# Объектно-событийное программирование

## Делегаты

**Делегат (delegate)** — это тип, который позволяет хранить ссылки на функции. Объявляются делегаты практически также, как и функции, но только безо всякого тела функции и с ключевым словом **delegate.**

В объявлении любого делегата указывается возвращаемый тип и список параметров.

[спецификаторы ] delegate тип имя\_делегата ( [ параметры ] )

Спецификаторы делегата имеют тот же смысл, что и для класса, причем допускаются только спецификаторы new, public, protected, internal и private.

Тип – тип функции, параметры – параметры функции.

public delegate void D ( int i );

После определения делегата можно объявлять переменную с типом этого делегата. Далее эту переменную можно инициализировать как ссылку на любую функцию, которая имеет точно такой же возвращаемый тип и список параметров, как и у делегата. После этого функцию можно вызывать с использованием переменной делегата так, будто бы это и есть сама функция.

Пример 1.

class Program

{

**delegate double Process(double number1, double number2);//делегат**

static double Multiply(double number1, double number2)

{

return number1 \* number2;

}

static double Divide(double number1, double number2)

{

return number1 / number2;

}

static void Main(string[] args)

{ string[] Numbers=null;

string buf;

double number1=0, number2=0;

bool ok = false;

do

{

Console.WriteLine("Введите 2 числа через запятую");

buf = Console.ReadLine();

if (buf.Contains(','))

{

Numbers = buf.Split(',');

ok = true;

}

if (!ok)

Console.WriteLine("Error!");

if(ok)

{

ok = Double.TryParse(Numbers[0], out number1);

if (!ok)

Console.WriteLine("Error!");

ok = Double.TryParse(Numbers[1], out number2);

if (!ok)

Console.WriteLine("Error!");

}

} while (!ok);

**Process proc; //объект типа делегат**

int answer=0;

do

{

Console.WriteLine("Введите операцию:\n 1. Умножение \n 2.Деление \n 3. Выход");

answer=Convert.ToInt32(Console.ReadLine());

switch (answer)

{

case 1: **proc= new Process(Multiply);** //подставляем функцию Console.WriteLine("Результат = {0}",**proc(number1,number2))**; //вызов функции

break;

case 2: **proc= new Process(Divide);** //подставляем функцию Console.WriteLine("Результат = {0}",**proc(number1,number2))**; //вызов функции

break;

}

}while(answer<3);

}

}

Пример 2. Вызов методов класса с помощью делегата

namespace Lect5\_1

{

// Объявляем делегат

delegate string strMod(string stx);//модификаци строки

class ModifyString

{

public string replaceSpaces(string a) // Метод заменяет пробелы дефисами.

{

Console.WriteLine("Замена пробелов дефисами.");

return a.Replace(' ', '-');

}

public string removeSpaces(string a)// Метод удаляет пробелы.

{

string temp = "";

int i;

Console.WriteLine("Удаление пробелов.");

for (i = 0; i < a.Length; i++)

if (a[i] != ' ') temp += a[i];

return temp;

}

public string reverse(string a)// Метод реверсирует строку

{

string temp = "";

int i, j;

Console.WriteLine("Реверсирование строки.");

for (i = a.Length - 1; i >= 0; i--)

temp += a[i];

return temp;

}

static void Main(string[] args)

{

// Создаем экземпляр класса ModifyString.

ModifyString ms = new ModifyString();

// Создание делегата.

strMod strOp = new strMod(ms.replaceSpaces);

string str;

// Вызываем методы посредством делегата.

str = strOp("Это простой тест.");

Console.WriteLine("Результирующая строка: " + str);

Console.WriteLine();

strOp = new strMod(ms.removeSpaces);

str = strOp("Это простой тест.");

Console.WriteLine("Результирующая строка: " + str);

Console.WriteLine();

strOp = new strMod( ms.reverse);

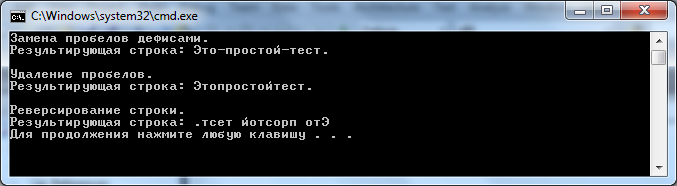
str = strOp("Это простой тест.");

Console.WriteLine("Результирующая строка: " + str);

}

}

}



## Поддержка многоадресатной передачи (multicasting).

**Многоадресатная передача** — это способность создавать список вызовов (или цепочку вызовов) методов, которые должны автоматически вызываться при вызове делегата. Для этого достаточно создать экземпляр делегата, а затем для добавления методов в эту цепочку использовать оператор "+=". Для удаления метода из цепочки используется оператор " - = " (можно + и – соответственно). Делегат с многоадресатной передачей имеет одно ограничение: он должен возвращать тип void.

namespace Lect5\_1

{

// Объявляем делегат

delegate void strMod( ref string stx);//модификаци строки

class ModifyString

{

static void replaceSpaces(ref string a) // Метод заменяет пробелы дефисами.

{

Console.WriteLine("Замена пробелов дефисами.");

a=a.Replace(' ', '-');

}

static void removeSpaces(ref string a)// Метод удаляет пробелы.

{

string temp = "";

int i;

Console.WriteLine("Удаление пробелов.");

for (i = 0; i < a.Length; i++)

if (a[i] != ' ') temp += a[i];

a= temp;

}

static void reverse( ref string a)// Метод реверсирует строку

{

string temp = "";

int i;

Console.WriteLine("Реверсирование строки.");

for (i = a.Length - 1; i >= 0; i--)

temp += a[i];

a= temp;

}

static void Main(string[] args)

{

ModifyString ms = new ModifyString(); // Создаем экземпляр класса ModifyString.

// Создаем экземпляры делегатов.

strMod strOp;

strMod replaceStr = new strMod(replaceSpaces);

strMod removeStr = new strMod(removeSpaces);

strMod reverseStr = new strMod(reverse);

string str = "Это простой тест.";

// Организация многоадресатной передачи.

strOp = replaceStr;

strOp += reverseStr;

// Вызов делегата с многоадресатной передачей.

strOp(ref str);

Console.WriteLine("Результирующая строка: " + str);

Console.WriteLine();

// Удаляем метод замены пробелов и

// добавляем метод их удаления.

strOp -= replaceStr;

strOp += removeStr;

str = "Это простой тест."; // Восстановление

// исходной строки.

// Вызов делегата с многоадресатной передачей.

strOp(ref str);

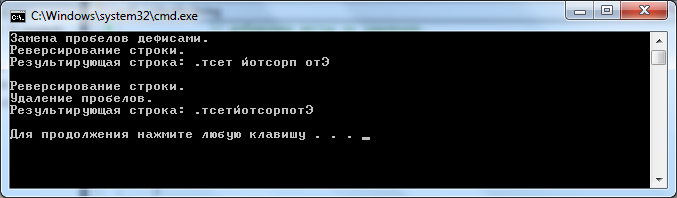
Console.WriteLine("Результирующая строка: " + str);

Console.WriteLine();

}

}

}



Как и все в С#, делегаты являются **классами**. В пространстве имен System определены два класса: Delegate и MulticastDelegate. При определении делегата в качестве базового компилятором назначается один из этих классов в зависимости от типа возвращаемого значения: для типа void базовым является MulticastDelegate, для остальных типов – Delegate. Существенным отличием этих двух типов делегатов является возможность сохранения в одном делегате типа MulticastDelegate указателей сразу на несколько функций. Эти указатели хранятся в виде списка, который называют **списком вызова**. При обращении к такому делегату функции будут выполняться по очереди в порядке их помещения в список вызова.

Добавить функцию в список вызова можно с помощью статического метода Combine(), который сцепляет списки вызовов или массивы делегатов (метод получает в качестве параметра массив делегатов), или двух делегатов (метод получает в качестве параметров два делегата):

Delegate Combine(Delegate[]);

Delegate Combine(Delegate, Delegate);

Также для этих целей можно использовать перегруженную операцию **"+="**. Второй операнд операции представляет собой делегат или функцию, которая добавляется в список вызова исходного делегата.

Обратную операцию – удаление метода из списка вызова выполняют статический метод Remove() и операция **"-="**.

Delegate Remove(Delegate, Delegate);

Удаляется последнее вхождение метода или списка вызовов делегата из списка вызовов другого делегата.

Делегаты становятся еще более гибкими средствами программирования благодаря двум свойствам: **ковариантности и контравариантности**.

**Ковариантность** позволяет присвоить делегату метод, возвращаемым типом которого служит класс, производный от класса, указываемого в возвращаемом типе делегата (полиморфизм).

**Контравариантность** позволяет присвоить делегату метод, типом параметра которого служит класс, являющийся базовым для класса, указываемого в объявлении делегата.

## События

События похожи на исключения тем, что они тоже генерируются, т.е. выдаются объектами, и тем, что для них тоже можно предоставлять реагирующий на них выполнением какого-нибудь действия код. Однако существует и несколько отличий, наиболее важное из которых состоит в отсутствии для обработки событий структуры, эквивалентной try. . . catch. Вместо применения этой структуры на события нужно подписываться (subscribe).

Под подпиской на событие подразумевается предоставление кода, который должен выполняться при генерации данного события, в виде обработчика событий (event handler).

На событие можно подписывать несколько обработчиков, которые тогда все будут вызываться при генерации этого события. Эти обработчики могут являться как частью того класса объекта, который генерирует данное событие, так и частью других классов.

Сами обработчики событий представляют собой просто функции. Единственным ограничением для такой функции является то, что ее возвращаемый тип и параметры должны обязательно соответствовать тем, которых требует событие. Это ограничение входит в состав определения события и задается делегатом.

Базовая последовательность обработки выглядит следующим образом:

1. Приложение создает объект, который может генерировать событие.
2. Приложение подписывается на событие.
3. При генерации события подписчику отправляется соответствующее уведомление.

## Обработка событий

Для обработки события на него нужно **подписываться**, предоставляя функцию — обработчик событий, возвращаемый тип и параметры которой должны совпадать с возвращаемым типом и параметрами делегата, закрепленного для применения с этим событием.

class Program

{

static int counter=0;

static string displayString="This string will appear one letter at a time\n";

static void Main(string[] args)

{

//создается объект, который будет генерировать события

Timer myTimer=new Timer(100);

//подключение обработчика к событию myTimer.Elapsed – подписка на событие

**myTimer.Elapsed+=new ElapsedEventHandler(WriteChar);**

myTimer.Start();//запуск таймера

Console.ReadLine();

}

static void WriteChar(object sourse, ElapsedEventArgs e)

{

Console.Write(displayString[counter++ % displayString.Length]);

}

}

Объект Timer инициирует событие Elapsed. Обработчик для этого события должен обязательно иметь такой же возвращаемый тип и параметры, как тип-делегат System.Timers.ElapsedEventHandler, который является одним из стандартных делегатов, поставляемых с .NET Framework.

Работа приложения не может быть завершена до того, как будут обработаны те или иные события, поэтому функция Main() помещается в режим ожидания до тех пор пока пользователь не нажмет какую-нибудь клавишу.

## Определение событий

Перед определением события требуется обязательно определить используемый вместе с событием тип делегата, т.е. тип делегата, типу и параметрам которого должен соответствовать метод обработки событий. Для выполнения этого используется стандартный синтаксис делегатов, с помощью которого необходимый делегат определяется как общедоступный.

namespace events

{

delegate void MyEventHandler();// Объявляем делегат для события

class MyEvent // Объявляем класс события

{

public event **MyEventHandler** SomeEvent;

// Этот метод вызывается для генерирования события

public void OnSomeEvent()

{

//проверяем, есть ли элементы в списке вызова

if (SomeEvent != null)

SomeEvent();//вызываем метод, подписанный на событие

}

}

class EventDemo

{

static void handler() // Обработчик события

{

Console.WriteLine("Произошло событие.");

}

static void Main(string[] args)

{

MyEvent evt = new MyEvent();

// Добавляем метод handler() в список вызовов события - подписка

evt.SomeEvent += new MyEventHandler(handler);

// Генерируем событие,

evt.OnSomeEvent();

}

}

}

Подобно делегатам события могут предназначаться для **многоадресатной передачи**. В этом случае на одно уведомление о событии может отвечать несколько объектов.

delegate void MyEventHandler();// Объявляем делегат для события

class MyEvent // Объявляем класс события

{

public event MyEventHandler SomeEvent;

public void OnSomeEvent() {

if (SomeEvent != null)

SomeEvent();

}

}

class X

{

public void Xhandler()

{

Console.WriteLine("Событие, полученное объектом X.");

}

}

class Y

{

public void Yhandler()

{

Console.WriteLine("Событие, полученное объектом Y.");

}

}

class EventDemo

{

static void handler()

{

Console.WriteLine("Событие, полученное классом EventDemo.");

}

static void Main(string[] args)

{

MyEvent evt = new MyEvent();

X xOb = new X();

Y yOb = new Y();

// Добавляем метод handler() в список события,

evt.SomeEvent += new MyEventHandler(handler);

evt.SomeEvent += new MyEventHandler(xOb.Xhandler);

evt.SomeEvent += new MyEventHandler(yOb.Yhandler);

// Генерируем событие,

evt.OnSomeEvent();

Console.WriteLine();

}

}

## Примеры программ с использованием делегатов и событий

**Пример 1.** Решение квадратного уравнения (делегаты)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Equation

{

//делегат для определения метода решения уравнения

public delegate void Method();

public class Equation

{

double a, b, c;//коэффициенты уравнения

double x1, x2;//корни уравнения

int count;//количество корней

int type = 1;//тип уравнения: 0 - ур-е 0 порядка, 1 - 1-ого порядка, 2- квадратное

public Method SolveEquation = null;//объект типа делегат для хранения метода для решения уравнения в зависимости от типа

public Equation(params double[] coef) //конструктор

{

switch (coef.Length)

{

case 3:

{

a = coef[0];

b = coef[1];

c = coef[2];

break;

}

case 2:

{

a = 0.0;

b = coef[0];

c = coef[1];

break;

}

case 1:

{

a = 0.0;

b = 0.0;

c = coef[0];

break;

}

default: throw new Exception("Данный набор коэффициентов не может определять квадратное уравнение");

}

SolveEquation += DefineTypeEquation; //в список вызова добавляется метод для определения тип уравнения

SolveEquation += Solve;//в список вызова добавляется метод для решения уравнения

SolveEquation += PrintSolution;//в список вызова добавляется метод для печати корней уравнения

}

public void DefineTypeEquation()

{

if (a == 0)

if (b == 0) // уравнение 0-ого порядка

type = 0;

else // линейное уравнение

type = 1;

else // квадратное уравнение

type = 2;

}

public void Solve()

{

switch (type)

{

case 0: NullSolve(); break;

case 1: LinSolve(); break;

case 2: QSolve(); break;

}

}

// решение квадратного уравнения

void QSolve()

{

double disc = b \* b - 4 \* a \* c;

if (disc < 0.0)

count = 0;

else

if (disc == 0.0)

{

count = 1;

x1 = -b / (2 \* a);

x2 = x1;

}

else

{

count = 2;

x1 = (-b + Math.Sqrt(disc)) / (2 \* a);

x2 = (-b - Math.Sqrt(disc)) / (2 \* a);

}

}

//решение линейного уравнения

void LinSolve()

{

count = 1;

x1 = -c / b;

}

// решение уравнения 0-ого порядка

void NullSolve()

{

if (c == 0)

count = Int32.MaxValue;

else

count = 0;

}

// метод печати результата решения уравнения

public void PrintSolution()

{

switch (count)

{

case 2:

Console.WriteLine("x1={0}, x2={1}", x1, x2);

break;

case 1:

Console.WriteLine("x={0}", x1);

break;

case 0:

Console.WriteLine("Корней нет");

break;

case Int32.MaxValue:

Console.WriteLine("Любое x является решением");

break;

default:

Console.WriteLine("Уравнение еще не решено");

break;

}

}

}

class Program

{

static void Main(string[] args)

{

try

{

Equation e1 = new Equation(1, -2, -1);

e1.SolveEquation();

Equation e2 = new Equation(1, -2);

e2.SolveEquation();

Equation e3 = new Equation(0);

e3.SolveEquation();

}

catch (Exception ex)

{

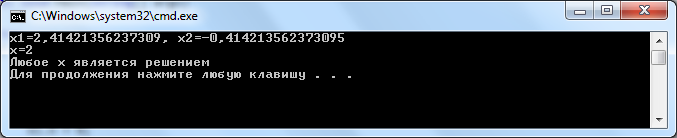
Console.WriteLine(ex.Message);

}

}

}

}



**Пример 2.** Добавим событие, которое будет отслеживать событие изменения коэффициентов квадратного уравнения – при изменении любого коэффициента требуется решить уравнение заново.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Equation

{

public delegate void Method();//делегат для определения метода ещения уравнения

**public delegate void EquationDelegate(Equation e); //делегат для отслеживания изменения корней уравнения**

public class Equation

{

double a, b, c;//коэффициенты уравнения

double x1, x2;//корни уравнения

int count;//количество корней

int type = 1;//тип уравнения: 0 - ур-е 0 порядка, 1 - 1-ого порядка, 2- квадратное

public Method SolveEquation = null;//объект типа делегат для хранения метода для решения уравнения в зависимости от типа

**public event EquationDelegate ReplaceCoefEvent;//событие - изменение корней уравнения**

public Equation(params double[] coef) //конструктор

{

switch (coef.Length)

{

case 3:

{

a = coef[0];

b = coef[1];

c = coef[2];

break;

}

case 2:

{

a = 0.0;

b = coef[0];

c = coef[1];

break;

}

case 1:

{

a = 0.0;

b = 0.0;

c = coef[0];

break;

}

default: throw new Exception("Данный набор коэффициентов не может определять квадратное уравнение");

}

SolveEquation += DefineTypeEquation; //в список вызова добавляется метод для определения тип уравнения

SolveEquation += Solve;//в список вызова добавляется метод для решения уравнения

SolveEquation += PrintSolution;//в список вызова добавляется метод для печати корней уравнения

}

**public double A**

**{**

**get**

**{**

**return a;**

**}**

**set**

**{**

**a = value;**

**// вызов обработчиков события изменения коэффициентов**

**if (ReplaceCoefEvent != null)**

**ReplaceCoefEvent(this);**

**}**

**}**

**public double B**

**{**

**get**

**{**

**return b;**

**}**

**set**

**{**

**b = value;**

**// вызов обработчиков события изменения коэффициентов**

**if (ReplaceCoefEvent != null)**

**ReplaceCoefEvent(this);**

**}**

**}**

**public double C**

**{**

**get**

**{**

**return c;**

**}**

**set**

**{**

**c = value;**

**// вызов обработчиков события изменения коэффициентов**

**if (ReplaceCoefEvent != null)**

**ReplaceCoefEvent(this);**

**}**

**}**

public void DefineTypeEquation()

{

if (a == 0)

if (b == 0) // уравнение 0-ого порядка

type = 0;

else // линейное уравнение

type = 1;

else // квадратное уравнение

type = 2;

}

public void Solve()

{

switch (type)

{

case 0: NullSolve(); break;

case 1: LinSolve(); break;

case 2: QSolve(); break;

}

}

// решение квадратного уравнения

void QSolve()

{

double disc = b \* b - 4 \* a \* c;

if (disc < 0.0)

count = 0;

else

if (disc == 0.0)

{

count = 1;

x1 = -b / (2 \* a);

x2 = x1;

}

else

{

count = 2;

x1 = (-b + Math.Sqrt(disc)) / (2 \* a);

x2 = (-b - Math.Sqrt(disc)) / (2 \* a);

}

}

//решение линейного уравнения

void LinSolve()

{

count = 1;

x1 = -c / b;

}

// решение уравнения 0-ого порядка

void NullSolve()

{

if (c == 0)

count = Int32.MaxValue;

else

count = 0;

}

// метод печати результата решения уравнения

public void PrintSolution()

{

switch (count)

{

case 2:

Console.WriteLine("x1={0}, x2={1}", x1, x2);

break;

case 1:

Console.WriteLine("x={0}", x1);

break;

case 0:

Console.WriteLine("Корней нет");

break;

case Int32.MaxValue:

Console.WriteLine("Любое x является решением");

break;

default:

Console.WriteLine("Уравнение еще не решено");

break;

}

}

}

class Program

{

**static void ReSolve(Equation e)**

**{**

**e.DefineTypeEquation();**

**e.Solve();**

**e.PrintSolution();**

**}**

static void Main(string[] args)

{

try

{

Equation e1 = new Equation(1, -2, -1);

e1.ReplaceCoefEvent += ReSolve;

e1.SolveEquation();

**e1.A = 0;**

**e1.B = 0;**

**e1.C = 0;**

}

catch (Exception ex)

{

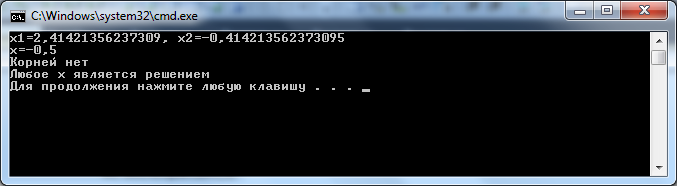
Console.WriteLine(ex.Message);

}

}

}

}



**Пример 3. Приложение для мгновенного обмена сообщениями**.

С# позволяет программисту создавать события любого типа. Однако в целях следовать рекомендациям, подготовленным Microsoft специально для этих целей. Центральное место в этих рекомендациях занимает требование того, чтобы обработчики событий имели два параметра. Первый должен быть ссылкой на объект, который будет генерировать событие. Второй должен иметь тип EventArgs и содержать остальную информацию, необходимую обработчику. Таким образом, .NET-совместимые обработчики событий должны иметь следующую общую форму записи:

void handler(object source, EventArgs arg) {}

Обычно параметр source передается вызывающим кодом. Параметр типа EventArgs содержит дополнительную информацию, компонентной совместимости со средой .NET Framework необходимо которую в случае ненадобности можно проигнорировать. Класс EventArgs не содержит полей, которые используются при передаче дополнительных данных обработчику; он используется в качестве базового класса, из которого можно выводить класс, содержащий необходимые поля. Но поскольку многие обработчики обходятся без дополнительных данных, в класс EventArgs включено статическое поле Empty, которое задает объект, не содержащий никаких данных.

**Рассмотрим приложение для мгновенного обмена сообщениями**. Оно создает объект, представляющий соединение с удаленным пользователем. Этот объект может генерировать событие при поступлении сообщения от удаленного пользователя по этому соединению.

Приложение для мгновенного обмена сообщениями должно определять функцию, пригодную для использования с указанным в событии типом делегата, и передает ссылку на эту функцию событию. Эта функция-обработчик событий может представлять собой метод в другом объекте, например, объекте дисплея, и отображает мгновенные сообщения на экране по мере их поступления.

При генерации события подписчику отправляется соответствующее уведомление. Это означает, что при поступлении мгновенного сообщения через объект соединения в объекте, представляющем дисплей, вызывался бы метод обработчика событий. Поскольку используется стандартный метод, объект, генерирующий событие, может передавать любую имеющую отношение к делу информацию через параметры и тем самым делать события очень разнообразными. В случае взятого в качестве примера приложения одним из таких параметров мог бы быть текст мгновенного сообщения, который обработчик событий тогда бы мог отображать в объекте, представляющем дисплей.

Приложение

Объект, предоставляющий соединение

Объект, предоставляющий дисплей

Создает

Создает

Подписывается на

Рисунок

Рассмотрим пример реализации сценария мгновенного обмена сообщениями. Объект Connection, генерирует события, за обработку которых отвечает объект Display.

public delegate void MessageHandler(string text);//делегат

public class Connection

{

public event MessageHandler MessageArraved;//событие

private Timer timer;

public Connection()

{

timer = new Timer(100);

//регистрируется обработчик события

timer.Elapsed += new ElapsedEventHandler(CheckForMessage);

}

public void Connect()

{

timer.Start();//таймер включен

}

public void Disconnect()

{

timer.Stop();//таймер выключен

}

private static Random rnd = new Random();

private void CheckForMessage(object source, ElapsedEventArgs e)

{

Console.WriteLine("Checking for new messages");

//логика, по которой генерируется событие

if ((rnd.Next(9) == 0) && (MessageArraved != null))

MessageArraved("Hello!");

}

}

//класс, который подписывается на событие

public class Display

{

public void DisplayMessage(string message)

{

Console.WriteLine("Поступило новое сообщение: {0}", message);

}

}

class Program

{

static void Main(string[] args)

{

Connection connection = new Connection();

Display display = new Display();

connection.MessageArraved += new MessageHandler(display.DisplayMessage);

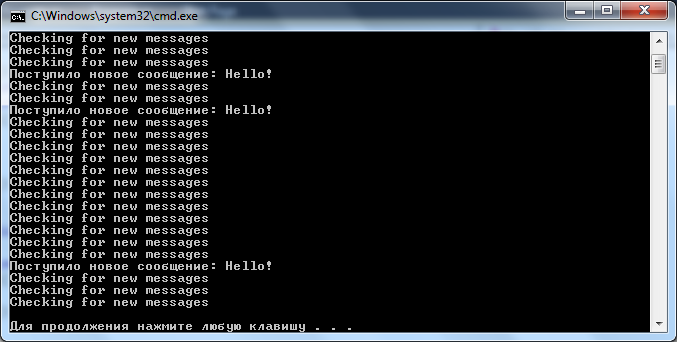
connection.Connect();

Console.ReadLine();

}

}

Событие должно генерироваться только в случае, если выражение   
MessageArrived ! = null в результате дает true. С помощью этого выражения определяется, есть ли у делегата какие-то подписчики. Если подписчиков нет, тогда MessageArrived вычисляется в null, и в генерации события нет никакого смысла.



## Универсальные обработчики событий

Можно использовать один обработчик событий для нескольких идентичных событий, генерируемых разными объектами, и при этом все равно знать, какой конкретно объект сгенерировал данное событие.

Для этого в качестве параметра обработчика событий можно отправить ссылку на объект, генерирующий событие, и таким образом настроить ответ этого обработчика на определенный объект. Ссылка открывает доступ к исходному объекту и его свойствам.

//класс для передачи параметров

**public class MessageArrivedEventArgs**

**{**

**string message;**

**public string Message**

**{**

**get**

**{**

**return message;**

**}**

**}**

**public MessageArrivedEventArgs()**

**{**

**message = "no message send";**

**}**

**public MessageArrivedEventArgs(string newMessage)**

**{**

**message = newMessage;**

**}**

**}**

**public delegate void MessageHandler(Connection connect, MessageArrivedEventArgs e);**

public class Connection

{

public event MessageHandler MessageArraved;

private Timer timer;

**private string name;**

**public string Name**

**{**

**set**

**{**

**name = value;**

**}**

**get**

**{**

**return name;**

**}**

**}**

public Connection()

{

timer = new Timer(100);

timer.Elapsed += new ElapsedEventHandler(CheckForMessage);

}

public void Connect()

{

timer.Start();

}

public void Disconnect()

{

timer.Stop();

}

private static Random rnd = new Random();

**private void CheckForMessage**(**object source, ElapsedEventArgs e**)

{

Console.WriteLine("Checking for new messages");

if ((rnd.Next(9) == 0) && (MessageArraved != null))

**MessageArraved(this, new MessageArrivedEventArgs("Hello!"));**

}

}

public class Display

{

public void DisplayMessage(Connection sourse, MessageArrivedEventArgs e)

{

**Console.WriteLine("Поступило новое сообщение от: {0},", sourse.Name);**

**Console.WriteLine("текст сообщения: {0}", e.Message);**

}

}

class Program

{

static void Main(string[] args)

{

**Connection connect1 = new Connection();**

**connect1.Name = "First connection";**

**Connection connect2 = new Connection();**

**connect2.Name = "Second connection";**

**Display display = new Display();**

**connect1.MessageArraved += new MessageHandler (display.DisplayMessage);**

**connect2.MessageArraved += new MessageHandler (display.DisplayMessage);**

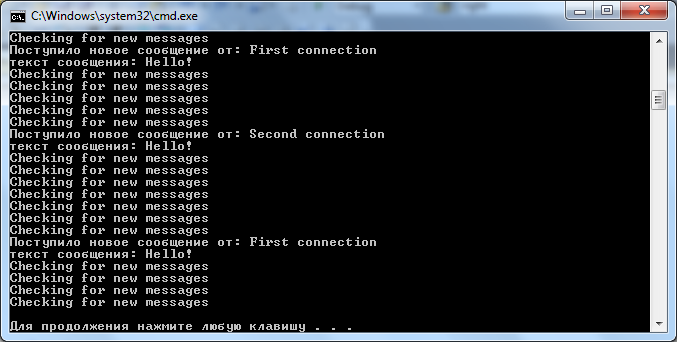
**connect1.Connect();**

**connect2.Connect();**

Console.ReadLine();

}

}



## Анонимные методы

Вместо определения методов для обработки событий, можно использовать анонимные методы. Анонимным методом называется метод, который не является методом какого-либо определенного класса. Анонимный метод создается для применения в качестве целевого метода для делегата.

Синтаксис создания анонимного метода, выглядит следующим образом:

delegate (параметры)

{

// Код анонимного метода.

};

На месте параметры указывается перечень параметров, соответствующих параметрам делегата, экземпляр которого создается, в том виде, в котором они должны использоваться в коде анонимного метода:

delegate (Connection source, MessageArrivedEventArgs e)

{

// Код анонимного метода, соответствующего событию MessageHandler

};

myConnectionl.MessageArrived +=

delegate(Connection source, MessageArrivedEventArgs e)

{

Console.WriteLine("Поступило новое сообщение от: {0},", source.Name);

Console.WriteLine("текст сообщения:: {0}", e.Message);

};

Анонимные методы являются локальными для того блока кода, в котором содержатся, и имеют доступ ко всем локальным переменным в этой области действия. В случае использования такой переменной она становится **внешней**. Внешние переменные не уничтожаются при выходе за пределы области действия, как это происходит с другими локальными переменными; они продолжают существовать до тех пор, пока не будут уничтожены методы, в которых они используются.

## Лямбда-выражения

Лямбда-выражения (lambda expressions) являются новой конструкцией в С# 3.0 и могут упрощать некоторые аспекты программирования на языке С#.

Лямбда-исчисление лежит в основе функциональных языков программирования.

*Это теория, которая рассматривает функции как правила, а не как графики. выражение в лямбда-исчислении представлено в префиксной форме: a+b ⬄ +(a,b)*

*λx. ( λ абстракция) – это выражение, определяющее функцию, где х – формальный параметр функции. Например λx.+х1 ⬄ x+1, +x1 – тело функции.*

*(λx.+х1) 4 ⬄ f(x){return x+1;}, f(4), т. е. 4 – фактический параметр функции.*

Лямбда-выражения можно использовать для упрощения синтаксиса анонимных методов.

Лямбда-выражение состоит из трех частей:

* список (нетипизированных) параметров в круглых скобках;
* операция =>;
* оператор на языке С#.

**Лямбда-выражения можно интерпретировать двумя способами:**

1. Лямбда-выражение используется в качестве делегата, т.е. лямбда-выражение присваивается переменной типа делегата (см. пример).
2. Лямбда-выражение интерпретируется как дерево выражения (используется в LINQ запросах).

### Использование лямбда-выражений для анонимных методов

Как правило, лямбда-выражения используются в сочетании с другими механизмами С#, например, с анонимными методами.

**Анонимным методом** называется метод, который не является методом какого-либо определенного класса. Анонимный метод создается для применения в качестве целевого метода для делегата.

При добавлении обработчика событий для какого-то события необходимо выполнить следующие шаги:

1. Определить метод обработчика событий, причем так, чтобы его возвращаемый тип и параметры совпадали с возвращаемым типом и параметрами делегата, требуемого для подписываемого события.

2. Объявить переменную с типом делегата, используемого для события.

3. Инициализировать переменную делегата экземпляром типа делегата, который ссылается на метод обработчика событий.

4. Добавить переменную делегата в список подписчиков события.

На практике обычно переменная для хранения делегата не используется, а экземпляра делегата просто используется при подписке на событие.

Timer myTimer = new Timer(100) ;

// подписка на событие Elapsed объекта Timer.

myTimer.Elapsed += new ElapsedEventHandler(WriteChar);

Это событие подразумевает использование делегата типа ElapsedEventHandler, экземпляр которого и создается с помощью идентификатора метода WriteChar. В результате при генерации объектом Timer события Elapsed будет вызываться метод WriteChar ().

Компилятор С# сможет достигать точно такого же результата и с помощью даже меньшего количества кода:

myTimer.Elapsed += WriteChar;

Компилятору С# известен тип делегата, который требуется событию Elapsed, поэтому он сможет самостоятельно добавить все недостающие сведения. Однако в большинстве случаев поступать подобным образом все же не рекомендуется, поскольку такой подход затрудняет чтение кода и не позволяет видеть, что именно в нем происходит.

При использовании анонимного метода приведенная выше последовательность шагов сокращается до одного единственного шага:

1. Использовать анонимный метод с возвращаемым типом и параметрами, соответствующими возвращаемому типу и параметрам делегата, который требуется для подписываемого события.

myTimer.Elapsed += **delegate**(object source, ElapsedEventArgs e)

{

Console.WriteLine("Обработчик события будет вызван через {0} миллисекунд.", (**source as Timer**).Interval);

};

Этот код будет работать так же, как и отдельный обработчик. Главное отличие состоит в том, что использованный здесь анонимный метод будет фактически скрыт от остальной части кода. То есть применять этот обработчик повторно, в другом месте приложения, не получится.

Лямбда-выражения можно использовать для упрощения синтаксиса анонимных методов.

myTimer.Elapsed += (source, e) **=>** Console.WriteLine("Обработчик события будет вызван через {0} миллисекунд.", (source as Timer).Interval);

**Пример.**

В этом примере лямбда-выражения применяются для генерации функций, которые используются для возврата результатов выполнения операции над двумя входными параметрами. Лямбда-выражения компилируются в методы, возвращаемый тип и параметры которых соответствуют типу делегата.

//делегат для представления метода

delegate int BynaryIntOperation(int paramA, int paramB);

class Program

{

//метод, параметром которого является делегат, т.е функция передается

//как параметр метода

static void PerformOperations(BynaryIntOperation del)

{

for (int valX = 1; valX <= 5; valX++)

{

for (int valY = 1; valY <= 5; valY++)

{

**int delResult = del(valX, valY);**

Console.Write("f({0},{1})={2}", valX, valY, delResult);

if (valY != 5) Console.Write(",");

}

Console.WriteLine();

}

}

static void Main(string[] args)

{

Console.WriteLine("f(a,b)=a+b");

//лямбда выражение

PerformOperations((**paramA, paramB) => paramA + paramB**);

Console.WriteLine();

//лямбда выражение

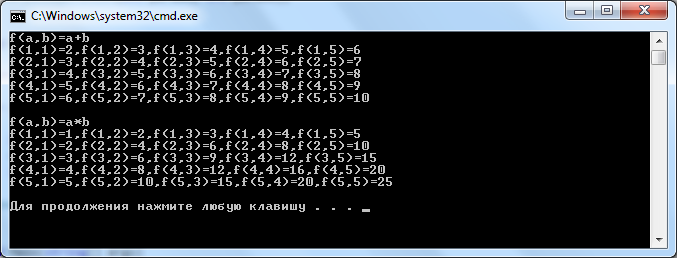
Console.WriteLine("f(a,b)=a\*b");

PerformOperations((**paramA, paramB) => paramA \* paramB**);

Console.WriteLine();

}

}



### Параметры лямбда-выражений

В рассмотренном выше примере типы операндов, используемых в лямбда-выражении не определялись:

PerformOperations((**paramA, paramB) => paramA + paramB**);

При желании их можно определять:

PerformOperations((int **paramA, int paramB) => paramA + paramB**);

При таком подходе код получается более наглядным, но утрачивает краткость и гибкость. Лямбда-выражения с неявным определением типов из предыдущего примера можно было бы использовать для типов делегатов с другими числовыми типами, например, long.

Списки параметров в лямбда-выражениях всегда состоят из разделенного запятыми перечня либо всех неявно типизированных, либо всех явно типизированных параметров.

PerformOperations((int **paramA, paramB) => paramA + paramB**); - ошибка, т.к. один параметр определен явно, а тип другого нужно определять компилятору.

Если параметр только один, тогда круглые скобки могут опускаться; в противном случае они являются обязательной частью списка параметров.

**paramA => paramA + paramА**;

Можно определять лямбда-выражения, не имеющие параметров:

**() => Math.PI;**

### Тело операторов лямбда-выражений

Во всем приводившемся пока что коде в теле оператора лямбда-выражений использовалось единственное выражение. Это выражение воспринималось как возвращаемое значение лямбда-выражения.

Также в лямбда-выражении можно использовать несколько операторов:

**(paraml, param2) =>**

**{**

**// Здесь идет несколько операторов.**

**[return результат]//если результат отличен от void**

**}**

Например, для предыдущего примера:

**//1 способ**

**PerformOperations(delegate(int paramA, int paramB)**

**{**

**return paramA + paramB;**

**});**

**//2 способ**

**PerformOperations((paramA, paramB) =>**

**{**

**return paramA + paramB;**

**});**

### Лямбда-выражения и события

Покажем связь между делегатами, событиями и лямбда-выражениями.

Рассмотрим пример с генерацией событий.

1. **Использование делегатов.**

class Program

{

class Car

{

const int MAXSPEED = 200;

public string Name { get; set; }

public int currSpeed { get; set; }

private bool carIsDead = false;

public Car()

{

Name = "NoName";

currSpeed = 0;

}

public Car(string name, int year, int speed)

{

Name = name;

currSpeed = speed;

}

public override string ToString()

{

if (carIsDead)

return Name + ", " + ", " + currSpeed;

else return "Exploded";

}

public void SpeedUp(int delta)

{

currSpeed += delta;

}

// 1. Определить тип делегата.

**public delegate void CarEngineHandler(string msgForCaller);**

//2. Определить переменную-член типа этого делегата.

**private CarEngineHandler listOfHandlers;**

// 3. Добавить регистрационную функцию для вызывающего кода.

**public void RegisterWithCarEngine(CarEngineHandler methodToCall)**

**{**

**listOfHandlers = methodToCall;**

**}**

public void Accelerate(int delta)

{

if (carIsDead)

{

if (listOfHandlers != null)

listOfHandlers("ОЙ! Машина сломалась...");

}

else

{

currSpeed += delta;

// Автомобиль почти сломан?

if ((MAXSPEED - currSpeed)<10 && listOfHandlers != null)

{

listOfHandlers("Внимание! Скоро взорвется!");

}

if (currSpeed >= MAXSPEED)

carIsDead = true;

else

Console.WriteLine("CurrentSpeed = {0}", currSpeed);

}

}

}

static void Main(string[] args)

{

Car cl = new Car("MyCar", 2010, 60);

// сообщим ему, какой метод вызывать, когда он захочет //отправить сообщение.

**cl.RegisterWithCarEngine(new Car.CarEngineHandler (OnCarEngineEvent));**

Console.WriteLine("\*\*\*\*\* Поехали! \*\*\*\*\*");

for (int i = 0; i < 8; i++)

**cl.Accelerate(20);** }

//метод адрес которого хранится в делегате

public static void OnCarEngineEvent(string msg)

{

Console.WriteLine("\n\*\*\*\*\* Сообщение от Car Object \*\*\*\*\*");

Console.WriteLine("=> {0}", msg); Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n" );

}

}

}

1. **Используем делегат и события**

class Car

{

const int MAXSPEED = 200;

public string Name { get; set; }

public int currSpeed { get; set; }

private bool carIsDead = false;

public Car()

{

Name = "NoName";

currSpeed = 0;

}

public Car(string name, int year, int speed)

{

Name = name;

currSpeed = speed;

}

public override string ToString()

{

if (carIsDead)

return Name + ", " + ", " + currSpeed;

else return "Expoded";

}

public void SpeedUp(int delta)

{

currSpeed += delta;

}

// Этот делегат работает в сочетании с событиями Саг.

public delegate void CarEngineHandler(string msg);

// Car может посылать следующие события.

**public event CarEngineHandler Exploded;//взорвалась**

**public event CarEngineHandler AboutToBlow;** public void Accelerate(int delta)

{

// Если автомобиль сломан, инициировать событие Exploded.

if (carIsDead)

{

**if (Exploded != null)**

**Exploded("Sorry, this car is dead...");**

}

else

{

currSpeed += delta;

// Почти сломан?

**if (20 == MAXSPEED - currSpeed**

**&& AboutToBlow != null)**

**{**

**AboutToBlow("Внимание!!! Скоро взорвется!!!");**

}

if (currSpeed >= MAXSPEED) carIsDead = true;

else

Console.WriteLine("CurrentSpeed = {0}", currSpeed);

}

}

}

static void Main(string[] args)

{

Car cl = new Car("MyCar", 2010, 60);

// Зарегистрировать обработчики событий.

**cl.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);**

**cl.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);**

**cl.Exploded += new Car.CarEngineHandler(CarAboutToBlow);**

Console.WriteLine("\n\*\*\*\*\* Speeding up \*\*\*\*\*");

for (int i = 0; i < 8; i++)

cl.Accelerate(20);

}

public static void CarIsAlmostDoomed(string msg)

{

Console.WriteLine(msg);

}

public static void CarExploded(string msg)

{

Console.WriteLine(msg);

}

public static void CarAboutToBlow(string msg)

{

Console.WriteLine("Critical Message: {0}", msg);

}

1. Используем события .Net

**public class CarEventArgs : EventArgs**

**{**

**public readonly string msg;**

**public CarEventArgs(string message)**

**{**

**msg = message;**

**}**

**}**

class Car

{

const int MAXSPEED = 200;

public string Name { get; set; }

public int currSpeed { get; set; }

private bool carIsDead = false;

public Car()

{

Name = "NoName";

currSpeed = 0;

}

public Car(string name, int year, int speed)

{

Name = name;

currSpeed = speed;

}

public override string ToString()

{

if (carIsDead)

return Name + ", " + ", " + currSpeed;

else return "Expoded";

}

public void SpeedUp(int delta)

{

currSpeed += delta;

if (currSpeed > MAXSPEED) carIsDead = true;

}

// Этот делегат работает в сочетании с событиями Саг.

**public delegate void CarEngineHandler(object sender, CarEventArgs e);**

// Car может посылать следующие события.

public event CarEngineHandler Exploded;//взорвалась

public event CarEngineHandler AboutToBlow; //сообщение о взрыве

public void Accelerate(int delta)

{

// Если автомобиль сломан, инициировать событие Exploded.

if (carIsDead)

{

if (Exploded != null)

Exploded(**this, new CarEventArgs("Sorry, this car is dead..."**));

}

else

{

currSpeed += delta;

// Почти сломан?

if (20 == MAXSPEED - currSpeed

&& AboutToBlow != null)

{

AboutToBlow(**this, new CarEventArgs("Внимание!!! Скоро взорвется!!!"**));

}

if (currSpeed >= MAXSPEED)

carIsDead = true;

else

Console.WriteLine("CurrentSpeed = {0}", currSpeed);

}

}

}

static void Main(string[] args)

{

Car cl = new Car("Nissan", 2010, 60);

**cl.AboutToBlow += new Car.CarEngineHandler(CarAboutToBlow);**

**cl.AboutToBlow += new Car.CarEngineHandler(CarIsAlmostDoomed);**

**cl.Exploded += new Car.CarEngineHandler(CarExploded);**

Console.WriteLine("\*\*\*\*\* Speeding up \*\*\*\*\*");

for (int i = 0; i < 8; i++)

cl.Accelerate(20);

}

public static void CarAboutToBlow(object sender, CarEventArgs e)

{

// произведем проверку во время выполнения перед приведением.

if (sender is Car)

{

**Car c = (Car)sender;**

Console.WriteLine(" {0}: {1}", c.Name, e.msg);

}

}

public static void CarIsAlmostDoomed(object sender, CarEventArgs e)

{

**Car c = (Car)sender;**

Console.WriteLine("Critical Message from {0}: {1}", c.Name, e.msg);

}

public static void CarExploded(object sender, CarEventArgs e)

{

**Car c = (Car)sender;**

Console.WriteLine("Critical Message from {0}: {1}", c.Name, e.msg);

}

1. Анонимные методы и лямбда выражения (изменения только в функции Main)

//анонимные методы и лямбда выражения

static void Main(string[] args)

{

Car cl = new Car("Nissan", 2010, 60);

**int aboutToBlowCounter = 0;** //внешняя переменная

//анонимный метод

**cl.AboutToBlow += delegate**

**{**

**Console.WriteLine("ОЙ! Вы едете слишком быстро!!! ");**

**aboutToBlowCounter++;**

**};**

//лямбда выражение

**cl.AboutToBlow += (object sender, CarEventArgs e) =>**

**{**

**Car c = (Car)sender;**

**Console.WriteLine("Critical Message from {0}: {1}", c.Name, e.msg);**

**aboutToBlowCounter++;**

**};**

**cl.Exploded += new Car.CarEngineHandler(CarExploded);**

Console.WriteLine("\*\*\*\*\* Speeding up \*\*\*\*\*");

for (int i = 0; i < 8; i++)

cl.Accelerate(20);

Console.WriteLine("Метод AboutToBlow вызывается {0} раз", aboutToBlowCounter);

}

public static void CarExploded(object sender, CarEventArgs e)

{

Car c = (Car)sender;

Console.WriteLine("Critical Message from {0}: {1}", c.Name, e.msg);

}

### Лямбда-выражения и коллекции

Лямбда-выражение с параметрами можно представлять в виде одного из следующих обобщенных типов, которые определены в пространстве имен System.

* Action<> для лямбда-выражений без параметров и с возвращаемым типом void.
* Action<> для лямбда выражений с параметрами и возвращаемым типом void.
* Func<> для лямбда-выражений с параметрами в количестве вплоть до четырех и возвращаемым типом, который не является типом void.

В С# 3.0 в Action<> было предусмотрено вплоть до четырех параметров обобщенного типа, по одному для каждого параметра в лямбда-выражении, а в Func<> — вплоть до пяти параметров обобщенного типа, т.е. четыре для возможных четырех параметров в лямбда-выражении и один для возвращаемого типа в нем. В Func<> возвращаемый тип всегда идет последним в списке.

Например:

1. Func<string,double,int> является эквивалентом типа делегата в следующей форме:

**public delegate int SomeDelegate(string arg1, double arg2)**

и может быть представлено в виде лямбда-выражения

**(string arg1, double arg2)=>(int)(arg1.Length\*arg2);**

1. Func<int, int, int> может быть представлено в виде делегата

**public delegate int SomeDelegate( int arg1, int arg2)**

или лямбда-выражения

**(int arg1, int arg2) => arg1 + arg2;**

В версии .NET 4 семейства делегатов Action<...> и Func<...> расширены и принимают вплоть до 16 аргументов.

Использование анонимного метода для создания делегата:

Func<string,int> returnLength;

**returnLength = delegate (string text){return text.Length; };**

Console.WriteLine(returnLength("Hello"));

Использование лямбда-выражения:

Func<string,int> returnLength;

**returnLength = (string text) => { return text.Length; };**

Console.WriteLine(returnLength("Hello"));

Сокращенный вариант:

Func<string,int> returnLength;

returnLength = text => text.Length;

Console.WriteLine(returnLength("Hello"));

**Фильтрация, сортировка и действия на списках**

Для типа List<T> определены методы:

* FindAll() – принимает предикат Predicate<T> и возвращает новый список с элементами исходного списка, соответствующими предикату.
* Sort() принимает экземпляр Comparison<T> и сортирует список указанным
* образом.
* метод ForEach() принимает действие Action<T> и выполняет его над каждым элементом.

**Predicate<T>** - это делегат, предоставляющий метод, в котором задан набор критериев, он позволяет определить, удовлетворяет ли этим критериям заданный объект:

public delegate bool Predicate<in T>(T obj);

Т – тип сравниваемого объекта, можно использовать либо указанный тип, либо более общий тип (контрвариантность). Этот делегат используется несколькими методами классов Array и List<T> для поиска элементов в коллекции. Как правило, делегат Predicate<T> представляется лямбда-выражением.

**Comparison<T>** - делегат, предоставляет метод для сравнения двух объектов типа Т:

public delegate int Comparison<in T>(T x, T y);

**Action<T>** - делегат, который принимает один параметр и не возвращает значений.

class Film

{

public string Name { get; set; }

public int Year { get; set; }

public Film()

{

Name = "NoName";

Year = 2015;

}

public Film(string name, int year)

{

Name = name;

Year = year;

}

public override string ToString()

{

return Name+", "+Year.ToString();

}

}

class Program

{

static void Main(string[] args)

{

List<Film> films = new List<Film>

{

new Film { Name = "Jaws", Year = 1975 },

new Film { Name = "Singing in the Rain", Year = 1952 },

new Film { Name = "Some like it Hot", Year = 1959 },

new Film { Name = "The Wizard of Oz", Year = 1939 },

new Film { Name = "It’s a Wonderful Life", Year = 1946 },

new Film { Name = "American Beauty", Year = 1999 },

new Film { Name = "High Fidelity", Year = 2000 },

new Film { Name = "The Usual Suspects", Year = 1995 }

};

//создание многократно используемого делегата для вывода списка на консоль

Action<Film> print = film => Console.WriteLine(film.ToString());

films.ForEach(print);//вывод списка

Console.WriteLine("\nFiltred List");

films.FindAll(film => film.Year < 1960).ForEach(print);//вывод отфильтрованного списка

Console.WriteLine("\nSorted List");

films.Sort((f1, f2) => f1.Name.CompareTo(f2.Name));//сортировка списка

films.ForEach(print);

}

}

Рассмотрим методы расширения, которые предлагаются в пространстве имен **System.Linq** для типов-массивов на примере метода **Aggregate** (аккумулирующая функция).

Метод Aggregate() имеет три перегрузки:

public static TSource Aggregate<TSource>(thisIEnumerable<TSource> source,

Func<TSource, TSource, TSource> func);

Аккумулирующая функция (может предоставляться в виде лямбда-выражения) будет применяться для каждой пары элементов в коллекции от начала до конца, причем результат каждой операции будет являться входными данными следующей операции вычисления.

Например,

int[] intArray = { 1, 2, 5 };

Console.WriteLine(intArray.Aggregate<int>((a, b) => a + b));

Результатом будет сумма элементов массива. Первый раз функция выполняет вычисления с параметрами 1 и 2, получает 3. Второй раз выполняется 3+5 =8.

**Пример 1:**

class Program

{

static void Main(string[] args)

{

string[] strArray = { "Masha", "Sasha", "Pasha" };//коллекция

Console.WriteLine(strArray.Aggregate((a,b)=>a+" "+b));

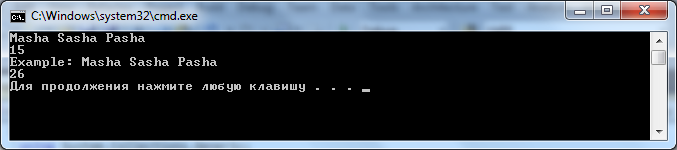
Console.WriteLine(strArray.Aggregate **<string, int**> (0, (a, b) => a + b.Length));

Console.WriteLine(strArray.Aggregate **<string, string, string>** ("Example:", (a, b) => a + " " + b,a=>a));

Console.WriteLine(strArray.Aggregate **<string, string, int>** ("Example:", (a, b) => a + " " + b, a=>a.Length));

}

}

****