Лекція 5

# РОЗШИРЕНІ МОЖЛИВОСТІ МОВИ ПРОГРАМУВАННЯ C#

# Лекція 5. Розширені можливості мови програмування C#

## План

1. Колекції, узагальнення та інтерфейси C#.
2. Делегати. Події. Лямбда-вирази.
3. LINQ.

# 1. Колекції, узагальнення та інтерфейси в C#

Для зберігання невеликих наборів однотипних даних у C# використовуються **масиви**. Однак, оскільки вони зберігають фіксовану кількість елементів, то на практиці їх застосування не завжди є зручним.

Для зберігання заздалегідь невизначеної кількості не обов'язково однорідних об'єктів зручніше застосовувати так звані **колекції**, що реалізують стандартні структури даних, такі як **стеки**, **черги**, **списки**, **словники** і т.п., які можуть стати в нагоді для вирішення різних спеціальних завдань.

Більшість класів колекцій міститься у просторах імен:

- **System.Collections** – прості неузагальнені класи колекцій;
- **System.Collections.Generic** – узагальнені чи типізовані класи колекцій;
- **System.Collections.Specialized** – спеціальні класи колекцій;
- **System.Collections.Concurrent** – класи колекцій для паралельного виконання завдань та багатопотокового доступу.

# 1. Колекції, узагальнення та інтерфейси в C#

Колекції у C# можна розділити на дві групи: **неузагальнені** та **узагальнені** .

**Неузагальнені колекції** працюють лише з об'єктами певних типів даних (зазвичай це System.Object). Наприклад:

```
// Динамічний масив елементів типу object
ArrayList objList = new ArrayList() { 1, 2, "Hello world!", 'c', 3.0f};
```

**Узагальнені колекції** є аналогами **шаблонних** класів мови C++. Вони дозволяють уніфіковано працювати з об'єктами різних типів даних. Наприклад:

```
// Список цілих елементів
List<int> numbers = new List<int>() { 1, 2, 3, 4, 5 };
```

# 1. Колекції, узагальнення та інтерфейси в C#

Для роботи з колекціями застосовуються спеціальні **інтерфейси IEnumerator** та **IEnumerable**, **IEnumerator<T>** та **IEnumerable<T>** (останні два – у разі узагальнених класів).

Інтерфейс **IEnumerator** представляє ітератор для послідовного перебору колекції, наприклад, у циклі foreach.

Інтерфейс **IEnumerable** за допомогою свого методу **GetEnumerator** надає ітератор для всіх класів, що реалізують цей інтерфейс. Він є базовим для всіх колекцій.

Аналогічні функції надають узагальнені інтерфейси **IEnumerator<T>** та **IEnumerable<T>** .

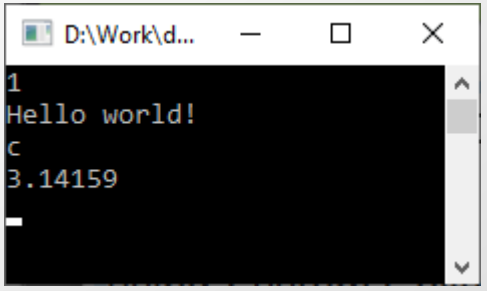
# 1. Колекції, узагальнення та інтерфейси в C#

Розглянемо приклад роботи з неузагальненим динамічним масивом.

```
using System;
using System.Collections;

namespace CollectTest
{
   class Program
   {
      static void Main(string[] args)
      {
         // Неузагальнена колекція ArrayList
         ArrayList objList = new ArrayList() { 1, 2, "Hello world!", 'c', 3.0f};
         objList.Add("3.14159");
         objList.Remove(2);
         objList.RemoveAt(3);

         foreach (object i in objList)
            Console.WriteLine(i);
         Console.ReadLine();
      }
   }
}
```



# 1. Колекції, узагальнення та інтерфейси в C#

Найбільш поширені неузагальнені типи з **System.Collections** .

| Клас | Призначення |
|---|---|
| ArrayList | Колекція розміру, що динамічно змінюється, містить об'єкти в певному порядку |
| BitArray | Компактний масив бітових значень |
| Hashtable | Колекція пар <ключ, значення>, організованих на основі хеш-коду ключа |
| Queue | Стандартна черга об'єктів, що працює за принципом FIFO |
| SortedList | Колекція пар <ключ, значення>, відсортованих за ключом та доступних за ключом та за індексом |
| Stack | Стек LIFO, що підтримує функціональність push та pop, а також зчитування |

# 1. Колекції, узагальнення та інтерфейси в C#

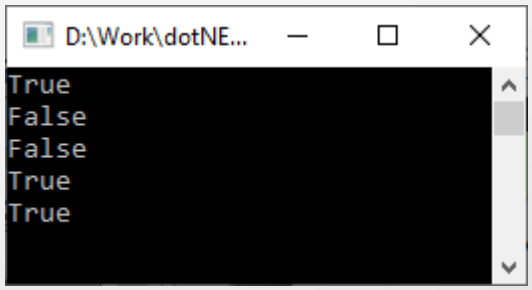
Розглянемо приклади роботи з класами із System.Collections.

**BitArray**

```
using System;
using System.Collections;

namespace BitArrayTest
{
   class Program
   {
      static void Main(string[] args)
      {
         BitArray ba = New BitArray(5, false);

         ba.Set(1, true);
         ba.Set(2, true);
         ba = ba.Not();
         foreach (bool it in ba)
            Console.WriteLine(it);
         Console.ReadLine();
      }
   }
}
```



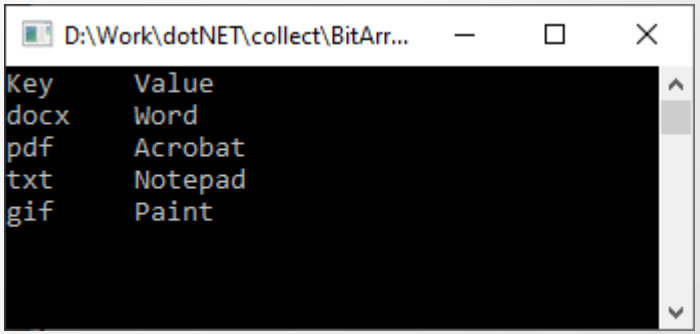
# 1. Колекції, узагальнення та інтерфейси в C#

**Hashtable**

```
using System;
using System.Collections;
class Programm
{
   static void Main()
   {
      Hashtable ht = New Hashtable();

      ht.Add("txt", "Notepad");
      ht.Add("docx", "Word");
      ht.Add("pdf", "Acrobat");
      ht.Add("gif", "Paint");

      Console.WriteLine("Key \tValue");
      foreach (DictionaryEntry it in ht)
         Console.WriteLine("{0}\t{1}", it.Key, it.Value);
      Console.ReadLine();
   }
}
```



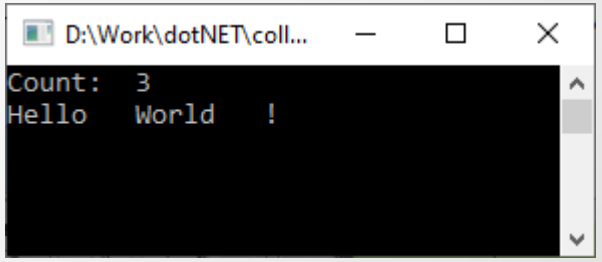
# 1. Колекції, узагальнення та інтерфейси в C#

**Queue**

```
using System;
using System.Collections;
public class Programm
{
   public static void Main()
   {
      Queue q = New Queue ();

      q.Enqueue("Hello");
      q.Enqueue("World");
      q.Enqueue("!");

      Console.WriteLine("Count:\t{0}", q.Count);
      foreach (object it in q)
         Console.Write("{0}\t", it);
      Console.WriteLine();
      Console.ReadLine();
   }
}
```



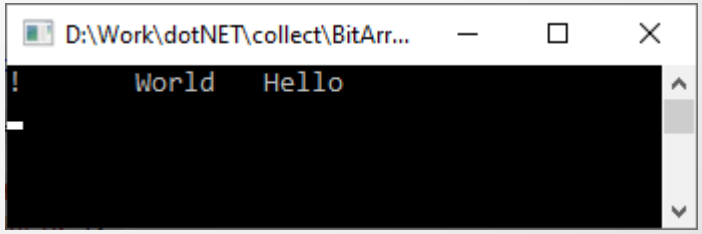
# 1. Колекції, узагальнення та інтерфейси в C#

**Stack**

```
using System;
using System.Collections;
public class Programm
{
   public static void Main()
   {
      Stack q = New Stack ();

      q.Push("Hello");
      q.Push("World");
      q.Push("!");

      foreach (object it in q)
         Console.Write("{0}\t", it);
      Console.WriteLine();
      Console.ReadLine();
   }
}
```



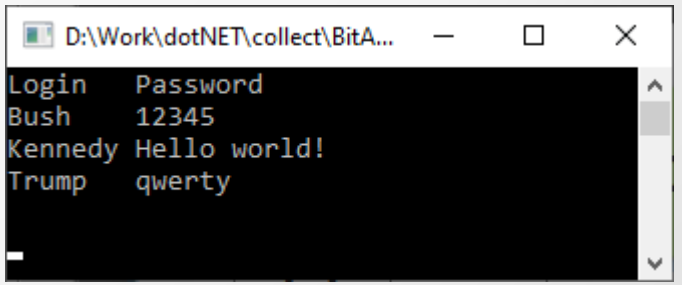
# 1. Колекції, узагальнення та інтерфейси в C#

**SortedList**

```
using System;
using System.Collections;
public class SamplesSortedList
{
   public static void Main()
   {
      SortedList sl = New SortedList();

      sl.Add("Trump", "qwerty");
      sl.Add("Bush", "12345");
      sl.Add("Kennedy", "Hello world!");

      Console.WriteLine("Login\tPassword");
         foreach (DictionaryEntry it in sl)
      Console.WriteLine("{0}\t{1}", it.Key, it.Value);
      Console.WriteLine();
      Console.ReadLine();
   }
}
```



# 1. Колекції, узагальнення та інтерфейси в C#

На практиці кращим є застосування **узагальнених колекцій**, описаних у просторі імен **System.Collections.Generic**. Основні з них наведені у наступній таблиці:

| Узагальнений клас | Призначення |
|---|---|
| Dictionary<TKey, TValue> | Узагальнена колекція пар <ключ, значення> |
| LinkedList<T> | Двозв'язковий список |
| List<T> | Динамічний список |
| Queue<T> | Узагальнена черга |
| SortedDictionary<TKey, TValue> | Узагальнений словник відсортованої множини пар <ключ, значення> |
| SortedSet<T> | Колекція унікальних відсортованих об'єктів |
| Stack<T> | Узагальнений стек |

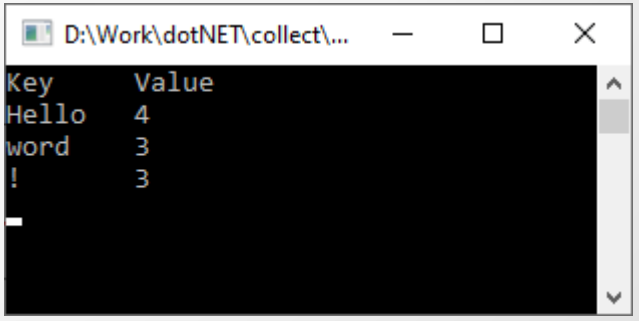
# 1. Колекції, узагальнення та інтерфейси в C#

Розглянемо деякі приклади роботи із узагальненими класами.

**Dictionary <TKey, TValue>**

```
using System;
using System.Collections.Generic;
public class Programm
{
   public static void Main()
   {
      Dictionary<string, int> dic = new Dictionary<string, int>();

      dic.Add("Hello", 1);
      dic.Add("word", 2);
      dic.Add("!", 3);
      dic["Hello"] = 4;
      dic["word"]++;
      Console.WriteLine("Key\tValue");
      foreach (KeyValuePair<string, int> it in dic)
         Console.WriteLine("{0}\t{1}", it.Key, it.Value);
      Console.ReadLine();
   }
}
```

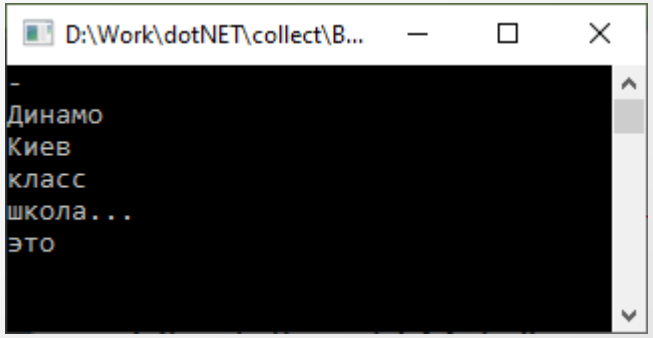


# 1. Колекції, узагальнення та інтерфейси в C#

**SortedSet<T>**

```
using System;
using System.Collections.Generic;
public class Programm
{
   public static void Main()
   {
      SortedSet<string> set = new SortedSet<string>();

      set.Add("Динамо");
      set.Add("Київ");
      set.Add("-");
      set.Add("це");
      set.Add("клас");
      set.Add("Динамо");
      set.Add("Київ");
      set.Add("-");
      set.Add("це");
      set.Add("школа...");
      foreach (string it in set)
         Console.WriteLine(it);
      Console.ReadLine();
   }
}
```



# 2. Делегати. Події. Лямбда-вирази

На практиці часто застосовуються так звані **зворотні виклики** (**callback**), коли у функцію в якості параметра передається покажчик на деяку іншу функцію, яка потім викликається. Наприклад, при реалізації універсального алгоритму сортування у відповідну функцію як параметр передається адреса іншої функції, в якій задається правило порівняння об'єктів, що сортуються.

Наприклад, у стандартній бібліотеці мови C реалізована функція швидкого сортування **qsort()**, шаблон якої має такий вигляд:

```
void qsort (void* base, size_t num, size_t size, int (*comparator)(const void*, const void*));
```

Тут четвертий параметр **comparator** задає покажчик на функцію, що реалізує операцію порівняння об'єктів, які сортуються. Параметрами цієї функції є безтипові покажчики на об'єкти, що порівнюються, а як результат повертається цілочисельна ознака їх рівності.

Використання зворотних викликів є поширеною практикою, хоча воно має й недоліки, наприклад, неможливість відстежити типи аргументів, що може призвести до помилок, які важко відслідкувати.

У .NET Framework для безпечної організації зворотних викликів використовується механізм **делегатів**.

# 2. Делегати. Події. Лямбда-вирази

**Делегат** – це типобезпечний об'єкт, який вказує на деякий метод програми, що може бути викликаний пізніше. Делегати успадковуються від класу **System.MulticastDelegate**.

Делегати містять таку інформацію про метод, на який вони вказують:

- **адрес**;
- **аргументи**;
- **значення**, що повертається.

Визначення делегата у C# схоже на опис сигнатури (шаблону) функції в мовах C/C++. Наприклад, такий опис делегату:

```
public delegate double BinaryOp(double lhs, double rhs);
```

може вказувати на будь-який метод, що приймає два дійсні параметри і повертає як результат дійсне число.

Після визначення делегата його потрібно створити. Це, наприклад, можна зробити так:

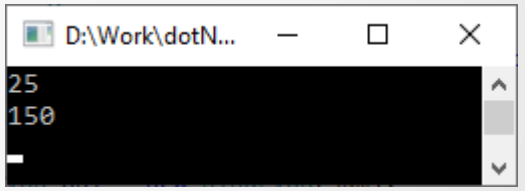
```
BinaryOp op = new BinaryOp(MyMethod);
```

Тут MyMethod – ім'я методу, на яке буде вказувати делегат.

# 2. Делегати. Події. Лямбда-вирази

Наприклад:

```
using System;
public class Programm
{
   public delegate double BinaryOp(double lhs, double rhs);
   public static void Main()
   {
      BinaryOp op1 = new BinaryOp(Sum), op2 = new BinaryOp(Mul);
      Process(op1, 10, 15);
      Process(op2, 10, 15);
      Console.ReadLine();
   }
   public static double Sum(double lhs, double rhs)
   {
      return lhs + rhs;
   }
   public static double Mul(double lhs, double rhs)
   {
      return lhs * rhs;
   }
   public static void Process(BinaryOp op, double lhs, double rhs)
   {
      Console.WriteLine(op(lhs, rhs));
   }
}
```

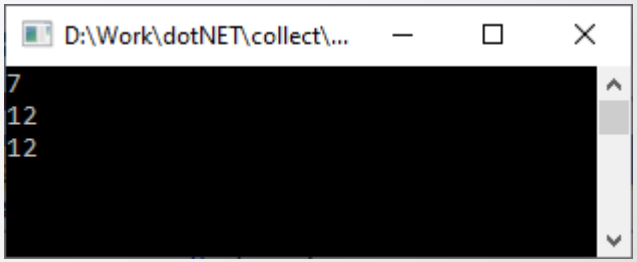


# 2. Делегати. Події Лямбда-вирази

Делегати підтримують можливість групового виклику. Наприклад:

```
using System;

public class Programm
{
   public delegate void PrintBinaryOp(double lhs, double rhs);
   public static void Main()
   {
      PrintBinaryOp op = new PrintBinaryOp(PrintSum) + new PrintBinaryOp(PrintMul);
      op += new PrintBinaryOp(PrintMul);
      op(3, 4);
      Console.ReadLine();
   }
   public static void PrintSum(double lhs, double rhs)
   {
      Console.WriteLine(lhs + rhs);
   }
   public static void PrintMul(double lhs, double rhs)
   {
      Console.WriteLine(lhs * rhs);
   }
}
```




**Примітка. Для групових операцій над методами в делегатах перевизначено операції "+", "-", "+=", "-=" .**

# 2. Делегати. Події. Лямбда-вирази

Крім делегатів у C# широко використовуються події (events), що сигналізують про те, що відбулася деяка дія, яка вимагає відповідної обробки. Події оголошуються за допомогою ключового слова **event**, після якого вказується тип делегата, який представляє подію. Наприклад:

```
delegate void EventHandler(string message); // Оголошення делегата EventHandler
event EvantHandler Notify; // Оголошення події Notify для делегата EventHandler
```

Розглянемо наступний приклад. Потрібно реалізувати програму, яка керує рахунком клієнта так, щоб він отримував повідомлення про рух коштів на його рахунку. При цьому безпосередню реалізацію доставки клієнту повідомлення (e-mail, sms чи viber) бажано відокремити від управління рахунком.

# 2. Делегати. Події. Лямбда-вирази

Проста реалізація такого класу може мати такий вигляд:

```
public class Account
{
   private int Total; // Поточна сума на рахунку
   public delegate void AccountMessengerHandler(string msg); // Делегат повідомлення клієнта
   public event AccountMessengerHandler Notify; // Відповідна подія

   public Account(int sum)
   {
      Total = sum;
   }
   public void AddAccount(int sum) // Надходження на рахунок
   {
      Total + = sum;
      // if (Notify! = null)
      // Notify ($ "На рахунок надійшло: {sum}. Залишок на рахунку: {Total}");
      Notify?.Invoke($"На рахунок надійшло: {sum}\nЗалишок на рахунку: {Total}\n");
   }
   public void SubAccount(int sum) // Зняття з рахунку
   {
      Total -= sum;
      Notify?.Invoke($"З рахунку знято: {sum}\nЗалишок на рахунку: {Total}\n");
   }
}
```

# 2. Делегати. Події. Лямбда-вирази

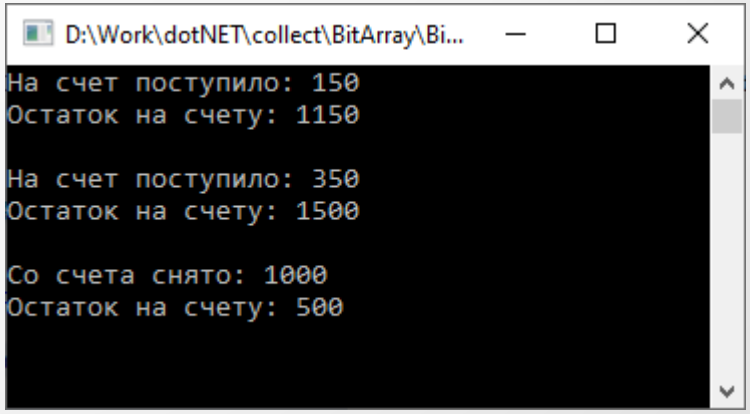
Тоді використання цього класу може виглядати так:

```
public class Programm
{
  public static void Main()
  {
    Account account = new Account(1000);

    account.Notify += DisplayNotify; // Додавання оброблювача для події Notify

    account.AddAccount(150);
    account.AddAccount(350);
    account.SubAccount(1000);

    Console.ReadLine();
  }
  // Обробник події-повідомлення
  private static void DisplayNotify(string message)
  {
    Console.WriteLine(message);
  }
}
```



# 2. Делегати. Події. Лямбда-вирази

Спільно з делегатами та подіями використовуються і так звані **лямбда-функції** – спеціальні безіменні об'єкти (замикання), які можна викликати як функції. Лямбда починається зі списку параметрів, за яким слідує лексема =>, а за нею оператори, що реалізують програмну логіку лямбда-функції.

У загальному вигляді лямбда-функцію можна описати таким чином:

Список_Аргументів => Оператори

Наприклад, за допомогою лямбда-виразу попередній приклад можна було б переписати так:

```
public class Programm
{
    public static void Main()
    {
        Account account = new Account(1000);

        account.Notify += (string message) => Console.WriteLine(message) ;
        account.AddAccount(150);
        account.AddAccount(350);
        account.SubAccount(1000);
        Console.ReadLine();
    }
}
```

# 3. LINQ

**LINQ** (Language-Integrated Query) – це проста мова запитів до різних джерел даних, що підтримують інтерфейс IEnumerable. Наприклад, до **масивів** і стандартних **колекцій**, наборів даних **DataSet**, документів **XML** і т.п.

Існує кілька різновидів LINQ:

- **LINQ to Objects** – для роботи з масивами та колекціями;
- **LINQ to Entities** – звернення до баз даних через технологію Entity Framework;
- **LINQ to Sql** – для доступу до даних у MS SQL Server;
- **LINQ to XML** – під час роботи з файлами XML
- **LINQ to DataSet** – взаємодії з об'єктом DataSet;
- **Parallel LINQ** – під час реалізації паралельних запитів.

# 3. LINQ

Наприклад:

```
using System;
using System.Linq;


public class Programm
{
   public static void Main()
   {
      string[] students = { "Макаренко Д.Б.", "Козюра А.П.", "Зиков О.Б.", "Крутяк А.Є.", "Довгий С.А.",
                            "Васильєв С.В.", "Карпачова Т.Б." };

      var selStud = from s in students // Задаємо кожен об'єкт із students як s
                    where s.ToUpper().StartsWith("К") // Критерій відбору
                    orderby s // Упорядкування за зростанням
                    select s; // Вибір відповідного об'єкта

      foreach (string it in selStud)
         Console.WriteLine(it);
      Console.ReadKey();
   }
}
```