Заняття 1: Введення, Git, класи та структури

Git-flow – Організація процесу розділення версій на гілки (branch) та злиття їх (merge) назад у головну гілку (main/master). Перед процесом злиття в головну гілку роблять **pull request** – коли інші перевіряють код в гілці на помилки і дають дозволи на злиття з головною гілкою.

Клас – шаблон з даними (філдами, змінними та ін.) та функціями по якому створюються об'єкти. По замовчуванні клас має ідентифікатор дозволу **internal**. Клас є видимим в межах проекту. Ідентифікатори доступу допомагають реалізувати інкапсуляцію. Бажано прописувати ідентифікатори доступу явно (не зловживати ідентифікаторами по замовчуванню).

Якщо в класі немає інших конструкторів, то під час створення об’єкта буде викликатись **конструктор по замовчуванню**. Він зникає, якщо явно вказаний інший конструктор.

!!! Хороша практика роботи – один клас на один файл. Назва файлу та назва класу має бути однаковими.

Ключове слово **this** – представляє на поточний екземпляр об’єкта класу. Для прикладу його використовують коли назва параметру співпадає з назвою поля:

```
Public int age;
testFunc(int age)
{
this.age = age;
}
```

Ланцюжок виклику конструкторів – це коли один конструктор викликає інший конструктор в межах даного об’єкта. Виклик відбувається через ключове слово **this**.

```
public Person() : this(“Undefined”)
{
//additional logic
}
public Person(string name) : this(name, 18)
{
//additional logic
}
public Person(string name, int age)
{
this.name = name;
this.age = age;
}
```

Ініціалізатор об’єкта – альтернативний варіант створення та ініціалізації об’єкта:

```
var user = new Person {name = “John”, age = 55};
```

За допомогою ініціалізації об'єктів можна надавати значення всім доступним полям і властивостям об'єкта в момент створення.

При використанні ініціалізації слід враховувати наступні моменти:

За допомогою ініціалізатора ми можемо встановити значення лише доступних із позакласу полів та властивостей об'єкта.

Наприклад, у прикладі вище поля name і age мають модифікатор доступу public, тому вони доступні будь-якої частини програми.

Ініціалізатор виконується після конструктора, тому якщо і в конструкторі, і в ініціалізаторі встановлюються значення тих самих полів і властивостей, то значення, що встановлюються в конструкторі, замінюються значеннями з ініціалізатора.

Типи даних:

- Цілочисленні типи (byte, sbyte, short, ushort, int, uint, long, ulong)
- Типи з плаваючою комою (float, double)
- Тип decimal
- Тип bool
- Тип char
- Перерахування enum
- Структури (struct)
- Посилальні типи: (ссылочные типы данных)
- Тип object
- Тип string
- Класи (class)
- Інтерфейси (interface)
- Делегати (delegate)

Кортеж – колекція, яка представляє константні значення. Має вигляд (string name, int age).

Деконструктор дозволяє виконати декомпозицію об'єкта на окремі частини.

```
var person = new PersonV4("Tom", 33);
(string name, int age) = person;
```

Для реалізації **деконструктора** необхідно, щоб в класі був реалізований метод Deconstruct

```
public void Deconstruct(out string personName, out int personAge)
{
    personName = name;
    personAge = age;
}
```

Різниця між **типами-посиланнями** (об’єктами) та **типами-значннями** (об’єкти структур, базові типи):

- Спосіб збереження в мап’яті: типи-значень розміщуються в стеку повністю, типи-посилання – в стеку лише посилання на об’єкт, а поля у керованій купі;

- Відмінність в базовому типі: для значимих типів System.ValueType, а для посилальних типів може бути будь-який не sealed тип крім System.ValueType;
- Значимі типи не можуть бути успадкованими;
- При передачі у функцію значимі типи передаються за значенням, а посилальні – за посиланням (за значенням копіюється тільки посиання);
- Типи значень не потребують фіналізації (перевизначення методу System.Object.Finalize()), бо вони ніколи не розміщуються в керованій купі;
- Типи-значення припиняють існування, коли виходять за рамки контексту, в якому визначались, а типи-посилання знищуються збирачем сміття

Спільним для **типів-значень** та **типів-посилань** є те, що для тих і тих можна визначати конструктори.

Характеристика	**class**	**record (он же record class)**	**record struct**	**struct**
Ссылочный/значимый	ссылочный	ссылочный	значимый	значимый
Есть ли позиционная запись	нет	да	да	нет
Неизменяемость (иммутабельность) по умолчанию	нет	с позиционной записью — да	нет	нет
Наследование	да	только от record	нет	нет
Реализация интерфейсов	да	да	да	да
Автоматически генерируются операторы сравнения	сравнение по ссылке	сравнение по значению	сравнение по значению	нет
Создание через инициализатор объектов	да	с позиционной записью — нет	да	да
Автоматически генерируется конструктор без параметров	да	с позиционной записью — нет	да	да

Статичний клас – це клас, який оголошується з ключовим словом static. У порівнянні з нестатичним класом, статичний клас має наступні властивості (відмінності):

- неможна створювати об’єкти статичного класу;
- статичний клас повинен містити тільки статичні члени.

Упаковка/розпаковка – приведення типів-значень до типів-посилань та навпаки.

```
int number = 10;
object my_number = number; // упаковка
int number_from_object = (int)my_number; // распаковка
```

Можемо розпакувати лише у вихідний тип даних. В іншому разі буде помилка рантайму.

Посилання:

Завантажити та встановити:

https://git-scm.com/downloads

Windows: http://gitextensions.github.io/

Windows/Mac: https://www.sourcetreeapp.com/ чи https://desktop.github.com/ чи https://www.gitkraken.com/jc

Туторіали:

https://www.w3schools.com/git/

https://www.atlassian.com/git/tutorials

https://githowto.com/setup

https://www.tutorialspoint.com/git/index.htm

https://opensource.com/article/18/1/step-step-guide-git

Static classes:

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static

https://www.tutorialsteacher.com/csharp/csharp-static

https://henriquesd.medium.com/singleton-vs-static-class-e6b2b32ec331#:~:text=Differences%20between%20Singleton%20and%20Static,and%20therefore%20cannot%20be%20inherited.

https://dotnethow.net/static-vs-non-static-classes-in-c-understanding-the-differences-and-use-cases/

https://debug.to/3494/abstract-class-vs-partial-class-vs-static-class-vs-sealed-class

Корисні посилання:

https://www.geeksforgeeks.org/introduction-to-visual-studio/

https://www.edureka.co/blog/visual-studio-tutorial/

https://learn.microsoft.com/en-us/visualstudio/version-control/git-with-visual-studio?view=vs-2022

https://www.bestprog.net/uk/2018/12/05/static-classes-methods-variables-static-constructors-keyword-static_ua/

https://habr.com/ru/articles/675560/

https://www.bestprog.net/uk/2019/10/10/c-categories-groups-of-data-types-in-c-value-types-reference-types-basic-data-types-overview-ua/#q03

Заняття 2: ООП. Реалізація принципів ООП

Кортеж (tuple) – колекція різнотипних значень, але типи відомі наперед.

```
public static (int sum, double average) GetArrCalculations(int[] numbers)
    {
      var result = (sum: 0, average: .0);
      foreach (var number in numbers)
      {
        result.sum += number;
      }
      result.average = Convert.ToDouble(result.sum) / numbers.Length;
      return result;
    }

var tupleResult = GetArrCalculations(new int[] { 1, 2, 3, 4, 6 });
Console.WriteLine(tupleResult);
Console.WriteLine($"Sum: {tupleResult.sum} Avg: {tupleResult.average}");
```

Структура – значимий тип даних, значення яких записується в стеках. Зараз використовується рідко і у випадках коли є дуже простий набір значимих типів даних. Переважно зараз використовуються **рекорди** та **класи**. Для використання структури її необхідно ініціалізувати. Для ініціалізації створення об'єктів структури, як і у випадку класів, застосовується виклик конструктура з оператором new. Навіть якщо в коді структури не визначено жодного конструктора, проте є, як мінімум, один конструктор - конструктор за замовчуванням, який генерується компілятором (конструктор не приймає параметрів і створює об'єкт структури зі значеннями за замовчуванням). Можна викликати **конструктори ланцюжком**. Також зручно користуватись ключовим словом **with** для копіювання значень одної структури в іншу (юзається і в рекордах).

```
struct Person
{
  public string name = "no name";
  public int age = 18;
  public Person() { }
  public Person(string name = "Tom", int age = 1)
  {
    this.name = name;
    this.age = age;
  }
  public void Print()
  {
    Console.WriteLine($"Name: {name}  Age: {age}");
  }
}
```

Копіювання по посиланню та по значенні – по посиланні ми працюємо з адресом пам’яті і зміни змінної через будь яке скопійоване посилання впливають на вихідне значення; по значенні ми працюємо з копією значення і зміни в будь якій скопійованій змінній не впливають на вихідне значення.

readonly vs **const**: readonly можна змінити в конструкторі, а константу не можна змінити після оголошення (оголошення та ініціалізація відбуваються одночасно). **readonly**

юзають найчастіше при **dependency injection** - коли є змінна (поле) для зберігання, для прикладу, мікросервісу. Тоді ми лише раз, через конструктор, ініціалізуємо цю змінну відповідним мікросервісом.

Статичний клас – це клас, який оголошується з ключовим словом static. У порівнянні з нестатичним класом, статичний клас має наступні властивості (відмінності):

- неможна створювати об’єкти статичного класу;
- статичний клас повинен містити тільки статичні члени.

Статичні класи, методи та змінні ефективні у наступних випадках:

- коли потрібно створити так звані методи розширення. Методи розширення використовуються для розширення функцій класу. Ці методи є статичними;
- коли у програмі є деякий спільний ресурс, до якого можуть мати звертання методи різних класів які обробляють даний ресурс (читають або змінюють його значення). Цей спільний ресурс оголошується як статична змінна. Наприклад, таким ресурсом може бути деякий лічильник викликів, метод що реалізує унікальну обробку, унікальна файлова змінна (ресурс) тощо;
- коли потрібно об’єднати між собою групи статичних методів;
- коли потрібно використовувати спільні приховані (private) дані класу та організовувати доступ до цих даних зі статичних та нестатичних методів.

Властивості статичних полів:

- Статичні поля зберігають стан всього класу/структури.
- Статичне поле визначається як і просте, лише перед типом поля вказується ключове слово static.
- На рівні пам'яті для статичних полів створюватиметься ділянка в пам'яті, яка буде спільною для всіх об'єктів класу.

Крім звичайних конструкторів, у класу також можуть бути **статичні конструктори**. **Статичні конструктори** мають такі відмінні риси:

- Статичні конструктори не повинні мати модифікатор доступу та не приймають параметрів
- Як і в статичних методах, в статичних конструкторах не можна використовувати ключове слово для посилання на поточний об'єкт класу і можна звертатися тільки до статичних членів класу
- Статичні конструктори не можна викликати у програмі вручну. Вони виконуються автоматично при першому створенні об'єкта даного класу або при першому зверненні до його статичних членів (якщо такі є)

Статичні конструктори зазвичай використовуються для ініціалізації статичних даних або виконують дії, які потрібно виконати лише один раз.

Функціональне програмування базується на чисто-функції. Коли функція робить одну конкретну дію і не залежить від оточення чи глобальних змінних.

Агрегація менше жорсткий зв'язок між вкладеним об'єктом та об'єктом в який вкладено. Без вкладеного зовнішній може функціонувати.

Композиція – жорсткий зв'язок між вкладеним об'єктом та об'єктом в який вкладено. Без вкладеного об'єкта зовнішній не може функціонувати.

Процедурне програмування базується на процедурах. Декомпозиція на менші функціональні частини, які можна використовувати окремо в різних частинах програми.

ООП – парадигма програмування, який базується на інкапсуляції, наслідуванні та поліморфізмі.

Інкапсуляція – приховування внутрішньої реалізації і надання санкціонованого доступу до інтерфейсу класу. Клас стає, як чорний ящик. Абстрагує код клієнта від внутрішньої реалізації класу.

Наслідування – створення нового класу на основі вже створеного. За допомогою наслідування ми можемо розширяти функціонал базового класу в дочірньому класі без копі/пасту. Абстрагуємось від базового класу, використовуючи дочірній. В C# відсутнє множинне наслідування для класів, проте дозволено реалізовувати безліч **інтерфейсів**.

Інтерфейс – контракт, тобто опис полів та методів, які клас зобов'язується виконати у разі імплементації інтерфейсу.

Поліморфізм – один інтерфейс і багато реалізацій. Абстрагуємось від конкретної реалізації.

Модифікатори доступу допомагають реалізувати інкапсуляцію. По замовчуванню для класу – internal, а для полів та функцій – private. ***!!!Хороша практика*** – завжди явно прописувати модифікатори доступу. Є наступні типи модифікаторів доступу:

- private – поле/функція доступні лише всередині поточного класу;
- protected private – поле/функція доступні в поточному класі або класах, які були наслідувані від поточного в даному проекті;
- protected – поле/функція доступні в поточному класі або класах, які були наслідувані від поточного;
- internal – поле/функція доступні в будь якому місці поточного проекту;
- public – поле/функція доступні в будь якому місці поточного проекту, а також для інших програм та збірок;

Для спрощення доступу до полів класу використовують **властивості** та **автовластивості**. Це автоматичні замінники гетерів та сетерів. **Автовластивості** використовують у випадках, коли гетери та сетери не мають додаткової логіки.

Властивості:

```
private string _hobby;
  public string Hobby
    {
       get
       {
```

```
            return _hobby;
        }
        set
        {
            _hobby = value;
        }
    }
```

Автовластивості:

```
private string _firstName;
public string Name { get; set; } = "noname"; //Значення по замовчуванні
```

readonly властивість (можна тільки прочитати):

```
private string _companyName = "Google";
public string CompanyName
    {
        // v1
        // get { return _companyName; }
        // v2
        get;
    }
```

writeonly властивість (можна тільки записати):

```
private string _firstName;
public string FirstName
    {
        set { _firstName = value; }
    }
```

!!!Хороший стиль програмування: назву поля починати з нижнього підкреслення (_xxx), а назви властивостей з великої літери Xxxx

virtual – ключове слово, яке означає, що метод ми можемо явно переозначити в наслідникy.

```
public virtual void Print()
        {
            Console.WriteLine($"Name: {Name}\nCost: {Cost}");
        }
```

override – явне переозначення методу з базового класу (найкраще робити так).

```
public override void Print()
        {
            base.Print(); // вызываем реализацию из базового класса
            Console.WriteLine($"Pixels: {Pixels}");
        }
```

При використанні **override** переписується функціональність базового класу з дочірнього. При цьому, якщо зробити приведення до базового типу, все рівно буде викликатись переозначений метод (при інших переозначеннях такого не відбуватиметься).

Якщо зробити public **override sealed** void Print() – означає, що заборонено подальше переозначення методу.

Стилі переозначення, які працюватимуть, але є поганими з точки зору втрати методу базового класу:

Неявне приховування методу – втрачаємо реалізацію методу з такою назвою з базового класу

```
public void Print()
    {
      Console.WriteLine($"Pixels: {Pixels}");
    }
```

Явне приховування методу - втрачаємо реалізацію методу з такою назвою з базового класу

```
public new void Print()
    {
      Console.WriteLine($"Pixels: {Pixels}");
    }
```

Посилання:
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

https://www.c-sharpcorner.com/UploadFile/mkagrahari/introduction-to-object-oriented-programming-concepts-in-C-Sharp/

https://www.tutorialsteacher.com/csharp/oop

https://medium.com/codex/c-object-oriented-programming-oop-2d92a5cd336f

https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

Заняття 3: Абстрактні класи та інтерфейси

Інтерфейс – **контракт**, тобто опис полів та методів, які клас зобов'язується виконати у разі імплементації інтерфейсу.

```
interface IMovable
   {
      // реализация метода по замовчуванню
      void Move() => Console.WriteLine("Walking");
      //Інші поля і методи
   }

public class Person : IMovable { }
```

З інтерфейсами можна зробити таку річ:

```
IMovable persone = new Persone();
IMovable animal = new Animal();
IMovable robot = new Robot();
```

Після «=» (new Xxxx) створюються конкретні класи, які реалізовують даний інтерфейс. Тому ми можемо метод з інтерфейсу викликати через змінну інтерфейсу, якій присвоєне посилання на конкретний об'єкт, із реалізацією в класі даного об'єкта. Проте, всі додаткові методи та поля, які створені в цьому класі, не будуть доступними (все обмежується даними та методами з інтерфейсу).

Інтерфейс представляє тип посилання, який може визначати деякий функціонал - набір методів і властивостей без реалізації. Потім цей функціонал реалізують класи та структури, які застосовують дані інтерфейси.

Для визначення інтерфейсу використається ключове слово interface. Як правило, назви інтерфейсів у C# починаються з великої літери I, наприклад, IComparable, IEnumerable (так звана угорська(венгерская) нотація), проте це не обов'язкова вимога, а більше стиль програмування.

Що може визначити інтерфейс? Загалом інтерфейси можуть визначати такі сутності:

- Методи
- Властивості
- Індексатори
- Події
- Статичні поля та константи (починаючи з версії C# 8.0)

Проте інтерфейси неспроможні визначати нестатичні змінні.

Ще один момент в оголошенні інтерфейсу: якщо його члени - методи та властивості не мають модифікаторів доступу, то фактично за умовчанням доступ public, оскільки мета інтерфейсу - визначення функціоналу для реалізації його класом. Це стосується також констант і статичних змінних, які в класах і структурах за замовчуванням мають модифікатор private.

В інтерфейсах вони мають за замовчуванням модифікатор public. Але, починаючи з версії C# 8.0, ми можемо явно вказувати модифікатори доступу у компонентів інтерфейсу.

Як і класи, стандартні інтерфейси мають рівень доступу internal, тобто такий інтерфейс доступний тільки в рамках поточного проекту. Але за допомогою модифікатора public ми можемо зробити інтерфейс загальнодоступним.

Також, починаючи з версії C# 8.0, інтерфейси підтримують реалізацію методів і властивостей за умовчанням. Це означає, що ми можемо визначити в інтерфейсах повноцінні методи та властивості, які мають реалізацію як у звичайних класах та структурах.

Інтерфейси мають ще одну важливу функцію: в C# не підтримується множинне успадкування, тобто ми можемо успадкувати клас тільки від одного класу, на відміну, скажімо, від мови C++, де успадкування можна використовувати. Інтерфейси дозволяють частково обійти це обмеження, оскільки C# класи і структури можуть реалізувати відразу кілька інтерфейсів. Всі реалізовані інтерфейси вказуються через кому.

```
interface IMovable
   {
      // реалізація методу по замовчуванню
      void Move() => Console.WriteLine("Walking");
      // реалізація властивості по замовчуванню
      // int MaxSpeed { get { return 0; } }

      // Якщо інтерфейс має приватні методи та властивості (тобто з модифікатором private), то вони повинні мати реалізацію за умовчанням. Те саме стосується статичних методів (не обов'язково приватних)
      public const int minSpeed = 0;     // мінімальна швидкість
      private static int maxSpeed = 60;   // максимальна швидкість
// час, за який проходимо distance зі швидкістю speed
      static double GetTime(double distance, double speed) => distance / speed;
      static int MaxSpeed
      {
         get => maxSpeed;
         set
         {
            if (value > 0) maxSpeed = value;
         }
      }
   }
```

!!!!Наслідування інтерфейсів!!!!

Всі об’єкти в C# є наслідниками базового класу **object**. Від нього передались наступні методи:

- ToString()
- GetHashCode()
- GetType()
- Equals();

ToString() - служить для отримання рядкового представлення даного об'єкта.

GetHashCode() - дозволяє повернути деяке числове значення, яке буде відповідати даному об'єкту або його хеш-коду. За цим числом, наприклад, можна порівнювати об'єкти. Можна визначати різні алгоритми генерації подібного числа або взяти реалізацію базового типу.

Метод **Equals()** приймає як параметр об'єкт будь-якого типу, який потім приводити до поточного класу. Якщо переданий об'єкт є типом поточного класу, то повертаємо результат порівняння вказаних полів двох об'єктів поточного класу. Якщо об'єкт представляє інший тип, то повертається false. При необхідності реалізацію методу можна зробити складнішою, наприклад, порівнювати за декількома властивостями за їх наявності. Слід зазначити, що з методом Equals слід реалізувати метод GetHashCode. Якщо порівнювати два складні об'єкти, то краще використовувати метод Equals, а не стандартну операцію ==

GetType() – повертає об'єкт Type, тобто тип об'єкта.

Глибоке порівняння – порівняння по значенні, а не по посиланню.

!!!Почитати що в середині словника!!!

Абстрактний клас – клас хоча б з одним абстрактним членом. Позначається (як і абстрактні члени) ключовим словом **abstract**. Особливістю є абстрактність даних сутностей. Тобто, за допомогою цього ми абстрагуємось від конкретної реалізації. Дану реалізацію ми покладаємо на дочірній клас. Якщо не реалізувати абстрактну сутність в дочірньому класі, то і цей дочірній клас стає абстрактним.

!!!Особливістю є те, що ми не можемо створити **екземпляр абстрактного класу**.

Абстрактними можуть бути такі члени класу:

- Методи
- Властивості
- Індексатори
- Події

Абстрактні члени класів не повинні мати **private** модифікатор.

В **дочірньому класі** реалізація повинна іти з **override** ключовим словом, бо інакше буде **відмова від реалізації** абстрактного члена і дочірній клас стане абстрактним.

Головні відмінності між **абстрактним класом і інтерфейсом** (***додатково уточнити***):

- АК має хоча б 1 абстрактний член;
- Не можна зробити успадкування від кількох АК, а І можна імплементувати скільки завгодно;
- Хоча І можуть мати реалізацію за замовчуванням, але клас, який імплементує І має переозначити функціонал з І, а з АК переозначення не абстрактного методу робити не обов’язково;
- І не може бути sealed або abstract але методи в І можуть юзати new для пере означення методів з однаковими сигнатурами;
- Клас успадковуємо, а інтерфейс реалізовуємо;

Основа для порівняння	**Інтерфейс**	**Анотація класу**
Основний	Коли ви тільки знаєте вимоги не про його реалізацію, ви використовуєте "Інтерфейс".	Коли ви частково знаєте про реалізації, ви використовуєте "Абстрактні класи".
Методи	Інтерфейс містить лише абстрактні методи.	Абстрактний клас містить абстрактні методи, а також конкретні методи.
Модифікатор методів доступу	Методи інтерфейсу завжди "Public" і "Abstract", навіть якщо ми не оголошуємо. Отже, це можна сказати як 100%, чистий абстрактний клас.	Не обов'язково, що метод в абстрактному класі буде публічним і абстрактним. Він також може мати конкретні методи.
Обмежений модифікатор методів	Метод інтерфейсу не може бути оголошений наступними модифікаторами: Загальнодоступний: приватний і захищений Анотація: фінальний, статичний, синхронізований, рідний, рок.	Немає обмежень на модифікатори змінних абстрактного класу.
Модифікатор доступу до змінних	Модифікатор Acess, дозволений для змінних інтерфейсу, є загальнодоступними, статичними та остаточними, чи ми декларуємо чи ні.	Змінні в абстрактному класі не повинні бути публічними, статичними, кінцевими.
Обмежені модифікатори для змінних	Інтерфейсні змінні не можуть бути оголошені як приватні, захищені, перехідні, мінливі.	Не існує обмежень на модифікатори абстрактних змінних класу.
Ініціалізація змінних	Змінні інтерфейси повинні бути ініціалізовані під час його декларування.	Не обов'язково, щоб абстрактні змінні класу повинні бути ініціалізовані під час його декларування.
Примірник і статичні блоки	Всередині інтерфейсу не можна оголосити примірник або статичний блок.	Абстрактний клас допускає примірник або статичний блок всередині нього.
Конструктори	Не можна оголосити конструктор всередині інтерфейсу.	Ви можете оголосити конструктор всередині абстрактного класу.

Різниця між **is** і **as:**

- **as** використовується для приведення одного типу до іншого. При цьому при **невдалому** приведенні **не відбудеться викидання ексепшину** (при звичайному касті відбудеться);

- **is** використовується для порівняння двох типів. Якщо один можна привести до іншого, то повертається true, в іншому випадку – false;

Посилання:
https://www.geeksforgeeks.org/is-vs-as-operator-keyword-in-c-sharp/

https://www.c-sharpcorner.com/blogs/is-and-as-keyword-difference-in-c-sharp

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

https://www.c-sharpcorner.com/UploadFile/mkagrahari/introduction-to-object-oriented-programming-concepts-in-C-Sharp/

https://www.tutorialsteacher.com/csharp/oop

https://medium.com/codex/c-object-oriented-programming-oop-2d92a5cd336f

https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

Заняття 4: Перевантаження операторів, приклади

Методи розширення(extension methods) дозволяють додавати нові методи вже існуючі типи без створення нового похідного класу. Ця функціональність буває особливо корисною, коли нам хочеться додати до певного типу новий метод, але сам тип (клас чи структуру) ми змінити не можемо, оскільки ми не маємо доступу до вихідного коду. Або якщо ми не можемо використати стандартний механізм успадкування, наприклад, якщо класи визначені з модифікатором sealed.

Для того, щоб створити метод розширення, спочатку треба створити статичний клас, який і міститиме цей метод. Потім оголошуємо статичний метод. Власне метод розширення - це звичайний статичний метод, який як перший параметр завжди приймає таку конструкцію: this имя_типа назва_параметра, (this string str).

```
public static class StringExtension
   {
      public static int CharCount(this string str, char c)
      {
         int counter = 0;
         for (int i = 0; i < str.Length; i++)
         {
            if (str[i] == c)
               counter++;
         }
         return counter;
      }
   }

string s = "Hello world";
char c = 'l';
int i = s.CharCount(c);
```

Перевантаження операторів призначене для додавання логіки роботи над кастомними об’єктами через оператори +, -, ++, -- і т.д.

можна перевантажувати:

- унарні оператори +x, -x, !x, ~x, ++, --, true, false
- бінарні оператори +, -, *, /, %
- оператори порівняння ==, !=, <, >, <=, >=
- порозрядні оператори &, |, ^, <<, >>
- логічні оператори &&, ||
- Необхідно перевантажувати парами:
- == та !=
- < та >
- <= та >=

```
class Counter
   {
      public int Value { get; set; }

      public static Counter operator +(Counter counter1, Counter counter2)
      {
         return new Counter { Value = counter1.Value + counter2.Value };
      }
```

```
    public static bool operator >(Counter counter1, Counter counter2)
    {
      return counter1.Value > counter2.Value;
    }
}
```

Явне та неявне перетворення типів: необхідне для перетворення двох несумісних типів один в інший.

explicit convert type – явне приведення типів. Для того, щоб у своєму класі організувати явне приведення до певного типу необхідно зробити наступне:

```
public static explicit operator int(CounterV2 counter)
    {
      return counter.Seconds;
    }
або
    public static explicit operator CounterV2(Timer timer)
    {
      int h = timer.Hours * 3600;
      int m = timer.Minutes * 60;
      return new CounterV2 { Seconds = h + m + timer.Seconds };
    }
```

implicit convert type – неявне перетворення типів:

```
public static implicit operator Timer(CounterV2 counter)
    {
      int h = counter.Seconds / 3600;
      int m = (counter.Seconds % 3600) / 60;
      int s = counter.Seconds % 60;
      return new Timer { Hours = h, Minutes = m, Seconds = s };
    }
```

Перевантаження індексаторів: реалізація логіки отримання елементів внутрішнього масиву об’єкта по індексу або по іншому полю (ім’я, адрес та ін..) без виклику внутрішнього масиву.

```
class Organisation
  {
    User[] users;
    public Organisation(User[] users)
    {
      this.users = users;
    }

    public User this[int index]
    {
      get => users[index];
      set => users[index] = value;
    }

    public User this[string userName]
    {
      get
```

```
        {
            foreach (var user in users)
            {
                if (user.UserName == userName)
                {
                    return user;
                }
            }
            throw new InvalidOperationException("Unknown name");
        }
    }
}

Console.WriteLine(myOrg[1]);
 Console.WriteLine(myOrg["Anton"]);
```

Посилання:

https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/operator-overloading

https://www.bestprog.net/uk/2022/09/07/c-overloading-of-operators-general-information-overloading-of-unary-operators-ua/

Заняття 5: Generic. Шаблонні класи та інтерфейси

Generic – узагальнення. Засіб мови C#, що дозволяє створювати програмний код, що містить єдине (типізоване) рішення задачі для різних типів, з його подальшим застосуванням для будь-якого конкретного типу (int, float, char тощо).

Переваги використання generic:

- спрощення програмного коду;
- забезпечення типової безпеки – тип з яким працюватиме сутність вказується при її оголошенні, тому подальше відслідковування коректності типів бере на себе компілятор;
- виключена необхідність явного приведення типу;
- підвищується продуктивність бо немає необхідності використовувати boxing і unboxing.

Як мовний засіб, узагальнення можуть бути застосовані до:

- класів;
- структур;
- інтерфейсів;
- методів;
- делегатів.

Синтаксис для класів:

```
class ClassName<T1, T2, ..., TN>
{
  // Реалізація методів та полів класу,
  // які працюють з узагальненими типами T1, T2, ..., TN
  // ...
}

ClassName<t1, t2, ..., tN> obj = new ClassName<t1, t2, ..., tN>(arguments);
```

Синтаксис для структур:

```
struct StructName<T1, T2, ..., TN>
{
  // ...
}

StructName<t1, t2, ..., tN> val = new StructName<t1, t2, ..., tN>(arguments);
```

В програмах на узагальнення накладаються наступні **обмеження**:

- властивості не можуть бути узагальненими, але можуть використовуватись в узагальненому класі (структурі);
- індексатори не можуть бути узагальненими, але можуть використовуватись в узагальненому класі (структурі);
- перевантажені оператори (operator) не можуть бути узагальненими. Однак використання параметру типу T тут допускається;
- події (event) не можуть бути узагальненими але можуть використовуватись в узагальнених класах (структурах);

- до узагальненого класу не можна застосовувати модифікатор extern;
- типи покажчиків не можна використовувати в аргументах типу;
- якщо в узагальненому класі (структурі) використовується статичне поле (static), то в об'єкті кожного конкретного типу (int, double тощо) створюється унікальна копія цього поля. Тобто, немає єдиного статичного поля для всіх об'єктів різних типів, що конструюються;
- до узагальненого типу T не можуть бути застосовані арифметичні операції (+, –, * та інші) а також операції порівняння. Це пов'язано з тим, що при створенні екземпляру з типом-заповнювачем (інстанціюванні) замість параметру типу може бути використаний тип даних, який не підтримує ці операції.

При оголошенні ми можемо зробити обмеження, на тип даних, який підставлятиметься замість узагальнення:

- Classes (об'єкт якогось конкретного класу)
- Interfaces (об'єкт якогось конкретного інтерфейсу)
- class - the generic parameter must represent a class
- struct - the generic parameter must represent a structure
- new () – the generic parameter must represent a type that has a public parameterless constructor. Дане обмеження має бути останнім у переліку, якщо обмежень кілька (інакше помилка при компіляції)

Приклади:

```
class Messenger<T> where T : struct
{ }
class Messenger<T> where T : class
{ }
class Messenger<T> where T : new()
{ }
class Smartphone<T> where T : Messenger, new()
{ }
```

При наслідуванні обмеження з базового класу переходять на дочірні класи і їх необхідно явно прописувати.

Стосовно інтерфейсів є поняття **коваріантність, контрваріантність та інваріантність**:

Коваріантність: дозволяє використовувати більш детальний тип, ніж був використаний при оголошенні. Узагальнений тип може використовуватись лише для повернення.

```
interface IMessenger<out T>
   {
      T WriteMessage(string text);
   }
```

Контрваріантність: дозволяє використовувати більш універсальний (абстрактний) тип, ніж був використаний при оголошенні. Узагальнений тип може використовуватись лише для передачі через параметри.

```
interface IMessengerV2<in T>
```

```
{
    void SendMessage(T message);
}
```

Інваріантність: дозволяє використовувати тільки заданий тип.

```
interface IMessengerV3<T>
{
    void SendMessage(T message);
}
```

Посилання:

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics

https://www.tutorialsteacher.com/csharp/csharp-generics

https://www.programiz.com/csharp-programming/generics

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-interfaces

https://www.c-sharpcorner.com/UploadFile/Ashush/generics-in-C-Sharp-part-i/

https://www.c-sharpcorner.com/uploadfile/Ashush/generics-in-C-Sharp-part-ii/

Заняття 6: Generic. SOLID, other principles

Термін "**SOLID**" являє собою акронім для набору практик проектування програмного коду та побудови гнучкої та адаптивної програми. Цей термін був введений відомим американським фахівцем у галузі програмування Робертом Мартіном (Robert Martin).

Сам акронім утворений за першими буквами назв SOLID-принципів:

- **Single Responsibility Principle** (Принцип єдиного обов'язку)
- **Open/Closed Principle** (Принцип відкритості/закритості)
- **Liskov Substitution Principle** (Принцип підстановки Лісков)
- **Interface Segregation Principle** (Принцип розподілу інтерфейсів)
- **Dependency Inversion Principle** (Принцип інверсії залежностей)

Принципи SOLID - це не патерни, їх не можна назвати певними догмами, які треба обов'язково застосовувати при розробці, проте їх використання дозволить поліпшити код програми, спростити можливі його зміни та підтримку.

Single Responsibility Principle (Принцип єдиного обов'язку):

Кожен компонент повинен мати одну і тільки одну причину зміни. У C# як компонент може виступати клас, структура, метод. А під обов'язком тут розуміється набір дій, що виконують єдине завдання. Тобто суть принципу полягає в тому, що клас/структура/метод повинні виконувати одне завдання. Весь функціонал компонента повинен бути цілісним, мати високу зв'язність (high cohesion) та низьке зчеплення (low coupling).

Часто виникають випадки порушення принципу SRP. Не рідко даний принцип порушується при змішуванні в одному класі функціональності різних рівнів за стосунку. Для прикладу: клас виконує розрахунки та виводить їх результати користувачу, тобто, поєднує в собі бізнес-логіку та роботу з інтерфейсом користувача. Або клас керує збереженням/отриманням даних та виконуванням над ними розрахунків, що також не бажано. Клас необхідно використовувати тільки для одного завдання – або бізнес-логіки, або розрахунків, або робота з даними. Іншим поширеним випадком є наявність в класі або його методах повністю не пов'язаного між собою функціоналу.

Поширена наступна схема розподілу функціоналу між компонентами:

- Логіка збереження даних;
- Валідація;
- Механізм інформування користувача;
- Обробка помилок;
- Логування;
- Вибір класу або створення його об’єкту;
- Форматування;
- Парсинг;
- Маппінг даних.

Open/Closed Principle (Принцип відкритості/закритості) можна сформулювати так:

Сутність програми повинна бути відкрита для розширення, але закрита для зміни. Наступний приклад **ПОГАНИЙ**:

```
public class EmployeeReport
   {
      public string TypeReport { get; set; }
      public void GenerateReport(Employee em)
      {
         if (TypeReport == "CSV")
         {
            // Генерация отчета в формате CSV
         }
         if (TypeReport == "PDF")
         {
            // Генерация отчета в формате PDF
         }
      }
   }
```

Вирішенням проблеми є створення інтерфейсу та реалізація його для кожного типу звіту (CSV, PDF т.д.)

```
public interface IEmployeeReport
   {
      void GenerateReport(Employee em);
   }
   public class EmployeeCsvReport : IEmployeeReport
   {
      public void GenerateReport(Employee em)
      {
         // реалізація для CSV
      }
   }
   public class EmployeePdfReport : IEmployeeReport
   {
      public void GenerateReport(Employee em)
      {
         // реалізація для PDF
      }
   }
```

В такому разі при додаванні нового типу звіту ми просто створюємо новий клас, який реалізовуватиме логіку для даного типу і не зачіпатимемо логіку звітування для вже існуючих типів звітів.

Liskov Substitution Principle (Принцип підстановки Лісков) є деяким посібником зі створення ієрархій успадкування. Початкове визначення даного принципу, яке було дано Барбарою Лисков у 1988 році, виглядало так:

Якщо кожного об'єкта o1 типу S існує об'єкт o2 типу T, такий, що з будь-якої програми P, визначеної термінах T, поведінка P не змінюється при заміні o2 на o1, то є підтипом T.

Тобто іншими словами, клас S може вважатися підкласом T, якщо заміна об'єктів T на об'єкти S не призведе до зміни роботи програми. Загалом цей принцип можна сформулювати так:

Повинна бути можливість замість базового типу підставити будь-який підтип.

Фактично принцип підстановки Лиск допомагає чіткіше сформулювати ієрархію класів, визначити функціонал для базових і похідних класів і уникнути можливих проблем при застосуванні поліморфізму.

Interface Segregation Principle (Принцип розподілу інтерфейсів) сформулювати так: клієнти не повинні залежати від методів, якими не користуються. Тобто, якщо в ієрархії класів є різний функціонал, специфічний для даного класу, то цей функціонал можна винести в окремий інтерфейс.

Техніки виявлення порушення цього принципу:

- Занадто великі інтерфейси;
- Компоненти в інтерфейсах слабо узгоджені(перегукується із принципом єдиної відповідальності);
- Методи без реалізації(перегукується з принципом Лісків;

Dependency Inversion Principle (Принцип інверсії залежностей) служить для створення слабко пов'язаних сутностей, які легко тестувати, модифікувати та оновлювати. Цей принцип можна сформулювати так:

- Модулі верхнього рівня не повинні залежати від модулів нижнього рівня.
- І ті, й інші повинні залежати від абстракцій.
- Абстракції не повинні залежати від деталей.
- Деталі мають залежати від абстракцій.

```
public interface IMessanger
   {
      void Send();
   }
   public class Email: IMessanger
   {
      public void Send() { }
   }
   public class Sms : IMessanger
   {
      public void Send() { }
   }
   public class Notification
   {
      private IMessanger _messanger;
      public Notification(IMessanger messanger)
      {
         _messanger = messanger;
      }
      public void Send()
      {
```

```
            _messanger.Send();
        }
    }
    public static class DependencyInversionPrincipleDemo
    {
        public static void Demo()
        {
            IMessanger emailMessanger = new Email();
            IMessanger smsMessanger = new Sms();
            Notification notification1 = new Notification(emailMessanger);
            notification1.Send();
            Notification notification2 = new Notification(smsMessanger);
            notification2.Send();
        }
    }
```

Окрім SOLID є ще ряд принципів, які покликані покращити якість розробленого коду:

DRY(don't repeat yourself) – не потрібно повторювати код в кількох місцях. Краще виділяти у сутності – методи, класи, інтерфейси і т.д. Це веде до спрощення керуванням кодом (при необхідності зміни потрібно буде змінити лише в одному місці програми), а, відповідно, і зменшення кількості помилок, які виникли б при копі/пастах та їх виправленнях.

KISS(keep it super simple) – необхідно намагатись писати код найбільш простіший.

YANGI(you ain't gonna need it) – необхідно реалізовувати лише той функціонал, який необхідний.

Посилання:

https://dou.ua/lenta/articles/solid-principles/

https://highload.today/uk/solid/

https://www.infoworld.com/article/3693755/applying-the-dry-kiss-and-yagni-principles-in-c-sharp.html#:~:text=The%20DRY%20(%E2%80%9CDon't,in%20mind%20when%20we%20code.

https://senior.ua/articles/kiss-dry-solid-yagni--navscho-dotrimuvatis-principv-programuvannya

https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

Заняття 7: Складальник сміття

При використанні типів посилань, наприклад, об'єктів класів, для них буде відводитися місце в стеку, тільки там буде зберігатися не значення, а адреса на ділянку пам'яті в хіпі або купі, в якому і будуть знаходитися самі значення даного об'єкта. І якщо об'єкт класу перестає використовуватися, то при очищенні стека посилання на ділянку пам'яті також очищається, проте це не призводить до негайного очищення ділянки пам'яті в купі. Згодом збирач сміття (garbage collector) побачить, що на цю ділянку пам'яті більше немає посилань і очистить її.

Наприклад:

```
Test();

void Test()
{
    Person tom = new Person("Tom");
    Console.WriteLine(tom.Name);
}

record class Person(string Name);
```

У методі Test створюється об'єкт Person. За допомогою оператора new у купі для зберігання об'єкта CLR виділяє ділянку пам'яті. А в стек додає адресу на цю ділянку пам'яті. У неявно визначеному методі Main ми викликаємо метод Test. І після того, як Test відпрацює, місце в стеку очищається, а збирач сміття очищає раніше виділену під зберігання об'єкта Person ділянку пам'яті.

Складальник сміття не запускається відразу після видалення стека посилання на об'єкт, розміщений у купі. Він запускається в той час, коли середовище CLR виявить потребу, наприклад, коли програмі потрібна додаткова пам'ять.

Як правило, об'єкти в купі розташовуються невпорядковано, між ними можуть бути порожнечі. Купа досить сильно фрагментована. Тому після очищення пам'яті в результаті чергового складання сміття об'єкти, що залишилися, переміщаються в один безперервний блок пам'яті. Разом з цим відбувається оновлення посилань, щоб правильно вказувати на нові адреси об'єктів.

Також слід зазначити, що з великих об'єктів існує своя купа - Large Object Heap. У цю купу розміщуються об'єкти, розмір яких більше 85 000 байт. Особливість цієї купи полягає в тому, що при складанні сміття стиснення пам'яті не проводиться через великі витрати, пов'язані з розміром об'єктів.

Незважаючи на те, що на стиснення зайнятого простору потрібен час, та й додаток не зможе продовжувати роботу, поки не відпрацює збирач сміття, проте завдяки подібному підходу також відбувається оптимізація програми.

Тепер, щоб знайти вільне місце в купі середовищі CLR не потрібно шукати острівці порожнього простору серед зайнятих блоків. Їй достатньо звернутися до покажчика купи, який вказує на вільну ділянку пам'яті, що зменшує кількість звернень до пам'яті.

Крім того, щоб знизити витрати від роботи збирача сміття, всі об'єкти в купі поділяються на покоління. Усього існує три **покоління об'єктів: 0, 1 та 2-ге.**

До покоління 0 належать нові об'єкти, які ще жодного разу не піддавалися збиранню сміття.

До покоління 1 відносяться об'єкти, які пережили одну збірку, а до **покоління 2** - об'єкти, що пройшли більше одного складання сміття.

Коли збирач сміття починає роботу, він спочатку аналізує об'єкти з покоління 0. Ті об'єкти, які залишаються актуальними після очищення, підвищуються до покоління 1. Якщо після обробки об'єктів покоління 0 досі необхідна додаткова пам'ять, то збирач сміття приступає до об'єктів з покоління 1. Ті об'єкти, на які вже немає посилань, знищуються, а ті, які, як і раніше, актуальні, підвищуються до покоління 2.

Оскільки об'єкти з покоління 0 є молодшими і часто перебувають у адресному просторі пам'яті поруч друг з одним, їх видалення проходить із найменшими витратами.

/// Клас System.GC

Функціонал збирача сміття у бібліотеці класів .NET представляє клас System.GC. Через статичні методи цей клас дозволяє звертатися до збирача сміття. Як правило, потреба у застосуванні цього класу відсутня.

Найбільш поширеним випадком його використання є складання сміття при роботі з некерованими ресурсами, при інтенсивному виділенні великих обсягів пам'яті, при яких необхідне таке ж швидке звільнення.

Розглянемо деякі **методи та властивості класу System.GC**:

Метод **AddMemoryPressure** інформує середовище CLR про виділення великого обсягу некерованої пам'яті, яку слід врахувати під час планування складання сміття. У зв'язку з цим методом використовується метод **RemoveMemoryPressure**, який вказує на CLR, що раніше виділена пам'ять звільнена, і її не треба враховувати при складання сміття.

Метод **Collect** приводить у дію механізм складання сміття. Перевантажені версії методу дозволяють вказати покоління об'єктів, аж до якого треба зібрати сміття.

Метод **GetGeneration(Object)** дозволяє визначити номер покоління, до якого належить переданий як параметр об'єкт.

Метод **GetTotalMemory** повертає обсяг пам'яті в байтах, яке зайняте в купі, що керується.

Метод **WaitForPendingFinalizers** припиняє роботу поточного потоку до звільнення всіх об'єктів, для яких здійснюється складання сміття.

Приклад:

```
long totalMemory = GC.GetTotalMemory(false);
```

```
Console.WriteLine(totalMemory);

int number = 1000;
totalMemory = GC.GetTotalMemory(false);
Console.WriteLine(totalMemory);

GC.Collect();
GC.WaitForPendingFinalizers();
```

За допомогою перевантажених версій методу **GC.Collect** можна виконати більш точне налаштування складання сміття. Так, його перевантажена версія приймає як параметр число - номер покоління, до якого треба виконати очищення. Наприклад, GC.Collect(0) -видаляються лише об'єкти покоління 0.

Ще одна перевантажена версія приймає ще й другий параметр – перерахування **GCCollectionMode**. Цей перелік може набувати трьох значень:

- **Default**: значення за промовчанням для цього переліку (Forced)
- **Forced**: викликає негайне виконання складання сміття
- **Optimized**: дозволяє збирачеві сміття визначити, чи є поточний момент оптимальним для збирання сміття

Наприклад, негайне складання сміття до першого покоління об'єктів: GC.Collect(1, GCCollectionMode.Forced);

Більшість об'єктів, що використовуються в програмах на C#, відносяться до керованого або managed-коду. Такі об'єкти управляються CLR і легко очищаються збирачем сміття. Однак разом з тим зустрічаються також такі об'єкти, які задіяють некеровані об'єкти (підключення до файлів, баз даних, мережеві підключення і т.д.). Такі некеровані об'єкти звертаються до API операційної системи.

Складальник сміття може впоратися з керованими об'єктами, однак він не знає, як видаляти некеровані об'єкти. І тут розробник повинен сам реалізовувати механізми очищення лише на рівні програмного коду. **Звільнення некерованих ресурсів** передбачає реалізацію одного з двох механізмів:

- **Створення деструктора**
- **Реалізація класом інтерфейсу System.IDisposable**
- **Створення деструкторів:**

Якщо ви раптом програмували мовою C++, то, напевно, вже знайомі з концепцією деструкторів. Метод деструктора має ім'я класу (як і конструктор), перед яким стоїть знак тильди (~). Деструктори можна визначити лише у класах. Деструктор на відміну від конструктора не може мати модифікаторів доступу та параметри. При цьому кожен клас може мати лише один деструктор.

```
class Person
{
   public string Name { get; }
   public Person(string name) => Name = name;

   ~Person()
```

```
    {
        Console.WriteLine($"{Name} has deleted");
    }
}
```

В даному випадку в деструкторі, з метою демонстрації, просто виводиться рядок на консоль, що повідомляє, що об'єкт видалено. Але в реальних програмах в деструктор вкладається логіка звільнення некерованих ресурсів. Однак насправді при очищенні збирач сміття викликає не деструктор, а метод Finalize. Все тому, що компілятор C# компілює деструктор у конструкцію, яка еквівалентна наступній:

```
protected override void Finalize()
{
    try
    {
        // тут інструкції деструктора
    }
    finally
    {
        base.Finalize();
    }
}
```

Метод Finalize вже визначений у базовому для всіх типів класі Object, проте цей метод не можна просто перевизначити. І фактична його реалізація відбувається через створення деструктора.

```
Test();
GC.Collect();   // очищення пам'яті під об'єкт tom
Console.Read();
void Test()
{
    Person tom = new Person("Tom");
}

public class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;

    ~Person()
    {
        Console.WriteLine($"{Name} has been deleted");
    }
}
```

Навіть після завершення методу Test і відповідно видалення зі стека посилання на об'єкт Person у купі, може не наслідувати негайний виклик деструктора. Тільки після завершення всієї програми гарантовано відбудеться очищення пам'яті.

Однак з .NET 5 та в наступних версіях при завершенні програми деструктори не викликаються. Тому в програмі вище для швидшого очищення пам'яті застосовується метод **GC.Collect** і для гарантованого виклику деструктора встановлюється затримка за допомогою виклику **Console.Read(),** який очікує від користувача введення. На рівні пам'яті це виглядає

так: збирач сміття при розміщенні об'єкта в купі визначає, чи підтримує даний об'єкт метод Finalize. І якщо об'єкт має метод Finalize, то покажчик на нього зберігається у спеціальній таблиці, що називається черга фіналізації. Коли настає момент складання сміття, збирач бачить, що даний об'єкт повинен бути знищений, і якщо він має метод Finalize, він копіюється в ще одну таблицю і остаточно знищується лише при наступному проході збирача сміття.

Точний час виклику деструктора не визначено. Крім того, під час фіналізації двох пов'язаних об'єктів порядок виклику деструкторів не гарантується. Тобто якщо об'єкт A зберігає посилання на об'єкт B, і при цьому обидва ці об'єкти мають деструктори, то для об'єкта B деструктор може вже відпрацювати в той час, як для об'єкта A деструктор тільки почне роботу. І тут ми можемо зіткнутися з наступною проблемою: а якщо нам негайно треба викликати деструктор і звільнити всі пов'язані з об'єктом некеровані ресурси? У цьому випадку ми можемо використовувати другий підхід - реалізацію інтерфейсу **IDisposable**.

Інтерфейс IDisposable

```
Test();
void Test()
{
   Person? tom = null;
   try
   {
      tom = new Person("Tom");
   }
   finally
   {
      tom?.Dispose();
   }
}

public class Person : IDisposable
{
   public string Name { get; }
   public Person(string name) => Name = name;

   public void Dispose()
   {
      Console.WriteLine($"{Name} has been disposed");
   }
}
```

Ми розглянули два підходи. Який із них кращий? З одного боку, метод Dispose дозволяє будь-якої миті часу викликати звільнення пов'язаних ресурсів, з другого - програміст, використовує наш клас, може забути поставити в коді виклик методу Dispose. Загалом бувають різні ситуації. І щоб поєднувати плюси обох підходів, ми можемо використовувати **комбінований підхід**.

Microsoft пропонує нам використовувати наступний формалізований шаблон:

```
public class SomeClass : IDisposable
{
   private bool disposed = false;
```

```
    // реалізація інтерфейсу IDisposable.
    public void Dispose()
    {
       Dispose(true); // звільняємо некеровані ресурси
       GC.SuppressFinalize(this); // пригнічуємо фіналізацію
    }

    protected virtual void Dispose(bool disposing)
    {
       if (disposed) return;
       if (disposing)
       {
          // Звільняємо керовані ресурси
       }
       // звільняємо некеровані об'єкти
       disposed = true;
    }

    // Деструктор
    ~SomeClass()
    {
       Dispose(false);
    }
}
```

Логіка очищення реалізується перевантаженою версією методу Dispose (bool disposing). Якщо параметр disposing має значення true, цей метод викликається з публічного методу Dispose, якщо false - то з деструктора. При виклику деструктора як параметр disposing передається значення false, щоб уникнути очищення керованих ресурсів, оскільки ми можемо бути впевненими у тому стані, що вони досі перебувають у пам'яті. І тут залишається покладатися на деструктори цих ресурсів. Та й в обох випадках звільняються некеровані ресурси.

Ще один важливий момент - виклик у методі Dispose методу GC.SuppressFinalize(this).GC.SuppressFinalize не дозволяє системі виконати метод Finalize для цього об'єкта. Якщо ж у класі деструктор не визначено, то виклик цього методу не матиме жодного ефекту. Таким чином, навіть якщо розробник не використовує у програмі метод Dispose, все одно відбудеться очищення та звільнення ресурсів.

Загальні рекомендації щодо використання Finalize та Dispose

- Деструктор слід реалізовувати тільки в тих об'єктів, яким він дійсно необхідний, оскільки метод Finalize дуже впливає на продуктивність.
- Після виклику методу Dispose необхідно блокувати у об'єкта виклик методу Finalize за допомогою GC.SuppressFinalize
- При створенні похідних класів від базових, які реалізують інтерфейс IDisposable, слід також викликати метод Dispose базового класу:

```
public class Base : IDisposable
{
   public virtual void Dispose()
   {
      //
```

```
    }
}
public class Derived : Base
{
    private bool IsDisposed = false;
    public override void Dispose()
    {
        if (IsDisposed) return;
        IsDisposed = true;
        base.Dispose();// Звернення до методу Dispose базового класу
    }
}
```

Віддавайте перевагу комбінованому шаблону, що реалізує як метод Dispose, так і деструктор.

Для автоматичного виклику методу Dispose() використовують конструкцію **using:**

```
using (Person tom = new Person("Tom"))
{
}
```

Конструкція using сформовує блок коду та створює об’єкт певного типу, який реалізовує інтерфейс IDisposable та його метод Dispose. Після виходу з блоку коду, в об’єкта викликається метод Dispose. Дана конструкція можлива лише для типів, які реалізовують інтерфейс **IDisposable**.

Починаючи з версії 8.0 через using можемо задати область дії область видимості зовнішньої функції:

```
void Test()
{
    using Person tom = new Person("Tom");
    // змінна tom доступна в блоці Test()
    Console.WriteLine($"Name: {tom.Name}");
    Console.WriteLine("Конец метода Test");
}
```

В даному випадку метод Dispose викличеться після виконання методу Test().

Посилання:

https://www.loginworks.com/blogs/garbage-collector-work-net-c/

https://ironpdf.com/blog/net-help/csharp-using/

https://dou.ua/lenta/articles/principles-of-garbage-collection/

Заняття 8: Делегати, події

Делегати репрезентують такі об'єкти, які вказують на методи. Тобто делегати – це вказівники на методи і за допомогою делегатів ми можемо викликати ці методи. З допомогою делегатів ми можемо з методами поводитись як зі змінними.

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

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

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

```
Message mes; // 2. Створюємо змінну делегата
mes = Hello; // 3. Привласнюємо цю змінну адресу методу
mes(); // 4. Викликаємо метод
void Hello() => Console.WriteLine("Hello world");
delegate void Message(); // 1. Оголошуємо делегат
```

При цьому делегати необов'язково можуть вказувати лише на методи, які визначені в тому класі, де визначена змінна делегата. Це можуть бути також методи інших класів і структур.

```
Message message1 = Welcome.Print;
Message message2 = new Hello().Display;
message1(); // Welcome
message2(); // Hello
delegate void Message();
class Welcome
{
   public static void Print() => Console.WriteLine("Welcome");
}
class Hello
{
   public void Display() => Console.WriteLine("Hello");
}
```

Місце визначення делегата

```
class Program
{
   delegate void Message(); // 1. Оголошуємо делегат
   static void Main()
   {
      Message mes; // 2. Створюємо змінну делегата
      mes = Hello; // 3. Привласнюємо цю змінну адресу методу
      mes(); // 4. Викликаємо метод
      void Hello() => Console.WriteLine("Hello");
   }
}
```

```
delegate void Message(); // 1. Оголошуємо делегат
class Program
```

```
{
   static void Main()
   {
      Message mes; // 2. Створюємо змінну делегата
      mes = Hello; // 3. Привласнюємо цю змінну адресу методу
      mes(); // 4. Викликаємо метод
      void Hello() => Console.WriteLine("Hello");
   }
}
```

Параметри та результат делегата

```
Operation operation = Add; // делегату присвоюється метод Add
int result = operation(4, 5); // Власне Add(4, 5)
Console.WriteLine(result); // 9

operation = Multiply; // Тепер делегат вказує на метод Multiply
result = operation(4, 5); // Власне Multiply(4, 5)
Console.WriteLine(result); // 20

int Add(int x, int y) => x + y;
int Multiply(int x, int y) => x * y;
```

Присвоєння посилання на метод

Створення об'єкта делегата за допомогою конструктора, в який передається необхідний метод

```
Operation operation1 = Add;
Operation operation2 = new Operation(Add);
int Add(int x, int y) => x + y;
```

Відповідність методів делегату

Як було написано вище, методи відповідають делегату, якщо вони мають один і той же тип, що повертається і той самий набір параметрів. Але треба враховувати, що до уваги також беруться модифікатори ref, in та out.

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

Цьому делегату відповідає, наприклад, наступний метод:

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

А такі методи не відповідають:

```
double SomeMethod2(int g, double n) { return g + n; }
void SomeMethod3(double n, int g) { }
void SomeMethod4(ref int g, double n) { }
void SomeMethod5(out int g, double n) { g = 6; }
```

Додавання та віднімання методів до делегату

У прикладах вище змінна делегата вказувала на один метод. Насправді делегат може вказувати на безліч методів, які мають ту ж сигнатуру і повертають вказаний тип. Усі методи у делегаті потрапляють у спеціальний список - список виклику чи **invocation list**. І при виклику делегата всі методи цього списку послідовно викликаються. І ми можемо додавати до цього списку не один, а кілька методів. Для додавання методів делегат застосовується **операція +=**

```
Message message = Hello;
message += HowAreYou; // Тепер message вказує на два методи
message(); // викликаються обидва методи - Hello і HowAreYou
void Hello() => Console.WriteLine("Hello");
void HowAreYou() => Console.WriteLine("How are you?");
```

У цьому випадку до списку виклику делегата message додаються два методи - Hello і HowAreYou. І при виклику message викликаються відразу обидва ці методи. Однак, в реальності відбуватиметься **створення нового об'єкта делегата**, який отримає методи старої копії делегата і новий метод, новий об'єкт делегата буде присвоєний змінній message. При додаванні делегатів ми можемо додати посилання на той самий метод кілька разів і в списку виклику делегата тоді буде кілька посилань на той самий метод. Відповідно при виклику делегата доданий метод буде викликатися стільки разів, скільки він був доданий:

```
Message message = Hello;
message += HowAreYou;
message += Hello;
message += Hello;
message();
```

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

```
Message? message = Hello;
message += HowAreYou;
message();  // викликаються всі методи делегату
message -= HowAreYou;   // видаляємо метод HowAreYou
if (message != null) message(); // викликається метод Hello

void Hello() => Console.WriteLine("Hello");
void HowAreYou() => Console.WriteLine("How are you?");
```

При видаленні методів із делегата фактично буде **створюватись новий делегат**, який у списку виклику методів міститиме на один метод менше. При видаленні методу може скластися ситуація, що в делегаті не буде методів, і тоді змінна матиме значення null. Тому в даному випадку змінна не просто визначена як змінна типу Message, а саме Message?, тобто типу, який може представляти як делегат Message, так і значення null. Крім того, перед другим викликом ми перевіряємо змінну значення на null.

При видаленні делегат може містити **кілька посилань на один** і той самий метод, то операція -= починає пошук із кінця списку виклику делегата і видаляє лише перше знайдене входження. Якщо такого методу у списку виклику делегата немає, то операція -= не має жодного ефекту.

Об'єднання делегатів

```
Message mes1 = Hello;
Message mes2 = HowAreYou;
```

```
Message mes3 = mes1 + mes2; // об'єднуємо делегати
mes3(); // Викликаються всі методи з mes1 і mes2
void Hello() => Console.WriteLine("Hello");
void HowAreYou() => Console.WriteLine("How are you?");
```

У цьому випадку об'єкт mes3 представляє об'єднання делегатів mes1 і mes2. Об'єднання делегатів означає, що до списку виклику делегата mes3 потраплять усі методи із делегатів mes1 та mes2. І за виклику делегата mes3 всі ці методи одночасно будуть викликані.

Виклик делегата з параметрами

У прикладах вище делегат викликався в стандартний спосіб. Якщо б делегат приймав параметри, то йому необхідно передати необхідні значення:

```
Message mes = Hello;
mes();
Operation op = Add;
int n = op(3, 4);
Console.WriteLine(n);
void Hello() => Console.WriteLine("Hello");
int Add(int x, int y) => x + y;
```

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

```
Message mes = Hello;
mes.Invoke(); // Hello
Operation op = Add;
int n = op.Invoke(3, 4); // 7
void Hello() => Console.WriteLine("Hello");
int Add(int x, int y) => x + y;
```

Якщо делегат приймає параметри, то методу Invoke передаються значення цих параметрів. Якщо делегат порожній, тобто у його списку виклику немає посилань на жоден з методів (тобто делегат дорівнює Null), то за виклику такого делегата ми отримаємо виняток:

```
Message? mes;
//mes(); //! Помилка: делегат дорівнює null
Operation? op = Add;
op -= Add; // делегат op порожній
int n = op(3, 4); // !Помилка: делегат дорівнює null
```

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

```
Message? mes = null;
mes?.Invoke(); // помилки немає, делегат просто не викликається
Operation? op = Add;
op -= Add; // делегат op порожній
int? n = op?.Invoke(3, 4); // Помилки немає, делегат просто не викликається, а n = null
```

Якщо делегат повертає деяке значення, то повертається значення останнього методу зі списку викликів (якщо у списку викликів кілька методів). Наприклад:

```
Operation op = Subtract;
op += Multiply;
```

```
op += Add;
Console.WriteLine(op(7, 2)); // Add(7,2) = 9
int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;
```

Узагальнені делегати

```
Operation<decimal, int> squareOperation = Square;
decimal result1 = squareOperation(5); // 25
Operation<int, int> doubleOperation = Double;
int result2 = doubleOperation(5); // 10
decimal Square(int n) => n * n;
int Double(int n) => n + n;
delegate T Operation<T, K>(K val);
```

Делегати як параметри методів

```
DoOperation(5, 4, Add);         // 9
DoOperation(5, 4, Subtract);    // 1
DoOperation(5, 4, Multiply);    // 20
void DoOperation(int a, int b, Operation op)
{
   Console.WriteLine(op(a, b));
}
int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;
```

DoOperation як параметри приймає два числа і деяку дію у вигляді делегата Operation. Всередині методу викликаємо делегат Operation, передаючи йому числа з перших двох параметрів. При виклику методу DoOperation ми можемо передати до нього як третій параметр метод, який відповідає делегату Operation.

Повернення делегатів із методу

```
Operation operation = SelectOperation(OperationType.Add);
operation = SelectOperation(OperationType.Add); // 14
operation = SelectOperation(OperationType.Subtract); // 6
operation = SelectOperation(OperationType.Multiply); // 40
Operation SelectOperation(OperationType opType)
{
   switch (opType)
   {
      case OperationType.Add: return Add;
      case OperationType.Subtract: return Subtract;
      default: return Multiply;
   }
}
int Add(int x, int y) => x + y;
int Subtract(int x, int y) => x - y;
int Multiply(int x, int y) => x * y;
enum OperationType
{
   Add,    Subtract,    Multiply
}
```

Із делегатами тісно пов'язані **анонімні методи**. Анонімні методи використовують для створення екземплярів делегатів. Визначення анонімних методів починається з ключового слова delegate, після якого йде у дужках список параметрів та тіло методу у фігурних дужках:

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

Наприклад

```
MessageHandler handler = delegate (string mes)
{
   Console.WriteLine(mes);
};
handler("hello world!");
```

Анонімний метод не може існувати сам по собі, він використовується для ініціалізації екземпляра делегата,як у даному випадку змінна **handler є анонімним методом**. І через цю змінну делегата можна викликати цей анонімний метод.

Інший приклад анонімних методів - передача як аргумент для параметра, який представляє делегат:

```
ShowMessage("hello!", delegate (string mes)
{
   Console.WriteLine(mes);
});

static void ShowMessage(string message, MessageHandler handler)
{
   handler(message);
}
delegate void MessageHandler(string message);
```

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

```
MessageHandler handler = delegate
{
   Console.WriteLine("анонімний метод");
};
handler("hello world!"); // анонімний метод
delegate void MessageHandler(string message);
```

Тобто, якщо анонімний метод містить параметри, вони обов'язково повинні відповідати параметрам делегата або анонімний метод взагалі може не містити жодних параметрів, тоді він відповідає будь-якому делегату, який має той же тип значення, що повертається. При цьому параметри анонімного методу не можуть бути опущені, якщо один або кілька параметрів визначено модифікатором out.

Так само, як і звичайні методи, анонімні можуть повертати результат:

```
Operation operation = delegate (int x, int y)
{
   return x + y;
};
int result = operation(4, 5); // 9
delegate int Operation(int x, int y);
```

При цьому анонімний метод має доступ до всіх змінних, визначених у зовнішньому коді:

```
int a = 8;
Operation operation = delegate (int x, int y)
{
   return x + y + a;
};
int result = operation(4, 5); // 17
```

У яких ситуаціях використовують анонімні методи? Коли нам треба визначити одноразову дію, яка не має багато інструкцій та ніде більше не використовується. Зокрема, їх можна використовувати для обробки подій.

Лямбда-вирази представляють спрощений запис анонімних методів. Лямбда-вирази дозволяють створити ємні лаконічні методи, які можуть повертати деяке значення і які можна передати як параметри в інші методи. Ламбда - вирази мають наступний синтаксис: ліворуч від лямбда-оператора => визначається список параметрів, а праворуч блок виразів, який використовує ці параметри:

```
(список_параметрів) => вирази
```

З погляду типу даних лямбда-вираз представляє делегат. Наприклад, визначимо найпростіше лямбда-вираз:

```
Message hello = () => Console.WriteLine("Hello");
hello();      // Hello
hello();      // Hello
hello();      // Hello
delegate void Message();
```

hello представляє делегат Message - тобто деяка дія, яка нічого не повертає і не приймає жодних параметрів. Як значення цій змінній надається лямбда-вираз. Цей лямбда-вираз має відповідати делегату Message – воно теж не приймає жодних параметрів, тому ліворуч від **лямбда-оператора** йдуть «()». А праворуч від лямбда-оператора йде вираз, що виконується - Console.WriteLine("Hello"). Потім у програмі можна викликати цю змінну як метод.

Якщо лямбда-вираз містить кілька дій, то вони поміщаються у фігурні дужки:

```
Message hello = () =>
{
   Console.Write("Hello ");
   Console.WriteLine("World");
};
hello();      // Hello World
delegate void Message();
```

Вище ми визначили змінну hello, яка представляє делегат Message. Але починаючи з версії C# 10 ми можемо застосовувати неявну типізацію (визначення змінної за допомогою оператора var) щодо лямбда-виразів:

```
var hello = () => Console.WriteLine("Hello");
hello();      // Hello
hello();      // Hello
hello();      // Hello
```

Але який тип у разі представляє змінна hello? При неявній типізації компілятор сам намагається зіставити лямбда-вираз з урахуванням його оголошення з яким-небудь делегатом. Наприклад, вище певний лямбда-вираз hello за замовчуванням компілятор буде розглядати як змінну **вбудованого делегата Action**, який не приймає жодних параметрів і нічого не повертає.

Параметри лямбди

```
Operation sum = (x, y) => Console.WriteLine($"{x} + {y} = {x + y}");
sum(1, 2);      // 1 + 2 = 3
sum(22, 14);    // 22 + 14 = 36
delegate void Operation(int x, int y);
```

У даному випадку компілятор бачить, що лямбда-вираз sum є типом Operation,а значить обидва параметри лямбди представляють тип int. Тому жодних проблем не виникне. Однак якщо ми застосовуємо неявну типізацію, то компілятор може мати труднощі, щоб вивести тип делегата для лямбда-вираження, наприклад, у наступному випадку

```
var sum = (x, y) => Console.WriteLine($"{x} + {y} = {x + y}");   // ! Помилка
```

У цьому випадку можна вказати тип параметрів

```
var sum = (int x, int y) => Console.WriteLine($"{x} + {y} = {x + y}");
sum(1, 2);      // 1 + 2 = 3
```

Якщо лямбда має один параметр, для якого не потрібно вказувати тип даних, дужки можна опустити:

```
PrintHandler print = message => Console.WriteLine(message);
print("Hello");         // Hello
delegate void PrintHandler(string message);
```

Повернення результату

```
var sum = (int x, int y) => x + y;
int sumResult = sum(4, 5);                  // 9
Operation multiply = (x, y) => x * y;
int multiplyResult = multiply(4, 5);        // 20
delegate int Operation(int x, int y);
```

або

```
var subtract = (int x, int y) =>
{
   if (x > y) return x - y;
   else return y - x;
};
int result1 = subtract(10, 6);  // 4
```

```
int result2 = subtract(-10, 6);  // 16
```

Додавання та видалення функцій у лямбда-вирази

```
var hello = () => Console.WriteLine("Hello");
var message = () => Console.Write("Hello");
message += () => Console.WriteLine("World"); // Додаємо анонімний лямбда-вираз
message += hello; // додаємо лямбда-вираз зі змінної hello
message += Print; // додаємо метод
message();
Console.WriteLine("--------------"); // для поділу висновку
message -= Print; // видаляємо метод
message -= hello; // видаляємо лямбда-вираз із змінної hello
message?.Invoke(); // на випадок, якщо у message більше немає дій
void Print() => Console.WriteLine("Welcome to C#");
```

Лямбда - вираз як аргумент методу

```
int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result1 = Sum(integers, x => x > 5); //сума чисел більше 5 = 30
int result2 = Sum(integers, x => x % 2 == 0); //сума парних чисел = 20
int Sum(int[] numbers, IsEqual func)
{
    int result = 0;
    foreach (int i in numbers)
    {
        if (func(i))
            result += i;
    }
    return result;
}
delegate bool IsEqual(int x);
```

Лямбда - вираз як результат методу

```
Operation operation = SelectOperation(OperationType.Add); // 14
operation = SelectOperation(OperationType.Subtract); // 6
operation = SelectOperation(OperationType.Multiply); // 40
Operation SelectOperation(OperationType opType)
{
    switch (opType)
    {
        case OperationType.Add: return (x, y) => x + y;
        case OperationType.Subtract: return (x, y) => x - y;
        default: return (x, y) => x * y;
    }
}
enum OperationType
{
    Add, Subtract, Multiply
}
delegate int Operation(int x, int y);
```

Події сигналізують системі про те, що сталися певні дії. І якщо нам треба відстежити ці дії, то ми можемо застосовувати події. У .NET є кілька **вбудованих делегатів**, які

використовуються у різних ситуаціях. І найбільш використовуваними, з якими часто доводиться стикатися, є **Action, Predicate та Func.**

Делегат **Action** представляє деяку дію, яка нічого не повертає, тобто як тип, що повертається, має тип void

```
public delegate void Action()
public delegate void Action<in T>(T obj)
```

Цей делегат має ряд перевантажених версій. Кожна версія приймає різну кількість параметрів: від Action<in T1> до Action<in T1, in T2,....in T16>. Таким чином можна передати до 16 значень у метод. Як правило, цей делегат передається як параметр методу і передбачає виконання певних дій у відповідь на дії, що відбулися.

Наприклад:

```
DoOperation(10, 6, Add);        // 10 + 6 = 16
DoOperation(10, 6, Multiply);   // 10 * 6 = 60
void DoOperation(int a, int b, Action<int, int> op) => op(a, b);
void Add(int x, int y) => Console.WriteLine($"{x} + {y} = {x + y}");
void Multiply(int x, int y) => Console.WriteLine($"{x} * {y} = {x * y}");
```

Делегат **Predicate** приймає один параметр і повертає значення типу bool

```
delegate bool Predicate<in T>(T obj);
Predicate<int> isPositive = (int x) => x > 0;
Console.WriteLine(isPositive(20)); //true
Console.WriteLine(isPositive(-20)); //false
```

Ще одним поширеним делегатом є **Func**. Він повертає результат дії та може приймати параметри. Він також має різні форми: від Func<out T>(), де T - тип значення, що повертається, до Func<in T1, in T2,...in T16, out TResult>(), тобто може приймати до 16 параметрів .

```
TResult Func<out TResult>()
TResult Func<in T, out TResult>(T arg)
TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)
TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)
TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)

.....
```

```
int result1 = DoOperation(6, DoubleNumber); // 12
int result2 = DoOperation(6, SquareNumber); // 36
int DoOperation(int n, Func<int, int> operation) => operation(n);
int DoubleNumber(int n) => 2 * n;
int SquareNumber(int n) => n * n;
Func<int, int, string> createString = (a, b) => $"{a}{b}";
Console.WriteLine(createString(1, 5));  // 15
```

Замикання(closure) представляє об'єкт функції, який запам'ятовує своє лексичне оточення навіть у тому випадку, коли вона виконується поза своєю областю видимості.

Технічно замикання включає три компоненти:

- зовнішня функція, яка визначає деяку область видимості та в якій визначені деякі змінні та параметри - **лексичне оточення**
- змінні та параметри (лексичне оточення), які визначені у зовнішній функції
- вкладена функція, яка використовує змінні та параметри зовнішньої функції

У мові C# реалізувати замикання можна у різний спосіб - з допомогою локальних функцій і лямбда-виразів.

Створення замикань через локальні функції:

```
var fn = Outer(); // fn = Inner, оскільки метод Outer повертає функцію Inner
// Викликаємо внутрішню функцію Inner
fn(); // 6
fn(); // 7
fn(); // 8
Action Outer() // метод чи зовнішня функція
{
   int x = 5; // лексичне оточення – локальна змінна
   void Inner() // локальна функція
   {
      Console.WriteLine(x++); // Операції з лексичним оточенням
   }
   return Inner; // Повертаємо локальну функцію
}
```

Реалізація за допомогою лямбда-виразів

```
var outerFn = () =>
{
   int x = 10;
   var innerFn = () => Console.WriteLine(++x);
   return innerFn;
};
var fn = outerFn();   // fn = innerFn, так як outerFn повертає innerFn
// викликаємо innerFn
fn();   // 11
fn();   // 12
fn();   // 13
```

Застосування параметрів

```
var fn = Multiply(5);
Console.WriteLine(fn(5));   // 25
Console.WriteLine(fn(6));   // 30
Console.WriteLine(fn(7));   // 35

Operation Multiply(int n)
{
   int Inner(int m)
   {
      return n * m;
   }
   return Inner;
}
```

```
delegate int Operation(int n);

var multiply = (int n) => (int m) => n * m;
var fn = multiply(5);
Console.WriteLine(fn(5));   // 25
Console.WriteLine(fn(6));   // 30
Console.WriteLine(fn(7));   // 35
```

Посилання

https://ru.stackoverflow.com/questions/516687/%D0%92-%D1%87%D0%B5%D0%BC-%D1%81%D1%83%D1%82%D1%8C-%D0%BA%D0%BE%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D0%B8-%D0%B8-%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%B0%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D0%B8-%D0%B4%D0%B5%D0%BB%D0%B5%D0%B3%D0%B0%D1%82%D0%BE%D0%B2

https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/covariance-contravariance/using-variance-in-delegates

https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%B8_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%B0%D0%B2%D0%B0%D1%80%D0%B8%D0%B0%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

Заняття 9: Взаємодія з файловою системою. JSON and XML. LINQ

9.1 JSON

JSON(***JavaScript Object Notation***) є одним із найбільш популярних форматів для зберігання та передачі даних. Основна функціональність роботи з JSON зосереджена в просторі імен System.Text.Json. Ключовим типом є клас JsonSerializer, який дозволяє серіалізувати об'єкт в json і, навпаки, десеріалізувати код json в об'єкт C#.

Для збереження об'єкта в json у класі JsonSerializer визначено статичний метод Serialize() та його асинхронний двійник SerializeAsyc(), які мають низку перевантажених версій. Деякі з них:

- string Serialize(Object obj, Type type, JsonSerializerOptions options) - серіалізує об'єкт obj типу type і повертає код json у вигляді рядка.Останній необов'язковий параметр options дозволяє встановити додаткові опції серіалізації
- string Serialize<T>(T obj, JsonSerializerOptions options) - типізована версія серіалізує об'єкт obj типу T і повертає код json у вигляді рядка.
- Task SerializeAsync(Stream utf8Json, Object obj, Type type, JsonSerializerOptions options) - серіалізує об'єкт obj типу type і записує його в потік utf8Json. Останній необов'язковий параметр options дозволяє встановити додаткові опції серіалізації.
- Task SerializeAsync<T>(Stream utf8Json, T obj, JsonSerializerOptions options) - типізована версія серіалізує об'єкт obj типу T у потік utf8Json.

Для десеріалізації коду json в об'єкт C# застосовується метод Deserialize() та його асинхронний двійник DeserializeAsync(), які мають різні версії. Деякі з них:

- object? Deserialize(string json, Type type, JsonSerializerOptions options) - десеріалізує рядок json в об'єкт типу type та повертає десеріалізований об'єкт. Останній необов'язковий параметр options дозволяє встановити додаткові опції десеріалізації
- T? Deserialize<T>(string json, JsonSerializerOptions options) - десеріалізує рядок json в об'єкт типу T і повертає його.
- ValueTask<object?> DeserializeAsync(Stream utf8Json, Type type, JsonSerializerOptions options, CancellationToken token) - десеріалізує дані з потоку utf8Json, який представляє об'єкт JSON, в об'єкт типу type. Останні два параметри необов'язкові: options дозволяє встановити додаткові опції десеріалізації, а token встановлює CancellationToken для скасування завдання. Повертається десеріалізований об'єкт, загорнутий у ValueTask
- ValueTask<T?> DeserializeAsync<T>(Stream utf8Json, JsonSerializerOptions options, CancellationToken token) - десеріалізує дані з потоку utf8Json, який представляє об'єкт JSON, в об'єкт типу T. Повертається десеріалізований об'єкт, обернутий у Value

Розглянемо застосування класу простому прикладі. Серіалізуємо та десеріалізуємо найпростіший об'єкт:

```
Person tom = new Person("Tom", 37);
string json = JsonSerializer.Serialize(tom);
```

```
Console.WriteLine(json);
Person? restoredPerson = JsonSerializer.Deserialize<Person>(json);
Console.WriteLine(restoredPerson?.Name); // Tom

class Person
{
   public string Name { get; }
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
}
```

Деякі зауваження щодо серіалізації/десеріалізації:

- Об'єкт, який піддається десеріалізації, повинен мати або конструктор без параметрів, або конструктор, для всіх властивостей в json-об'єкті, що десеріалізується (відповідність між параметрами конструктора і властивостями json-об'єкта встановлюється на основі назв, причому регістр не грає значення).
- Серіалізації підлягають лише публічні властивості об'єкта (з модифікатором public).

Запис та читання файлу json

Оскільки методи SerializeAsyc/DeserializeAsync можуть приймати потік типу Stream, то ми можемо використовувати файловий потік для збереження і подальшого вилучення даних:

```
using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))
{
   Person tom = new Person("Tom", 37);
   await JsonSerializer.SerializeAsync<Person>(fs, tom);
   Console.WriteLine("Data has been saved to file");
}
using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))
{
   Person? person = await JsonSerializer.DeserializeAsync<Person>(fs);
   Console.WriteLine($"Name: {person?.Name}  Age: {person?.Age}");
}
```

Налаштування серіалізації за допомогою JsonSerializerOptions

За замовчуванням JsonSerializer серіалізує об'єкти мініміфікованого коду (без зайвих пробілів та табуляцій – для мінімального розміру файлу). За допомогою додаткового параметра **JsonSerializerOptions** можна налаштувати механізм серіалізації/десеріалізації, використовуючи властивості JsonSerializerOptions. Деякі з його властивостей:

- AllowTrailingCommas: встановлює, чи потрібно додавати після останнього елемента в json кому. Якщо true, кома додається
- DefaultIgnoreCondition: встановлює, чи серіалізуватиметься/десеріалізуватиметься в json властивості зі значеннями за умовчанням

- IgnoreReadOnlyProperties: аналогічно встановлює, чи серіалізуватимуться властивості, призначені тільки для читання
- WriteIndented: встановлює, чи додаватимуться в json пробіли (умовно кажучи, для краси). Якщо true - встановлюються додаткові опції

```
Person tom = new Person("Tom", 37);
var options = new JsonSerializerOptions
{
   WriteIndented = true
};
string json = JsonSerializer.Serialize<Person>(tom, options);
Console.WriteLine(json);
Person? restoredPerson = JsonSerializer.Deserialize<Person>(json);
Console.WriteLine(restoredPerson?.Name);
```

Налаштування серіалізації за допомогою атрибутів

За умовчанням серіалізації підлягають усі публічні властивості. Крім того, у вихідному об'єкті json усі назви властивостей відповідають назвам властивостей об'єкта C#. Однак за допомогою атрибутів **JsonIgnore** та **JsonPropertyName** ми можемо змінювати назву властивості в JSON та ігнорувати властивості для серіалізації/десеріалізації.

- **JsonIgnore** дозволяє виключити із серіалізації певну властивість.
- **JsonPropertyName** дозволяє замінювати оригінальну назву властивості.

Приклад використання:

```
class Person
{
   [JsonPropertyName("firstname")]
   public string Name { get; }
   [JsonIgnore]
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }
}
```

9.2 XML

XML є одним із найпоширеніших стандартів документів, який дозволяє у зручній формі зберігати складні за структурою дані та перевіряти чи коректні дані збереглись. Приклад такого файлу:

```
<?xml version="1.0" encoding="utf-8" ?>
<people>
  <person name="Tom">
    <company>Microsoft</company>
    <age>37</age>
  </person>
  <person name="Bob">
    <company>Google</company>
```

```
    <age>41</age>
  </person>
</people>
```

XML - документ починається з рядка (хідера)<?xml version="1.0" encoding="utf-8" ?>. Він задає версію (1.0) та кодування(utf-8) xml.Далі йде власне вміст документа.

XML-документ повинен мати **один єдиний** кореневий елемент, всередину якого містяться всі інші елементи. В даному випадку таким елементом є <people>. Всередині кореневого елемента є набір елементів <person>. Поза кореневим елементом ми не можемо розмістити елементи person.

Кожен елемент визначається за допомогою **тега**, що відкриває і закриває, наприклад, <person> і </person>, всередині яких міститься значення або вміст елементів. Також елемент може мати скорочене оголошення: < person /> - в кінці елемента міститься слеш. Елемент може мати **вкладені елементи та атрибути**. В даному випадку кожен елемент person має два **вкладені елементи** company і age, та **атрибут** name. **Атрибути** визначаються в тілі елемента та мають таку форму: назва = "значення". Наприклад: < person name = "Bill Gates" >, в даному випадку атрибут називається name і має значення Bill Gates.

Усередині простих елементів міститься їхнє значення. Наприклад, <company>Google</company> - елемент company має значення Google. Назви елементів є реєстрозалежними, тому <company> і <COMPANY> будуть представляти різні елементи.

Таким чином, весь список Users з коду C# зіставляється з кореневим елементом <people>, кожен об'єкт Person - з елементом <person>, а кожна властивість об'єкта Person - з атрибутом або вкладеним елементом елемента <person>.

Що використовувати для властивостей – вкладені елементи чи атрибути? Це питання переваг - ми можемо використовувати як атрибути, так і вкладені елементи.

Так, у попередньому прикладі цілком можна використовувати замість атрибута вкладений елемент:

```
<?xml version="1.0" encoding="utf-8" ?>
<people>
  <person>
    <name>Tom</name>
    <company>Microsoft</company>
    <age>37</age>
  </person>
  <person>
    <name>Bob</name>
    <company>Google</company>
    <age>41</age>
  </person>
</people>
```

Робота з XML за допомогою класів System.Xml

Для роботи з XML C# можна використовувати кілька підходів. У перших версіях фреймворку основний функціонал роботи з XML надавав простір імен System.Xml. У ньому визначено ряд класів, які дозволяють маніпулювати XML-документом:

- XmlNode: представляє вузол xml. Як вузл може використовуватися весь документ, так і окремий елемент
- XmlDocument: представляє весь xml-документ
- XmlElement: представляє окремий елемент. Наслідується від класу XmlNode
- XmlAttribute: представляє атрибут елемента
- XmlText: представляє значення елемента у вигляді тексту, тобто той текст, який знаходиться в елементі між його тегами, що відкриваються і закриваються
- XmlComment: представляє коментар у xml
- XmlNodeList: використовується для роботи зі списком вузлів

Ключовим класом, який дозволяє маніпулювати вмістом xml, є XmlNode, тому розглянемо деякі його основні методи та властивості:

- Властивість Attributes повертає об'єкт XmlAttributeCollection, який представляє колекцію атрибутів поточного вузла.
- Властивість ChildNodes повертає колекцію дочірніх вузлів для цього вузла.
- Властивість HasChildNodes повертає true, якщо поточний вузол має дочірні вузли.
- Властивість FirstChild повертає перший дочірній вузол
- Властивість LastChild повертає останній дочірній вузол
- Властивість InnerText повертає текстове значення вузла
- Властивість InnerXml повертає всю внутрішню розмітку xml вузла
- Властивість Name повертає назву вузла. Наприклад, <user> - Name = "user"
- Властивість ParentNode повертає батьківський вузол поточного вузла

```
XmlDocument xDoc = new XmlDocument();
xDoc.Load("../../../people.xml");
XmlElement? xRoot = xDoc.DocumentElement; // отримуємо кореневий елемент
if (xRoot != null)
{
   foreach (XmlElement xnode in xRoot)
   {
      XmlNode? attr = xnode.Attributes.GetNamedItem("name");
      Console.WriteLine(attr?.Value);
   }
}
```

Щоб розпочати роботу з документом xml, нам треба створити об'єкт XmlDocument і потім завантажити в нього xml-файл: xDoc.Load("people.xml"); Під час аналізу xml ми отримуємо кореневий елемент документа за допомогою властивості xDoc.DocumentElement. Далі вже відбувається власне розбір вузлів документа.

У циклі foreach(XmlNode xnode in xRoot) пробігаємось по всіх дочірніх вузлах кореневого елемента. Оскільки дочірні вузли є елементами <person>, ми можемо отримати їх атрибути: XmlNode attr = xnode.Attributes.GetNamedItem("name"); та вкладені елементи: foreach (XmlNode childnode in xnode.ChildNodes)

Щоб визначити, що за вузол перед нами, ми можемо порівняти його назву: if (childnode.Name == "company") Подібним чином ми можемо створити об'єкти класів і структур за даними xml:

```
var people = new List<Person>();
….
XmlElement? xRoot = xDoc.DocumentElement;
if (xRoot != null)
{
   foreach (XmlElement xnode in xRoot)
   {
      Person person = new Person();
      XmlNode? attr = xnode.Attributes.GetNamedItem("name");
      person.Name = attr?.Value;
      foreach (XmlNode childnode in xnode.ChildNodes)
      {
         if (childnode.Name == "company")
            person.Company = childnode.InnerText;
         if (childnode.Name == "age")
            person.Age = int.Parse(childnode.InnerText);
      }
      people.Add(person);
   }
   foreach (var person in people)
      Console.WriteLine($"{person.Name} ({person.Company}) - {person.Age}");
}
```

Зміна документа XML

Для редагування xml-документа (зміни, додавання, видалення елементів) ми можемо скористатися методами класу **XmlNode**:

- AppendChild: додає до кінця поточного вузла новий дочірній вузол
- InsertAfter: додає новий вузол після певного вузла
- InsertBefore: додає новий вузол до певного вузла
- RemoveAll: видаляє всі дочірні вузли поточного вузла
- RemoveChild: видаляє біля поточного вузла один дочірній вузол і повертає його

Клас **XmlDocument** додає ще низку методів, які дозволяють створювати нові вузли:

- CreateNode: створює вузол будь - якого типу
- CreateElement: створює вузол типу XmlDocument
- CreateAttribute: створює вузол типу XmlAttribute
- CreateTextNode: створює вузол типу XmlTextNode
- CreateComment: створює коментар

```
…….
XmlElement? xRoot = xDoc.DocumentElement;
XmlElement personElem = xDoc.CreateElement("person");
XmlAttribute nameAttr = xDoc.CreateAttribute("name");
XmlElement companyElem = xDoc.CreateElement("company");
XmlElement ageElem = xDoc.CreateElement("age");
XmlText nameText = xDoc.CreateTextNode("Mark");
XmlText companyText = xDoc.CreateTextNode("Facebook");
```

```
XmlText ageText = xDoc.CreateTextNode("30");
nameAttr.AppendChild(nameText);
companyElem.AppendChild(companyText);
ageElem.AppendChild(ageText);
personElem.Attributes.Append(nameAttr);
personElem.AppendChild(companyElem);
personElem.AppendChild(ageElem);
xRoot?.AppendChild(personElem);
xDoc.Save("../../../people.xml");
```

Видалення вузлів

```
…….
XmlElement? xRoot = xDoc.DocumentElement;
XmlNode? firstNode = xRoot?.FirstChild;
if (firstNode is not null) xRoot?.RemoveChild(firstNode);
xDoc.Save("../../../people.xml");
```

XPath представляє мову запитів у XML. Він дозволяє вибирати елементи, які відповідають певному селектору. Розглянемо деякі найпоширеніші селектори:

- «**.**» - вибір поточного вузла
- «**..**» - вибір батьківського вузла
- «*» - вибір усіх дочірніх вузлів поточного вузла
- «person» - вибір всіх вузлів з певним ім'ям, у разі з ім'ям "person"
- «@name» - вибір атрибуту поточного вузла, після знаку @ вказується назва атрибуту (у разі "name")
- «@*» - вибір усіх атрибутів поточного вузла
- «element[3]» - вибір певного дочірнього вузла за індексом. Тут - третього вузла
- «//person» - вибір у документі всіх вузлів з ім'ям "person"
- «person[@name='Tom']» - вибір елементів із певним значенням атрибута. Тут вибираються всі елементи " person" з атрибутом name='Tom'
- «person[company='Microsoft']» - вибір елементів із певним значенням вкладеного елемента. В даному випадку вибираються всі елементи "person", у яких дочірній елемент "company" має значення 'Microsoft'
- //person/company» - вибір в документі всіх вузлів з іменем "company", які розміщені в елементах "person"

Дії запитів XPath засновано на застосуванні двох методів класу XmlElement:

- SelectSingleNode(): вибір одного елемента з вибірки. Якщо вибірка, по запиту, має кілька вузлів, то береться перший
- SelectNodes(): вибірка по запиту колекції вузлів у вигляді об'єкта XmlNodeList

```
…..
XmlElement? xRoot = xDoc.DocumentElement;
XmlNodeList? nodes = xRoot?.SelectNodes("*");
if (nodes is not null)
{
    foreach (XmlNode node in nodes)
    Console.WriteLine(node.OuterXml);
}
```

```
XmlNodeList? personNodes = xRoot?.SelectNodes("person");
if (personNodes is not null)
{
    foreach (XmlNode node in personNodes)
    Console.WriteLine(node.SelectSingleNode("@name")?.Value);
}

XmlNodeList? companyNodes = xRoot?.SelectNodes("//person/company");
if (companyNodes is not null)
{
    foreach (XmlNode node in companyNodes)
    Console.WriteLine(node.InnerText);
}
```

Linq to Xml. Створення документа XML

Ще один підхід до роботи з Xml представляє технологія LINQ to XML. Вся функціональність LINQ to XML міститься у просторі імен System.Xml.Linq. Розглянемо основні класи цього простору імен:

- XAttribute: представляє атрибут xml – елемента
- XComment: представляє коментар
- XDocument: представляє весь XML – документ
- XElement: представляє окремий xml - елемент

Ключовим класом є XElement, який дозволяє отримувати вкладені елементи та керувати ними. Серед його методів можна назвати такі:

- Add(): додає новий атрибут або елемент
- Attributes(): повертає колекцію атрибутів для цього елемента
- Elements(): повертає всі дочірні елементи цього елемента
- Remove(): видаляє цей елемент із батьківського об'єкта
- RemoveAll(): видаляє всі дочірні елементи та атрибути цього елемента

Отже, використовуємо функціональність LINQ to XML і створимо новий XML-документ:

```
using System.Xml.Linq;
XDocument xdoc = new XDocument();

XElement tom = new XElement("person");
XAttribute tomNameAttr = new XAttribute("name", "Tom");
XElement tomCompanyElem = new XElement("company", "Microsoft");
XElement tomAgeElem = new XElement("age", 37);
tom.Add(tomNameAttr);
tom.Add(tomCompanyElem);
tom.Add(tomAgeElem);

XElement bob = new XElement("person");
XAttribute bobNameAttr = new XAttribute("name", "Bob");
XElement bobCompanyElem = new XElement("company", "Google");
XElement bobAgeElem = new XElement("age", 41);
```

```
bob.Add(bobNameAttr);
bob.Add(bobCompanyElem);
bob.Add(bobAgeElem);

XElement people = new XElement("people");
people.Add(tom);
people.Add(bob);
xdoc.Add(people);
xdoc.Save("../../../people_v2.xml");
```

Або

```
XDocument xdoc = new XDocument(new XElement("people",
   new XElement("person",
      new XAttribute("name", "Tom"),
      new XElement("company", "Microsoft"),
      new XElement("age", 37)),
   new XElement("person",
      new XAttribute("name", "Bob"),
      new XElement("company", "Google"),
      new XElement("age", 41))));
xdoc.Save("../../../people_v3.xml");
```

Вибірка елементів у LINQ to XML

```
XDocument xdoc = XDocument.Load("../../../people.xml");
XElement? people = xdoc.Element("people");
if (people is not null)
{
   foreach (XElement person in people.Elements("person"))
   {
      XAttribute? name = person.Attribute("name");
      XElement? company = person.Element("company");
      XElement? age = person.Element("age");
      Console.WriteLine($"Person: {name?.Value}");
      Console.WriteLine($"Company: {company?.Value}");
      Console.WriteLine($"Age: {age?.Value}");
      Console.WriteLine();
   }
}
```

Або

```
XDocument xdoc = XDocument.Load("../../../people.xml");
var microsoft = xdoc.Element("people")?   // кореневий вузол people
                           .Elements("person")  // всі елементи person
                             // всі person, в кого company = Microsoft
                           .Where(p => p.Element("company")?.Value == "Microsoft")
                            // для кожного елемента створюємо анонімний об'єкт
                           .Select(p => new
                           {
                              name = p.Attribute("name")?.Value,
                              age = p.Element("age")?.Value,
                              company = p.Element("company")?.Value
                           });
```

```
if (microsoft is not null)
{
   foreach (var person in microsoft)
   {
      Console.WriteLine($"Name: {person.name}  Age: {person.age}");
   }
}
```

Або

```
XDocument xdoc = XDocument.Load("../../../people.xml");
var tom = xdoc.Element("people")?   //кореневий вузол people
                    .Elements("person")            // всі елементи person
                    .FirstOrDefault(p => p.Attribute("name")?.Value == "Bob");

var name = tom?.Attribute("name")?.Value;
var age = tom?.Element("age")?.Value;
var company = tom?.Element("company")?.Value;
Console.WriteLine($"Name: {name}  Age: {age}  Company: {company}");
```

Зміна документа в LINQ to XML

Додавання даних

```
XElement? root = xdoc.Element("people");
if (root != null)
{
   root.Add(new XElement("person",
            new XAttribute("name", "Sam"),
            new XElement("company", "JetBrains"),
            new XElement("age", 28)));
   xdoc.Save("../../../people.xml");
}
Console.WriteLine(xdoc);
```

Зміна даних

```
// елементи person з name = "Tom"
var tom = xdoc.Element("people")?
                    .Elements("person")
                    .FirstOrDefault(p => p.Attribute("name")?.Value == "Tom");

if (tom != null)
{
   var name = tom.Attribute("name");
   if (name != null) name.Value = "Tomas";

   var age = tom.Element("age");
   if (age != null) age.Value = "22";

   xdoc.Save("../../../people.xml");
}
Console.WriteLine(xdoc);
```

Видалення даних

```
XElement? root = xdoc.Element("people");
if (root != null)
```

```
{
   // person з name = "Bob"
   var bob = root.Elements("person")
                  .FirstOrDefault(p => p.Attribute("name")?.Value == "Bob");
   if (bob != null)
   {
      bob.Remove();
      xdoc.Save("../../../people.xml");
   }
}
Console.WriteLine(xdoc);
```

Серіалізація у XML.XmlSerializer

Для зручного збереження та вилучення об'єктів із файлів xml може використовуватися клас XmlSerializer із простору імен **System.Xml.Serialization**. Цей клас спрощує збереження складних об'єктів у форматі xml і подальше їх вилучення. Для створення об'єкта XmlSerializer можна застосовувати різні конструктори, але майже всі вони вимагають вказівки типу даних, які будуть серіалізуватися та десеріалізуватися:

```
using System.Xml.Serialization;
XmlSerializer xmlSerializer = new XmlSerializer(typeof(Person));
//[Serializable]
class Person { }
```

У цьому випадку XmlSerializer працюватиме лише з об'єктами класу Person. Слід враховувати, що XmlSerializer передбачає деякі обмеження. Наприклад, клас, що підлягає серіалізації, повинен мати стандартний конструктор без параметрів та повинен мати модифікатор доступу public. Також серіалізації підлягають лише відкриті члени. Якщо в класі є поля або властивості з private модифікатором, то при серіалізації вони будуть ігноруватися. Крім того, властивості повинні мати загальнодоступні гетери та сеттери.

Серіалізація

```
using System.Xml.Serialization;
Person person = new Person("Tom", 37);
XmlSerializer xmlSerializer = new XmlSerializer(typeof(Person));

using (FileStream fs = new FileStream("../../../person_v4.xml", FileMode.OpenOrCreate))
{
   xmlSerializer.Serialize(fs, person);
}

[Serializable]
public class Person
{
   public string Name { get; set; } = "Undefined";
   public int Age { get; set; } = 1;

   public Person() { }
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
```

```
    }
}
```

Десеріалізація

```
using System.Xml.Serialization;
XmlSerializer xmlSerializer = new XmlSerializer(typeof(Person));
using (FileStream fs = new FileStream("person_v4.xml", FileMode.OpenOrCreate))
{
    Person? person = xmlSerializer.Deserialize(fs) as Person;
    Console.WriteLine($"Name: {person?.Name}  Age: {person?.Age}");
}
```

Серіалізація та десеріалізація колекцій

```
using System.Xml.Serialization;
Person[] people = new Person[]
{
    new Person("Tom", 37),
    new Person("Bob", 41)
};

XmlSerializer formatter = new XmlSerializer(typeof(Person[]));
using (FileStream fs = new FileStream("../../../people_v5.xml", FileMode.OpenOrCreate))
{
    formatter.Serialize(fs, people);
}

using (FileStream fs = new FileStream("../../../people_v5.xml", FileMode.OpenOrCreate))
{
    Person[]? newpeople = formatter.Deserialize(fs) as Person[];
    if (newpeople != null)
    {
        foreach (Person person in newpeople)
        {
            Console.WriteLine($"Name: {person.Name} --- Age: {person.Age}");
        }
    }
}
```

Або

```
var microsoft = new Company("Microsoft");
var google = new Company("Google");
Person[] people = new Person[]
{
    new Person("Tom", 37, microsoft),
    new Person("Bob", 41, google)
};
XmlSerializer formatter = new XmlSerializer(typeof(Person[]));
using (FileStream fs = new FileStream("../../../people_v6.xml", FileMode.OpenOrCreate))
{
    formatter.Serialize(fs, people);
}
using (FileStream fs = new FileStream("../../../people_v6.xml", FileMode.OpenOrCreate))
```

```
{
    Person[]? newpeople = formatter.Deserialize(fs) as Person[];
    if (newpeople != null)
    {
        foreach (Person person in newpeople)
        {
            Console.WriteLine($"Name: {person.Name}");
            Console.WriteLine($"Age: {person.Age}");
            Console.WriteLine($"Company: {person.Company.Name}");
        }
    }
}
public class Company
{
    public string Name { get; set; } = "Undefined";
    public Company() { }
    public Company(string name) => Name = name;
}
public class Person
{
    public string Name { get; set; } = "Undefined";
    public int Age { get; set; } = 1;
    public Company Company { get; set; } = new Company();
    public Person() { }
    public Person(string name, int age, Company company)
    {
        Name = name;
        Age = age;
        Company = company;
    }
}
```

9.3 LINQ

LINQ(Language - Integrated Query) представляє просту та зручну мову запитів до джерела даних. Як джерело даних може бути об'єкт, що реалізує інтерфейс IEnumerable (наприклад, стандартні колекції, масиви), набір даних DataSet, документ XML. Але незалежно від типу джерела LINQ дозволяє застосувати до всіх той самий підхід для вибірки даних.

Існує кілька різновидів LINQ:

- LINQ to Objects: застосовується для роботи з масивами та колекціями
- LINQ to Entities: використовується при зверненні до баз даних через технологію Entity Framework
- LINQ to XML: застосовується під час роботи з файлами XML
- LINQ to DataSet: застосовується під час роботи з об'єктом DataSet (таблицями як ексель)
- Parallel LINQ (PLINQ): використовується для виконання паралельних запитів

У цьому розділі мова йтиме насамперед про LINQ to Objects, але в наступних матеріалах також торкнуться й інших різновидів LINQ. Основна частина функціональності LINQ зосереджена у просторі імен **System.LINQ**. У проектах під .NET 6 цей простір імен підключається за замовчуванням.

У чому зручність LINQ? Подивимося на найпростіший приклад:

Виберемо з масиву рядки, які починаються на певну літеру, наприклад, літеру "T", та відсортуємо отриманий список:

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };
var selectedPeople = new List<string>();
foreach (string person in people)
{
   if (person.ToUpper().StartsWith("T"))
      selectedPeople.Add(person);
}
selectedPeople.Sort();
```

Для роботи з колекціями можна використовувати два способи:

- Оператори запитів LINQ
- Методи розширень LINQ

Оператори запитів LINQ

```
var selectedPeople = from p in people //p репрезентує кожен елемент в people
                     where p.ToUpper().StartsWith("T") //фільтр
                     orderby p  // впорядкування по зростанню
                     select p; // вибираємо заданий елемент в колекцію
```

Методи розширення LINQ

```
var selectedPeople = people.Where(p => p.ToUpper().StartsWith("T")).OrderBy(p => p);
```

Перелік методів розширень LINQ

- Select: визначає проекцію вибраних значень
- Where: визначає фільтр вибірки
- OrderBy: впорядковує елементи по зростанню
- OrderByDescending: впорядковує елементи по спаданню
- ThenBy: задає додаткові критерії для сортування елементів по зростанню
- ThenByDescending: задає додаткові критерії для сортування елементів по спаданню
- Join: з’єднує колекції по певному принципу (заданій ознаці)
- Aggregate: застосовує до елементів колекції агрегатну функцію, яка зводить їх до одного об’єкта
- GroupBy: групує елементів по ключу
- ToLookup: групує елементи по ключу, одночасно всі елементи додаються в словник
- GroupJoin: одночасно з’єднує колекції та групує елементи по ключу
- Reverse: розставляє елементи у зворотньому порядку
- All: визначає чи всі елементи колекції задовольняють певну умову
- Any: визначає чи хоча б один елемент колекції задовільняє певну умову
- Contains: визначає, чи містить колекція заданий елемент
- Distinct: видаляє дублікати з колекції
- Except: повертає різницю двох колекцій – елементи, які зустрічаються тільки в одній колекції
- Union: об’єднує дві однородні колекції

- Intersect: повертає колекцію з пересічення двох колекцій (містить елементи, які є в обох колекціях)
- Count: повертає кількість елементів у колекції, які задовольняють задану вимогу
- Sum: повертає суму числових значень в колекції
- Average: повертає середнє значення елементів колекції
- Min: повертає мінімальне значення
- Max: повертає максимальне значення
- Take: вибирає певне число елементів (задане число)
- Skip: пропускає певне число елементів (задане число)
- TakeWhile: повертає елементи з колекції, поки вони задовольняють умову
- SkipWhile: пропускає елементи з колекції, поки вони задовольняють умову, і потім повертає елементи, які залишились
- Concat: об’єднує дві колекції
- Zip: об’єднує дві колекції у відповідності до певної умови
- First: повертає перший елемент колекції
- FirstOrDefault: повертає перший елемент колекції, або значення по замовчуванні
- Single: повертає один елемент колекції. Якщо в колекції більше або менше елементів – генерується виняток
- SingleOrDefault: повертає один елемент колекції. Якщо в колекції менше елементів – повертає значення по замовчуванню. Якщо більше як один елемент – генерується виняток
- ElementAt: повертає елемент по вказаному індексу. Якщо індекс за межами діапазону – виняток
- ElementAtOrDefault: повертає елемент по вказаному індексу. Якщо індекс за межами – значення по замовчуванню
- Last: повертає останній елемент колекції. Якщо колекція порожня – виняток
- LastOrDefault: повертає останній елемент колекції. Якщо колекція порожня – значення по замовчуванню

Приклад 1

```
var people = new List<Person>
{
   new Person ("Tom", 23),  new Person ("Bob", 27),  new Person ("Sam", 29),
   new Person ("Alice", 24)
};
var names = from p in people select p.Name;
```

Приклад 2

```
var names = people.Select(u => u.Name);
```

Приклад 3

```
var personel = from p in people
            select new
            {
               FirstName = p.Name,
               Year = DateTime.Now.Year - p.Age
            };
```

Іноді виникає потреба зробити у запитах LINQ якісь додаткові **проміжні обчислення**. Для цього ми можемо задати у запитах свої змінні за допомогою оператора let:

```
var personnel = from p in people
                let name = $"Mr. {p.Name}"
                let year = DateTime.Now.Year - p.Age
                select new
                {
                   Name = name,
                   Year = year
                };
```

Вибірка з кількох джерел

```
var courses = new List<Course> { new Course("C#"), new Course("Java") };
var students = new List<Student> { new Student("Tom"), new Student("Bob") };
var enrollments = from course in courses    //  один курс
                  from student in students      //  один студент
                  select new { Student = student.Name, Course = course.Title };    // з'єднання кожного студента з кожним курсом
```

SelectMany та зведення об'єктів

Метод SelectMany дозволяє звести набір колекцій до однієї колекції.

```
var companies = new List<Company>
{
   new Company("Microsoft", new List<Person> {new Person("Tom"), new Person("Bob")}),
   new Company("Google", new List<Person> {new Person("Sam"), new Person("Mike")}),
};
var employees = companies.SelectMany(c => c.Staff);
record class Company(string Name, List<Person> Staff);
record class Person(string Name);
```

або

```
var employees = from c in companies
                from emp in c.Staff
                select emp;
```

або

```
var employees = companies.SelectMany(c => c.Staff,
                        (c, emp) => new { Name = emp.Name, Company = c.Name });
```

або

```
var employees = from c in companies
                from emp in c.Staff
                select new { Name = emp.Name, Company = c.Name };
```

Фільтрування колекції

```
string[] people = { "Tom", "Alice", "Bob", "Sam", "Tim", "Tomas", "Bill" };
var selectedPeople = people.Where(p => p.Length == 3); // "Tom", "Bob", "Sam", "Tim"
```

або

```
var selectedPeople = from p in people
                     where p.Length == 3
                     select p;
```

або

```
int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };
var evens1 = numbers.Where(i => i % 2 == 0 && i > 10); // метод розширення
var evens2 = from i in numbers// оператори запиту
             where i % 2 == 0 && i > 10
             select i;
```

або

```
var people = new List<Person>
{
   new Person ("Tom", 23, new List<string> {"english", "german"}),
   new Person ("Bob", 27, new List<string> {"english", "french" }),
   new Person ("Sam", 29, new List<string>  { "english", "spanish" }),
   new Person ("Alice", 24, new List<string> {"spanish", "german" })
};
var selectedPeople = from p in people
                     where p.Age > 25
                     select p;
record class Person(string Name, int Age, List<string> Languages);
```

або

```
var selectedPeople = from person in people
                     from lang in person.Languages
                     where person.Age < 28
                     where lang == "english"
                     select person;
```

або

```
var selectedPeople = people.SelectMany(u => u.Languages,
                        (u, l) => new { Person = u, Lang = l })
                     .Where(u => u.Lang == "english" && u.Person.Age < 28)
                     .Select(u => u.Person);
```

Додатковий метод розширення - **OfType**() дозволяє відфільтрувати дані колекції за певним типом

```
var people = new List<Person>
{
   new Student("Tom"), new Person("Sam"), new Student("Bob"), new Employee("Mike")
};
var students = people.OfType<Student>();
record class Person(string Name);
record class Student(string Name) : Person(Name);
record class Employee(string Name) : Person(Name);
```

Сортування

```
int[] numbers = { 3, 12, 4, 10 };
```

```
var orderedNumbers = from i in numbers
                     orderby i
                     select i;
```
або
```
string[] people = { "Tom", "Bob", "Sam" };
var orderedPeople = from p in people
                    orderby p select p;
```
або
```
var orderedNumbers = numbers.OrderBy(n => n);
```
або
```
var orderedPeople = people.OrderBy(p => p);
```
або
```
var people = new List<Person>
{
   new Person("Tom", 37), new Person("Sam", 28), new Person("Tom", 22),
    new Person("Bob", 41),
};
var sortedPeople1 = from p in people
                    orderby p.Name
                    select p;
```
або
```
var sortedPeople2 = people.OrderBy(p => p.Name);
```
або
```
int[] numbers = { 3, 12, 4, 10 };
var orderedNumbers = from i in numbers
                     orderby i descending
                     select i;
```
або
```
var orderedNumbers = numbers.OrderByDescending(n => n);
```
або
```
var people = new List<Person>
{
   new Person("Tom", 37), new Person("Sam", 28), new Person("Tom", 22),
  new Person("Bob", 41),
};
var sortedPeople1 = from p in people
                    orderby p.Name, p.Age
                    select p;
```
або
```
string[] people = new[] { "Kate", "Tom", "Sam", "Mike", "Alice" };
var sortedPeople = people.OrderBy(p => p, new CustomStringComparer());
class CustomStringComparer : IComparer<String>
```

```
{
    public int Compare(string? x, string? y)
    {
        int xLength = x?.Length ?? 0; // якщо x рівний null, то довжина 0
        int yLength = y?.Length ?? 0; // якщо x рівний null, то довжина 0
        return xLength - yLength;
    }
}
```

Об'єднання, перетин та різниця колекцій

```
string[] soft = { "Microsoft", "Google", "Apple" };
string[] hard = { "Apple", "IBM", "Samsung" };
var result = soft.Except(hard); // різниця
var result = soft.Intersect(hard); // перетин
string[] soft = { "Microsoft", "Google", "Apple", "Microsoft", "Google" };
var result = soft.Distinct();// видалення дублів
var result = soft.Union(hard); // об’єднання колекцій
```

або

```
Person[] students = { new Person("Tom"), new Person("Bob"), new Person("Sam") };
Person[] employees = { new Person("Tom"), new Person("Bob"), new Person("Mike") };
var people = students.Union(employees);
class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;
    public override bool Equals(object? obj)
    {
        if (obj is Person person) return Name == person.Name;
        return false;
    }
    public override int GetHashCode() => Name.GetHashCode();
}
```

Агрегатні операції

```
int[] numbers = { 1, 2, 3, 4, 5 };
int query = numbers.Aggregate((x, y) => x - y); // -13
```

або

```
string[] words = { "Gaudeamus", "igitur", "Juvenes", "dum", "sumus" };
var sentence = words.Aggregate("Text:", (first, next) => $"{first} {next}");// Text: Gaudeamus igitur Juvenes dum sumus
```

або

```
int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };
int size = numbers.Count();  // 10
```

або

```
int size = numbers.Count(i => i % 2 == 0 && i > 10); // 3
```

або

```
int sum = numbers.Sum();
```

або

```
Person[] people = { new Person("Tom", 37), new Person("Sam", 28), new Person("Bob", 41) };
int ageSum = people.Sum(p => p.Age); // 106
```

або

```
int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

int min = numbers.Min();// Min: 1
int max = numbers.Max();// Max: 88
double average = numbers.Average();// Average: 34
```

або

```
Person[] people = { new Person("Tom", 37), new Person("Sam", 28), new Person("Bob", 41) };
int minAge = people.Min(p => p.Age); // Min Age: 28
int maxAge = people.Max(p => p.Age); // Max Age: 41
double averageAge = people.Average(p => p.Age); // Average Age: 35,33
```

Методы Skip и Take

Ряд методів у LINQ дозволяють отримати частину колекції, зокрема такі методи як Skip, Take, SkipWhile, TakeWhile.

```
string[] people = { "Tom", "Sam", "Bob", "Mike", "Kate" };
var result = people.Skip(2);    // "Bob", "Mike", "Kate"
var result = people.SkipLast(2);    // "Tom", "Sam", "Bob"
```

Метод SkipWhile() пропускає елементи, починаючи з першого, поки вони відповідають заданій вимозі, а решту повертає:

```
var result = people.SkipWhile(p => p.Length == 3);    // "Mike", "Kate", "Bob"
```

Метод Take() повертає задану кількість елементів, починаючи з першого:

```
var result = people.Take(3);    // "Tom", "Sam", "Mike"
```

Метод TakeLast()повертає задану кількість елементів, в кінці колекції:

```
var result = people.TakeLast(3);    // "Mike", "Kate", "Bob"
```

Метод TakeWhile() повертає елементи, починаючи з першого, поки вони відповідають заданій вимозі

```
var result = people.TakeWhile(p => p.Length == 3);    // "Tom", "Sam
```

Посторінковий вивід

За допомогою спільної роботи методів Take и Skip, ми можемо вибрати певну кількість елементів починаючи з певного. Наприклад

```
string[] people = { "Tom", "Sam", "Mike", "Kate", "Bob", "Alice" };

// пропускаємо 3 елемента і вибираємо 2

var result = people.Skip(3).Take(2);    // "Kate", "Bob"
```

Групування group by()

```
Person[] people =
{
   new Person("Tom", "Microsoft"), new Person("Sam", "Google"),
   new Person("Bob", "JetBrains"), new Person("Mike", "Microsoft"),
   new Person("Kate", "JetBrains"), new Person("Alice", "Microsoft"),
};
var companies = from person in people
            group person by person.Company;
record class Person(string Name, string Company);
```

або

```
var companies = people.GroupBy(p => p.Company);
```

або

```
var companies = from person in people
            group person by person.Company into g
            select new { Name = g.Key, Count = g.Count() };
```

Вкладені запити

```
var companies = from person in people
            group person by person.Company into g
            select new
            {
               Name = g.Key,
               Count = g.Count(),
               Employees = from p in g select p
            };
```

або

```
var companies = people
               .GroupBy(p => p.Company)
               .Select(g => new
                {
                   Name = g.Key,
                   Count = g.Count(),
                   Employees = g.Select(p => p)
                });
```

Об’єднання колекцій

З'єднання LINQ використовується для об'єднання двох різнотипних наборів в один. Для з'єднання використовується оператор join або метод Join(). Як правило, ця операція застосовується до двох наборів, які мають один загальний критерій.

```
from об’єкт1 in набір1
```

```
join об’єкт2 in набір2 on об’єкт2.властивість2 equals об’єкт1.властивість1

Person[] people =
{
   new Person("Tom", "Microsoft"), new Person("Sam", "Google"),
   new Person("Bob", "JetBrains"), new Person("Mike", "Microsoft"),
};
Company[] companies =
{
   new Company("Microsoft", "C#"),
   new Company("Google", "Go"),
   new Company("Oracle", "Java")
};
var employees = from p in people
            join c in companies on p.Company equals c.Title
            select new { Name = p.Name, Company = c.Title, Language = c.Language };
record class Person(string Name, string Company);
record class Company(string Title, string Language);
```

Метод Join() приймає чотири параметри:

- другий список, який з'єднуємо з поточним
- делегат, який визначає властивість об'єкта з поточного списку, яким йде з'єднання
- делегат, який визначає властивість об'єкта з другого списку, яким йде з'єднання
- делегат, який визначає новий об'єкт у результаті з'єднання

Перепишемо попередній приклад з використанням методу Join():

```
var employees = people.Join(companies, // набір2
            p => p.Company, // властивість-селектор об’єкту з набору 1
            c => c.Title, // властивість-селектор об’єкту з набору 2
            (p, c) => new { Name = p.Name, Company = c.Title, Language = c.Language }); // результат
```

Метод **GroupJoin()** крім з'єднання послідовностей також виконує групування.

```
GroupJoin(IEnumerable < TInner > inner,
     Func < TOuter, TKey > outerKeySelector,
     Func < TInner, TKey > innerKeySelector,
     Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
```

Метод GroupJoin() приймає чотири параметри:

- другий список, який з'єднуємо з поточним
- делегат, який визначає властивість об'єкта з поточної колекції, яким йде з'єднання і яким буде йти угруповання
- делегат, який визначає властивість об'єкта з другої колекції, яким йде з'єднання
- делегат, який визначає новий об'єкт у результаті з'єднання. Цей делегат отримує групу - об'єкт поточної колекції, яким йшло угруповання, і набір об'єктів з другої колекції, які складають групу

```
var personnel = companies.GroupJoin(people, // набір2
            c => c.Title, // властивість-селектор об’єкту з набору 1
            p => p.Company, // властивість-селектор об’єкту з набору 2
            (c, employees) => new   // результат
```

```
    {
        Title = c.Title,
        Employees = employees
    });
```

Метод Zip() послідовно поєднує відповідні елементи поточної послідовності з другою послідовністю, яка передається метод як параметр. Тобто перший елемент з першої послідовності поєднується з першим елементом з другої послідовності, другий елемент з першої послідовності з'єднується з другим елементом з другої послідовності і так далі. Результатом методу є колекція кортежів, де кожен кортеж зберігає пару відповідних елементів з обох послідовностей:

```
var courses = new List<Course> { new Course("C#"), new Course("Java") };
var students = new List<Student> { new Student("Tom"), new Student("Bob") };
var enrollments = courses.Zip(students);
record class Course(string Title);  // навчальний курс
record class Student(string Name);  // студент
```

Перевірка наявності та отримання елементів

Метод All() перевіряє, чи задовольняють всі елементи даної послідовності певну умову. Якщо так, то повертається true.

```
string[] people = { "Tom", "Tim", "Bob", "Sam" };
bool allHas3Chars = people.All(s => s.Length == 3);     // true
bool allStartsWithT = people.All(s => s.StartsWith("T"));   // false
```

Метод Any() схожий за поведінкою до попереднього, але повертає true, якщо принаймні один елемент задовольняє умову:

```
bool allHasMore3Chars = people.Any(s => s.Length > 3);     // false
bool allStartsWithT = people.Any(s => s.StartsWith("T"));   // true
```

Метод Contains() повертає true, якщо колекція містить певний елемент.

```
bool hasTom = people.Contains("Tom");     // true
bool hasMike = people.Contains("Mike");     // false
```

Слід відмітити, що для порівняння використовується реалізація методу Equals. Відповідно, якщо ми працюємо з власними класами, то можемо реалізувати натупне:

```
Person[] people = { new Person("Tom"), new Person("Sam"), new Person("Bob") };
var tom = new Person("Tom");
var mike = new Person("Mike");
bool hasTom = people.Contains(tom);     // true
bool hasMike = people.Contains(mike);     // false

class Person
{
    public string Name { get; }
    public Person(string name) => Name = name;
    public override bool Equals(object? obj)
    {
        if (obj is Person person) return Name == person.Name;
        return false;
    }
    public override int GetHashCode() => Name.GetHashCode();
```

```
}
```

Проте Contains не завжди може повернути очікувані дані:

```
string[] people = { "tom", "Tim", "bOb", "Sam" };
bool hasTom = people.Contains("Tom");     // false
bool hasMike = people.Contains("Bob");     // false
```

Така поведінка не завжди є доречною. В цьому випадку ми можемо реалізувати порівняння за допомогою IComparer, а потім передати цю реалізацію, в якості другого параметру, в метод Contains:

```
bool hasTom = people.Contains("Tom", new CustomStringComparer());     // true
bool hasMike = people.Contains("Bob", new CustomStringComparer());     // true
class CustomStringComparer : IEqualityComparer<string>
{
   public bool Equals(string? x, string? y)
   {
      if (x is null || y is null) return false;
      return x.ToLower() == y.ToLower();
   }
   public int GetHashCode(string obj) => obj.ToLower().GetHashCode();
}
```

First/FirstOrdefault

Метод First():

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };
var first = people.First();  // Tom
```

або

```
var firstWith4Chars = people.First(s => s.Length == 4);  // Kate
```

Якщо колекція порожня або немає елемента, який задовольняє умови, то буде згенеровано виняток.

```
var firstWith5Chars = people.First(s => s.Length == 5);  // ! виняток
var first = new string[] { }.First();  // ! виняток
```

Метод FirstOrDefault() працює аналогічно, але для випадків, коли мало б генеруватися виняток, даний метод повертатиме значення за замовчуванням:

```
var first = people.FirstOrDefault();  // Tom
var firstWith4Chars = people.FirstOrDefault(s => s.Length == 4);  // Kate
var firstOrDefault = new string[] { }.FirstOrDefault();// null
```

або

```
string? firstWith5Chars = people.FirstOrDefault(s => s.Length == 5, "Undefined"); // Undefined
string? firstOrDefault = new string[] { }.FirstOrDefault("hello"); // hello
int firstNumber = new int[] { }.FirstOrDefault(100); // 100
```

Last/LastOrDefault

Метод Last() працює аналогічно до First, але повертає останній елемент.

```
string last = people.Last(); // Sam
string lastWith4Chars = people.Last(s => s.Length == 4); // Mike
```

Метод LastOrDefault() працює аналогічно як FirstOrDefault() для останнього елемента

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };
string? last = people.LastOrDefault();// Sam
string? lastWith4Chars = people.LastOrDefault(s => s.Length == 4); // Mike
string? lastWith5Chars = people.LastOrDefault(s => s.Length == 5); // null
string? lastWith5CharsOrDefault = people.LastOrDefault(s => s.Length == 5, "Undefined");
// Undefined
string? lastOrDefault = new string[] { }.LastOrDefault("hello");  // hello
```

Відкладене та негайне виконання LINQ

Є два способи виконання запиту LINQ: відкладене(deferred) та негайне(immediate) виконання. При відкладеному виконанні LINQ-вираз не виконується, поки не буде проведено ітерацію або перебір за вибіркою, наприклад, у циклі foreach. Зазвичай подібні операції повертають об'єкт IEnumerable<T> або IOrderedEnumerable<T>.

Повний список відкладених операцій LINQ:

- AsEnumerable
- Cast
- Concat
- DefaultIfEmpty
- Distinct
- Except
- GroupBy
- GroupJoin
- Intersect
- Join
- OfType
- OrderBy
- OrderByDescending
- Range
- Repeat
- Reverse
- Select
- SelectMany
- Skip
- SkipWhile
- Take
- TakeWhile
- ThenBy
- ThenByDescending
- Union
- Where

Приклади

```
string[] people = { "Tom", "Sam", "Bob" };
var selectedPeople = people.Where(s => s.Length == 3).OrderBy(s => s);
// виконання LINQ-запиту
foreach (string s in selectedPeople)
   Console.WriteLine(s);
```

Фактичне виконання методу виконується не в точці оголошення: var selectedPeople = people.Where..., а при переборі в циклі foreach.

Фактично LINQ-запит розбивається на 3 етапи:

- Отримання джерела даних
- Створення запиту
- Виконання запиту та отримання результатів

В нашому випадку:

- Отримання джерела даних – оголошення масиву people;
- Створення запиту – оголошення змінної selectedPeople;
- Виконання запиту та отримання результатів – виконання foreach.

Після оголошення запиту він може використовуватись довільну кількість разів і до виконання запиту джерело даних може змінитись. Наприклад:

```
string[] people = { "Tom", "Sam", "Bob" };
var selectedPeople = people.Where(s => s.Length == 3).OrderBy(s => s);
people[2] = "Mike";
// виконання LINQ-запиту
foreach (string s in selectedPeople)
   Console.WriteLine(s);
```

Тепер вибірка міститиме два елементи, а не три, оскільки останній елемент не задовольнятиме умову.

Змінна запиту не виконує ніяких дій та не повертає результату. Вона тільки зберігає набір команд, які необхідні для отримання результату. Тобто виконання запиту відкладається після його створення. Отримання результату відбувається в циклі foreach.

Негайне виконання команд

З допомогою певних методів ми можемо використовувати методи з негайним виконанням. Це методи, які повертають одне атомарне значення або один елемент або дані типу Array, List и Dictionary. Повний перелік негайних методів в LINQ:

- Aggregate
- All
- Any
- Average
- Contains
- Count
- ElementAt
- ElementAtOrDefault
- Empty
- First
- FirstOrDefault

- Last
- LastOrDefault
- LongCount
- Max
- Min
- SequenceEqual
- Single
- SingleOrDefault
- Sum
- ToArray
- ToDictionary
- ToList
- ToLookup

Приклад:

```
string[] people = { "Tom", "Sam", "Bob" };
// оголошення та виконання LINQ-запиту
var count = people.Where(s => s.Length == 3).OrderBy(s => s).Count();
Console.WriteLine(count);   // 3 - до зміни
people[2] = "Mike";
Console.WriteLine(count);   // 3 – після зміни
```

Результатом методу Count буде об’єкт int, тому спрацює негайне виконання.

Створення запиту: people.Where(s => s.Length == 3).OrderBy(s => s).

Потім до нього застосовується метод Count(), який виконує запит, неявно перебирає елементи в колекції, яку генерує запит, та повертає число елементів, які задовольняють умову в цій послідовності.

Також ми можемо реалізувати метод Count() таким чином, щоб він враховував зміни та виконувався окремо від оголошення запиту:

```
// оголошення LINQ-запиту
var selectedPeople = people.Where(s => s.Length == 3).OrderBy(s => s);
// виконання запту
Console.WriteLine(selectedPeople.Count());   // 3 - до зміни
people[2] = "Mike";
// виконання запту
Console.WriteLine(selectedPeople.Count());   // 2 – після зміни
```

Також для негайного виконання запитів і кешуванні його результатів ми можемо застосувати методи перетворення ToArray<T>(), ToList<T>(), ToDictionary() і т.д. Ці методи отримують результат запитів у вигляді об’єктів Array, List та Dictionary відповідно. Наприклад:

```
// оголошення та виконання LINQ-запиту
var selectedPeople = people.Where(s => s.Length == 3).OrderBy(s => s).ToList();
// зміна масиву ніяк не торкається selectedPeople
people[2] = "Mike";
// виконання запту
foreach (string s in selectedPeople)
   Console.WriteLine(s);
```

Делегати в запитах LINQ

В оголошенні багатьох методів розширення LINQ, в якості аргументів, використовуються делегати (Func<TSource, bool>). Наприклад оголошення методу Where:

```
public static IEnumerable<TSource> Where<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, bool> predicate
)
```

У схожих методах, як правило, зручно передавати лямбда-вирази. Але ми можемо передавати і повноцінні методи. Наприклад:

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };
var result = people.Where(LenghtIs3);
bool LenghtIs3(string name) => name.Length == 3;
```

або

Select() застосовується в колекціях цілих чисел та перетворює число в його квадрат:

```
int[] numbers = { -2, -1, 0, 1, 2, 3, 4, 5, 6, 7 };
var result = numbers.Where(i => i > 0).Select(Square);
foreach (int i in result)
   Console.WriteLine(i);
int Square(int n) => n * n;
```

Метод Select приймає параметр типу Func<TSource, TResult> selector. Оскільки в нас колекція int, то вхідним параметром делегату буде об’єкт ипу int. В якості типу вихідного параметру буде int, бо квадрат int також буде int.

Заняття 15: Введення в MS SQL Server та T-SQL. DDL. DML

15.1 Загальні дані

SQL Server підходить для різних проектів: від невеликих додатків до великих високонавантажених проектів. SQL Server було створено компанією Microsoft. SQL Server довгий час був виключно системою керування базами даних для Windows, проте починаючи з версії 16 ця система доступна і на Linux. SQL Server характеризується такими особливостями як:

- Продуктивність. SQL Server працює дуже швидко.
- Надійність та безпека. SQL Server надає шифрування даних.
- Простота. З цієї СУБД доволі легко працювати та вести адміністрування.
- Центральним аспектом у MS SQL Server, як і будь-який СУБД, є база даних. База даних представляє сховище даних, організованих певним способом.

Нерідко фізично база даних представляє файл на жорсткому диску, хоча така відповідність необов'язково. Для зберігання та адміністрування баз даних застосовуються системи управління базами даних (database management system) або СУБД (DBMS). І саме **MS SQL Server** є однією з таких СУБД. Для організації баз даних MS SQL Server використовує реляційну модель.

Реляційна модель передбачає зберігання даних у вигляді таблиць, кожна з яких складається з рядків та стовпців. Кожен рядок зберігає окремий об'єкт, а стовпці розміщуються атрибути цього об'єкта. Для ідентифікації кожного рядка у межах таблиці застосовується первинний ключ (**primary key**). Як первинний ключ може виступати один або кілька стовпців. Використовуючи первинний ключ, ми можемо посилатися на певний рядок у таблиці. Відповідно два рядки не можуть мати один і той же первинний ключ.

Через ключі одна таблиця може бути пов'язана з іншою, тобто між двома таблицями можуть бути організовані зв'язки. А сама таблиця може бути представлена у вигляді відношення (**"relation"**). Для взаємодії з базою даних використовується мова SQL (**Structured Query Language**).

Клієнт (наприклад, зовнішня програма) надсилає запит мовою SQL за допомогою спеціального API. СУБД належним чином інтерпретує та виконує запит, а потім надсилає клієнту результат виконання.

Виділяються два різновиди мови SQL: **PL-SQL та T-SQL**. PL-SQL використовується у таких СУБД як Oracle та MySQL. T-SQL (Transact-SQL) застосовується у SQL Server. Саме тому в рамках поточного керівництва розглядатиметься саме T-SQL.

Залежно від завдання, яке виконує команда T-SQL, він може належати до одного з таких типів:

DDL (Data Definition Language/Мова визначення даних). До цього типу належать різні команди, які створюють базу даних, таблиці, індекси, процедури, що зберігаються і т.д. Загалом визначають дані. Зокрема, до цього типу ми можемо віднести такі команди:

- CREATE : створює об'єкти бази даних (саму базу даних, таблиці, індекси тощо)

- ALTER : змінює об'єкти бази даних
- DROP : видаляє об'єкти бази даних
- TRUNCATE : видаляє всі дані з таблиць

DML (Data Manipulation Language/Мова маніпуляції даними). До цього типу відносять команди з вибору даних, їх оновлення, додавання, видалення - загалом усі команди, з допомогою якими ми можемо управляти даними. До цього типу належать такі команди:

- SELECT : отримує дані з БД
- UPDATE : оновлює дані
- INSERT : додає нові дані
- DELETE : видаляє дані

DCL (Data Control Language/Мова керування доступу до даних). До цього типу відносять команди, які керують правами доступу до даних. Зокрема, це такі команди:

- GRANT : надає права на доступ до даних
- REVOKE : відкликає права на доступ до даних

Базу даних часто ототожнюють із набором таблиць, які зберігають дані. Але це зовсім так. Краще сказати, що база даних представляє сховище об'єктів. Основні з них:

Tables	Таблиці бази даних, в яких зберігаються власне дані
Views	Перегляди (віртуальні таблиці) для відображення даних, відібраних з таблиць
Stored Procedures	Збережені процедури - виконують код на мові SQL по відношенню до даних БД (наприклад, отримує дані або змінює їх)
Triggers	Трігери – спеціальні збережені процедури, що викликаються при зміні даних в таблиці
User Defined function	Створювані користувачем функції, які виконують певне завдання
Indexes	Індекси – додаткові структури, покликані підвищити продуктивність роботи з даними
User Defined Data Types	Визначувані користувачем типи даних
Keys	Ключі – один з видів обмежень цілісності даних
Constraints	Обмеження цілісності – об’єкти для забезпечення логічної цілісності даних
Users	Користувачі, що мають доступ до бази даних
Roles	Ролі, що дозволяють об’єднувати користувачів в групи
Rules	Правила бази даних, що дозволяють контролювати логічну цілісність даних
Defaults	Умовчання або стандартні установки бази даних

У SQL Server використовується два типи баз даних: **системні** та **користувацькі**. Системні бази даних потрібні серверу SQL для коректної роботи. А бази даних користувача

створюються користувачами сервера і можуть зберігати будь-яку довільну інформацію. Їх можна змінювати та видаляти, створювати заново. Власне це бази даних, які ми будемо створювати і з якими ми працюватимемо.

Системні бази даних. У MS SQL Server за умовчанням створюється чотири системні бази даних:

- master : ця головна база даних сервера, у разі відсутності або пошкодження сервер не зможе працювати. Вона зберігає всі логіни користувачів сервера, їх ролі, різні конфігураційні налаштування, імена та інформацію про бази даних, які зберігаються на сервері, а також ряд іншої інформації.
- model : ця база даних представляє шаблон, основі якого створюються інші бази даних. Тобто, коли ми створюємо через SSMS свою бд, вона створюється як копія бази model.
- msdb : зберігає інформацію про роботу, яку виконує такий компонент як планувальник SQL. Також вона зберігає інформацію про бекапи баз даних.
- tempdb : Ця база даних використовується як сховище для тимчасових об'єктів. Вона знову перестворюється при кожному запуску сервера.

Всі ці бази можна побачити через SQL Server Management Studio у вузлі Databases -> System Databases. Ці бази даних не слід змінювати, крім бд model. Якщо на етапі встановлення сервера було обрано та встановлено компонент PolyBase, то також на сервері за умовчанням будуть розташовані ще три бази даних, які використовуються цим компонентом: **DWConfiguration, DWDiagnostics, DWQueue.**

15.2 Робота через SQL Server Management Studio

Створення бази даних користувача. Для цього ми можемо використовувати скрипт на мові SQL або все зробити за допомогою графічних засобів в SQL Management Studio.

За допомогою графічних засобів: відкриємо SQL Server Management Studio і натиснемо правою кнопкою миші на вузол Databases -> У контекстному меню виберемо пункт New Database ->Відкривається вікно для створення бази даних -> У полі Database необхідно запровадити назву нової бд. -> Наступне поле Owner задає власника бази даних. За умовчанням воно має значення <defult> , тобто власником буде той, хто створює цю базу даних. Залишимо це поле без змін. -> Далі йде таблиця для встановлення загальних налаштувань бази даних. Вона містить два рядки - перший для установки налаштувань для головного файлу, де зберігатимуться дані, і другий рядок для конфігурації файлу логування. Зокрема, ми можемо встановити такі налаштування: **Logical Name** : логічне ім'я, яке присвоюється файлу бази даних. **File Type** : є кілька типів файлів, але, як правило, основна робота ведеться з файлами даних (ROWS Data) та файлом лога (LOG). **Filegroup** : позначає групу файлів. Група файлів може зберігати безліч файлів і може використовуватися для розбиття бази даних на частини для розміщення у різних місцях. **Initial Size** (MB) : встановлює початковий розмір файлів під час створення (фактичний розмір може відрізнятися від цього значення). **Autogrowth/Maxsize** : при досягненні бази даних початкового розміру SQL Server використовує це значення для збільшення файлу. **Path** : каталог, де зберігатимуться бази даних. **File Name** : безпосереднє ім'я фізичного файлу. Якщо вона не зазначена, то застосовується логічне ім'я. -> Після введення назви бази даних натиснемо кнопку ОК, і бд буде створена.

Після цього бд з'явиться серед баз даних сервера. Якщо ця бд згодом не буде потрібна, то її можна видалити, натиснувши на неї правою кнопкою миші та вибравши в контекстному меню пункт Delete

Ключовим об'єктом у базі даних є **таблиці**. Таблиці складаються з **рядків** та **стовпців**. Стовпці визначають тип інформації, що зберігається, а рядки містять значення цих стовпців. Для створення таблиці в SQL Server Management Studio можна застосувати скрипт мовою SQL, або скористатися графічним дизайнером. Для створення таблиці в SQL Server Management Studio розкриємо вузол бази даних в SQL Server Management Studio, натиснемо на його подузол Tables правою кнопкою миші і далі в контексті меню виберемо New -> Table... -> Відкриється дизайнер таблиці. У центральній частині таблиці необхідно ввести дані про стовпці таблиці. Дизайнер містить три поля: **Column Name** - ім'я стовпця. **Data Type** - тип даних стовпця. Тип даних визначає, які дані можуть зберігатися у цьому стовпці. Наприклад, якщо стовпець представляє числовий тип, він може зберігати лише числа. **Allow Nulls -** чи може бути відсутнє значення у стовпця, тобто чи може він бути порожнім -> У вікні Properties, яка містить властивості таблиці, в полі Name треба ввести ім'я таблиці, а в полі Identity ввести ім’я ідентифікатора (Id), тобто вказуючи, що стовпець Id буде ідентифікатором. Ім'я таблиці має бути унікальним у межах бази даних. Як правило, назва таблиці відображає назву сутності, яка зберігається в ній. Наприклад, ми хочемо зберегти студентів, тому таблиця називається Students (слово студент у множині англійською мовою). Існують різні думки з приводу того, чи варто використовувати назву сутності в однині або множині (Student або Students). Питання найменування таблиці повністю лягає на розробника бази даних. -> Слід зазначити, що стовпець Id виконуватиме роль первинного ключа (**primary key**). Первинний ключ унікально ідентифікує кожного рядка. У ролі первинного ключа може бути один стовпець, а може й кілька. Для встановлення первинного ключа натиснемо на відповідний стовпець (Id) правою кнопкою миші і в меню виберемо пункт Set Primary Key. Після цього навпроти поля Id має з'явитися золотий ключик. Цей ключик вказуватиме, що стовпець Id виконуватиме роль первинного ключа. -> Після збереження у базі даних з'явиться відповідна таблиця.

Ми можемо помітити, що назва таблиці фактично починається з префікса dbo . Цей префікс представляє схему. Схема визначає контейнер, що зберігає об'єкти. Тобто схема логічно розмежовує бази даних. Якщо схема явним чином не вказується при створенні об'єкта, то схема за умовчанням буде схема dbo.

Основні дії для роботи з таблицею в SQL Server Management Studio

- Delete дозволяє вилучити таблицю.
- Design відкриє вікно дизайнера таблиці, де ми можемо за потреби внести зміни до її структури.
- Edit Top 200 Rows додавання даних до таблиці. Вона відкриває у вигляді таблиці 200 перших рядків та дозволяє їх змінити.
- Select Top 1000 Rows і буде запущено скрипт, який відобразить перші 1000 рядків з таблиці

15.3 Типи даних

Під час створення таблиці всім її стовпців необхідно вказати певний тип даних. Тип даних визначає, які значення можуть зберігатися в стовпці, скільки вони займатимуть місця у

пам'яті. Мова T-SQL надає багато різних типів. Залежно від характеру значень їх можна розділити на групи.

Числові типи даних

- BIT : зберігає значення від 0 до 16. Може виступати аналогом булевого типу у мовах програмування (у разі значення true відповідає 1, а значення false - 0). При значеннях до 8 (включно) займає 1 байт, при значеннях від 9 до 16 – 2 байти.
- TINYINT : зберігає числа від 0 до 255. Займає 1 байт. Добре підходить для зберігання невеликих чисел.
- SMALLINT : зберігає числа від -32 768 до 32 767. Займає 2 байти
- INT : зберігає числа від -2147483648 до 2147483647. Займає 4 байти. Найбільш використовуваний тип зберігання чисел.
- BIGINT : зберігає дуже великі числа від -9223372036854775808 до 9223372036854775807, які займають у пам'яті 8 байт.
- DECIMAL : зберігає числа з фіксованою точністю. Займає від 5 до 17 байт залежно від кількості чисел після коми. Цей тип може приймати два параметри precision і scale: DECIMAL(precision, scale). Параметр precision представляє максимальну кількість цифр, які можуть зберігати число. Це значення має знаходитися в діапазоні від 1 до 38. За замовчуванням воно дорівнює 18. Параметр scale представляє максимальну кількість цифр, які можуть містити число після коми. Це значення має знаходитися в діапазоні від 0 до значення параметра precision. За умовчанням воно дорівнює 0.
- NUMERIC : даний тип аналогічний типу DECIMAL.
- SMALLMONEY : зберігає дробові значення від -214748.3648 до 214748.3647. Призначений для зберігання грошових величин. Займає 4 байти. Еквівалентний типу DECIMAL(10,4).
- MONEY : зберігає дробові значення від -922337203685477.5808 до 922337203685477.5807. Представляє грошові величини та займає 8 байт. Еквівалентний типу DECIMAL(19,4).
- FLOAT : зберігає цифри від –1.79E+308 до 1.79E+308. Займає від 4 до 8 байт залежно від дрібної частини. Може мати форму визначення у вигляді FLOAT(n), де n є число біт, які використовуються для зберігання десяткової частини числа (мантиси). По замовчанні n = 53.
- REAL : зберігає цифри від –340E+38 to 3.40E+38. Займає 4 байти. Еквівалентний типу FLOAT(24).

Типи даних, що представляють дату та час

- DATE : зберігає дати від 0001-01-01 (1 січня 0001 року) до 9999-12-31 (31 грудня 9999 року). Займає 3 байти.
- TIME : зберігає час у діапазоні від 00:00:00.0000000 до 23:59:59.9999999. Займає від 3 до 5 байт. Може мати форму TIME(n), де n представляє кількість цифр від 0 до 7 для дробової частини секунд.
- DATETIME : зберігає дати та час від 01/01/1753 до 31/12/9999. Займає 8 байт.
- DATETIME2 : зберігає дати та час у діапазоні від 01/01/0001 00:00:00.0000000 до 31/12/9999 23:59:59.9999999. Займає від 6 до 8 байт, залежно від точності часу. Може мати форму DATETIME2(n), де n представляє кількість цифр від 0 до 7 для дробової частини секунд.

- SMALLDATETIME : зберігає дати та час у діапазоні від 01/01/1900 до 06/06/2079, тобто найближчі дати. Займає від 4 байти.
- DATETIMEOFFSET : зберігає дати та час у діапазоні від 0001-01-01 до 9999-12-31. Зберігає детальну інформацію про час з точністю до 100 наносекунд. Займає 10 байт.

Поширені формати дат:

- yyyy-mm-dd - 2017-07-12
- dd/mm/yyyy - 12/07/2017
- mm-dd-yy - 07-12-17 У такому форматі двоцифрові числа від 00 до 49 сприймаються як дати в діапазоні 2000-2049. А числа від 50 до 99 як діапазон чисел 1950 – 1999.
- mm dd, yyyy - July 12, 2017

Розповсюджені формати часу:

- hh:mi - 13:21
- hh:mi am/pm - 1:21 pm
- hh:mi:ss - 1:21:34
- hh:mi:ss:mmm - 1:21:34:12
- hh:mi:ss:nnnnnnn - 1:21:34:1234567

Рядкові типи даних

- CHAR : зберігає рядок довжиною від 1 до 8000 символів. На кожен символ виділяє по 1 байти. Не підходить для багатьох мов, оскільки зберігає символи не в кодуванні Unicode. Кількість символів, які можуть зберігати стовпець, передається в дужках. Наприклад, для стовпця з типом CHAR(10)буде виділено 10 байт. І якщо ми збережемо в стовпці рядок менше 10 символів, то він буде доповнений пробілами.
- VARCHAR : зберігає рядок. На кожен символ виділяється 1 байт. Можна вказати конкретну довжину стовпця - від 1 до 8 000 символів, наприклад, VARCHAR(10). Якщо рядок повинен мати більше 8000 символів, то визначається розмір MAX, а на зберігання рядка може виділятися до 2 Гб: VARCHAR(MAX). Не підходить для багатьох мов, оскільки зберігає символи не в кодуванні Unicode. На відміну від типу CHAR якщо в стовпець з типом VARCHAR(10)буде збережено рядок 5 символів, то в столці буде збережено саме п'ять символів.
- NCHAR : зберігає рядок у кодуванні Unicode довжиною від 1 до 4000 символів. На кожен символ виділяється 2 байти. Наприклад,NCHAR(15)
- NVARCHAR : зберігає рядок у кодуванні Unicode. На кожен символ виділяється 2 байти. Можна задати конкретний розмір від 1 до 4 000 символів: . Якщо рядок повинен мати більше 4000 символів, то визначається розмір MAX, а на зберігання рядка може виділятися до 2 Гб.
- TEXT та NTEXT є застарілими і тому їх не рекомендується використовувати. Замість них застосовуються VARCHAR та NVARCHAR відповідно.

Бінарні типи даних

- BINARY : зберігає бінарні дані у вигляді послідовності від 1 до 8000 байт.
- VARBINARY : зберігає бінарні дані як послідовності від 1 до 8 000 байт, або до 2^31–1 байт під час використання значення MAX (VARBINARY(MAX)).

- IMAGE є застарілим, і замість нього рекомендується застосовувати тип VARBINARY.

Інші типи даних

- UNIQUEIDENTIFIER : унікальний ідентифікатор GUID (насправді рядок з унікальним значенням), який займає 16 байт.
- TIMESTAMP : деяке число, яке зберігає номер версії рядка таблиці. Займає 8 байт.
- CURSOR : представляє набір рядків.
- HIERARCHYID : представляє позицію в ієрархії.
- SQL_VARIANT : може зберігати дані будь-якого іншого типу даних T-SQL.
- XML : зберігає документи XML або фрагменти документів XML. Займає у пам'яті до 2 Гб.
- TABLE : подає визначення таблиці.
- GEOGRAPHY : зберігає географічні дані, такі як широта та довгота.
- GEOMETRY : зберігає координати місцезнаходження на площині.

15.4 Робота зі скриптами

create database InternetShop -- створення бази даних під назвою InternetShop

go -- роздільник пакетів. Пакет - набір команд, які виконуються в рамках запиту.

use InternetShop -- встановлюємо з'єднання із створеною базою.

Можлива ситуація, що ми вже маємо файл бази даних, який, наприклад, створено на іншому комп'ютері. Файл бази даних представляє файл з розширенням **mdf**, і цей файл ми можемо переносити. Однак навіть якщо ми скопіюємо його на комп'ютер із встановленим MS SQL Server, просто так скопійована база даних на сервері не з'явиться. Для цього необхідно виконати прикріплення бази даних до сервера. У цьому випадку застосовується вираз

```
CREATE DATABASE название_базы_данных
ON PRIMARY(FILENAME='путь_к_файлу_mdf_на_локальном_компьютере')
FOR ATTACH;
```

drop database InternetShop - видалення бази даних

Створення таблиці

```
CREATE TABLE Customers
(
   Id INT,
   Age INT,
   FirstName NVARCHAR(20)
);
```

Перейменування таблиці

```
EXEC sp_rename 'Customers', 'Clients';
```

Видалення таблиці

```
DROP TABLE Customers
```

Обмеження (Constraint)

Атрибут IDENTITY зробить стовпець ідентифікатором. Цей атрибут може бути призначений для стовпців числових типів INT, SMALLINT, BIGINT, TYNIINT, DECIMAL і NUMERIC. При додаванні нових даних до таблиці SQL Server інкрементуватиме на одиницю значення цього стовпця в останньому записі. Як правило, у ролі ідентифікатора виступає той самий стовпець, який є первинним ключем, хоча в принципі це необов'язково.

UNIQUE – атрибут, який слідкує щоб в стовпці не було повторень.

NULL/ NOT NULL – атрибути, які дозволяють/забороняють записувати null у стовпець. За замовчанням стовпець допускатиме значення NULL. Коли стовпець виступає як первинний ключ - за замовчанням стовпець має значення NOT NULL.

Атрибут DEFAULT визначає значення за промовчанням для стовпця. Якщо при додаванні даних для стовпця не буде передано значення, то для нього використовуватиметься значення за замовчуванням.

Ключове слово CHECK визначає обмеження для діапазону значень, які можуть зберігатися в стовпці. Для цього після слова CHECK вказується в дужках умова, якій має відповідати стовпець або кілька стовпців.

```
CREATE TABLE Customers
(
   Id INT PRIMARY KEY IDENTITY (100, 1), -- IDENTITY(seed, increment)
   Age INT NULL DEFAULT 18 CHECK(Age > 0 AND Age < 100),
   FirstName NVARCHAR(20) NOT NULL,
   Phone VARCHAR(20) UNIQUE
)

CREATE TABLE Customers
(
   Id INT PRIMARY KEY IDENTITY (1, 1),
   Age INT,
   FirstName NVARCHAR(20),
   Phone VARCHAR(20),
   UNIQUE(Phone)
)

CREATE TABLE Customers
(
   Id INT PRIMARY KEY IDENTITY,
   Age INT DEFAULT 18,
   FirstName NVARCHAR(20) NOT NULL,
   Phone VARCHAR(20) UNIQUE,
   CHECK((Age >0 AND Age<100) AND (Phone !=''))
)
```

За допомогою ключового слова CONSTRAINT можна встановити ім'я для обмежень. Як обмеження можуть використовуватися PRIMARY KEY, UNIQUE, DEFAULT, CHECK. Імена обмежень можна встановити на рівні стовпців. Вони вказуються після CONSTRAINT перед атрибутами:

```
CREATE TABLE Customers
(
   Id INT CONSTRAINT PK_Customer_Id PRIMARY KEY IDENTITY,
   Age INT
      CONSTRAINT DF_Customer_Age DEFAULT 18
      CONSTRAINT CK_Customer_Age CHECK(Age >0 AND Age < 100),
   FirstName NVARCHAR(20) NOT NULL,
   Phone VARCHAR(20) CONSTRAINT UQ_Customer_Phone UNIQUE
)
```

Обмеження можуть носити довільні назви, але, як правило, застосовуються такі префікси:

- "PK_" - для PRIMARY KEY
- "FK_" - для FOREIGN KEY
- "CK_" - для CHECK
- "UQ_" - для UNIQUE
- "DF_" - для DEFAULT

В принципі необов'язково вказувати імена обмежень, при установці відповідних атрибутів SQL Server автоматично визначає їх імена. Але знаючи ім'я обмеження, ми можемо до нього звертатися, наприклад, для його видалення.

Найкращий варіант:

```
CREATE TABLE Customers
(
   Id INT IDENTITY,
   Age INT CONSTRAINT DF_Customer_Age DEFAULT 18,
   FirstName NVARCHAR(20) NOT NULL,
   Phone VARCHAR(20),
   CONSTRAINT PK_Customer_Id PRIMARY KEY (Id),
   CONSTRAINT CK_Customer_Age CHECK(Age >0 AND Age < 100),
   CONSTRAINT UQ_Customer_Email UNIQUE (Email),
   CONSTRAINT UQ_Customer_Phone UNIQUE (Phone)
)
```

Зовнішні ключі використовуються для встановлення зв'язку між таблицями. Зовнішній ключ встановлюється для стовпців із залежної, підлеглої таблиці, і вказує на один із стовпців із головної таблиці. Хоча, як правило, зовнішній ключ вказує на первинний ключ із пов'язаної головної таблиці, але це не є неодмінною умовою. Зовнішній ключ також може вказувати на інший стовпець, який має унікальне значення.

```
[FOREIGN KEY] REFERENCES главная_таблица (столбец_главной_таблицы)
   [ON DELETE {CASCADE|NO ACTION}]
   [ON UPDATE {CASCADE|NO ACTION}]
```

Для створення обмеження зовнішнього ключа на рівні стовпця після ключового слова REFERENCES вказується ім'я пов'язаної таблиці та у круглих дужках ім'я зв'язаного стовпця, на який вказуватиме зовнішній ключ. Також зазвичай додаються ключові слова FOREIGN KEY , але їх необов'язково вказувати. Після виразу REFERENCES йде вираз ON DELETE та

ON UPDATE, які визначають, що має відбутись з пов’язаною таблицею у разі видалення/оновлення зв’язного стовбця головної.

```
CREATE TABLE Customers
(
   Id INT PRIMARY KEY IDENTITY,
   Age INT DEFAULT 18,
   FirstName NVARCHAR(20) NOT NULL,
   Phone VARCHAR(20) UNIQUE
);
CREATE TABLE Orders
(
   CustomerId INT REFERENCES Customers (Id)
);

CREATE TABLE Orders
(
   CustomerId INT,
   FOREIGN KEY (CustomerId)  REFERENCES Customers (Id)
);
```

Найкращий варіант:

```
CREATE TABLE Orders
(
   CustomerId INT,
   CONSTRAINT FK_Orders_To_Customers FOREIGN KEY (CustomerId) REFERENCES Customers (Id)
);
```

Опції для ON DELETE та ON UPDATE:

CASCADE : автоматично видаляє або змінює рядки із залежної таблиці під час видалення або зміни пов'язаних рядків у головній таблиці.

```
CREATE TABLE Orders
(
   Id INT PRIMARY KEY IDENTITY,
   CustomerId INT,
   FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE CASCADE
)
```

NO ACTION : запобігає будь-яким діям у залежній таблиці при видаленні або зміні зв'язаних рядків у головній таблиці. Тобто фактично якихось дій відсутні (на практиці не юзається).

```
CREATE TABLE Orders
(
   CustomerId INT,
   FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE NO ACTION
);
```

SET NULL : при видаленні зв'язаного рядка з головної таблиці встановлює значення NULL для стовпця зовнішнього ключа (використовується найчастіше).

```
CREATE TABLE Orders
(
   CustomerId INT,
   FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET NULL
);
```

SET DEFAULT : при видаленні зв'язаного рядка з головної таблиці встановлює для стовпця зовнішнього ключа значення за замовчанням, яке задається за допомогою атрибута DEFAULT. Якщо для стовпця не встановлено значення за замовчанням, то застосовується значення NULL.

```
CREATE TABLE Orders
(
   CustomerId INT,
   FOREIGN KEY (CustomerId) REFERENCES Customers (Id) ON DELETE SET DEFAULT
)
```

Можливо, в якийсь момент ми захочемо змінити таблицю, що вже є.

Наприклад, додати або видалити стовпці, змінити тип стовпців, додати або видалити обмеження.

Тобто потрібно змінити визначення таблиці. Для зміни таблиць використовується вираз ALTER TABLE .

```
ALTER TABLE название_таблицы [WITH CHECK | WITH NOCHECK]

{ ADD название_столбца тип_данных_столбца [атрибуты_столбца] |

 DROP COLUMN название_столбца |

 ALTER COLUMN название_столбца тип_данных_столбца [NULL|NOT NULL] |

 ADD [CONSTRAINT] определение_ограничения |

 DROP [CONSTRAINT] имя_ограничения}
```

Додавання нового стовпця

```
ALTER TABLE Customers

ADD Address NVARCHAR(50) NOT NULL DEFAULT 'No Info';
```

Видалення стовпця

```
ALTER TABLE Customers

DROP COLUMN Address;
```

Зміна типу стовпця

```
ALTER TABLE Customers

ALTER COLUMN FirstName NVARCHAR(200);
```

Додавання обмеження CHECK

```
ALTER TABLE Customers

ADD CHECK (Age > 21);
```

Якщо в таблиці є рядки, в яких у стовпці Age є значення, що не відповідають цьому обмеженню, то команда SQL завершиться з помилкою.

Щоб уникнути подібної перевірки на відповідність і додати обмеження, незважаючи на наявність невідповідних йому даних,

використовується вираз WITH NOCHECK

```
ALTER TABLE Customers WITH NOCHECK

ADD CHECK (Age > 21);
```

За замовчуванням використовується значення WITH CHECK , яке перевіряє відповідність обмеженням.

Додавання зовнішнього ключа

Нехай спочатку до бази даних буде додано дві таблиці, ніяк не пов'язані:

```
CREATE TABLE Customers
(
    Id INT PRIMARY KEY IDENTITY,
    Age INT DEFAULT 18,
    FirstName NVARCHAR(20) NOT NULL,
    LastName NVARCHAR(20) NOT NULL,
    Email VARCHAR(30) UNIQUE,
    Phone VARCHAR(20) UNIQUE
);
CREATE TABLE Orders
(
    Id INT IDENTITY,
    CustomerId INT,
    CreatedAt Date
);
```

Додамо обмеження зовнішнього ключа до стовпця CustomerId таблиці Orders:

```
ALTER TABLE Orders
ADD FOREIGN KEY(CustomerId) REFERENCES Customers(Id);
```

Додавання первинного ключа

```
ALTER TABLE Orders
ADD PRIMARY KEY (Id);
```

Додавання обмежень з іменами

```
ALTER TABLE Orders
ADD CONSTRAINT PK_Orders_Id PRIMARY KEY (Id),
```

CONSTRAINT FK_Orders_To_Customers FOREIGN KEY(CustomerId) REFERENCES Customers(Id);

ALTER TABLE Customers

ADD CONSTRAINT CK_Age_Greater_Than_Zero CHECK (Age > 0);

Видалення обмежень

ALTER TABLE Orders

DROP FK_Orders_To_Customers;

Основи T-SQL. DML

Для додавання даних застосовується команда INSERT , яка має такий формальний синтаксис:

INSERT [INTO] имя_таблицы [(список_столбцов)] VALUES (значение1, значение2, ... значениеN)

```
CREATE TABLE Products
(
   Id INT IDENTITY PRIMARY KEY,
   ProductName NVARCHAR(30) NOT NULL,
   Manufacturer NVARCHAR(20) NOT NULL,
   ProductCount INT DEFAULT 0,
   Price MONEY NOT NULL
)

INSERT Products VALUES ('iPhone 7', 'Apple', 5, 52000)
```

```
INSERT INTO Products
VALUES
('iPhone 6', 'Apple', 3, 36000),
('Galaxy S8', 'Samsung', 2, 46000),
('Galaxy S8 Plus', 'Samsung', 1, 56000)

INSERT INTO Products (ProductName, Manufacturer, ProductCount, Price)
VALUES ('Mi6', 'Xiaomi', DEFAULT, 28000)
```

Якщо всі стовпці мають атрибут DEFAULT, який визначає значення за замовчуванням, або допускають значення NULL,

то для всіх стовпців можна вставити значення за замовчуванням

```
INSERT INTO Products
DEFAULT VALUES
```

Для отримання даних застосовується команда SELECT . У спрощеному вигляді вона має наступний синтаксис:

```
SELECT список_столбцов FROM имя_таблицы
```

Наприклад, нехай раніше була створена таблиця Products, і до неї додані деякі початкові дані

```
CREATE TABLE Products
(
   Id INT IDENTITY PRIMARY KEY,
   ProductName NVARCHAR(30) NOT NULL,
   Manufacturer NVARCHAR(20) NOT NULL,
```

```
    ProductCount INT DEFAULT 0,
    Price MONEY NOT NULL
);

INSERT INTO Products
VALUES
('iPhone 6', 'Apple', 3, 36000),
('iPhone 6S', 'Apple', 2, 41000),
('iPhone 7', 'Apple', 5, 52000),
('Galaxy S8', 'Samsung', 2, 46000),
('Galaxy S8 Plus', 'Samsung', 1, 56000),
('Mi6', 'Xiaomi', 5, 28000),
('OnePlus 5', 'OnePlus', 6, 38000)
```

Отримаємо всі об'єкти з цієї таблиці:

```
SELECT * FROM Products
```

Якщо нам треба отримати дані не по всіх, а по якихось конкретних стовпцях, то всі ці специфікації стовпців перераховуються через кому після SELECT

```
SELECT ProductName, Price FROM Products
```

Специфікація стовпця необов'язково має представляти його назву.

Це може бути будь-який вираз, наприклад, результат арифметичної операції.

```
SELECT ProductName + ' (' + Manufacturer + ')', Price, Price * ProductCount
FROM Products
```

За допомогою оператора AS можна змінити назву вихідного стовпця або визначити його псевдонім:

```
SELECT
ProductName + ' (' + Manufacturer + ')' AS ModelName,
Price,
Price * ProductCount AS TotalSum
FROM Products
```

Оператор DISTINCT дозволяє вибрати унікальні рядки.

Наприклад, у нашому випадку в таблиці може бути по кілька товарів від тих самих виробників. Виберемо всіх виробників

```
SELECT DISTINCT Manufacturer
FROM Products
```

Вираз SELECT INTO дозволяє вибрати з однієї таблиці деякі дані до іншої таблиці, при цьому друга таблиця створюється автоматично.

```
SELECT ProductName + ' (' + Manufacturer + ')' AS ModelName, Price
INTO ProductSummary
FROM Products
```

```
SELECT * FROM ProductSummary
```

Після виконання цієї команди у базі даних буде створено ще одну таблицю ProductSummary, яка матиме два стовпці ModelName і Price,

а дані для цих стовпців будуть взяті з таблиці Products:

Оператор ORDER BY дозволяє відсортувати видобуті значення за певним стовпцем

```
SELECT *
```

```
FROM Products

ORDER BY ProductName
```

Сортування також можна проводити за псевдонімом стовпця, який визначається за допомогою оператора AS

```
SELECT ProductName, ProductCount * Price AS TotalSum

FROM Products

ORDER BY TotalSum
```

За умовчанням застосовується сортування за зростанням (ASC). За допомогою додаткового оператора DESC можна задати сортування за спаданням.

```
SELECT ProductName

FROM Products

ORDER BY ProductName DESC
```

```
SELECT ProductName

FROM Products

ORDER BY ProductName ASC
```

```
SELECT ProductName, Price, Manufacturer

FROM Products

ORDER BY Manufacturer, ProductName
```

```
SELECT ProductName, Price, Manufacturer

FROM Products
```

ORDER BY Manufacturer ASC, ProductName DESC

SELECT ProductName, Price, ProductCount

FROM Products

ORDER BY ProductCount * Price

Оператор TOP дозволяє вибрати певну кількість рядків із таблиці

SELECT TOP 4 ProductName

FROM Products

Додатковий оператор PERCENT дозволяє вибрати відсоткову кількість рядків із таблиці. Наприклад, виберемо 75% рядків:

SELECT TOP 75 PERCENT ProductName

FROM Products

Оператор TOP дозволяє витягти певну кількість рядків, починаючи з початку таблиці.

Для вилучення набору рядків з будь-якого місця застосовуються оператори OFFSET і FETCH .

Важливо, що ці оператори застосовуються лише у відсортованому наборі даних після виразу ORDER BY.

Наприклад, виберемо всі рядки, починаючи з третього:

SELECT * FROM Products

ORDER BY Id

OFFSET 2 ROWS;

Тепер виберемо лише три рядки, починаючи з третього:

```
SELECT * FROM Products
ORDER BY Id
   OFFSET 2 ROWS
   FETCH NEXT 3 ROWS ONLY;
```

Після оператора FETCH вказується ключове слово FIRST або NEXT (яке саме в даному випадку не має значення)

і потім вказується кількість рядків, які треба отримати.

Ця комбінація операторів зазвичай використовується для посторінкової навігації, коли необхідно отримати певну сторінку з даними.

Для фільтрації в команді SELECT застосовується оператор WHERE . Після цього оператора ставиться умова, якій має відповідати рядок

```
WHERE умова
```

Якщо умова є істинною, то рядок потрапляє в результуючу вибірку. Як можна використовувати операції порівняння.

Ці операції порівнюють два вирази. У T-SQL можна застосовувати такі операції порівняння:

= : порівняння на рівність (на відміну від сі-подібних мов у T-SQL для порівняння на рівність використовується один знак одно)

<> порівняння нерівностей

< менше ніж

> більше ніж

!< не менше ніж

!> не більше ніж

<= менше ніж або дорівнює

>= більше ніж або одно

```
SELECT * FROM Products
WHERE Manufacturer = 'Samsung'
```

```
SELECT * FROM Products
WHERE Price > 45000
```

```
SELECT * FROM Products
WHERE Price * ProductCount > 200000
```

AND : операція логічного І. Вона об'єднує два вирази

```
SELECT * FROM Products
WHERE Manufacturer = 'Samsung' AND Price > 50000
```

OR : операція логічного АБО. Вона також поєднує два вирази

```
SELECT * FROM Products
WHERE Manufacturer = 'Samsung' OR Price > 50000
```

NOT : операція логічного заперечення. Якщо вираз у цій операції помилкове, то загальна умова є істинною.

```
SELECT * FROM Products
WHERE NOT Manufacturer = 'Samsung'
```

v2

```
SELECT * FROM Products
WHERE Manufacturer <> 'Samsung'
```

```
SELECT * FROM Products
WHERE Manufacturer = 'Samsung' OR Price > 30000 AND ProductCount > 2
```

```
SELECT * FROM Products
WHERE (Manufacturer = 'Samsung' OR Price > 30000) AND ProductCount > 2
```

```
SELECT * FROM Products
WHERE ProductCount IS NULL
```

```
SELECT * FROM Products

WHERE ProductCount IS NOT NULL
```

Оператор IN дозволяє визначити набір значень, які мають стовпці:

```
SELECT * FROM Products

WHERE Manufacturer IN ('Samsung', 'Xiaomi', 'Huawei')
```

v2

```
SELECT * FROM Products

WHERE Manufacturer = 'Samsung' OR Manufacturer = 'Xiaomi' OR Manufacturer = 'Huawei'
```

```
SELECT * FROM Products

WHERE Manufacturer NOT IN ('Samsung', 'Xiaomi', 'Huawei')
```

Оператор BETWEEN визначає діапазон значень за допомогою початкового та кінцевого значення, якому має відповідати вираз

```
SELECT * FROM Products

WHERE Price BETWEEN 20000 AND 40000
```

```
SELECT * FROM Products

WHERE Price NOT BETWEEN 20000 AND 40000
```

SELECT * FROM Products

WHERE Price * ProductCount BETWEEN 100000 AND 200000

Оператор LIKE приймає шаблон рядка, якому має відповідати вираз.

WHERE выражение [NOT] LIKE шаблон_строки

Для визначення шаблону може застосовуватися ряд спеціальних символів підстановки:

% : відповідає будь-якому підрядку, який може мати будь-яку кількість символів, при цьому підрядок може і не містити жодного символу

_ : відповідає будь-якому одиночному символу

[ ] : відповідає одному символу, який вказаний у квадратних дужках

[-] : відповідає одному символу з певного діапазону

[^] : відповідає одному символу, який не вказано після символу ^

Деякі приклади використання підстановок:

WHERE ProductName LIKE 'Galaxy%'

Відповідає таким значенням як "Galaxy Ace 2" або "Galaxy S7"

WHERE ProductName LIKE 'Galaxy S_'

Відповідає таким значенням як "Galaxy S7" або "Galaxy S8"

WHERE ProductName LIKE 'iPhone [78]'

Відповідає таким значенням як iPhone 7 або iPhone8

WHERE ProductName LIKE 'iPhone [6-8]'

Відповідає таким значенням як iPhone 6, iPhone 7 або iPhone8

WHERE ProductName LIKE 'iPhone [^7]%'

Відповідає таким значенням як iPhone 6, iPhone 6S або iPhone8. Але не відповідає значенням "iPhone 7" та "iPhone 7S"

WHERE ProductName LIKE 'iPhone [^1-6]%'

Відповідає таким значенням як iPhone 7, iPhone 7S і iPhone 8. Але не відповідає значенням "iPhone 5", "iPhone 6" та "iPhone 6S"

--

SELECT * FROM Products

WHERE ProductName LIKE 'iPhone [6-8]%'

SELECT * FROM Products

WHERE ProductName LIKE '%one%'

Для зміни вже наявних рядків у таблиці застосовується команда UPDATE . Вона має такий формальний синтаксис:

UPDATE имя_таблицы

SET столбец1 = значение1, столбец2 = значение2, ... столбецN = значениеN

[FROM выборка AS псевдоним_выборки]

[WHERE условие_обновления]

UPDATE Products

SET Price = Price + 5000

UPDATE Products

SET Manufacturer = 'Samsung Inc.'

WHERE Manufacturer = 'Samsung'

UPDATE Products

SET Manufacturer = 'Apple Inc.'

FROM

(SELECT TOP 2 FROM Products WHERE Manufacturer='Apple') AS Selected

WHERE Products.Id = Selected.Id

Для видалення застосовується команда DELETE:

DELETE [FROM] имя_таблицы

WHERE условие_удаления

```
DELETE Products
WHERE Id=9
```

```
DELETE Products
WHERE Manufacturer='Xiaomi' AND Price < 15000
```

```
DELETE Products FROM
(SELECT TOP 2 * FROM Products
WHERE Manufacturer='Apple') AS Selected
WHERE Products.Id = Selected.Id
```

Якщо необхідно видалити всі рядки незалежно від умови, то умову можна не вказувати:

```
DELETE Products
-------------------------------------
```

Типові інтерфейси

IClonable: призначений для глибокого копіювання об'єктів. У цьому випадку створюється новий об'єкт та в нього перекопійовуються усі значення полів старого. Необхідно, бо об'єкти – це значення-посилання, а при використанні звичайного = для присвоєння одного посилання іншому буде присвоюватись якраз саме посилання. Для поверхневого копіювання можна також використати **MemberwiseClone(),** але тоді вкладені об'єкти копіюватимуться по посиланню (лише типи-значення копіюватимуться по значенні).

```
class Person : ICloneable
    {
      public string Name { get; set; }
      public int Age {get; set; }
      public Person(string name, int age)
      {
        Name = name;
        Age = age;
      }
      public object Clone() => MemberwiseClone();
    }
```

IComparable: призначений для порівняння поточного об'єкта з об'єктом, який передається як параметр object? o. На виході він повертає ціле число, яке може мати одне із трьох значень:

- Меньше нуля. Отже, поточний об'єкт повинен перебувати перед об'єктом, який передається як параметр;
- Рівне нулю. Отже, обидва об'єкти рівні;
- Більше нуля. Отже, поточний об'єкт повинен перебувати після об'єкта, що передається як параметр.

Використовується (між іншим) для сортування об'єктів (за допомогою метода **Sort**()).

```
class Person : IComparable
{
   public string Name { get;}
   public int Age { get; set; }
   public Person(string name, int age)
   {
      Name = name; Age = age;
   }
   public int CompareTo(object? o)
   {
      if(o is Person person) return Name.CompareTo(person.Name);
      else throw new ArgumentException("Некоректне значення параметра");
   }
}
```

IEnumerable та IEnumerator: призначені для реалізації перебирання колекції об'єктів у циклі (зокрема у foreach).

Інтерфейс IEnumerable має метод, який повертає посилання на інший інтерфейс - перечислювач

```
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

А інтерфейс IEnumerator визначає функціонал для перебору внутрішніх об'єктів у контейнері

```
public interface IEnumerator
{
   bool MoveNext(); // переміщення однією позицію вперед у контейнері елементів
   object Current { get; } // поточний елемент у контейнері
   void Reset(); // Переміщення на початок контейнера
}
```

Метод MoveNext() переміщує покажчик на поточний елемент наступну позицію в послідовності. Якщо послідовність ще закінчилася, то повертає true. Якщо послідовність закінчилася, то повертається false.

Властивість Current повертає об'єкт у послідовності, який вказує покажчик.

Метод Reset() скидає покажчик позиції початкове положення.

Ітератор по суті є блоком коду, який використовує оператор yield для перебору набору значень. Цей блок коду може представляти тіло методу, оператора або блок get у властивостях.

Ітератор використовує дві спеціальні інструкції:

yield return – визначає елемент, що повертається (повертає значення та запам’ятовує стан)

yield break – вказує, що послідовність більше не має елементів

```
class Numbers
   {
      public IEnumerator<int> GetEnumerator()
      {
         for (int i = 0; i < 6; i++)
         {
            yield return i * i;
         }
      }
   }

   static class Int32Extension
   {
      public static IEnumerator<int> GetEnumerator(this int number)
      {
         int k = (number > 0) ? number : 0;
         for (int i = number - k; i <= k; i++) yield return i;
      }
   }
```

https://www.programiz.com/csharp-programming/yield-keyword#:~:text=yield%20return%20%2D%20returns%20an%20expression,yield%20break%20%2D%20terminates%20the%20iteration

Корисні книги

1. А. Васильєв. «Програмування на C# для початківців. Основні відомості"

2. Джеффрі Ріхтер. CLR via C#. Програмування на платформі Microsoft .NET Framework мовою C#

3. Філіп Джепікс, Ендрю Троєлсен. «Мова програмування C# 7 та платформи .NET та .NET Core»

4. Марк Дж. Прайс. C# 7 і .NET Core. Крос-платформна технологія для професіоналів»