тип данных.
public delegate int StringComparer(string x, string y);
```

Перепишем метод сортировки, заменив `IComparer<string>` на `StringComparer` и будем использовать для сравнения переданный метод:

```
//Этот метод принимает указатель на соответствующую функцию
public static void SortStrings(string[] array, StringComparer comparer)
{
    for (int i = array.Length - 1; i > 0; i--)
    for (int j = 1; j <= i; j++)
    {
        var element1 = array[j - 1];
        var element2 = array[j];
        //раз это указатель на функцию, значит, эту функцию можно вызвать
        if (comparer(element1, element2) > 0)
        {
            var temporary = array[j];
            array[j] = array[j - 1];
            array[j - 1] = temporary;
        }
    }
}
```

Напишем методы для сравнения строк:

```
static int CompareStringLength(string x, string y)
{
    return x.Length.CompareTo(y.Length);
}

static int CompareString(string x, string y)
{
    return x.CompareTo(y);
}
```

Теперь можно их использовать:

```
    var strings = new[] { "A", "B", "AA" };

    //Здесь создается указатель на метод CompareStringLength
    var lengthComparer =
        new StringComparer(CompareStringLength);
    //... и передается в метод SortStrings
    SortStrings(strings, lengthComparer);

    //Можно писать так. Компилятор сам догадается, что вы хотели создать
    //указатель на метод, а не вызвать его.
    SortStrings(strings, CompareString);
```

Мы смогли сократить количество инфраструктурного кода, сделать программу более компактной и читаемой благодаря использованию делегатов.

Хранение делегатов в памяти

Делегаты могут указывать не только на статические методы, но и на динамические. В этом случае внутри метода можно использовать поля экземпляра класса. Давайте рассмотрим такой пример:

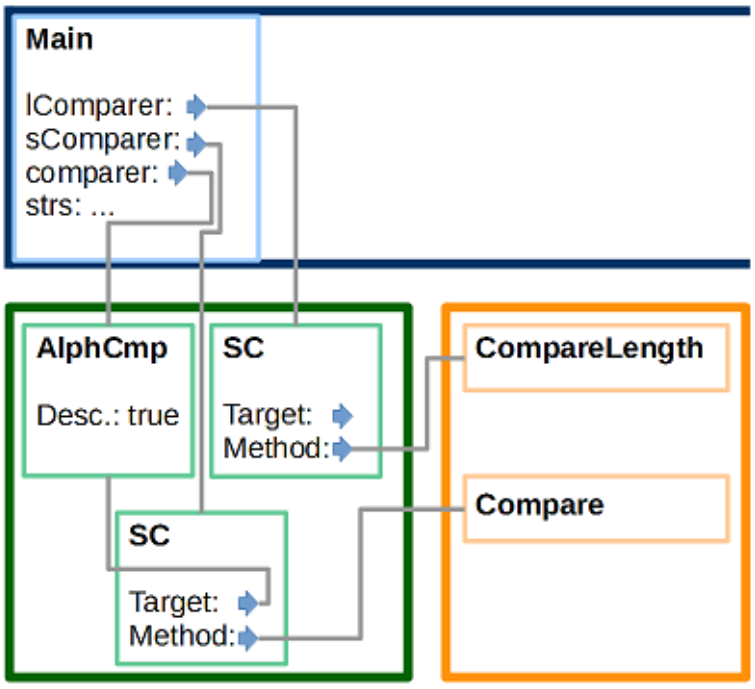
```
class Comparer
{
    //От значения динамческого поля Descending
    //зависит работа метода CompareStrings
    public bool Descending { get; set; }
    public int CompareStrings(string x, string y)
    {
        return x.CompareTo(y) * (Descending ? -1 : 1);
    }
}
```

Теперь можно использовать этот динамический метод:

```
var strings = new[] { "A", "B", "AA" };
var obj = new Comparer { Descending = true };

//Можно без проблем указывать на динамические методы.
//Для того, чтобы указать на метод, нужно обратиться к нему как для вызова
//(с указанием объекта для динамических методов, указанием класса
// для статических методов, определенных в другом классе),
//НО БЕЗ КРУГЛЫХ СКОБОК.
SortStrings(strings, obj.CompareStrings);
```

Как устроены делегаты и как это все выглядит в памяти? Рассмотрим это на простом примере, где есть статический метод для сравнения строк по длине и динамический метод для сравнения по алфавиту, который вызывается из объекта. В памяти это будет выглядеть так. Названия сокращены для компактности.

```
static void Main()
{
    var strs = …

    var lComparer =
        new SC(CompareLength);

    SortStrings(sts, lComparer));

    var comparer = new AlphCmp
            { Descending = true };

    var sComparer =
        new SC(comparer.Compare);

    SortStrings(strs, sComparer);
}
```



Давайте вкратце вспомним на какие части делится память. Синим выделен стек, в него добавляется контекст текущего метода. Зеленым — куча, в ней хранятся все объекты. Оранжевым — часть кучи, в которой хранятся все методы. Методы всегда хранятся отдельно от объектов, как статические так и динамические.

Как вы можете увидеть, внутри делегата есть два поля — `Target` и `Method`. Поле `Method` является ссылкой на метод, на который указывает делегат. Поле `Target` указывает на объект, из которого взят метод если он динамический, если метод статический, то поле `Target` содержит `null`.

Func и Action

Также можно создавать делегаты с generic-типами. Давайте, для примера, сделаем метод для сортировки универсальным, чтобы его можно было использовать не только для сортировки массива строк. Для начала напишем generic-делегат:

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

И немного поменяем метод сортировки:

```
public static void Sort<T>(T[] array, Comparer<T> comparer)
{
    for (int i = array.Length - 1; i > 0; i--)
    for (int j = 1; j <= i; j++)
    {
        var element1 = array[j - 1];
        var element2 = array[j];
        if (comparer(element1, element2) > 0)
```

```
        {
            var temporary = array[j];
            array[j] = array[j - 1];
            array[j - 1] = temporary;
        }
    }
}
```

Теперь можем его использовать:

```
var strings = new[] { "A", "B", "AA" };
//Обратите внимание на вывод типов
//Нам нигде не понадобилось явно указывать тип T
//По первому аргументу компилятор догадывается,
//что T - это string, и понимает, что был передан правильный метод
Sort(strings, CompareStringLength);
var obj = new Comparer { Descending = true };
Sort(strings, obj.CompareStrings);
```

Во всех предыдущих примерах мы использовали свой собственный делегат, но на самом деле это избыточно. Любую функцию можно описать с помощью типов принимаемых аргументов и возвращаемого типа. И в языке C# уже существуют встроенные типы данных для методов — `Func` и `Action`. Они являются generic-делегатами, в которых можно указать произвольное количество типов. `Func` используется для методов, которые возвращают какое-то значение. Сначала идут типы аргументов, а последним записывается возвращаемый тип. Если метод не принимает никакие аргументы, то указывается только возвращаемый тип. `Action` используется для методов, которые не возвращают никое значение (`void`), в нем указывается только передаваемые типы. Давайте перепишем метод сортировки с использованием `Func`:

```
//Func<T, T, int> - первые два типа - аргументы,
//последний - тип возвращаемого значения
public static void Sort<T>(T[] array, Func<T, T, int> comparer)
{
    for (int i = array.Length - 1; i > 0; i--)
    for (int j = 1; j <= i; j++)
    {
        var element1 = array[j - 1];
        var element2 = array[j];
        if (comparer(element1, element2) > 0)
        {
            var temporary = array[j];
            array[j] = array[j - 1];
            array[j - 1] = temporary;
        }
```

```
    }
}
```

Теперь нам не нужно писать собственные делегаты!

Лямбда-выражения

Мы начали использовать делегаты для того, чтобы избавиться от лишнего инфраструктурного кода. И мы достигли больших успехов, однако можно сделать программу еще более компактной с помощью **анонимных делегатов**. Анонимный делегат позволяет не писать метод, а определить его по месту использования. Нам не нужно думать о том где создать метод и как его правильно оформить, компилятор сам напишет метод за вас, сам сгенерирует ему имя и определит тип возвращаемого значения. Давайте посмотрим как это выглядит:

```
    var strings = new[] { "A", "B", "AA" };
    //Передаем в метод сортировки анонимный делегат
    Sort(strings, delegate(string x, string y)
    {
        return x.Length.CompareTo(y.Length);
    });
```

Есть еще более простой способ записи — **лямбда-выражения**. По сути это те же самые анонимные делегаты, но с другим синтаксисом.

```
var strings = new[] { "A", "B", "AA" };
//Лямбда-выражение - еще более краткая форма записи.
//Теперь компилятор догадывается не только до типа возвращаемого значения,
//но и до типа аргументов.
Sort(strings, (x, y) => x.Length.CompareTo(y.Length));
```

Лямбда-выражения имеют разные варианты записи, рассмотрим их:

```
//если функция принимает один аргумент,
//то можно не писать круглые скобки
Func<int, int> f = x => x + 1;
Console.WriteLine(f(1)); //2

//Так писать нельзя, т.к. компилятор
//не сможет автоматически вывести тип
var pow2 = x => x * x

//Но мы можем указать тип аргументов
//тогда можно использовать var
var pow2 = (int x) => x * x
```

```
//Если аргументов нет - просто пишем круглые скобки
var generator = () => rnd.Next();
Console.WriteLine(generator());

//если метод нельзя записать в одну строку
//нужно взять тело метода в скобки
//и вернуть значение с помощью return
var h = (double a, double b) =>
    {
        b = a % b;
        return b % a;
    };

//у Action все аналогично
Action<int> print = x => Console.WriteLine(x);
//теперь print будет работать также как и Console.WriteLine
print(generator());

//будут работать одинаково
Action printRandomNumber = () => Console.WriteLine(rnd.Next());
var printRandomNumber1 = () => print(generator());
```

Замыкание

Давайте рассмотрим такую ситуацию, когда лямбда-выражение объявленное внутри метода использует его локальные переменные. Это называется **замыканием**. Лямбда-выражение становится связанным с этой переменной, если мы поменяем переменную, то она изменится и в лямбда-выражении. Например:

```
//В лямбда-выражении используется локальная переменная Descending
bool Descending = true;
Func<string, string, int> cmp =
    (x, y) => x.CompareTo(y) * (Descending ? -1 : 1);
var strings = new[] { "A", "B", "AA" };
Sort(strings, cmp); //A AA B
//При изменении ее значения,
//лямбда-выражение начинает работать по другому
Descending = false;
Sort(strings, cmp); //B AA A
```

Выглядит несложно, но на самом деле замыкание в некоторых ситуациях может вести себя по странному. Рассмотрим для примера такую задачу, мы решили создать `List` с методами. Каждый метод просто складывает переданное ему число со своим номером в списке (индексом).

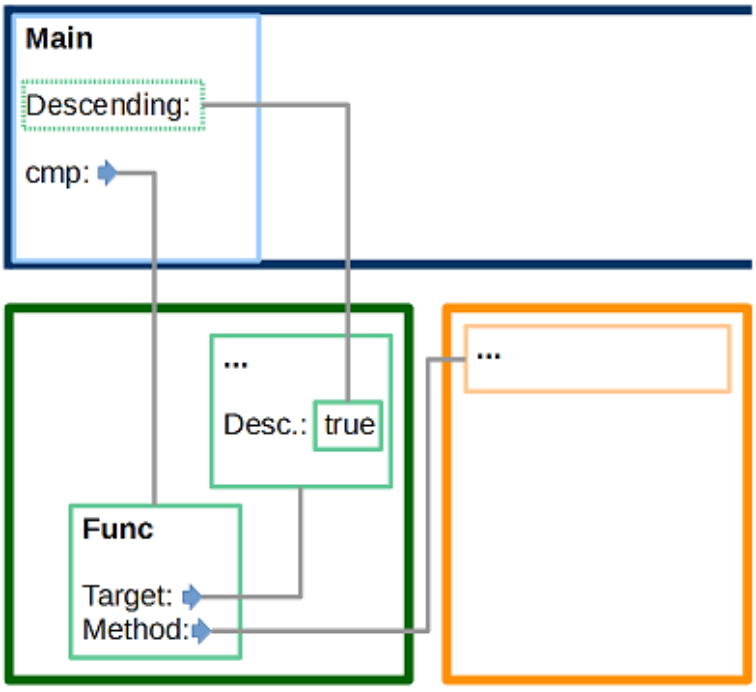
```
var functions = new List<Func<int, int>>();
```

```
for (int i = 0; i < 10; i++)
    functions.Add(x => x + i);

foreach (var e in functions)
    Console.WriteLine(e(0));
```

Мы ожидаем, что в консоль выведутся числа от `0` до `9`. Но если мы запустим код, то увидим, что этот код выведет десять раз число `10`. Почему? Для того, чтобы это понять, нужно разобраться как устроено замыкание в памяти.

Рассмотрим это на первом примере.

```
static void Main()
{
    bool Descending = true;
    Func<string,string,int> cmp=
       (x,y)=>x.CompareTo(y)
                *(Descending?-1:1);

    var strings = …

    Sort(strings, cmp);
    Descending = false;
    Sort(strings,cmp);
}
```



Когда происходит замыкание, компилятор генерирует класс, который содержит все локальные переменные, которые используются в лямбда-выражении. Поля этого класса связаны с локальными переменными, и когда меняются переменные, меняются и поля.

Теперь попытаемся понять из-за чего возникла проблема с созданием лямбда-выражений в цикле. А причина в том, что у всех них замыкание произошло с одной и той же локальной переменной `i`, поэтому все методы и складывают числа с одним и тем же числом — последним значением `i`, которое равно `10`. Соответственно, для того, чтобы код работал правильно, нужно чтобы методы были связаны с разными переменными. Это можно сделать, например, так:

```
var functions = new List<Func<int, int>>();

for (int i = 0; i < 10; i++)
    var j = i;
    functions.Add(x => x + j);
```

```
foreach (var e in functions)
    Console.WriteLine(e(0));
```

Теперь все работает правильно. Код выводит числа от `0` до `9`.