Delegates

Delegate

- Безопасный к типам callback
- Invoke чтобы вызвать

Простейший пример объявления делегата:

```
internal delegate void WriteMessage();

static void Main(string[] args)
{
    WriteMessage writeDelegate = WriteA;
    writeDelegate.Invoke();
}

internal static void WriteA() => Console.WriteLine("Write Aaa!");
```

```
delegate int BinaryOperation(int x, int y);

private static int Add(int x, int y) => x + y;
private static int Multiply (int x, int y) => x * y;

static void Main(string[] args)
{
    BinaryOperation del = Add;
    Console.WriteLine(del.Invoke(4, 5));

    del = Multiply;
    Console.WriteLine(del.Invoke(4, 5));
}
```

```
internal delegate void MyDelegate(int value);

internal static void WriteA(int value) => Console.WriteLine($"Aaa {value}");
internal static void WriteB(int value) => Console.WriteLine($"Bbb {value}");
internal class Example
{
    internal void WriteC(int value) => Console.WriteLine($"Ccc {value}");
}

private static void UseDelegate(Int32 value, MyDelegate del) => del(value);

static void Main(string[] args)
{
    MyDelegate delegateA = WriteA;          // Статический метод
    MyDelegate delegateB = new MyDelegate(WriteB);
    var example = new Example();
    MyDelegate delegateC = example.WriteC;  // Экземплярный

    delegateA(1);
    delegateB.Invoke(2);
    delegateC(3);

    MyDelegate chain = delegateA;
```

```
    chain = (MyDelegate)Delegate.Combine(chain, delegateB);
    chain += delegateC;
    chain -= delegateA;

    UseDelegate(4, chain);
}
```

```
internal delegate void Message(Int32 value);

internal class Message : System.MulticastDelegate
{
    public Feedback(Object object, IntPtr method);
    public virtual void Invoke(Int32 value);

    // Устаревшие асинхронные методы
    public virtual IAsyncResult BeginInvoke(Int32 value, AsyncCallback callback,
Object object);
    public virtual void EndInvoke(IAsyncResult result);
}
```

Generic delegates

По-большому счету обычно не нужно объявлять свои делегаты, достаточно стандартных generic вариантов

- Action<T> - не возвращает значение

```
public delegate void Action(); // Этот делегат не обобщенный
public delegate void Action<T>(T obj);
public delegate void Action<T1, T2>(T1 arg1, T2 arg2);
public delegate void Action<T1, T2, T3>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate void Action<T1, ..., T16>(T1 arg1, ..., T16 arg16);
```

- Func<T> - возвращает значение

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg);
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<T1, T2, T3, TResult>(T1 arg1, T2 arg2, T3 arg3);
...
public delegate TResult Func<T1,..., T16, TResult>(T1 arg1, ..., T16 arg16);
```

```
static void WriteSum(int x, int y) => Console.WriteLine("Sum: " + (x + y));
static void WriteMulpiply(int x, int y) => Console.WriteLine("Multiply: " + (x *
y));
static int Add(int x, int y) => x + y;

static void Operation(int x, int y, Action<int, int> op) => op(x, y);

static void Main(string[] args)
{
    Operation(10, 6, WriteSum); // Можно не создавать объект делегата
    Operation(3, 3, WriteMulpiply);
    Operation(7, 5, (x,y) => {Console.WriteLine(x - y);}); // lambda

    Action<int, int> op = WriteSum;
    op.Invoke(1, 1);

    Func<int, int, int> add = Add;
    int result = add(1,2);
    Console.WriteLine(result);
}
```

- Predicate<T> - в принципе тоже самое, что Func<T,bool>
- Делегат для проверки условия
- Может использоваться для фильтрации коллекций в linq
- Вообще он по сути Deprecated
- Всегда вместо него используйте Func<T,bool>

```
Predicate<int> isPositive = delegate (int x) { return x > 0; };
Predicate<Person> oscarFinder = (Person p) => { return p.Name == "Oscar"; };
```

Рекомендации:

- Всегда используйте Action / Func
- Их может не хватить при использовании out, ref параметров. По возможности можно не использовать out, ref параметры 😃

```
// Не надо так
delegate bool Tester(int i);

class AClass
{
    public Tester MyTester {get;set;}
}

// Надо так
class AClass
{
    public Func<int,bool> MyTester {get;set;}
}
```

Covariance & Contravariance

- Делегаты ковариантны для выходного значения
- и контрвариантны для параметров

```
static void Main(string[] args)
{
    Factory d;
    d = Create;          // ковариантность
    Base b = Create(3);
    b.Display();
}

delegate Base Factory(int value);
private static Derived Create(int value) => new Derived { I = value };

class Base
{
    public int I {get;set;}
    public void Display() => Console.WriteLine(I);
}
class Derived : Base {}
```

Для generic делегатов вариативность аналогична generic методам

- out - возвращаемый тип-параметр ковариантен
- in - параметр метода контрвариантен

```
Action<object> actObject = SetObject; // static void SetObject(object o) { }
Action<string> actString = actObject;

// Syntax:
public delegate void Action<in T>(T obj);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
```

Event

Рассмотрим пример кода:

```
public class Publisher
{
    public delegate void CallEveryOne();
    public CallEveryOne call = null;
}
public class Subscriber
{
    public Subscriber(Publisher p) { p.call += ShowMessage; }
    public void ShowMessage() => Console.WriteLine("S");
}

static void Main(string[] args)
{
    var p = new Publisher();
    var s = new Subscriber(p);
    p.call.Invoke();
}
```

Проблемы:

- либо надо давать подписчикам делегат полностью (но тогда они видят полную цепочку вызовов и могут запускать и изменять ее)
- либо они не имеют прав для контроля над действием

Решение:

- Event энкапсулирует multicast delegate.
- Реализует Observer паттерн

```
public class Publisher
{
    public delegate void CallEveryOne();
    public event CallEveryOne call;

    public void RunCall() => call();
}

public class Subscriber
{
    public Subscriber(Publisher p)
    {
```

```
            p.call += ShowMessage;
        }

        public void ShowMessage() => Console.WriteLine("Subscriber");
    }

    static void Main(string[] args)
    {
        var p = new Publisher();
        var s = new Subscriber(p);
        p.RunCall();
    }
```

Типичное использование обычных event:

```
button1.Click += new EventHandler(button1_Click);

...

///
/// Здесь button1_Click — обработчик нажатия
///
void button1_Click(Object sender, EventArgs e)
{
    // Действия после нажатия на кнопку
}
```

Пример с созданием собственного EventArgs msdn, msdn:

```
static void Main(string[] args)
{
    Counter c = new Counter(new Random().Next(10));
    c.ThresholdReached += c_ThresholdReached;
    while (true)
    {
        Console.WriteLine("adding one");
        c.Add(1);
    }
}

static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
{
    Console.WriteLine($"{e.Threshold} was reached at {e.TimeReached}.");
    Environment.Exit(0);
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

```
class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold)
    {
        threshold = passedThreshold;
    }

    public void Add(int x)
    {
        total += x;
        if (total < threshold)
            return;

        ThresholdReachedEventArgs args = new ThresholdReachedEventArgs
        {
            Threshold = threshold,
            TimeReached = DateTime.Now
        };
        OnThresholdReached(args);
    }
```

```
    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
    {
        ThresholdReached?.Invoke(this, e);
    }

    public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;
}
```

Определения для EventArgs

```
//
// Summary:
//     Represents the base class for classes that contain event data, and provides
a
//     value to use for events that do not include event data.
public class EventArgs
{
    //
    // Summary:
    //     Provides a value to use with events that do not have event data.
    public static readonly EventArgs Empty;

    //
    // Summary:
    //     Initializes a new instance of the System.EventArgs class.
    public EventArgs();
}
```

Определения для EventHandler

```
//
// Summary:
//     Represents the method that will handle an event when the event provides
data.
//
// Parameters:
//   sender:
//     The source of the event.
//
//   e:
//     An object that contains the event data.
//
// Type parameters:
//   TEventArgs:
//     The type of the event data generated by the event.
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Lambdas

- Обнаружив лямбда-выражение вместо делегата компилятор генерит private метод
- Этот метод называется анонимная функция
- Автоматически будет статическим или нет в зависимости от используемых переменных

```
Func<string> d1 = () => "test";
Func<int, string> d2 = (int x) => x.ToString();
Func<int, int, string> d3 = (int x, int y) => (x + y).ToString();

Func<int, string> d4 = x => x.ToString();
Func<int, int, string> d5 = (x, y) => (x + y).ToString();

delWithOut d6 = (out int x) => x = 5;
Func<int, int, string> d7 = (x, y) =>
    {
        int sum = x + y;
        return sum.ToString();
    };

delegate void delWithOut(out int z);
```

- Если код используется в нескольких местах - логично написать метод
- Если в одном - вполне можно использовать lambda
- Они должны упрощать код и ускорять разработку, код с ними не должен становиться сложнее
- У Рихтера есть правило - не использовать лямбда выражения для больше чем 3 строк кода (хотя это, конечно, спорно)

```
public double Benchmark(int times, Action func)
{
    var watch = new System.Diagnostics.Stopwatch();
    double totalTime = 0.0;

    for (int i = 0; i < times; i++)
    {
        watch.Start();
        func();
        watch.Stop();

        totalTime += watch.ElapsedTicks;
        watch.Reset();
    }

    return totalTime / (10000 * times);
}

double t = Benchmark(50, () => {var xs = Enumerable.Range(0, 1000000).ToList(); });
```

Closures

- Умеет использовать объекты вызывающего метода (локальные переменные и т.п.) в контексте анонимных функций и лямбд
- Jon Skeet, Тепляков, SOF

```
static Action CreateAction()
{
    int count = 0;
    return () =>
    {
        count++;
        Console.WriteLine($"Count = {count}");
    };
}

static void Main(string[] args)
{
    var action = CreateAction();
    action(); // 1
    action(); // 2
}
```

```
static Action CreateAction()
{
    DisplayClass1 c1 = new DisplayClass1();
    c1.count = 0;
    Action action = new Action(c1.ActionMethod);
    return action;
}

// "<>c__DisplayClass1".
private sealed class DisplayClass1
{
    // "<CreateAction>b__0".
    public void ActionMethod()
    {
        count++;
        Console.WriteLine("{0}. Count = {1}", message, count);
    }
    public int count;
}
```

- Переменная count из стека переехала в кучу в специально созданный объект
- Имена для класса и метода даются такие, чтобы мы не смогли их повторить
- Все это происходит за кадром и мы не влияем на это

- Что происходит с переменными?
- Происходит захват переменной на манер передачи ref параметра

```
string str = "Initial value";
Action action = () =>
{
    Console.WriteLine(str);
    str = "Modified by closure";
};
str = "After delegate creation";
action();
Console.WriteLine(str);
```

```
// After delegate creation
// Modified by closure
```

- С циклами могут возникнуть сложности:

```
var funcs = new List<Func<int>>();
for (int i = 0; i < 3; i++)
{
    funcs.Add(() => i);
}

foreach (var f in funcs)
    Console.WriteLine(f());
```

```
// 3
// 3
// 3
```

- Можно поправить:

```
var funcs = new List<Func<int>>();
for (int i = 0; i < 3; ++i)
{
    int tmp = i;
    funcs.Add(() => tmp);
}
foreach (var f in funcs)
    Console.WriteLine(f());
```