Делегаты

Делегат – это объект, который может ссылаться на метод. Во время выполнения программы один и тот же делегат можно использовать для вызова различных методов, просто заменив метод, на который ссылается этот делегат. Таким образом, метод, который будет вызван делегатом, определяется не в период компиляции программы, а во время ее работы. Делегат в C# соответствует указателю на функцию в C++.

Общий вид объявления делегата:

```
delegate тип_возвращаемого_значения имя_делегата(список_формальных_параметров);
```

Использование делегата:

```
Имя_делегата указатель_на_делегат;
Указатель_на_делегат = new имя_делегата(имя_функции);
```

Здесь используется только имя функции (параметры не указываются).

Пример 1.

```
delegate int getIntDelegate(); // Объявление делегата

class Test
{
    private int _x;
    private decimal _y;
    public Test(int x, decimal y)
    {
        _x = x;
        _y = y;
    }

    public int GetX()
    {
        return _x;
    }

    public int GetIntY()
    {
        return (int)_y;
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        var test = new Test(5, -2.5m);

        // Объявление переменной типа делегата с выставлением для нее в соответствие
метода
        getIntDelegate first = new getIntDelegate(test.GetX);
        getIntDelegate second = new getIntDelegate(test.GetIntY);

        // Использование делегата для вызова метода
        var x = first();
        var y = second();

        Console.WriteLine(x);
        Console.WriteLine(y);

        Console.ReadLine();
    }
}
```

Примечание. В данной функции переменная second лишняя, вместо нее можно было использовать first.

В этом примере переменная типа делегат лишь заменяет имя метода, при этом фиксировано, к какому объекту относится представляемый метод.

Пример 2. Делегаты позволяют использовать имя метода в качестве формальных параметров.

```
delegate decimal method1(decimal param);

class Example
{
    private decimal[] _array;
    private decimal _a;
    private int _size;

    public Example()
    {
        _size = 5;
        _array = new decimal[_size];

        for(var i = 0; i < _array.Length; i++)
        {
            Console.Write($"Array[{i}] = ");
            _array[i] = Convert.ToDecimal(Console.ReadLine());
        }
    }

    public void MethodUsesDelegateAsParameter(method1 method)
    {
        for(var i = 0; i< _array.Length; i++)
        {
            var item = (decimal)method(_array[i]);
            Console.WriteLine($"Array[{i}] = {item}");
            _a += item;
        }
        Console.WriteLine($"Summ = {_a}");
    }

    public static decimal GetPower2(decimal item)
    {
        return item * item;
    }

    public decimal GetMult2(decimal item)
    {
        return item * 2;
    }

}
```

Делегату method1 могут соответствовать только функции, имеющие тип возвращаемого значения decimal и имеющие один формальный параметр типа decimal.

Пример 3. Многоадресный делегат. Одному делегату можно ставить в соответствие несколько функций. В таком случае они будут выполнены в такой последовательности, как они были прикреплены делегату.

```
delegate int MyDeleg(ref string st);

class MyClass
{
    public static int Method1(ref string x)
    {
        Console.WriteLine("I'm Method1");
```

```
        x += "1!";
        return 5;
    }

    public static int Method2(ref string y)
    {
        Console.WriteLine("I'm Method2");
        y += "2!";
        return 55;
    }
}
```

```
MyDeleg deleg;
MyDeleg deleg1 = new MyDeleg(MyClass.Method1);
MyDeleg deleg2 = new MyDeleg(MyClass.Method2);

string myString = "Погнали!";

deleg = deleg1;
deleg += deleg2;

var k = deleg(ref myString);

Console.WriteLine(myString);
Console.WriteLine($"k = {k}");
Console.ReadLine();
```

```
I'm Method1
I'm Method2
Погнали!1!2!
k = 55
```

Делегаты расширяют знакомые нам средства программирования в двух случаях:

- Делегаты как указатели на функцию, что позволяет использовать функции в качестве формальных/фактических параметров других функций.
- Многоадресные делегаты, таким образом получим возможность одним вызовом обеспечить выполнение ряда функций.

Использование делегата в роли псевдонима функции может иногда уменьшить объем наших записей, но не расширяет наши возможности.

Использование делегата Func<>

Делегат Func<T, TResult> имеет следующее определение

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

T – тип входного параметра метода, который инкапсулируется данным делегатом;
TResult – тип возвращаемого значения метода, который инкапсулируется данным делегатом.

Для использования делегата Func<,> необходимо написать ключевое слово Func, ввести в угловых скобках через запятую тип входного параметра и тип возвращаемого значения для данного метода, указать наименование метода (который будет использоваться в рамках данного делегата), поставить знак «=», указать имя параметра, который будет использоваться в теле метода, поставить знак «=>», написать тело метода, которое возвратит результат типа TResult.

```
Func<int, int> FibonacciSeq = null;
```

```
FibonacciSeq = index => index > 1 ? FibonacciSeq(index - 1) + FibonacciSeq(index -
2) : index;
```

В данном выражении определен метод FibonacciSeq, который вызывается рекурсивно для предыдущих элементов последовательности Фибоначчи.
Метод вызывается обычным способом

```
int item15 = FibonacciSeq(15);
```

Асинхронное программирование

При программировании высоконагруженных систем, web-приложений или систем графического редактирования (например) может возникнуть проблема ожидания отклика программного средства. Для пользователя ожидание отклика программы является критичным фактором. Решением данной проблемы может послужить использование асинхронности в разрабатываемом приложении (использование асинхронных методов или блоков кода). В данном случае такие задачи будут выполняться не в основном потоке.
Для написания асинхронного метода необходимо:

1. При описании метода указать модификатор async (что не является прямым показателем того, что метод будет асинхронным);
2. Данный метод должен возвращать:
    a. Task или Task<T>
    b. ValueTask<T> // Структура
    c. Void
3. Метод с модификатором async должен содержать хотя бы одно выражение с ключевым словом await.

Пример определения асинхронного метода FibonacciSequenceAsync()

```
private void FibonacciSequence()
{
    Func<int, int> FibonacciSeq = null;
    FibonacciSeq = index => index > 1 ? FibonacciSeq(index - 1) + FibonacciSeq(index
- 2) : index;

    for (int i = 0; i < _count; i++)
        Console.WriteLine($"Ф_{_name} ({i+1}) = {FibonacciSeq.Invoke(i)}"); //
Использование Expression при необходимости
    Console.WriteLine(new string('_', 40) + _name + " Завершен");
}

#region Async
private async void FibonacciSequenceAsync()
{
    await Task.Run(() => FibonacciSequence());
}
#endregion
```

В случае необходимости получения результата в асинхронном методе необходимо ключевое слово await использовать после знака присваивания или ключевого слова return.

Вызов асинхронного метода ничем не отличается от вызова обычного синхронного метода.

Вызов асинхронных операций

Существует два возможных вызова асинхронных операций:

1. Последовательный
2. Параллельный

Последовательный. Несмотря на то, что вызов заданий расположен в асинхронном методе, они будут выполняться в том порядке, в котором описаны в программном коде.

```
private static void GetItemInPower(double item) => Console.WriteLine($"{item} ^
{item} = {Math.Pow(item, item)}");

 public async static void GetItemInPowerAsync(double item)
 {
     await Task.Run(() => GetItemInPower(item));
     await Task.Run(() => GetItemInPower(item * 2));
     await Task.Run(() => GetItemInPower(item * 3));
 }

 static void Main(string[] args)
 {
     GetItemInPowerAsync(2);
         //2 ^ 2 = 4
         //4 ^ 4 = 256
         //6 ^ 6 = 46656
     Console.ReadLine();
 }
```

Параллельный. Для того, чтобы распараллелить выполнение вышеописанных заданий необходимо запустить задачи через статический метод WhenAll(), принимающий в качестве параметров массив задач.

```
private static void GetItemInPower(double item) => Console.WriteLine($"{item} ^
{item} = {Math.Pow(item, item)}");

public async static void GetItemInPowerAsync(double item)
{
    await Task.Run(() => GetItemInPower(item));
    await Task.Run(() => GetItemInPower(item * 2));
    await Task.Run(() => GetItemInPower(item * 3));
}
public async static void GetItemInPowerParallelAsync(double item)
{
    Task first = Task.Run(() => GetItemInPower(item));
    Task second = Task.Run(() => GetItemInPower(item * 2));
    Task third = Task.Run(() => GetItemInPower(item * 3));
    await Task.WhenAll(new[] { first, second, third });
}

static void Main(string[] args)
{
    GetItemInPowerParallelAsync(2);
    //6 ^ 6 = 46656   // Результат работы непредсказуем
    //4 ^ 4 = 256
    //2 ^ 2 = 4

    Console.ReadLine();
}
```

В данном случае все три задачи запустятся одновременно. Но, помимо обычных задач создаться также задача, которая будет выполнена после завершения всех задач, переданных методу WhenAll().