Кам’янець-Подільський національний університет імені Івана Огієнка

КАФЕДРА КОМП’ЮТЕРНИХ НАУК

Навчальна дисципліна «Об’єктно-орієнтоване програмування»

ЛАБОРАТОРНА РОБОТА #0109

Тема:

Делегати і події

Варіант №3

**Виконала**:  
студентка 1-го курсу  
групи KNms1-B24  
Некрасова Ю.Д.

**Прийняв**:  
Слободянюк О.В.

Кам’янець-Подільський – 2025

1. **Короткі теоретичні відомості**

**Делегати**

Почнемо з визначення терміну *делегат*(delegate). Делегат – це об'єкт, який може посилатися на метод. Таким чином, створюючи делегата, ви, по суті, створюєте об'єкт, який може містити посилання на метод. Більш того, цей метод можна викликати за допомогою відповідного посилання. Таким чином, делегат може викликати метод, на який він посилається.

Делегат в C# в деякому роді аналогічний покажчику на функцію в C/C++.

***Делегат***задає визначення функціонального типу (класу) даних. Екземплярами класу є функції. Опис *делегата*в мові C# є описом ще одного окремого випадку класу. Кожен *делегат*описує множину функцій із заданою сигнатурою. Кожна функція (метод), сигнатура якої співпадає з сигнатурою *делегата*, може розглядатися як екземпляр класу, заданого *делегатом*. Синтаксис оголошення *делегата*має наступний вигляд:

[<специфікатор доступу>] delegate <тип результату > <ім'я класу> (<список аргументів>);

Цим оголошенням класу задається функціональний тип - множина функцій із заданою сигнатурою, у яких аргументи визначаються списком, заданим в оголошенні *делегата*, і тип значення, що повертається, визначається типом результату *делегата*.

Специфікатор доступу може бути, як завжди, опущений. Де слід розміщувати оголошення *делегата*? Як і у всякого класу, є дві можливості:

* безпосередньо у просторі імен, разом з оголошеннями інших класів, структур, інтерфейсів;
* усередині іншого класу, разом з оголошеннями методів і властивостей. Таке оголошення розглядається як оголошення вкладеного класу.

Так само, як і інтерфейси C#, делегати не задають реалізації. Фактично між деякими класами і делегатом полягає контракт на реалізацію делегата. Класи, згодні, з контрактом повинні оголосити у себе статичні або динамічні функції, сигнатура яких співпадає з сигнатурою делегата. Якщо контракт виконується, то можна створити екземпляри делегата, привласнивши їм як значення функції, що задовольняють контракту. Відмітьте, контракт є жорстким: не допускається ситуація, при якій у делегата тип параметра, - object, а у екземпляра відповідний параметр має тип, узгоджений з object, наприклад, int.

Одне з найбільш важливих застосувань *делегатів*пов'язане з *функціями вищих порядків*. ***Функцією вищого порядку***називається така функція (метод) класу, у якої один або декілька аргументів належать до функціонального типу. Без цих функцій в програмуванні обійтися досить важко. Класичним прикладом є функція обчислення інтеграла, у якої один з аргументів задає підінтегральну функцію. Іншим прикладом може служити функція, що сортує об'єкти. Аргументом її є функція Compare, що порівнює два об'єкти. Залежно від того, яка функція порівняння буде передана на вхід функції сортування, об'єкти сортуватимуться по-різному, наприклад, по імені, або по ключу, або по декількох полях. Використання функції вищого порядку приведене в Прикладі #5 Завдання 1.

**Операції над делегатами. Клас Delegate.**

При визначенні функціонального типу, наприклад:

public delegate int FType(int X);

змінна FType належить *класу Delegate*. Чому ж її не можна оголосити звичним чином? Справа не тільки в синтаксичних особливостях цього класу. Річ у тому, що *клас Delegate*є абстрактним класом. Ось його оголошення:

public abstract class Delegate: ICloneable, ISerializable

Для абстрактних класів реалізація не визначена, і це означає, що не можна створювати екземпляри класу. *Клас Delegate*слугує базовим класом для класів-спадкоємців. Але створювати спадкоємців можуть тільки компілятори і системні програми – цього не можна зробити в програмі на C#. Саме тому введено ключове слово delegate, яке побічно дозволяє працювати з *класом Delegate*, створюючи вже не абстрактний, а реальний клас. Відмітьте, при цьому всі динамічні і статичні методи *класу Delegate*стають доступними програмістові.

У делегатів є одна чудова властивість - їх можна комбінувати. Уявіть собі, що існує список робіт, які потрібно виконувати, залежно від обставин, в різних комбінаціях. Якщо функції, що виконують окремі роботи, належать одному класу, то для вирішення завдання можна використовувати *делегатів* і використовувати *техніку їх комбінування*. Можливість *комбінування делегатів* з'явилася, в першу чергу, для підтримки роботи з подіями. Коли виникає деяка подія, то повідомлення про нього посилається різним об'єктам, кожний з яких по-своєму обробляє подію. Реалізується ця можливість на основі *комбінування делегатів*, званою інакше *багатоадресною передачею* (multicasting).

У чому суть *комбінування делегатів*? Вона полягає в наступному. До *екземпляра делегата* дозволяється по черзі приєднувати інші *екземпляри делегата*того ж типу (за допомогою методу Combine() або операції “+=”). Оскільки кожен екземпляр зберігає посилання на функцію, то в результаті створюється список посилань. Цей список називається ***списком викликів (invocation list)***. Коли викликається екземпляр, що має *список виклику*, то по черзі, в порядку приєднання, починають викликатися і виконуватися функції, задані посиланнями. Так один виклик породжує виконання списку робіт (дивися приклад #3 Завдання 1).

Зрозуміло, що, якщо є операція приєднання *делегатів*, то повинна бути і зворотна операція, що дозволяє видаляти *делегатів*із списку. Вона здійснюється за допомогою методу Remove() або операції “-=”.

Розглянемо основні *методи і властивості класу Delegate*. Почнемо з двох статичних методів – Combine і Remove. Перший з них приєднує *екземпляри делегата*до списку, другий - видаляє із списку. Обидва методи мають схожий синтаксис:

Combine(del1, del2)

Remove(del1, del2)

Аргументи del1 і del2 повинні бути одного функціонального класу. При додаванні del2 в список, в якому del2 вже присутній, буде доданий другий екземпляр. При спробі видалення del2 із списку, в якому del2 немає, Remove благополучно завершить роботу, не видаючи повідомлення про помилку.

*Клас Delegate*відноситься до незмінних класів, тому обидва методи повертають посилання на нового *делегата*. Посилання, що повертається, належить батьківському *класу Delegate*, тому його необхідно явно перетворити до потрібного типу, якому належать del1 і del2. Звичайне використання цих методів має вигляд:

del1 = (<type>) Combine(del1, del2);

del1 = (<type>) Remove(del1, del2);

Метод GetInvocationList є динамічним методом класу - він повертає *список викликів*екземпляра, що викликав метод. Потім можна влаштувати цикл foreach, по черзі отримуючи елементи списку.

**Події**

На основі делегатів побудований ще один важливий засіб C#: *подія*(event). Подія – це, по суті, автоматичне повідомлення про виконання деякої дії. Події працюють таким чином. Об'єкт, якому необхідна інформація про деяку подію, реєструє обробник для цієї події. Коли очікувана подія відбувається, викликаються всі зареєстровані обробники. Обробники подій представляються делегатами.

Події – це члени класу, які оголошуються з використанням ключового слова event. Найбільш поширена форма оголошення події має наступний вигляд:

event подієвий\_делегат об'єкт;

Тут елемент *подієвий\_делегат*означає ім'я делегата, використовуваного для підтримки оголошеної події, а елемент *об'єкт*– це ім'я створюваного подієвого об'єкту.

Оголошення події - це двохетапний процес:

* Спочатку оголошується делегат - функціональний клас, що задає сигнатуру. Як наголошувалося при розгляді делегатів, оголошення делегата може бути поміщене в деякий клас, наприклад, *клас Sender*. Але, найчастіше, це оголошення знаходиться поза класом в просторі імен. Оскільки одна і та ж сигнатура може бути у різних подій, то для них досить мати одного делегата. Для деяких подій можна використовувати стандартних делегатів, вбудованих в каркас. Тоді достатньо знати тільки їх імена.
* Якщо делегат визначений, то в *класі Sender*, що створює події, досить оголосити подію як екземпляр відповідного делегата. Це робиться точно так, як і при оголошенні функціональних екземплярів делегата. Виключенням є додавання службового слова event. Формальний синтаксис оголошення такий:

[атрибути] [модифікатори] event [тип, заданий делегатом] [ім'я події]

Найчастіше, атрибути не задаються, а модифікатором є модифікатор доступу - public. Приведемо приклад оголошення *делегата і події*, що представляє екземпляр цього делегата:

namespace Events

{

public delegate void FireEventHandler(object Sender, int time, int build);

public class TownWithEvents

{

public event FireEventHandler FireEvent;

...

}//TownWithEvents

...

}//namespace Events

Тут делегат FireEventHandler описує клас подій, сигнатура яких містить три аргументи. Подія FireEvent в класі TownWithEvents є екземпляром класу, заданого делегатом.

**Як генеруються події**

Причини виникнення події можуть бути різними. Тому цілком імовірно, що одна і та ж подія “запалюватиметься” в різних методах класу в той момент, коли виникне одна з причин появи події. Оскільки дії по включенню можуть повторюватися, корисно до складу методів класу додати захищену процедуру, що генерує подію. Навіть якщо подія “запалюється” тільки в одній точці, написання такої процедури вважається ознакою хорошого стилю. Цій процедурі зазвичай дається ім'я, що починається із слова On, після якого слідує ім'я події. Називатимемо таку процедуру Onпроцедурою. Вона проста і складається з виклику оголошеної події, включеної в тест, який перевіряє перед викликом, а чи є хоч один обробник події, здатний прийняти відповідне повідомлення. Якщо таких немає, то нічого включати подію. Розглянемо приклад:

protected virtual void OnFireEvent(int time, int build)

{

if (FireEvent != null) FireEvent(this, time, build);

}

Зверніть увагу: ті, хто приймає повідомлення про подію, повинні наперед приєднати обробники подій до об'єкту FireEvent, задаючого подію. Приєднання обробників повинне передувати запаленню події. При такому нормальному ході речей, знайдеться хоч би один слухач повідомлення про подію - отже, FireEvent не буде рівне null.

Продемонструємо використання простої події.

using System;

delegate void MyEventHandler(); // Оголошуємо делегат для події

class MyEvent

{

public event MyEventHandler SomeEvent;

//Цей метод викликається для генерування події

public void OnSomeEvent()

{

if (SomeEvent != null) SomeEvent();

}

}

class EventDemo

{

static void handler()

{ // Обробник події

Console.WriteLine("Відбулася подія.");

}

public static void Main()

{

MyEvent evt = new MyEvent();

// Додаємо метод handler() у список обробників події

evt.SomeEvent += new MyEventHandler(handler);

// Генеруємо подію

evt.OnSomeEvent();

}

}

1. **Умова завдання 1.**

Виконати завдання згідно свого варіанту (3). Розробити клас для обчислення визначених інтегралів за формулою правих прямокутників з використанням делегатів.

1. **Послідовність виконання роботи**

* **Створення делегату** MathFunction **для передачі математичної функції** − дозволяє передавати будь-яку функцію, що приймає double і повертає double.
* **Створення класу** Integral **для обчислення інтегралу** − клас містить необхідні поля, конструктор та метод обчислення.
* **Реалізація методу** Calculate() **у класі** Integral − виконує обчислення інтегралу методом правих прямокутників.
* **Опис функції** Function **для прикладу обчислення** − приклад функції для інтегрування (наприклад, f(x)=x2).
* **Задання вхідних даних у методі** Main — межі інтегрування, кількість прямокутників, виклик методу обчислення.
* **Створення об'єкта** Integral **та виклик методу** Calculate() − передається приклад функції через делегат.
* **Виведення результату на екран** − показуються функція, метод, межі, кількість кроків і обчислене значення.

1. **Код програми**

using System;

namespace IntegralCalculator

{

public delegate double MathFunction(double x); // Делегат для математичної функції

public class Integral

{

private double a, b; // Межі інтегрування

private int n; // Кількість прямокутників

private MathFunction func;

public Integral(double a, double b, int n, MathFunction func) // Конструктор класу

{

this.a = a;

this.b = b;

this.n = n;

this.func = func;

}

public double Calculate() // Метод обчислення визначеного інтегралу методом правих прямокутників

{

double h = (b - a) / n; // ширина одного прямокутника

double sum = 0.0;

for (int i = 1; i <= n; i++) // Обчислення суми площ прямокутників

{

double x = a + i \* h; // праві кінці підінтервалів

sum += func(x); // значення функції у правій точці

}

return h \* sum;

}

}

class Program

{

static double Function(double x) // Функція, яку інтегруємо: f(x) = x^2

{

return x \* x;

}

static void Main(string[] args)

{

double lowerLimit = 0; // Вхідні дані

double upperLimit = 2;

int rectangles = 100;

Console.WriteLine("Function: f(x) = x^2");

Console.WriteLine("Method: Right rectangles");

Console.WriteLine($"Interval: from {lowerLimit} to {upperLimit}");

Console.WriteLine($"Number of rectangles: {rectangles}");

Integral integral = new Integral(lowerLimit, upperLimit, rectangles, Function); // Створення об'єкта інтегралу і обчислення

double result = integral.Calculate();

Console.WriteLine($"Calculated integral value: {result:F6}"); // Виведення результату

}

}

}

1. **Приклад виконання програми**

Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.

1. **Умова завдання 2.**

Вивчити і реалізувати Завдання #1 - #5, в яких використовуються делегати відповідно до варіанту (3).

//Приклад #3: Multicasting + Delegate Operations

public delegate void Fun(int x, int y); // Оголошення делегата, який представляє метод з двома параметрами типу int

class TestMain

{

private static void Sum(int x, int y) // Метод для виведення суми (імітація, не обчислення)

{

Console.WriteLine("Called Sum " + x + "," + y);

}

private static void Prod(int x, int y) // Метод для виведення добутку (імітація, не обчислення)

{

Console.WriteLine("Called Product " + x + "," + y);

}

public static void Main()

{

Fun x = new Fun(Sum); // Створення делегатів і прив’язка до методів

Fun y = new Fun(Prod);

x += y; // Додавання методу Prod до ланцюжка виклику x

Console.WriteLine("Calling x(20, 30):");

x(20, 30); // Викликає Sum(20,30) і Prod(20,30)

x += x; // Подвоєння делегата: x тепер викликає всі методи двічі

Console.WriteLine("\nCalling x(10, 20):");

x(10, 20); // Викликає Sum, Prod, Sum, Prod

x -= y; // Видалення методу Prod із делегата

x -= y;

x -= y; // Третє видалення не вплине, бо Prod вже видалено

Console.WriteLine("\nCalling x(1, 1):");

x(1, 1); // Лише Sum(1,1)

}

}

1. **Аналіз коду**

* Оголошення делегата Fun, який представляє будь-який метод з параметрами int, int і типом повернення void:

***public delegate void Fun(int x, int y);***

* Опис методів, які відповідають делегату (вони не обчислюють результат, лише демонструють виклик):

***private static void Sum(int x, int y)***

***private static void Prod(int x, int y)***

Sum — виводить суму-імітацію (Called Sum x,y).

Prod — виводить добуток-імітацію (Called Product x,y).

* Ініціалізація делегатів: ***x*** тепер вказує на метод Sum, ***y*** − на метод Prod:

***Fun x = new Fun(Sum);***

***Fun y = new Fun(Prod);***

* Об’єднання делегатів:

***x += y;***

***x(20, 30);***

До делегата ***x*** додається метод Prod − тепер ***x*** викликає і Sum, і Prod.

* Подвоєння делегата:

***x += x;***

***x(10, 20);***

Делегат ***x*** додає сам себе − його список викликів подвоюється: Sum, Prod, Sum, Prod.

* Видалення методів із делегата:

***x -= y; x -= y; x -= y;***

***x(1, 1);***

Видалення Prod тричі. Після цього залишиться тільки Sum.

1. **Приклад виконання програми**

Зображення, що містить текст, знімок екрана, програмне забезпечення, Мультимедійне програмне забезпечення

Вміст, створений ШІ, може бути неправильним.