Делегаты

Назначение делегатов – представлять в программе методы(функции). Делегаты используются в .NET механизмом обработки событий.

Этапы применения делегатов:

- определение (объявление) делегата как типа;
- создание экземпляра делегата;
- вызов делегата.

Синтаксис определения (объявления) делегата:
модификаторы delegate тип результата имя_делегата(спецификация _ параметров);
Необязательные **модификаторы** объявления делегата:
new, public, protected, internal, private.
Делегат – тип может быть объявлен как локальный тип класса или структуры либо как декларация верхнего уровня в единице компиляции.
Модификатор **new** применим только для делегатов, локализованных в классе или структуре.
Примеры определений типов делегатов:

```
public delegate int[] Row(int num);
public delegate void Print(int [] ar);
```

Делегат – это тип ссылок.

Используя введенные делегаты, можно так определить ссылки:

Row delRow;

Print delPrint;

После такого определения ссылки **delRow** и **delPrint** имеют значение **null**.

С каждой из этих ссылок можно связать экземпляр соответствующего типа делегатов. Экземпляр делегата создается конструктором делегата.

Конструктор в определение делегата компилятор встраивает автоматически.

При обращении к конструктору в качестве аргумента можно использовать:

метод класса (имя метода, уточненное именем класса);

метод объекта (имя метода, уточненное именем объекта);

ссылку на уже существующий в программе экземпляр делегата.

Экземпляр делегата может представлять как метод класса, так и метод объекта. Однако в обоих случаях метод должен соответствовать по типу возвращаемого значения и по спецификации параметров объявлению делегата.
Определение типа делегатов может размещаться в пространстве имен наряду с другими объявлениями классов и структур. В этом случае говорят о внешнем размещении определения делегата. Кроме того, определения делегатов могут входить в объявления классов и структур. В этом случае имеет место локализация делегата.

Делегат в C# представляет собой еще одну разновидность класса: будучи типом, он определяет сигнатуру функции.

Пример Внешнее размещение определения делегата

```
using System;
public delegate int [] Row(int num);
public delegate void Print (int [] ar);
public class Example
{
    static public int[] series(int num)
    {
        int arLen = (int)Math.Log10(num) + 1;
        int [] res = new int[arLen];
        for (int i = arLen - 1;i >= 0; i--)
        {

            res[i] = num % 10;
            num /= 10;
        }
        return res;
    }
    static public void display(int[] ar)
    {
        for (int i = 0; i < ar.Length;i++)
            Console.Write("{0}\t" , ar[i]);
        Console.WriteLine();
    }
}
```

```
class Program
    {
        static void Main()
        {
            Row delRow;
            Print delPrint;

            delRow = new Row(Example.series);
            delPrint = new Print(Example.display);

            int[] myAr = delRow(13579);
            delPrint(myAr);

            int[] newAR = { 11, 22, 33, 44, 55,65 };
            delPrint(newAR);

            Example.display(myAr);
        }
    }
```

Результат выполнения программы:

1 3 5 7 9

11 22 33 44 55 65

1 3 5 7 9// явное обращение к методу Exampl.display().

В CTS есть абстрактные классы System.Delegate и System. MulticastDelegate . Любой реальный делегат является наследником System. MulticastDelegate, но программист не может создать наследника этого класса традиционным для ООП способом.

Каждое определение делегата – типа вводит новый тип ссылок, производный от единого базового класса System.Delegate. От этого базового класса каждый делегат – тип наследует конструктор и ряд полезных методов и свойств. Среди них отметим:

public MethodInfo Method{get;} - свойство, представляющее заголовок метода, на который в данный момент “настроен” экземпляр делегата.

public object Target {get;} – свойство, позволяющее определить, какому классу принадлежит тот объект, метод которого представляет экземпляр делегата. Если значение этого свойства есть null, то экземпляр делегата представляет в этот момент статический метод класса.

Добавим в приведенную программу операторы:

Console.WriteLine(“delRow is equals {0}.”, delRow.Method);

Console.WriteLine(“delPrint is equals {0}.”, delPrint.Method);

Результат выполнения этих операторов:

delRow is equals Int32[] series(Int32).

delPrint is equals Void display(Int32[]).

В следующем примере с помощью делегатов реализуется сортировка списка книг.

Каждый делегат сортирует книги по какому-то одному из четырех признаков.

```
using System;
using System.Collections;
namespace SortBooksDelegatesDemo
{
    class Book              // Содержит описание одной книги
    {
        // Поля класса:
        internal string Title, Author, Publish;
        internal int Pages;
        // Конструкторы класса:
        public Book(){}
        public Book(string aTitle, string aAuthor, string aPublish,
                                                          int aPages)
        {
            Title = aTitle;
            Author = aAuthor;
            Publish = aPublish;
            Pages = aPages;
        }
    }
```

```
class Books                    // Контейнер для множества книг
    {
        // Хранилище книг:
        public static ArrayList theBooks = new ArrayList();
        // Печать списка книг:
        public static void PrintBooks(string s)
        {
            Console.WriteLine("\n" + s);
            for (int i = 0; i < theBooks.Count; i++)
            {
                Console.WriteLine("{0}. \"{1}\". {2}. {3,4}c.",
                    (theBooks[i] as Book).Author,
                    (theBooks[i] as Book).Title,
                    (theBooks[i] as Book).Publish,
                    (theBooks[i] as Book).Pages.ToString());
            }
        }
```

```
// Объявление делегата. Каждый экземпляр делегата сравнивает
        // книги по какому-то одному полю:
        public delegate int SortItems(Book B1, Book B2);

        // Методы сравнения имеют такую же сигнатуру, как и
делегат:
        public static int SortByTitle(Book B1, Book B2)
        {
            return string.Compare(B1.Title, B2.Title);
        }
        public static int SortByAuthor(Book B1, Book B2)
        {
            return string.Compare(B1.Author, B2.Author);
        }
        public static int SortByPublish(Book B1, Book B2)
        {
            return string.Compare(B1.Publish, B2.Publish);
        }
        public static int SortByPages(Book B1, Book B2)
        // Для типа int не определен метод Compare
        {
            return B1.Pages - B2.Pages;
        }
```

```
// Метод SortBooks сортирует книги методом «всплывающего пузырька».
        // Критерий сортировки определяется экземпляром делегата SortItems
        public static void SortBooks(SortItems Compare)
        {
            bool IsSorted;              // Признак окончания сортировки
            do
            {
                IsSorted = true;        // Считаем, что сортировка закончилась
                // Просмотр книг:
                for (int i = 0; i < theBooks.Count - 1; i++)
                {
                    if (Compare(theBooks[i] as Book,
                                theBooks[i + 1] as Book) > 0)
                    {    // Сортировка нарушена. Меняем книги местами
                        Book temp = new Book();
                        temp = theBooks[i] as Book;
                        theBooks[i] = theBooks[i + 1];
                        theBooks[i + 1] = temp;
                        IsSorted = false;// Требуется новый проход
                    }
                }
            }
            while (IsSorted != true);

        }
```

```
public static void Main()
        {
        // Наполняем список книг:
            theBooks.Add(new Book("C# и платформа .NET",
                                  "Троэлсен", "Питер", 796));
            theBooks.Add(new Book("Visual Studio .NET 2003",
                                  "Гарнаев", "БХВ-Петербург", 688));
            theBooks.Add(new Book("Разработка приложений на C++ и C#",
                                  "Секунов", "Питер", 608));
            PrintBooks("Исходный набор:");
            SortItems SI = new SortItems(SortByTitle); // Экземпляр делегата
            SortBooks(SI);                             // Сортируем
            PrintBooks("Сортировка по названию:");     // Печатаем список

            SI = new SortItems(SortByAuthor);
            SortBooks(SI);
            PrintBooks("Сортировка по автору:");
            SI = new SortItems(SortByPublish);
            SortBooks(SI);
            PrintBooks("Сортировка по издательству:");
            SI = new SortItems(SortByPages);
            SortBooks(SI);
            PrintBooks("Сортировка по кол-ву страниц:");
            Console.ReadLine();
        }
    }
}
```

Использование методов экземпляров в качестве делегатов

```
using System;

delegate string StrMod( string str);

class StringOps
{
    //замена пробелов дефисами
    public string replaceSpases(string a)
    {
        Console.WriteLine("Замена пробелов дефисами");
        return a.Replace(" ", "-");
    }

    // Символы строки в обратном порядке
    public string reverse(string a)
    {
        string temp = "";
        int i, j;
        Console.WriteLine("Символы строки в обратном порядке");
        for (j = 0, i = a.Length - 1; i >= 0; i--, j++)
            temp += a[i];
        return temp;
    }
}
```

```
class DelegateTest
{
    public static void Main()
    {
        StringOps so = new StringOps();// Создание
экземпляра StringOps
        // Инициализация делегата
        StrMod strOp = so.replaceSpases;
        string str;
        //вызов метода с помощью делегата
        str = strOp("это тест");
        Console.WriteLine("Результирующая строка  " +
str);
        Console.WriteLine();

        strOp = so.reverse;
        str = strOp("это тест 2");
        Console.WriteLine("Результирующая строка  "+ str);
        Console.WriteLine();
    }
}
```

Массивы делегатов

Делегаты, точнее ссылки на экземпляры делегатов можно объединять в массивы.

Такая возможность позволяет программисту задавать наборы действий, которые затем будут автоматически выполнены в указанной последовательности.

Пример.

Рассмотрим модель перемещения робота по плоской поверхности. Пусть робот умеет выполнять четыре команды: вперед, назад, направо, налево. Каждая команда изменяет его положение на один шаг.

Система управления роботом должна формировать последовательность команд, которые робот выполняет автоматически. После выполнения полученной последовательности команд робот должен сообщить о достигнутом местоположении.

```
class Robot
{
    int x, y;
    public void right() { x++; }
    public void left() { x--; }
    public void forward() { y++; }
    public void backward() { y--; }
    public void position()
    { Console.WriteLine("TheRobot position: x = {0}, y = {1}", x, y); }
 }

    delegate void Steps();
    class Program
    {
        static void Main()
        {
            Robot rob = new Robot();
            Steps[] trace = { new Steps(rob.backward),
                              new Steps(rob.backward),new
Steps(rob.left)};
            for (int i = 0; i < trace.Length; i++)
            {
                Console.WriteLine("Method = {0},Target = {1}",
                    trace[i].Method, trace[i].Target);
                trace[i]();
            }
            rob.position();
        }
    }
```

Результаты выполнения программы

```
Method = Void backward(), Target = Robot
Method = Void backward(), Target = Robot
Method = Void left(),Target = Robot
The Robot position: x = -1,y = -2
```

Анонимные методы

Анонимный метод – это, по существу блок кода, который передается делегату.

```
using System;
// Объявление делегата
delegate void Countit();
class AnonMethDemo  {
    public static void Main() {
        //Код реализации  счета передается как анонимный метод
        Countit count = delegate  {
            for (int i = 0; i < 5; i++)
                Console.WriteLine(i);
        };
        count();
    }
}
```

Передача аргументов анонимному методу.

```
using System;
// Объявление делегата
delegate void Countit(int k);
class AnonMethDemo  {
    public static void Main() {
        //Код реализации  счета передается как
анонимный метод
        Countit count = delegate (int k)  {
            for (int i = 0; i <= k; i++)
                Console.WriteLine(i);
        };
        count(5);
    }
}
```

Возвращение значения из анонимного метода

```
using System;
// Объявление делегата
delegate int Countit(int k);
class AnonMethDemo  {
    public static void Main() {
        int result;

        //Код реализации  счета передается как анонимный
метод
        Countit count = delegate (int k)  {
            int sum = 0;
            for (int i = 0; i <= k; i++)
            {
                Console.WriteLine(i);
                sum += i;
            }
            return sum;
        };
        result = count(5);
```

Многоадресные групповые экземпляры делегатов

Делегат (его экземпляр) может содержать ссылки сразу на несколько методов, соответствующих типу делегата. При однократном обращении к такому экземпляру делегата в этом случае автоматически организуется последовательный вызов всех методов, которые он представляет.

Для поддержки такой многоадресности используются методы, которые каждый делегат наследует от базового класса MulticastDelegate, а он в свою очередь является наследником класса System.Delegate.

Основные методы

public static Delegate Combine(Delegate a, Delegate b) – метод объединяет(группирует) два экземпляра делегата, создавая двухадресный экземпляр делегата. Аргументами при обращении должны быть ссылки на экземпляры делегатов. Обращение к этому варианту метода Combine для упрощения можно заменять операцией «+=» или последовательностью операций «+» и «=».
многоадресный_ **делегат += ссылка на делегат;**

public static Delegate Combine (params Delegate [ ], delegates) – метод объединяет (группирует) несколько экземпляров делегатов. Аргументы – список ссылок на экземпляры делегатов. Возвращаемое значение –

public virtual Delegate[ ] GetInvocationList() метод возвращает массив экземпляров делегатов , объединенных (сгруппированных) в конкретном многоадресном экземпляре делегата, к которому применен данный метод. Метод автоматически переопределяется при объявлении каждого типа делегатов. Применим к конкретным экземплярам делегатов. Возвращаемое значение – массив экземпляров делегатов(ссылок на них), представляющих все методы, которые адресует экземпляр делегата.

public static Delegate Remove(Delegate source, Delegate value) – метод удаляет из многоадресного экземпляра делегата, заданного первым параметром, конкретную ссылку, заданную вторым параметром – делегатом. Аргументами при обращении должны быть две ссылки на экземпляры – делегаты. Первый аргумент представляет многоадресный экземпляр делегата, второй представляет экземпляр делегата, значение которого (ссылку на метод) нужно удалить из многоадресного экземпляра делегата. Обращение к методу Remove для упрощения можно заменять операцией «-=» или последовательностью операций «-» и «=».

Многоадресный_ делегат -= ссылка_на_делегат;

Применение многоадресных экземпляров делегатов позволяет в рассмотренном выше примере расширить систему команд управления роботом, не изменяя его внутренней структуры.

```
using System;

delegate void Steps();

class Robot
{
    int x, y;
    public void right() { x++; }
    public void left() { x--; }
    public void forward() { y++; }
    public void backward() { y--; }
    public void position()
    { Console.WriteLine("TheRobot position: x = {0}, y =
{1}", x, y); }
```

```
class Program
    {
        static void Main()
        {
            Robot rob = new Robot();
            Steps delR = new Steps(rob.right);
            Steps delL = new Steps(rob.left);
            Steps delF = new Steps(rob.forward);
            Steps delB = new Steps(rob.backward);
            Steps delRF = delR + delF;
            Steps delRB = delR + delB;
            Steps delLF = delL + delF;
            Steps delLB = delL + delB;
            delLB();
            rob.position();
            delRB();
            rob.position();
        }
    }
}
```

Если любой из делегатов цепочки пошлет исключение, то обработка цепочки прерывается и выполняется поиск подходящего обработчика исключений.
Если делегаты цепочки принимают параметры по ссылке, то изменения таких параметров за счет операторов тела метода, вызванного через делегат, воспринимаются следующими делегатами цепочки.
Для корректного обращения к многоадресному делегату рекомендуется выполнять приведение типов.

Имеется возможность вызывать делегаты цепочки в произвольном порядке и получать возвращаемые значения каждого из них.
Для этого используется нестатический метод **GetInvocationList(**), он возвращает массив делегатов **Delegate[ ]**, входящих в тот многоадресный делегат, к которому метод применен.

Делегаты и обратные вызовы.

Обратным вызовом называют обращение из исполняемой функции к другой функции, которая зачастую определяется не на этапе компиляции, а в процессе выполнения программы.

В языках C и C++ для реализации обратных вызовов в качестве параметра в функцию передается указатель на ту функцию, к которой нужно обращаться при исполнении программы.

Опасность и ненадежность обращения к функции по ее адресу состоит в том, что, зная только адрес, невозможно проверить правильность обращения, так как необходимо выполнить достаточно большой объем проверок. Нужно убедиться, что количество аргументов и их типы соответствуют параметрам функции, что верен тип возвращаемого значения и т.д. Возможность указанных проверок в языке C# обеспечивают делегаты.

При разработке средств обратного вызова в языке C# было решено обеспечить программистов не только возможностью контролировать сигнатуру методов, но и отличать методы классов от методов объектов. Каждый делегат включает в качестве полей ссылку на объект, для которого нужно вызвать метод и имя конкретного метода. Если ссылка на объект равна null, то имя метода воспринимается как имя статического метода.

Пример. Использование ссылки на экземпляр делегата в качестве параметра для организации обратных вызовов.

```
using System;
public delegate double Proc(double x);
public class Series
{
    int n;
    double xmi, xma;
    public Series(int ni, double xn, double xk)
    {n = ni; xmi = xn; xma = xk;}
    public void display(Proc fun)
    {
        Console.WriteLine("Proc:{0},xmi =
{1:f2};xma = {2:f2},n = {3}.", fun.Method,
                xmi, xma, n);
        for (double x = xmi; x <= xma; x += (xma
- xmi) / (n - 1))
                Console.Write("{0:f2\t", fun(xmi));
    }
}
```

```
class Program
    {
        static double mySin(double x)
       { return Math.Sin(x);}
        static double myLine(double x)
        {return x*5;}
        static void Main(string[] args)
        {
            Series sr = new Series(7, 0.0, 1.0);
            sr.display(mySin);
                Console. WriteLine();

            sr.display (myLine);
            Console.WriteLine();
        }
    }
```

Ковариантность и контрвариантность

Ковариантность позволяет присвоить делегату метод, когда возвращаемый тип метода является классом, производным от класса, который определен возвращаемым типом делегата.

Контрвариантность позволяет присвоить делегату метод, когда тип параметра метода является базовым классом для класса, который определен декларацией делегата

```
using System;
namespace Demo
{
    class X
    {
        public int val;
    }
    class Y : X{
        delegate X Changelt(Y obj);
    class CoContraVariance
    {
            static X incrA(X obj)
            {
                X temp = new X();
                temp.val = obj.val + 1;
                return temp;
            }
            static Y incrB(Y obj)
            {
                Y temp = new Y();
                temp.val = obj.val + 2;
                return temp;
            }
```

```
static void Main(string[] args)
            {
                Y Yob = new Y();
                // благодаря контрвариантности
следующая строка кода верна
                Changelt change = incrA;
                X Xob = change(Yob);
                Console.WriteLine("Xob:" +
Xob.val);
                // благодаря ковариантности
следующая строка кода верна
                change = incrB;
                Yob = (Y)change(Yob);
                Console.WriteLine("Yob:" +
Yob.val);
            }
        }
        }
    }
```