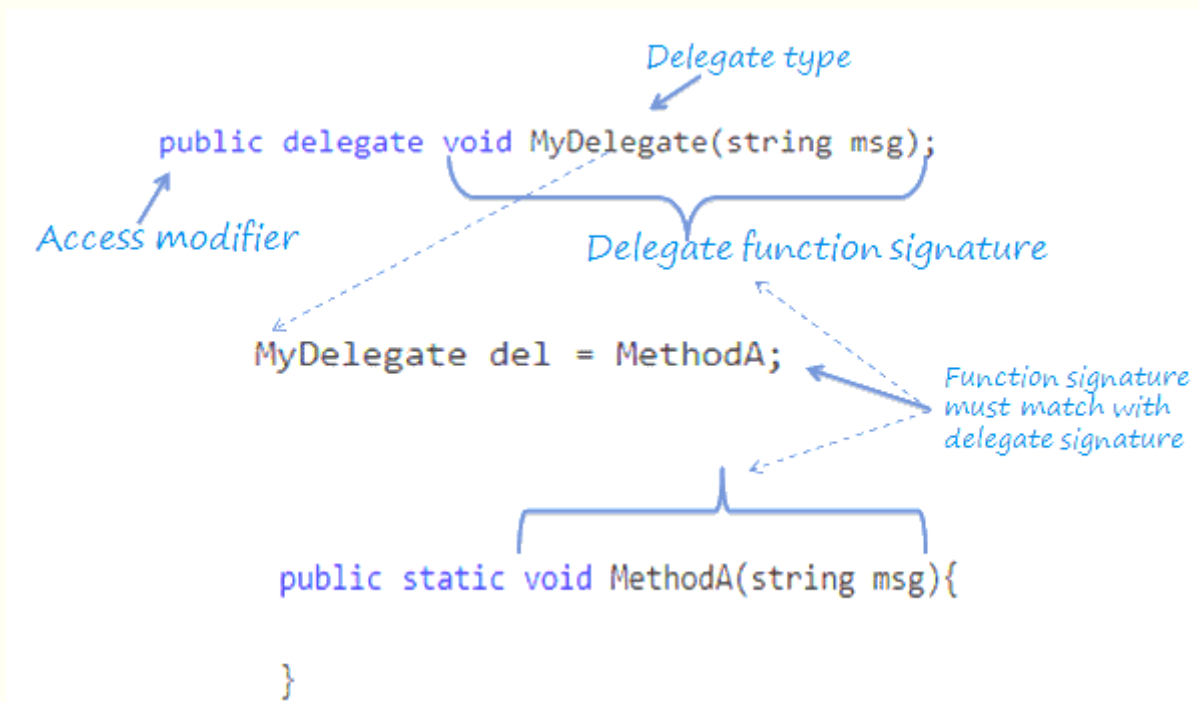




ДЕЛЕГАТИ ТА АНОНІМНІ ТИПИ

Питання 5.4.

Делегати



■ **Делегат** – це тип, який представляє посилання на методи з конкретним списком параметрів та вихідним типом.

- При інстанціюванні делегата його екземпляр можна асоціювати з будь-яким методом, що має сумісну сигнатуру та вихідний тип.
- Викликати (`invoke`, `call`) метод можна через екземпляр делегата.

■ Приклади:

- Делегат, що представляє метод, який додає 2 числа і повертає результат:
- `delegate int AddNumbers(int value1, int value2);`
- Делегат, який представляє метод, що логує виняток та нічого не повертає:
- `delegate void LogException(Exception ex);`
- Делегат, що представляє функцію, яка перетворює деякий узагальнений тип у рядок:
- `delegate string FormatAsString<T>(T input);`

Приклад використання делегата

```
class Program
{
    delegate double MathCalculation(float value1, float value2);

    public static class Calculator
    {
        public static double AddNumbers(float value1, float value2)
        { return value1 + value2; }

        public static double DivideNumbers(float value1, float value2)
        { return value1 / value2; }
    }

    static void Main(string[] args)
    {
        // присвоюємо змінним типу "делегат" метод з доречною сигнатурою
        MathCalculation add = Calculator.AddNumbers;
        MathCalculation divide = Calculator.DivideNumbers;

        // виклик делегата за допомогою Invoke
        var result = add.Invoke(2, 3);
        Console.WriteLine(result);

        // виклик делегата без Invoke
        var result2 = divide(100, 3);
        Console.WriteLine(result2);
    }
}
```

- Робота з делегатами включає 3 етапи:
 - Оголошення делегата;
 - Встановлення цільового (target) метода;
 - Виклик (invoke) делегата.
- Оголошення делегата виглядає так же, як і для методу, крім ключового слова `delegate` на початку.
 - Як і класи та інтерфейси, делегати можна оголошувати поза класами чи всередині них.
 - Доступні модифікатори `private`, `public`, `internal`.
- Присвоїти делегату можна посилання на будь-який метод, що відповідає сигнатурі делегата.
 - Виклик (calling) методу, на який посилається делегат, називають **викликом делегата** (*delegate invoking*).

Передача делегата в якості параметра

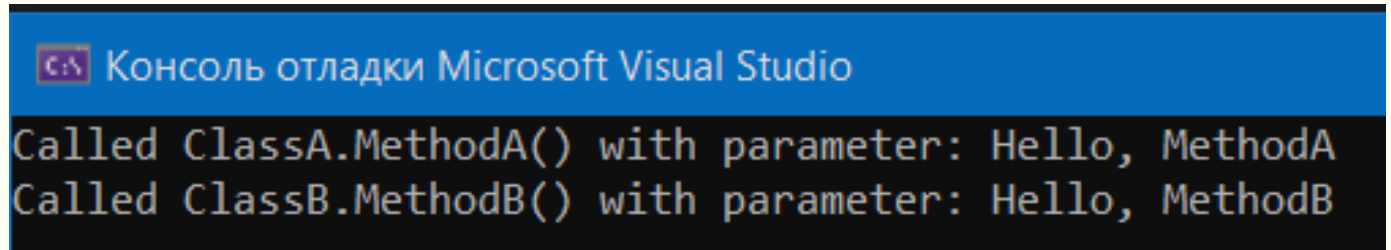
```
namespace DelegateDemo
{
    class ClassA
    {
        public static void MethodA(string message)
        {
            Console.WriteLine("Called ClassA.MethodA() with parameter: " + message);
        }
    }

    class ClassB
    {
        public static void MethodB(string message)
        {
            Console.WriteLine("Called ClassB.MethodB() with parameter: " + message);
        }
    }

    class Program
    {
        static void InvokeDelegate(MyDelegate del) // параметр типу MyDelegate
        {
            del(string.Format("Hello, {0}", del.Method.Name.ToString()));
        }

        static void Main(string[] args)
        {
            MyDelegate del = ClassA.MethodA;
            InvokeDelegate(del);

            del = ClassB.MethodB;
            InvokeDelegate(del);
        }
    }
}
```



Консоль отладки Microsoft Visual Studio

```
Called ClassA.MethodA() with parameter: Hello, MethodA
Called ClassB.MethodB() with parameter: Hello, MethodB
```

Зіставлення делегата

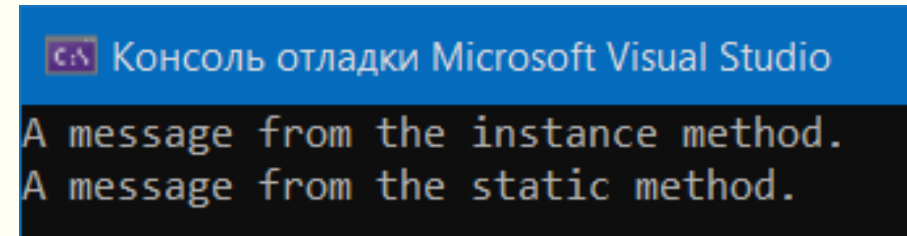
```
// Declare a delegate
delegate void Del();

class SampleClass {
    public void InstanceMethod() {
        System.Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod() {
        System.Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass {
    static void Main() {
        SampleClass sc = new SampleClass();
        Del d = sc.InstanceMethod;
        d();
        d = SampleClass.StaticMethod;
        d();
    }
}
```

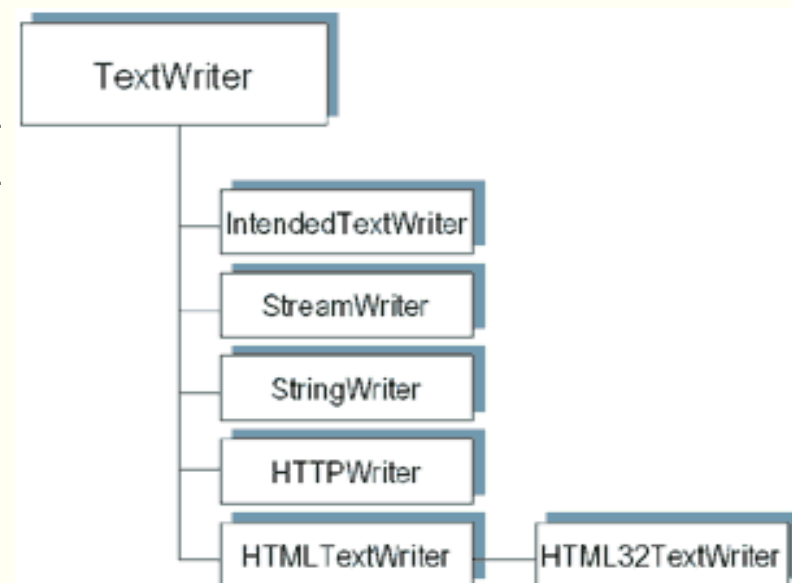
- Делегат, зіставлений одночасно зі статичним методом та методом екземпляра, повертає інформацію від кожного з них



Коваріація у делегатах

- Коли ви призначаєте делегату метод, сигнатура методу не обов'язково повинна точно відповідати делегату.
 - Коваріація дозволяє методу мати *більш похідний вихідний тип*, ніж той, що визначено в делегаті.
 - Оскільки і StreamWriter, і StringWriter успадковуються від TextWriter, ви можете використовувати CovarianceDel з обома методами.

```
class Program {  
    public delegate TextWriter CovarianceDel();  
    public static StreamWriter MethodStream() { return null; }  
    public static StringWriter MethodString() { return null; }  
  
    static void Main() {  
        CovarianceDel del;  
        del = MethodStream;  
        del = MethodString;  
        Console.ReadLine();  
    }  
}
```



Контраваріантність у делегатах

- Контраваріантність дозволяє метод з *типами параметрів, що є менш похідними*, ніж типи в делегаті.
 - Оскільки метод DoSomething може працювати з TextWriter, він, безумовно, може працювати і з StreamWriter.
 - Завдяки контраваріантності ви можете викликати делегат і передавати екземпляр StreamWriter у метод DoSomething().

```
class Program {  
    public static void DoSomething(TextWriter textWriter) { }  
    public delegate void ContravarianceDel(StreamWriter streamWriter);  
  
    static void Main() {  
        ContravarianceDel del = DoSomething;  
        Console.ReadLine();  
    }  
}
```

Області застосування делегатів

- Делегати використовуються у наступних випадках:
 - Обробка подій (Event handlers)
 - Функції/методи зворотного виклику (Callbacks)
 - LINQ
 - Реалізація шаблонів проектування
- У делегатів немає специфічних можливостей, які неможливо реалізувати за допомогою звичайних методів.
 - Делегати дозволяють методам передаватись в якості параметрів.
 - Як і інтерфейси, делегати дозволяють знизити зв'язність (decouple) та узагальнювати код.
 - За потреби вирішити, який метод викликати під час виконання, використовується делегат.
 - Делегати забезпечують спосіб спеціалізації поведінки класу без його субкласування.


```

class Program
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Початковий синтаксис делегата для
        // ініціалізації іменованим методом.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: делегат може ініціалізуватись за допомогою
        // вбудованого (inline) коду, що називають "анонімним методом".
        // Цей метод приймає рядок у якості вхідного параметру.
        TestDelegate testDelB = delegate (string s) { Console.WriteLine(s); };

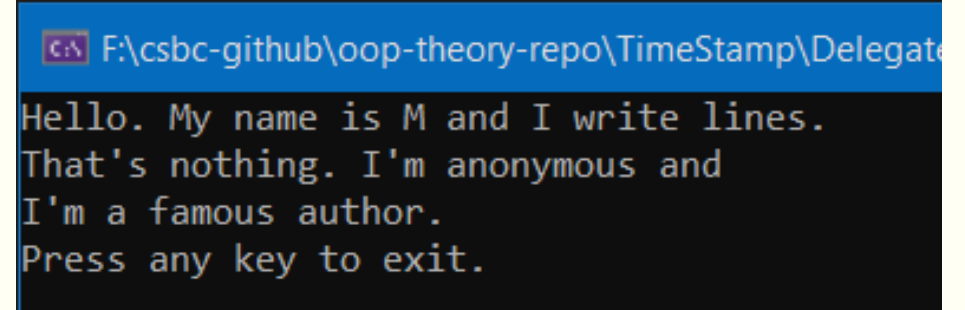
        // C# 3.0. Делегат може ініціалізуватись за допомогою
        // лямбда-виразу. Лямбда також приймає рядок у якості
        // вхідного параметру (x). Тип x виводиться (is inferred) компілятором.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Виклик делегатів.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Тримаємо вікно консолі в режимі debug.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

Еволюція делегата в C# 1.0 – C# 3.0



```

F:\csbc-github\oop-theory-repo\TimeStamp\Delegate
Hello. My name is M and I write lines.
That's nothing. I'm anonymous and
I'm a famous author.
Press any key to exit.

```

- В C # 1.0 екземпляр делегата створювався шляхом явної ініціалізації за допомогою методу, який був визначений в іншому місці коду.
 - В C # 2.0 введена концепція **анонімних методів** як способу написання різних неіменованих вбудованих блоків операторів, які можуть бути виконані у виклику делегата.
 - В C # 3.0 введені **лямбда-вирази**, по суті аналогічні анонімним методам, але більш виразні і чіткі.
 - Ці дві функції разом називають **анонімними функціями**.
 - Як правило, в додатках, призначених для .NET Framework 3.5 і пізніших версій, слід використовувати лямбда-вирази.

Анонімні методи

- Оператор `delegate` створює анонімний метод, який може конвертуватись у тип `delegate` (delegate type):

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };  
Console.WriteLine(sum(3, 4)); // output: 7
```

- Вбудовані узагальнені делегати Action, Predicate і Func.
- Використовуючи оператор `delegate`, можна опустити список параметрів.
 - Це єдина функціональність, що не підтримується лямбда-виразами.

```
Action greet = delegate { Console.WriteLine("Hello!"); };  
greet();
```

```
Action<int, double> introduce = delegate {  
    Console.WriteLine("This is world!"); }; introduce(42, 2.7);
```

```
// Вивід:  
// Hello!  
// This is world!
```

Анонімні методи

- Починаючи з C# 9.0, можна задавати порожні змінні (discards) для задавання 2 або більше вхідних параметрів анонімного методу, які не будуть ним використовуватись:

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };  
Console.WriteLine(constant(3, 4)); // вивід: 42
```

- Если только один параметр имеет имя _, для обеспечения обратной совместимости _ рассматривается как имя этого параметра в анонимном методе.
- Починаючи з C# 9.0, можна застосовувати модифікатор `static` в оголошенні анонімного методу:

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };  
Console.WriteLine(sum(10, 4)); // вивід: 14
```

- Статичний анонімний метод не може перехоплювати (capture) локальні змінні або стан екземпляра з оточуючих областей (enclosing scopes).

Лямбда-вирази

- Починаючи з C# 3, лямбда-вирази надають більш виразний та короткий спосіб створення анонімних функцій.

- Використовується лямбда-оператор `=>`

```
Func<int, int, int> sum = (a, b) => a + b;  
Console.WriteLine(sum(3, 4)); // вивід: 7
```

- Лямбда-вираз – це вираз в одній з двох форм:

- Вирази-лямбди (Expression lambda) мають вираз у своєму тілі: `(input-parameters) => expression`
 - Лямбди-інструкції (лямбди операторів, Statement lambda) мають блок інструкцій у своєму тілі: `(input-parameters) => { <sequence-of-statements> }`

- Лямбда-вираз може бути перетворений в тип `delegate` відповідно до типів параметрів та вихідного значення.

- Якщо лямбда-вираз не повертає значення, лямбду можна перетворити в один з типів делегата Action.
 - Інакше лямбду перетворюють в один з типів делегата Func.

Лямбда-вирази

- Наприклад, лямбда-вираз, що має 2 параметри та не повертає значення, можна перетворити в делегат `Action<T1,T2>`.
 - Лямбда-вираз, який має 2 параметри та повертає значення, можна перетворити в делегат `Func<T,TResult>`.
 - Далі в прикладі лямбда-вираз `x => x * x`, який вказує параметр з іменем `x` і повертає значення `x` у квадраті, присвоюється змінній типу делегата:

```
Func<int, int> square = x => x * x;  
Console.WriteLine(square(5)); // Вивід: 25
```

- Лямбда-вирази можна використовувати в будь-якому коді, для якого потрібні екземпляри типів делегатів або дерева виразів, наприклад, у якості аргументу метода `Task.Run(Action)` для передачі коду, який повинен виконуватись у фоновому режимі.
 - Можна також використати лямбда-вирази, застосовуючи LINQ у C#, наприклад:

```
int[] numbers = { 2, 3, 4, 5 };  
var squaredNumbers = numbers.Select(x => x * x);  
Console.WriteLine(string.Join(" ", squaredNumbers)); // Вивід: 4 9 16 25
```

Вхідні параметри лямбда-виразу

- Вхідні параметри лямбда-виразу заключаються в круглї дужки.
 - Нуьова кїлькїсть вхїдних параметрїв задається пустими дужками:
 - **Action line** = () => Console.WriteLine();
 - Якщо лямбда-вираз має тїльки один вхїдний параметр, круглї дужки необов'язковї:
 - **Func<double, double> cube** = x => x * x * x;
 - Два або бїльше вхїдних параметри роздїляються комами:
 - **Func<int, int, bool> testForEquality** = (x, y) => x == y;
- Інодї компїлятор не може вивести типи вхїдних параметрїв.
 - Можна вказати типи даних в явному виглядї:
 - **Func<int, string, bool> isTooLong** = (int x, string s) => s.Length > x;
- Для вхїдних параметрїв усї типи потрїбно задати або явно, або неявно.
 - Інакше компїлятор видає помилку CS0748.
 - Починаючи з C# 9.0, можна використовувати пустї змїнні, щоб вказати 2 чи бїльше вхїдних параметрїв лямбда-виразу, якї не використовуються в виразї:
 - **Func<int, int, int> constant** = (_, _) => 42;

Лямбда-вирази та кортежі

- Кортеж можна ввести в якості аргумента лямбда-виразу, а лямбда-вираз може повертати кортеж.
 - Інколи компілятор C# використовує визначення типу для визначення типів компонентів кортежа.
 - Далі кортеж з трьома компонентами використовується для передачі послідовності чисел в лямбда-вираз.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);  
var numbers = (2, 3, 4);  
var doubledNumbers = doubleThem(numbers);  
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");  
// Вивід: The set (2, 3, 4) doubled: (4, 6, 8)
```

- Тим не менш, кортеж з іменованими компонентами можна визначити:

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);  
var numbers = (2, 3, 4);  
var doubledNumbers = doubleThem(numbers);  
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

```

class Program
{
    static void Hello(string s)
    {
        Console.WriteLine(" Hello, {0}!", s);
    }

    static void Goodbye(string s)
    {
        Console.WriteLine(" Goodbye, {0}!", s);
    }

    delegate void Del(string s);

    static void Main()
    {
        Del a, b, c, d;

        // Создаем делегат а ссылающийся на метод Hello:
        a = Hello;

        // Создаем делегат b ссылающийся на метод Goodbye:
        b = Goodbye;

        // Формируем композицию делегатов а и b - c:
        c = a + b;

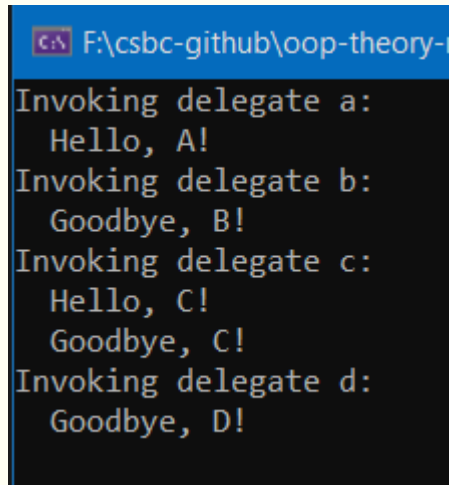
        // Удаляем а из композиции делегатов c, создавая делегат d,
        // который в результате вызывает только метод Goodbye:
        d = c - a;

        Console.WriteLine("Invoking delegate a:");
        a("A");
        Console.WriteLine("Invoking delegate b:");
        b("B");
        Console.WriteLine("Invoking delegate c:");
        c("C");
        Console.WriteLine("Invoking delegate d:");
        d("D");

        Console.ReadLine();
    }
}

```

Багатоадресні (multicast) делегати



```

F:\csbc-github\oop-theory-
Invoking delegate a:
Hello, A!
Invoking delegate b:
Goodbye, B!
Invoking delegate c:
Hello, C!
Goodbye, C!
Invoking delegate d:
Goodbye, D!

```

Ви можете використовувати оператор + або +=, щоб додати інший метод у список викликів існуючого екземпляра делегата.

- Аналогічно, також можна видалити метод зі списку викликів, використовуючи оператор присвоєння/декремента (- або -=).
- Ця особливість слугує основою для подій у C#.

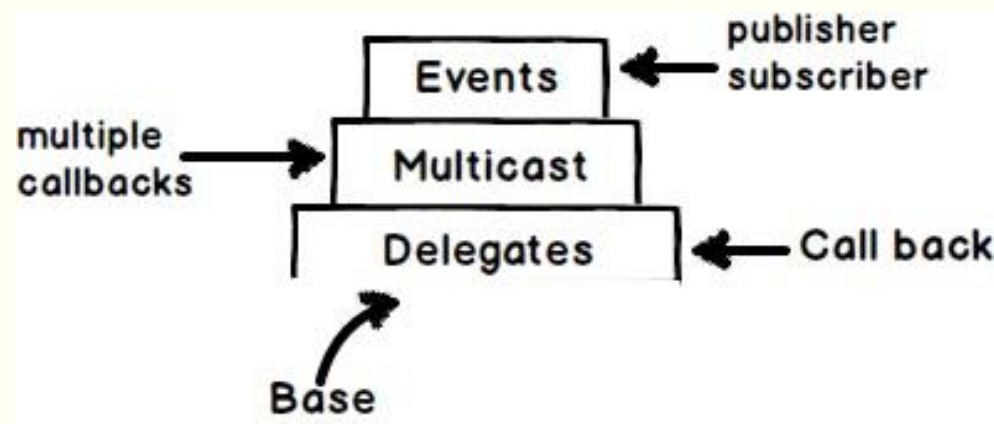
Це можливо, оскільки делегати наслідуються від класу `System.MulticastDelegate`, породженого від `System.Delegate`.

- Ви можете використовувати члени, визначені в цих базових класах для ваших делегатів.

Події та делегати



- Події (events) – це члени, які дозволяють класу повідомляти про зміни, а користувачам класу – реагувати на них.
 - Події інкапсулюють делегат та реалізують модель publisher-subscriber.
 - Ви можете підписатись на подію, а потім будете сповіщені, коли видавець події ініціює нову подію.
 - Ця система використовується для встановлення слабкого зв'язку між компонентами в додатку.



Події та делегати

- Оголошення події:
 - `public event EventHandler Changed;`
 - Ключове слово `event` повідомляє компілятору про те, що це не публічне поле, а конструкція, що спеціальним чином розкривається та приховує від програміста деталі реалізації механізму подій.
- Події в мові C# можуть бути визначені 2 способами:
 - «Неявна» реалізація події (field-like event).
 - «Явна» реалізація події.
 - Слова “явна” та “неявна” тут не є офіційними термінами, а просто описують спосіб реалізації по смислу.
- Зсередини «неявна» реалізація події має два основних блоки:
 - `add {...}` – викликається при підписці на подію;
 - `remove {...}` – викликається при відписці від події.
- Ці блоки компілюються в окремі методи з унікальними іменами, обидва
 - приймають один параметр – делегат типу, що відповідає типу події. Ім'я параметру – завжди `value`.
 - не повертають нічого

Події та делегати

```
class Class {
    EventHandler changed;
    public event EventHandler Changed {
        add {
            EventHandler eventHandler = this.changed;
            EventHandler comparand;
            do {
                comparand = eventHandler;
                eventHandler = Interlocked.CompareExchange<EventHandler>(ref this.changed,
                    comparand + value, comparand);
            } while(eventHandler != comparand);
        }
        remove {
            EventHandler eventHandler = this.changed;
            EventHandler comparand;
            do {
                comparand = eventHandler;
                eventHandler = Interlocked.CompareExchange<EventHandler>(ref this.changed,
                    comparand - value, comparand);
            } while (eventHandler != comparand);
        }
    }
}
```

Рядки

```
class Class {
    public event EventHandler Changed;
}
```

трансляються компілятором у код зліва.

Пояснення коду

- Область видимості (зліва від ключового слова `event`) визначає зону видимості цих методів.
 - Також створюється делегат з ім'ям події, який завжди приватний.
 - Саме тому ми не можемо викликати подію, реалізовану неявним способом, з нащадка класу.
- `Interlocked.CompareExchange` виконує порівняння першого аргументу з третім і якщо вони рівні, замінює перший аргумент на другий.
 - Ця дія потокобезпечна. Цикл використовується для випадку, коли після присвоєння змінної `comparand` делегата події, і до виконання порівняння інший потік змінює цей делегат.
 - У такому випадку `Interlocked.CompareExchange` не виконує заміни, гранична умова циклу не виконується і відбувається наступна спроба.
- При явній реалізації події програміст оголошує делегат-поле для події та вручну додає чи видаляє підписників через блоки `add/remove`, обидва з яких повинні присутствовать.
 - Таке оголошення часто використовується для створення свого механізму подій зі збереженням зручностей мови C# у роботі з ними.

Виклик (Raise) та споживання (Consume) подій

```
class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold) {
        threshold = passedThreshold;
    }

    public void Add(int x) {
        total += x;
        if (total >= threshold)
        {
            OnThresholdReached(EventArgs.Empty);
        }
    }

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    public event EventHandler ThresholdReached;
}
```

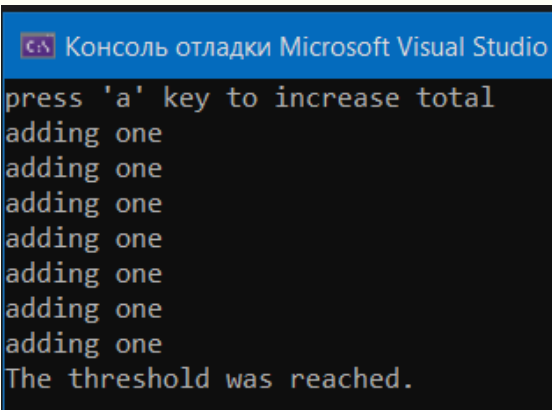
27.10.2020

- У прикладі викликається та споживається подія, що не має даних.
 - Він містить клас Counter, у якому присутня подія ThresholdReached.
 - Подія викликається, коли значення лічильника \geq порогового значення.
 - Делегат EventHandler пов'язаний з подією, оскільки не постачається даних події.

```
class Program
{
    static void c_ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
        Environment.Exit(0);
    }

    static void Main(string[] args)
    {
        Counter c = new Counter(new Random().Next(10));
        c.ThresholdReached += c_ThresholdReached;

        Console.WriteLine("press 'a' key to increase total");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("adding one");
            c.Add(1);
        }
    }
}
```



```
Консоль отладки Microsoft Visual Studio
press 'a' key to increase total
adding one
adding one
adding one
adding one
adding one
adding one
The threshold was reached.
```

```

class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold)
    {
        threshold = passedThreshold;
    }

    public void Add(int x)
    {
        total += x;
        if (total >= threshold)
        {
            ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
            args.Threshold = threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
    {
        EventHandler<ThresholdReachedEventArgs> handler = ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}

```

Виклик (Raise) та споживання (Consume) подій (приклад 2)

- Приклад показує виклик та споживання події, що постачає дані.
 - Делегат EventHandler<TEventArgs> пов'язаний з подією, а екземпляр об'єкту даних користувацької (custom) події постачається.
 - Використовуючи делегат типу EventHandler<TEventArgs>, ви визначаєте новий клас, породжений від класу EventArgs.
 - Клієнт, який хоче обробити нову подію, потребує доступ до цього нового класу.

Виклик (Raise) та споживання (Consume) подій (приклад 2)

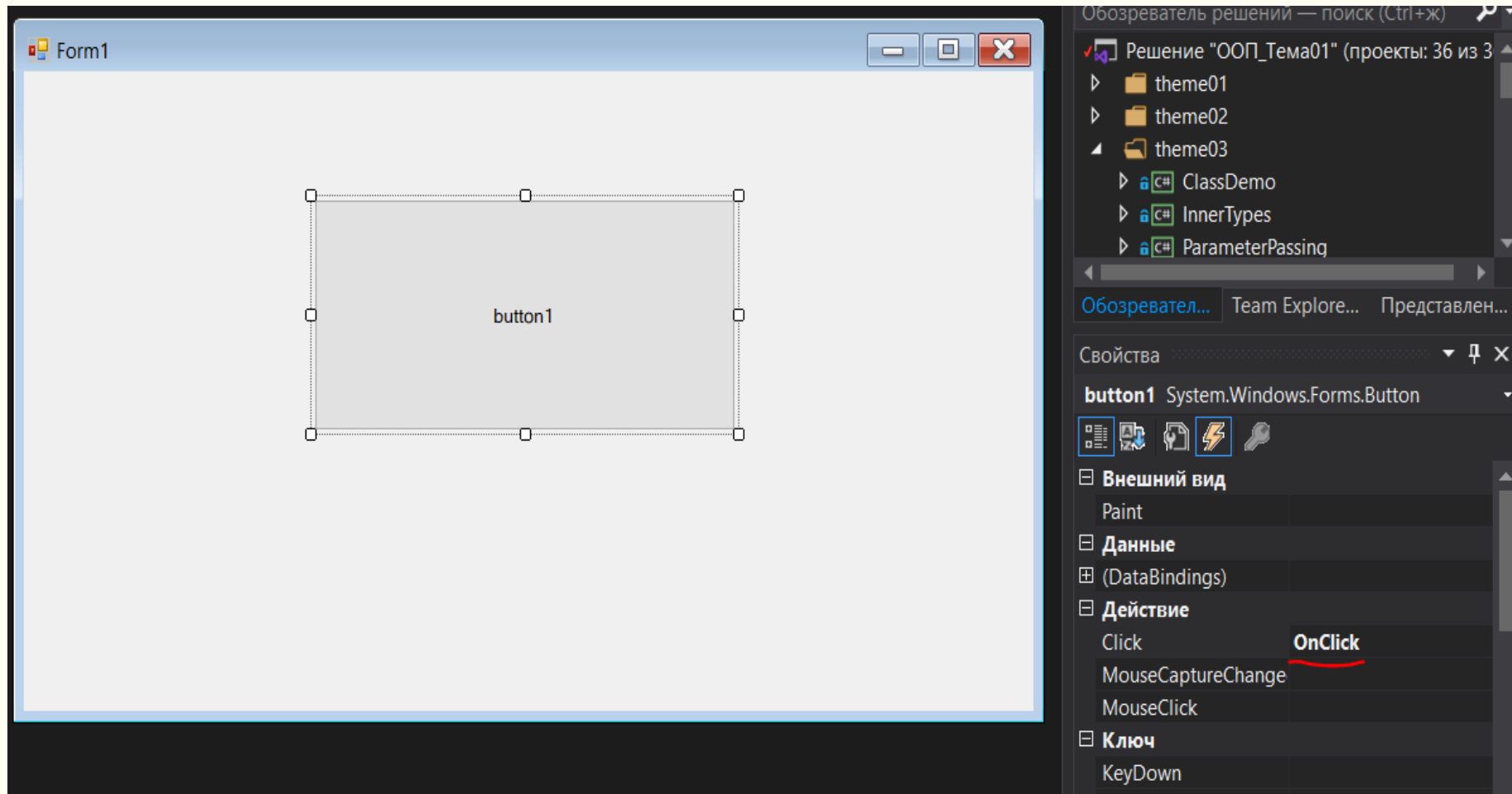
```
class Program
{
    static void Main(string[] args)
    {
        Counter c = new Counter(new Random().Next(10));
        c.ThresholdReached += c_ThresholdReached;

        Console.WriteLine("press 'a' key to increase total");
        while (Console.ReadKey(true).KeyChar == 'a')
        {
            Console.WriteLine("adding one");
            c.Add(1);
        }
    }

    static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
    {
        Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold, e.TimeReached);
        Environment.Exit(0);
    }
}
```

- Породжений від EventArgs параметр містить event data.
 - Параметр object sender містить посилання на елемент управління або об'єкт, який викликає подію.

Приклад у додатках з графічним інтерфейсом

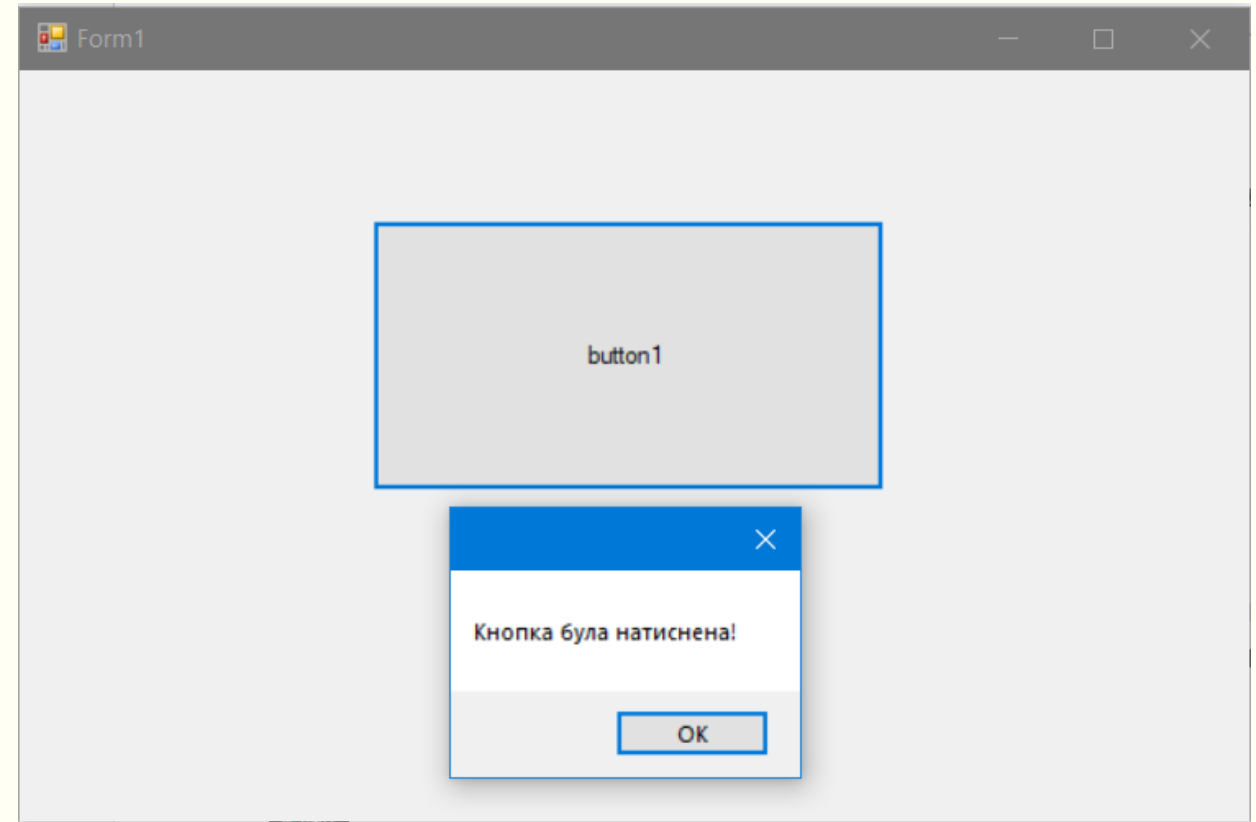


- Розглянемо приклад WinForms-проєкту з однією кнопкою та подією Click – натиснення на кнопку.
 - Визначаємо обробник – метод `OnClick()`, який опише реакцію на натиснення кнопки.

Приклад у додатках з графічним інтерфейсом

```
namespace ButtonClick2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void OnClick(object sender, EventArgs e)
        {
            MessageBox.Show("Кнопка була натиснена!");
        }
    }
}
```



Локальні функції (починаючи з C# 7.0)

- Локальні функції – це закриті методи типу, вкладені в деякий його член.
- Оголошувати та викликати локальні функції можна в:
 - Методах, зокрема методах ітератора та асинхронних методах
 - Конструкторах
 - Методах доступу властивостей
 - Методах доступу подій
 - Анонимних функціях
 - Фіналізаторах
 - Інших локальних функціях

```
private static string GetText(string path, string filename) {  
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");  
    var text = reader.ReadToEnd();  
    return text;  
  
    string AppendPathSeparator(string filepath) {  
        return filepath.EndsWith(@"\") ? filepath : filepath + @\"\\\";  
    }  
}
```

Синтаксис локальних функцій

- Локальні функції можуть викликатись лише з того елемента, в який вони вкладені.
- Допускаються модифікатори:
 - *async*
 - *unsafe*
 - *static* (починаючи з C# 8.0). Статична локальна функція не може захоплювати (capture) локальні змінні чи стан екземпляру.
 - *extern* (починаючи з C# 9.0). Зовнішня(external) локальна функція повинна бути статичною.
- Усі локальні змінні, визначені в члені-контейнері, включаючи параметри його методу, доступні в нестатичній локальній функції.
 - На відміну від визначення методу, визначення локальної функції не може містити модифікатор доступу, оскільки всі локальні функції приватні (закриті).
 - Включення модифікатора `private` призведе до помилки компіляції CS0106: "The modifier 'private' is not valid for this item."

Синтаксис локальних функцій

- Починаючи з C# 9.0, можна застосовувати атрибути до локальних функцій, їх параметрів та параметрів типів (type parameters – у наступному питанні):

```
#nullable enable
private static void Process(string?[] lines, string mark) {
    foreach (var line in lines) {
        if (IsValid(line)) { // Логіка обробки... }
    }

    bool IsValid([NotNullWhen(true)] string? line) {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length; }
}
```

- Код містить спеціальний атрибут NotNullWhen, який допомагає компілятору виконувати статичний аналіз у нулабельному контексті.

Переваги локальних функцій над анонімними функціями

- Продуктивність.
 - При створенні лямбди повинен створюватись делегат, що є в такому випадку unnecessary allocation.
 - Локальні функції – просто функції без делегатів, вони більш ефективні при захопленні локальних змінних: лямбди зазвичай захоплюють змінні в клас, а локальні функції можуть застосовувати структуру (передану за допомогою ref), що також avoids an allocation.
 - Звідси, виклик локальних функцій дешевший, вони можуть вбудовуватись (inline).
- Локальні функції можуть бути рекурсивними.
 - Лямбди теж можуть бути рекурсивними, проте відповідний код для анонімних методів буде досить дивний.
- Локальні функції можуть бути узагальненими (generic).
 - Лямбди не можуть бути узагальненими, оскільки вони повинні присвоюватись змінній конкретного типу.
- Локальні функції можуть реалізовуватись як ітератор.
 - Лямбди не можуть використовувати ключове слово yield return (і yield break), щоб реалізувати IEnumerable<T>-повертаючу функцію.

Переваги локальних функцій над анонімними функціями

- Локальні функції більш природно та лаконічно виглядають, ніж присвоєння лямбди делегату:
 - `int add(int x, int y) => x + y;`
 - `Func<int, int, int> add = (x, y) => x + y;`
- Локальні функції можуть визначатись будь-де у функції, навіть після оператора `return`.

```
public double DoMath(double a, double b)
{
    var resultA = f(a);
    var resultB = f(b);
    return resultA + resultB;

    double f(double x) => 5 * x + 3;
}
```

Анонімні типи в C#

- **Анонімний тип** – це тип без назви, який містить тільки readonly-властивості.
 - Він не може містити інші члени: поля, методи, події тощо.
 - Створюється за допомогою оператора new разом з ініціалізатором об'єкта.
 - Ключове слово var використовується для тримання посилань на анонімні типи:

```
var student = new { Id = 1, FirstName = "James", LastName = "Bond" };
```

- Властивості анонімних типів не можуть ініціалізуватись null-значеннями, анонімною функцією чи вказівником.
 - Для доступу використовується стандартний оператор «.»

```
Console.WriteLine(student.Id); //виведе: 1
Console.WriteLine(student.FirstName); //виведе: James
Console.WriteLine(student.LastName); //виведе: Bond
student.Id = 2; //Error: cannot chage value
student.FirstName = "Steve"; //Error: cannot chage value
```

Анонімні типи в C#

- Доступне створення анонімних масивів:

```
var students = new[] {  
    new { Id = 1, FirstName = "James", LastName = "Bond" },  
    new { Id = 2, FirstName = "Steve", LastName = "Jobs" },  
    new { Id = 3, FirstName = "Bill", LastName = "Gates" }  
};
```

- Анонімний тип завжди буде локальний для методу, де він був визначений.
 - Також анонімний тип не можна повертати з методу.
- Переважно анонімні типи створюються за допомогою фрази `Select` у LINQ-запитах з метою повернення підмножини властивостей з кожного об'єкта в колекції.
 - Всередині всі анонімні типи напряду породжені від класу `System.Object`.
 - Компілятор генерує клас з деякою автозгенерованою назвою та застосовує доречний тип до кожної властивості на основі значення виразу (value expression).
 - Проте ваш код доступу до цього класу не має.

```
<>f__AnonymousType0`3[System.Int32,System.String,System.String]
```

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        var student = new { id = 1, FirstName = "James", LastName = "Bond" };  
        Console.WriteLine(student.GetType().ToString());  
    }  
}
```




ДЯКУЮ ЗА УВАГУ!

Наступне питання: Параметричний поліморфізм. Узагальнені типи даних

-
-
- <https://www.slideshare.net/yu5k3/delegates-and-events-in-c>