# Практическая работа № 1 События

**Цель практической работы**

Познакомиться с основами программирования событий.

**Теоретический материал**

**Делегаты**

Делегатыпредставляют объекты, которые указывают на методы. Методы, на которые ссылаются делегаты, должны иметь те же параметры и тот же тип возвращаемого значения.

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

delegate int Operation(int x, int y);

//здесь делегат ссылается на функцию, которая в //качестве параметров принимает два значения типа int и //возвращает некоторое число

delegate void GetMessage();

Для использования делегата надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод *Invoke* (или *BeginInvoke/EndInvoke* при асинхронном режиме выполнения делегатов).

Пример с использованием делегата:

class Program

{

// 1. Объявляем делегат

delegate int Operation(int x, int y);

static void Main(string[] args)

{

// присваивание адреса метода через конструктор

// 2. Создаем переменную делегата

Operation del = new Operation(Add);

// 3. делегат указывает на метод Add

// 4. Вызываем метод

int result = del.Invoke(4, 5);

Console.WriteLine(result);

del = Multiply;

// теперь делегат указывает на метод Multiply

result = del.Invoke(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;

}

}

Здесь объявляется делегат Operation, возвращающий целочисленное значение и принимающий два параметра, также целочисленные. Методы (Add и Multiply), на которые указывает созданный экземпляр делегата, определяются с использованием конструктора.

Метод Invoke() при вызове делегата можно опустить и использовать сокращенную форму:

del = Multiply;

// теперь делегат указывает на метод Multiply

result = del.Invoke(4, 5);

Использование делегатов обусловлено тем, что не всегда 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.

**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

}

Пример:

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.

В отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

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.

Анонимные методы используются, когда надо определить однократное действие, которое не имеет много инструкций и нигде больше не используется.

Также, как и обычные методы, анонимные могут возвращать результат:

class Program

{

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();

}

}

**Лямбда-выражения**

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.

Синтаксис лямбда-выражений:

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 x, y => x + y; – лямбда-выражение, x и y – параметры, x + y – выражение.

При этом не надо указывать тип параметров, а при возвращении результата не надо использовать оператор return.

Возвращаемое значение лямбда-выражений должно быть тем же, что и у делегата. Если лямбда-выражение принимает один параметр, то скобки вокруг параметра можно опустить, если параметров нет – ставятся пустые скобки.

Также бывает, что лямбда-выражение не возвращает никакого значения, и вызывает 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("Вызов сообщения через делегат");

}

}

Одним из частых примеров использования лямбда-выражений является обработка событий. Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно.

**События**

События сигнализируют системе о том, что произошло определенное действие. События объявляются в классе с помощью ключевого слова event, после которого идет название делегата.

// Объявляем делегат

public delegate void AccountStateHandler(

string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры, что и делегат, и возвращать тот же тип, что и делегат. Рассмотрим пример:

class Account

{

// Объявляем делегат

public delegate void AccountStateHandler(string message);

// Событие, возникающее при выводе денег

public event AccountStateHandler Withdrawn;

// Событие, возникающее при добавление на счет

public event AccountStateHandler Added;

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;

if (Added != null)

Added($"На счет поступило {sum}");

}

public void Withdraw(int sum)

{

if (sum <= \_sum)

{

\_sum -= sum;

if (Withdrawn != null) Withdrawn($"Сумма {sum} снята со счета");

}

else

{

if (Withdrawn != null) Withdrawn("Недостаточно денег на счете");

}

}

}

События Withdrawn и Added объявлены как экземпляры делегата AccountStateHandler, поэтому для их обработки потребуется метод, принимающий строку в качестве параметра. Затем в методах Put и Withdraw эти события вызываются. Перед вызовом мы проверяем, закреплены ли за этими событиями обработчики. Так как эти события представляют делегат AccountStateHandler, принимающий в качестве параметра строку, то и при вызове событий мы передаем в них строку.

Для прикрепления обработчика события к определенному событию используется операция += и, соответственно, для открепления – операция -=:

событие += метод\_обработчика\_события.

Метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип:

class Program

{

static void Main(string[] args)

{

Account account = new Account(200);

// Добавляем обработчики события

account.Added += Show\_Message;

account.Withdrawn += Show\_Message;

account.Withdraw(100);

// Удаляем обработчик события

account.Withdrawn -= Show\_Message;

account.Withdraw(50);

account.Put(150);

Console.ReadLine();

}

private static void Show\_Message(string message)

{

Console.WriteLine(message);

}

}

Оба способа прикрепления обработчиков равноценны:

account.Added += Show\_Message;

account.Added += new

Account.AccountStateHandler(Show\_Message);

**Задание на практическую работу**

1. Разработать консольное приложение для выполнения заданий.
2. Предусмотреть понятный, аккуратный интерфейс ввода и вывода данных на консоль.

**Задание 1.**

"Светофор" (Базовые события)

**Задача**: Создать класс TrafficLight, который имитирует работу светофора с событиями.

**Требования**:

* Светофор должен циклически переключаться между Красный->Желтый->Зеленый->Желтый->Красный
* Событие LightChanged должно срабатывать при каждом изменении цвета
* Подписчик события выводит текущий цвет в консоль
* Добавить метод Run(), который запускает светофор на 3 цикла

**Задание 2.**

"Умный дом" (Кастомные EventArgs)

**Задача**: Реализовать класс SmartThermostat, который реагирует на изменение температуры.

**Требования**:

* Событие TemperatureAlert с кастомным EventArgs (текущая температура, сообщение)
* При достижении температуры > 30°C - событие "Жарко!"
* При температуре < 15°C - событие "Холодно!"
* При 18-22°C - событие "Комфортно"
* Метод UpdateTemperature для изменения температуры

**Задание 3.**

"Кнопка с историей" (Управление подписками)

**Задача**: Создать класс HistoryButton, который запоминает все нажатия.

**Требования**:

* Событие Clicked при нажатии кнопки
* Свойство ClickHistory (List<DateTime>) - хранит время всех нажатий
* Метод Subscribe для подписки с возможностью указать имя подписчика
* Метод Unsubscribe для отписки
* При нажатии выводить: "Кнопка нажата [время]. Подписчик: [имя]"

**Пример разработки приложения**

**Задание 1.**

"Светофор" (Базовые события)

**Задача**: Создать класс TrafficLight, который имитирует работу светофора с событиями.

**Требования**:

* Светофор должен циклически переключаться между Красный->Желтый->Зеленый->Желтый->Красный
* Событие LightChanged должно срабатывать при каждом изменении цвета
* Подписчик события выводит текущий цвет в консоль
* Добавить метод Run(), который запускает светофор на 3 цикла

Шаг 1: Создаем перечисление для цветов светофора

public enum TrafficLightColor

{

Red,

Yellow,

Green

}

Шаг 2: Создаем класс TrafficLight с событием

public class TrafficLight

{

// Событие, которое будет вызываться при изменении цвета

public event Action<TrafficLightColor> LightChanged;

private TrafficLightColor \_currentColor;

private readonly int[] \_timings = { 3, 1, 3, 1 }; // Тайминги для Red, Yellow, Green, Yellow

public TrafficLight()

{

\_currentColor = TrafficLightColor.Red; // Начальный цвет

}

}

Шаг 3: Реализуем логику переключения цветов

private void ChangeColor()

{

var initialColor = \_currentColor;

switch (\_currentColor)

{

case TrafficLightColor.Red:

\_currentColor = TrafficLightColor.Yellow;

break;

case TrafficLightColor.Yellow:

\_currentColor = \_previousColor == TrafficLightColor.Red

? TrafficLightColor.Green

: TrafficLightColor.Red;

break;

case TrafficLightColor.Green:

\_currentColor = TrafficLightColor.Yellow;

break;

}

\_previousColor = initialColor;

LightChanged?.Invoke(\_currentColor);

}

private TrafficLightColor \_previousColor;

Шаг 4: Реализуем метод Run()

public void Run(int cycles = 3)

{

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

{

// Полный цикл: Red -> Yellow -> Green -> Yellow -> Red

foreach (var timing in \_timings)

{

ChangeColor();

Thread.Sleep(timing \* 1000); // Пауза в секундах

}

}

}

Шаг 5: Тестируем светофор

class Program

{

static void Main()

{

var trafficLight = new TrafficLight();

// Подписываемся на событие

trafficLight.LightChanged += color =>

{

Console.ForegroundColor = GetConsoleColor(color);

Console.WriteLine($"Светофор: {color}");

Console.ResetColor();

};

Console.WriteLine("Запуск светофора на 3 цикла:");

trafficLight.Run();

}

private static ConsoleColor GetConsoleColor(TrafficLightColor color)

{

return color switch

{

TrafficLightColor.Red => ConsoleColor.Red,

TrafficLightColor.Yellow => ConsoleColor.Yellow,

TrafficLightColor.Green => ConsoleColor.Green,

\_ => ConsoleColor.White

};

}

}

**Контрольные вопросы**

1. Что такое делегаты в C# и какова их основная цель?
2. Как объявить делегат и как его использовать?
3. В чем разница между многократным делегатом и делегатом с одним методом?
4. Что такое анонимные методы и как они отличаются от обычных методов?
5. Как создать анонимный метод и в каких случаях его следует использовать?
6. Что такое лямбда-выражения в C# и как они связаны с делегатами?
7. Какова синтаксическая структура лямбда-выражения?
8. В чем преимущества использования лямбда-выражений по сравнению с анонимными методами?
9. Как обрабатывать исключения, возникающие при работе с событиями?