Делегаты

Делегат это объект

Делегат – это объект, который содержит в себе:

- адрес метода,
- тип параметров,
- тип возвращаемого значения.

Делегаты .NET могут указывать как на статические методы, так и на методы экземпляра.

Один делегат хранит в себе список методов и вызывает их в порядке следования.

```
delegate void D(int i);

class Program
{
    static void f(int i) { Console.WriteLine("fff"); }
    static void g(int i) { Console.WriteLine("ggg"); }

    static void Main(string[] args)
    {
        D d = new D(f);
        d += new D((new Program()).g);
        d(0);
    }
}
```

Сгенерированные методы

```
delegate bool MyDelegate(int x);
```

Делегат наследует библиотечный класс MulticastDelegate, и имеет все члены этого класса. Но среди них нет метода Invoke(), этот метод добавляет компилятор при генерации класса делегата.

```
public sealed class MyDelegate : System.MulticastDelegate
{
    public MyDelegate(object target, uint functionAddress);

    public bool Invoke(int x);

    public IAsyncResult BeginInvoke(int x, AsyncCallback cb, object state);

    public bool EndInvoke(IAsyncResult result);
}
```

Синхронные и асинхронные вызовы

Метод делегата Invoke() предназначен для синхронного вызова целевой функции делегата.

```
bool b = d.Invoke(3);                                   // синхр. вызов
```

Методы BeginInvoke() и EndInvoke() предназначены для вызова функции в отдельном потоке.

```
AsincResult ar = d.BeginInvoke(3, null, null);   // асинхр. вызов
 …
bool b = d.EndInvoke(ar);                               // синхр. вызов
```

Ковариантность делегатов

Архитектурный принцип подстановки:
Если класс D наследует класс B, то программа не перестанет работать, если мы всюду подставим вместо объектов B объекты D.

Ковариантность делегатов:
Если делегат возвращает базовый тип B, он может ссылаться не только на функции, которые возвращают B, но и на функции, которые возвращают производный тип D.

Иначе говоря, если мы вызываем метод через делегат и ожидаем получить B, то по принципу подстановки ничего страшного не случится, если мы получим не B, а D.

Почему ко- ?
по той же причине, что и в интерфейсах.

Обобщенные делегаты

Если тип параметра делегата – object, он может ссылаться на любые функции с одним параметром, но это не будет типобезопасным.

Больше безопасности дают обобщенные делегаты.

```
delegate T D<T>(T i);

class Program
{
    static int f(int i) { return i + 1; }
    static string g(string s) { return s + "1"; }

    static void Main(string[] args)
    {
        D<int>    d1 = f;
        D<string> d2 = g;
    }
}
```

События

Если мы хотим, чтобы объект X оповещал о своих изменениях другие объекты, мы должны сделать следующее.

1. Объявить тип делегата, и если надо, типы его параметров.
2. Создать в объекте X экземпляр делегата в виде закрытого поля.
3. Создать в объекте X отрытый метод для регистрации целевых функций в экземпляре делегата.

Все это инкапсулирует в себе член класса event – событие.

Подписка на событие

Класс X периодически издает событие Step, а класс Program слушает это событие.

```
class X
{
    public event EventHandler Step;

    public void Run()
    {
        for (int i = 1; i < 5; i++)
        {
            Thread.Sleep(500);
            if (Step != null)
            {
                Step(this, EventArgs.Empty);
            }
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        X x = new X();
        x.Step += x_Step;
        x.Run();
    }

    static void x_Step(object sender, EventArgs e)
    {
        Console.WriteLine("step");
    }
}
```

Шаблон события EventHandler: void объект_событие(object sender, EventArgs e)
Подписка на событие производится операцией += , отписка операцией -=.
Экземпляры делегатов разных типов – синглетоны.
Для событий не определена операция присвоения.

Обобщенный делегат EventHandler<T>

Если событие несет информацию, необходимо воспользоваться обобщенным делегатом и, если нужно, объявить тип второго аргумента.

```
class X
{
    public event EventHandler<int> Step;

    public void Run()
    {
        for (int i = 1; i < 5; i++)
        {
            Thread.Sleep(500);
            if (Step != null)
            {
                Step(this, i);
            }
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        X x = new X();
        x.Step += x_Step;
        x.Run();
    }

    static void x_Step(object sender, int e)
    {
        Console.WriteLine(e);
    }
}
```

Анонимные методы

```
class Program
{
    static void Main(string[] args)
    {
        X x = new X();

        x.Step += x_Step;

        x.Run();
    }

    static void x_Step(object sender, int e)
    {
        Console.WriteLine(e);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        X x = new X();

        x.Step += delegate(object sender, int e) {
            Console.WriteLine(e);
        };

        x.Run();
    }
}
```

Анонимные методы интересны тем, что могут обращаться к локальным переменным метода, в котором они определены.
Начиная с .NET 3.5, роль анонимных методов отошла к лямбда-выражениям.

```
class Program
{
    static void Main(string[] args)
    {
        X x = new X();

        x.Step += (s, e) => Console.WriteLine(e);

        x.Run();
    }
}
```

Самостоятельно

Объявить делегат, который ссылается на произвольную бинарную операцию над целыми числами, т.е. int Op(int x, int y). Создать объект делегата и вызвать его синхронно и асинхронно.

Самостоятельно

Уже объявлен класс Calc, в котором есть методы Plus и Minus.

```
class Calc
{
    protected double Plus(double a, double b) {…}
    protected double Minus(double a, double b) {…}
}
```

Объявлен также тип OpArgs:

```
class OpArgs: EventArgs  {  public double A, B;  }
```

Объявите класс Calc2, который наследует Calc и издает события Plused и Minused каждый раз, когда вызываются методы Plus и Minus (для этого перекройте методы Plus и Minus в производном классе). Тип событий Plused и Minused должен быть `EventHandler<OpArgs>`.