САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ

- **Сотавов Абакар Капланович**
- **Ассистент кафедры Информатики**(наб. канала Грибоедова, 30/32, ауд. 2038
- e-mail: sotavov@unecon.ru

- Материалы на сайте: http://de.unecon.ru/course/view.php?id=440

Делегаты

Определение делегата

Делегат — это вид класса, предназначенный для хранения ссылок на методы. Делегат, как и любой другой класс, можно передать в качестве параметра, а затем вызвать инкапсулированный в нем метод.

Делегаты используются для поддержки событий, а также как самостоятельная конструкция языка.

Описание делегата задает сигнатуру методов, которые могут быть вызваны с его помощью:

[атрибуты] [спецификаторы] delegate тип имя([параметры])

Пример описания делегата:

```
public delegate void D ( int i );
```

Базовым классом делегата является класс System.Delegate

Использование делегатов

Делегаты применяются в основном для следующих целей:

- получения возможности определять вызываемый метод не при компиляции, а динамически во время выполнения программы;
- обеспечения связи между объектами по типу «источник — наблюдатель»;
- создания универсальных методов, в которые можно передавать другие методы (поддержки механизма обратных вызовов).

Передача делегата через список параметров

```
namespace ConsoleApplication1 {
    public delegate double Fun( double x );   // объявление делегата
    class Class1 {
        public static void Table( Fun F, double x, double b )
        {   Console.WriteLine( " ----- X ----- Y -----" );
            while (x <= b)
            {  Console.WriteLine( "| {0,8} | {1,8} |", x, F(x));
               x += 1; }
        }
        public static double Simple( double x ) { return 1; }

        static void Main()
        { Table( Simple, 0, 3 );
          Table( Math.Sin, -2, 2 );
          Table( delegate (double x ){ return 1; }, 0, 3 );
}}}
```

определять вызываемый метод во время выполнения программы

```
namespace ConsoleApplication15
{
   delegate void Del(ref string s);            // объявление делегата
   class Program
   {
      public static void Metod1(ref string s)            // метод 1
      {          s = "Metod1 ";        }
      public static void Metod2(ref string s)            // метод 2
      {           s += "Metod2 ";       }
      static void Main(string[] args)
      {
         string s = "Методы:";
         Del d = new Del(Metod1);
         Console.WriteLine("Сколько Методов 1 или 2? ");
         if ((Console.ReadLine())=="2") d += new Del(Metod2);
         d(ref s);
         Console.WriteLine(s);
}   }}
```

«источник — наблюдатель»

```
public delegate void Del(object o);            // объявление делегата
   class Subj                                  // класс-источник
   {       Del dels;                   // объявление экземпляра делегата
      public void Register(Del d)              // регистрация делегата
      {          dels += d;       }
      public void OOPS()                       // что-то произошло
      {
         Console.WriteLine("ОЙ!");
         if (dels != null) dels(this);     // оповещение наблюдателей
      }
   }
   class ObsA                                  // класс-наблюдатель
   {
      public void Do(object o)          // реакция на событие источника
      {          Console.WriteLine("Бедняжка!");       }
   }
   class ObsB                                  // класс-наблюдатель
   {
      public static void See(object o)   // реакция на событие источника
      {          Console.WriteLine("Да ну, ерунда!");       }
   }
```

```
static void Main()
      {
         Subj s = new Subj();              //  объект класса-источника

         ObsA o1 = new ObsA();              //              объекты
         ObsA o2 = new ObsA();              //      класса-наблюдателя

         s.Register(new Del(o1.Do));      //      регистрация методов
         s.Register(new Del(o2.Do));      // наблюдателей в источнике
         s.Register(new Del(ObsB.See));   //  ( экземпляры делегата )

         s.OOPS();                        //    инициирование события
      }
   }
}
```

Операции с делегатами

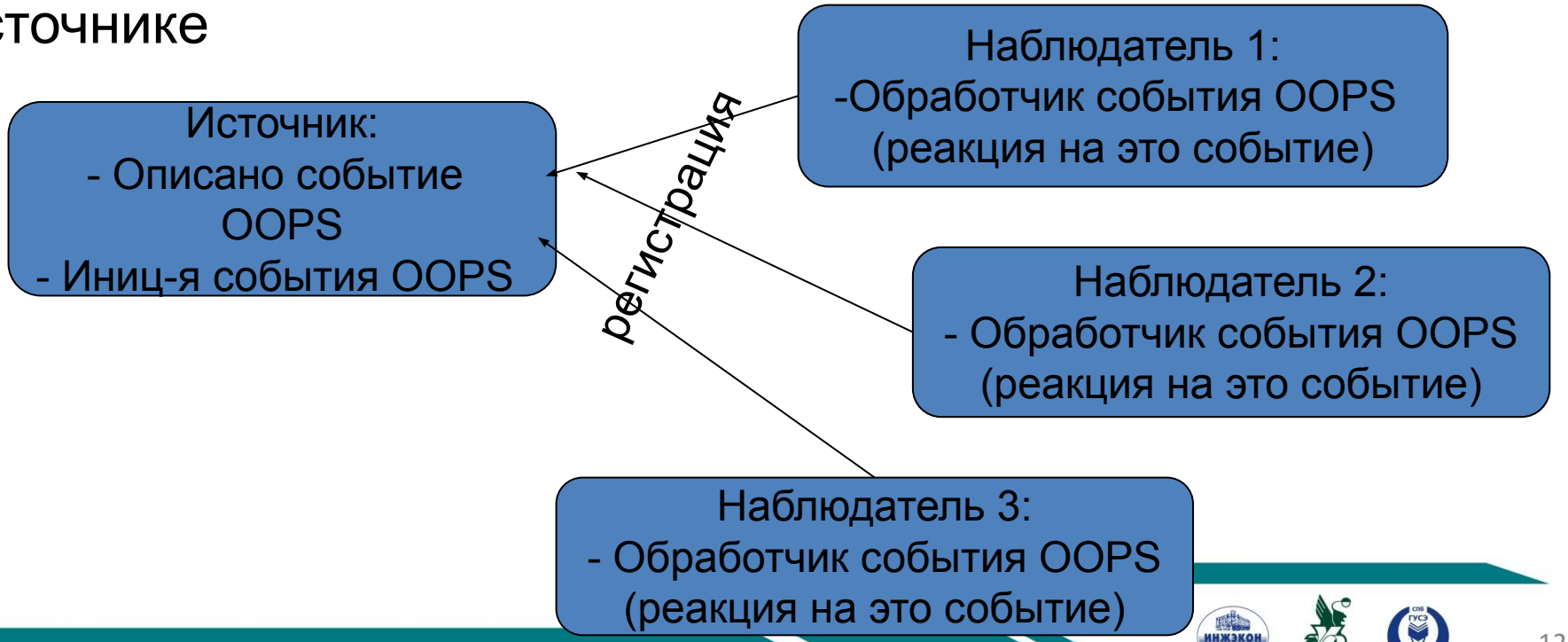
- Делегаты можно *сравнивать на равенство и неравенство*. Два делегата равны, если они оба не содержат ссылок на методы или если они содержат ссылки на одни и те же методы в одном и том же порядке.
- С делегатами одного типа можно *выполнять операции* простого и сложного *присваивания*.
- Делегат, как и строка string, является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.
- Использование делегата имеет тот же синтаксис, что и вызов метода. Если делегат хранит ссылки на несколько методов, они вызываются последовательно в том порядке, в котором были добавлены в делегат.

```
Del d1 = new Del(o1.Do);      // o1.Do
Del d2 = new Del(o2.Do);      // o2.Do
Del d3 = d1 + d2;             // o1.Do и o2.Do
d3 += d1;                     // o1.Do, o2.Do и o1.Do
d3 -= d2;                     // o1.Do и o1.Do
```

События

Определение события

- *Событие* — элемент класса, позволяющий ему посылать другим объектам (наблюдателям) уведомления об изменении своего состояния.
- Чтобы стать наблюдателем, объект должен иметь обработчик события и зарегистрировать его в объекте-источнике



Синтаксис события

[атрибуты] [спецификаторы] event тип имя_события

применяются *спецификаторы* new, public, protected, internal, private , static,virtual, sealed, override, abstract и extern

```
public delegate void Del();                    // объявление делегата

  class Subj                                   // класс-источник
  {
     public event Del Oops;                       // объявление события
     public void CryOops()                 // метод, инициирующий событие
     {
        Console.WriteLine( "ОЙ!" );
        if ( Oops != null ) Oops();
     }
  }

  class ObsA                                   // класс-наблюдатель
  {
     public void Do()                    // реакция на событие источника
     {          Console.WriteLine( "Бедняжка!" );       }
  }

  class ObsB                                   // класс-наблюдатель
  {
     public static void See()            // реакция на событие источника
     {          Console.WriteLine( "Да ну, ерунда!" );       }
  }
```

```
static void Main(string[] args)
    {

        Subj s = new Subj();            //   объект класса-источника

        ObsA o1 = new ObsA();           //              объекты
        ObsA o2 = new ObsA();           //       класса-наблюдателя

        s.Oops += new Del(o1.Do);       //              добавление
        s.Oops += new Del(o2.Do);       //              обработчиков
        s.Oops += new Del(ObsB.See);    //                  к событию

        s.CryOops();                    //     инициирование события


    }
```

Механизм событий

- События построены на основе делегатов: с помощью делегатов вызываются методы-обработчики событий. Поэтому *создание события* в классе состоит из следующих частей:
  - описание делегата, задающего сигнатуру обработчиков событий;
  - описание события;
  - описание метода (методов), инициирующих событие.
- Синтаксис события:

  [атрибуты] [спецификаторы] event тип имя

Пример

```
public delegate void Del( object o );     // объявление делегата
class A
{
    public event Del Oops;                 // объявление события
    ...
}
```

- *Обработка событий* выполняется в классах-получателях сообщения. Для этого в них описываются методы-обработчики событий, сигнатура которых соответствует типу делегата. Каждый объект (не класс!), желающий получать сообщение, должен зарегистрировать в объекте-отправителе этот метод.
- Событие — это удобная абстракция для программиста. На самом деле оно состоит из закрытого статического класса, в котором создается экземпляр делегата, и двух методов, предназначенных для добавления и удаления обработчика из списка этого делегата.
- Внешний код может работать с событиями единственным образом: добавлять обработчики в список или удалять их, поскольку вне класса могут использоваться только операции += и -=. Тип результата этих операций — void, в отличие от операций сложного присваивания для арифметических типов. Иного способа доступа к списку обработчиков нет.

библиотеке *.NET* описано огромное количество стандартных делегатов, предназначенных для реализации механизма обработки событий. Большинство этих классов оформлено по одним и тем же правилам:

имя делегата заканчивается суффиксом EventHandler ;

делегат получает два параметра:

- первый параметр задает источник события и имеет тип object ;
- второй параметр задает аргументы события и имеет тип EventArgs или производный от него.

можно обойтись стандартным классом делегата System.EventHandler

Имя обработчика события принято составлять из префикса On и имени события.

```
using System;
namespace ConsoleApplication1
{
    class Subj
    {
        public event EventHandler Oops;
        public void CryOops()
        {
            Console.WriteLine("ОЙ!");
            if (Oops != null) Oops(this, null);
        }
    }
    class ObsA
    {
        public void OnOops(object sender, EventArgs e)
        {           Console.WriteLine("Бедняжка!");       }
    }
```

```
class ObsB
   {
      public static void OnOops(object sender, EventArgs e)
      {          Console.WriteLine("Да ну, ерунда!");       }
   }
   class Class1
   {
      static void Main()
      {
         Subj s = new Subj();
         ObsA o1 = new ObsA();
         ObsA o2 = new ObsA();
         s.Oops += new EventHandler(o1.OnOops);
         s.Oops += new EventHandler(o2.OnOops);
         s.Oops += new EventHandler(ObsB.OnOops);

         s.CryOops();
      }   } }
```

Многопоточные приложения

пространстве имен **System.Threading**.

класс **Thread-** представляющий отдельный поток

Основные элементы класса Thread

Элемент	Вид	Описание
CurrentThread	Статическое свойство	Возвращает ссылку на выполняющийся поток (только для чтения)
Name	Свойство	Установка текстового имени потока
Priority	Свойство	Получить/установить приоритет потока (используются значения перечисленияThreadPriority)
ThreadState	Свойство	Возвращает состояние потока (используются значения перечисленияThreadState)
Abort	Метод	Генерирует исключение ThreadAbortException. Вызов этого метода обычно завершает работу потока
Sleep	Статический метод	Приостанавливает выполнение текущего потока на заданное количество миллисекунд
Interrupt	Метод	Прерывает работу текущего потока
Join	Метод	Блокирует вызывающий поток до завершения другого потока или указанного промежутка времени и завершает поток
Resume	Метод	Возобновляет работу после приостановки потока
Start	Метод	Начинает выполнение потока, определенного делегатом ThreadStart
Suspend	Метод	Приостанавливает выполнение потока. Если выполнение потока уже приостановлено, то игнорируется

Листинг

```
using System.Threading;
.....................
static void Main(string[] args)
{
    // получаем текущий поток
    Thread t = Thread.CurrentThread;

    //получаем имя потока
    Console.WriteLine("Имя потока: {0}", t.Name);
    t.Name = "Метод Main";
    Console.WriteLine("Имя потока: {0}", t.Name);

    Console.WriteLine("Запущен ли поток: {0}", t.IsAlive);
    Console.WriteLine("Приоритет потока: {0}", t.Priority);
    Console.WriteLine("Статус потока: {0}", t.ThreadState);

    // получаем домен приложения
    Console.WriteLine("Домен приложения: {0}«,
Thread.GetDomain().FriendlyName);

    Console.ReadLine();
```

```
Имя потока:
Имя потока: Метод Main
Запущен ли поток: True
Приоритет потока: Normal
Статус потока: Running Домен
приложения: ThreadApp.vshost.exe
```

Статус потока

Статусы потока содержатся в перечислении **ThreadState**:

Название	Описание
Aborted	поток остановлен, но пока еще окончательно не завершен
AbortRequested	для потока вызван метод Abort, но остановка потока еще не произошла
Background	поток выполняется в фоновом режиме
Running	поток запущен и работает (не приостановлен)
Stopped	поток завершен
StopRequested	поток получил запрос на остановку
Suspended	поток приостановлен
SuspendRequested	поток получил запрос на приостановку
Unstarted	поток еще не был запущен
WaitSleepJoin	поток заблокирован в результате действия методов Sleep или Join

Приоритеты потоков

Приоритеты поток располагаются в перечислении **ThreadPriority**:

Название	Описание
Lowest	Выполнение потока Thread может быть запланировано после выполнения потоков с любыми другими приоритетами.
BelowNormal	Выполнение потока Thread может быть запланировано после выполнения потоков с приоритетом Normal и до потоков с приоритетом Lowest.
Normal	Выполнение потока Thread может быть запланировано после выполнения потоков с приоритетом AboveNormal и до потоков с приоритетом BelowNormal. По умолчанию потоки имеют приоритет Normal.
AboveNormal	Выполнение потока Thread может быть запланировано после выполнения потоков с приоритетом Highest и до потоков с приоритетом Normal.
Highest	Выполнение потока Thread может быть запланировано до выполнения потоков с любыми другими приоритетами.

Листинг

ConsoleApplication6.zip

```
class PriorityTest
{
    bool loopSwitch;
    public PriorityTest()
    {
        loopSwitch = true;
    }
    public bool LoopSwitch
    {
        set { loopSwitch = value; }
    }
    public void ThreadMethod()
    {
        long threadCount = 0;
        while (loopSwitch)
        {
            threadCount++;
        }
        Console.WriteLine("{0} with {1,11} priority " +"has a count = {2,13}", Thread.CurrentThread.Name,
Thread.CurrentThread.Priority.ToString(),  threadCount.ToString());
    }
}
```

```
static void Main()
  {
      PriorityTest priorityTest = new PriorityTest();
      ThreadStart startDelegate =
new ThreadStart(priorityTest.ThreadMethod);
      Thread threadOne = new Thread(startDelegate);
      threadOne.Name = "ThreadOne";
      Thread threadTwo = new Thread(startDelegate);
      threadTwo.Name = "ThreadTwo";
      threadTwo.Priority = ThreadPriority.BelowNormal;
      threadOne.Start();
      threadTwo.Start();
      Thread.Sleep(2000);
      priorityTest.LoopSwitch = false;
      Console.ReadLine();
  }
```

Делегат ThreadStart

```
public static void Count()
    {
        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Втор
ой поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(400);
        }
    }
```

ConsoleApplication7.zip

```
static void Main(string[] args)
    {
        // создаем новый поток
        Thread myThread = new Thread(new
ThreadStart(Count));
        myThread.Start(); // запускаем поток

        for (int i = 1; i < 9; i++)
        {
            Console.WriteLine("Главный
поток:");
            Console.WriteLine(i * i);
            Thread.Sleep(300);
        }
         Console.ReadLine();
    }
```

Потоки с параметрами и ParameterizedThreadStart

```
public static void Count(object x)
      {
         for (int i = 1; i < 9; i++)
         {
            int n = (int)x;
         Console.WriteLine("Второй
поток:");
            Console.WriteLine(i * n);
            Thread.Sleep(400);
         }
      }
```

ConsoleApplication8.zip

```
static void Main(string[] args)
   {
      int number = 4;
      // создаем новый поток
      Thread myThread = new Thread(new
ParameterizedThreadStart(Count));
      myThread.Start(number);

      for (int i = 1; i < 9; i++)
      {
         Console.WriteLine("Главный поток:");
         Console.WriteLine(i * i);
         Thread.Sleep(300);
      }

      Console.ReadLine();
   }
```

Параллельное программирование и библиотека TPL

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен **System.Threading.Tasks**. Данный класс описывает отдельную задачу, которая запускается в отдельном потоке

Листинг

```
static void Display()
        {
            Console.WriteLine("Начало работы метода
Display");
            // имитация работы метода
            Thread.Sleep(3000);
            Console.WriteLine("Завершение работы метода
Display");
        }
```

ConsoleApplication9.zip

```
static void Main(string[] args)
        {
            Task task = new Task(Display);

            task.Start();
            Console.WriteLine("Выполняется работа метода
Main");
            task.Wait();
            Console.ReadLine();
        }
```

Параллельные задачи с параметрами

```
static void DisplayMessage(string message)
        {
            Console.WriteLine("Сообщение: {0}", message);
            Console.WriteLine("Id задачи: {0}", Task.CurrentId);
        }
```

ConsoleApplication10.zip

```
static void Main(string[] args)
        {
            Task task1 = new Task(() => DisplayMessage("вызов метода с параметрами"));
            task1.Start();
            Console.ReadLine();

        }
```

Возвращение результатов из задач

```
static int Factorial(int x)
        {
            int result = 1;
            for (int i = 1; i <= x; i++)
            {
                result *= I;
            }
            return result;
        }
```

ConsoleApplication11.zip

```
static void Main(string[] args)
        {
            Task<int> task1 = new Task<int>(() => Factorial(5));
            task1.Start();
            Console.WriteLine("Факториал числа 5 равен {0}", task1.Result);
            Console.ReadLine();

        }
```

Асинхронное программирование

Асинхронные делегаты

Синхронно

```
static int Display()
        {
            Console.WriteLine("Начинается работа метода
Display....");
            int result = 0;
            for (int i = 1; i < 10; i++)
            {       result += i * I;              }
            Thread.Sleep(3000);
            Console.WriteLine("Завершается рабо
Display....");
            return result;
        }
public delegate int DisplayHandler();
        static void Main(string[] args)
        {
            DisplayHandler handler = new
DisplayHandler(Display);
            int result = handler.Invoke();

            Console.WriteLine("Продолжается работа метода
Main");
            Console.WriteLine("Результат равен {0}", result);

            Console.ReadLine();
```

ConsoleApplication12.zip

```
Начинается работа метода Display....
Завершается работа метода Display....
Продолжается работа метода Main
Результат равен 285
```

асинхронный вызовов делегата

```
static int Display()
        {
            Console.WriteLine("Начинается работа метода Display....");

            int result = 0;
            for (int i = 1; i < 10; i++)
            {
                result += i * i;
            }
            Thread.Sleep(3000);
            Console.WriteLine("Завершается работа метода D
            return result;
        }
```

ConsoleApplication13.zip

```
Начинается работа метода Display....
Продолжается работа метода Main
Завершается работа метода Display....
Результат равен 285
```

```
public delegate int DisplayHandler();
        static void Main(string[] args)
        {
            DisplayHandler handler = new DisplayHandler(Display);
             IAsyncResult resultObj = handler.BeginInvoke(null, null);
             Console.WriteLine("Продолжается работа метода Main");
            int result = handler.EndInvoke(resultObj);
            Console.WriteLine("Результат равен {0}", result);
             Console.ReadLine();
        }
```

Начиная с .NET 4.5 была изменена концепция создания асинхронных вызовов.
Во фреймворк были добавлены два новых ключевых слова async и await, цель которых - упростить написание асинхронного кода.
Ключевое слово async указывает, что метод или лямбда-выражение будет выполняться асинхронно. А оператор await позволяет остановить текущий поток, пока не завершится работа метода, помеченного как async.

Листинг

```
static Task<int> Factorial(int
x)
      {
          int result = 1;
          return Task.Run(() =>
          {
              for (int i = 1; i <= x;
i++)
              {
                  result *= i;
              }
              return result;
          });
```

```
static async Task DisplayResultAsync()
        {
            int num = 5;

            int result = await Factorial(num);
            Thread.Sleep(3000);
            Console.WriteLine("Факториал числа {0} равен {1}", num,
result);
        }
```

```
static void Main(string[] args)
{
    Task t = DisplayResultAsync();
    t.Wait();
    Console.ReadLine();
}
```

ConsoleApplication14.zip

СПАСИБО ЗА ВНИМАНИЕ!

ИНЖЭКОН ENGECON

ФИНЭК
Санкт-Петербург

СПб ГУСЭ

ОБЪЕДИНЯЯ ЛУЧШЕЕ