C# делегаты

Делегат это объект, который может ссылаться на метод.

Когда создается делегат, то создается объект, содержащий ссылку на метод. Делегат позволяет вызвать метод, на который он ссылается.

Один и тот же делегат можно использовать для вызова разных методов при выполнении программы. Для этого надо изменить метод, на который ссылается делегат.

Метод вызываемый делегатом определяется на этапе выполнения программы, а не на этапе компиляции. В этой гибкости и заключается преимущества использования делегатов.

Формализм объявления делегата:

delegate возвращаемый_тип имя(список_параметров);

Список_параметров называется сигнатурой.

Возвращаемый_тип – это тип возвращаемый методами, которые будут вызываться этим делегатом.

C# делегаты

Сигнатура – параметры, необходимые для методов вызываемых этим делегатом.

Как только будет создан экземпляр делегата, он может вызывать и ссылаться на те методы возвращаемый тип и параметры которых, соответствуют указанным в объявлении.

Вызываемые методы могут быть методами уже существующих объектов или статическими методами классов. **Главное, чтобы их сигнатура и возвращаемый тип совпадали с объявленными в делегате.**

```
using System;
namespace Mydelegate {
   delegate string StrMod( string str);
  class Program {
       // замена пробелов дефисами
       static string ReplaceSpaces(string s) {
           Console.WriteLine("Замена пробелов дефисами");
           return s.Replace(' ',  '-');
       }
```

C# делегаты

```
// удаление пробелов
        static string RemoveSpaces(string s) {
          string temp = "";
          Console.WriteLine("Удаление пробелов");
          for (int  i = 0; i < s.Length; i++) {
              if (s[i] != ' ')
                  temp += s[i];
          }
          return temp;
      }
      //Обращение строки
      static string Reverse(string s) {
          string temp = "";
          Console.WriteLine("Обращение строки");
          for (int j = 0, int i = s.Length -1; i >= 0; i--, j++) {
              temp += s[i];
          }
           return temp;
      }
```

C# делегаты

```
static void Main(void) {
  string str;
  StrMod strOp = new StrMod(ReplaceSpaces); // Конструируем делегат
 //Вызываем метод с помощью экземпляра делегата
    str = strOp(“Это простой тест”);
     Console.WriteLine(" Результат : " + str);
      Console.ReadKey();

     StrMod strOp = new StrMod(RemoveSpaces); // Конструируем делегат
      str = strOp(“Это простой тест”);
     Console.WriteLine(" Результат : " + str);
      Console.ReadKey();

     StrMod strOp = new StrMod(Reverse); // Конструируем делегат
      str = strOp(“Это простой тест”);
     Console.WriteLine(" Результат : " + str);
      Console.ReadKey();
  }
}
```

C# делегаты

Можно использовать т.н. групповое преобразование методов, позволяющее присвоить имя метода делегату, не используя оператор new или явны вызов конструктора делегата.

При этом упрощается синтаксис

Для предыдущего примера можно делать так

```
// имя метода прямо присваивается экземпляру делегата
  StrMod  strOp = ReplaceSpaces;
   . . . . . .. . . . . .
    strOp = RemoveSpaces;
 . . . . . .. . . . . .
    strOp = Reverse;
 . . . . . .. . . . . .
```

Делегат может сослаться не только на статические методы, но и на методы любого объекта. Для этого надо чтобы этот объект существовал.

C# делегаты

```
using System;
namespace Mydelegate {
    delegate string StrMod(string str);
  class StringOps {
        string ReplaceSpaces(string s) {
            Console.WriteLine("Замена пробелов дефисами");
            return s.Replace(' ',  '-');
        }
        string RemoveSpaces(string s) {
            string temp = "";
            Console.WriteLine("Удаление пробелов");
            for (int  i = 0; i < s.Length; i++) {
                if (s[i] != ' ')
                    temp += s[i];
            }
            return temp;
        }
```

C# делегаты

```
   string Reverse(string s) {
          string temp = "";
          Console.WriteLine("Обращение строки");
          for (int j = 0, int i = s.Length -1; i >= 0; i--, j++) {
              temp += s[i];
          }
           return temp;  }
    }
  class Program {
        static void Main () {
      StringOps so = new new StringOps(); // создали объект
              StrMod strOp = so.ReplaceSpaces;
 …… // те же действия, что раньше
              StrMod strOp = so.RemoveSpaces;
 …… // те же действия, что раньше
              StrMod strOp = so.Reverse;
…… // те же действия, что раньше
    }
}
```

C# делегаты

Групповая адресация

Это возможность создать цепочку вызовов для методов, которые вызываются автоматически и последовательно при обращении к делегату.

Для этого надо создать экземпляр делегата, а потом добавить методы с помощью оператора + или += Для удаления метода из цепочки служит оператор - или -= .

Если делегат возвращает значение, то им становится значение, возвращаемое последним методом в списке вызовов. Поэтому делегат, в котором используется групповая адресация обычно имеет возвращаемый тип **void**

C# события

События – это механизм автоматического уведомление о том, что произошло некоторое событие. События работают так:

- Объект, проявляющий интерес к конкретному событию, регистрирует обработчик этого события
- Когда событие происходит, вызываются все зарегистрированные обработчики этого события.

Обработчики события обычно представлены делегатами.

События являются членами класса и объявлятся с помощью ключевого слова **event**

Формализм объявления

event делегат_события имя_события

делегат_события – имя делегата, используемого для поддержки события

имя_события – конкретный объект объявляемого события

C# события

```
// Объявление делегата для события
 delegate void MyEventHandler();
//Объявление класса, содержащего событие
 class MyEvent {
     public event MyEventHandler SomeEvent;
//метод в классе для генерации события с помощью делегата SomeEvent
     public void OnSomeEvent() {
       if(SomeEvent != null)
             SomeEvent();
     }
}
 class DemoEvent {
     static void Handler() { // обработчик события
       Console.WriteLine( “ Произошло событие !!!”);
     }
     static void Main() {
       MyEvent evt = new MyEvent(); // конструируем объект содержащий
                    //событие
```

C# события

```
        //Добавляем метод Handler() в список событий
        evt.SomeEvent += Handler ; // фактически добавляем метод вызываемый
                                   //делегатом для события
        // запускаем событие
        evt.OnSomeEvent();
    }
}
```

Все события активизируются с помощью делегатов. Поэтому тип делегата для события определяет возвращаемый тип и сигнатуру события. В примере это тип void и пустая сигнатура.

В классе события MyEvent объявляется метод OnSomeEvent(), вызываемый для сигнализации о запуске события. Он вызывается когда происходит событие.

В методе OnSomeEvent() вызывается обработчик событий с помощью делегата SomeEvent. Он вызывается только если событие SomeEvent не является пустым (SomeEvent ! = null)

C# события

Т.к. интерес к событию должен быть зарегистрирован в других частях программы, то чтобы получать уведомление о нем, метод OnSomeEvent может быть вызван и до регистрации любого обработчика событий, что недопустимо. Для того чтобы не было вызова по пустой ссылке делегат события проверяется не «пустоту»

Событие -> уведомление –>делегат -> обработчик

В классе DemoEvent создается обработчик Handler().

В методе Main() объект класса события MyEvent, а Handler() регистрируется как обработчик этого события путем его добавления в список событий

evt.SomeEvent += Handler ;

Обработчики могут добавляться в список событий только с помощью оператора **+=,** а удаляться из него с помощью **-=**

Вызов метода evt.OnSomeEvent) приводит к вызову всех событий, зарегистрированных обработчиком. В данном случае зарегистрирован только один такой обработчик.

C# события

Как и делегаты, события поддерживают групповую адресацию. Это дает возможность нескольким объектам реагировать на уведомление о событии.

```
using System;
// Пример  групповой адресации события
namespace Myevent{
   // Объявлякем тип делегата для события
   delegate void MyEventHandler();
   class MyEvent { //класс  генерирующий событие
      //Объявляем делегат, содержащий событие
      public event MyEventHandler SomeEvent;
      public void OnSomeEvent() {//Этот метод генерирует событие
         if (SomeEvent != null)
        SomeEvent();
      }
   }
```

C# события

```
class X {
    public void Xhandler()  // Обработчик события совместимый с делегатом /
            //MyEventHandler
    { Console.WriteLine("Событие получено объектом класса X"); }
 }
 class Y {
    public void Yhandler()
    { Console.WriteLine("Событие получено объектом класса Y"); }
 }
 class Z {
    int id;
    public Z( int z) {
        id = z;
    }
    public void Zhandler()
    {
        Console.WriteLine("Событие получено объектом " + id + " класса Z ");
    }
 }
```

C# события

```
class Program{
    static public void Handler()     // Статический ббработчик события совместимый с
                                     // делегатом MyEventHandler
    {
        Console.WriteLine("Событие получено объектом класса Program");
    }
    static void Main() {
        MyEvent evt = new MyEvent(); //Объект в котором генерируется событие
        X xOB = new X();
        Y yOB = new Y();
        // Добавляем обработчики в список событий
        evt.SomeEvent += xOB.Xhandler;
        evt.SomeEvent += yOB.Yhandler;
        evt.SomeEvent += Handler;
        //Генерируем событие
        evt.OnSomeEvent();
        Console.WriteLine();
        Console.ReadKey();
```

C# события

```
// Удаляем обработчик из цепочки
        evt.SomeEvent -= yOB.Yhandler;
        Console.WriteLine("Обработчик события для объекта yOB удален ");
        //Генерируем событие
        evt.OnSomeEvent();
        Console.WriteLine();
        Console.ReadKey();
        // Удаляем обработчик
        evt.SomeEvent -= xOB.Xhandler;
        evt.SomeEvent -= Handler;
        Console.WriteLine("Все обработчики удалены");
        //Генерируем событие
        evt.OnSomeEvent();
        Console.ReadKey();
```

C# события

```
            Z obZ1 = new Z(1);
            Z obZ2 = new Z(2);
            Z obZ3 = new Z(3);
            evt.SomeEvent += obZ1.Zhandler;
            evt.SomeEvent += obZ2.Zhandler;
            evt.SomeEvent += obZ3.Zhandler;
            Console.WriteLine("Обработчики события добавлены для всех трех
              объектов типа Z");
Console.WriteLine();
            //Генерируем событие
           Console.WriteLine(" Событие сгенерировано");
            Console.WriteLine();
            evt.OnSomeEvent();  //
            Console.ReadKey();
        }
    }
}
```

C# события

В этом примере создаются три класса X, Y и Z, в которых определяются обработчики событий совместимых с делегатом MyEventHandler. Поэтому эти обработчики могут быть включены в цепочку событий. Обработчики этих классов не являются статическими, поэтому до включения их в цепочку объекты соответствующих классов должны быть созданы.

Обработчик Handler класса Program статический, поэтому может включаться в цепочку когда угодно. При этом уведомление о событии распространяется на весь класс.

В случае если, в качестве обработчика используется метод экземпляра, то события адресуются конкретным экземплярам объектов. Следовательно, каждый объект класса, которому нужно получить уведомление о событии, должен быть зарегестрирован отдельно.

C# события

Обработка событий в среде .NET

Для совместимости со средой необходимо, чтобы:

- Обработчик событий должен иметь 2 параметра
- Первый параметр – ссылка на объект формирующий событие. Второй параметр – должен иметь тип EventArgs. В нем должна содержаться дополнительная информация о событии, которая требуется обработчику

Форма .NET совместимого обработчика

```
void имя_обработчика( object отправитель, EventArgs аргумент) {
    // тело обработчика
}
```

Отправитель – параметр, передаваемый вызывающим кодом с помощью ключевого слова **this**

Аргумент **-** параметр типа EventArgs содержит дополнительную информацию о событии и может быть проигнорирован

C# события

Класс EventArgs не содержит полей, которые могут быть использованы для передачи дополнительной информации обработчику. **Он является базовым классом, от которого получается производный класс, содержащий все необходимые поля с дополнительной информацией.**

Например создадим свой класс – наследник класса EventArgs, содержащий поле где будет находится информация о номере события

```
class MyEventArgs : EventArgs
{
    public int EventNum;
}
```

Объект этого класса будет использоваться в классе связанном с событием (генерирующем событие)

Объявим делегат с двумя аргументами

```
delegate void MyEventHandler( object source, MyEventArgs arg);
```

C# события

```
class MyEvent { // класс - событие
    static int count = 0;
    //Объявляем делегат, содержащий событие
    public event MyEventHandler SomeEvent;
    //Этот метод генерирует событие
    public void OnSomeEvent() {
        MyEventArgs arg = new MyEventArgs();
        if (SomeEvent != null) {
            arg.EventNum = count++;
            SomeEvent(this, arg);  // генерация события
// При этом передаются два аргумента – кто инициировал
   событие и доп. информация о событии
        }
    }
}
```

C# лямбда выражения

Метод, на который ссылается делегат, часто используется только для этой цели. Т.е. такой метод вызывается только через делегат, но сам никогда вызываться не будет. В таких случаях можно использовать анонимную функцию, а не создавать сам метод.

Анонимная функция это кодовый блок без имени, передаваемый конструктору делегата.

В C# есть две разновидности анонимных функций – анонимные методы и лямбда выражения.

Анонимные метод – способ создания безымянного блока кода, связанного с конкретным экземпляром делегата. Для этого надо после ключевого слова delegate указать сам кодовый блок. Тип делегата должен быть объявлен раньше.

Анонимный метод может иметь параметры и возвращать значение, в соответствии с объявленным типом делегата

C# лямбда выражения

```
delegate int CountIt(int end);  // объявление типа делегата
class AnonMethod {
    static void Main() { //посчет суммы чисел от 0  до числа end
      int result;
// параметр передается анонимному методу делегата
      CountIt count = delegate( int end) { // начало кодового блока
            int sum = 0;
            for(int i = 0; i < = end; i++) {
                  Console.WriteLine(i);
                  sum += i;
            }
            return sum; // возвращается значение из метода
      }; // конец кодового блока  ( ; - обязательна)
      result = count(5);
       Console.WriteLine(“ Сумма первых 5 чисел  равна “ + result);
    }
}
```

C# лямбда выражения

Лямбда выражение – метод создания анонимной функции.

Во всех этих выражениях применяется оператор => который разделяет выражение на 2 части.

В левой части указывается входной параметр (или список параметров), а в правой – тело лямбда выражения.

Оператор => описывается такими словами, как «переходит» или «становится».

В одиночном лямбда выражении, часть стоящая справа от => воздействует на параметры, стоящие слева. Возвращаемый результат это результат выполнения лямбда оператора.

Формализм

параметр => выражение

или если параметров несколько

(список параметров) => выражение

C# лямбда выражения

count => count + 2

Здесь count – параметр, на который воздействует выражение count + 2. В результате count увеличивается на 2.

n => n%2 == 0

Здесь выражение возвращает логическое значение (true – если параметр четный или false – если параметр нечетный). В результате параметр n примет логическое значение.

Лямбда выражение применяется в два этапа.

- Сначала объявляется тип делегата совместимый с лямбда - выражением . Затем экземпляр делегата, которому присваивается лямбда – выражение.
- После этого лямбда – выражение вычисляется при обращении к экземпляру делегата. Результатом вычисления выражения становится возвращаемое значение.

C# лямбда выражения

```
// объявление делегата, с аргументом типа int, возвращающем тип int
 delegate int Incr( int v);
// объявление делегата, с аргументом типа int, возвращающем тип bool
 delegate bool IsEven( int v);
 class LambdaDemo {
     //Создаем экемпляр делегатф Incr, ссылающийся на лямбда - выражение
     Incr incr = count => count +2;
// используем    лямбда – выражение incr
     Console.WriteLine("Использование лямбда – выражение incr");
     int x = -10;
     while( x <= 0) {
       Console.Write( x + " ");
       x = incr(x); // увеличиваем x на 2
     }
     Console.Write( "\n ");
     //Создаем экемпляр делегата IsEven, ссылающийся на  другое выражение
      IsEven iseven = n => n%2 == 0;
```

C# лямбда выражения

```
    // используем    лямбда – выражение iseven

    Console.WriteLine(“Использование лямбда – выражение iseven”);
    for(int i = 0; i <= 10; i++) {
      if(iseven( I) {
       Console.WriteLine( I + “ четное”);
      }
    }
}
```

Результат запуска программы:

Использование лямбда – выражение incr

-10 –8 –6 –4 –2 0

Использование лямбда – выражение iseven

2 - четное

4 - четное

6 - четное

8 - четное

10 - четное

C# лямбда выражения

Обратите внимание на совместимость лямбда-выражений incr и iseven с соответствующими делегатами.

Как компилятор разбирается с типами данных, используемых в разных лямбда-выражениях?

Компилятор определяет значение типа параметра и типа результата выражения по типу соответствующего делегата.

Иногда приходится явно указывать тип параметра лямбда-выражения.

```
Incr incr = (int count) => count +2;
```

При этом скобки – обязательны

Если в лямбда-выражении используется несколько параметров, то их обязательно заключать в скобки

```
(low, high, val) =>  val >= low && val  <= high;
```

При этом делегат объявляется так

```
delegate bool InRange(int lower, int upper, int v);
```

C# лямбда выражения

Эземпляр этого делегата создается так

```
InRange rangeOk = (low, high, val) => low && val  <= high;
```

После этого лямбда-выражение может использоваться. Например

```
 if( rangeOk(1,5,3))
    Console.WriteLine(“число 3 находится между числами 1 и 5”);
```

Блочные лямбда - выражения

Это выражения в правой части которых могут быть несколько операторов. Для того, чтобы его создать необходимо просто заключить тело выражения в фигурные скобки { }

Блочное выражение ничем не отличается по функциональности от одиночного

C# лямбда выражения

```
// Объявление делегата с аргументом типа int и возвращаемым значением типа int
 delegate int IntOP( int end);
 class LamdaStatement {
     static void Main() {
 //Блочное лямбда – выражение возвращает факториал
 //передаваемого ему значения
       IntOp fact = n => {
             int r = 1;
             for( int i = 1; i <= n; i++)
                   r = r * i;
             return r ;
               };
       Console.WriteLine(“Факториал 3 равен ” + fact(3));
       Console.WriteLine(“Факториал 6 равен ” + fact(6));
     }
}
```

C# лямбда выражения

В этом примере в теле блочного выражения объявляется переменная, организуется цикл и используется оператор return. Оно очень похоже на анонимный метод. Поэтому многие анонимные методы могут быть преобразованы в в блочные лямбда – выражения.

Следует отметить, что оператор return обуславливает возврат из блочного лямбда – выражения, но не возврат из метода в котором присутствует это выражение.

Анонимные методы и лямбда – выражения удобны для работы с событиями, поскольку обработчик события события часто вызывается только в коде, реализующем механизм обработки событий.

Рекомендуется писать обработчики событий с использованием лямбда – выражений