№9 Делегаты, события и лямбда выражения

**Курносенко Софья**

**Задание**

1. Используя делегаты (множественные) и события промоделируйте ситуации, приведенные в таблице ниже. Можете добавить новые типы (классы), если существующих недостаточно. При реализации методов везде, где возможно, использовать лямбда-выражения.

|  |  |
| --- | --- |
| 5, 6 | Создать класс *Игра* с событиями *Атака* и *Лечить*. В main создать некоторое количество игровых объектов*.* Подпишите объекты на события произвольным образом. Реакция на события у разных объектов может быть разной (без изменения, увеличивается / уменьшается уровень жизни). Проверить состояния игровых объектов после наступления событий, возможно, неоднократном. |

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/

https://docs.microsoft.com/en-us/dotnet/csharp/programmingguide/statements-expressions-operators/lambda-expressions

2. Создайте пять методов пользовательской обработки строки (например, удаление знаков препинания, добавление символов, замена на заглавные, удаление лишних пробелов и т.п.). Используя стандартные типы делегатов (Action, Func) организуйте алгоритм последовательной обработки строки написанными вами методами. Далее приведен перечень заданий.

https://docs.microsoft.com/enus/dotnet/api/system.action?view=netframework-4.8

https://docs.microsoft.com/en-us/dotnet/api/system.func1?view=netframework-4.8

https://docs.microsoft.com/en-us/dotnet/api/system.predicate1?view=netframework-4.8 3

# Делегаты, события и лямбды

## ► Делегаты

**Делегат – это тип, способный хранить ссылки на методы.** То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

**Определение делегатов**

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры. Например:

delegate void Message();

Делегат Message в качестве возвращаемого типа имеет тип void (то есть ничего не возвращает) и не принимает никаких параметров. Это значит, что этот делегат может указывать на любой метод, который не принимает никаких параметров и ничего не возвращает.

Рассмотрим применение этого делегата:

class Program

{

delegate void Message(); // 1. Объявляем делегат

static void Main(string[] args)

{

Message mes; // 2. Создаем переменную делегата

if (DateTime.Now.Hour < 12)

{

mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода

}

else

{

mes = GoodEvening;

}

mes(); // 4. Вызываем метод

Console.ReadKey();

}

private static void GoodMorning()

{

Console.WriteLine("Good Morning");

}

private static void GoodEvening()

{

Console.WriteLine("Good Evening");

}

}

Здесь сначала мы определяем делегат:

delegate void Message(); // 1. Объявляем делегат

В данном случае делегат определяется внутри класса, но также можно определить делегат вне класса внутри пространства имен.

Для использования делегата объявляется переменная этого делегата:

Message mes; // 2. Создаем переменную делегата

С помощью свойства DateTime.Now.Hour получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода. Обратите внимание, что эти **методы имеют** **то же возвращаемое значение и тот же набор параметров** (в данном случае отсутствие параметров), **что и делегат**.

mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода

Затем через делегат вызываем метод, на который ссылается данный делегат:

mes(); // 4. Вызываем метод

Вызов делегата производится подобно вызову метода.

Посмотрим на примере другого делегата:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

// присваивание адреса метода через контруктор

Operation del = Add; // делегат указывает на метод Add

int result = del(4, 5); // фактически Add(4, 5)

Console.WriteLine(result);

del = Multiply; // теперь делегат указывает на метод Multiply

result = del(4, 5); // фактически Multiply(4, 5)

Console.WriteLine(result);

Console.Read();

}

private static int Add(int x, int y)

{

return x + y;

}

private static int Multiply(int x, int y)

{

return x \* y;

}

}

В данном случае **делегат** Operation **возвращает значение** типа int и имеет два параметра типа int. Поэтому этому делегату соответствует любой метод, который возвращает значение типа int и принимает два параметра типа int. В данном случае это методы Add и Multiply. То есть мы можем присвоить переменной делегата любой из этих методов и вызывать.

Поскольку делегат принимает два параметра типа int, то при его вызове необходимо передать значения для этих параметров: del(4,5).

**Пример**

Если делегат, а точнее метод, ссылку на который он хранит, возвращает значение, то при множественном делегировании во внимание принимается значение возвращенное последним методом в списке делегата. То есть, значение, которое вернет именно последний метод, мы сможем присвоить переменной вывести или в консоль при вызове переменной делегата.

class Program

{

delegate int myDelegate(int a);

static void Main(string[] args)

{

myDelegate del = ShowPrice\_1;

del += ShowPrice\_2;

**Console.WriteLine(del(7));**

}

public static int ShowPrice\_1(int price)

{

Console.WriteLine("\*ShowPrice\_1\*");

return price + 1;

}

public static int ShowPrice\_2(int price)

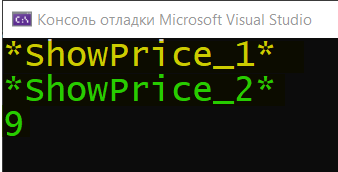
{

Console.WriteLine("\*ShowPrice\_2\*");

return price + 2;

}

}



Делегаты необязательно могут указывать только на методы, которые определены в том же классе, где определена переменная делегата. Это могут быть также методы из других классов и структур.

class Math

{

public int Sum(int x, int y) { return x + y; }

}

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Math math = new Math();

Operation del = math.Sum;

int result = del(4, 5); // math.Sum(4, 5)

Console.WriteLine(result); // 9

Console.Read();

}

}

#### Присвоение ссылки на метод

Выше переменной делегата напрямую присваивался метод. Есть еще один способ - создание объекта делегата с помощью конструктора, в который передается нужный метод:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation del = Add; //прямое присваивание метода

Operation del2 = new Operation(Add); //через конструктор

Console.Read();

}

private static int Add(int x, int y) { return x + y; }

}

Оба способа равноценны.

#### Соответствие методов делегату

Как было написано выше, методы соответствуют делегату, если они имеют один и тот же возвращаемый тип и один и тот же набор параметров. Но надо учитывать, что во внимание также принимаются модификаторы **ref** и **out**. Например, пусть у нас есть делегат:

delegate **void** SomeDel(**int** a, **double** b);

Этому делегату соответствует, например, следующий метод:

**void** SomeMethod1(**int** g, **double** n) { }

А следующие методы НЕ соответствуют:

**int** SomeMethod2(int g, double n) { }

void SomeMethod3(**double** n, **int** g) { }

void SomeMethod4(**ref** int g, double n) { }

void SomeMethod5(**out** int g, double n) { g = 6; }

Здесь метод SomeMethod2 имеет другой возвращаемый тип, отличный от типа делегата. SomeMethod3 имеет другой набор параметров. Параметры SomeMethod4 и SomeMethod5 также отличаются от параметров делегата, поскольку имеют модификаторы ref и out.

#### Добавление методов в делегат

В примерах выше переменная делегата указывала на один метод. В реальности же **делегат может указывать на множество методов, которые имеют ту же сигнатуру и возвращаемые тип.** Все методы в делегате попадают в специальный список - список вызова или invocation list. И при вызове делегата все методы из этого списка последовательно вызываются. И мы можем добавлять в этот спиок не один, а несколько методов:

class Program

{

delegate void Message();

static void Main(string[] args)

{

Message mes1 = Hello;

mes1 += HowAreYou; // теперь mes1 указывает на два метода

mes1(); // вызываются оба метода - Hello и HowAreYou

Console.Read();

}

private static void Hello()

{

Console.WriteLine("Hello");

}

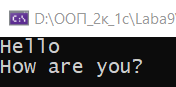
private static void HowAreYou()

{

Console.WriteLine("How are you?");

}

}



В данном случае в список вызова делегата mes1 добавляются два метода - Hello и HowAreYou. И при вызове mes1 вызываются сразу оба этих метода.

Для добавления делегатов применяется операция **+=**. Однако стоит отметить, что в реальности будет происходить создание нового объекта делегата, который получит методы старой копии делегата и новый метод, и новосозданный объект делегата будет присвоен переменной mes1.

При добавлении делегатов следует учитывать, что мы можем добавить ссылку на один и тот же метод несколько раз, и в списке вызова делегата тогда будет несколько ссылок на один и то же метод. Соответственно при вызове делегата добавленный метод будет вызываться столько раз, сколько он был добавлен:

|  |
| --- |
| Message mes1 = Hello;  mes1 += HowAreYou;  mes1 += Hello;  mes1 += Hello;    mes1(); |

Консольный вывод:

Hello

How are you?

Hello

Hello

Подобным образом мы можем удалять методы из делегата с помощью операции **-=**:

static void Main(string[] args)

{

Message mes1 = Hello;

mes1 += HowAreYou;

mes1(); // вызываются все методы из mes1

mes1 -= HowAreYou; // удаляем метод HowAreYou

mes1(); // вызывается метод Hello

Console.Read();

}

При удалении методов из делегата фактически будет создаватья новый делегат, который в списке вызова методов будет содержать на один метод меньше.

При удалении следует учитывать, что если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение. Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.

#### Объединение делегатов

Делегаты можно объединять в другие делегаты. Например:

class Program

{

delegate void Message();

static void Main(string[] args)

{

Message mes1 = Hello;

Message mes2 = HowAreYou;

Message mes3 = mes1 + mes2; // объединяем делегаты

mes3(); // вызываются все методы из mes1 и mes2

Console.Read();

}

private static void Hello()

{

Console.WriteLine("Hello");

}

private static void HowAreYou()

{

Console.WriteLine("How are you?");

}

}

В данном случае объект mes3 представляет объединение делегатов mes1 и mes2. Объединение делегатов значит, что в список вызова делегата mes3 попадут все методы из делегатов mes1 и mes2. И при вызове делегата mes3 все эти методы одновременно будут вызваны.

### Вызов делегата

В примерах выше делегат вызывался как обычный метод. Если делегат принимал параметры, то при ее вызове для параметров передавались необходимые значения:

class Program

{

delegate int Operation(int x, int y);

delegate void Message();

static void Main(string[] args)

{

Message mes = Hello;

mes();

Operation op = Add;

op(3, 4);

Console.Read();

}

private static void Hello() { Console.WriteLine("Hello"); }

private static int Add(int x, int y) { return x + y; }

}

Другой способ вызова делегата представляет метод **Invoke()**:

class Program

{

delegate int Operation(int x, int y);

delegate void Message();

static void Main(string[] args)

{

Message mes = Hello;

mes.Invoke();

Operation op = Add;

op.Invoke(3, 4);

Console.Read();

}

private static void Hello() { Console.WriteLine("Hello"); }

private static int Add(int x, int y) { return x + y; }

}

Если делегат принимает параметры, то в метод Invoke передаются значения для этих параметров.

Следует учитывать, что если делегат пуст, то есть в его списке вызова нет ссылок ни на один из методов (то есть делегат равен Null), то при вызове такого делегата мы получим исключение, как, например, в следующем случае:

Message mes = null;

//mes(); // ! Ошибка: делегат равен null

Operation op = Add;

op -= Add; // делегат op пуст

op(3, 4); // !Ошибка: делегат равен null

Поэтому при вызове делегата всегда лучше проверять, не равен ли он null. Либо можно использовать метод Invoke и оператор условного null:

Message mes = null;

mes?.Invoke(); // ошибки нет, делегат просто не вызывается

Operation op = Add;

op -= Add; // делегат op пуст

op?.Invoke(3, 4); // ошибки нет, делегат просто не вызывается

Если делегат возвращает некоторое значение, то возвращается значение последнего метода из списка вызова (если в списке вызова несколько методов). Например:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation op = Subtract;

op += Multiply;

op += Add;

Console.WriteLine(op(7, 2)); // Add(7,2) = 9

Console.Read();

}

private static int Add(int x, int y) { return x + y; }

private static int Subtract(int x, int y) { return x - y; }

private static int Multiply(int x, int y) { return x \* y; }

}

### Делегаты как параметры методов

Также делегаты могут быть параметрами методов:

class Program

{

delegate void GetMessage();

static void Main(string[] args)

{

if (DateTime.Now.Hour < 12)

{

Show\_Message(GoodMorning);

}

else

{

Show\_Message(GoodEvening);

}

Console.ReadLine();

}

private static void Show\_Message(GetMessage \_del)

{

\_del?.Invoke();

}

private static void GoodMorning()

{

Console.WriteLine("Good Morning");

}

private static void GoodEvening()

{

Console.WriteLine("Good Evening");

}

}

### Обобщенные делегаты

Делегаты могут быть обобщенными, например:

delegate T Operation<T, K>(K val);

class Program

{

static void Main(string[] args)

{

Operation<decimal, int> op = Square;

Console.WriteLine(op(5));

Console.Read();

}

static decimal Square(int n)

{

return n \* n;

}

}

## Свойства делегатов

→ Наследовать от делегата нельзя

→ Объявление делегата можно размещать непосредственно в пространстве имен или внутри класса (в любом месте, где может быть определен класс)



→ Тип делегата определяется его именем

→ Сигнатура метода должна совпадать с сигнатурой делегата

→ Делегаты допускают передачу методов в качестве параметров

→ Делегаты могут ссылаться на статические методы

→ Делегаты могут быть параметрами методов

→ Делегат можно вызывать как обычный метод



**Операции над делегатами**

▪ можно сравнивать на равенство и неравенство не содержат ссылок или если ссылки на одни и те же методы в одном и том же порядке

▪ выполнять операции простого и составного присваивания

del += HelloI; // добавляем делегат

del -= HelloI; // удаляем делегат

▪ является неизменяемым типом данных, поэтому при любом изменении создается новый экземпляр, а старый впоследствии удаляется сборщиком мусора.

**Виды делегатов**

◦ единичный – ссылается на один метод

◦ множественный (многоадресный) – ссылается на несколько методов

## ► Применение делегатов

class Account

{

int \_sum; // Переменная для хранения суммы

public Account(int sum)

{

\_sum = sum;

}

public int CurrentSum

{

get { return \_sum; }

}

public void Put(int sum)

{

\_sum += sum;

}

public void Withdraw(int sum)

{

if (sum <= \_sum)

{

\_sum -= sum;

}

}

}

Допустим, в случае вывода денег с помощью метода Withdraw нам надо как-то уведомлять об этом самого клиента и, может быть, другие объекты. Для этого создадим делегат **AccountStateHandler**. Чтобы использовать делегат, нам надо создать переменную этого делегата, а затем присвоить ему метод, который будет вызываться делегатом.

Итак, добавим в класс Account следующие строки:

class Account

{

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Создаем переменную делегата

AccountStateHandler \_del;

// Регистрируем делегат

public void RegisterHandler(AccountStateHandler del)

{

\_del = del;

}

// Далее остальные строки класса Account

Теперь изменим метод Withdraw следующим образом:

public void Withdraw(int sum)

{

if (sum <= \_sum)

{

\_sum -= sum;

if (\_del != null)

\_del($"Сумма {sum} снята со счета");

}

else

{

if (\_del != null)

\_del("Недостаточно денег на счете");

}

}

Теперь при снятии денег через метод Withdraw мы сначала проверяем, имеет ли делегат ссылку на какой-либо метод (иначе он имеет значение null). И если метод установлен, то вызываем его, передавая соответствующее сообщение в качестве параметра.

Теперь протестируем класс в основной программе:

class Program

{

static void Main(string[] args)

{

// создаем банковский счет

Account account = new Account(200);

// Добавляем в делегат ссылку на метод Show\_Message

// а сам делегат передается в качестве параметра метода RegisterHandler

account.RegisterHandler(Show\_Message);

// Два раза подряд пытаемся снять деньги

account.Withdraw(100);

account.Withdraw(150);

Console.ReadLine();

}

private static void Show\_Message(String message)

{

Console.WriteLine(message);

}

}

Запустив программу, мы получим два разных сообщения:

Сумма 100 снята со счета

Недостаточно денег на счете

Таким образом, мы создали механизм обратного вызова для класса Account, который срабатывает в случае снятия денег. Поскольку делегат объявлен внутри класса Account, то чтобы к нему получить доступ, используется выражение Account.AccountStateHandler.

Опять же может возникнуть вопрос: почему бы в коде метода Withdraw() не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда у нас есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как эти классы будут использоваться. А использовать эти классы будет другой разработчик.

Так, здесь мы выводим сообщение на консоль. Однако для класса Account не важно, как это сообщение выводится. Классу Account даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

В результате, если мы создаем консольное приложение, мы можем через делегат выводить сообщение на консоль. Если мы создаем графическое приложение Windows Forms или WPF, то можно выводить сообщение в виде графического окна. А можно не просто выводить сообщение. А, например, записать при списании информацию об этом действии в файл или отправить уведомление на электронную почту. В общем любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса Account.

Хотя в примере наш делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько методов. Кроме того, при необходимости мы можем удалить ссылки на адреса определенных методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе Account метод RegisterHandler и добавим новый метод UnregisterHandler, который будет удалять методы из списка методов делегата:

// Регистрируем делегат

public void RegisterHandler(AccountStateHandler del)

{

\_del += del; // добавляем делегат

}

// Отмена регистрации делегата

public void UnregisterHandler(AccountStateHandler del)

{

\_del -= del; // удаляем делегат

}

В первом методе делегаты \_del и del  объединяются в один, который потом присваивается переменной \_del. Во втором методе удаляется делегат del. Теперь перейдем к основной программе:

class Program

{

static void Main(string[] args)

{

Account account = new Account(200);

Account.AccountStateHandler colorDelegate = new Account.AccountStateHandler(Color\_Message);

// Добавляем в делегат ссылку на методы

account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));

account.RegisterHandler(colorDelegate);

// Два раза подряд пытаемся снять деньги

account.Withdraw(100);

account.Withdraw(150);

// Удаляем делегат

account.UnregisterHandler(colorDelegate);

account.Withdraw(50);

Console.ReadLine();

}

private static void Show\_Message(String message)

{

Console.WriteLine(message);

}

private static void Color\_Message(string message)

{

// Устанавливаем красный цвет символов

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine(message);

// Сбрасываем настройки цвета

Console.ResetColor();

}

}

В метод мы можем передать как объект делегата с помощью конструктора, то есть через new:

account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));

так и сам метод:

account.RegisterHandler(Show\_Message);

В целях тестирования мы создали еще один метод - Color\_Message, который выводит то же самое сообщение только красным цветом. Для первого делегата создается отдельная переменная. Но большой разницы между передачей обоих в метод account.RegisterHandler нет: просто в одном случае мы сразу передаем объект, создаваемый конструктором account.RegisterHandler(new Account.AccountStateHandler(Show\_Message));

Во втором случае создаем переменную и ее уже передаем в метод account.RegisterHandler(colorDelegate);.

В строке account.UnregisterHandler(colorDelegate); этот метод удаляется из списка вызовов делегата, поэтому этот метод больше не будет срабатывать. Консольный вывод будет иметь следующую форму:

Сумма 100 снята со счета

Сумма 100 снята со счета

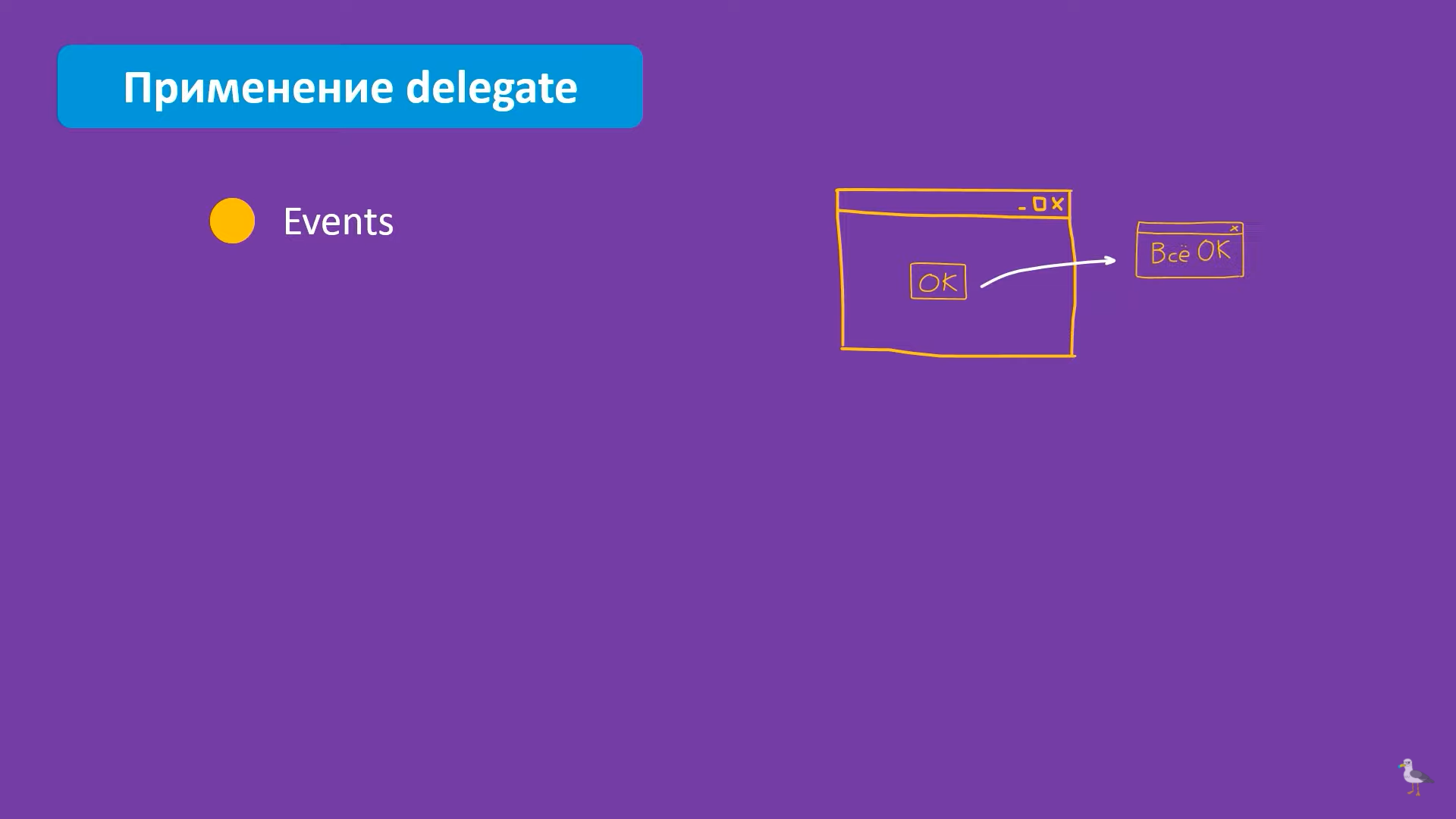
Недостаточно денег на счете

Недостаточно денег на счете

Сумма 50 снята со счета

## Мы можем применять делегаты с событиями

Допустим у нас есть окно с кнопкой “OK”, при нажатии на кнопку мы выводим сообщение “Всё OK”.



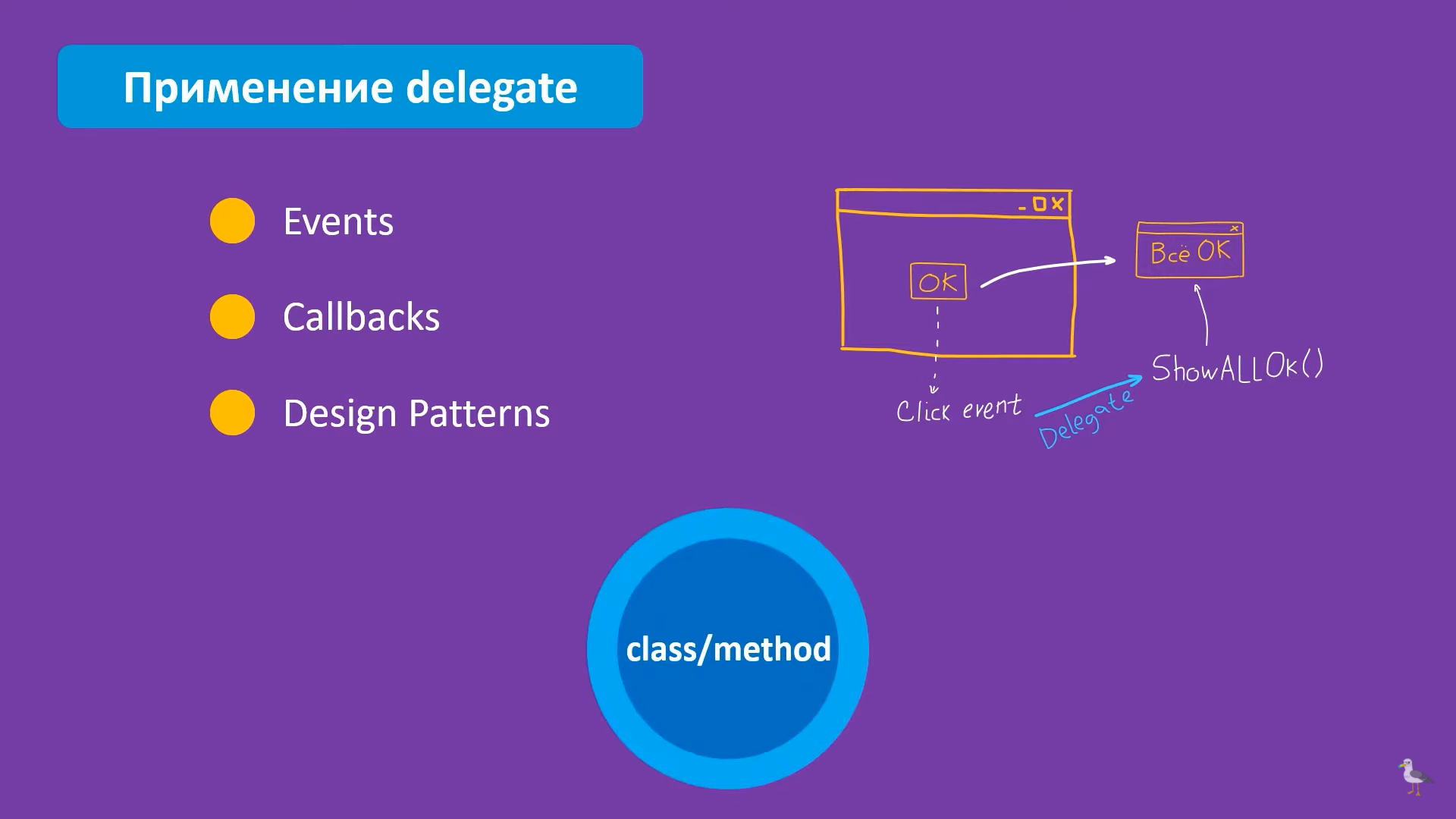
У кнопки есть стандартное событие “click”. У нас же есть метод “ShowAllOk”, который выводит сообщение “Всё OK”.

Вопрос: как связать событие и метод?

Ответ: событию нужна ссылка на метод, эту ссылку обеспечивает делегат.

## 

То есть нам дали [разработчики Microsoft] окно с кнопкой, на которую навешано событие “click”. А вот мы решаем, что эта кнопка будет делать, что зависит от метода.



**Пример**

Нам нужно создать класс, в котором есть метод, который подключается к бирже каждые 2 секунды и в режиме реального времени в одну из своих переменных записывает текущую цену. Мы же значение в этой переменной обработаем по своему усмотрению.

using System;

using System.Threading;

namespace StockOnliner

{

//класс для получения актуальных цен

public class StockExchangeMonitor

{

public delegate void PriceChange(int price);

//свойство PriceChangeHandler типа делегата PriceChange будет в дальнейшем содержать ссылку на метод(ы)

//определенные другим разработчиком (тем, кто не может вмешаться в логику класса StockExchangeMonitor)

public PriceChange PriceChangeHandler { get; set; }

//начинаем получать цены с биржи

public void Start()

{

//цены получаем безостановочно

while (true)

{

//ни к какой бирже мы, конеччно, подключаться не будем, "актуальные цены" сформирует рандомайзер

int bankOfAmericaPrice = (new Random()).Next(100);

//свойство PriceChangeHandler типа делегата PriceChange отправляет "полученное с биржи" значение в метод,

//на который у него ссылка, назначенная другим разработчиком

PriceChangeHandler(bankOfAmericaPrice);

//значения получаем каждые 2 секунды

Thread.Sleep(2000);

}

}

}

class Program

{

static void Main(string[] args)

{

StockExchangeMonitor stockExchangeMonitor = new StockExchangeMonitor();

//теперь полученное с биржи значение будет обрабатывать метод ShowPrice

stockExchangeMonitor.PriceChangeHandler = ShowPrice;

stockExchangeMonitor.Start();

}

public static void ShowPrice(int price)

{

Console.WriteLine($"New price is: {price}");

}

}

}

## ► Анонимные методы

С делегатами тесно связаны **анонимные методы**. **Анонимные методы используются для создания экземпляров делегатов.**

Определение анонимных методов начинается с ключевого слова **delegate**, после которого идет в скобках список параметров и тело метода в фигурных скобках:

delegate(параметры)

{

    // инструкции

}

Например:

class Program

{

**delegate void MessageHandler(string message);**

static void Main(string[] args)

{

**MessageHandler handler = delegate (string mes)**

**{**

**Console.WriteLine(mes);**

**};**

handler("hello world!");

Console.Read();

}

}

Анонимный метод не может существовать сам по себе, он используется для инициализации экземпляра делегата, как в данном случае переменная handler представляет анонимный метод. И через эту переменную делегата можно вызвать данный анонимный метод.

Другой пример анонимных методов - передача в качестве аргумента для параметра, который представляет делегат:

class Program

{

delegate void MessageHandler(string message);

static void Main(string[] args)

{

ShowMessage("hello!", **delegate (string mes)**

**{**

**Console.WriteLine(mes);**

**}**);

Console.Read();

}

static void ShowMessage(string mes, **MessageHandler handler**)

{

handler(mes);

}

}

Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата. Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются. При этом даже если делегат принимает несколько параметров, то **в анонимном методе можно вовсе опустить параметры**:

class Program

{

delegate void MessageHandler(string message);

static void Main(string[] args)

{

MessageHandler handler = delegate

{

Console.WriteLine("анонимный метод");

};

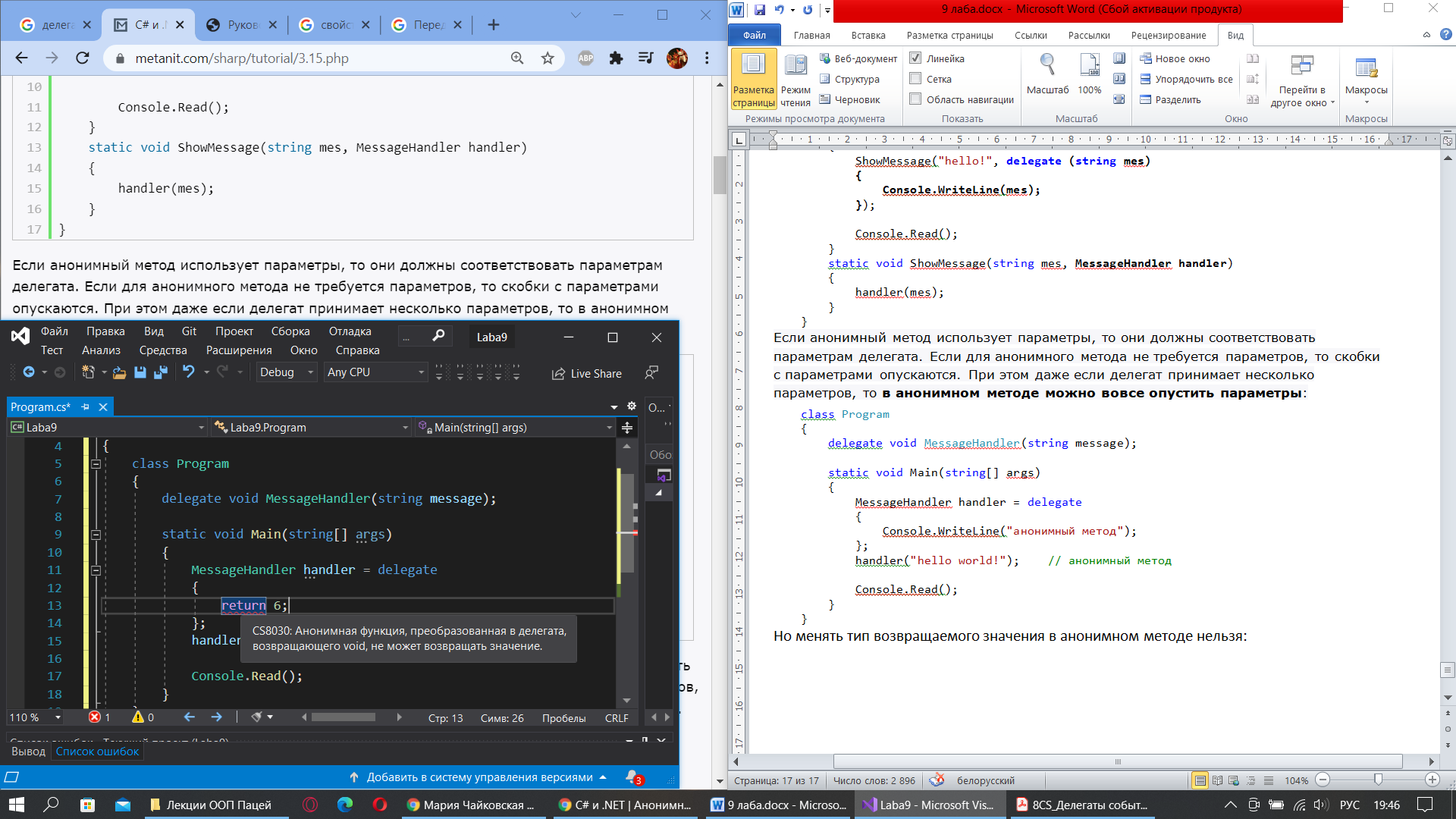
handler("hello world!"); // анонимный метод

Console.Read();

}

}

Но менять тип возвращаемого значения в анонимном методе нельзя:



То есть если анонимный метод содержит параметры, они обязательно должны соответствовать параметрам делегата. Либо анонимный метод вообще может не содержать никаких параметров, тогда он соответствует любому делегату, который имеет тот же тип возвращаемого значения.

При этом параметры анонимного метода не могут быть опущены, если один или несколько параметров определены с модификатором **out**.

Так же, как и обычные методы, анонимные могут возвращать результат:

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation operation = delegate (int x, int y)

{

return x + y;

};

int d = operation(4, 5);

Console.WriteLine(d); // 9

Console.Read();

}

При этом анонимный метод имеет доступ ко всем переменным, определенным во внешнем коде:

delegate int Operation(int x, int y);

static void Main(string[] args)

{

int z = 8;

Operation operation = delegate (int x, int y)

{

return x + y + z;

};

int d = operation(4, 5);

Console.WriteLine(d); // 17

Console.Read();

}

В каких ситуациях используются анонимные методы? Когда нам надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется. В частности, их можно использовать для обработки событий, которые будут рассмотрены далее.

## ► Лямбды

***Лямбда-выражения*** представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы.

Ламбда-выражения имеют следующий синтаксис: слева от лямбда-оператора => определяется список параметров, а справа блок выражений, использующий эти параметры: (список\_параметров) => выражение. Например:

class Program

{

delegate int Operation(int x, int y);

static void Main(string[] args)

{

Operation operation = (x, y) => x + y;

Console.WriteLine(operation(10, 20)); // 30

Console.WriteLine(operation(40, 20)); // 60

Console.Read();

}

}

Здесь код (x, y) => x + y; представляет лямбда-выражение, где x и y - это параметры, а x + y - выражение. При этом нам не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

Таким образом:

Operation operation = delegate (int x, int y)

{

return x + y;

};

то же, что и:

Operation operation = (x, y) => x + y;

При этом надо учитывать, что каждый параметр в лямбда-выражении неявно преобразуется в соответствующий параметр делегата, поэтому **типы соответствующих параметров делегата и лямбда-выражения должны быть одинаковыми**. Кроме того, **количество параметров должно быть таким же, как и у делегата**. И **возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата**. То есть в данном случае использованное лямбда-выражение соответствует делегату Operation как по типу возвращаемого значения, так и по типу и количеству параметров.

Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить:

class Program

{

delegate int Square(int x); // объявляем делегат, принимающий int и возвращающий int

static void Main(string[] args)

{

Square square = i => i \* i; // объекту делегата присваивается лямбда-выражение

int z = square(6); // используем делегат

Console.WriteLine(z); // выводит число 36

Console.Read();

}

}

Бывает, что параметров не требуется. В этом случае вместо параметра в лямбда-выражении используются пустые скобки. Также бывает, что лямбда-выражение не возвращает никакого значения:

class Program

{

delegate void Hello(); // делегат без параметров

static void Main(string[] args)

{

Hello hello1 = () => Console.WriteLine("Hello");

Hello hello2 = () => Console.WriteLine("Welcome");

hello1(); // Hello

hello2(); // Welcome

Console.Read();

}

}

В данном случае лямда-выражение ничего не возвращает, так как после лямбда-оператора идет действие, которое ничего не возвращает.

Как видно, из примеров выше, нам необязательно указывать тип параметров у лямбда-выражения. Однако нам обязательно нужно указывать тип, если делегат, которому должно соответствовать лямбда-выражение, имеет параметры с модификаторами **ref** и **out**:

class Program

{

delegate void ChangeHandler(ref int x);

static void Main(string[] args)

{

int x = 9;

ChangeHandler ch = (ref int n) => n = n \* 2;

ch(ref x);

Console.WriteLine(x); // 18

Console.Read();

}

}

Лямбда-выражения также могут выполнять другие методы:

class Program

{

delegate void Hello(); // делегат без параметров

static void Main(string[] args)

{

Hello message = () => Show\_Message();

message();

}

private static void Show\_Message()

{

Console.WriteLine("Привет мир!");

}

}

### Лямбда-выражения как аргументы методов

Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно:

class Program

{

delegate bool IsEqual(int x);

static void Main(string[] args)

{

int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// найдем сумму чисел больше 5

int result1 = Sum(integers, x => x > 5);

Console.WriteLine(result1); // 30

// найдем сумму четных чисел

int result2 = Sum(integers, x => x % 2 == 0);

Console.WriteLine(result2); //20

Console.Read();

}

private static int Sum(int[] numbers, IsEqual func)

{

int result = 0;

foreach (int i in numbers)

{

if (func(i))

result += i;

}

return result;

}

}

Метод Sum принимает в качестве параметра массив чисел и делегат IsEqual и возвращает сумму чисел массива в виде объекта int. В цикле проходим по всем числам и складываем их. Причем складываем только те числа, для которых делегат IsEqual func возвращает true. То есть делегат IsEqual здесь фактически задает условие, которому должны соответствовать значения массива. Но на момент написания метода Sum нам неизвестно, что это за условие.

При вызове метода Sum ему передается массив и лямбда-выражение:

int result1 = Sum(integers, x => x > 5);

То есть параметр x здесь будет представлять число, которое передается в делегат:

if (func(i))

А выражение x > 5 представляет условие, которому должно соответствовать число. Если число соответствует этому условию, то лямбда-выражение возвращает true, а переданное число складывается с другими числами.

Подобным образом работает второй вызов метода Sum, только здесь уже идет проверка числа на четность, то есть если остаток от деления на 2 равен нулю:

|  |  |
| --- | --- |
|  |  |

int result2 = Sum(integers, x => x % 2 == 0);

### Блочные лямбда-выражения

//объявление

privatedelegateboolisOdd(inta);

staticvoidMain()

{

//определение

isOddtestIsOdd= s4 =>

{

if(s4%2 == 0) return true;

else return false;

};

//вызов

inta = 10;

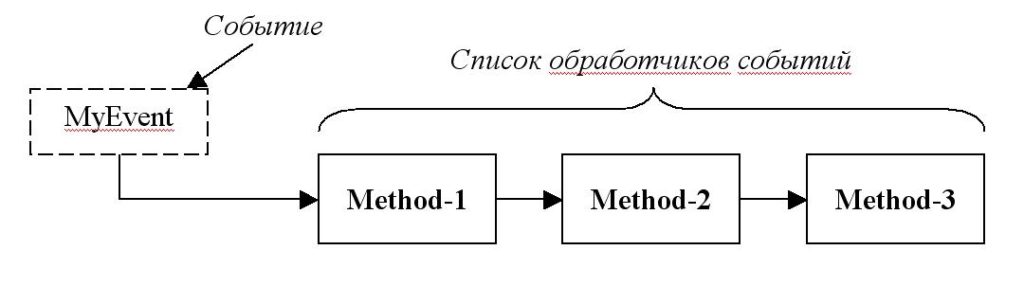
### testIsOdd(a);

## ►События

**Событие** – это автоматическое сообщение о том, что в программе состоялось некоторое действие.

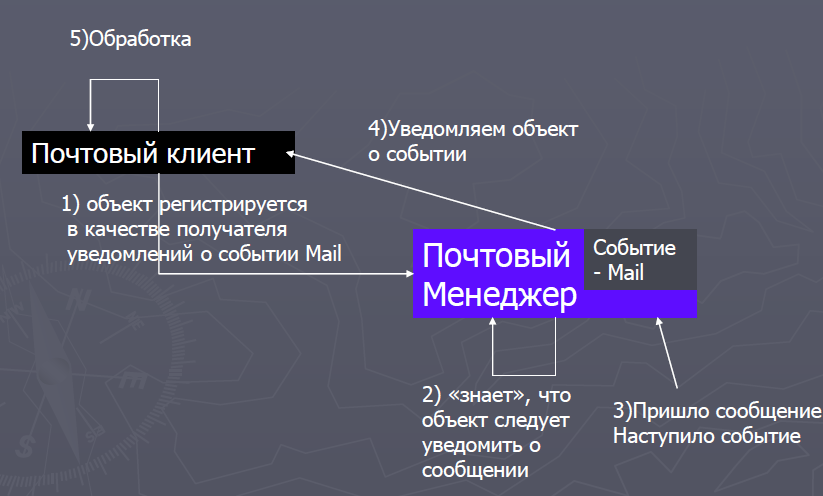
На выполнение того или другого события программа может соответственно реагировать. Реакция на событие осуществляется с помощью, так называемых, обработчиков события. **Обработчик события** – это обычный метод, который выполняет некоторые действия в программе, в случае если состоялось (сгенерировалось) событие.

События работают в объединении с делегатами. Такое объединение позволяет формировать списки (цепочки) обработчиков события (методов), которые должны вызваться при вызове (запуске, генерировании) данного события. Такой подход является эффективным при написании больших программных систем, поскольку он позволяет упорядочить большой сложный программный код, в котором очень легко допустить логическую ошибку.



**Событие** – это **элемент класса**, позволяющий ему [классу] посылать другим объектам уведомления об изменении своего состояния.

Класс, являющийся отправителем уведомления, публикует события, которые он может инициировать, а другие классы, являющиеся получателями, подписываются на получение этих событий.



### Определение и вызов 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.

События **объявляются в классе** с помощью ключевого слова **event**, после которого указывается тип делегата, который представляет событие:

delegate void AccountHandler(string message);

### event AccountHandler Notify;

В данном случае вначале определяется делегат AccountHandler, который принимает один параметр типа string. Затем с помощью ключевого слова **event** определяется событие с именем Notify, которое представляет делегат AccountHandler. Название для события может быть произвольным, но в любом случае событие должно представлять некоторый делегат.

Определив событие, мы можем его вызвать в программе как метод, используя имя события:

### Notify("Произошло действие");

### Однако при вызове событий мы можем столкнуться с тем, что событие равно null в случае, если для его не определен обработчик. Поэтому при вызове события лучше его всегда проверять на null. Например, так:

### if (Notify != null) Notify("Произошло действие");

### Или так:

### Notify?.Invoke("Произошло действие");

В этом случае, поскольку событие представляет делегат, то мы можем его вызвать с помощью метода **Invoke()**, передав в него необходимые значения для параметров.

### Добавление обработчика события

С событием может быть связан один или несколько обработчиков. **Обработчики событий** – это именно то, что выполняется при вызове событий. Нередко в качестве обработчиков событий применяются **методы**. Каждый обработчик событий по списку параметров и возвращаемому типу должен соответствовать делегату, который представляет событие. Для добавления обработчика события применяется операция **+=**:

|  |
| --- |
| Notify += обработчик события;  То есть, мы **подписали метод** на событие, или **зарегистрировали метод** в событии. |

**Бессмысленный пример**

using System;

namespace space

{

//создаем делегат, чтобы создать событие

delegate void Handler(string a, int b);

class Program

{

//создаем событие Stuck делегата Handler

static event Handler Stuck;

static void Main(string[] args)

{

//подписываем на событие Stuck метод Show

Stuck += Show;

//вызываем событие

Stuck("TEXT", 100);

}

static void Show (string a, int b)

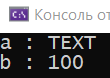
{

Console.WriteLine($"a : {a} \nb : {b}");

}

}

}



По факту, пример просто демонстрирует использование события, мы даже можем убрать ключевое слово event из этой программы и ничего не измениться. Компилятор просто создаст переменную с именем Stuck с типом Handler. Повторюсь: смысл примера – практика без логики.

**DELEGATE vs EVENT**

**Пример для делегата**

namespace space

{

//перечисление с возможными должностями сотрудников

enum Position

{

Manager,

VicePresident,

}

//класс для создания объектов сотрудников

class Employee

{

public Position Position { get; set; } //должность

public int Bonus { get; set; } //начисляемый сотруднику бонус

}

class Program

{

public delegate int CalculateBonus(Position position);

//свойство типа делегата, данное свойство в дальнейшем будет ссылаться на метод(ы) необходимый(ые) для вычисления бонуса

public CalculateBonus CalculateBonusForEmployee { get; set; }

//метод, который принимает объект сотрудника...

private void CalculateBonuses(Employee employee)

{

//затем посылает должность данного сотрудника в метод для высчитывания положенного ему бонуса

//и в конце полученное из этого метода значение записывается в свойство Bonus нашего объекта

employee.Bonus = CalculateBonusForEmployee(employee.Position);

}

static void Main(string[] args)

{

}

}

}

Мы понятия не имеем, что за метод будет вызван, но знаем, что передав позицию сотрудника, мы получим рассчитанный бонус. Таким образом, мы ДЕЛИГИРУЕМ код извне, то есть просим код извне помочь нам с расчетом бонуса. У нас нет никакого события, у нас есть функционал, который нужно выполнить, и нам нужна помощь кода извне.

**Пример для события**

Мы будем поздравлять сотрудников с праздниками.

namespace space

{

//класс для создания объектов сотрудников

class Employee

{

//Id сотрудника

public int Id { get; set; }

}

class Program

{

public delegate void Congratulate(int employeeId);

public event Congratulate Congratulated;

public void CongratulateEmployee(Employee employee)

{

//бизнес-логика -> это нечто связанное с поздравлением (например, начисление премии, выдача путевки в санаторий)

//...код бизнес логики...

//оповещение

Congratulated(employee.Id);

}

static void Main(string[] args)

{

}

}

}

Внимание: для выполнения действий по поздравлению сотрудника нам не нужна помочь извне, мы все делаем сами. В конце мы оповещаем всех подписчиков на событие, о том что сотрудник с конкретным идентификатором поздравлен. То есть мы НЕ зависим от внешней логики.

**Делегат:**

◦ делегировать

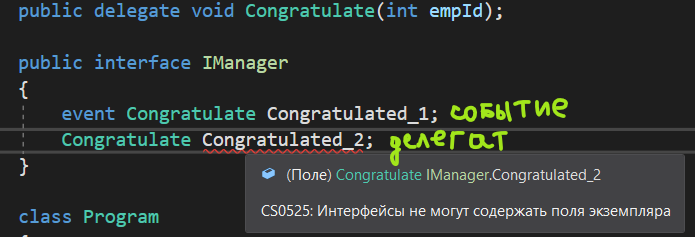
◦ не можем объявить на уровне интерфейса

◦ может быть параметром метода

**Событие:**

◦ оповещать

◦ можем объявить на уровне интерфейса



◦ не может быть параметром метода

## ► Делегаты Action и Func

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action и Func.

Вместо определения нового типа делегата с каждым типом параметра и возврата можно использовать делегаты Action<T> и Func<T>.

**Action**

Делегат Action является обобщенным, **принимает параметры и возвращает значение void**:

public delegate void Action<T>(T obj)

Этот класс делегата существует в различных вариантах, так что ему можно передавать до 16 разных типов параметров.

Класс Action без обобщенного параметра предназначен для вызова методов без параметров, Action<in Т> — для вызова метода с одним параметром, Action<in T1, in Т2> — для вызова метода с двумя параметрами и Action<in T1, in Т2, in ТЗ, in Т4, in Т5, in Т6, in Т7, in Т8> — для вызова метода с восемью параметрами.

**Пример**

class Program

{

static void Main(string[] args)

{

Action<int, int> op;

op = Add;

op += Substract;

op(10, 3);

}

static void Add(int x, int y)

{

Console.WriteLine("Сумма чисел : " + (x + y));

}

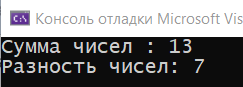
static void Substract(int x, int y)

{

Console.WriteLine("Разность чисел: " + (x - y));

}

}



**Func**

Делегаты Func<T> могут использоваться аналогичным образом. Func<T> **позволяет вызывать методы с типом возврата и разными типами параметров**. Подобно Action<T>, Func<T> определен в разных вариантах для передачи до 16 типов параметров и типа возврата.

TResult Func<out TResult>() //может иметь ссылки на методы с возвращаемым значением TResult, без параметров

TResult Func<in T, out TResult>(T arg) //может иметь ссылки на методы с возвращаемым значением TResult, с одним параметром типа T

TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)

TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)

TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)

//...........................................

**Пример**

using System;

namespace Laba9

{

class Program

{

static void Main(string[] args)

{

Func<int, int, string> op;

op = Add;

Console.WriteLine(op(10, 3));

}

static string Add(int x, int y)

{

return "Сумма чисел : " + (x + y);

}

}

}

****