ЛАБОРАТОРНА РОБОТА № 6

Тема: ДЕЛЕГАТИ. ПОДІЇ. АНОНІМНІ МЕТОДИ. ЛЯМБДА-ОПЕРАТОРИ. ЛЯМБДА-ВИРАЗИ.

Мета: придбати практичні навички роботи створення та використання делегатів, анонімних методи, лямбда-операторів, лямбда-виразів.

Постановка задачі:

До програмного коду попередніх лабораторних робіт добавити та описати:

1. класи-делегати (мінімум три делегата);
2. події;
3. анонімні методи;
4. лямбда –оператори;
5. лямбда –вирази;
6. протестувати програму.
7. зробити висновки.

ХІД ВИКОНАННЯ РОБОТИ

1 У пункті меню «***Тема, мета та постановка задачі ЛР №6*** **»** звітного HTML-документу розмістити :

1.1 Тему лабораторної роботи №6
1.2 Мету лабораторної роботи №6
1.3 Постановку задачі лабораторної роботи №6
1.4 Список делегатів та подій. Написати призначення делегатів та подій.

2 У пункті меню «*Місце розташування проекту.Середовище розробки*» вказаним **місце розташування проекту** (шлях до папки з проектом). Це може бути GitHub або GoogeDisk. Також написати **середовище розробки програми.**

3 У пункті меню « *Кодування ПЗ (програмування)» розмістити програмні коди та результати їх виконання :*

3.1 Класів – делегатів;
3.2 Подій;
3.3 Анонімних методів;
3.4 Лямбда–операторів;
3.5 Лямбда –виразів;

4 У пункті меню «*Впровадження та супроводження ПЗ*» розмістити посилання на папку з проектом, виконувальним файлом.exe, з можливістю відкриття проекту зі звіту.

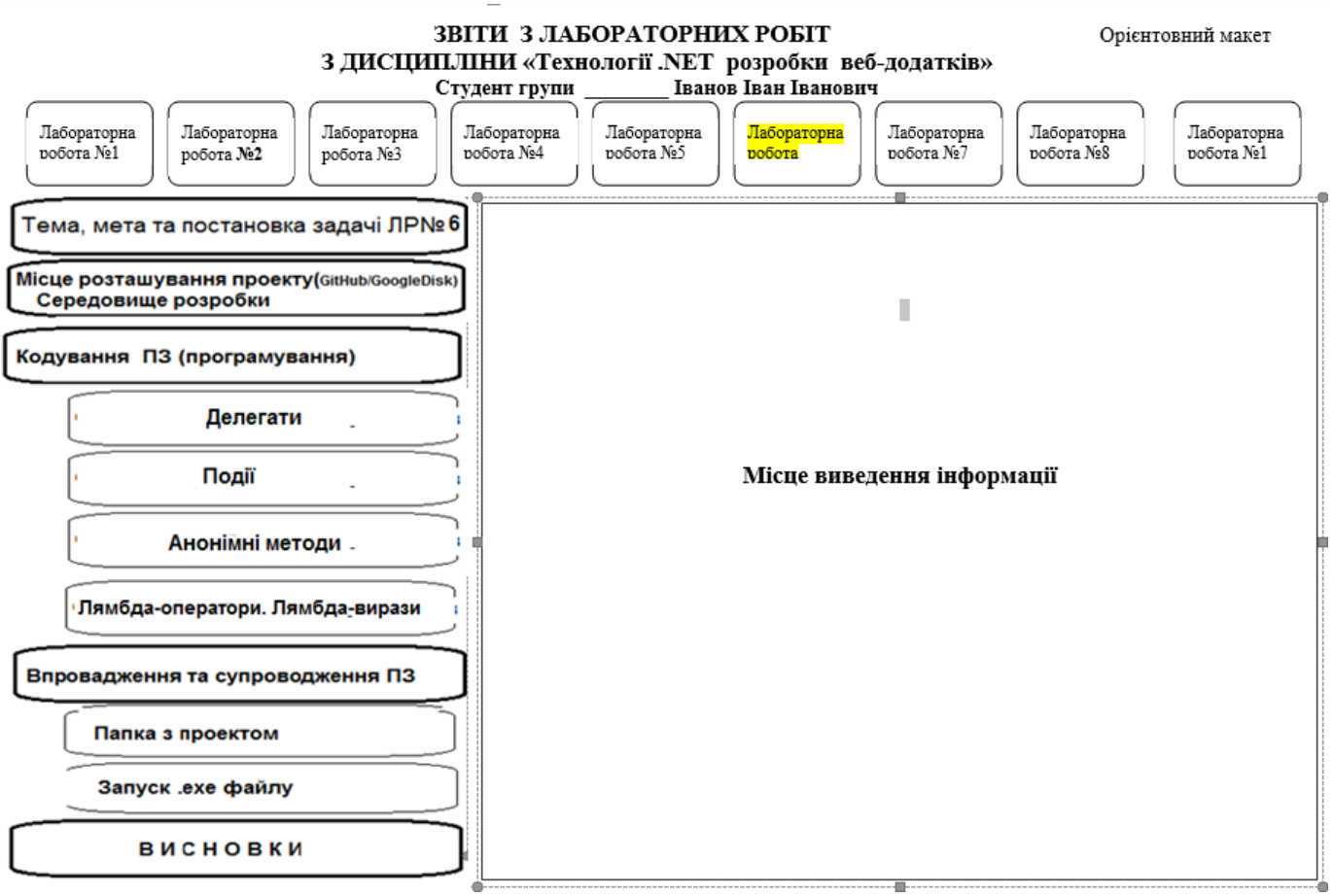
5 У пункті меню «*ВИСНОВКИ* »написати висновки по лабораторній роботі №6

6 Самостійно відпрацювати матеріал, приведенний у посібникау Герберта

Шилдта «C# 4.0» Полное руководство.

Відпрацювати матеріал за посиланням https://metanit.com/sharp/tutorial/

7 Для оформлення звітного HTML-документу приведен орієнтовний макет.



ПРИМІТКА:

Звіт з лабораторних робіт слід підготувати у вигляді **гіпертекстового документа у форматі html.** Документ має містити меню, яке включає команди, що подані нижче. Слід реалізувати запуск програм на виконання з гіпертекстового документа.
Звіт та проекти лабораторних робіт слід записати на Гугл-Диск а. На диску має бути файл readme.txt, який містить відомості про автора звіту та проектів.

Титульна html- сторінка.

Назва роботи
Автор (ПІБ, група, курс, № заліковки)
Фото
Рік навчання

ТЕОРЕТИЧНІ ВІДОМОСТІ

Делегати представляють такі об'єкти, які вказують на методи. Тобто делегати - це покажчики на методи і за допомогою делегатів ми можемо викликати дані методи.

Визначення делегатів

Для оголошення делегата використовується ключове слово **delegate** , після якого йде повертається тип, назва і параметри. наприклад:

```
1    delegate void Message();
```

Делегат Message як повертається типу має тип void (тобто нічого не повертає) і не приймає ніяких параметрів. Це означає, що цей делегат може вказувати на будь-який метод, який не приймає ніяких параметрів і нічого не повертає.

Розглянемо примение цього делегата:

```
1    class Program
2    {
3        delegate void Message(); // 1. Объявляем делегат
4
5        static void Main(string[] args)
6        {
7            Message mes; // 2. Создаем переменную делегата
8            if (DateTime.Now.Hour < 12)
9            {
10               mes = GoodMorning; // 3. Присваиваем этой переменной адрес метода
11           }
12           else
13           {
14               mes = GoodEvening;
15           }
16           mes(); // 4. Вызываем метод
17           Console.ReadKey();
18       }
19       private static void GoodMorning()
20       {
21           Console.WriteLine("Good Morning");
22       }
23       private static void GoodEvening()
24       {
25           Console.WriteLine("Good Evening");
26       }
27   }
```

Тут спочатку ми визначаємо делегат:

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

В даному випадку делегат визначається всередині класу, але також можна визначити делегат поза класом всередині простору імен.

Для використання делегата оголошується змінна цього делегата:

```
1    Message mes; // 2. Создаем переменную делегата
```

За допомогою властивості `DateTime.Now.Hour` отримуємо поточну годину. І в залежності від часу в делегат передається адреса певного методу. Зверніть увагу, що методи ці мають той же повертається значення і той же набір параметрів (в даному випадку відсутність параметрів), що і делегат.

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

Потім через делегат викликаємо метод, на який посилається даний делегат:

```
mes(); // 4. Вызываем метод
```

Виклик делегата проводиться подібно викликом методу.

Подивимося на прикладі іншого 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 даному випадку делегат Operation повертає значення типу int і має два параметри типу int. Тому цього делегату відповідає будь-який метод, який повертає значення типу int і приймає два параметри типу int. В даному випадку це методи Add і Multiply. Тобто ми можемо привласнити змінної делегата будь-який з цих методів і викликати.

Оскільки делегат бере два параметра типу int, то при його виклику необхідно передати значення для цих параметрів: `del(4,5)`.

Делегати необов'язково можуть вказувати тільки на методи, які визначені в тому ж класі, де визначена змінна делегата. Це можуть бути також методи з інших класів і 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

```

```
16          Console.Read();
17      }
18  }
```

Присвоєння посилання на метод

Вище змінної делегата безпосередньо присвоювався метод. Є ще один спосіб - створення об'єкта делегата за допомогою конструктора, в який передається потрібний метод:

```
1   class Program
2   {
3       delegate int Operation(int x, int y);
4
5       static void Main(string[] args)
6       {
7           Operation del = Add;
8           Operation del2 = new Operation(Add);
9
10          Console.Read();
11      }
12      private static int Add(int x, int y) { return x + y; }
13  }
```

Обидва способи рівноцінні.

Відповідність методів делегату

Як було написано вище, методи відповідають делегату, якщо вони мають один і той же повертається тип і один і той же набір параметрів. Але треба враховувати, що до уваги також приймаються модифікатори **ref** і **out** . Наприклад, нехай у нас є делегат:

```
1   delegate void SomeDel(int a, double b);
```

Цьому делегату відповідає, наприклад, наступний метод:

```
1   void SomeMethod1(int g, double n) { }
```

А такі методи НЕ відповідають:

```
1   int SomeMethod2(int g, double n) { }
2   void SomeMethod3(double n, int g) { }
3   void SomeMethod4(ref int g, double n) { }
4   void SomeMethod5(out int g, double n) { g = 6; }
```

`Тут метод SomeMethod2 має інший повертається тип, відмінний від типу делегата. SomeMethod3` має інший набір параметрів. Параметри SomeMethod4 і SomeMethod5 також відрізняються від параметрів делегата, оскільки мають модифікатори ref і out.

Додавання методів в делегат

У прикладах вище змінна делегата вказувала на один метод. В реальності ж делегат може вказувати на безліч методів, які мають ту ж сигнатуру і повертаються тип. Всі методи в делегата потрапляють в спеціальний список - список виклику або invocation list. І при виклику делегата все методи з цього списку послідовно викликаються. І ми можемо додавати в цей спіок не один, а кілька методів:

```
1   class Program
2   {
3       delegate void Message();
4
5       static void Main(string[] args)
```

```
6       {
7           Message mes1 = Hello;
8           mes1 += HowAreYou;  // теперь mes1 указывает на два метода
9           mes1(); // вызываются оба метода - Hello и HowAreYou
10          Console.Read();
11      }
12      private static void Hello()
13      {
14          Console.WriteLine("Hello");
15      }
16      private static void HowAreYou()
17      {
18          Console.WriteLine("How are you?");
19      }
20  }
```

В даному випадку в список виклику делегата mes1 додаються два методу - Hello і HowAreYou. І при виклику mes1 викликаються відразу обидва цих методу.

Для додавання делегатів застосовується операція **+ =** . Однак варто зазначити, що в реальності буде відбуватися створення нового об'єкта делегата, який отримає методи старої копії делегата і новий метод, і новий створений об'єкт делегата буде присвоєно змінної mes1.

При додаванні делегатів слід враховувати, що ми можемо додати посилання на один і той же метод кілька разів, і в списку виклику делегата тоді буде кілька посилань на один і той же метод. Відповідно при виклику делегата доданий метод буде викликатися стільки раз, скільки він був доданий:

```
1   Message mes1 = Hello;
2   mes1 += HowAreYou;
3   mes1 += Hello;
4   mes1 += Hello;
5
6   mes1();
```

Консольний висновок:

```
Hello
How are you?
Hello
Hello
```

Подібним чином ми можемо видаляти методи з делегата за допомогою операції **- =** :

```
1    static void Main(string[] args)
2    {
3        Message mes1 = Hello;
4        mes1 += HowAreYou;
5        mes1(); // вызываются все методы из mes1
6        mes1 -= HowAreYou;  // удаляем метод HowAreYou
7        mes1(); // вызывается метод Hello
8
9        Console.Read();
10   }
```

При видаленні методів з делегата фактично буде создаватся новий делегат, який в списку виклику методів буде містити на один метод менше.

При видаленні слід враховувати, що якщо делегат містить кілька посилань на один і той же метод, то операція - = починає пошук з кінця списку виклику делегата і видаляє тільки перше знайдене входження. Якщо подібного методу в списку виклику делегата немає, то операція - = не має ніякого ефекту.

об'єднання делегатів

Делегати можна об'єднувати в інші делегати. наприклад:

```
1    class Program
2    {
3        delegate void Message();
4
5        static void Main(string[] args)
6        {
7            Message mes1 = Hello;
8            Message mes2 = HowAreYou;
9            Message mes3 = mes1 + mes2; // объединяем делегаты
10           mes3(); // вызываются все методы из mes1 и mes2
11
12           Console.Read();
13       }
14       private static void Hello()
15       {
16           Console.WriteLine("Hello");
17       }
18       private static void HowAreYou()
19       {
20           Console.WriteLine("How are you?");
21       }
22   }
```

В даному випадку об'єкт mes3 представляє об'єднання делегатів mes1 і mes2. Об'єднання делегатів значить, що в список виклику делегата mes3 потраплять всі методи з делегатів mes1 і mes2. І при виклику делегата mes3 всі ці методи одночасно будуть викликані.

Виклик делегата

У прикладах вище делегат викликався як звичайний метод. Якщо делегат брав параметри, то при її виклику для параметрів передавалися необхідні значення:

```
1    class Program
2    {
3        delegate int Operation(int x, int y);
4        delegate void Message();
5
6        static void Main(string[] args)
7        {
8            Message mes = Hello;
9            mes();
10           Operation op = Add;
11           op(3, 4);
12           Console.Read();
13       }
14       private static void Hello() { Console.WriteLine("Hello"); }
15       private static int Add(int x, int y) { return x + y; }
16   }
```

Інший спосіб виклику делегата представляє метод **Invoke ()** :

```
1    class Program
2    {
3        delegate int Operation(int x, int y);
4        delegate void Message();
5
6        static void Main(string[] args)
7        {
8            Message mes = Hello;
```

```
9           mes.Invoke();
10          Operation op = Add;
11          op.Invoke(3, 4);
12          Console.Read();
13      }
14      private static void Hello() { Console.WriteLine("Hello"); }
15      private static int Add(int x, int y) { return x + y; }
16  }
```

Якщо делегат бере параметри, то в метод Invoke передаються значення для цих параметрів.

Слід враховувати, що якщо делегат порожній, тобто в його списку виклику немає посилань ні на один з методів (тобто делегат дорівнює Null), то при виклику такого делегата ми отримаємо виняток, як, наприклад, в наступному випадку:

```
1   Message mes = null;
2   //mes();        // ! Ошибка: делегат равен null
3
4   Operation op = Add;
5   op -= Add;      // делегат op пуст
6   op(3, 4);       // !Ошибка: делегат равен null
```

Тому при виклику делегата завжди краще перевіряти, чи не рівний він null. Або можна використовувати метод Invoke і оператор умовного null:

```
1   Message mes = null;
2   mes?.Invoke();        // ошибки нет, делегат просто не вызывается
3
4   Operation op = Add;
5   op -= Add;          // делегат op пуст
6   op?.Invoke(3, 4);   // ошибки нет, делегат просто не вызывается
```

Якщо делегат повертає деяке значення, то повертається значення останнього методу зі списку виклику (якщо в списку виклику кілька методів). наприклад:

```
1    class Program
2    {
3        delegate int Operation(int x, int y);
4
5        static void Main(string[] args)
6        {
7            Operation op = Subtract;
8            op += Multiply;
9            op += Add;
10           Console.WriteLine(op(7, 2));    // Add(7,2) = 9
11           Console.Read();
12       }
13       private static int Add(int x, int y) { return x + y; }
14       private static int Subtract(int x, int y) { return x - y; }
15       private static int Multiply(int x, int y) { return x * y; }
16   }
```

Делегати як параметри методів

Також делегати можуть бути параметрами методів:

```
1    class Program
2    {
3        delegate void GetMessage();
4
5        static void Main(string[] args)
6        {
7            if (DateTime.Now.Hour < 12)
```

```
8           {
9               Show_Message(GoodMorning);
10          }
11          else
12          {
13               Show_Message(GoodEvening);
14          }
15          Console.ReadLine();
16      }
17      private static void Show_Message(GetMessage _del)
18      {
19          _del?.Invoke();
20      }
21      private static void GoodMorning()
22      {
23          Console.WriteLine("Good Morning");
24      }
25      private static void GoodEvening()
26      {
27          Console.WriteLine("Good Evening");
28      }
29  }
```

Узагальнені делегати

Делегати можуть бути узагальненими, наприклад:

```
1   delegate T Operation<T, K>(K val);
2
3   class Program
4   {
5       static void Main(string[] args)
6       {
7           Operation<decimal, int> op = Square;
8
9           Console.WriteLine(op(5));
10          Console.Read();
11      }
12
13      static decimal Square(int n)
14      {
15          return n * n;
16      }
17  }
```

Застосування делегатів

У минулій темі докладно були розглянуті делегати. Однак дані приклади, можливо, не показують справжньої сили делегатів, так як потрібні нам методи в даному випадку ми можемо викликати і безпосередньо без всяких делегатів. Однак найбільш сильна сторона делегатів полягає в тому, що вони дозволяють делегувати виконання деякого коду ззовні. І на момент написання програми ми можемо не знати, що за код буде виконуватися. Ми просто викликаємо делегат. А який метод буде безпосередньо виконуватися при виклику делегата, буде вирішуватися потім. Наприклад, наші класи будуть поширюватися у вигляді окремої бібліотеки класів, яка буде підключатися до проекту іншого розробника. І цей розробник захоче визначити якусь свою логіку обробки, але змінити вихідний код нашої бібліотеки класів він не може.

Розглянемо детальний приклад. Нехай у нас є клас, що описує рахунок в банку:

```
1 class Account
2 {
3     int _sum; // Переменная для хранения суммы
```

```
4
5     public Account(int sum)
6     {
7         _sum = sum;
8     }
9
10    public int CurrentSum
11    {
12        get { return _sum; }
13    }
14
15    public void Put(int sum)
16    {
17        _sum += sum;
18    }
19
20    public void Withdraw(int sum)
21    {
22        if (sum <= _sum)
23        {
24            _sum -= sum;
25        }
26    }
27}
```

Припустимо, в разі виведення грошей за допомогою методу `Withdraw`нам треба якось повідомляти про це самого клієнта і, може бути, інші об'єкти. Для цього створимо делегат **AccountStateHandler** . Щоб використовувати делегат, нам треба створити змінну цього делегата, а потім привласнити йому метод, який буде викликатися делегатом.

Отже, додамо в клас Account наступні рядки:

```
1 class Account
2 {
3     // Объявляем делегат
4     public delegate void AccountStateHandler(string message);
5     // Создаем переменную делегата
6     AccountStateHandler _del;
7
8     // Регистрируем делегат
9     public void RegisterHandler(AccountStateHandler del)
10    {
11        _del = del;
12    }
13
14    // Далее остальные строки класса Account
```

Тут фактично проробляються ті ж кроки, що були вище, і є практично все крім виклику делегата. В даному випадку у нас делегат бере параметр типу string. Тепер змінимо метод `Withdraw`наступним чином:

```
1 public void Withdraw(int sum)
2 {
3     if (sum <= _sum)
4     {
5          _sum -= sum;
6
7         if (_del != null)
8             _del($"Сумма {sum} снята со счета");
9     }
10    else
11    {
```

```
12         if (_del != null)
13             _del("Недостаточно денег на счете");
14     }
15}
```

Тепер при знятті грошей через метод Withdraw ми спочатку перевіряємо, чи має делегат посилання на який-небудь метод (інакше він має значення `null`). І якщо метод встановлений, то викликаємо його, передаючи відповідне повідомлення в якості параметра.

Тепер протестуємо клас в основній програмі:

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         // создаем банковский счет
6         Account account = new Account(200);
7         // Добавляем в делегат ссылку на метод Show_Message
8         // а сам делегат передается в качестве параметра метода RegisterHandler
9         account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
10        // Два раза подряд пытаемся снять деньги
11        account.Withdraw(100);
12        account.Withdraw(150);
13        Console.ReadLine();
14    }
15    private static void Show_Message(String message)
16    {
17        Console.WriteLine(message);
18    }
19}
```

Запустивши програму, ми отримаємо два різних повідомлення:

```
Сума 100 знята з рахунку
Недостатньо грошей на рахунку
```

Таким чином, ми створили механізм зворотного виклику для класу Account, який спрацьовує в разі зняття грошей. Оскільки делегат оголошений всередині класу Account, то щоб до нього отримати доступ, використовується вираз `Account.AccountStateHandler`.

Знову ж може виникнути питання: чому б у коді методу `Withdraw()` не виводити повідомлення про зняття грошей? Навіщо потрібно задіяти якийсь делегат?

Справа в тому, що не завжди у нас є доступ до коду класів. Наприклад, частина класів може створюватися і компілюватиметься однією людиною, який не знатиме, як ці класи будуть використовуватися. А використовувати ці класи буде інший розробник.

Так, тут ми виводимо повідомлення на консоль. Однак для класу Account не важливо, як це повідомлення виводиться. Класу Account навіть не відомо, що взагалі буде робитися в результаті списання грошей. Він просто посилає повідомлення про це через делегат.

В результаті, якщо ми створюємо консольний додаток, ми можемо через делегат виводити повідомлення на консоль. Якщо ми створюємо графічне додаток Windows Forms або WPF, то можна виводити повідомлення у вигляді графічного вікна. А можна не просто виводити повідомлення. А, наприклад, записати при списанні інформацію про цю дію в файл або відправити повідомлення на електронну пошту. Загалом будь-якими способами обробити виклик делегата. І спосіб обробки не буде залежати від класу Account.

Хоча в прикладі наш делегат брав адресу на один метод, в дійсності він може вказувати відразу на кілька методів. Крім того, при необхідності ми можемо видалити посилання на адреси певних

методів, щоб вони не викликалися при виклику делегата. Отже, змінимо в класі Account метод RegisterHandler і додамо новий метод UnregisterHandler, який буде видаляти методи зі списку методів делегата:

```
1 // Регистрируем делегат
2 public void RegisterHandler(AccountStateHandler del)
3 {
4     _del += del; // добавляем делегат
5 }
6 // Отмена регистрации делегата
7 public void UnregisterHandler(AccountStateHandler del)
8 {
9     _del -= del; // удаляем делегат
10}
```

У першому методі об'єднує делегати `_del`і `del`в один, який потім присвоюється змінної `_del`. У другому методі видаляється делегат `del`. Тепер перейдемо до основної програми:

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Account account = new Account(200);
6          Account.AccountStateHandler colorDelegate = new
7  Account.AccountStateHandler(Color_Message);
8
9          // Добавляем в делегат ссылку на методы
10         account.RegisterHandler(new Account.AccountStateHandler(Show_Message));
11         account.RegisterHandler(colorDelegate);
12         // Два раза подряд пытаемся снять деньги
13         account.Withdraw(100);
14         account.Withdraw(150);
15
16         // Удаляем делегат
17         account.UnregisterHandler(colorDelegate);
18         account.Withdraw(50);
19
20         Console.ReadLine();
21     }
22     private static void Show_Message(String message)
23     {
24         Console.WriteLine(message);
25     }
26     private static void Color_Message(string message)
27     {
28         // Устанавливаем красный цвет символов
29         Console.ForegroundColor = ConsoleColor.Red;
30         Console.WriteLine(message);
31         // Сбрасываем настройки цвета
32         Console.ResetColor();
33     }
   }
```

З метою тестування ми створили ще один метод - Color_Message, який виводить те ж саме повідомлення тільки червоним кольором. Для першого делегата створюється окрема змінна. Але великої різниці між передачею обох в метод `account.RegisterHandler`немає: просто в одному випадку ми відразу передаємо об'єкт, створюваний конструктором`account.RegisterHandler(new Account.AccountStateHandler(Show_Message));`

У другому випадку створюємо змінну і її вже передаємо в метод `account.RegisterHandler(colorDelegate);`.

У рядку `account.UnregisterHandler(colorDelegate);` цей метод видаляється зі списку дзвінків делегата, тому цей метод більше не буде спрацьовувати. Консольний висновок буде мати наступну форму:

Події

Події сигналізують системі про те, що відбулася певна дія. І якщо нам треба відстежити ці дії, то як раз ми можемо застосовувати події.

Наприклад, візьмемо наступний клас, який описує банківський рахунок:

```
1 class Account
2 {
3     public Account(int sum)
4     {
5         Sum = sum;
6     }
7     // сумма на счете
8     public int Sum { get; private set;}
9     // добавление средств на счет
10    public void Put(int sum)
11    {
12        Sum += sum;
13    }
14    // списание средств со счета
15    public void Take(int sum)
16    {
17        if (Sum >= sum)
18        {
19            Sum -= sum;
20        }
21    }
22}
```

У конструкторі встановлюємо початкову суму, яка зберігається у властивості Sum. За допомогою методу Put ми можемо додати кошти на рахунок, а за допомогою методу Take, навпаки, зняти гроші з рахунку. Спробуємо використовувати клас у програмі - створити рахунок, покласти і зняти з нього гроші:

```
1 static void Main(string[] args)
2 {
3     Account acc = new Account(100);
4     acc.Put(20);    // добавляем на счет 20
5     Console.WriteLine($"Сумма на счете: {acc.Sum}");
6     acc.Take(70);   // пытаемся снять со счета 70
7     Console.WriteLine($"Сумма на счете: {acc.Sum}");
8     acc.Take(180);  // пытаемся снять со счета 180
9     Console.WriteLine($"Сумма на счете: {acc.Sum}");
10    Console.Read();
11}
```

Консольний висновок:

```
Сумма на счете: 120
Сумма на счете: 50
Сумма на счете: 50
```

Всі операції працюють як і належить. Але що якщо ми хочемо повідомляти користувача про результати його операцій. Ми могли б, наприклад, для цього змінити метод Put наступним чином:

```
1public void Put(int sum)
2{
3    Sum += sum;
```

```
4    Console.WriteLine($"На счет поступило: {sum}");
5}
```

Здавалося, тепер ми будемо сповіщені про операції, побачивши відповідне повідомлення на консолі. Але тут є ряд зауважень. На момент визначення класу ми можемо точно не знати, яку дію ми хочемо зробити в методі Put у відповідь на додавання грошей. Це може висновок на консоль, а може бути ми захочемо повідомити користувача по email або sms. Більш того ми можемо створити окрему бібліотеку класів, яка буде містити цей клас, і додавати її в інші проекти. І вже з цих проектів вирішувати, яка мета повинна виконуватися. Можливо, ми захочемо використовувати клас Account в графічному додатку і виводити при додаванні на рахунок в графічному повідомленні, а не консоль. Або нашу бібліотеку класів буде використовувати інший розробник, у якого своя думка, що саме робити при додаванні на рахунок. І всі ці питання ми можемо вирішити,

Визначення і виклик подій

Події оголошуються в класі з допомогою ключового слова **event** , після якого вказується тип делегата, який представляє подія:

```
1 delegate void AccountHandler(string message);
2 event AccountHandler Notify;
```

В даному випадку спочатку визначається делегат AccountHandler, який приймає один параметр типу string. Потім за допомогою ключового слова **event** визначається подія з ім'ям Notify, яке представляє делегат AccountHandler. Назва для події може бути довільною, але в будь-якому випадку вона повинна представляти деякий делегат.

Визначивши подію, ми можемо її викликати в програмі як метод, використовуючи ім'я події:

```
1 Notify("Произошло действие");
```

Оскільки подія Notify представляє делегат AccountHandler, який приймає один параметр типу string - рядок, то при виклику події нам треба передати в нього рядок.

Однак при виклику подій ми можемо зіткнутися з тим, що подія одно null в разі, якщо для його не визначений обробник. Тому при виклику події краще його завжди перевіряти на null. Наприклад, так:

```
1if(Notify !=null) Notify("Произошло действие");
```

Или так:

```
1Notify?.Invoke("Произошло действие");
```

В цьому випадку оскільки подія представляє делегат, то ми можемо його викликати за допомогою методу **Invoke ()** , передавши в нього необхідні значення для параметрів.

Об'єднаймо все разом і створимо і викличемо подію:

Объединим все вместе и создадим и вызовем событие:

```
1 class Account
2 {
3     public delegate void AccountHandler(string message);
4     public event AccountHandler Notify;              // 1.Определение события
5     public Account(int sum)
6     {
7         Sum = sum;
8     }
9     public int Sum { get; private set;}
```

```
10    public void Put(int sum)
11    {
12        Sum += sum;
13        Notify?.Invoke($"На счет поступило: {sum}");   // 2.Вызов события
14    }
15    public void Take(int sum)
16    {
17        if (Sum >= sum)
18        {
19            Sum -= sum;
20            Notify?.Invoke($"Со счета снято: {sum}");   // 2.Вызов события
21        }
22        else
23        {
24            Notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}"); ;
25        }
26    }
27}
```

Тепер за допомогою події Notify ми повідомляємо систему про те, що були додані кошти і про те, що кошти зняті з рахунку або на рахунку недостатньо коштів.

Додавання обробника події

З подією може бути пов'язаний один або кілька обробників. Обробники подій - це саме те, що виконується при виклику подій. Нерідко в якості обробників подій застосовуються методи. Кожен обробник подій за списком параметрів і що повертається типу повинен відповідати делегату, який представляє подія. Для додавання обробника події застосовується операція **+ =** :

```
1Notify += обработчик события;
```

Определим обработчики для события Notify, чтобы получить в программе нужные уведомления:

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         Account acc = new Account(100);
6         acc.Notify += DisplayMessage;   // Добавляем обработчик для события Notify
7         acc.Put(20);    // добавляем на счет 20
8         Console.WriteLine($"Сумма на счете: {acc.Sum}");
9         acc.Take(70);   // пытаемся снять со счета 70
10        Console.WriteLine($"Сумма на счете: {acc.Sum}");
11        acc.Take(180);  // пытаемся снять со счета 180
12        Console.WriteLine($"Сумма на счете: {acc.Sum}");
13        Console.Read();
14    }
15    private static void DisplayMessage(string message)
16    {
17        Console.WriteLine(message);
18    }
19}
```

В даному випадку в якості обробника використовується метод DisplayMessage, який відповідає за списком параметрів і що повертається типу делегату AccountHandler. В результаті при виклику події `Notify?.Invoke()` буде викликатися метод DisplayMessage, якому для параметра message буде передаватися рядок, яка передається в `Notify?.Invoke()`. У DisplayMessage просто виводимо отримане від події повідомлення, але можна було б визначити будь-яку логіку.

Якби в даному випадку обробник не був би встановлений, то при виклику події `Notify?.Invoke()` нічого не відбувалося, так як подія Notify було б так само null.

Консольний висновок програми:

:

```
На счет поступило: 20
Сумма на счете: 120
Со счета снято: 70
Сумма на счете: 50
Недостаточно денег на счете. Текущий баланс: 50
Сумма на счете: 50
```

Тепер ми можемо виділити клас Account в окрему бібліотеку класів і додавати в будь-який проект.

Додавання і видалення обробників

Для однієї події можна встановити кілька обробників і потім в будь-який момент часу їх видалити. Для видалення обробників застосовується операція **- =** . наприклад:

```
1 class Program
2 {
3     static void Main(string[] args)
4     {
5         Account acc = new Account(100);
6         acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage
7         acc.Notify += DisplayRedMessage;    // добавляем обработчик DisplayMessage
8         acc.Put(20);    // добавляем на счет 20
9         acc.Notify -= DisplayRedMessage;     // удаляем обработчик DisplayRedMessage
10        acc.Put(20);    // добавляем на счет 20
11        Console.Read();
12    }
13
14    private static void DisplayMessage(string message)
15    {
16        Console.WriteLine(message);
17    }
18
19    private static void DisplayRedMessage(String message)
20    {
21        // Устанавливаем красный цвет символов
22        Console.ForegroundColor = ConsoleColor.Red;
23        Console.WriteLine(message);
24        // Сбрасываем настройки цвета
25        Console.ResetColor();
26    }
27}
```

Консольний висновок:

```
На рахунок надійшло: 20
На рахунок надійшло: 20
На рахунок надійшло: 20
```

Як оброблювачів можуть використовуватися не тільки звичайні методи, але також делегати, анонімні методи і лямбда-вирази. Використання делегатів і методів:

```
1 static void Main(string[] args)
2 {
3     Account acc = new Account(100);
4     // установка делегата, который указывает на метод DisplayMessage
5     acc.Notify += new ActionHandler(DisplayMessage);
6     // установка в качестве обработчика метода DisplayMessage
7     acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage
```

```
8
9     acc.Put(20);    // добавляем на счет 20
10    Console.Read();
11}
12
13private static void DisplayMessage(string message)
14{
15    Console.WriteLine(message);
16}
```

В даному випадку різниці між двома обработчиками ніякої не буде.

Установка в якості обробника анонімного методу:

```
1 static void Main(string[] args)
2 {
3     Account acc = new Account(100);
4     acc.Notify += delegate (string mes)
5     {
6         Console.WriteLine(mes);
7     };
8
9     acc.Put(20);
10    Console.Read();
11}
```

Установка в якості обробника лямбда-вирази:

```
1static void Main(string[] args)
2{
3    Account acc = new Account(100);
4    acc.Notify += mes =>Console.WriteLine(mes);
5
6    acc.Put(20);
7    Console.Read();
8}
```

Управління обробниками

За допомогою спеціальних акссесоров **add / remove** ми можемо управляти додаванням і видаленням оброблювачів. Як правило, подібна функціональність рідко потрібно, але тим не менше ми її можемо використовувати. наприклад:

```
1 class Account
2 {
3     public delegate void AccountHandler(string message);
4     private event AccountHandler _notify;
5     public event AccountHandler Notify
6     {
7         add
8         {
9             _notify += value;
10            Console.WriteLine($"{value.Method.Name} добавлен");
11        }
12        remove
13        {
14            _notify -= value;
15            Console.WriteLine($"{value.Method.Name} удален");
16        }
17    }
18    public Account(int sum)
19    {
20        Sum = sum;
21    }
```

```
22    public int Sum { get; private set;}
23    public void Put(int sum)
24    {
25        Sum += sum;
26        _notify?.Invoke($"На счет поступило: {sum}");
27    }
28
29    public void Take(int sum)
30    {
31        if (Sum >= sum)
32        {
33            Sum -= sum;
34            _notify?.Invoke($"Со счета снято: {sum}");
35        }
36        else
37        {
38            _notify?.Invoke($"Недостаточно денег на счете. Текущий баланс: {Sum}");
39;
40        }
41    }
  }
```

Тепер опредленія події розбивається на дві частини. Спочатку просто визначається змінна, через яку ми можемо викликати пов'язані обробники:

```
1private event AccountHandler _notify;
```

У другій частині визначаємо акссесори add і remove. Аксессор **add** викликається при додаванні обробника, тобто при операції + =. Додається обробник доступний через ключове слово **value** . Тут ми можемо отримати інформацію про обробнику (наприклад, ім'я методу через value.Method.Name) і визначити деяку логіку. В даному випадку для простоти просто виводиться повідомлення на консоль:

```
1add
2{
3    _notify += value;
4    Console.WriteLine($"{value.Method.Name} добавлен");
5}
```

Блок remove викликається при видаленні обробника. Аналогічно тут можна задати деяку додаткову логіку:

```
1remove
2{
3    _notify -= value;
4    Console.WriteLine($"{value.Method.Name} удален");
5}
```

Усередині класу подія викликається також через змінну _notify. Але для додавання і видалення обробників в програмі використовується як раз Notify:

```
1
2 class Program
3 {
4     static void Main(string[] args)
5     {
6         Account acc = new Account(100);
7         acc.Notify += DisplayMessage;       // добавляем обработчик DisplayMessage
8         acc.Put(20);    // добавляем на счет 20
9         acc.Notify -= DisplayMessage;     // удаляем обработчик DisplayRedMessage
10        acc.Put(20);    // добавляем на счет 20
11
12        Console.Read();
13    }
```

```
14    private static void DisplayMessage(string message) =>
  Console.WriteLine(message);
  }
```

Консольний висновок програми:

```
DisplayMessage доданий
На рахунок надійшло: 20
DisplayMessage видалений
```

Клас даних події AccountEventArgs

Нерідко при виникненні події оброблювачу події потрібно передати деяку інформацію про подію. Наприклад, додамо і в нашу програму новий клас **AccountEventArgs** з наступним кодом:

```
1 class AccountEventArgs
2 {
3     // Сообщение
4     public string Message{get;}
5     // Сумма, на которую изменился счет
6     public int Sum {get;}
7
8     public AccountEventArgs(string mes, int sum)
9     {
10        Message = mes;
11        Sum = sum;
12    }
13}
```

Даний клас має дві властивості: Message - для зберігання виведеного повідомлення і Sum - для зберігання суми, на яку змінився рахунок.

Тепер застосуємо клас AccoutEventArgs, змінивши клас Account наступним чином:

```
  class Account
1 {
2     public delegate void AccountHandler(object sender, AccountEventArgs e);
3     public event AccountHandler Notify;
4     public Account(int sum)
5     {
6         Sum = sum;
7     }
8     public int Sum { get; private set;}
9     public void Put(int sum)
10    {
11        Sum += sum;
12        Notify?.Invoke(this, new AccountEventArgs($"На счет поступило {sum}", sum));
13    }
14    public void Take(int sum)
15    {
16        if (Sum >= sum)
17        {
18            Sum -= sum;
19            Notify?.Invoke(this, new AccountEventArgs($"Сумма {sum} снята со счета",
20sum));
21        }
22        else
23        {
24            Notify?.Invoke(this, new AccountEventArgs("Недостаточно денег на счете",
25sum)); ;
26        }
27    }
  }
```

У порівнянні з попередньою версією класу Account тут змінилося тільки кількість параметрів у делегата і відповідно кількість параметрів при виклику події. Тепер вони також беруть об'єкт AccountEventArgs, який зберігає інформацію про подію, що отримується через конструктор.

Тепер змінимо основну програму:

```
class Program
{
    static void Main(string[] args)
    {
        Account acc = new Account(100);
        acc.Notify += DisplayMessage;
        acc.Put(20);
        acc.Take(70);
        acc.Take(150);
        Console.Read();
    }
    private static void DisplayMessage(object sender, AccountEventArgs e)
    {
        Console.WriteLine($"Сумма транзакции: {e.Sum}");
        Console.WriteLine(e.Message);
    }
}
```

У порівнянні з попереднім варіантом тут ми тільки змінюємо кількість параметрів і сутність їх використання в обробнику DisplayMessage.

Анонімні методи

З делегатами тісно пов'язані **анонімні методи** . Анонімні методи використовуються для створення екземплярів делегатів.

Визначення анонімних методів починається з ключового слова **delegate** , після якого йде в дужках список параметрів і тіло методу в фігурних дужках:

```
    delegate(параметры)
    {
        // инструкции
    }
```

наприклад:

```
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();
    }
}
```

Анонімний метод не може існувати сам по собі, він використовується для ініціалізації примірника делегата, як в даному випадку змінна handler представляє анонімний метод. І через цю змінну делегата можна викликати даний анонімний метод.

Інший приклад анонімних методів - передача в якості аргументу для параметра, який представляє делегат:

```
1 class Program
2 {
3     delegate void MessageHandler(string message);
4     static void Main(string[] args)
5     {
6         MessageHandler handler = delegate
7         {
8             Console.WriteLine("анонимный метод");
9         };
10        handler("hello world!");    // анонимный метод
11
12        Console.Read();
13    }
14}
```

Якщо анонімний метод використовує параметри, то вони повинні відповідати параметрам делегата. Якщо для анонімного методу не потрібно параметрів, то дужки з параметрами опускаються. При цьому навіть якщо делегат бере кілька параметрів, то в анонімному методі можна зовсім опустити параметри:

```
1 class Program
2 {
3     delegate void MessageHandler(string message);
4     static void Main(string[] args)
5     {
6         MessageHandler handler = delegate
7         {
8             Console.WriteLine("анонимный метод");
9         };
10        handler("hello world!");    // анонимный метод
11
12        Console.Read();
13    }
14}
```

Тобто якщо анонімний метод містить параметри, вони обов'язково повинні відповідати параметрам делегата. Або анонімний метод взагалі може не містити ніяких параметрів, тоді він відповідає будь-якому делегату, який має той же тип значення.

При цьому параметри анонімного методу не можуть бути опущені, якщо один або декілька параметрів визначені з модифікатором **out** .

Також, як і звичайні методи, анонімні можуть повертати результат:

```
1 delegate int Operation(int x, int y);
2 static void Main(string[] args)
3 {
4     Operation operation = delegate (int x, int y)
5     {
6         return x + y;
7     };
8     int d = operation(4, 5);
9     Console.WriteLine(d);       // 9
```

```
10    Console.Read();
11}
```

При цьому анонімний метод має доступ до всіх змінним, певним в зовнішньому коді:

```
1 delegate int Operation(int x, int y);
2 static void Main(string[] args)
3 {
4     int z = 8;
5     Operation operation = delegate (int x, int y)
6     {
7         return x + y + z;
8     };
9     int d = operation(4, 5);
10    Console.WriteLine(d);       // 17
11    Console.Read();
12}
```

В яких ситуаціях використовуються анонімні методи? Коли нам треба визначити одноразове дію, яка не має багато інструкцій і ніде більше не використовується. Зокрема, їх можна використовувати для обробки подій, які будуть розглянуті далі.

Лямбды

Лямбда-вирази подають спрощену запис анонімних методів. Лямбда-вирази дозволяють створити ємкі лаконічні методи, які можуть повертати деяке значення і які можна передати в якості параметрів в інші методи.

Ламбда-вирази мають наступний синтаксис: зліва від лямбда-оператора `=>`визначається список параметрів, а праворуч блок виразів, що використовує ці параметри: `(список_параметров) => выражение`. наприклад:

```
1 class Program
2 {
3     delegate int Operation(int x, int y);
4     static void Main(string[] args)
5     {
6         Operation operation = (x, y) => x + y;
7         Console.WriteLine(operation(10, 20));       // 30
8         Console.WriteLine(operation(40, 20));       // 60
9         Console.Read();
10    }
11}
```

Здесь код `(x, y) => x + y;` представляет лямбда-выражение, где x и y - это параметры, а `x + y` - Тут код `(x, y) => x + y;`представляє лямбда-вираз, де x і y - це параметри, а `x + y`- вираз. При цьому нам не треба вказувати тип параметрів, а при поверненні результату не треба використовувати оператор return.

При цьому треба враховувати, що кожен параметр в лямбда-виразі неявно перетворюється у відповідний параметр делегата, тому типи параметрів повинні бути однаковими. Крім того, кількість параметрів повинно бути таким же, як і у делегата. І яке значення лямбда-виразів має бути тим же, що і у делегата. Тобто в даному випадку використане лямбда-вираз відповідає делегату Operation як за типом значення, що повертається, так і за типом і кількістю параметрів.

Якщо лямбда-вираз приймає один параметр, то дужки навколо параметра можна опустити:

```
1
2 class Program
3 {
```

```
4     delegate int Square(int x); // объявляем делегат, принимающий int и возвращающий
5 int
6     static void Main(string[] args)
7     {
8         Square square = i => i * i; // объекту делегата присваивается лямбда-
9 выражение
10
11        int z = square(6); // используем делегат
12        Console.WriteLine(z); // выводит число 36
          Console.Read();
      }
  }
```

В даному випадку лямда-вираз нічого не повертає, так як після лямбда-оператора йде дія, яке нічого не повертає.

Як видно, з прикладів вище, нам необов'язково вказувати тип параметрів у лямбда-вирази. Однак, нам обов'язково потрібно вказувати тип, якщо делегат, якому має відповідати лямбда-вираз, має параметри з модифікаторами **ref** і **out** :

```
1 class Program
2 {
3     delegate void Hello(); // делегат без параметров
4     static void Main(string[] args)
5     {
6         Hello hello1 = () => Console.WriteLine("Hello");
7         Hello hello2 = () => Console.WriteLine("Welcome");
8         hello1();       // Hello
9         hello2();       // Welcome
10        Console.Read();
11    }
12}
```

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

Как видно, из примеров выше, нам не обязательно 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:

```
1 class Program
2 {
3     delegate void ChangeHandler(ref int x);
4     static void Main(string[] args)
5     {
6         int x = 9;
7         ChangeHandler ch = (ref int n) => n = n * 2;
8         ch(ref x);
9         Console.WriteLine(x);   // 18
10        Console.Read();
11    }
12}
```

Лямбда-выражения также могут выполнять другие методы:

```
1 class Program
2 {
3     delegate void Hello(); // делегат без параметров
4     static void Main(string[] args)
5     {
6         Hello message = () => Show_Message();
7         message();
8     }
```

```
9      private static void Show_Message()
10     {
11         Console.WriteLine("Привет мир!");
12     }
13 }
```

Лямбда-вирази як аргументи методів

Як і делегати, лямбда-вирази можна передавати в якості аргументів методу для тих параметрів, які представляють делегат, що досить зручно:

```
  class Program
  {
1     delegate bool IsEqual(int x);
2
3     static void Main(string[] args)
4     {
5         int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
6
7         // найдем сумму чисел больше 5
8         int result1 = Sum(integers, x => x > 5);
9         Console.WriteLine(result1); // 30
10
11        // найдем сумму четных чисел
12        int result2 = Sum(integers, x => x % 2 == 0);
13        Console.WriteLine(result2);  //20
14
15        Console.Read();
16    }
17
18    private static int Sum (int[] numbers, IsEqual func)
19    {
20        int result = 0;
21        foreach(int i in numbers)
22        {
23            if (func(i))
24                result += i;
25        }
26        return result;
      }
  }
```

Метод Sum приймає як параметр масив чисел і делегат IsEqual і повертає суму чисел масиву у вигляді об'єкта int. У циклі проходимо по всіх чисел і складаємо їх. Причому складаємо тільки ті числа, для яких делегат `IsEqual func` повертає true. Тобто делегат IsEqual тут фактично задає умову, якій повинні відповідати значення масиву. Але на момент написання методу Sum нам невідомо, що це за умову.

При виклику методу Sum йому передається масив і лямбда-вираз:

```
1int result1 = Sum(integers, x => x > 5);
```

Тобто параметр x тут буде представляти число, яке передається в делегат:

```
1if (func(i))
```

А вираз `x > 5` являє умова, якому має відповідати число. Якщо число відповідає цій умові, то лямбда-вираз повертає true, а передане число складається з іншими числами.

Подібним чином працює другий виклик методу Sum, тільки тут уже йде перевірка числа на парність, тобто якщо залишок від ділення на 2 дорівнює нулю:

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