МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Д. В. Настенко, А. Б. Нестерко

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ
Частина 1. Основи об’єктно-орієнтованого програмування на мові C#.

Навчальний посібник

Київ

НТУУ «КПІ»

2016

Об’єктно-орієнтоване програмування. Частина 1. Основи об’єктно-орієнтованого програмування на мові C#.: Навчальний посібник. / Д.В. Настенко, А. Б. Нестерко. – К.: НТУУ «КПІ», 2016. - 76с.

Гриф надано Вченою радою НТУУ «КПІ»

(Протокол № __ від _____2016 р.)

Електронне навчальне видання

ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ
Частина 1. Основи об’єктно-орієнтованого програмування на мові C#.

Навчальний посібник

для бакалаврів напряму підготовки 6.050701- "Електротехніка та електротехнології" програми професійного спрямування "Системи управління виробництвом та розподілом електроенергії".

Викладено основи об’єктно-орієнтованого програмування на прикладі мови C#, та основні принципи та засоби створення консольних додатків в середовищі Visual Studio .Net з використанням платформи .Net Framework. Для кращого засвоєння матеріалу в посібнику наведено багато прикладів класів та програм з поясненнями щодо їх створення та роботи.

Посібник призначений для бакалаврів напряму підготовки 6.050701- "Електротехніка та електротехнології" програми професійного спрямування "Системи управління виробництвом та розподілом електроенергії".

Автори: Настенко Дмитро Васильович , ст. викладач
Нестерко Артем Борисович, асистент

Відповідальний редактор: Яндульський Олександр Станіславович, д.т.н., професор

Рецензенти:
Тугай Юрій Іванович, д. т. н
Шполянський Олег Григорович, к.т.н.

Зміст

Вступ

Дисципліна «Об’єктно-орієнтоване програмування» є складовою частиною в фаховій підготовці бакалавра напряму підготовки 6.050701- "Електротехніка та електротехнології" програми професійного спрямування "Системи управління виробництвом та розподілом електроенергії".

Дисципліна знайомить студентів із сучасним методами та принципами розробки програмного забезпечення ПК: операційною системою Windows, інтегрованим середовищем розробки Visual Studio .NET, об’єктно-орієнтованим програмуванням та мовою C#. Вивчення курсу супроводжується значною кількістю лабораторних робіт, що дозволяє студентам отримати стійкі практичні навички по застосуванню програмних засобів, що вивчаються. Дисципліна “Об’єктно-орієнтоване програмування" є необхідною для успішного вивчення подальших курсів, пов’язаних з управлінням сучасною енергетикою.

Дисципліна відноситься до циклу дисциплін природничо-наукової підготовки. Дисципліни, які забезпечують “Об’єктно-орієнтоване програмування”, наступні: “Обчислювальна техніка та алгоритмічні мови", “Вища математика”, “Фізика”, “Теоретичні основи електротехніки” та “Англійська мова”.

Дисципліна викладається студентам спеціальностей “Системи управління виробництвом та розподілом електроенергії” у п’ятому, шостому та сьомому семестрах.

Метою вивчення дисципліни є отримання знань з основ та принципів об’єктно-орієнтованого програмування та методів взаємодії прикладних програм з різноманітними пристроями релейного захисту.

Даний навчальний посібник розроблено згідно програми п’ятого семестру та кредитного модуля «Основи об’єктно-орієнтованого програмування».

Розділ 1. Введення до об’єктно-орієнтовного програмування

1.1. Основні поняття та принципи ООП

Об'є́ктно-орієнто́ване програмува́ння (**ООП**) — одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою. Основи ООП **інкапсуляція, успадкування** та **поліморфізм**. Незважаючи на те, що ця парадигма з'явилась в 1960-тих роках, вона не мала широкого застосування до 1990-тих. Сьогодні багато мов програмування (зокрема, Java, C#, C++, VB, Python, PHP, Ruby та Objective-C, ActionScript 3) підтримують ООП.

Парадигма (дав.-гр. παράδειγμα, paradeigma - приклад, зразок) - у сучасній філософії науки - система теоретичних, методологічних і аксіологічних установок, які взяті за зразок.

Клас – це шаблон який визначає форму, зміст та поведінку об’єкту. **Об’єкт** – це екземпляри класу.

Інкапсуляція – це об’єднання в одному цілому даних та алгоритмів обробки цих даних. Цей термін також включає в себе приховування даних, тобто приховування від зовнішнього користувача деталей реалізації об’єкта.

Успадкування – це властивість класу породжувати нащадків. Тобто: клас-нащадок (дочірній клас) успадковує від базового(батьківського) частину полів і методів, та може доповнюватись новими полями та методами.

Поліморфізм – це властивість споріднених об’єктів вирішувати подібні проблеми різними способами.

1.2. Платформа .Net Framework

.Net Framework – це інтегрована в Windows платформа, для створення програмних додатків різних типів, серед яких Windows-додатки, Web-додатки, додатки для мобільних телефонів і т.д. При розробці .Net Framework переслідувалися наступні цілі:

- створення узгодженого середовища виконання програм в незалежності від того де вони розміщені на локальному комп'ютері чи в Інтернеті;

- мінімізація конфліктів між різними версіями програмного забезпечення;

- гарантування безпеки виконання стороннього коду;

- забезпечення єдиних принципів розробки програм для різних типів додатків;

- створення середовища розробки для платформонезалежних гетерогенних додатків.

Два основних компоненти .Net Framework, це незалежне від мов програмування середовище виконання (Common Language Runtime, CLR) та базова бібліотека класів .NET.

Visual Studio .NET – інструмент для швидкої розробки різних типів програм на платформі .NET.

1.3. Мова C#. Типи даних. Прості оператори.

Мова C# (читається як *Сі шарп*)- одна з мов для написання додатків для архітектури .NET. Ця мова об'єктно-орієнтована, тому код програми в ній складається не з процедур та функцій, а з класів.

Найпростіша програма на мові C# виглядатиме так:

```
class Program
{
    static void Main()
    {
    }
}
```

де **Program** – ім'я класу, **Main()** – метод, точка входу в програму, з якої починається виконання коду додатку. Метод з ім'ям Main в програмі може бути лише один. Фігурні дужки {} в мові C# виділяють набір операторів, або ***блок***.

В середині класу або методу можна оголошувати змінні:

<Тип_змінної> <Ім'я_змінної> [=початкове_значення];

Наприклад:

```
int i;              //ціла змінна
double d = 1.5;     //дійсна змінна з
                    //початковим значенням 1,5
```

За допомогою подвійних косих (//) задаються коментарі до кінця рядка. А за допомогою пар символів /* та */ - багаторядкові коментарі.

Область видимості змінної в C# - блок коду (обмежений фігурними дужками {}). Змінна створюється на вході до області видимості і знищується при виході з неї.

Таблиця 1.1. Вбудовані типи даних C#

Тип C#	**Тип .Net**	**Кількість байт**	**Діапазон значень**
`bool`	Boolean	1	true або false
`byte`	Byte	1	від 0 до 255
`sbyte`	Sbyte	1	від -128 до 127
`short`	Int16	2	від -32 768 до 32 767
`ushort`	UInt16	2	від 0 до 65 535
`int`	Int32	4	від -2 147 483 648 до 2 147 483 647
`uint`	UInt32	4	від 0 до 4 294 967 295
`long`	Int64	8	від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`ulong`	UInt64	8	від 0 до 18 446 744 073 709 551 615
`float`	Single	4	наближено від $\pm 1.5\times10^{-45}$ до $\pm 3.4\times10^{38}$ з сімома значущими цифрами
`double`	Double	8	наближено від $\pm 5\times10^{-324}$ до $\pm 1.7\times10^{308}$ з 15 або 16 значущими цифрами
`decimal`	Decimal	12	наближено від $\pm 1.0\times10^{-28}$ до $\pm 7.9\times10^{28}$ з 28 значущими цифрами
`char`	Char	2	16-бітний символ Unicode

Зауваження: При створенні константи типу float після числа треба дописати «f» або «F» (1.2F), decimal - «m» або «M». Шістнадцяткові константи починаються з префікса «0x» (0xA – шістнадцяткове A, десяткове 10).

Таблиця 1.2. Оператори C# (анг. operator)

Оператор	**Опис**
=	присвоєння
+	додавання
-	віднімання
*	множення
/	ділення (для цілих операндів – цілочисельне ділення)
%	залишок від ділення (для цілих)
==	дорівнює
!=	не дорівнює
>	більше
<	менше
>=	більше або дорівнює
<=	менше або дорівнює
&&	логічне І (AND)
||	логічне Або (OR)
!	логічне Ні (Not)
&	побітове І
|	побітове Або
^	побітове виключне Або (XOR)
~	побітове Ні
<<	побітовий зсув вліво (SHL)
>>	побітовий зсув вправо (SHR)
+=	додати до змінної (i+=10 збільшує змінну і на 10)
-=	відняти від змінної
*=	помножити змінну
/=	поділити змінну
++x	префіксний інкремент (збільшити змінну у виразі на 1)
--x	префіксний декремент
x--	постфіксний інкремент (збільшити змінну після обчислення виразу на 1)
x++	постфіксний декремент
&=	побітове І
|=	побітове Або
^=	побітове виключне Або (XOR)
~=	побітове Ні
?:	тернарний оператор (схожий на if then else, A?B:C – означає обчислити значення виразу A, якщо воно істинне, то повернути значення B, інакше повернути значення C)

Приклади:

```
int i = 1;
int j;
j = i++;  //після обчислень і рівне 2, а j=1

int i = 1;
int j;
j = ++i;  //після обчислень і рівне 2 і j=2
```

1.4. Клас System.Math

Для виконання більш складних математичних обчислень використовуються поля та методи класу **Math** в просторі імен **System.** Простором імен називають логічну структуру в додатках .NET, яка дозволяє групувати споріднені класи та розв'язувати можливі конфлікти імен класів.

Приклад:

Скласти програму для обчислення змінної за заданою формулою:

$$\Delta = \frac{tg\alpha - \lg^2 \beta}{\gamma}$$

1. Ознайомившись з класом **Math** за допомогою помічника, знайдемо методи для знаходження тангенсу та натурального логарифму – Math.Tan та Math.Log.
2. Програми для обчислення цього виразу виглядатиме так:

```
class Program
{
    static void Main(string[] args)
    {
        double alpha = 4.5;
        double beta = 3.2;
        double gamma = 2.3;

        double log = System.Math.Log(beta);
        double delta =(System.Math.Tan(alpha)-
               log * log)/gamma;
    }
}
```

Оператор **using** дозволяє вказати простір імен з яким ми працюємо, та напряму звертатися до класів цього простору. Оператори **using** завжди розміщуються на початку файлу програми, перед усіма іншими операторами. Враховуючи це, спростимо наш приклад.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        double alpha = 4.5;
        double beta = 3.2;
        double gamma = 2.3;

        double log = Math.Log(beta);
        double delta = (Math.Tan(alpha) -
               log * log) / gamma;
    }
}
```

1.5. Ввід та вивід за допомогою System.Console

Тепер розглянемо, як організувати ввід та вивід. В консольному додатку найпростіше скористатися методами класу **Console** простору імен **System.**

- **Console.Read()** – зчитати окремий символ з клавіатури. Повертає число int, т.т. його результат потрібно привести до типу char, щоб отримати символ.

```
char ch = (char)Console.Read();
```

- **Console.ReadLine()** – читання рядку символів.

```
string st = Console.ReadLine();
```

Для того, щоб перетворити отриманий рядок у число, можна скористатися об'єктом **Convert** простору імен **System.**

```
double d =Convert.ToDouble(Console.ReadLine());
```

- **Console.Write**() та **Console.WriteLine**() – використовуються для організації форматованого виводу на екран. Причому **Write** - виводить рядок символів на екран, а виводить **WriteLine** рядок символів з переводом виводу на новий рядок.

```
Console.WriteLine("Hello World!");
```

Щоб вивести на екран значення якоїсь змінної, можна скористатися форматованим виводом. Наприклад:

```
int i = 1234;
Console.WriteLine("i = {0,5}",i);
```

Де `{0,5}` форматує значення виразу або змінної, що виводиться. Перше число у фігурних дужках відповідає за порядковий номер змінної у списку змінних перерахованих через кому (***нумерація починається з 0***). Наступне число – відповідає за кількість позицій на екрані виділених для відображення значення змінної (в наведеному прикладі - 5).

Для виводу чисел з плаваючою крапкою (комою), можна вказати і кількість цифр після крапки. Наприклад:

```
double d = 1234.12345;
Console.WriteLine("d = {0,7:f3}",d);
```

Виведе на екран: `d = 1234,123`. Тут `f` – символ форматування, вказує що число треба виводити, як дійсне з плаваючою крапкою, а число 3 – що дробова частина числа повинна містити три цифри.

Таблиця 1.3. Основні символів форматування числових даних

Символи форматування	**Опис**
`F або f`	Для виводу чисел з фіксованою точністю
`E або e`	Для виводу чисел в експоненційному форматі
`P або p`	Для виводу процентів
`N або n`	Для виводу чисел з розділеними розрядами
`C або c`	Для виводу значень чисел в національних валютах

D або d	Для виводу чисел в десятковому форматі
G або g	Для виводу чисел з фіксованою точністю або в експоненційному форматі
X або x	Для виводу цілих шістнадцяткових чисел

Контрольні запитання та завдання

1. Назвіть основні принципи об’єктно-оріентованого програмування. Дайте означення кожного з них.
2. Які основні цілі та завдання платформи .Net Framework?
3. Які базові типи даних C# ви знаєте? Наведіть приклади оголошення змінних базових типів.
4. Назвіть основні оператори C#. Наведіть приклади їх використання.
5. За допомогою «Бібліотеки MSDN» [1], [2] ознайомтеся з основними полями та методами класу System.Math. Створіть власні приклади їх застосування.
6. Які є методи вводу та виводу класу System.Console? Перевірте на практиці їх роботу.
7. Навести приклади використання форматованого виводу для кожного символу форматування, що згадується в цьому розділі.

Розділ 2. Основи мови C#

2.1. Перерахування enum

Перерахування – дозволяють створити в програмі набір констант. Вони являють собою типи зі значеннями. Спрощений синтаксис перерахувань наступний:

```
enum <Ім_я_перерахування> {список_констант}
```

де `список_констант` – перераховані через кому константи, що входять до перерахування.

Наприклад:

```
enum Days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

є перерахуванням днів тижня, де понеділку `Mon` ставиться у відповідність число 0, `Tue` – 1 (вівторок), і т. д. Якщо ж ми хочемо понеділку поставити у відповідність – 1, вівторку – 2 і т.д., то перерахування матиме вигляд:

```
enum Days {Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun};
```

де значення першого елементу встановлено в 1, а решта елементів приймають значення на 1 більше ніж попередній елемент.

Аналогічно можна задавати значення для кожного елементу перерахування. При цьому, якщо два елементи перерахування матимуть однакові числові значення, то вони вважатимуться рівними.

Для доступу до елементів перерахування використовується *оператор крапка*. Наприклад для звертання до вівторка: **`Days.Tue`**. Якщо ж нам потрібно за елементом перерахування отримати його числове значення, то можна скористатися **оператором явного перетворення** типів.

```
Console.WriteLine((int)Days.Wed);
```

Тут оператор `(int)` – вказує на те, що значення виразу треба привести до цілочисельного типу **`int`**. В результаті на екран виведеться число 3.

Всі перерахування в C# зв'язані з базовим класом `Enum`, для якого передбачено ряд корисних методів. Наприклад, якщо нам потрібно дізнатися ім'я елементу перерахування за його значенням, можна скористатися методом `GetName`:

```
using System;
class Program
{
    enum Days { Mon=1, Tue, Wed, Thu, Fri, Sat, Sun };
    static void Main(string[] args)
    {
        Console.WriteLine(
            Enum.GetName(typeof(Days),3)
        );
    }
}
```

Ця програма виведе на екран назву третього дня тижня - середи: `Wed`.

2.2. Умовний оператор if

Умовний оператор дозволяє програмі перевірити логічну умову, і в залежності від результату перевірки, виконати той чи інший фрагмент коду. Синтаксис умовного оператору наступний:

```
if (<Умова>)
     оператор1;
[else
     оператор2;]
```

Якщо `Умова` прийме значення **`true`**, то виконується перший оператор або набір операторів, інакше другий оператор або набір операторів. Причому друга частина умовного оператору **`else`** … - не є обов'язковою.

При використані набору операторів вони об'єднуються в блок, та обмежуються фігурними дужками, тоді синтаксис умовного оператору набуває наступного вигляду:

```
if (<Умова>){
     оператори1
 }
```

```
[else{
    Оператори2
}]
```

2.3. Оператор switch

Інший оператор для керування логікою програми в C# - це оператор **switch**. Він дозволяє керувати ходом програми при скінченому наборі варіантів вибору, що зарані відомі. Синтаксис цього оператору наступний:

```
switch (<вираз>){
    case <значення1>:
        оператори1
        break;
    [[case <значення2>:
        оператори2
        break;]
     . . .
    default:
        оператори_за_замовчуванням
        break;]
}
```

Тут тип `виразу` може бути одним зі злічених типів (байт, символ, ціле і т.д.) або рядком. Якщо значення виразу співпаде зі значенням `значення1`, то виконається набір операторів `оператори1`, зі значенням `значення2`, то виконається набір операторів `оператори2`. Якщо ж не підійде жоден з перераховних варіантів, то виконається набір операторів `оператори_за_замовчуванням` після ключового слова **default**.

Наприклад:

```
int i = 1;
switch (i){
    case 1:
        Console.WriteLine("Один");
        break;
    case 2:
        Console.WriteLine("Два");
        break;
    default:
        Console.WriteLine("Інше число");
        break;
}
```

Наступний приклад показує як для різних значень виконати одні й ті самі дії:

```
int i = 2;
switch(n){
    case 1:
    case 2:
    case 3:
        Console.WriteLine("Це 1, 2 або 3");
        break;
    default:
        Console.WriteLine("Інше число");
        break;
}
```

Щоб моделювати більш складні дії в операторі **switch**, можна замінити оператор **break** на **goto**, наприклад:

```
int d = 1, k=1;
switch (d){
    case 1:
        k++;
        break;
    case 2:
        k += 3;
        goto case 1;
}
```

Ключове слово **default** – необов'язкове в операторі **switch**.

Контрольні запитання та завдання

1. Що таке перерахування? Наведіть власний приклад перерахування.
2. Як використовується оператор явного перетворення типів?
3. За допомогою «Бібліотеки MSDN» [1], [2] ознайомтеся з основними методами класу System.Enum.
4. Описати роботу оператора if. Навести власні приклади застосування оператора.

5. Як використовується оператор switch? Створити власні приклади використання оператора switch з аикористанням перерахувань.

Розділ 3. Вирази та оператори C#

3.1. Вирази C#

Вираз – послідовність з одного або декількох операндів та від нуля до декількох операторів, яку можна обчислити, отримавши в результаті одне значення, об'єкт або простір імен. Вираз може складатися з літералу, виклику метода, оператору та його операндів, а також з простого імені. В якості простого імені може використовуватися ім'я змінної, члену типа, параметру метода, простору імен або типу.

У виразі можуть використовуватись оператори, які в свою чергу в якості параметрів використовують інші вирази, або виклики методів, для визначення параметрів яких викликаються інші методи. Таким чином вирази бувають прості та складені.

Приклади виразів:

```
(Int32)Math.Floor((dtTo - dtFrom).TotalSeconds / 5)
DateTime.MinValue
```

Літерали - це сталі значення, у яких немає власних імен. Наприклад `21` і `"Text"` у наступному прикладі є літералами.

```
int y = 21;
string str = "Text";
```

3.2. Оператори C#

Дії програми виражаються в **операторах (анг. statements)**. До них відносяться оголошення змінних, присвоювання значень, виклик методів, прохід по колекціях, умовні переходи, тощо. Порядок виконання операторів в програмі називається потоком керування або потоком виконання. Потік керування може відрізнятися при кожнім старті програми, в залежності від реакції програми на вхідні дані, які вона отримує під час виконання.

Оператор може складатися з одного рядку коду, що закінчується крапкою з комою або з набору однорядкових операторів в **блоці**. Блок

операторів обмежується фігурними дужками ({}) і може містити вкладені блоки.

Таблиця 3.1. Типи операторів C# (анг. statement)

Категорія	**Опис**

Оператори оголошення	Оператори оголошення представляє собою нову змінну або константу.
	`int y = 21;`
	`string str = "Text";`
	
	`const double d = 2.33;`

Оператори виразів	Оператори виразів, обчислюють значення виразу, та зберігають його в змінній.
	`a = -a * x * x / (2 * i - 1) / (2 * i);`
	`sum += a;`

Оператори вибору	Оператори вибору дозволяють розгалуження коду, в залежності від однієї або кількох умов. `if, else, switch, case`
Оператори ітерацій	Оператори ітерацій дозволяють перегляд колекцій або повторення одного і того ж набору операторів до виконання заданої умови. `do, for, foreach, in, while`
Оператори переходу	Оператори переходу виконують передачу керування іншому розділу коду `break, continue, default, goto, return, yield`
Оператори обробки виключень	Оператори обробки виключень дозволяють акуратно відновитися після виключних ситуацій, що виникають пі час виконання. `throw, try-catch, try-finally, try-catch-finally`
Оператори checked і unchecked	Оператори checked і unchecked дозволяють вказати, чи дозволено числовим операціям переповнення, коли результат зберігається у змінній, що замала для зберігання результуючого значення. `checked, unchecked`
Оператор await	Оператор для керування в асинхронних методах

Оператор yield return.	Оператор для отримання результату у вигляді колекції
Оператор fixed	Не дозволяє збірнику сміття переносити змінну
Оператор lock	Оператор lock дозволяє обмежувати одночасний доступ до блоків коду тільки до одного потоку.
Пустий оператор	Пустий оператор складається з крапки з комою (;). Не виконує жодних дій.

3.3. Оператори ітерацій

В попередньому розділі було розглянуто оператори переходу, в цьому ж розглянемо оператори ітерацій (або цикли) та пов'язані з ними оператори переходу.

Цикли – програмні конструкції, що дозволяють виконувати один той самий блок коду до тих пір, поки не виконається якась умова. В мові C# передбачено чотири види циклів:

- **`for`**
- **`while`**
- **`do ... while`**
- **`foreach ... in`**

Перші три перейшли в мову C# з класичного C, а **`foreach`** – Visual Basic.

Цикл **`for`** – найбільш універсальний з циклів у мовах C, C++ та Java, з його допомогою можна змоделювати роботу всіх інших. Синтаксис цього циклу:

```
for([ініціалізатор];[умова];[ітератор])
     оператор;
```

`Ініціалізатор` – вираз або набір виразів, перерахованих через кому, що обчислюється перед початком циклу. Зазвичай використовується для задання початкових значень змінних, що використовуються в циклі.

Умова – логічний вираз, значення якого перевіряється перед кожною операцією цикла, якщо значення true, то виконується тіло циклу, якщо ж false – цикл завершується.

Ітератор - вираз або набір виразів, перерахованих через кому, що обчислюються в кінці кожної ітерації циклу.

Слід зазначити, що всі три віщенаведені елементи циклу – необовязкові. І найпростіший цикл, має вигляд:

```
for (;;){
          //... набір операторів
}
```

Оператор – тіло циклу, виконується на кожній ітерації. Може замінюватись набором операторів у фігурних дужках.

Приклади:

```
for (int i = 1; i < 10; i++){
    Console.WriteLine(i);
}
```

цикл, що виводить на екран значення від 1 до 9. Зауважимо, що змінна i оголошена в середині циклу, тому її область видимості обмежена тілом циклу.

```
int i, j;
for (i = 1, j = 2; i < 10 && j < 24; i++, j+=2){
    Console.WriteLine(i*j);
}
```

цикл де одночасно використовуються два лічильники.

Цикл **while** – класичний цикл з передумовою, має наступний синтаксис:

```
while (<умова>)
    оператор;
```

Умова – повертає значення true або false. Це значення обчислюється та перевіряється перед кожною ітерацією, якщо значення true – то виконується тіло циклу, якщо false – цикл завершується.

Оператор – тіло циклу, виконується на кожній ітерації. Може замінюватись набором операторів у фігурних дужках.

Приклад:

```
int n = 1;
while (n < 5){
    Console.WriteLine("n = {0}", n);
    n++;
}
```

Виводить на екран рядки:

```
n = 1
n = 2
n = 3
n = 4
```

Цикл **do ... while –** цикл з післяумовою. Аналогічний до **while** за виключенням того, що умова перевіряється в кінці кожної ітерації, а не перед початком. Має наступний синтаксис:

```
do
   оператор;
while (<умова>);
```

Умова – повертає значення true або false. Це значення обчислюється та перевіряється наприкінці кожної ітерації, якщо значення true – то ще раз виконується тіло циклу, якщо false – цикл завершується.

Оператор – тіло циклу, виконується на кожній ітерації. Може замінюватись набором операторів у фігурних дужках.

Приклад:

```
int n = 1;
do {
    Console.WriteLine("n = {0}", n);
    n++;
}
while (n < 0);
```

Виводить на екран рядок:

```
n = 1
```

І стає помітним, що так як умова перевіряється в кінці, то тіло циклу обов'язково виконається хоч один раз.

Цикл **`foreach...in`** – спеціальний цикл для роботи з набором елементів – колекцією (прикладом одного з типів колекцій є масив, про масиви і колекції детальніше на наступних заняттях). Має такий синтаксис:

```
foreach(<тип> <ім'я_зміної> in <вираз_колекція>)
    оператор;
```

`ім'я_зміної` – ім'я змінної, яка використовується для доступу перебором до всіх елементів, заданих за допомогою `виразу_колекції`.

`тип` – тип даних змінної.

`Оператор` – тіло циклу, виконується на кожній ітерації. Може замінюватись набором операторів у фігурних дужках.

Приклад:

```
int[] myArray = { 1, 3, 5, 8, 2 };
foreach (int i in myArray) {
    Console.WriteLine("i = " + i);
}
```

виведе на екран:

```
i = 1
i = 3
i = 5
i = 8
i = 2
```

3.4. Оператори переходу

Іноді виникає необхідність припинити виконання циклу за якоюсь умовою в тілі циклу, для цього в мові C# передбачені оператори **`break`** та **`continue`**.

`break` – перериває виконання циклу і передає управління до наступного за циклом оператору.

`continue` – перериває поточну ітерацію циклу і переходить одразу до наступної ітерації.

Контрольні запитання та завдання

1. Що таке вирази C#?
2. Назвіть основні типи операторів C#.
3. Які є оператори ітерацій?
4. Навести власні приклади різних операторів ітерацій.
5. Створити приклади використання операторів break і continue в циклах.

Розділ 4. Масиви

4.1. Поняття масиву

Масив – набір елементів однакового типу, звертатися до яких можна за одним спільним ім'ям. При оголошенні масиву використовують наступний синтаксис:

```
<тип>[] <ім'я_масиву>;
```

де `тип` – тип даних, `ім'я_масиву` – ім'я масиву за яким в подальшому можна звертатись до його елементів.

Після оголошення масиву необхідно за допомогою оператору **new** задати його розмірність.

Наприклад:

```
int[] myArray;
myArray = new int[10]; //масив цілих з десяти
                       //елементів

double[] NumArr = new double[5];
                     // масив з 5ти дійсних
```

Можна також одразу ініціювати масив початковими значеннями:

```
int[] myArray = new int[5] { 1, 3, 5, 8, 2 };
```

або ще простіше:

```
int[] myArray = { 1, 3, 5, 8, 2 };
```

без ключового слова **new** розмір масиву визначиться автоматично.

Звертатися до елементів масиву можна виказавши його ім'я та індекс елемента у квадратних дужках. Проте слід зазначити, що у мові C# ***нумерація елементів починається з нуля***, і перший елемент масиву матиме індекс 0.

4.2. Клас System.Array

Всі масиви C# є нащадками класу **System.Array** у якого є ряд корисних властивостей та методів. Наприклад властивість:

Length – повертає кількість елементів масиву;

Rank – повертає розмірність масиву (1 для одновимірного і т.д.).

Тепер розглянемо приклад програми для виводу на єкран масиву за допомогою циклу **for**:

```
using System;
class Program {
    static void Main() {
        int[] arr = { 1, 3, 5, 8, 2 };
        for (int i = 0; i < arr.Length; i++) {
            Console.WriteLine("arr[{0}]={1}", i,
arr[i]);
        }
        Console.ReadKey();
    }
}
```

Ця програма виведе на екран наступне:

```
arr[0]=1
arr[1]=3
arr[2]=5
arr[3]=8
arr[4]=2
```

В цій програмі можна замінити цикл **for** на **foreach**:

```
foreach (int i in arr) {
    Console.WriteLine(i);
}
```

але слід враховувати, що у випадку циклу **foreach** ітераційну змінну можна використовувати тільки для читання.

Серед методів **System.Array** слід виділити наступні:

Clear() – очищує мсив, заповнивши всі елементи 0 або пустими значеннями **null**;

Copy() – копіює заданий діапазон значень з одного масиву в інший;

Clone() – створює повну копію масиву разом зі значеннями;

IndexOf(),LastIndexOf() – відповідно шукають перше та останнє входження елементу в масив;

Sort() – сортує одновимірний масив за зростанням;

BinarySearch() – швидкий пошук в сортованому масиві;

Reverse() – змінює порядок слідування елементів масиву на протилежний.

Розглянемо приклад:

```
using System;
class Program {
    static void Main() {
        int[] arr = { 1, 8, 5, 3, 2 };//створимо масив
        //створимо копію
        int[] arrNew = (int[])arr.Clone();
        Array.Sort(arrNew);   //відсортуємо масив
        for (int i = 0; i < arrNew.Length; i++) {
            //виведемо відсортовані значення
            Console.WriteLine(
                 "arrNew[{0}]={1}", i, arrNew[i]
             );
        }
        Console.ReadKey();
    }
}
```

результатом роботи програми буде:

```
arrNew[0]=1
arrNew[1]=2
arrNew[2]=3
arrNew[3]=5
arrNew[4]=8
```

4.3. Багатовимірні масиви

Багатовимірний масив – це масив який визначається двома або більше вимірами, а до його елементів треба звертатися за домомогою двох або більше індексів.

Найпростіші багатовимірні масиви – двовимірні прямокутні масиви (або матриці). Для їх оголошення використовують наступний синтаксис:

```
<тип>[,] <ім'я_масиву>;
```

де `тип` – тип даних, `ім'я_масиву` – ім'я масиву за яким в подальшому можна звертатись до його елементів.

Приклад створення матриці 5 на 7:

```
int[,] matrix = new int[5, 7];
```

до її елементів можна звертатися наступним чином:

```
matrix[0,0] = 7;  //лівий верхній елемент
matrix[3,5] = 10; //десь всередині
matrix[4,6] = 0;  //правий нижній елемент
```

Тепер розглянемо як заповнити та вивести на екран прямокутну матрицю за допомогою циклів:

```
using System;
class Program {
    static void Main() {
        //створюємо матрицю
        int[,] matrix = new int[5, 7];
        //заповнюємо елемнти матриці сумою індексів
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 7; j++) {
                matrix[i, j] = i + j;
            }
        }
        //виведемо отримані значення на екран
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 7; j++) {
                Console.Write("{0,3}",matrix[i, j]);
            }
            Console.WriteLine();
        }
        Console.ReadKey();
    }
}
```

На екрані отримаємо:

```
  0   1   2   3   4   5   6
  1   2   3   4   5   6   7
  2   3   4   5   6   7   8
  3   4   5   6   7   8   9
  4   5   6   7   8   9  10
```

Аналогічно до двовимірних можна створювати тривимірні, чотиривимірні і т.д. масиви:

```
int[, ,]matrix3 = new int[5, 7, 5];
int[, , ,]matrix4 = new int[4, 4, 3, 3];
```

4.4. Зубчасті масиви

Зубчасті масиви **(**jagged array**)** – це масиви рядки яких можуть мати різну довжину. Причому кожен рядок зубчастого масиву є одновимірним масивом. Ці масиви оголошуються на ступним чином:

```
тип[][] ім'я_масиву;
```

Щоб створити двовимірний зубчастий масив, спочатку треба задати кількість рядків, наприклад:

```
int[][] arrJ = new int[3][];
```

потім задати розмірність кожного з рядків:

```
arrJ[0] = new int[3];
arrJ[1] = new int[4];
arrJ[2] = new int[2];
```

звертатися до елементів масиву потрібно так:

```
arrJ[0][0] = 2;
arrJ[0][1] = 4;
//...
arrJ[2][1] = 17;
```

Часто зубчасті масиви доцільніше використовувати замість прямокутних. Наприклад в методі Гауса для розв'язання системи лінійних рівнянь, якщо в матриці необхідно переставляти місцями рядки, то при використанні зубчастого масиву відпадає необхідність поелементного копіювання даних.

Приклад:

```
using System;
class Program {
    static void Main() {
```

```
        //створюємо зубчасту матрицю 3x4

        float[][] arrJ = new float[3][];
        for (int i = 0; i < arrJ.Length; i++) {
            arrJ[i] = new float[4];
        }

        //заповнюємо елементи рядка його номером

        for (int i = 0; i < arrJ.Length; i++) {
            for (int j = 0; j < arrJ[i].Length; j++) {
                arrJ[i][j] = i;
            }
        }

        //міняємо місцями перший рядок з останнім

        float[] x = arrJ[0];
        arrJ[0] = arrJ[2];
        arrJ[2] = x;
    }
}
```

Контрольні запитання та завдання

1. Що таке масив? Наведіть власні приклади оголошення масивів.
2. За допомогою «Бібліотеки MSDN» [1], [2] ознайомтеся з основними властивостями та методами класу System.Array.
3. Класифікація масивів.
4. В чому різниця між прямокутними та зубчастими масивами?
5. Навести приклади використання прямокутних та зубчастих масивів.

Розділ 5. Основи ООП в C#

5.1. Класи в C#. Поля та методи

Клас – це шаблон який визначає форму, зміст та поведінку об'єкту. Об'єкт – це екземпляри класу. За допомогою класу реалізується перший з основних принципів об'єктно-орієнтованого програмування – ***інкапсуляція***. Інкапсуляція – це об'єднання в одному цілому даних та алгоритмів обробки цих даних. Цей термін також включає в себе приховування даних, тобто приховування від зовнішнього користувача деталей реалізації об'єкта. Дані зберігаються у вигляді ***полів***, а алгоритми обробки у вигляді ***методів***. Поля, ще іноді називають змінними класу. Оголошення класу в мові C# наступне:

```
[модифікатор_доступу] class <ім'я_класу> {
    // оголошення полів
    [мод._доступу] <тип1> <ім'я_поля1>
         [=початкове_значення1];
    [мод._доступу] <тип2> <ім'я_поля2>;
         [=початкове_значення2]
    // ...
    // оголошення методів
    [мод._доступу] <тип_значення1> <ім'я_методу1>
           ([параметри1]){
        //тіло методу
    }
    [мод._доступу] <тип_значення2> <ім'я_методу2>
           ([параметри2]){
        //тіло методу
    }
    //...
}
```

Де `[модифікатор_доступу]` - дозволяє вказувати рівень доступу до класу або елементів класу, приймає значення з таб. 5.1.

Таблиця 5.1. Модифікатори рівня доступу

Модифікатор	**Опис**
`private`	компонент доступний тільки в середині класу, приймається за замовчуванням
`protected`	компонент доступний тільки в середині класу або в класі нащадку

internal	компонент доступний тільки в середині класу або в цій же програмі (або зборці)
protected internal	компонент доступний тільки в середині класу, в класі нащадку або в цій же програмі (або зборці)
public	компонент доступний без обмежень

Оголошення змінних класів або полів ми вже розглядали на попередніх заняттях, зазначимо тільки, що при створенні об'єкту, якщо не задані початкові значення, поля отримають наступні початкові значення:

Таблиця 5.2. Значення за замовчуванням

Тип поля	**Значення за замовчуванням**
Всі числові типи	0
bool	false
char	'\0'
Об'єктні типи	null – що означає пусте посилання

Приклад класу, який складається тільки з полів:

```
public class Transformer {  //клас трансформатор
    public int windingCount = 2;//кількість обмоток
    public int temperature; //поточна температура
    public string power;    //потужність
    public string model;    //марка
    public int yearBuilt;   //рік випуску
}
```

Більш детально зупинимося на методах:

`тип_значення1, тип_значення2` - тип змінної, що повертається методом. Якщо метод ніого не повертає – цей тип задається ключовим словом **`void`**, інакше тіло циклу обов'язково повинно містити оператор **`return`**, за допомогою якого метод повертає значення заданого типу. Якщо тип об'єктний, то повертається посилання (вказівник) на цей об'єкт.

`параметри1, параметри2` – необов'язковий список параметрів вигляду:

```
    [ref|out|params] <тип_параметру1> <ім'я_параметру1>
[, <тип_параметру2> <ім'я_параметру2>[, ...]]
```

наприклад:

```
public class Power {   //клас електрична потужність
    //... поля якщо потрібно
    // метод обчислює повну потужність
    public float Full(float U, float I) {
        return U * I;
    }
    // метод для обчислення активної потужності
    public float Active(
         float U, float I, float cosFi
         ) {
        return Full(U, I) * cosFi;
    }
}
```

5.2. Створення об'єкту. Конструктор класу

Для описаного класу `Transformer` об'єкт створюємо за допомогою ключового слова **`new`**.

```
Transformer myTrans1 = new Transformer();
```

Тепер ми можемо звертатися до полів та методів створеного об'єкту використовуючи оператор «крапка» (.) разом з посиланням на об'єкт:

```
myTrans1.windingCount = 2;
myTrans1.model = "АТДЦТН-400/330/110/35";
myTrans1.yearBuilt = 2003;
Console.WriteLine("Марка={0}", myTrans1.model);
```

В цьому прикладі `Transformer()` після ключового слова **`new`** – це ***конструктор класу***. Більш детально з конструкторами познайомимося в наступному розділі. Зараз же зазначимо, що це спеціальний метод, який викликається при створенні об'єкту і ім'я його співпадає з іменем класу.

5.3. Типи передачі параметрів

Для простих типів передача параметрів може бути за значенням та за посиланням, для об'єктних тільки за посиланням. При ***передачі за значенням***, в методі створюється копія змінної-параметра в яку записується значення і при зміні значення в середині методу значення в базовій змінній

не змінюється. Для того, щоб змінювати значення в середині методу використовується ***передача за посиланням***.

За замочуванням передача простих параметрів відбувається за значенням. Для передачі простих параметрів за посиланням використовується ключове слово **`ref`**. Наприклад:

```
using System;
using System.Collections.Generic;
using System.Text;

class Test_ref {
    void Mov(int a, int b) {
        int x = a;
        a = b;
        b = x;
    }
    void Xchg(ref int a, ref int b) {
        int x = a;
        a = b;
        b = x;
    }
    static void Main(string[] args) {
        int a = 3, b = 4;
        Test_ref p = new Test_ref();
        p.Mov(a, b);
         //значення a, b не змінюються
        Console.WriteLine("a={0}, b={1}", a, b);
        p.Xchg(ref a, ref b);
         //a та b обмінялися значеннями
        Console.WriteLine("a={0}, b={1}", a, b);
        Console.ReadKey();
    }
}
```

Виведе на екран наступне:

```
a=3, b=4
a=4, b=3
```

Для передачі простих параметрів за значеням або за допомогою **`ref`** їм обовязково задавати початкові значення.

Для більшості випадків від методів достатньо, щоб вони повертали

одне значення, щоб отримати з методу додаткові значення – використовуються ***вихідні параметри***. Вихідні парамери оголошуються за допомогою ключового слова **out**, і вимагають обов'язкової ініціалізації в середині методу. Приклад:

```
using System;
using System.Collections.Generic;
using System.Text;

class Test_out {
    void Power(int c, out int a, out int b) {
        a = c * c;
        b = c * c * c;
    }
    static void Main(string[] args) {
        int a, b, c = 5;
        Test_out p = new Test_out();
        p.Power(c, out a, out b);
        Console.WriteLine("a={0}, b={1}", a, b);
        Console.ReadKey();
    }
}
```

Виведе на екран значення квадрату та кубу числа 5:

```
a=25, b=125
```

Щоб мати можливість передати в метод, змінну кількість параметрів можна використати ключове слово **params.** Воно може використовуватись тільки один раз і для останнього параметру, вказуючи, що метод може мати будь-яку кількість параметрів цього типу.

```
using System;
using System.Collections.Generic;
using System.Text;

class Test_params {
    int Sum(params int[] a) {
        int s = 0;
        foreach(int i in a) {
            s += i;
```

```
        }
        return s;
    }

    static void Main(string[] args) {
        Test_params p = new Test_params();
        Console.WriteLine(
            "Sum={0}", p.Sum(1, 2, 3, 4)
        );
        Console.WriteLine(
            "Sum={0}", p.Sum(1, 2, 3, 4, 5)
        );
        Console.ReadKey();
    }
}
```

Виведе на екран:

```
Sum=10
Sum=15
```

5.4. Перевантаження методів

Перевантаження методів - дозволяє використовувати одні й ті самі імена методів змінюючи набір аргументів. Це корисно коли схожі дії треба виконати для різних типів даних, або для різних наборів даних. Наприклад:

```
using System;
using System.Collections.Generic;
using System.Text;

class Test_Overload {
    //для додавання цілих
    int Add(int a, int b) {
        return a + b;
    }
    //для додавання дійсних
    double Add(double a, double b) {
        return a + b;
    }
    //для збільшення на 1
    int Add(int a) {
        return ++a;
    }
    static void Main(string[] args) {
```

```
            Test_Overload t = new Test_Overload ();
            Console.WriteLine("Sum={0}", t.Add(1, 2));
            Console.WriteLine("Sum={0}", t.Add(3.4, 4.4));
            Console.WriteLine("Sum={0}", t.Add(7));
            Console.ReadKey();
        }
}
```

Виведе:

```
Sum=2
Sum=7,8
Sum=8
```

Контрольні запитання та завдання

1. Навести власний приклад класу, що пов’язаний з майбутньою спеціальністю. Клас повинен містити поля різних типів та методи з різними наборами аргументів.
2. Які бувають модифікатори рівня доступу?
3. Який синтаксис створення об’єкту?
4. Навести приклади для різних типів передачі параметрів.
5. Що таке перевантаження методів? Створити приклади перевантаження методів.

Розділ 6. Детальніше про класи

6.1. Конструктори. Перевантаження конструкторів

Конструктор – це спеціальний метод класу, що викликається при створенні об'єкту класу. Ім'я конструктора завжди співпадає з ім'ям класу. Якщо розробником класу явно не визначено жодного конструктора, то для класу автоматично створюється конструктор без аргументів. Конструктори не повертають жодних значень, тому при їх оголошенні не потрібно вказувати тип, навіть **void**. Приклад [5]:

```
public class Car {                  //клас машина
    public string color;            //колір
    public string model;            //модель
    public int yearBuilt;           //рік випуску

    //конструктор без аргументів
    public Car() {
        Console.WriteLine("Створюється машина");
    }
}
```

Для описаного класу `Car` об'єкт створюємо за допомогою ключового слова **`new`**.

```
Car myCar = new Car();
```

При цьому при створенні об'єкту `myCar` буде визвано, явно заданий конструктор `Car()`.

Як і для інших, методів для конструкторів можливе перевантаження, тобто можна створювати конструктори з різними наборами аргументів. Найчастіше конструктори використовують, що б задавати початкові значення полів. Наприклад:

```
using System;
using System.Collections.Generic;
using System.Text;
public class Car {                  //клас машина

    public string color;            //колір
    public string model;            //модель
    public int yearBuilt;           //рік випуску
```

```
    public Car() {
        //конструктор без аргументів
        Console.WriteLine("Створюється машина");
    }

    public Car(string c, string m, int yb) {
        //конструктор з аргументами
        color = c;
        model = m;
        yearBuilt = yb;
    }
}
class Test_60 {
   static void Main(string[] args) {
       Car c0 = new Car();
       Car c1 = new Car("Black", "BMW", 2000);
       Console.ReadKey();
    }
}
```

6.2. Вмкористання this

Ключове слово **this** – використовується для посилання на поточний об’єкт. Одне з застосувань, те що воно дозволяє оголошувати аргументи з тими ж назвами, що і поля. Так конструктор з попереднього прикладу можна переписати наступним чином:

```
    public Car(string color, string model,
        int yearBuilt) {

        this.color = color;
        this.model = model;
        this.yearBuilt = yearBuilt;
    }
```

тут, `this.color` - означатиме звертання до поля `color`, а просто `color` - до аргументу конструктора `color`. Також ключове слово **this** можна використовувати при посиланні на інший конструктор при перевантаженні, цей механізм дозволяє запобігати надмірному дублюванню коду програми.

```
    public Car() {
        Console.WriteLine("Створюється машина");
    }


    public Car(string color, string model,
         int yearBuilt) : this() {

        this.color = color;
        this.model = model;
        this.yearBuilt = yearBuilt;
    }
```

В цьому прикладі, для конструктору з аргументами спочатку здійсниться виклик конструктору без аргументів. Інший приклад:

```
    public Car(string color, string model,
         int yearBuilt) {

        this.color = color;
        this.model = model;
        this.yearBuilt = yearBuilt;
    }

    public Car(string model, int yearBuilt)
        : this("", model, yearBuilt) {
    }
```

Демонструє, як на базі існуючого створити конструктор з меншою кількістю аргументів.

6.3. Властивості

Властивості – це один з механізмів інкапсуляції, вони одночасно дозволяють встановлювати та зчитувати значення полів за допомогою методів та приховувати поля від користувача. Властивості також зручно використовувати для перевірки введених даних на відповідність.

Для властивостей можна одночасно визначати два методи **`get`** та **`set`** (**`get`** – для отримання значення, **`set`** – для встановлення) або по одинці кожен з них. Приклад:

```
public class Car {
    private string model;
```

```
    public string Model {
        //зчитує значення поля model
        get { return model; }
        //встановлює значення поля model
        set { model = value; }
    }
}
class Test_61 {
    static void Main(string[] args) {

        Car c0 = new Car();
        c0.Model = "Audi";
        Console.WriteLine(c0.Model);
        Console.ReadKey();
    }
}
```

При компіляції, ці методи перетворяться на **get_Model** та **set_Model** відповідно, тому при написанні програм не рекомендується створення методів з подібними іменами.

Якщо визначено тільки метод **get** – то поле доступне тільки для читання, якщо **set** – то тільки для запису.

6.4. Статичні компоненти. Деструктори

Статичні поля, методи та властивості – задаються за допомогою ключового слова **static,** та належать самому класу, а не конкретному об'єкту. Тому і звертання до них виконується **за ім'ям класу**, а не за ім'ям об'єкту. Прикладами статичних полів та методів можуть слугувати поля та методи вже знайомого нам класу **Math** (такі як `Math.PI`, `Math.Abs(x)`, `Math.Pow(x, y)` і т. д.).

Деструктор – це спеціальний метод класу, що викликається при знищенні об'єкту. Оголошується за ім'ям класу та за допомогою символу **~**, без модифікаторів. Може використовуватись, наприклад, для закриття пов'язаних з об'єктом файлів, або для підрахунку створених об'єктів класу. Наприклад:

```
public class Car {
    public static int numOfCars = 0; //кількість машин
```

```
    public Car(){                          //конструктор
        ++numOfCars;
    }
    ~Car(){                                //деструктор
        --numOfCars;
    }
}
class Test_62 {
    static void Main(string[] args) {
        Car c0 = new Car();
        {
            Car c1 = new Car();
            Console.WriteLine(Car.numOfCars);
        }
        Console.WriteLine(Car.numOfCars);
        Console.ReadKey();
    }
}
```

В цьому прикладі статичне поле **numOfCars** збільшується на одиницю при створенні об'єкту класу **Car**, та зменьшується на одиницю при знищенні об'єкту. Проте при виводі значення поля **numOfCars**, найвирогідніше, в обох випадках буде число 2. Це відбуватиметься через те, що в середовищі **.Net** за виділення та звільнення пам'яті відповідає спеціальна служба **збирання сміття** (англ. garbage collection), яка звільняє пам'ять виділену під об'єкт не одразу ж при виході з зони видимості змінної, а при потребі. Для керування процесом збирання сміття в середовищі **.Net** створено спеціальний клас **System.GC.**

Статичний конструктор – оголошується за допомогою ключового слова **static**, без модифікаторів та не має аргументів. Викликається один раз перед створенням першого екземпляру класу. Може використовуватись для ініціалізації статичних полів та різного типу налаштувань класу. Приклад:

```
    static Car() {
        Console.WriteLine("Static Car");
    }
```

6.5. Простори імен

Простори імен (ключове слово **`namespace`**) – призначені для локалізації імен ідентифікаторів, і попередження їх конфліктів. В .Net простори імен в основному використовуються для:

- групування споріднених класів:

- для розв'язання конфлікту імен для різних класів з однаковими іменами.

Синтаксис:

```
namespace <ім'я_простору> {
   // Описова частина
}
```

В якості прикладу розглянемо класи для двовимірної та тривимірної точки:

```
namespace Graphics2D {
    public class Point{
        public int x,y;
    }
}

namespace Graphics3D {
    public class Point{
        public double x,y,z;
    }
}
```

Обидві описані класами `Point`, але двовимірна в просторі імен `Graphics2D`, а тривимірна – в `Graphics3D`.

Тепер, щоб використати в своєму коді двовимірний варіант, потрібно: або використовувати повне ім'я класу разом з ім'ям простору до якого він відноситься:

```
//...
    Graphics2D.Point p = new Graphics2D.Point();
//...
```

або скористатися оператором **`using`**:

```
using Graphics2D;
//...
     Point p = new Point();
//...
```

Зауваження: Простори імен можуть бути вкладеними один в один, створюючи таким чином ієрархію.

Наприклад:

```
namespace name1{
     namespace name2{
          //...
     }
}
```

Або те саме з використанням крапки:

```
namespace name1.name2{
          //...
     }
}
```

Контрольні запитання та завдання

1. Поняття конструктору. Приклади створення перевантажених конструкторів.
2. Які використовується ключове слово this?
3. Що таке властивості? Приклади використання властивостей.
4. Які поля та методи називаються статичними? Навести приклади використання статичних компонентів.
5. Для чого використовуються простори імен?

Розділ 7. Успадкування та поліморфізм

7.1. Успадкування в класах

Успадкування – це властивість класу породжувати нащадків. Тобто: клас-нащадок (дочірній клас) успадковує від базового(батьківського) частину полів і методів, та може доповнюватись новими полями та методами.

Поліморфізм – це властивість споріднених об'єктів вирішувати подібні проблеми різними способами.

Щоб один клас успадковував інший використовують наступний синтаксис:

```
class <ім'я_класу_нащадку> : <ім'я_базового_класу>
```

Для прикладу розглянемо базовий клас двовимірних графічних об'єктів:

```
public class Object2D {
    public int x, y; //поля-координати
    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //метод для "відображення"
    public void Draw() {
        Console.WriteLine(
             "Об'єкт з координатами {0},{1}",
             x, y
             );
    }
    //метод для "переміщення"
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
}
```

маючи цей клас, не важко створити нові - точку та коло.

```
// клас точка
public class Point : Object2D {
    public Point(int x, int y)
        // конструктор Point посилається
        // на батьківський
        // конструктор - Object2D
        : base(x, y) {
        //тут може бути додатковий код
    }
}
```

тут ключове слово **base** означає, що при створенні об'єкту точка спочатку викликається базовий конструктор `Object2D`, після чого в середині конструктора `Point` можна виконати унікальні для цього класу дії. Що наочніше демонструється в класі – коло.

```
// клас коло
public class Circle : Object2D {
    public int r; //нове поле - радіус
    public Circle(int x, int y, int r)
        : base(x, y) {
        this.r = r;
    }
}
```

В цьому класі введено додаткове поле для радіусу - `r`, і значення поля радіус – задається у власному конструкторі `Circle` після виконання батьківського конструктора `Object2D`.

Обидва ці класи автоматично успадковують методи `Draw()` і `Move(int dx, int dy)` батьківського класу, і результат роботи цих методів ні чим не відрізнятиметься від визову аналогічних методів для об'єкту `Object2D`.

7.2. Перевантаження батьківських методів

Проте більш цікавим є випадок коли методи відображення (`Draw()`) для кола та точки відрізнятимуться від базових. При цьому при оголошенні батьківського метода повинно використовуватись ключове слово **virtual** (можливий), а при оголошенні методу для дочірнього класу ключове слово – **override** (відхиляти, перевизначати).

Наведемо повний приклад:

```
using System;
using System.Collections.Generic;
using System.Text;
public class Object2D {
    public int x, y; //поля-координати
    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //метод для "відображення"
    public virtual void Draw() {
        Console.WriteLine(
            "Об'єкт з координатами {0},{1}",
            x, y
            );
    }
    //метод для "переміщення"
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
}

// клас точка
public class Point : Object2D {
    public Point(int x, int y)
        // конструктор Point посилається
        // на батьківський
        // конструктор - Object2D
        : base(x, y) {
    }
    public override void Draw() {
        //повністю перекриває
        //батьківський метод
        Console.WriteLine(
            "Точка з координатами {0},{1}",
            x, y
            );
    }
}
```

```
// клас коло
public class Circle : Object2D {
    public int r; //нове поле - радіус
    public Circle(int x, int y, int r)
        : base(x, y) {
        this.r = r;
    }
    public override void Draw() {
        //спочатку викликає батьківський метод
        base.Draw();
        //потім доповнює його
        Console.WriteLine("це коло радіусом {0}", r);
    }
}
class Test_70 {
    static void Main() {
        Point p0 = new Point(10,20);
        Point p1 = new Point(1, 40);
        Circle c0 = new Circle(15, 17, 10);
        p0.Draw();
        p1.Draw();
        c0.Draw();
        Console.ReadKey();
    }
}
```

Ця програма виведе на екран наступну інформацію:

```
Точка з координатами 10,20
Точка з координатами 1,40
Об'єкт з координатами 15,17
це коло радіусом 10
```

Крім того, з усіма об’єктами нащадками базового класу можна спільно працювати, як з об’єктами базового класу. Для прикладу залишимо класи `Object2D`, `Point` та `Circle` без змін перепишемо тільки клас `Test_70`:

```
class Test_71 {

    static void Main() {

        Point p0 = new Point(10,20);
        Point p1 = new Point(1, 40);
        Circle c0 = new Circle(15, 17, 10);
```

```
            //створення масиву вказівників
            //на об’єкти Object2D
            Object2D[] objects = new Object2D[3];

            //спочатку всі елементи цього масиву рівні null
            //заповнимо його посиланнями на
            //створені об’єкти
            objects[0] = p0;
            objects[1] = p1;
            objects[2] = c0;

            //тепер над ними можна виконувати групові дії
            //відображення
            foreach(Object2D o in objects) {
                o.Draw();
            }
            Console.WriteLine("\n\rпісля зміщення\n\r");

            //зміщення
            foreach(Object2D o in objects) {
                o.Move(5, -4);
            }
            Console.ReadKey();
        }
}
```

Результатом роботи цієї програми буде:

```
Точка з координатами 10,20
Точка з координатами 1,40
Об'єкт з координатами 15,17
це коло радіусом 10

після зміщення

Точка з координатами 15,16
Точка з координатами 6,36
Об'єкт з координатами 20,13
це коло радіусом 10
```

7.3. Клас System.Object

Слід зазначити, що взагалі всі класи .Net є нащадками базового класу **`System.Object`**. Це наслідування відбувається неявно, без буд-яких вказівок. Від **`System.Object`** всі класи успадковують ряд корисних

методів, серед яких:

- `public Type` **`GetType`** `()` – повертає об’єкт класу `Type`, що містить детальну інформацію поточний об’єкт (до якого класу відноситься, чи є об’єкт масивом, чи являється екземпляром класу або перерахуванням і т. д.
- `public virtual string` **`ToString`** `()` – повертає представлення об’єкту у вигляді рядка (для більності об’єктів – це ім’я класу, а для чисел значення у вигляді рядку), може перевантажуватись.
- `public virtual bool` **`Equals`** `(Object obj)` для порівняння поточного об’єкту з іншими), може перевантажуватись, повертає `true`, якщо об’єкти рівні.
- `public static bool` **`Equals`** `(Object objA, Object objB)` – для порівняння двох різних об’єктів.

7.4. Перевантаження операторів

Перевантаження операторів дозволяє дати визначення операторам для класів створених користувачем. Головна перевага механізму перевантаження операторів в тому, що він дозволяє використовувати природній синтаксис при написанні програм. Що приводить до зменшення витрат на написання програм.

Синтаксис перевантаження:

```
public static <тип_значення> operator
<унарний_оператор>  (<тип_параметру> <назва_параметру>)

public static <тип_значення> operator
<бінарний_оператор> (
      <тип_параметру1> <назва_параметру1>,
      <тип_параметру2> <назва_параметру2>
)
```

тут

`тип_значення` – тип результату дії оператора;

`унарний_оператор` – один з операторів **+, -, !, ~, ++, --, true, false**;

`бінарний_оператор` - один з операторів **+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=**;

`тип_параметру` – тип параметру в операторі;

`назва_параметру` – тип параметру в операторі.

Приклад перевантаження:

```
using System;
public class Point{
    double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // при додаванні точок утворюється нова,
    // координати якої рівні сумі відповідних координат
    public static Point operator +(Point a, Point b){
        return new Point(a.x+b.x,a.y+b.y);
    }

    // при множенні точки на число утворюється нова,
    // координати якої рівні добутку відповідних
    // координат на число
    public static Point operator *(Point a, double d) {
        return new Point(a.x * d, a.y * d);
    }

    // переозначимо метод ToString()
    public override string ToString() {
         return String.Format(
             "Точка ({0},{1})",this.x,this.y
             );
    }
}

class Program {
    static void Main(string[] args) {
         Point p1 = new Point(5, 8);
         Point p2 = new Point(7, 12);
```

```
            Console.WriteLine(
                p1 + "+" + p2 + "=" + (p1 + p2)
                );
            Console.WriteLine(
                p1 + "*" + 5 + "=" + (p1 * 5)
                );
            Console.ReadKey();
        }
}
```

Ця програма виведе на екрані:

```
Точка(5,8)+Точка(7,12)=Точка(12,20)
Точка(5,8)*5=Точка(25,40)
```

Те що в цій програмі перевантажено метод `ToString()`, дозволяє спростити вивід за допомогою методу `Console.WriteLine`, так як метод `ToString()` використовується за замовчуванням при перетворенні об'єкту в рядок.

Круглі дужки використоуються у виразах `(p1 + p2)` та `(p1 * 5)` для того, щоб спочатку виконались дії над об'єктами точка, а вже потім перетворення в рядок за допомогою методу `ToString()`.

Зауваження. Слід зазначити, що оператри **`==`** та **`!=`**, **`>`** та **`<`**, **`>=`** та **`<=`** потрібно перевантажувати тільки парами.

Для прикладу розглянемо порівняння точок:

```
    public static bool operator ==(Point a, Point b) {
        if(a.x == b.x && a.y == b.y) {
            return true;
        } else {
            return false;
        }
    }
    public static bool operator !=(Point a, Point b) {
        return !(a == b);
    }
```

Як бачимо з цього прикладу, при перевантаженні оператора недорівнює, можна скористатися тим, що він обернений до оператора дорівнює. Також можна спростити код оператора дорівнює, скориставшись тим що вираз

`(a.x == b.x && a.y == b.y)` - повертає значення булівського типу:

```
    public static bool operator ==(Point a, Point b) {
        return a.x == b.x && a.y == b.y;
    }
```

Текст програми повністю:

```
using System;
public class Point{
    double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return String.Format("Точка({0},{1})", this.x,
this.y);
    }

    public static bool operator ==(Point a, Point b) {
        return a.x == b.x && a.y == b.y;
    }
    public static bool operator !=(Point a, Point b) {
        return !(a == b);
    }
}
class Program {
    static void Main(string[] args) {
        Point p1 = new Point(5, 8);
        Point p2 = new Point(7, 12);
        Console.WriteLine(p1 + " та " + p2 +
            (p1 == p2 ? " - рівні " : "- не рівні"));
        Console.ReadKey();
    }
}
```

Результат роботи програми:

```
Точка(5,8) та Точка(7,12)- не рівні
```

7.5. Індексатор

Крім вищезгаданих операторів можна перевантажити оператор [], але для цього використовують спеціальну властивість – ***індексатор***. Індексатор можна використовувати для доступу до різних полів об’єкта або полів-масивів, оголошуються вони за допомогою ключового слова `this` та

квадратних дужок [].

Приклад використання індексатору для різних полів:

```
using System;

public class Car {                  //клас машина
    public string color;            //колір
    public string model;            //модель
    public int yearBuilt;           //рік випуску

    // індексатор
    public string this[int i] {
        // властивість на читання
        get {
            switch(i) {
                case 0:
                    return model;
                case 1:
                    return color;
                case 2:
                    return yearBuilt.ToString();
                default:
                    return null;
            }
        }
        // властивість для запису
        set {
            switch(i) {
                case 0:
                    model=value;
                    break;
                case 1:
                    color=value;
                    break;
                case 2:
                    yearBuilt = Convert.ToInt32(value);
                    break;
                default:
                    break;
            }
        }
    }
}

class Program {
    static void Main(string[] args) {
```

```
            Car c = new Car();
            c[0] = "Audi";
            c[1] = "Green";
            c[2] = "2000";
            for(int i = 0; i < 3; i++) {
                Console.WriteLine(c[i]);
            }
            Console.ReadKey();
        }
}
```

На екрані буде наступне:

```
Audi
Green
2000
```

В цій програмі використання індексатора дозволяє поставити у відповідність кожному полю класа індекс і звертатися до полів по імені об’єкта за допомогою цього індекса.

Приклад використання індексатору для масиву:

```
using System;
public class Vector {
    int[] v;

    // конструктор
    public Vector(int size) {
        v = new int[size];
    }

    // метод
    public int Length() {
        if(v == null) {
            return 0;
        } else {
            return v.Length;
        }
    }

    // індексатор
    public int this[int i] {
        get { return v[i]; }
        set { v[i] = value; }
    }
}
```

```
class Program {
    static void Main(string[] args) {
        Vector vect = new Vector(5);
        for(int i = 0; i < vect.Length(); i++) {
            vect[i] = i * 2;
            Console.WriteLine(
                 "vect[{0}] = {1}", i, vect[i]
                 );
        }
        Console.ReadKey();
    }
}
```

Програма виведе на екран:

```
vect[0] = 0
vect[1] = 2
vect[2] = 4
vect[3] = 6
vect[4] = 8
```

Як бачимо з цього прикладу, використання індексатору дозволяє звертатися до елементів поля масиву `v`, як до елеметів об'єкту `vect`.

Контрольні запитання та завдання

1. Що таке успадкування та поліморфізм?
2. Як реалізується механізм успадкування в C#?
3. Які використовується ключове слово base?
4. За допомогою яких ключових реалізується перевантаження батьківських методів?
5. Який об'єкт є базовим для всіх класів в середовищі .Net?
6. Які методи є спільними для всіх класів?
7. Для чого потрібно перевантаження операторів?
8. Які оператори можна перевантажувати?
9. Яка особливість перевантаження операторів порівняння?
10. Що таке індексатор?
11. Які області застосування індексаторів?

Розділ 8. Інтерфейси

8.1. Поняття інтерфейсу

Інтерфейси – це спеціальні класи, що містять тільки назви методів (без реалізації методів та без полів). Реалізацію ж, цих методів покладено на класи-нащадки інтерфейсів. Оголошують інтерфейси за допомогою ключового слова **interface**.

```
[модифікатор_доступу] interface <ім'я_інтерфейсу> {
        // оголошення методів
        [мод._доступу] <тип_значення1> <ім'я_методу1>
               ([параметри1]);
        [мод._доступу] <тип_значення2> <ім'я_методу2>
               ([параметри2]);
        }
        //...
    }
```

`ім'я_інтерфейсу` – зазвичай починається з великої літери **I** (використання великої літери **I** напочатку назви інтерфейсу не є обов'язковим, проте – бажане, для полегшення розуміння коду).

Основне застосування інтерфейсів – полегшить синхронізацію програмного коду написаного різними розробниками. Знову розглянемо приклад з графічними об'єктами. Нехай нам потрібно, щоб всі графічні об'єкти мали дві координати, відображалися та рухалися. Можемо створити інтерфейс:

```
using System;
public interface IGraphObject {
    // властивості для координат
    int X {
        get;
        set;
    }
    int Y {
        get;
        set;
    }
    //метод для відображення
    void Draw();
```

```
    //метод для переміщення
    void Move(int dx, int dy);
}
```

Тепер успадкуємо від цього інтерфейсу клас `Object2D`:

```
public class Object2D : IGraphObject {
    protected int x, y; //поля-координати

    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //реалізація успадкованих властивостей
    public int X {
        get { return x; }
        set { x = value; }
    }
    public int Y {
        get { return y; }
        set { y = value; }
    }

    //реалізація успадкованих методів
    //метод для відображення
    public virtual void Draw() {
        Console.WriteLine(
             "Об'єкт з координатами {0},{1}",
             x, y
             );
    }

    //метод для переміщення
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }

}
```

Перевага інтерфейсі над класами, це можливість успадковувати один клас від кількох інтерфейсів. Наприклад опишемо інтерфейс для обертання `IRotated`:

```
public interface IRotated {
    //метод для обертання
    void Rotate(int x0, int y0, double angle);
}
```

Додамо його в список предків для `Object2D` і нам доведеться додатково описати метод `Rotate`.

```
public class Object2D : IGraphObject, IRotated {
    protected int x, y; //поля-координати

    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //реалізація успадкованих властивостей
    public int X {
        get { return x; }
        set { x = value; }
    }
    public int Y {
        get { return y; }
        set { y = value; }
    }
    //реалізація успадкованих методів
    //метод для "відображення"
    public virtual void Draw() {
        Console.WriteLine(
             "Об'єкт з координатами {0},{1}",
             x, y
             );
    }
    //метод для "переміщення"
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
    //
    public void Rotate(int x0, int y0, double angle) {
        // метод що описує обертання
        // навколо заданого центру
        // ...
    }
}
```

Зауваження. Слід зазначити, що в C# клас в списку предків може мати лише один клас та декілька інтерфейсів перерахованих через кому.

8.2. Інтерфейс IComparable

Ще одна перевага інтерфейсів, це можливість за допомогою підтримки стандартних інтерфейсів середовища .Net користатися різноманітними методами стандартних об'єктів. Наприклад, щоб сортувати ваші об'єкти за допомогою класу **System.Array** – достатньо успадкувати їх від стандартного інтерфейсу **IComparable** і описати метод `int` **CompareTo**`(object obj)`. Цей метод повинен повертати:

- нуль – у випадку коли об'єкти, що порівнюються однакові;
- від'ємне ціле число – коли поточний об'єкт менше об'єкту-аргументу;
- додатне ціле число – коли поточний об'єкт більше об'єкту-аргументу.

Наприклад, нехай:

- одна точка рівна іншій, коли обидві їх координати рівні;
- точка **A(x1, y1)** більша точки **B(x2, y2)**, коли перша координата **A** більша за першу координату **B (x1>x2)**, або коли **x1=x2** і **y1>y2.**

Розглянемо відповідну програму:

```
using System;

public class Point : IComparable {
    double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return String.Format(
             "Точка({0},{1})", this.x, this.y
             );
    }
```

```
    //метод від IComparable
    public int CompareTo(object obj) {
        Point p = (Point)obj;
        if(this.x > p.x) {
            return 1;
        } else if(this.x < p.x) {
            return -1;
        } else {
            if (this.y > p.y) {
                return 1;
            } else if(this.y < p.y) {
                return -1;
            } else {
                return 0;
            }
        }
    }
}
class Program {
    static void Main(string[] args) {
        Point p1 = new Point(5, 8);
        Point p2 = new Point(7, 12);
        Point[] points = new Point[5];
        points[0] = p1;
        points[1] = p2;
        points[2] = new Point(5, 10);
        points[3] = new Point(5, 1);
        points[4] = new Point(15, 10);
        // для сортування використовується
        // стандартний метод Sort
        Array.Sort(points);

        foreach(Point p in points) {
            Console.WriteLine(p);
        }
        Console.ReadKey();
    }
}
```

Вона виведе на екран, точки у відсортованому вигляді:

```
Точка(5,1)
Точка(5,8)
Точка(5,10)
Точка(7,12)
Точка(15,10)
```

8.3. Оператори is та as

Щоб перевірити чи підтримує об'єкт той чи інший інтерфейс, можна скористатися одним з операторів **is** або **as**.

Оператор **is** – використовується для перевірки сумісності змінної або об'єкта одного типу з іншим типом даних і повертає значення **true** або **false**.

Наприклад:

```
int i;
if(i is IComparable){
    Console.WriteLine("Підтримує IComparable");
}
```

Дасть ствердну відповідь.

Оператор **as** – використовується для перетворення змінної одного типу в інший сумісний тип. Якщо типи не сумісні поверне **null**. Тому наступний фрагмент коду:

```
Point p0 = new Point(1, 2);
 IConvertible c = p0 as IConvertible;
 if(c == null) {
     Console.WriteLine(
          "Не підтримує IComparable"
          );
 }
```

Видасть:

```
Не підтримує IComparable
```

Контрольні запитання та завдання

1. Що таке інтерфейс? Призначення інтерфейсів.
2. Як використовується інтерфейс IComparable?
3. Створити приклади застосування інтерфейсів: власних та стандартних.
4. В чому відмінність між операторами is та as?
5. Створити приклад програми, в якій би використовувалися оператори is та as.

Розділ 9. Робота з рядками, датами, часом та проміжками часу.

9.1. Класи для роботи з рядками

Рядки (**`String`**) – тип даних для зберігання текстової інформації. Тип даних **`String`** – це послідовність, що не містить жодного або довільну кількість символів Юнікоду в UTF-16, тобто по 2 байти на кожен символ. В C# ключове слово `string` є псевдонімом для `String`. Тому `string` і `String` – еквівалентні.

Не дивлячись на те, що `String` – об'єктним типом, оператори `==` та `!=` визначені для порівняння вмісту об'єктів типу рядок, а не посилань як для інших класів. Наприклад:

```
using System;
using System.Collections.Generic;
using System.Text;

class Program {
    static void Main(string[] args) {
        string a = "hello";
        string b = "h";
        // Append to contents of 'b'
        b += "ello";
        Console.WriteLine(a == b);
        Console.WriteLine((object)a == (object)b);
        Console.ReadKey();
    }
}
```

Виведе на екран:

```
True
False
```

Що показує, те що вміст рядків однаковий, хоча об'єкти різні.

Для рядків `String` визначено оператори `+` та `+=` для додавання. Проте слід зазначити, що ці об'єкти являються незмінними: після створення їх неможливо змінити. Все методи `String` і оператори C#, які, начебто змінюють рядок, насправді повертають в результаті новий об'єкт-рядок.

```
using System;

class Program {
    static void Main(string[] args) {
        string a, b = "h";
        a = b;       //a і b посилаються на один об'єкт
        Console.WriteLine((object)a == (object)b);
        b += "ello";    //тепер це різні об'єкти
        Console.WriteLine((object)a == (object)b);
        Console.WriteLine("a==" + a);
        Console.WriteLine("b==" + b);
        Console.ReadKey();
    }
}
```

На екрані буде:

```
True
False
a==h
b==hello
```

Тобто спочатку посилання `a` і `b` були на один об’єкт, а після виконання оператору `+=` змінна `b` стала вказувати на новий об’єкт.

Так само, і оператор [] служить тільки для доступу для читання окремих символів, та не може використовуватись для зміни значень.

```
using System;

class Program {
    static void Main(string[] args) {
        string a = "hello";
        Console.WriteLine(a[0]);
        Console.WriteLine(a[1]);
        Console.WriteLine(a[4]);
        Console.ReadKey();
    }
}
```

Рядкові константи можна задавати в двох варіантах:

- в подвійних лапках ;
- в подвійних лапках з @ на початку.

В першому варіанті, використовуються Escape – послідовності, а в другому ні. Тому другий варіант зручний, щоб вказувати шляху до файлу.

Таблиця 9.1. Escape – послідовності

Escape – послідовність	**Опис**
\a	Дзвоник
\b	Повернення на одну позицію
\f	Перехід на нову сторінку
\n	Перехід на новий рядок
\r	Повернення каретки
\t	Горизонтальна табуляція
\v	Вертикальна табуляція
\0	null
\’	Символ - ’
\”	Символ – ”
\\	Символ - \
\udddd або \xdddd	Представляє знак Юнікоду де dddd – шістнадцяткове число

Наприклад:

```
using System;
class Program {
    static void Main(string[] args) {
        Console.WriteLine("\u0041BC\n\\");
        Console.WriteLine(@"C:\Temp\file1.txt");
        Console.ReadKey();
    }
}
```

Виведе на екран:

```
ABC
\
C:\Temp\file1.txt
```

Таблиця 9.2. Основні властвості та поля класу **`String`**

Назва	**Опис**
Empty	Статичне поле доступне тільки для читання, повертає пустий рядок
Length	Властивість, що повертає довжину рядка

Таблиця 9.3. Основні методи класу **`String`**

Назва	**Опис**
Compare	Статичний метод для порівняння рядків

`Contains`	Перевіряє, чи міститься підрядок в рядку
`Copy`	Статичний метод. Створює новий об'єкт, що є копією даного рядка
`CopyTo`	Копіює заданий діапазон символів в з рядка в масив символів
`EndsWith`	Перевіряє, чи завершується рядок заданим підрядком
`Format`	Статичний метод. Дозволяє створити новий рядок зі значень різних виразів за допомогою форматування
`IndexOf`	Повертає індекс входження підрядку або набору символів в рядок
`IndexOfAny`	Повертає індекс входження будь-якого з набору символів в рядок
`Insert`	Вставляє підрядок в рядок
`IsNullOrEmpty`	Статичний метод. Перевіряє чи є рядок пустим або `null`
`Join`	Статичний метод. Створює рядок з масиву рядків використовуючи заданий роздільник
`LastIndexOf`	Повертає індекс останнього входження підрядку або набору символів в рядок
`LastIndexOfAny`	Повертає індекс останнього входження будь-якого з набору символів в рядок
`Remove`	Видаляє з рядка задану кількість символів
`Replace`	Замінює входження символу або підрядку новим символом або підрядком
`Split`	Створює з рядку масив підрядків за вказаним набором символів роздільників
`StartsWith`	Перевіряє, чи починається рядок з заданого підрядку
`Substring`	Створює новий рядок з підрядку поточного рядку
`ToCharArray`	Створює масив символів з підрядка
`ToLower`	Переводить рядок до нижнього регістру
`ToUpper`	Переводить рядок до верхнього регістру
`Trim`	Видаляє всі входження вказаних символів на початку та в кінці рядку
`TrimEnd`	Видаляє всі входження вказаних символів в кінці рядку
`TrimStart`	Видаляє всі входження вказаних символів на початку рядку

Якщо методів класу **`String`** не достатньо, для перетворень текстової інформації можна використовувати клас – **`StringBuilder`** (простір імен **`System.Text`**), що дозволяє змінювати окремі символи та має свій набір

методів.

Наприклад:

```
using System;
using System.Collections.Generic;
using System.Text;

class Program {
    static void Main(string[] args) {
        StringBuilder sb = new StringBuilder("kyiv");
        sb[0] = Char.ToUpper(sb[0]);
        Console.WriteLine(sb.ToString());
        Console.ReadKey();
    }
}
```

Ця програма замінює першу літеру слова `kyiv` на заголовну, і виводить на екран наступне:

```
Kyiv
```

9.2. Структури для роботи з датою та часом

Для роботи з датою та часом в середовищі .Net реалізовано структури **`DateTime`** та **`TimeSpan`.**

Структура **`DateTime`** представляє дату та час в діапазоні від 00:00:00 1 січня 0001 року (н. е.) до 23:59:59 31 грудня 9999 року (н. е.). Значення часу вимірюється в 100-наносекундних одиницях, що називаються тактами, і точна дата представляється числом тактів від 00:00:00 1 січня 0001 року (н.е.) по григоріанському календарю.

Таблиця 9.4. Основні поля структури **`DateTime`**

Назва	**Опис**
`MaxValue`	Статичне. Повертає найменше можливе значення **`DateTime`**, доступне тільки для читання
`MinValue`	Статичне. Повертає найбільше можливе значення **`DateTime`**, доступне тільки для читання

Таблиця 9.5. Основні властивості структури **`DateTime:`**

Назва	**Опис**
`Date`	Для екземпляру **`DateTime`** повертає тільки дату без часу

Day	Для екземпляру **DateTime** повертає номер дня місяцю від 1 до 31
DayOfWeek	Для екземпляру **DateTime** повертає день тижня
DayOfYear	Для екземпляру **DateTime** повертає номер дня року від 1 до 366
Hour	Для екземпляру **DateTime** повертає години від 0 до 23
Millisecond	Для екземпляру **DateTime** повертає мілісекунди від 0 до 999
Minute	Для екземпляру **DateTime** повертає хвилини від 0 до 59
Month	Для екземпляру **DateTime** повертає місяць від 1 до 12
Now	Статичний. Повертає поточні дату та час.
Second	Для екземпляру **DateTime** повертає секунди від 1 до 59
Ticks	Для екземпляру **DateTime** повертає кількість тактів
TimeOfDay	Для екземпляру **DateTime** повертає час без дати
Today	Статичний. Повертає поточну дату
Year	Для екземпляру **DateTime** повертає рік від 1 до 9999

Таблиця 9.6. Основні методи структури **DateTime**

Назва	**Опис**
Add	Додає до екземпляру **DateTime** значення типу **TimeSpan**
AddDays	Додає до екземпляру **DateTime** вказане число днів
AddHoura	Додає до екземпляру **DateTime** вказане число годин
AddMilliseconds	Додає до екземпляру **DateTime** вказане число мілісекунд
AddMinutes	Додає до екземпляру **DateTime** вказане число хвилин
AddMonths	Додає до екземпляру **DateTime** вказане число місяців
AddSeconds	Додає до екземпляру **DateTime** вказане число секунд
AddTicks	Додає до екземпляру **DateTime** вказане число тактів
AddYearS	Додає до екземпляру **DateTime** вказане число років

DaysInMonth	Статичний. Для вказаного місяця повертає кількість днів
Parse	Статичний. Перетворює рядок в **DateTime**, якщо це можливо, якщо ні генерує помилку
TryParse	Статичний. Перетворює рядок в **DateTime**, і повертає значення типу **Boolean** з інформацією про успішність перетворення

Структура **TimeSpan** – представляє інтервал часу, що може містити як додатню так і від'ємну кількість днів, годин, хвилин, секунд та долей секунд.

Таблиця 9.7. Основні поля структури **TimeSpan**

Назва	**Опис**
MaxValue	Статичне. Повертає найменше можливе значення **TimeSpan**, доступне тільки для читання
MinValue	Статичне. Повертає найбільше можливе значення **TimeSpan**, доступне тільки для читання
TicksPerDay	Статичне. Константне. Кількість тактів в 1 дні
TicksPerHour	Статичне. Константне. Кількість тактів в 1 годині
TicksPerMillisecond	Статичне. Константне. Кількість тактів в 1 мілісекунді
TicksPerMinute	Статичне. Константне. Кількість тактів в 1 хвилині
TicksPerSecond	Статичне. Константне. Кількість тактів в 1 секунді
Zero	Статичне. Повертає нульове значення **TimeSpan**, доступне тільки для читання

Таблиця 9.8. Основні властивості структури **TimeSpan**

Назва	**Опис**
Days	Повертає ціле число – кількість днів в поточній структурі **TimeSpan**
Hours	Повертає ціле число – кількість годин в поточній структурі **TimeSpan**
Milliseconds	Повертає ціле число – кількість мілісекунд в поточній структурі **TimeSpan**
Minutes	Повертає ціле число – кількість хвилин в поточній структурі **TimeSpan**

`Seconds`	Повертає ціле число – кількість секунд в поточній структурі **`TimeSpan`**
`Ticks`	Повертає значення поточного інтервалу **`TimeSpan`** в тактах
`TotalDays`	Повертає дійсне, що відповідає значенню поточного інтервалу **`TimeSpan`** в днях
`TotalHours`	Повертає дійсне, що відповідає значенню поточного інтервалу **`TimeSpan`** в годинах
`TotalMilliseconds`	Повертає дійсне, що відповідає значенню поточного інтервалу **`TimeSpan`** в мілісекундах
`TotalMinutes`	Повертає дійсне, що відповідає значенню поточного інтервалу **`TimeSpan`** в хвилинах
`TotalSeconds`	Повертає дійсне, що відповідає значенню поточного інтервалу **`TimeSpan`** в секундах

Розглянемо програму, що демонструє всі ці властивості:

```
using System;

class TimeSpanPropertiesDemo {

    const string dataFmt =
        "{0,-12}{1,8}       {2,-18}{3,21}";

    static void ShowProperties(TimeSpan interval) {
        Console.WriteLine("{0,21}", interval);
        Console.WriteLine(
             dataFmt, "Days", interval.Days,
             "TotalDays", interval.TotalDays
             );
        Console.WriteLine(
             dataFmt, "Hours", interval.Hours,
             "TotalHours", interval.TotalHours
             );
        Console.WriteLine(
             dataFmt, "Minutes", interval.Minutes,
             "TotalMinutes", interval.TotalMinutes
             );
        Console.WriteLine(
             dataFmt, "Seconds", interval.Seconds,
             "TotalSeconds", interval.TotalSeconds
             );
        Console.WriteLine(
             dataFmt, "Milliseconds",
             interval.Milliseconds, "TotalMilliseconds",
```

```
                interval.TotalMilliseconds
                );

            Console.WriteLine(
                dataFmt, null, null,
                "Ticks", interval.Ticks
                );
        }

        static void Main() {
            ShowProperties(
                new TimeSpan(10, 20, 30, 40, 50)
                );
            Console.ReadKey();
        }
}
```

На екрані буде:

```
  10.20:30:40.0500000
Days              10      TotalDays               10,8546302083333
Hours             20      TotalHours                    260,511125
Minutes           30      TotalMinutes                  15630,6675
Seconds           40      TotalSeconds                   937840,05
Milliseconds      50      TotalMilliseconds              937840050
                          Ticks                   9378400500000
```

Для структур **DateTime** та **TimeSpan** визначено наступні оператори:

– Порівняння ==, !=, >, <, >=, <=, причому об'єкт типу **DateTime** порівнюють з об'єктом типу **DateTime**, а об'єкт типу **TimeSpan** – з об'єктом типу **TimeSpan;**

– Додавання: до об'єкту **DateTime** можна додати **TimeSpan** отримаємо значення типу **DateTime;** до **TimeSpan** додати **TimeSpan** отримаємо **TimeSpan;**

DateTime + TimeSpan -> DateTime

TimeSpan + TimeSpan -> TimeSpan

– Віднімання:

DateTime – DateTime -> TimeSpan

DateTime – TimeSpan -> DateTime

TimeSpan – TimeSpan -> TimeSpan

– Унарні оператори **+** та **–** :

+TimeSpan -> TimeSpan

–TimeSpan -> TimeSpan

Наприклад:

```
using System;

class Program_102{

    static void Main() {

        DateTime dt, dt2;
        dt = DateTime.Now;
        dt2 = new DateTime(2010, 10, 27, 15, 30, 44);
        TimeSpan ts = dt - dt2;
        Console.WriteLine(
             "dt={0} dt2={1} ts={2}", dt, dt2, ts
             );
        TimeSpan ts2 = new TimeSpan(1, 2, 3, 4);
        dt = dt - ts2;
        dt2 = dt2 + ts2;
        ts = ts + ts2;
        Console.WriteLine(
             "dt={0} dt2={1} ts={2}", dt, dt2, ts
             );
        Console.ReadKey();
    }
}
```

Виведе на екран приблизно наступне:

```
dt=27.05.2010 18:52:30 dt2=27.10.2010 15:30:44
ts=-152.20:38:13.9843750
dt=26.05.2010 16:49:26 dt2=28.10.2010 17:33:48
ts=-151.18:35:09.9843750
```

Контрольні запитання та завдання

1. За допомогою «Бібліотеки MSDN» [1], [2] ознайомтеся з основними властивостями та методами класів System.String і System.Text.StringBuilder.
2. В чому відмінність між класами String та StringBuilder?
3. Назвіть основні методи та властивості класів String та StringBuilder?

4. Створити власні приклади використання класів String та StringBuilder.
5. За допомогою «Бібліотеки MSDN» [1], [2] ознайомтеся з основними властивостями та методами структур System.DateTime та System.TimeSpan.
6. Назвіть основні методи та властивості DateTime?
7. Які основні поля та властивості структури TimeSpan?
8. Які оператори визначені для роботи з структурами DateTime та TimeSpan?
9. Навести власні приклади використання структур DateTime та TimeSpan.

Список літератури

1. Библиотека MSDN [Електронний ресурс]. – Режим доступу : URL http://msdn.microsoft.com/ru-ru/library/default.aspx – Назва з екрана.
2. .NET Framework [Електронний ресурс]. – Режим доступу : URL https://msdn.microsoft.com/ru-ru/library/w0x726c2(v=vs.110).aspx – Назва з екрана.
3. Справочник по C# [Електронний ресурс]. – Режим доступу : URL https://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx – Назва з екрана.
4. Visual Studio 2015 [Електронний ресурс]. – Режим доступу : URL https://msdn.microsoft.com/library/dd831853(v=vs.140).aspx – Назва з екрана.
5. Джейсон Прайс. Visual C# .NET. Полное руководство. / Джейсон Прайс. Майк Гандэрлой.: пер. с англ. – К.: ВЕК+, СПб.: КОРОНА принт, К.:НТИ, М.: Энтроп, 2004. - 954 с. - ISBN 966-7140-36-9.
6. Троэлсен Э. C# и платформа .NET. Библиотека программиста. / Эндрю Троелсен – СПб.: Питер, 2007. -900с.- ISBN – 978-5-318-00750-7.
7. Бишоп, Джудит. C# в кратком изложении / Дж. Бишоп, Хорспул Н.; пер. с англ. К. Г. Финогенова. - Москва: Бином. Лаборатория знаний, 2005. - (Программисту). - 472 с.: ил. - ISBN 5-94774-211-X
8. Сухарев, М. В. Основы Delphi. Профессиональный подход / М. В. Сухарев. - СПб. : Наука и Техника, 2004. - 596 с. - ISBN 5-94387-129-2.
9. C# : [наиболее полн. рук. : пер. с англ.] / [Х .М. Дейтел, Пол Дж. Дейтел, Тэм Р. Нието и др.]. - СПб. : БХВ-Петербург, 2006. - 1056 с. : ил. - ISBN 5-94157-817-2
10. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1 Пер. с англ. – М.: Издательско-торговый дом «Русская редакция», 2002. – 576с.:ил.- ISBN 5-7502-0210-0

11. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2 Пер. с англ. – М.: Издательско-торговый дом «Русская редакция», 2002. – 624с.:ил.- ISBN 5-7502-0220-8
12. Лабор В.В. Си Шарп: Создание приложений для Windows./ Лабор В.В. Мн.: Харвест, 2003. – 385 с.- ISBN 985-13-1405-6.
13. Вікіпе́дія [Електронний ресурс] . – Режим доступу : URL https://uk.wikipedia.org/wiki/%D0%93%D0%BE%D0%BB%D0%BE%D0%B2%D0%BD%D0%B0_%D1%81%D1%82%D0%BE%D1%80%D1%96%D0%BD%D0%BA%D0%B0 – Назва з екрана.