Програмне забезпечення

Програмне забезпечення (програмні засоби) (ПЗ) — сукупність програм системи обробки інформації і програмних документів, необхідних для експлуатації цих програм. Виконання програмного забезпечення комп'ютером полягає у маніпулюванні інформацією та керуванні апаратними компонентами комп'ютера. Наприклад, типовим для персональних комп'ютерів є відтворення інформації на екран та отримання її з клавіатури, потоку чи мережі.

Розрізняють *три класи ПЗ*:

1) **Системне ПЗ** — сукупність програм та програмних комплексів для забезпечення роботи комп’ютерів та комп’ютерних мереж. Розрізняють:
   a) базове СПЗ — мінімальний набір програмних засобів, які забезпечують роботу комп’ютера (ОС та операційні оболонки);
   b) сервісне СПЗ — сервісні програми, які розширюють можливості базового ПЗ (утиліти).
2) **Прикладне ПЗ**, що використовується для виконання конкретних функціональних завдань. Класифікація пакетів прикладних програм (ППП):
   a) проблемно-орієнтовані ППП: автоматизованого бухгалтерського обліку; фінансової діяльності; управлінням персоналом; управлінням виробництвом; банківські інформаційні системи;

b) ППП автоматизованого проектування — призначенні для підтримання роботи конструкторів та технологів (AutoCAD);

c) ППП загального призначення:

- СКБД;
- сервери БД — ПЗ по підтриманню архітектури клієнт-сервер;
- генератори звітів;
- текстові процесори;
- табличні процесори;
- засоби презентаційної графіки;
- інтегровані пакети (MS Office, Open Office);

d) методо-орієнтовані ППП: програмні продукти, які дозволяють використовувати математичні, статистичні та інші методи розв'язування задач;

e) офісні ППП:

- органайзери;
- програми перекладачі, програми розпізнавання текстів;
- комунікаційні системи;

- браузери, засоби створення Web-сторінок, засоби електронної пошти;

f) настільні видавничі пакети;

g) програмні засоби мультимедіа;

h) інтелектуальні системи (системи штучного інтелекту) — експертні системи, системи аналізу та розпізнавання мови (FIDE, MYSIN, Guru).

3) **Інструментальні засоби для розробки ПЗ** — програмні продукти, призначені для підтримки технології програмування. Вони поділяються на:

a) засоби для створення програм:

- мови та системи програмування;
- інструментальне середовище користувача.

Системи програмування містять: компілятор (транслятор), інтегроване середовище розробки, відладчик (debugger), засоби оптимізації коду програми, набір бібліотек, утиліти для роботи з бібліотеками, текстовими та двійковими файлами, довідкову систему, систему підтримки та керування продуктами програмного комплексу;

b) засоби для створення інформаційних систем (CASE-технології). CASE-технологія (Computer-Aided Software Engineering) — програмний комплекс, який автоматизує весь

технологічний процес аналізу, проектування, розробки та супроводу складних ПС. Засоби CASE-технології поділяються на:

- вбудовані в систему реалізації;
- незалежні від системи реалізації — засоби, орієнтовані на уніфікацію початкових стадій життєвого циклу програм і засобів їх документування, які забезпечують гнучкість реалізації.

Мови програмування

Мова програмування — це формалізована мова для опису алгоритму розв'язування задачі на ЕОМ. Мови програмування можна класифікувати наступним чином:

1) машинні мови (машинні коди);
2) машинно-орієнтовані мови (асемблер);
3) процедурно-орієнтовані мови (C, Pascal та інші);
4) об'єктно-орієнтовані мови (C++, C#, Java);
5) проблемно-орієнтовані мови (LISP, Prolog, SQL).

Етапи розробки програмного забезпечення

Згідно уніфікованого процесу розробки ПЗ (USDP) система розробки ПЗ містить у собі персонал, процес, проект і продукт (чотири П розробки ПЗ).

Персонал (ті хто створює ПЗ)	Процес (спосіб, яким це робиться)
Проект	Продукт (артефакти)

Стандартна послідовність кроків створення ПЗ:

1) зрозуміти природу та сферу застосування ПП;
2) вибрати процес розробки та створити план;
3) зібрати та з’ясувати вимоги до ПП;
4) спроектувати ПП;
5) реалізувати ПП;
6) протестувати ПП;
7) випустити продукт та забезпечити його супровід;

Процес розробки ПЗ

Моделі послідовного виконання стадій

Класичною моделлю процесу розробки ПЗ є *водоспадна (каскадна) модель*, яка належить до *моделей послідовного виконання стадій*. У межах водоспадної моделі процес розробки зображується у вигляді послідовності наступних фаз:

1) *аналіз вимог* — полягає у збиранні вимог до продукту. Результатом аналізу є як правило деякий текст;
2) *проектування* — описує внутрішню структуру продукту. Звичайно такий опис робиться у вигляді діаграм та текстів;
3) *реалізація* — програмування (кодування). Результатом є програмний код;
4) *інтеграція* — процес збирання ПП із окремих частин;
5) *тестування*;

Деталізований водоспадний процес створення ПЗ складається з наступних фаз:

1) концептуальний аналіз (визначення загальних принципів застосування);
2) об’єктно-орієнтований аналіз (виділення ключових класів);
3) проектування;
4) реалізація;

5) компонентне тестування;

6) інтеграція;

7) системне тестування;

8) супровід.

Схема водоспадного процесу наведена на рис. 1.

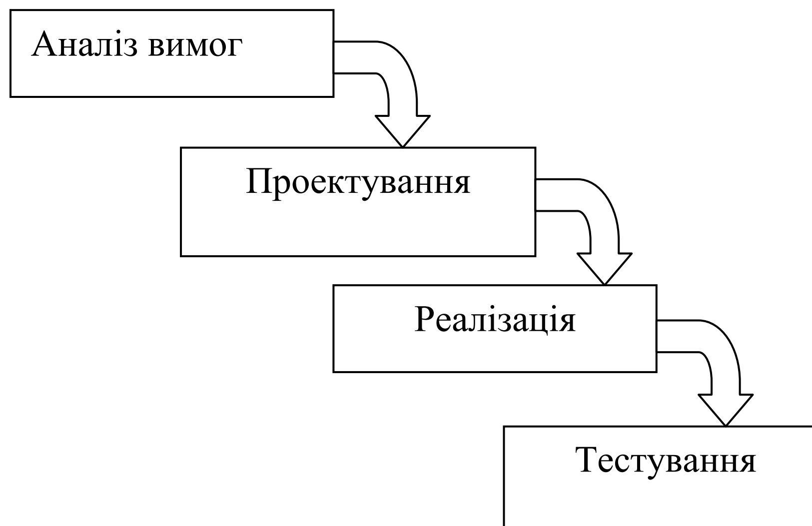

Рис. 1. Діаграма водоспадного процесу

У чистому вигляді водоспадний процес застосовується достатньо рідко (лише для невеликих за обсягом проектів). Більш реалістичною є водоспадна модель із зворотними зв'язками, наведена на рис. 2.

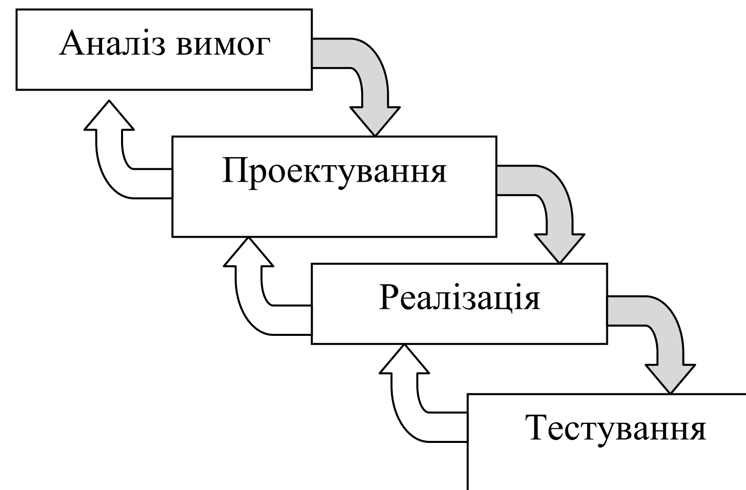

Рис. 2. Водоспадна модель із зворотнім зв'язком

Характеристики водоспадної моделі:

- Послідовне виконання стадій.
- Формальні перевірки по завершенні кожної стадій (інспекції, технічні огляди).
- Наявність документованих вимог і проекту.

Ітераційні моделі процесу розробки

Процеси, в яких водоспадна схема застосовується багаторазово, називаються *ітеративними* (ітераційними). До ітеративних моделей процесів відносяться *інкрементні та еволюційні моделі*.

В **інкрементних моделях** програмний продукт розробляється ітераціями — з додаванням на кожній функціональних можливостей. При цьому спочатку визначаються усі вимоги до ПС, і можливо розробляється попередній проект. Подальша розробка ПС розбивається на ітерації. У першій ітерації реалізується набір основних вимог, які забезпечують базову функціональність. Інші ітерації реалізуються в порядку критичності вимог для кінцевого користувача. При появі в середині ітерації нового набору вимог вони відкладаються до реалізації наступної версії.

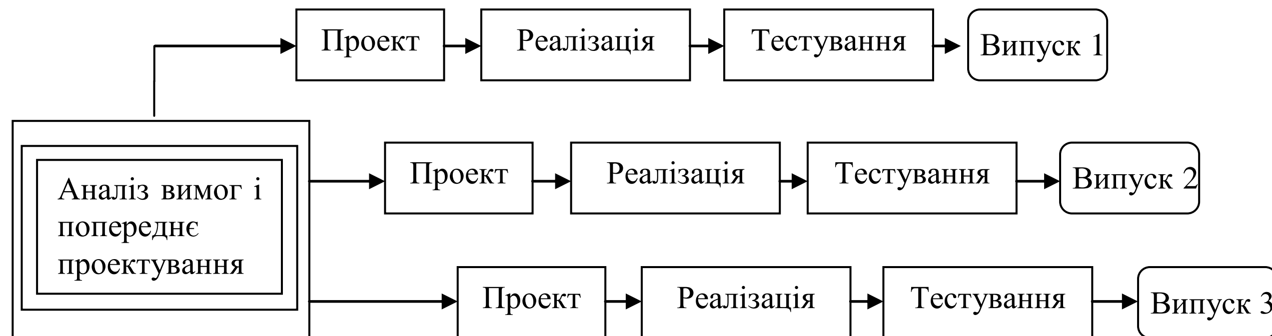

Рис. 3. Ітераційна модель з перекриттям ітерацій

Характеристики ітераційних моделей з приростом:

- Аналіз і проектування виконуються для усієї системи.
- Базові функціональні вимоги реалізуються першими.
- Інші вимоги реалізуються в наступних версіях.
- Проміжні версії придатні для використання.

На відміну від моделей з приростом, еволюційні моделі застосовуються в тих випадках, коли усі вимоги не можуть бути визначені одразу або відомо, що вони можуть змінитись. Розробка проекту по цим моделям також виконується ітераціями. Але кожна ітерація охоплює усі стадії розробки, від аналізу вибраного набору вимог до випуску версії. На кожній ітерації виконується прототипування вимог і проекту.

До найбільш відомих *еволюційних моделей* відноситься *спіральна модель*.

Спіральна модель розроблена Р. Боемом і застосовується для складних проектів або в тих випадках, коли проблеми проекту недостатньо зрозумілі. Модель відображає керований ризиком процес еволюції проекту від аналізу до готовності продукту.

На кожному витку спіралі (стадії) виконуються наступні дії:

1. Визначаються цілі стадії. Розглядаються альтернативні рішення для досягнення цих цілей.
2. Проводиться оцінювання цих рішень. Ідентифікуються ризики завершення стадії і виконується їх аналіз. Приймаються рішення про продовження або завершення стадії.

3. Розробляються робочі продукти стадії та план для наступної стадії.
4. Останній виток спіралі може мати структуру каскадної моделі.

Види ризиків:

- ризики, які стосуються технічних аспектів розробки;
- фінансові ризики;
- ризики експлуатації.

Характеристики спіральної моделі:

- Перший прототип моделює концепцію. Результатом є план вимог. Перед переходом до розробки наступного прототипу виконується аналіз ризику.
- Другий прототип моделює вимоги до ПЗ. Результатом є план розробки. Виконується аналіз ризику.
- Третій прототип моделює проект. В результаті створюється інтегрований і протестований прототип. Виконується аналіз ризику.
- Останній прототип (робочій) використовується як основа для детального проектування, кодування та тестування.

Схема спіральної моделі наведена на рис. 4.

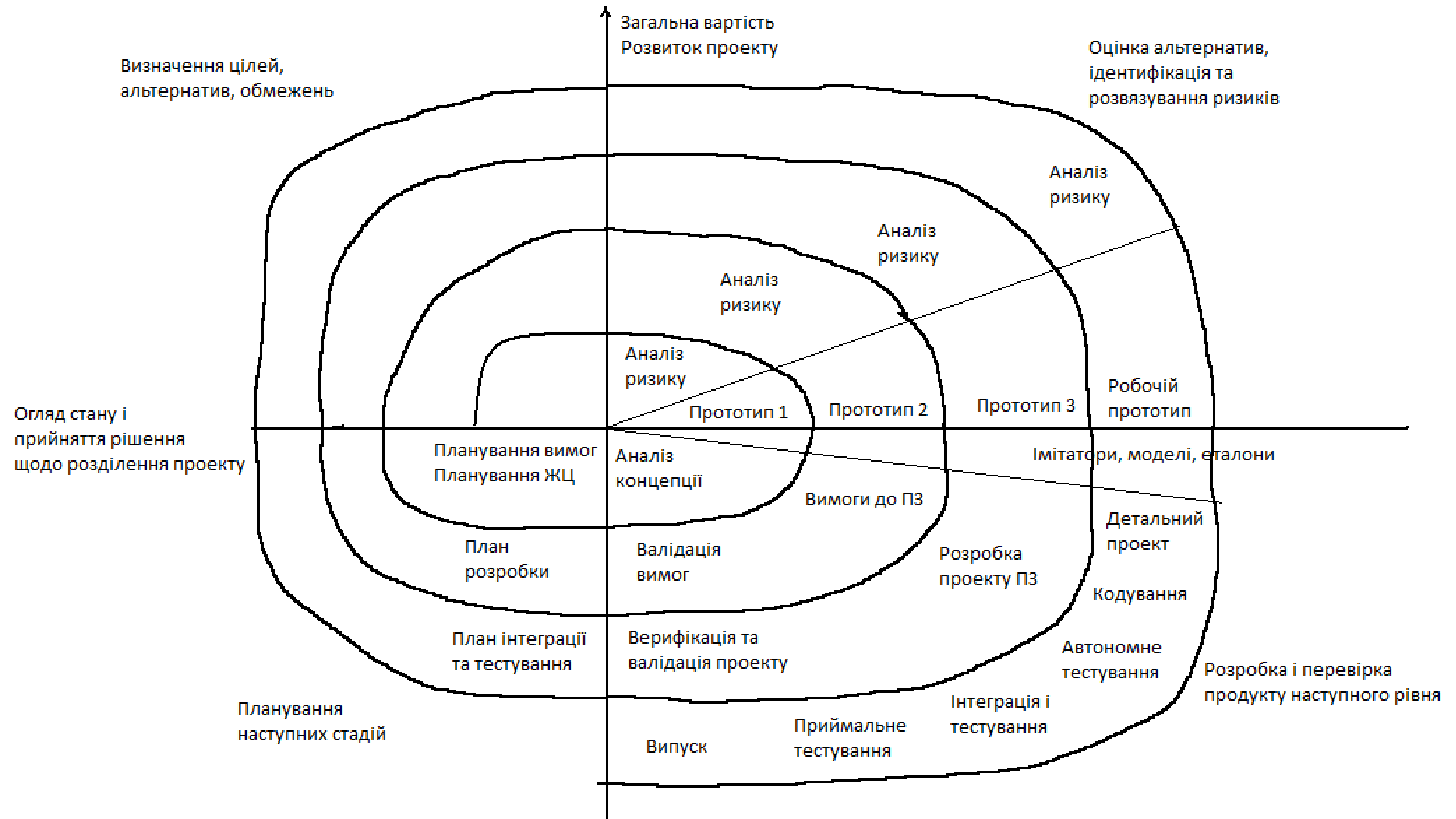

Рис. 4. Спіральна модель

Оцінки якості процесу розробки ПЗ за Хемфрі

Індивідуальний процес розробки ПЗ (PSP — Personal Software Process). PSP поділяється на стадії:

1) PSP0 — базовий процес. Вимоги до розробника: фіксація часу, витраченого на роботу над проектом; запис найдених дефектів та їх типів;
2) PSP1 — індивідуальний процес планування. Ця стадія повинна допомогти інженеру визначити зв'язок між розміром програми та часом її розробки. Вимоги: здатність оцінювати розмір задачі; систематичний підхід до опису результатів тестування; планування програмних завдань та розподіл їх по часу;
3) PSP2 — індивідуальний процес контролю якості. Вимоги: індивідуальна перевірка проекту та архітектури; перевірка коду;
4) PSP3 — циклічний індивідуальний процес. Включає: спосіб застосування PSP для кожного етапу; регресійне тестування — перевірка того, що тести, розроблені на попередніх ітераціях, успішно виконуються на усіх етапах.

Командний процес розробки ПЗ (TSP — Team Software Process). Завдання TSP:

1) збір самокерованої команди: узгодити мету; скласти свій план та процес; відслідковувати роботу;
2) демонстрація менеджерам, як треба керувати командами: мотивація, інструктаж, підтримання високої продуктивності команди;
3) прискорити просування по шкалі CMM;
4) забезпечити шляхи покращення професіоналізму;

Аналіз вимог до програмних продуктів

Результатом аналізу вимог є документ, який називається **специфікацією вимог до ПЗ** (SRS — Software Requirement Specification). Вимоги до ПЗ поділяють на *функціональні* та *експлуатаційні* [2].

Функціональні вимоги описують сервіси, які надає ПС, її поведінку у відповідних ситуаціях, реакцію на ті чи інші вхідні дані та дії, яка система дозволяє виконувати користувачам. Функціональні вимоги документуються у SRS. Бажаним є використання формальної мови при формулюванні функціональних вимог. Функціональна специфікація складається з трьох частин:

1) Опис зовнішнього інформаційного середовища, з яким буде взаємодіяти ПЗ. Мають бути описані усі канали вводу-виводу та всі інформаційні об’єкти, а також зв’язки між ними.
2) Визначення функцій ПЗ, визначених на множині станів інформаційного середовища. Вводяться позначення усіх функцій, визначаються їх вхідні дані та результати виконання, з вказівкою їх типів та всіх необхідних обмежень.
3) Опис надзвичайних ситуацій, які можуть виникнути при виконанні програми і реакцій на них, які має забезпечити виконуване ПЗ.

Експлуатаційні вимоги визначають характеристики ПЗ, які виявляються у процесі його використання:

1) правильність — функціонування у відповідності з технічним завданням; Це обов’язкова вимога для будь-якого ПЗ. На практиці мова може йти про певну ймовірність відсутності помилок;
2) універсальність — забезпечення правильної роботи для будь-яких допустимих даних і захист від неправильних даних;

3) надійність (завадостійкість) — забезпечення повної повторюваності результатів, тобто забезпечення їх правильності при різноманітних збоях. Для цього наприклад можуть створюватися контрольні точки, в яких проводиться збереження проміжних результатів;
4) можливість перевірки отриманих результатів. Для потрібно проводити документальну фіксацію вхідних даних та встановлених режимів;
5) точність результатів — забезпечення похибок не більших за задане граничне значення;
6) захищеність — забезпечення конфіденційності інформації. Ця вимога є надзвичайно актуальною для систем, у яких використовується інформація, яка складає державну чи комерційну таємницю;
7) програмна сумісність — можливість сумісної роботи з іншими програмами. Наприклад може вимагатися функціонування у певній ОС чи обмін даними з іншими програмами;
8) апаратна сумісність — можливість сумісного функціонування з певними видами обладнання;
9) ефективність — використання мінімального можливого обсягу ресурсів технічних засобів (зайнятість мікропроцесору, ОЗП, дискової пам’яті) та ОС;
10) здатність до адаптації — можливість швидкої модифікації з метою пристосування до змінених умов функціонування;
11) можливість паралельного використання кількома процесами. Для цього необхідно

створювати копії даних для кожного процесу;

У [1] Брауде використовує інший принцип класифікації вимог: C-вимоги (вимоги користувача) та D-вимоги (вимоги розробників ПЗ).

Вибір архітектури ПС

Архітектура ПС — це структура системи, яка містить елементи ПС, видимі ззовні властивості цих елементів і зв'язки між ними. Архітектура ПС складається із усіх важливих проектних рішень щодо структур програм та взаємодії між цими структурами. Проектні рішення забезпечують бажаний набір властивостей, які має реалізовувати ПЗ.

З точки зору кількості користувачів розрізняють однокористувацьку та багатокористувацьку (мережеву) архітектури. Мережеву архітектуру реалізують ПС, побудовані за принципом клієнт-сервер. В межах однокористувацької архітектури розрізняють:

1) програми — впорядкована послідовність формалізованих інструкцій для розв'язування задач на ЕОМ;
2) пакети програм — кілька окремих програм, які розв'язують задачі певної прикладної області. Прикладом є пакет математичних програм;

3) програмні комплекси — сукупність програм, які сумісно забезпечують розв'язання невеликого класу задач однієї прикладної області. При цьому для виконання задачі програмою диспетчером послідовно викликається кілька програм;
4) програмні системи — організований набір програм, які сумісно забезпечують розв'язання широкого класу задач.

Структури та формати даних

На етапі формування SRS ПЗ необхідно визначити структуру та формат даних, які використовуються у програмі. *Структура даних* — це множина елементів даних та зв'язків між ними. Розрізняють фізичну та логічну структуру даних.

Статичні структури даних являють собою структуровану множину простих типів. Розмір пам'яті, яка виділяється для таких даних, є постійним.

Напівстатичні структури даних мають змінну довжину, яка може змінюватися у певних межах, не перевищуючи граничне значення.

Динамічні структури даних не мають постійного розміру, тому пам'ять під окремі елементи таких структур виділяється у процесі виконання програми. Зв'язок між елементами встановлюється за допомогою вказівників.

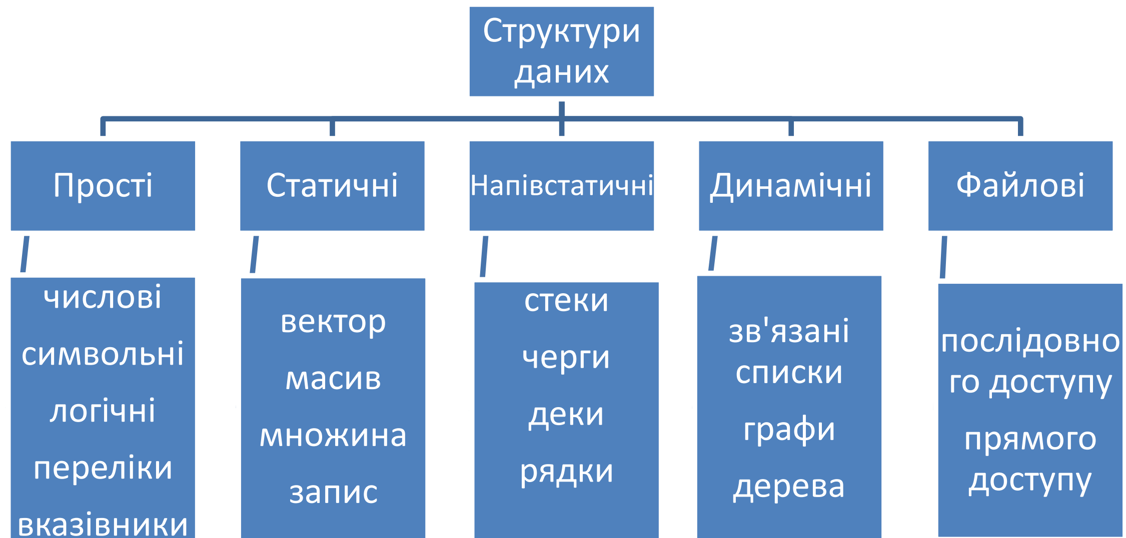

Рис. 5. Структури даних

Програмна платформа .NET Framework

.NET Framework — програмна платформа, першу версію якої випустила компанія Microsoft в 2002 році. *Основою платформи є загальномовне середовище виконання* **Common Language Runtime** (CLR), яке підходить для розробки програмних засобів на різних мовах програмування.

Іншою складовою компонентою платформи .NET є *загальна система типів* (**Common Type System**) або, скорочено, система CTS. В специфікації CTS наведено повний опис усіх можливих типів даних і програмних конструкцій, які підтримує виконуюче середовище CLR, того, вони можуть взаємодіяти і зображуватися у форматі метаданих .NET.

Будь-яка із визначених в CTS функціональних можливостей може не підтримуватися в окремій .NET-сумісній мові. Тому існує ще одна *загальномовна специфікація* (Common Language Specification (CLS), у якій описано тільки ту підмножину загальних типів і програмних конструкцій, які мають підтримувати усі без винятку .NET-сумісні мови.

Архітектура .NET Framework описана и опублікована в специфікації *Common Language Infrastructure* (CLI), затверджeній ISO и ECMA. В CLI описані типи даних .NET, формат метаданих про структуру програми, система виконання байт-коду, тощо.

Об'єктні класи .NET, доступні для усіх мов програмування, що підтримуються, містяться у бібліотеці **Framework Class Library** (FCL). До складу FCL входять класи:

- Windows Forms — засоби для створення настільних програм;
- ADO.NET — засоби для взаємодії та використання серверів баз даних;
- ASP.NET — засоби для створення клієнт-серверних Web-застосувань;
- Language Integrated Query (LINQ) — вбудована мова інтегрованих запитів;
- Windows Presentation Foundation — технологія створення та обробки ділової графіки,
- Windows Communication Foundation — технологія створення розподілених систем.

Ядро FCL називається **Base Class Library** (BCL).

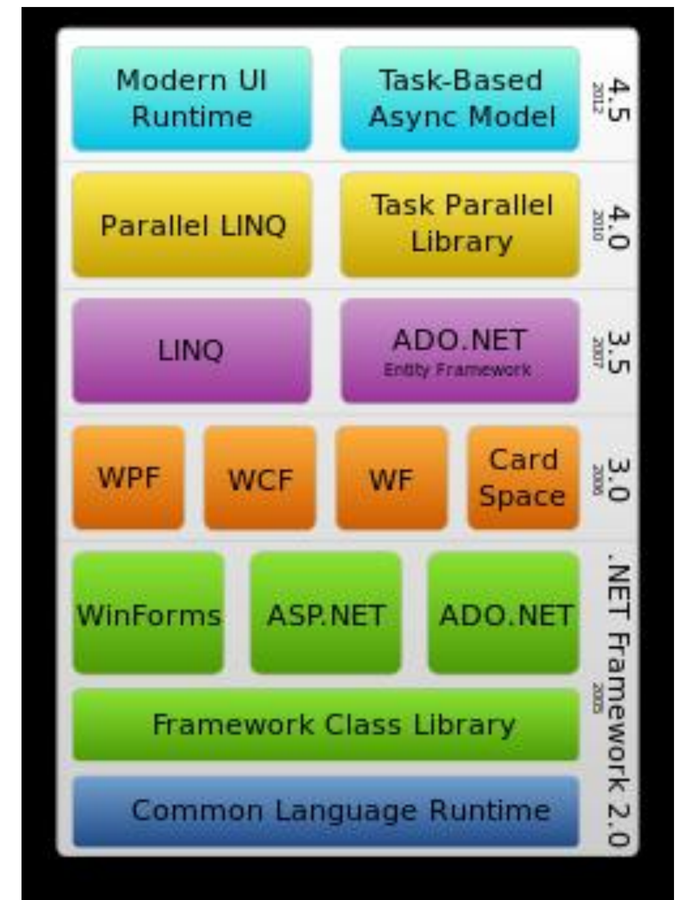

Рис. 6. Стек технологій .NET Framework

Створення та виконання ПЗ у середовищі .NET

1. Спочатку пишеться програма з використанням .NET-сумісної мови програмування.
2. Програма компілюється у MSIL-код, який зберігається у збірці (рис 7.).

Рис. 7. Компіляція коду у збірку

3. При першому запуску цього коду (в результаті запуску виконуваного файлу або при виклику із іншого коду) він у першу чергу компілюється у рідний код з використанням JIT-компілятора (just in time compiler) (рис. 8).

Рис. 8. Компіляція збірки у рідний код

4. Після цього рідний код виконується у контексті CLR (Common Language Runtime) разом з усіма іншими програмами чи процесами, як це наведено на рис. 9 або транслюється за допомогою утиліти NGen.exe у виконуваний код для цільового процесора.

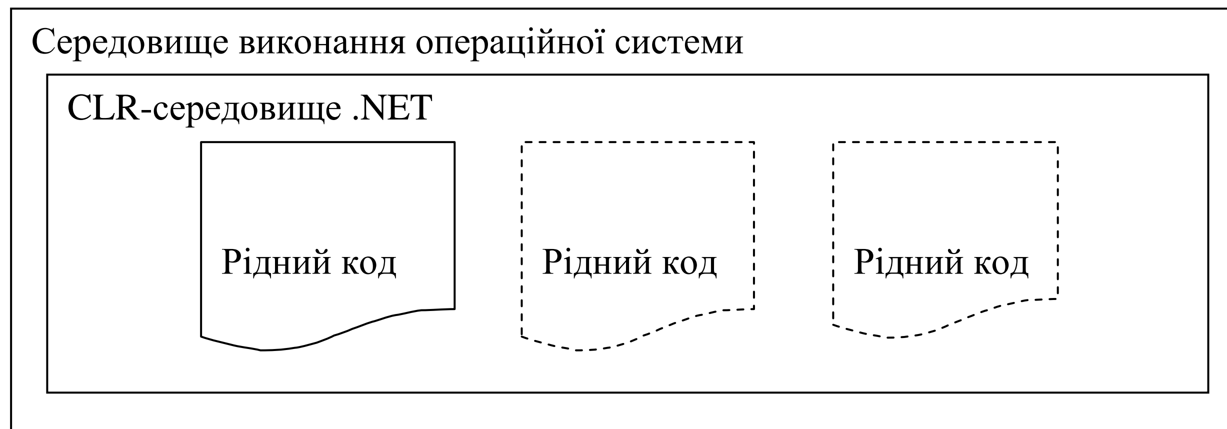

Рис. 9. Виконання рідного коду в CLR.

Мова програмування C#

Приклад першої консольної програми

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Hello, world");
        Console.ReadKey();
    }
}
```

Типи даних мови C#

Ієрархія основних системних типів CTS наведена на рис. 10.

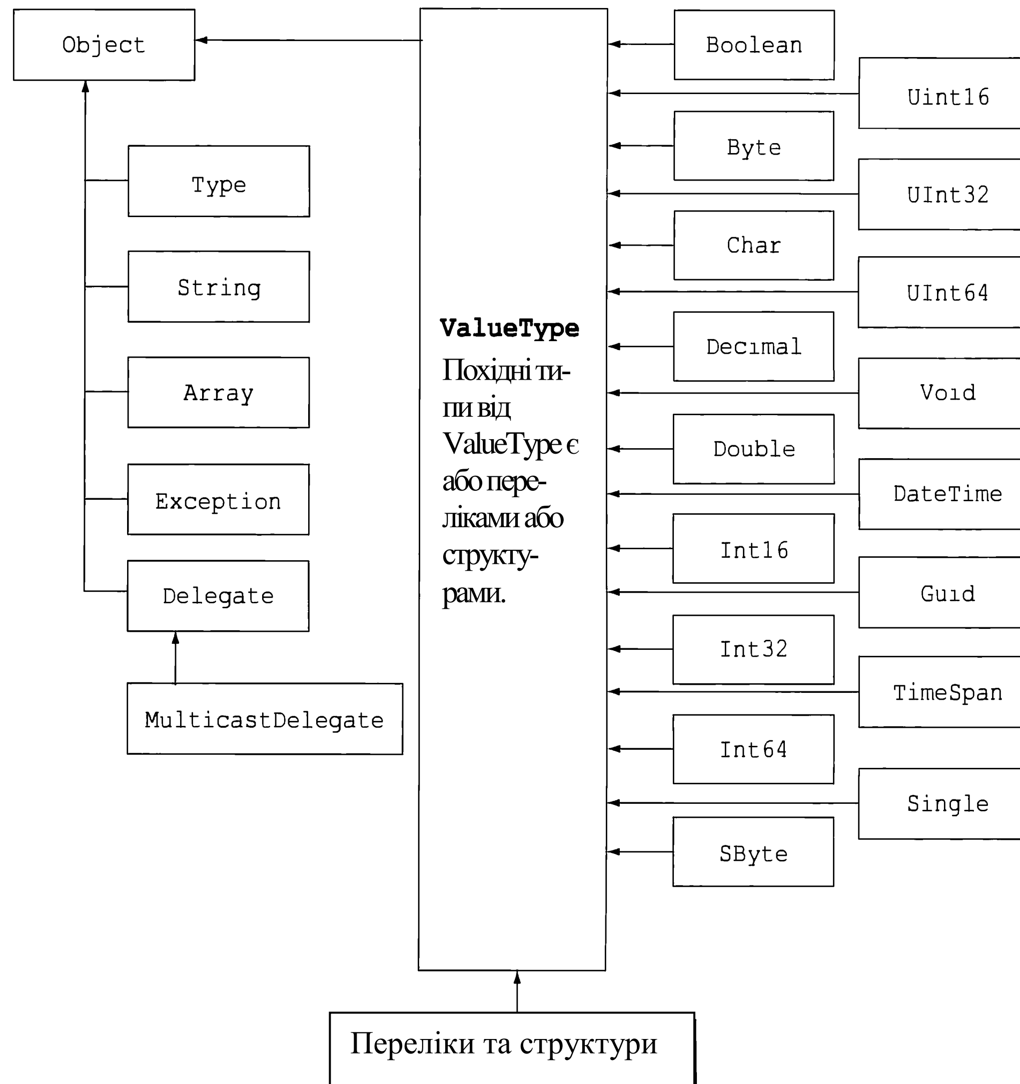

Рис. 10. Ієрархія системних типів

Цілочислові типи

Таблиця 1. Характеристики цілочислових типів мови C#

Тип	Діапазон	Розмір
sbyte	-128 … 127	8-розряде ціле число зі знаком
byte	0 … 255	8-розряде ціле число без знаку
char	Від U+0000 до U+ffff	16-розрядний символ Юнікоду
short	-32 768 … 32 767	16-розрядне ціле число зі знаком
ushort	0 … 65 535	16-розрядне ціле число без знаку
int	-2 147 483 648 … 2 147 483 647	32-розрядне ціле число зі знаком
uint	0 … 4 294 967 295	32-розрядне ціле число без знаку
long	-2^{63} … $2^{63}-1$	64-розрядне ціле число зі знаком
ulong	0 … $2^{64}-1$	64-розрядне ціле число без знаку

Опис змінних, зчитування та перетворення диних

Приклад зчитування даних з консолі та перетворення типів:

```
Console.Write("Input first number; a = ");
int a = Convert.ToInt32(Console.ReadLine());
Console.Write("Input second number; b = ");
int b = int.Parse(Console.ReadLine());
int c = a + b;
Console.WriteLine("{0} + {1} = {2}", a, b, c);
```

Приклад перетворення типів та форматування вводу-виводу

```
static void Main(string[] args)
```

```
{
  int x = 511;
  byte y = (byte)x;
  sbyte z = (sbyte)y;
  Console.WriteLine("x = {0}, y={1}, z = {2}", x, y, z);
  Console.Write("Input a: ");
  int a = Convert.ToInt32(Console.ReadLine());
  Console.Write("Input b: ");
  int b = Convert.ToInt32(Console.ReadLine());
  Console.Write("{0} divided into {1} is {2}\"", a, b, a/b);
  Console.ReadKey();
}
```

Робота з великими числами

У просторі імен `System.Numerics` визначена структура `BigInteger`, яка призначена для обробки великих цілих чисел. Для того, щоб отримати доступ до простору імен `System.Numerics`, потрібно додати до проекту посилання на збірку `System.Numerics.dll`.

Приклад.

```
static void UseBigInteger()
{
    Console.WriteLine("=> Use BigInteger:");
    BigInteger biggy =
        BigInteger.Parse("9999999999999999999999999999999999999999999999");
    Console.WriteLine("Value of biggy is {0}", biggy);
    Console.WriteLine("Is biggy an even value?: {0}", biggy.IsEven);
```

```
        Console.WriteLine("Is biggy a power of two?: {0}", biggy.IsPowerOfTwo);
        BigInteger reallyBig = biggy *
            BigInteger.Parse("8888888888888888888888888688888888888888888888888");
        Console.WriteLine("Value of reallyBig is {0}", reallyBig);
    }
```

Умовні оператори

Оператор if

Скорочена форма:

```
if (вираз)
   оператор;
```

Повна форма:

```
if (вираз)
   оператор;
else
   оператор;
```

Приклад 1.

```
if(x = 5);  //потенційна помилка, бо x = 5 – присвоєння
if(x == 5); //правильний варіант
```

Приклад 2. Знайти максимальне з трьох чисел x, y, z

```
int x, y , z, max;
```

```
if(x > y)
   if(x > z)
      max = x;
   else
      max = z;
else
   if(y>z) max = y;
   else
    max = z;
```

Функціональний if (тернарна операція)

Синтаксис:

```
умова ? вираз1 : вираз2
```

Якщо умова істинна, то результатом є значення 1-го виразу, інакше — другого виразу.

Приклади. Обчислення модуля та максимуму

```
abs =  x >= 0 ? x : -x;
max = x > y ? x : y;
```

Оператор розгалуження switch

Оператор `switch` дозволяє програмувати вибір із набору альтернатив. Синтаксис:

```
switch (вираз)
{
   case константа1:
```

```
        послідовність операторів
        break;
    case константа2:
        послідовність операторів
        break;
    .
    .
    .
    default:
        послідовність операторів
        break;
}
```

Кожній альтернативі відповідає своя унікальна константа. `Вираз` в операторі `switch` може бути цілочислового типу (`char`, `byte`, `short` і т.п.), типу переліку або типу `string`. Код для обробки кожної альтернативи *повинен завершуватися викликом оператора* `break`.

Приклад 1. Вивід інформації про парність цифр.

```
Console.Write("задайте цифру ");
```

```
char x = Convert.ToChar(Console.ReadLine());
switch(x)
{
   case '0': case '2': case '4': case '6': case '8':
      //для кожної альтернативи свій оператор case
      Console.Write("парна цифра");
      break;
   case '1': case '3': case '5': case '7': case '9':
      Console.Write("непарна цифра");
      break;
   default:
     //Необовя'язковий оператор. Виконується, якщо жодна альтернатива не є вірною
       Console.Write("не є цифрою");
       break;
 };
```

У попередньому прикладі послідовність гілок `case` дає змогу реалізувати *"ефект провалювання"*, характерний для `C++` чи `Java`.

Приклад 2. Використання рядків у операторі `switch` та використання оператора `goto` (у якості міток використовуються гілки `case` чи `default`).

```
Console.Write("Input a string: ");
string str = Console.ReadLine();
switch(str.ToLower())
{
    case "ab":
```

```
            Console.WriteLine("string in lower case = ab");
           goto case "bc"; // goto default
          case "bc":
               Console.WriteLine("string ... = bc");
                   break;
            default:
                   Console.WriteLine("another string");
                   break;
     }
```

Циклічні оператори

Цикл з лічильником for

```
for (ініціалізація; умова; інкремент) оператор
```

Приклад1. Обчислення суми додатних елементів одновимірного *n*-елементного масиву.

```
float s = 0;
for(int i = 0; i < n; i++)
    if(a[i] > 0) s = s + a[i];
```

Приклад 2. Обчислення найменшого значення функції $y = e^{|2x - 1|}\sin 3x$ на відрізку $[-2;3]$ з кроком $\varepsilon = 0{,}001$.

```
double min = exp(abs(-5))*sin(-6);
for(double x = -2; x <= 3; x += 0.001)
{
```

```
        double t = Math.Exp(Math.Abs(2*x-1))*Math.Sin(3*x);
        if(t < min)
            min = t;
    }
```

Цикли while (з передумовою), do while (з післяумовою)

Цикл з передумовою

```
while(вираз) оператор
```

Цикл з післяумовою

```
do оператор while(умова)
```

Приклад. Обчислити кількість від’ємних елементів та добутку елементів одновимірного масиву із *n* елементів, які потрапляють у проміжок [b;c].

```
float P = 1;//добуток
int k = 0;//кількість
int i = 0;
do{
    if(a[i] < 0)
        k++;
    if(b <= a[i] && a[i] <= c)
```

```
            P*=a[i];
        i++;
} while(i<=n-1);
```

Переліки

Переліки (enum) використовуються для створення набору символічних імен, яким відповідають деякі заздалегідь визначені числові значення. По замовчуванню, для збереження значень переліку використовується тип System.Int32, причому першому елементу переліку відповідає число 0, кожному наступному — на одиницю більше число за, яке ставиться у відповідність попередньому

```
class MyProgram
{
  enum Direction : byte // базовий тип
  {
      North = 2, South, East, West // значення 1-го елемента не 0, а 2
  }
  static void Main(string[] args)
  {
    Direction x = Direction.East;
    Direction y = (Direction)3;
    Console.WriteLine("x={0}, byte equivalent={1},\ny={2}",x, (byte)x, y);
  }
}
```

Структури

Структура — тип даних, що відноситься до типів значень (value types), змінні якого можуть містити довільну кількість полів різного типу та, можливо, методів для обробки цих полів. Можна вважати [9], що у C# структури є "полегшеними класами", оскільки вони підтримують інкапсуляцію, але не підтримують наслідування.

```
class MyProgram
{
    enum Direction : byte
    {
        North = 2, South, East, West
    }
    struct Route
    {
        public Direction direct;
        public double distance;
        public override string ToString()
        {return string.Format("({0}; {1})", direct, distance);}
    }
    static void Main(string[] args)
    {
        Route a;
        a.direct = Direction.north; a.distance = 30.5;
        Route b = a;
        b.distance = 20;
        Console.WriteLine(a + "\n" + b);
    }
}
```

Приклад. Написати програму для зображення рухомої фігури.

```
using System;
using System.Drawing;
using System.Windows.Forms;

public partial class Form1 : Form
{
    private enum Direction
    {
        North,
        South,
        East,
        West
    };

    struct Moving
    {
        public Direction Trend;
        public int Distance;
    }
    Point top = new Point(0, 0);
    public Form1()
    {
        InitializeComponent();
    }

    private void runButton_Click(object sender, EventArgs e)
    {
```

```
    if (directionComboBox.SelectedIndex >= 0)
    {
        Moving m = new Moving()
        {
            Distance = int.Parse(distanceTextBox.Text),
            Trend = (Direction)directionComboBox.SelectedIndex
        };
        switch (m.Trend)
        {
            case Direction.North:
                top.Y -= m.Distance;
                break;
            case Direction.South:
                top.Y += m.Distance;
                break;
            case Direction.East:
                top.X += m.Distance;
                break;
            case Direction.West:
                top.X -= m.Distance;
                break;
        }
        canvasPanel.Invalidate();
    }
}

private void canvasPanel_Paint(object sender, PaintEventArgs e)
{
```

```
            Graphics g = e.Graphics;
            g.DrawEllipse(Pens.BlueViolet, top.X, top.Y, 100, 100);
        }
}
```

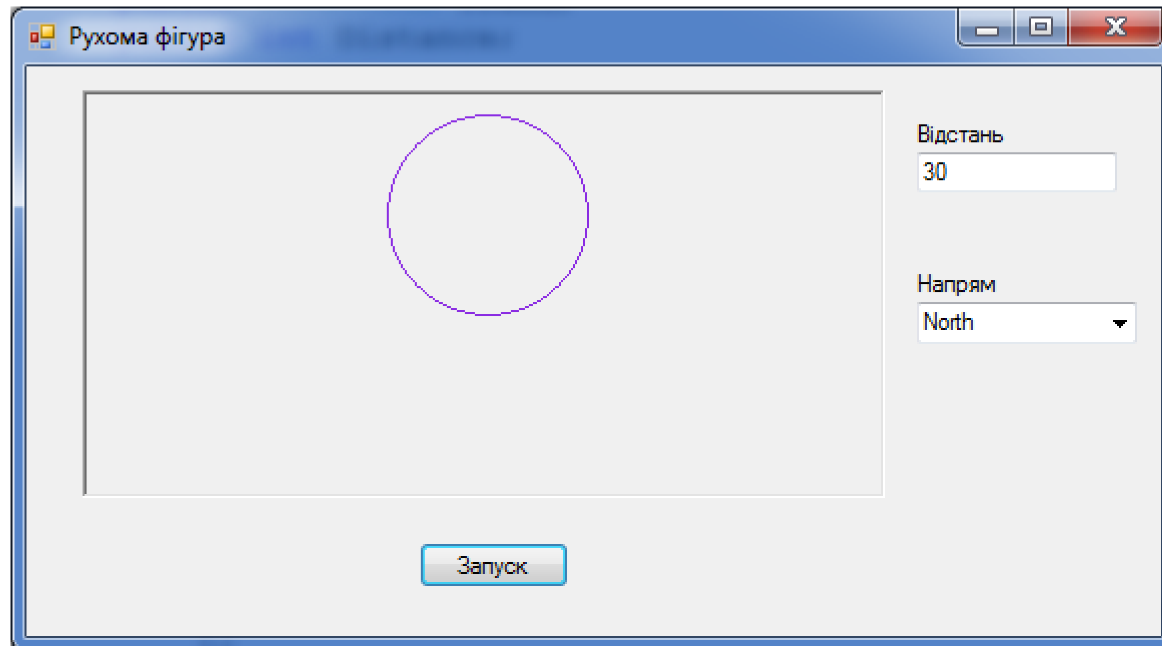

Рис. 11. Вигляд вікна програми

Область видимості змінних

Область видимості змінної охоплює блок коду, в якому описана змінна, а також усі блоки, безпосередньо у нього вкладені. У вкладеному блоці не допускається повторний опис змінних

(опис змінних усередині методів класу чи у підкласах класу, які мають однакове ім’я з деякими ідентифікаторами, описаними у класі, є допустимим).

Приклад 1.

```
public static void Main()
{
    int i = 0;
    {
        char i = 'a';   // CS0136, hides int i
    }
    i++;
}
```

Не можна виконувати ініціалізацію змінних у вкладених блоках коду, у випадку відсутності гарантії їх виконання.

Приклад 2.

```
static void Main(string[] args)
{
    int i;
    string text;
    for (i = 0; i < 10; i++)
    {
        text = "Line " + Convert.ToString(i);
        Console.WriteLine("{0}", text);
    }
    Console.WriteLine("Last text output in loop: {0}", text); // Помилка
}
```

Масиви

Масив — *це фіксований за розміром набір однотипних елементів*, *доступ* до яких можливий за допомогою *імені масиву* та *числових індексів*.

Одновимірні масиви

Елементи одновимірних масивів індексуються за допомогою одного числового невід'ємного індексу.

Оголошення одновимірних масивів:

```
тип[] ім'я_масиву;
```

Для *створення масиву* використовується оператор `new`

```
масив = new тип[розмір];
```

Ініціалізація масиву

```
тип[] ім'я_масиву = {знач1, знач2, …, значN}
```

Властивість `Length` показує кількість елементів масиву. Допустимими значеннями індексів елементів масиву є цілі числа від 0 до `Length - 1`.

Клас System.Array

Клас `System.Array` є базовим класом для усіх масивів CLR. Він надає засоби для створення, зміни, пошуку та сортування масивів.

В табл. 2 наведені деякі методи класу `Array`.

Таблиця 2. Основні методи класу `Array`

Методи	**Опис**
`Clear`	*Статичний метод*, який робить діапазон елементів рівним `0`, `false` чи `null` в залежності від типу елементів.
`Copy`	*Статичний метод*, який копіює діапазон елементів 1-го масиву, починаючи із заданого індексу, і записує їх в 2-й масив, починаючи із заданого індексу.
`CopyTo`	Копіює усі елементи поточного одновимірний масиву у заданий одновимірний масив, починаючи із заданого індексу.
`IndexOf`	*Статичний метод*, який виконує пошук у заданому масиві першого входження заданого елемента.
`Rank`	Властивість, яка повертає інформацію про кількість вимірів масиву.
`Resize`	*Статичний метод*, який змінює розмір заданого масиву на вказаний новий розмір

`Reverse`	*Статичний метод*, який змінює порядок елементів у заданому одновимірному масиві на обернений.
`Sort`	*Статичний метод*, який використовується для сортування масиву, базовий тип елементів якого забезпечує підтримку інтерфейсу `IComparer`

Приклад 1. Демонстрація методів класу `Array`

```
var a = new[] { 2, 6, 1, 3, 15, 2 };
Console.WriteLine("a: {0}", string.Join("; ", a));
Array.Clear(a, 1, 3);
int[] b = new int[a.Length];
Console.WriteLine("b: {0}", string.Join("; ", b));
Array.Resize(ref b, b.Length + 3);
Console.WriteLine("a: {0}", string.Join("; ", a));
a.CopyTo(b, 2);
Console.WriteLine("b: {0}", string.Join("; ", b));
Console.WriteLine("Index of 15 in a: {0}", Array.IndexOf(a, 15));
```

Для ітерації по усім елементам масиву використовується цикл `foreach`. У циклі `foreach` *допускається лише зчитування* значень елементів. Для зміни значень елементів потрібно використовувати індексацію.

Приклад 2. Заповнити масив заданого розміру випадковими числами з проміжку [0, 100). Вивести початковий та відсортований масиви. Знайти найбільший елемент масиву та середнє арифметичне елементів масиву.

```
static void Main(string[] args)
{
    Console.Write("Input array size ");
    int arraySize = Convert.ToInt32(Console.ReadLine());
    int[] a = new int[arraySize];
    Random rnd = new Random();
    Console.WriteLine("Initial array:");
    for (int i = 0; i < a.Length; i++)
        Console.Write("{0} ", a[i] = rnd.Next(100));
    Array.Sort(a);
    Console.WriteLine("\nSorted array:");
    foreach (int x in a)
        Console.Write(x + " ");
   Console.WriteLine("\nMax item: {0}\n Mean value: {2}", a.Max(), a.Average());
}
```

Приклад 3. Вивести елементи числового одновимірного масиву, які потрапляють у заданий діапазон і мають найбільшу суму цифр.

```
using System;

using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;

namespace WindowsFormsApplication8
{
    public partial class Form1 : Form
```

```
{
    public Form1()
    {
        InitializeComponent();
    }

    private void addButton_Click(object sender, EventArgs e)
    {
        int number;
        if(int.TryParse(valueTextBox.Text, out number))
            arrayListBox.Items.Add(valueTextBox.Text);
    }

    private void UpperNumericUpDown_ValueChanged(object sender, EventArgs e)
    {
        if (lowerNumericUpDown.Value > upperNumericUpDown.Value)
            lowerNumericUpDown.Value = upperNumericUpDown.Value;
    }

    int SumOfDigits(int n)
    {
        int s = 0;
        while (n > 0)
        {
            s += (n%10);
            n = n/10;
        }
        return s;
```

```
        }

        private void runButton_Click(object sender, EventArgs e)
        {
            int[] array = new int[arrayListBox.Items.Count];
            int maxSum = int.MinValue;
            List<int> numbers = new List<int>();
            for (int i = 0; i < array.Length; i++)
            {
                array[i] = int.Parse(arrayListBox.Items[i].ToString());
              if (array[i] >= lowerNumericUpDown.Value && array[i] <= upperNumericUpDown.Value)
                {
                    int curSum = SumOfDigits(array[i]);
                    if(maxSum == curSum)
                        numbers.Add(array[i]);
                    if (curSum > maxSum)
                    {
                        maxSum = curSum;
                        numbers.Clear();
                        numbers.Add(array[i]);
                    }
                }
            }
            MessageBox.Show("Result: " + string.Join("; ", numbers));
        }
    }
}
```

Вигляд вікна програми наведено на рис. 11

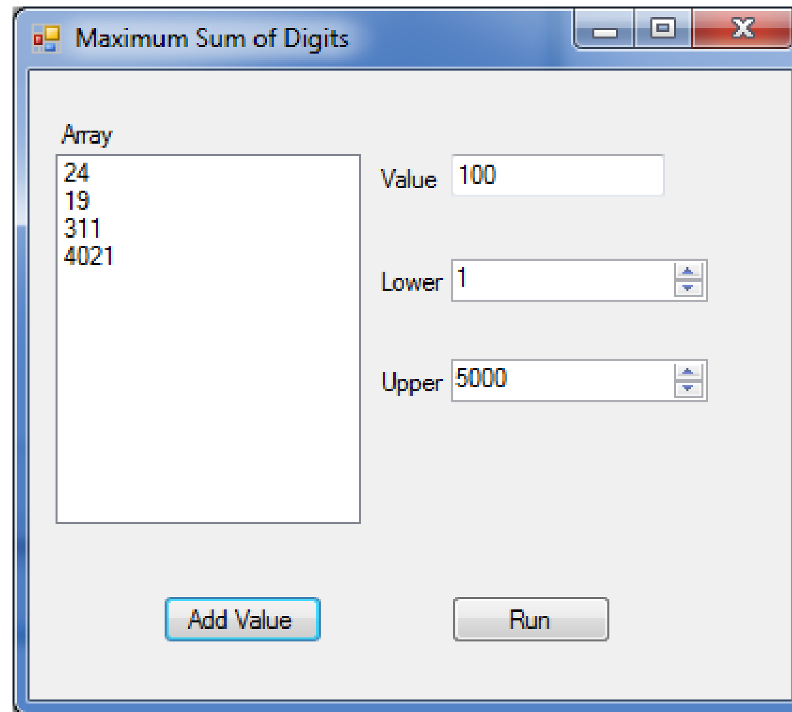

Рис. 11. Вигляд форми

Присвоєння масивів

```
int[] a = {10, 5, 8};
int[] b = a;
b[0] = 1;
foreach (int x in a)
    Console.Write("{0} ", x);
```

Багатовимірні масиви

Опис

```
double[,] hillHeight = new double[3,4];
double[,] hillHeight = {{1, 2, 3, 4}, {2, 3, 4, 5}, {3, 4, 5, 6}};
```

Доступ до елементів

```
hillHeight[2,1]
```

Приклад.

```
int [,] a = new int[2,3]{{11,2,3},{4444,555,6}};
for(int i = 0; i < a.GetLength(0); i++)
{
    for (int j = 0; j < a.GetLength(1); j++)
        Console.Write("{0,-10}", a[i, j]);
    Console.WriteLine();
}
int min = a.Min();// помилка
int min = a[0, 0];
foreach (int x in a)
    if (x < min)
        min = x;
```

Копіювання масивів

```
int[,] a = { { 4, 3, 5 }, { 1, 6, 2 } };
int[,] b = (int[,])a.Clone();
```

або

```
int[,] b = new int[2,3];
Array.Copy(a, b, a.Length);
```

Спорожнення (очистка)

```
int[,] a = { { 4, 3, 5 }, { 1, 6, 2 } };
Array.Clear(a, 0, a.Length);
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
        Console.Write(a[i, j] + " ");
    Console.WriteLine();
}
```

Приклад. Виділити кольором елементи таблиці, які більші за суму усіх інших елементів свого стовпчика та більші за суму усіх інших елементів свого рядка.

Ступінчасті (невирівняні) масиви

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[3];
jaggedArray[1] = new int[5];
jaggedArray[2] = new int[2];
foreach(int[] x in jaggedArray)
    foreach(int y in)
    // код
```

Функції та процедури

Функції та процедури в C# є різновидами методів і завжди описуються у класах. Опис функції в C# починається із вказівки *типу результату*, який повертає функція, після чого йде *ім'я функції* та опис її *параметрів*.

Якщо функція *не повертає* у викликаючу програму ніяке значення (тобто, є процедурою), то для типу результату використовується ключове слово `void`.

Для повернення у викликаючу програму результату виклику функції використовується оператор `return`, після якого вказується значення функції. У процедурах оператор `return` використовується для завершення роботи процедури. У функціях *кожна гілка коду має обов'язково повертати значення*:

```
static double GetVal(double checkVal)
{
    if (checkVal < 5)
        return 4.7;  // помилка
}
```

Приклад. Функція для знаходження суми додатних елементів числового масиву:

```
static double SumOfPositive(double[] a)
{
    double s = 0;
```

```
        foreach (double x in a)
            if (x > 0)
                s += x;
        return s;
    }
    static void Main(string[] args)
    {
        double[] a = {-1, 2, 5, -2, 3};
        Console.WriteLine(SumOfPositive(a));
    }
```

Передача параметрів

Значення, яке передається методу при його виклику, називається *аргументом* методу або *фактичним параметром*. Змінна, яка через яку передається аргумент, називається *формальним параметром*. Синтаксис опису параметрів такий самий, як і у змінних.

По замовчуванню, у C# параметри у функції передаються *по значенню*, тобто викликаюча функція отримує *копію аргументу* і зміна значень формальних параметрів у процесі виконання функції не призводить до зміни значень фактичних параметрів.

Приклад 1.

```
    static void Foo(int a)
    {
        a++;
```

```
}
static void Main(string[] args)
{
    int a = 10;
    Foo(a);
    Console.WriteLine(a);
}
```

Приклад 2.

```
static int[] b = {100, 100};
static void Incr(int[] a)
{
    for (int i = 0; i < a.Length; i++)
    {
        a[i]++;
    }
    a = b;
}

static void Main(string[] args)
{
    int[] a = { 3, 6, 1 };
    Console.WriteLine("Before: " + string.Join("; ", a));
    Incr(a);
    Console.WriteLine("After: " + string.Join("; ", a));
}
```

Якщо перед параметром методу йде ключове слово `ref`, то параметр передається *за посиланням*. Це означає, що усі зміни значення формального параметра зумовлюють зміни значення фактичного параметра. Якщо метод містить параметри, які передаються за посиланням, то при виклику цього методу перед відповідними фактичними параметрами *обов'язково потрібно вказувати* модифікатор `ref`.

Приклад 3.

```
static int[] b = {100, 100};
static void Set(ref int[] a)
{
    a = b;
}

static void Main(string[] args)
{
    int[] a = { 3, 6, 1 };
    Console.WriteLine("Before: " + string.Join("; ", a));
    Set(ref a);
    Console.WriteLine("After: " + string.Join("; ", a));
}
```

Якщо перед параметром методу йде ключове слово `out`, то параметр також передається *за посиланням*. Відмінність від параметрів, описаних за допомогою `ref`, полягає у тому, що передача у метод неініціалізованих параметрів можлива, якщо вони описані з використанням

модифікатора `out`, і заборонена, якщо перед параметром стоїть `ref`. Якщо метод містить параметри, які передаються за посиланням, то при виклику цього методу перед відповідними фактичними параметрами *обов'язково потрібно вказувати* модифікатор `ref`. Крім того, параметрам, описаним із використанням `out`, обов'язково має бути присвоєне якесь значення у тілі методу. У випадку `ref` таке присвоювання не є обов'язковим.

Приклад 4. Написати функцію для знаходження найбільшого елемента матриці та визначення його координат (номерів рядка та стовпця).

```
static double MaxValue(double[,] a, out int maxRow, out int maxCol)
{
     maxRow = maxCol = 0;
     for (int i = 0; i < a.GetLength(0); i++)
         for (int j = 0; j < a.GetLength(1); j++ )
             if (a[i, j] > a[maxRow, maxCol])
             {
                 maxRow = i;
                 maxCol = j;
             }
     return a[maxRow, maxCol];
 }
 static void Main(string[] args)
 {
     double[,] a = { { 1, 5, 2 }, { 6, 4, 5 } };
```

```
        int maxRow, maxCol;
        double max = MaxValue(a, out maxRow, out maxCol);
        Console.WriteLine("max={0}, maxRow={1}, maxCol={2}", max, maxRow, maxCol);
    }
```

Масиви параметрів

Масиви параметрів дають змогу викликати функцію із різною кількістю параметрів і визначаються за допомогою ключового слова `params`. Якщо масив параметрів присутній у описі функції, то він має *бути останнім у списку параметрів*.

Приклад 5. Процедура для виводу на консоль змінної кількості аргументів

```
static void Write(params string[] values)
{
    foreach (string str in values)
        Console.Write(str);
}
static void Main(string[] args)
{
    Write("first", "second", "third");
}
static void Write(params object[] values)
{
    foreach (object x in values)
        Console.Write(x);
}
static void Main(string[] args)
```

```
{
    Write("x=", 2, " y=", 4.5);
 }
```

Перезавантаження функцій

Можна описувати кілька функцій, які мають однакові імена, якщо при цьому вони мають різні сигнатури (ім’я та параметри функції, але не тип її результату).

```
static int MaxValue(int[] intArray)
{
   int maxVal = intArray[0];
   for (int i = 1; i < intArray.Length; i++)
   {
      if (intArray[i] > maxVal)
         maxVal = intArray[i];
   }
   return maxVal;
}

static double MaxValue(double[] doubleArray)
{
   double maxVal = doubleArray[0];
   for (int i = 1; i < doubleArray.Length; i++)
   {
      if (doubleArray[i] > maxVal)
      maxVal = doubleArray[i];
```

```
        }
        return maxVal;
    }
```

Також, починаючи з C# 4.0, можна використовувати *значення параметрів за замовчуванням*. При описі параметра методу можна після знаку "=" вказати значення, яке буде приймати цей параметр, якщо його значення явно не вказано при виклику методу. Параметри, які мають значення за замовчуванням, мають бути *розташовані у кінці списку параметрів*.

Приклад.

```
static int Incr(int value, int delta = 1)
{
    return value + delta;
}

static void Main(string[] args)
{
    Console.WriteLine("Incr(3, 5): {0}", Incr(3, 5));
    Console.WriteLine("Incr(3): {0}", Incr(3));
}
```

Робота з рядками

Клас `System.String` підтримує концепцію символьного рядка та надає значну кількість методів та властивостей для комфортної роботи із рядками. В C# ключове слово `string` є си-

нонімом до `System.String`. Змінні типу `String` являють собою послідовність, яка складається з нуля або більшої кількості символів Unicode. Тип `String` належить до типів посилань. Проте *оператори рівності* (== та `!=`) *визначені для об'єктів типу* `String`, а не для посилань.

Приклад 1.

```
string a = "hello";
string b = "h";
b += "ello"; // Append to contents of 'b'
Console.WriteLine(a == b);
Console.WriteLine((object)a == (object)b);
```

Рядки у C# є *незмінними*: вміст об'єкта-рядка *неможливо змінити після створення* об'єкта, хоча у зв'язку із особливістю синтаксису часто виникає ілюзія таких змін. Наприклад, при написанні наступного коду компілятор насправді створює новий об'єкт.

```
string b = "h";
b += "ello";
```

Оператор `[]` використовується для зчитування окремих символів рядка. Можливість зміни окремих символів рядка відсутня!

Рядкові літерали мають тип `String` і можуть бути написані в двох формах: в лапках і з використанням `@`. Літерали із точних рядків (дослівні літерали) починаються із знаку `@` і

мають ту перевагу, що для них escape-послідовності не опрацьовуються. Завдяки цьому можна зручно записувати, наприклад, шлях до файлів:

`@"c:\Docs\Source\a.txt"` замість `"c:\\Docs\\Source\\a.txt"`.

У класі `System.String` передбачено кілька конструкторів, які, зокрема, дозволяють ініціалізовувати рядок символьним масивом або заданою кількістю копій наперед визначеного символу. Над рядками можна виконувати операцію конкатенації:

```
string s1 = new string('s', 5);
var array = new[] {':', ' ', 's', 'y', 'm', 'b', 'o', 'l', ' ',
    ' ', 'c', 'o', 'p', 'i', 'e', 's'};
string s2 = new string(array);
s1 += s2;
Console.WriteLine(s1);
```

Деякі методи класу `System.String` наведені в табл. 3.

Таблиця 3. Основні методи класу `String`

Метод	**Опис**
`Length`	*Властивість*, яка повертає поточну довжину рядка.
`Compare`	*Статичний метод*, який дає змогу порівнювати рядки.
`Contains`	Перевірка входження підрядка у поточний рядок.

`Equals`	Метод, який дозволяє перевірити, чи містять два рядки ідентичні символьні дані
`Format`	*Статичний метод* для форматування рядків.
`Insert`	Метод, який повертає рядок, який отримується після вставки у вказану позицію поточного рядка іншого рядка.
`PadLeft`, `PadRight`	Методи, які дозволяють доповнювати рядок до потрібної довжини, використовуючи заданий символ.
`Remove`	Метод, який дає змогу отримати рядок, що отримується з поточного шляхом видалення заданої кількості символів починаючи із заданої позиції.
`Replace`	Метод, який дає змогу отримати рядок, що отримується з поточного шляхом заміни усіх входжень заданого підрядка на інший рядок.
`Split`	Метод, який розбиває рядок на масив підрядків, які відокремлюються у початковому рядку елементами заданого масиву розділювачів.
`ToUpper`, `ToLower`	Метод, який повертає рядок, що отримується із поточного перетворенням його символів до верхнього (нижнього) регістру.
`Trim`	Метод, який повертає рядок, що отримується із видаленням усіх заданих символів з обох кінців поточного рядка.

Використання методів табл. 3 та кількох інших методи продемонстровано у наступних прикладах.

Пошук слів, які закінчуються на заданий рядок

```
class Program
{
    static bool EndWithE(string str)
    {
        return str.EndsWith("e");
    }
    static void Main(string[] args)
    {
        string[] strArray = { "one", "two", "three", "four", "five" };
        string[] eArray = Array.FindAll(strArray, EndWithE);
        foreach (var x in eArray)
            Console.WriteLine(x);
    }
}
```

Розбиття рядка

```
        string str = Console.ReadLine();
        string[] words = str.Split(' ', '.', ',', ';', ':', '!', '?');
        foreach (string word in words)
            Console.WriteLine(word.Length > 0 ? word : "empty");
```

Конкатенація

```
Object[] objArray = { "one", 2, "three", "four", 5 };
string[] strArray = { "one", "two", "three", "four", "five" };
string str1 = String.Concat(objArray);
string str2 = String.Join("; ", strArray);
Console.WriteLine(str1 + "\n" + str2);
```

Перевірка входження

```
string s1 = "To Be or Not To Be";
string s2 = "Be";
bool b = s1.Contains(s2);
Console.WriteLine("Is the string, s2, in the string, s1?: {0}", b);
if (b)
{
  int ID1 = s1.IndexOf(s2);
  int ID2 = s1.LastIndexOf(s2);
  Console.WriteLine("Index of the first occurence \"{0}\" in \"{1}\" is {2}", s2, s1, ID1);
  Console.WriteLine("Index of the last occurence \"{0}\" in \"{1}\" is {2}",s2, s1, ID2);
}
```

Порівняння рядків

```
Console.WriteLine(string.Compare("S", "s"));              // 1
Console.WriteLine(string.Compare("S", "s", true));        // 0
Console.WriteLine(string.Compare("S", "s", StringComparison.Ordinal));//-32
```

Якщо виникає потреба інтенсивно змінювати вміст рядка, то для цього доцільно використовувати об’єкти класу `StringBuilder` із простору імен `System.Text`.

```
static void Main()
{
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    // Create a string composed of numbers 0 - 9
    for (int i = 0; i < 10; i++)
    {
        sb.Append(i.ToString());
    }
    System.Console.WriteLine(sb);  // displays 0123456789
    // Copy one character of the string (not possible with a String)
    sb[0] = sb[9];
    System.Console.WriteLine(sb);  // displays 9123456789
}
```

Для об’єктів класу `System.Text` не передбачено використання операції конкатенації. Тому для добавлення до рядка потрібно застосовувати метод `Append`. Також доступними є методи `Insert`, `Remove` та `Replace`, синтаксис та використання яких подібні до методів класу `System.String` із тією самою назвою.

Робота з датою та часом

Для обробки даних про дату, час та тривалість використовуються типи `System.DateTime` та `System.TimeSpan`.

Приклад 1. Виведення тривалості процесу сортування масиву

```
double[] dArray = new double[1 << 24];
Random rndObject = new Random();
for (int i = 0; i < dArray.Length; i++)
    dArray[i] = rndObject.NextDouble();
DateTime startTime = DateTime.Now;
Console.WriteLine("Start time: {0}", startTime.TimeOfDay);
Array.Sort(dArray);
DateTime finishTime = DateTime.Now;
Console.WriteLine("Finish time: {0}", finishTime.TimeOfDay);
TimeSpan duration = finishTime.TimeOfDay - startTime.TimeOfDay;
Console.WriteLine("Duration:{0}\nFull seconds:{1}", duration,
   duration.Seconds);
```

Властивості, функції та операції над часом та датою

```
DateTime d1 = DateTime.Parse("12:38:41 2015-12-20");
Console.WriteLine("d1 = {0}", d1);
DateTime d2 = DateTime.Parse("20/12/2015 12:38:41");
Console.WriteLine("d2 = {0}", d2);
Console.WriteLine("d1 == d2? {0}", d1 == d2);
DateTime d3 = DateTime.Today;
Console.WriteLine("d3 = {0}", d3);
Console.WriteLine("d1 == d3? {0}", d1 == d3);

DateTime dTime = DateTime.Now; // DateTime.Today
Console.WriteLine("Now: " + dTime);
Console.WriteLine("Date = " + dTime.ToShortDateString());
Console.WriteLine("Time = " + dTime.ToShortTimeString());
Console.Write("Input a time interval in format [д.]чч:мм[:сс[.дс]] ");
```

```
        TimeSpan interval;
        if (TimeSpan.TryParse(Console.ReadLine(), out interval))
        {
            dTime += interval; //dTime = dTime.AddDays(25);
            Console.WriteLine(@"After interval:
            Date = {0}, Time = {1}
            Day of week={2}, Day of year={3}, Hour={4}",
           dTime.Date, dTime.TimeOfDay, dTime.DayOfWeek, dTime.DayOfYear, dTime.Hour);
        }
```

Делегати

По суті, тип *делегата* .NET — це тип безпечного об’єкту, який "посилається" на метод або список методів, які можуть бути викликані пізніше через цей делегат. На відміну від вказівника на функцію C++, делегати .NET є класами, що мають вбудовану підтримку групового виконання і асинхронного виклику методів.

Приклад. Вивід таблиці значень операції

```
class Program
{
    delegate double ProcessDelegate(double paraml, double param2) ;
    static double Multiply (double paraml, double param2)
    {
        return paraml * param2;
    }
    static double Divide (double paraml, double param2)
```

```
    {
        return paraml / param2;
    }
    static void Main(string[] args)
    {
        ProcessDelegate process;
        Console.WriteLine("Enter 2 numbers separated with a comma:") ;
        string input = Console.ReadLine();
        int commaPos = input.IndexOf(',');
        double paraml = Convert.ToDouble(input.Substring(0, commaPos));
        double param2 = Convert.ToDouble(input.Substring(commaPos + 1,
        input.Length - commaPos - 1)) ;
        Console. WriteLine ("Enter M to multiply or D to divide:") ;
        input = Console.ReadLine();
        if (input == "M")
            process = new ProcessDelegate(Multiply);
        else
            process = Divide;
        Console.WriteLine("Result: {0}", process(paraml, param2));
    }
}
```

Робота з файлами та потоками

Ввід-вивід в програмах на C# виконується із використанням *потоків*. Класи для роботи з файловими потоками зосередженні у просторі імен `System.IO`. Основні класи простору `System.IO` наведені в таблиці 4:

Таблиця 4. Основні класи для обробки потоків

Клас	**Опис**
`BinaryReader`, `BinaryWriter`	Зчитування та запис елементарних типів даних (цілочислові, логічні, рядкові і т.п.) у двійковому вигляді.
`BufferedStream`	Тимчасове сховище для потоку байтів.
`Directory`	Клас, який надає статичні методи для маніпулювання структурою каталогів.
`File`	Клас, який надає статичні методи для маніпулювання набором файлів даної машини.
`FileStream`	Клас, який забезпечує прямий доступ до файлу даних, які обробляються як потік байтів.
`MemoryStream`	Клас, який забезпечує прямий доступ до файлу даних, які зберігаються у пам'яті машини.
`StreamWriter`, `StreamReader`	Класи, які призначені для запису чи зчитування текстових даних із використанням послідовного доступу до вмісту файлів.
`StringWriter`, `StringReader`	Класи, які аналогічні до попередніх, проте працюють із текстовим буфером, а не файлом.

Запис даних у потік

Для запису текстових даних використовуються класи `StreamWriter` та `StringWriter`, основні члени яких (описані у абстрактному класі `TextWriter`) наведені у табл. 5. Об’єкти класу `StreamWriter` створюються за допомогою конструкторів, перший параметр яких є потоком, або повним іменем файлу. По замовчуванню файл перезаписується. Можна використовувати конструктор, другий логічний параметр якого `append` рівний `true` для того, щоб отримати можливість додавати дані у кінець файлу. Можна також вказувати додатковий параметр типу `Encoding`, який вказує на кодову сторінку, яку треба застосовувати при роботі із файлом.

Таблиця 5. Основні члени класу `TextWriter`

Метод	**Опис**
`Close`	Звільнення усіх ресурсів та потоку.
`Flush`	Очищає усі буфери і записує усі буферизовані дані у потік (на носій) без його закриття.
`Write`	Запис даних у потік
`WriteLine`	Запис даних у потік із добавленням символу нового рядка

Приклад. Запис у потік.

```
struct Entry
{
    public string Name;
    public string Job;
    public decimal Salary;
}
static void Main(string[] args)
{
    StreamWriter sw = null;
    try
    {
        Entry[] entryArray =
        {
            new Entry{Name = "Ivanov", Job="Programmer", Salary= 5000},
            new Entry{Name = "Petrenko", Job = "Admin", Salary = 4000},
            new Entry{Name ="Stepanenko",Job = "Manager", Salary=8000},
            new Entry{Name = "Ivanov", Job = "Programmer", Salary=5000}
        };
        string path = "data.txt";
        sw = new StreamWriter(path); //sw=new StreamWriter(path, true);
        //sw.WriteLine("{0,15}{1,15}{2,10}", "Name", "Job", "Salary");
        foreach (var x in entryArray)
            sw.WriteLine("{0},{1},{2}", x.Name, x.Job, x.Salary);
    }
    catch (IOException e)
    {
        Console.WriteLine("An IO exception has been thrown\n{0}", e);
```

```
        }
        finally
        {
            if(sw != null)
              sw.Close();
        }
    }
```

Зчитування даних із потоку

Для зчитування текстових даних використовуються класи `StreamReader` та `StringReader`, основні члени яких наведені у табл. 6.

Таблиця 6. Основні члени класу `StreamReader`

Метод	**Опис**
`EndOfStream`	Властивість, яка повертає `True` лише тоді, коли досягнуто кінця потоку.
`Peek`	Повертає поточний доступний символ, не змінюючи положення зчитувального пристрою.
`Read`	Зчитує наступний символ із потоку
`ReadBlock`	Зчитує вказану максимальну кількість символів із потоку і записує дані у буфер
`ReadLine`	Зчитує та повертає рядок символів із заданого потоку. Повертає `null`, якщо досягнутий кінець потоку.
`ReadToEnd`	Зчитує та повертає усі символи від поточної позиції до кінця потоку.

Приклад. Зчитування даних із файлу

```
string path = "data.txt";
using (StreamReader sr = new StreamReader(path))
{
   string line=string.Format("{0,15}{1,15}{2,10}", "Name","Job","Salary");
    Console.WriteLine(line);
    Console.WriteLine(new string('-', line.Length));
    //string str = "";
    //sw.WriteLine(str.PadLeft(40, '-'));
    while ((line = sr.ReadLine()) != null)
    {
        string[] strArray = line.Split(',');
        Console.WriteLine("{0,-15}{1,-15}{2,-10}", strArray);
    }
}
```

Альтернативою може бути наступний спосіб використання циклу `while` для зчитування:

```
while(!sr.EndOfStream)
{
    string[] strArray = sr.ReadLine().Split(',');
    Console.WriteLine("{0,-15}{1,-15}{2,-10}", strArray);
}
```

Клас System.OI.File

Клас `File` надає засоби для копіювання, переміщення, шифрування, розшифрування та видалення файлів, а також для отримання інформації про файли і зчитування та запису інформації у них. Усі методи цього класу — статичні.

Таблиця 7. Деякі члени класу `File`

Метод	**Опис**
`AppendAllLines`	Відкриває файл, додає у нього усі рядки колекції та закриває файл.
`AppendAllText`	Відкриває файл, додає у нього вказаний рядок та закриває файл.
`Create`	Створює чи перезаписує вказаний файл та повертає об’єкт `FileStream` для взаємодії із цим файлом.
`Exists`	Перевіряє, чи існує файл із заданим іменем.
`GetAttributes`	Повертає атрибути файлу.
`Move`	Переміщає файл. Цей метод не можна використовувати для перезапису існуючого файлу.
`ReadAllBytes`	Відкриває бінарний файл, зчитує його вміст у масив байтів та закриває файл.

`ReadAllText`	Відкриває текстовий файл, зчитує його вміст у рядок та закриває файл
`ReadAllLines`	Відкриває текстовий файл, зчитує його вміст у масив рядків та закриває файл.
`SetAttributes`	Задає атрибути файлу.
`WriteAllLines`	Створює новий файл, записує у нього колекцію рядків та закриває файл.
`WriteAllText`	Створює новий файл, записує у нього рядок та закриває файл.

Приклад. Зміна даних

```
string dataFile = "data.txt";
string tempFile = "_temp.txt";
if (File.Exists(dataFile))
{
    using (StreamReader sr = new StreamReader(dataFile))
    {
        string contents = sr.ReadToEnd();
        contents = contents.Replace("Programmer", "Coder");
        StreamWriter sw = new StreamWriter(tempFile);
        sw.WriteLine(contents);
        sw.Close();
    }
```

```
        File.Delete(dataFile);
        File.Move(tempFile, dataFile);
    }
```

Файли прямого доступу

Для роботи із файлами прямого доступу використовуються об’єкти класу `System.IO.FileStream`.

Приклад.

```
    byte[] buffer = new byte[300];
    char[] charData = new char[300];
    string dataFile = @"..\..\Program.cs";
    try
    {
        FileStream fs = new FileStream(dataFile, FileMode.Open);
        fs.Seek(100, SeekOrigin.Begin);
        fs.Read(buffer, 0, buffer.Length);
        // fs.Write(buffer, 0, buffer.Length);
        fs.Close();
    }
    catch (IOException e)
    {
        Console.WriteLine("An IOException has been thrown\n{0}", e);
        Console.ReadKey();
    }
    Decoder d = Encoding.UTF8.GetDecoder();
    d.GetChars(buffer, 0, buffer.Length, charData, 0);
    Console.WriteLine(charData);
```

```
// Encoder e = Encoding.UTF8.GetEncoder();
// e.GetBytes(charData, 0, charData.Length, buffer, 0, true);
```

Колекції

Колекція — це набір об’єктів. У просторі імен `System.Collections` для роботи з колекціями визначені інтерфейси `IEnumerable`, `ICollection`, `IList`, `IDictionary`.

Інтерфейс `IEnumerable` надає можливість ітерації по елементам колекції. Підтримка цього інтерфейсу передбачає реалізацію методу

```
IEnumerator GetEnumerator()
```

Приклад. Вивід елементів колекції на консоль:

```
static void PrintCollection(IEnumerable collection)
{
    foreach (var item in collection)
    {
        Console.WriteLine(item);
    }
}
```

Інтерфейс ICollection

Інтерфейс `ICollection` є базовим для розробників неузагальнених колекцій. Опис:

```
public interface ICollection: IEnumerable.
```

Методи та властивості, описані в інтерфейсі, наведені у таблиці 8.

Таблиця 8. Методи та властивості інтерфейсу `ICollection`

Метод	**Опис**
`IEnumerator GetEnumerator()`	Повертає ітератор колекції.
`void CopyTo(Array` *`array`*`, int index)`	Копіює вміст колекції у масив *`array`*, починаючи з елемента із вказаним індексом.
Властивості	**Опис**
`int Count`	Кількість елементів колекції
`bool IsSynchronized`	`true`, якщо доступ до колекції є синхронізованим (потокобезпечним).
`object SyncRoot`	Повертає об'єкт, який може використовуватися для синхронізації доступу до колекції (з використанням `lock`)

Інтерфейс IList

`IList` — колекція для списку елементів, до елементів якого можна звертатися за індексом:

```
public interface IList: ICollection, IEnumerable;
```

Нові методи та властивості, визначені у цьому інтерфейсі, наведені у таблиці 9.

Таблиця 9. Методи та властивості інтерфейсу `IList`

Метод	**Опис**
`int Add(object value)`	Додає в колекцію новий елемент та повертає індекс, який вказує позицію, в яку був доданий елемент.
`void Clear()`	Видаляє усі елементи колекції.
`bool Contains(object value)`	Повертає `true`, якщо колекція містить елемент `value`.
`int IndexOf(object value)`	Повертає індекс першого входження елемента або –1, якщо елемент відсутній у колекції
`void Insert(int index, object value)`	Вставляє заданий об’єкт у задану індексом позицію колекції. Елементи, які мали індекси не менші за `index`, зсуваються вперед.
`void Remove(object value)`	Видаляє перше входження вказаного об’єкта.
`void RemoveAt(int index)`	Видаляє із колекції об’єкт, розташований за вказаним індексом.
Властивості	**Опис**
`bool IsFixedSize`	Повертає `true`, якщо колекція має фіксований розмір.
`bool IsReadOnly`	Повертає `true`, якщо колекцію не можна модифікувати.
`int Count`	Кількість елементів колекції.
`object Item[int index]`	Індексатор для доступу до елементів колекції.

Інтерфейс IDictionary

`IDictionary` — колекція пар *"ключ-значення"*, у якій можливий доступ до елементів по *унікальному* значенню ключа:

```
public interface IDictionary: ICollection,  IEnumerable.
```

Методи та властивості, описані в цьому інтерфейсі, наведені в табл. 10.

Таблиця 10. Методи та властивості інтерфейсу `IDictionary`

Метод	**Опис**
`void Add(object key, object value)`	Додає в колекцію нову пару ключ-значення.
`void Clear()`	Спорожню колекцію.
`bool Contains(object key)`	Повертає `true`, якщо колекція містить елемент з ключем `key`.
`void Remove(object key)`	Видаляє елемент колекції за заданим ключем .
Властивості	**Опис**
`bool IsFixedSize`	Повертає `true`, якщо колекція має фіксований розмір.
`bool IsReadOnly`	Повертає `true`, якщо колекцію не можна модифікувати.
`int Count`	Кількість елементів колекції.

`ICollection Keys`	Повертає колекцію ключів.
`ICollection Values`	Повертає колекцію значень.
`object this[object key]`	Індексатор, у якості індексу якого використовується ключ.

Для ітерації по словнику у циклі використовуються змінні типу `DictionaryEntry`.

Класи неузагальнених колекцій

Деякі класи, у яких реалізовано інтерфейси колекцій, наведено у таблиці 11.

Таблиця 11. Деякі класи-колекції із простору `System.Collections`

Метод	**Опис**
`ArrayList`	Динамічний масив.
`Hashtable`	Хеш-таблиця для пар ключ-значення.
`SortedList`	Відсортований список пар ключ-значення.
`Queue`	Черга — список, який функціонує за принципом "перший прийшов — перший оброблений".

Клас ArrayList

У класі `ArrayList` реалізуються інтерфейси `ICollection`, `IList`, `IEnumerable` та `ICloneable`. Для створення об'єктів класу можна використати один із наступних конструкторів:

```
public ArrayList()
public ArrayList(ICollection c)
public ArrayList(int capacity)
```

Деякі допоміжні методи класу `ArrayList` наведені у таблиці 12.

Таблиця 12. Деякі методи класу `ArrayList`

Метод	**Опис**
`AddRange(ICollection c)`	Добавлення елементів колекції *c* у кінець поточної колекції.
`BinarySearch`	Виконує бінарний пошук елемента впорядкованого списку та повертає його індекс.
`GetRange(int index, int count)`	Повертає частину викликаючої колекції.
`InsertRange(int index, ICollection c)`	Добавлення елементів колекції *c* у поточну колекцію, починаючи з елемента із заданим індексом.

`RemoveRange(int index, int count)`	Видаляє частину викликаючої колекції.
`SetRange(int index, ICollection c)`	Заміняє частину колекції, починаючи з елемента, який має заданий індекс, елементами колекції с.
`Sort`	Впорядковує колекцію
`object[] ToArray()`	Повертає масив, який містить копії елементів поточного списку.

Властивість `Capacity` дає змогу дізнаватися та задавати ємність колекції.

Клас Hashtable

Клас Hashtable реалізовує інтерфейс `IDictionary`. Пари «ключ-значення» упорядковані по хеш-коду

```
Hashtable h = new Hashtable();
h.Add("Smith", 2000);
h.Add("Turner", 3000);
//h["Turner"] = h[“Turner”] + 100;        // Помилка!
h["Carter"] = 2500;                       // додасться новий елемент
//h.Add("Smith", 500);                    // Помилка!
h["Smith"] = 5000;
h.Remove("Smith");
foreach (var x in h.Keys)
    Console.WriteLine(x);
foreach (var x in h.Values)
    Console.WriteLine(x);
foreach (DictionaryEntry x in h)
```

```
        Console.WriteLine("Key = {0}, Value = {1}", x.Key, x.Value);
```

Клас SortedList

Клас `SortedList` – колекція пар «ключ-значення», упорядкованих по ключу

```
    SortedList sl = new SortedList();
    sl.Add("First", 1);
    sl.Add("second", 2);
    sl["Third"] = 3;
    sl["Four"] = 4;
    foreach (DictionaryEntry de in sl)
        Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
```

Універсальні колекції

Універсальні класи (узагальнення) в платформі .NET являють собою концепцію параметризованих типів, які дозволяють розробляти класи та методи, специфікація типів для яких визначається лише у момент оголошення методів та створення екземплярів класів клієнтським кодом. Універсальні колекції зосереджені у просторі `System.Collections.Generic`.

Множина. Клас HashSet<T>

Множина — набір елементів, які не можуть повторюватися і для яких не визначений який-небудь порядок. Клас `HashSet<T>` реалізує інтерфейс `ISet<T>`. Об’єкти класу

HashSet<T> використовуються для збереження множин, усі елементи яких мають тип T. Перевага класу HashSet<T> — ефективність операцій пошуку елементів.

Таблиця 13. Деякі методи та властивості інтерфейсу ISet<T>

Метод	**Опис**
bool Add(T value)	Додає елемент в множину.
void Clear()	Спорожню колекцію.
bool Contains(T value)	Повертає true, якщо множина містить заданий елемент
bool Remove(T value)	Видаляє заданий елемент множина.
IntersectWith(IEnumerable<T> other)	Змінює множину, залишаючи тільки ті елементи, які є у колекції other
UnionWith(IEnumerable<T> other)	Змінює множину, об'єднуючи її елементи з елементами other
IsSubsetOf(IEnumerable<T> other)	Повертає true, якщо усі елементи множини містяться у колекції other
IsSupersetOf(IEnumerable<T> other)	Повертає true, якщо множина містить усі елементи колекції other
ExceptWith(IEnumerable<T>	

other)	
SymmetricExceptWith IEnumerable<T> other)	
Overlaps IEnumerable<T> other)	

Універсальний список. Клас List<T>

Універсальний клас `List<T>` з простору `System.Collections.Generic` — строго типізований список об'єктів, доступних по індексу.

```
static bool MyPredicate(string s) { return s.EndsWith("e"); }
static void Main()
{
    string[] strArray = { "one", "two", "three" };
    List<string> myList = new List<string>(strArray);
    myList[0] = "One";
    strArray = new string[2]{ "four", "five"};
    myList.AddRange(strArray);
    myList.InsertRange(3, new string[2] { "new1", "new2" });
    myList.RemoveRange(3, 2);
    PrintCollection(myList);
    strArray = myList.FindAll(MyPredicate).ToArray();
```

```
        PrintCollection(strArray);
        Console.WriteLine(myList.FindLast((x)=>x.StartsWith("t")));
    }
```

Універсальний словник. Клас Dictionary<TKey, TValue>

Універсальний клас `Dictionary<TKey, TValue>` — колекція ключів і значень.

```
 Dictionary<string, string> openWith = new Dictionary<string, string>();
 openWith.Add("bmp", "paint.exe");
 openWith.Add("txt", "notepad.exe");
 openWith["jpg"] = "paint.exe";
 openWith.Add("rtf", "wordpad.exe");
 openWith["rtf"] = "winword.exe";
 Console.WriteLine("Keys:");
 PrintCollection(openWith.Keys);
 Console.WriteLine("Values:");
 PrintCollection(openWith.Values);
 if (!openWith.ContainsKey("doc"))
     openWith["doc"] = "winword.exe";
 Console.WriteLine("Pairs:");
 foreach (var x in openWith)
     Console.WriteLine("{0}|{1}", x.Key, x.Value);
```

Реалізація парадигми об'єктно-орієнтованого програмування у C#

Основою ООП є *об'єктна модель*. Ця модель складається з наступних основних складових:

1) абстракція;
2) інкапсуляція;
3) ієрархія;
4) модульність.

Абстракція виділяє суттєві характеристики об'єктів певного виду (класу), які відрізняють його від усіх інших видів об'єктів.

Для того, щоб абстракція працювала, її реалізація має бути інкапсульована. На практиці це означає, що кожний клас має складатися із двох частин: *інтерфейса* та *реалізації*. При цьому об'єкти розглядаються як "*чорні скриньки*", інтерфейс яких фіксує лише зовнішнє зображення об'єктів, описуючи абстракцію поведінки усіх об'єктів даного класу. Реалізація містить механізми досягнення бажаної поведінки об'єктів.

Інкапсуляція — це процес поділу елементів абстракції, які визначають її структуру та поведінку.

Ієрархія — це ранжування, чи впорядкування абстракцій.

Основним видами ієрархії є *структура класів* (ієрархія наслідування або успадкування) та структура об’єктів (ієрархія "ціле/частина").

Об’єктам притманний стан, поведінка та індивідуальність (властивість об’єкта, яка відрізняє його від усіх інших об’єктів). Структура та поведінка схожих об’єктів визначається у їх спільному класі. Терміни *екземпляр* та *об’єкт* є синонімами.

Клас — це множина об’єктів, які мають спільну структуру, поведінку та семантику.

Формально клас — це тип користувача, який складається з полів, які призначені для збереження даних, та методів та властивостей, які оперують цими даними.

Модифікатори доступу до класів у C#

Доступ вказується за допомогою ідентифікаторів:

1) `internal` — (доступний у межах збірки);

2) `public` — загальнодоступний;

3) `abstract` — абстрактний (створення екземплярів є неможливим);

4) `sealed` — герметичний (не підтримує наслідування).

По замовуванню використовується `internal`.

Ключове слово `private` не можна використовувати у якості модифікатора доступу до класу!

```
internal class MyClass
{
    // Члени класу
}

public abstract class MyClass
{
    // Члени класу
}

public sealed class MyClass // герметизований клас
{
    // Члени класу
}

internal class MyBase { }

public class MyClass : MyBase // помилка доступ у
{
    // Члени класу.
}
```

Приклад. Використання абстрактних класів.

```
abstract class AbstractClass
{
    public int value;
    public AbstractClass() { value = 1; }
    public static int GetString()
    {
        return "Response from abstract class";
    }
}
class DerivedClass : AbstractClass
{
    public AbstractClass GetSelf() { return this; }
}
static void Main()
{
// AbstractClass x = new AbstractClass();
// не можна створювати екземпляр абстрактного класу
  Console.WriteLine(AbstractClass.GetString());
  DerivedClass a = new DerivedClass();
  AbstractClass x = a.GetSelf();
  Console.WriteLine(x.value);
}
```

Статичні класи

Статичні класи мають дві особливості:

1) не можна створювати об’єкти статичного класу;

2) статичний клас може містити лише статичні члени.

Прикладом статичних класів є клас `Console`.

```
static class MyClass
{
    static int counter;
    static MyClass() // статичний конструктор (не може мати параметри)
    { counter = 0; }
    static public void print()
    {
        Console.WriteLine(++counter);
    }
}
class Program
{
    static void Main(string[] args)
    {
        MyClass.print();
        MyClass.print();
    }
}
```

Методи класу System.Object

Клас `Object` є кореневим класом ієрархії .NET Framework. Його загальнодоступні методи доступні в усіх інших класах. Основні із них наведені у табл. 13.

Таблиця 13. Деякі методи класу `Object`

Метод	**Опис**
`Object()`	Конструктор. Автоматично викликається об'єктами похідних типів
`Finalize()`	Деструктор для об'єктів типу `Object`
`static bool ReferenceEquals(object, object)`	Перевірка того, що обидва параметри посилаються на той самий об'єкт
`virtual bool Equals(object)`	Перевірка еквівалентності
`static bool Equals(object, object)`	Статична версія попереднього методу
`Object MemberwiseClone()`	Копіює об'єкт шляхом створення нового об'єкту і копіювання його членів (без створення нових екземплярів цих членів)
`virtual string ToString()`	Повертає рядок, який відповідає екземпляру об'єкта
`System.Type GetType()`	Повертає тип об'єкта у вигляді об'єкта типу `System.Type`
`virtual int GetHashCode()`	Використовується у якості хеш-функції об'єктів

Приклад. Вивід інформації про тип.

```
class MyClass
{
    public int x = 1;
    private double y = 2;
    public void Incx() {}
}

class Program
```

```
{
    static void PrintMembers(Type obj)
    {
        foreach (var a in obj.GetMembers())
            Console.WriteLine("{0}", a);
    }
    static void Main(string[] args)
    {
        MyClass a = new MyClass();
        Console.WriteLine("String representation: {0}\nNamespace: {1}",
            a, a.GetType().Namespace);
        PrintMembers(a.GetType());
        Type t = typeof(MyClass);
        PrintMembers(t);
    }
}
```

Специфіка Equals для типів значень та типів посилань

```
class MyClass {public int x;}
struct MyStruct {public int x;}
MyClass a = new MyClass(), b = new MyClass();
a.x = b.x = 10;
MyStruct c, d;
c.x = d.x = 10;
Console.WriteLine("Is a equals to b? {0}\nIs c equals to d? {1}",
     a.Equals(b),c.Equals(d));
b = a; b.x = 20;
d = c; d.x = 20;
```

```
Console.WriteLine("Is a equals to b? {0}\nIs c equals to d? {1}",
     a.Equals(b), c.Equals(d));
```

Перевизначення Equals

```
class MyClass
{
    public double x;
    public override bool Equals(object obj)
    {
        if (obj.GetType().IsPrimitive)
            return x == Convert.ToDouble(obj);
        else
            if (obj.GetType() == typeof(MyClass))
                return x == ((MyClass)obj).x;
            else
                return false;
    }
    public MyClass Copy()
    {
        return (MyClass)MemberwiseClone();
    }
}

class Program
{
    static void Main(string[] args)
    {
        MyClass a = new MyClass();
```

```
        a.x = 5;
        int b = 3;
        double c = 5;
        MyClass d = a.Copy();
        Console.WriteLine("a.Equals(b)? {0}\n a.Equals(c)? {1}\n a==d?
        {2}\n a.Equals(d)? {3}", a.Equals(b), a.Equals(c), a == d, a.Equals(d));
    }
}
```

Конструктори і деструктори

Конструктори по замовчуванню

Якщо у класі не визначено конструктор, то автоматично створюється конструктор за замовчуванням, який не містить параметрів. Якщо у класі визначено принаймні один конструктор, то конструктор за замовчуванням не створюються.

```
class MyClass
{
    int x;
    public void print() {Console.WriteLine(x);}
}
class MyClass2
{
    int x;
    public MyClass2(int y) {x = y;}
}
        MyClass a = new MyClass();
```

```
            a.print();
            MyClass2 b = new MyClass2(); // помилка
```

Приватні конструктори

```
class MyClass
{
    private MyClass(int x)
    {
        this.x = x;
    }
    public int x;

    public static MyClass GetInstance(int x)
    {
        return new MyClass(x);
    }
}
```

Порядок виклику конструкторів

Перед викликом констуктора для об'єкта класу завжди викликається **конструктор по замовчуванню базового класу!**

```
    class BaseClass
    {
        public int x;
        public BaseClass(int y) { x = 1; }
```

```
}
class DerivedClass : BaseClass
{
    public DerivedClass(int y) { x = y;}
}
DerivedClass a = new DerivedClass(5); // помилка
```

Явна вказівка конструктора базового класу

```
class BaseClass
{
    public int x;
    public BaseClass(int y) { x = y; }
    //по замовчуванню компілятор додає до коду ініціалізатор base()
}
class DerivedClass : BaseClass
{
    public DerivedClass(int y) : base(y){}
    public DerivedClass(int y, int z) : this(y + z) { }
}
```

Порядок дій при ініціалізації

Порядок дій при ініціалізації у C#:

1. Ініціалізація статичних членів класу.
2. Ініціалізація нестатичних членів (явна ініціалізація та значення по замовчуванню).
3. Ініціалізація базового класу (рекурсія) з викликом вказаних конструкторів.
4. Виклик конструктора поточного класу.

Порядок дій при ініціалізації у Java:

1. Ініціалізація статичних членів класу (від кореневого до поточного).
2. Ініціалізація нестатичних членів та виклик конструктора.

Ініціалізація нестатичних членів та виклик конструктора:

1. Ініціалізація нестатичних членів та виклик конструктора базового класу (рекурсія).
2. Ініціалізація нестатичних членів поточного класу.
3. Виклик конструктора поточного класу.

```
class Class1{
    static int x = 0;
    public Class1(string s) {
        Console.WriteLine("Class1 constructor" + x++ + " call in " + s);
    }
}
class Base{
    int b = Base.f(3);
    public Base()
    {Console.WriteLine("Base constructor");}
    public static int f(int x) {
        Console.WriteLine("f call in Base");
        return x + 1;
    }
    static Class1 a = new Class1("Base a");
}
class Derived : Base{
    static Class1 b = new Class1("Derived (b)");
    Class1 bb = new Class1("Derived (bb)");
    public Derived()
    {Console.WriteLine("Derived constructor");}
}
class MyClass : Derived{
    public MyClass() { Console.WriteLine("MyClass constructor"); }
    Class1 cc = new Class1("MyClass (cc)");
    static Class1 c = new Class1("MyClass (c)");
    static void Main(string[] args)
    {
        Console.WriteLine("Main call");
```

```
            MyClass p = new MyClass();
        }
    }
```

Вивід C#:

```
Class1 constructor0 call in MyClass (c)
Main call
Class1 constructor1 call in MyClass (cc)
Class1 constructor2 call in Derived (b)
Class1 constructor3 call in Derived (bb)
Class1 constructor4 call in Base a
f call in Base
Base constructor
Derived constructor
MyClass constructor
```

Вивід Java:

```
Class1 constructor0 call in Base a
Class1 constructor1 call in Derived (b)
Class1 constructor2 call in MyClass (c)
Main call
f call in Base
Base constructor
Class1 constructor3 call in Derived (bb)
Derived constructor
Class1 constructor4 call in MyClass (cc)
MyClass constructor
```

Деякі обмеження структур

Структури *не можуть* мати нащадків (не можуть бути базовими класам). Також *не можна* використовувати *конструктори без параметрів* для структур.

Деструктори

```
class BaseClass
{
    public int x;
    ~BaseClass()
    { Console.WriteLine("destructor of BaseClass is called"); }
}
class DerivedClass : BaseClass
{
    ~DerivedClass()
    { Console.WriteLine("destructor of DerivedClass is called"); }
}
class Program
{
    static void Main(string[] args)
    {
        DerivedClass a = new DerivedClass();
        Thread.Sleep(2000);
    }
}
```

Поля та методи класу

Доступ для полів

У якості модифікаторів використовуються ключові слова public, private, internal, protected. Останні два слова можна комбінувати (тоді доступ до поля мають похідні типи із інших збірок). Поля та методи можуть описуватися з використанням ключового слова static, яке на те, що вони є статичними членами, спільними для усіх екземплярів класу. При описі полів можна також використовувати ключове слово readonly, яке вказує на те, що даному полю можна присвоювати значення лише у конструкторі або шляхом операції початкового присвоєння.

```
class MyClass
{
     public readonly int Mylnt = 17 ;
     static double a = 10;
     const byte b = 1;
}
```

Також можна описувати поля-константи. Для цього використовується ключове слова const. Поля-константи є статичними по означенню, тому для них не використовується ключове слово static.

Модифікатори для методів

Доступними є модифікатори: `static`, `abstract` (неявний віртуальний), `virtual`, `override`, `extern`

```
abstract class Shape
{
    abstract public double CalcArea();
}
class Rectangle : Shape
{
    double width, height;
    public Rectangle(double w, double h) {width = w; height = h;}
    public override double CalcArea()  { return width * height; }
}
class Program
{
    static void Main(string[] args)
    {
        Rectangle r = new Rectangle(20, 15);
        Console.WriteLine("Area = {0}", r.CalcArea());
    }
}
```

```
public class MyBaseClass
{
   public virtual void DoSomething ()
   {
```

```
    // Базова реалізация.
    }
}
public class MyDerivedClass : MyBaseClass
{
    public override void DoSomething ()
    // public override sealed void DoSomething () – в метод не можна
    // вносити зміни в похідних класах
    {
    // Реализація в класі-нащадку
    }
}
```

Ключове слово this

Ключове слово `this` — посилання на поточний екземпляр.

```
public void doSomething ()
{
 MyTargetClass myObj = new MyTargetClass ();
 myObj.DoSomethingWith(this);
}
```

Особливості наслідування

Для явної вказівки того, що потрібно *приховати* метод базового класу, перед його іменем потрібно вказати ключове слово `new`.

```
class MyBaseClass
{
```

```
        public virtual void f1() { Console.WriteLine("Base"); }
        public virtual void f2() { Console.WriteLine("Base"); }
    }
    class MyDerivedClass : MyBaseClass
    {
        public override void f1() { Console.WriteLine("Derived"); }
        new public void f2() { Console.WriteLine("Derived"); }
    }
    static void Main(string[] args)
    {
        MyDerivedClass myDerObj = new MyDerivedClass();
        MyBaseClass myBaseObj = myDerObj;
        myBaseObj.f1();
        myBaseObj.f2();
        myDerObj.f1();
        myDerObj.f2();
    }
```

Приховувати можна не тільки методи, але і поля.

Для виклику перевизначених та прихованих методів базового класу перед назвою методу використовується ключове слово `base`.

```
class MyDerivedClass : MyBaseClass
{
    new public void f2()
    {
            base.f2();
            Console.WriteLine("Derived");
```

```
    }
}
```

Властивості

Властивості є різновидом членів класу, які, як правило, поєднують у собі поле з методами доступу до нього. При описі властивостей можна використовувати ключові слова `public`, `protected`, `virtual`, `override`, `abstract`. Властивості складаються із імені та аксесорів `get` та `set`. Синтаксис:

```
Тип Ім'я
{
    get{
    // код аксессора для зчитування поля
    }
    set{
    // код аксессора для запису в поле
    }
}
```

Аксесор `set` приймає неявний параметр `value`, який містить значення, яке присвоюється полю.

Приклад. Реалізація класів геометричної фігури, прямокутника та квадарата.

```
abstract class Shape
{
    public readonly string Description;
    abstract protected double CalcArea();
    public virtual double Area { get { return CalcArea(); } }
    public Shape(string s) { Description = s; }
}

class Rectangle : Shape
{
    private double height;
    public double Height
    {
        get { return height; }
        set
        {
            if (value >= 0)
            height = value;
        else
            throw (new ArgumentOutOfRangeException("Invalid Height"));
        }
    }
    public double Width // автоматична властивість (обов’язковими є обидва блоки get та set)
    {
        get;
        set;
    }
    public Rectangle(double h, double w) :
         base("a parallelogram having four right angles")
```

```
    {
        Height = h;
        Width = w;
    }
    public Rectangle() : this(0, 0){}
    protected override double CalcArea()  { return Width * Height; }
}

class Square : Rectangle
{
    public Square(int height) : base(height, height)
    {
        Description = "a rectangle having four equal sides";
    }
    public override double Area
    {
        get { return base.Area; } // перевизначення властивості
    }
}
```

Вкладені класи

Допустимим є визначення класів усередині інших класів.

```
public class MyClass
{
   public class myNestedClass
   {
      public int nestedClassField;
   }
```

```
}
MyClass.myNestedClass myObj = new MyClass.myNestedClass();
```

Інтерфейси. Реалізація інтерфейсів

Інтерфейс — набір загальнодоступних методів і властивостей, які об’єднуються разом для інкапсуляції конкретної функціональності. Після визначення інтерфейсу його можна реалізувати у класі.

При роботі із інтерфейсами потрібно дотримуватися наступних обмежень:

1) при описі інтерфейсів модифікатори `abstarct, sealed` *недопустимі*;
2) для членів інтерфейсів *не можна використовувати модифікатори доступу* (`public, private, protected, internal`).
3) члени інтерфейсу *не можуть містити тіло коду*.
4) у інтерфейсах *не можна визначати поля* та *не можна визначати типи*.
5) *не можна використовувати* ключові слова `static, virtual, abstract` або `sealed`.

Інтерфейс може наслідувати методи та властивості інших інтерфейсів.

Приклад.

```
interface IMyInterface1
{
    void DoSomething();
```

```
}

interface IMyInterface2
{
    double MyDouble { get; }
   // Синтаксис схожий на синтаксис автоматичних
   // властивостей, але присутність обидвох блоків get, set необов'язкова
}

interface IMyDerivedInterface : IMyInterface1, IMyInterface2
{
    new void DoSomething(); // приховування методу допустиме
}
```

Реалізація інтерфейсів у класах

Члени інтерфейсів можна реалізовувати за допомогою ключових слів `virtual`, `abstract`. *Не можна* використовувати ключові слова `static` та `const`.

```
    interface MyInterface
    {
        string Name { get; set; }
        void Show();
        void ShowRep();
        string TransformName();
    }
    class MyBaseClass
    {
        string name;
```

```
    int length;
    public string Name //члени інтерфейсів можуть бути реалізовані у базових класах
    {
        get { return name; }
        set { name = value.Substring(0, length); }
    }
    public MyBaseClass(int length) { this.length = length;}
    public virtual void Show() { Console.WriteLine(name); }
}
class MyDerivedClass : MyBaseClass, MyInterface
{
int reps;
public MyDerivedClass(int length, int reps) :
          base(length) {this.reps = reps;}
public override void Show() //неявна реалізація інтерфейсу
{
    Console.WriteLine("\"{0}\"", Name);
}
void IMyInterface.ShowRep() //явна реалізація (не можна вказувати private, public)
{
    for(int i = 0; i < reps; i++)
        base.Show();
}
public string TransformName()
{
    char[] charArray = name.ToCharArray();
    Array.Reverse(charArray);
    return new string(charArray);
```

```
    }
}
class MyChildClass : MyDerivedClass   //дочірній клас також підтримимує
                                      // інтерфейс, реалізований у базовому
{
    public MyChildClass() : base(5, 2) { }
    public override sealed void Show()
    {
        Console.WriteLine(Name.ToUpper());
    }
    new public string TransformName()
    {
        string s = "";
        for (int i = 0; i < name.Length; i++)
            s += name[i] + " ";
        return s;
    }
}
static void Main(string[] args)
{
    MyChildClass x = new MyChildClass();
    x.Name = "abcde";
    x.Show();    // коректний виклик неявно реалізованого методу
    x.ShowRep(); // помилка; не можна викликати явно реалізований метод
    MyInterface y = x;
    y.ShowRep();
    Console.WriteLine(y.TransformName());// викликається MyDerivedClass.Transform()!
    Console.WriteLine(x.TransformName());
```

```
        y = new MyDerivedClass(4, 3);
        y.Name = "xyztuw";
        y.Show();
    }
```

Перевизначення операцій

Для пере визначення операцій використовується ключове слово `operator`, яке визначає *статичний* операторний метод, що містить код, який має виконуватися при виклику оператора.

Загальна форма визначення унарної операції:

```
public static тип_результату operator op(тип_параметра операнд)
{
    // код
}
```

Загальна форма визначення бінарної операції:

```
public static тип_результату operator op(тип1 операнд1, тип2 операнд2)
{
    // код
}
```

Тип операнда унарної операції має співпадати із типом класу, в якому він визначається. Для бінарних операцій хоча би один із операндів має бути того самого типу, що і клас.

Приклад. Визначення операцій для двовимірного вектора.

```
struct Point
{
    public int x, y;
}

struct Vector
{
    public double x, y;

    public Vector(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
    public override string ToString()
    {
        return string.Format("({0},{1})", x, y);
    }
    public static Vector operator +(Vector a, Vector b)
    {
        return new Vector(a.x + b.x, a.y + b.y);
    }
    public static double operator *(Vector a, Vector b)
    {
```

```
        return a.x * b.x + a.y * b.y;
    }
    public static bool operator ==(Vector a, Vector b)
    {
        return a.x == b.x && a.y == b.y;
    }
    public static bool operator !=(Vector a, Vector b)
    {
        return !(a == b);
    }
    // Треба перевизначати обидві операції
    // Перевизначення операцій перетворення типів
    public static implicit operator Vector(Point p)
    {
        return new Vector(p.x, p.y);
    }
    public static explicit operator Point(Vector v)
    {
        Point p;
        checked // явна перевірка переповнення
        {
            p.x = (int)v.x;
            p.y = (int)v.y;
        }
        return p;
    }
}

class Program
```

```
{
    public static void Main()
    {
        Vector x = new Vector(2, 3);
        Vector y = new Vector(5, 1);
        Console.WriteLine(y += x);
        Point p;
        p.x = -4; p.y = 5;
        Vector v = p;
        Console.WriteLine(v + x);
        v.y++; // v.y += 1e15;
        p = (Point)v;
        Console.WriteLine("p == v? {0}", p == v);
    }
}
```

Інтерфейс IEquatable

Параметризований інтерфейс `IEquatable<T>` забезпечує можливість порівняння об'єктів однакового типу. Для реалізації інтерфейсу потрібно перевизначити метод `bool Equals(T other)`.

Приклад. Порівняння точок

```
struct Point: IEquatable<Point>
{
    public int x, y;
    public bool Equals(Point other)
    {
```

```
        return x == other.x && y == other.y;
    }
}
```

Інтерфейс IDisposable (метод Dispose)

Підтримка інтерфейсу `IDisposable` передбачає реалізацію методу `void Dispose()`. Об'єкти, які підтримують цей інтерфейс, часто використовуються у конструкції наступного вигляду:

```
using (<class name> <variable name> = new <class name>())
{
}
```

Глибоке копіювання. Інтерфейс ICloneable

"Глибоке" копіювання виконується шляхом підтримки інтерфейсу `ICloneable` шляхом реалізації методу `Clone()`, який має повертати копію об'єкта.

Приклад. Демонстрація відмінностей між "глибоким" та "поверхневим" копіюванням.

```
class MyClass : ICloneable
{
    public int a;
    public int[] b = new int[1];
    public MyClass(int v) { a = b[0] = v; }
    public MyClass GetCopy() {return (MyClass)MemberwiseClone();}
    public object Clone()
    {
```

```
            MyClass x = new MyClass(0);
            x.a = a;
            x.b = (int[])b.Clone();
            return x;
        }
    }
    static void Main()
    {
        MyClass x = new MyClass(5);
        MyClass y = x.GetCopy();
        MyClass z = (MyClass)x.Clone();
        x.b[0] = x.a = 1;
        Console.WriteLine("x.a = {0}, x.b = {1}", x.a, x.b[0]);
        Console.WriteLine("y.a = {0}, y.b = {1}", y.a, y.b[0]);
        Console.WriteLine("z.a = {0}, z.b = {1}", z.a, z.b[0]);
    }
```

Інтерфейси IComparable та IComparer

Інтерфейси `IComparable` та `IComparer` дають змогу порівнювати об’єкти в .NET Framework.

Інтерфейс `IComparable` реалізується у класі, об’єкт якого порівнюється, шляхом визначення методу `CompareTo`, єдиним параметром якого є об’єкт, з яким порівнюється поточний об’єкт.

Інтерфейс `IComparable` реалізується у окремому класі, у якому визначається метод `CompareTo`, параметрами якого є два об'єкти, які порівнюються.

Обидва інтерфейси мають параметризовані версії `IComparable<T>` та `IComparer<T>`.

Приклад. Визначення методів для порівняння працівників за їх прізвищем чи зарплатою та впорядкування масивів, які містять об'єкти з інформацією про працівників.

```
struct Entry: IComparable<Entry>
{
    public string Name;
    public string Job;
    public decimal Salary;

    public override string ToString()
    {
        return string.Format("[Name: {0}; Job: {1}; Salary: {2}]",
            Name, Job, Salary);
    }

    public int CompareTo(Entry other)
    {
        return StringComparer.Ordinal.Compare(Name, other.Name);
    }
}

class MyComparer: IComparer<Entry>
{
```

```
    public int Compare(Entry x, Entry y)
    {
        return (int)(x.Salary - y.Salary);
    }
}

class Program
{
    static void Main()
    {
        var entries = new Entry[]
        {
            new Entry() {Name = "Lewis", Job = "Manager", Salary = 5000},
            new Entry() {Name = "Black", Job = "Seller", Salary = 3000},
            new Entry() {Name = "Adams", Job = "CEO", Salary = 10000}
        };
        Console.WriteLine("Sorted array");
        Console.WriteLine("By name:");
        Array.Sort(entries);
        Console.WriteLine(string.Join("; ", entries.AsEnumerable()));
        Console.WriteLine("By salary");
        Array.Sort(entries, new MyComparer());
        Console.WriteLine(string.Join("; ", entries.AsEnumerable()));
    }
}
```

Приклад. Написати клас для реалізації звичайних дробів.

Ітератори

Ітератор — це блок коду, який надає всі значення, які потрібно використовувати у блоці `foreach`, по черзі. Цей код може повертати значення типу `IEnumerator` або `IEnumerable`.

Для ітерації по класу використовується метод `GetEnumerator()`, який повертає значення типу `IEnumerator`.

Для ітерації по члену класу, наприклад методу, використовується `IEnumerable`. Значення, які повертаються для циклу `foreach`, в блоці ітератора вибираються з використанням ключового слова `yield`.

Синтаксис: `yield return значення`.

Приклад.

```
static IEnumerable GetList(int n)
{
    for (int i = 1; i <= n; i++)
        yield return "string " + i;
}

static void Main()
{
    foreach (var item in GetList(5))
```

```
    {
        Console.WriteLine(item);
    }
}
```

Приклад ітератора для класу

```
class Primes : IEnumerable
{
    int min, max;
    public Primes() : this(2, 100) { }
    public Primes(int minValue, int maxValue)
  {
     min = minValue; max = maxValue;
  }
    public IEnumerator GetEnumerator()
    {
        if (min < 3)
            yield return 2;
        for (int i = 3; i <= max; i += 2)
        {
            int maxBound = (int)Math.Sqrt(i);
            bool isPrime = true;
            for(int j = 3; j <= maxBound; j += 2)
                if (i % j == 0)
                {
                    isPrime = false;
                    break;
                }
            if (isPrime)
```

```
                    yield return i;
                }
            }
        }
        static void Main()
        {
            Primes p = new Primes(2, 200);
            Console.WriteLine("Prime numbers from 20 till 200: ");
            foreach (int x in p)
                Console.Write("{0} ", x);
        }
```

Часткові визначення класів

Часткове визначення дає змогу розбивати визначення класів на кілька етапів.

```
public partial class MyClass : IMylntefacel
{
    // …
}
public partial class MyClass : IMyInteface2
{
    // …
}
```

еквівалентне визначення:

```
public class MyClass : IMylntefacel, IMyInteface2
{
}
```

частково визначені методи (можуть бути статичними, але **завжди є приватними процедурами** (`void`) **і не містять параметрів** `out`). З ними **не можна використовувати** модифікатори `virtual, abstract, override, new, sealed` **і** `extern.`

```
public partial class MyClass
```

```
{
    partial void DoSomethingElse() ;
    public void DoSomething ()
    {
        Console.WriteLine("DoSomething() execution started.");
        DoSomethingElse();
        Console.WriteLine("DoSomething() execution finished.");
    }
}
public partial class MyClass
{
    partial void DoSomethingElse ()
    {
        Console.WriteLine("DoSomethingElse() called.");
    }
}
```

У випадку відсутності другого визначення часткового класу компілятор видаляє виклик `DoSomethingElse()` з `DoSomething ()`.

Операції is та as

Операція `is` перевіряє, чи є її операнд об’єктом заданого типу або чи може він бути перетвореним до певного типу.

Синтаксис: `<операнд> is <тип>`.

Операція `as` дозволяє перетворити тип до певного типу-посилання.

Синтаксис: `<операнд> as <тип>`. Якщо операнд не може бути перетворений, результатом буде `null`.

```
interface ImyInterface {void Do();}
class MyBaseClass {public void Do() { }}
```

```
class MyDerivedClass : MyBaseClass, ImyInterface {}
static void Check(object param)
{
    if (param is ImyInterface)
        Console.WriteLine("Variable {0} can be converted to ImyInterface", param);
    else
      Console.WriteLine("Variable {0} can't be converted to ImyInterface", param);
    if (param is MyBaseClass)
      Console.WriteLine("Variable {0} can be converted to MyBaseClass", param);
    else
      Console.WriteLine("Variable {0} can't be converted to MyBaseClass", param);
}

static void Main()
{
    MyDerivedClass y = new MyDerivedClass();
    MyBaseClass x = y as MyBaseClass; // MyBaseClass x = y;
    Check(x);
    x.Do();
    y = (MyDerivedClass)x;
    x = new MyBaseClass();
    Check(x);
    //y = (MyDerivedClass)x; // exception
    y = x as MyDerivedClass; // y = null
}
```

Приклад. Клас RealStack

```
class RealStack : ICloneable, IEnumerable
{
    double[] data;
    static int defaultCapacity = 100;
    int topIndex = -1;
    int capacity;
    public RealStack(int n)
    {
        capacity = n;
        data = new double[capacity];
```

```
}
public RealStack() : this(RealStack.defaultCapacity){}
public int Count { get { return topIndex + 1; } }
public int Capacity
{
    get { return capacity; }
    set { Array.Resize(ref data, capacity = value);}
}
public double Top { get { return data[topIndex + 1]; } }
public static bool operator ==(RealStack a, RealStack b)
{
    if (a.Count != b.Count || a.Capacity != b.Capacity)
        return false;
    else
    {
        for (int i = 0; i < a.Count; i++)
            if (a.data[i] != b.data[i])
                return false;
    }
    return true;
}
public static bool operator !=(RealStack a, RealStack b) { return !(a == b); }
public override bool Equals(object obj)
{
    if (obj is RealStack)
        return this == (RealStack)obj;
    else
        return false;
}
public override int GetHashCode() {return topIndex;}
public void Clear() { topIndex = -1; }
public bool Contains(double value) { IList x = data; return x.Contains(value); }
public void CopyTo(double[] array, int index) { data.CopyTo(array, index); }
public static explicit operator double[](RealStack a)
{
    double[] d = new double[a.topIndex + 1];
    Array.Copy(a.data, 0, d, 0, a.topIndex + 1);
    return d;
```

```
        }
        public object Clone()
        {
            RealStack a = new RealStack(capacity);
            Array.Copy(data, 0, a.data, 0, capacity);
            a.topIndex = topIndex;
            return a;
        }
        public IEnumerator GetEnumerator()
        {
            double[] d = (double[])this;
            return d.GetEnumerator();
        }
        public void Push(double value)
        {
            if (topIndex < capacity - 1)
                data[++topIndex] = value;
            else
                throw new InvalidOperationException("Stack overflow exception");
        }
        public double Pop()
        {
            if (topIndex >= 0)
                return data[topIndex--];
            else
                throw new InvalidOperationException("Stack is empty");
        }
    }
    static void Main()
    {
        RealStack a = new RealStack(10);
        a.Push(2);
        a.Push(10);
        a.Pop();
        a.Push(3);
        RealStack b = (RealStack)a.Clone();
        Console.WriteLine("{0}; {1}", b.Contains(3), a == b);
        b.Capacity = 4;
```

```
    PrintCollection(b);
}
```

Приклад. Клас MyVector

```
class MyVector : ICloneable, IComparable, IEnumerable
{
    double[] data;
    public MyVector(int n) : this(n, 0) { }
    public MyVector(int n, double value)
    {
        data = new double[n];
        for (int i = 0; i < n; i++)
            data[i] = value;
    }
    public MyVector(double[] dArray) { data = (double[])dArray.Clone(); }
    public object Clone() { return new MyVector(data); }
    public int Length { get { return data.Length; } }
    public static MyVector operator -(MyVector a)
    {
        MyVector b = new MyVector(a.Length);
        for (int i = 0; i < a.Length; i++)
            b.data[i] = -a.data[i];
        return b;
    }
    public static MyVector operator +(MyVector a, MyVector b)
    {
        if (a.Length != b.Length)
            throw new InvalidOperationException();
        MyVector c = new MyVector(a.Length);
        for (int i = 0; i < a.Length; i++)
            c.data[i] = a.data[i] + b.data[i];
        return c;
    }
    public static MyVector operator *(double x, MyVector a)
    {
        MyVector b = new MyVector(a.data);
        for (int i = 0; i < b.Length; i++)
            b[i] *= x;
```

```
    return b;
}
public static double operator *(MyVector a, MyVector b)
{
    if (a.Length != b.Length)
        throw new InvalidOperationException();
    double s = 0;
    for (int i = 0; i < a.Length; i++)
        s += a.data[i] * b.data[i];
    return s;
}
public static bool operator !=(MyVector a, MyVector b) { return !(a == b); }
public static bool operator ==(MyVector a, MyVector b)
{
    if (a.Length != b.Length)
        return false;
    for (int i = 0; i < a.Length; i++)
        if (a.data[i] != b.data[i])
            return false;
    return true;
}
public static MyVector operator -(MyVector a, MyVector b) { return a + (-b); }
public double this[int index]
{
    get {return data[index];}
    set {data[index]=value;}
}
public static implicit operator MyVector(double[] d) { return new MyVector(d); }
// public static implicit operator Vector(Vector a) {return new Vector(a.data);}
public static explicit operator double[](MyVector a)
{
    double[] dArray = new double[a.Length];
    Array.Copy(a.data, dArray, a.Length);
    return dArray;
}
public int CompareTo(object b)
{
    if (!(b is MyVector))
```

```
            throw new ArgumentException();
        MyVector a = b as MyVector;
        int m = (Length < a.Length) ? Length : a.Length;
        for (int i = 0; i < m; i++)
            if (data[i] != a[i])
                return (int)(data[i] - a[i]);
        if (Length > m)
            return 1;
        else if (Length < m)
            return -1;
        else
            return 0;
    }
    public override bool Equals(object obj)
    {
        if (!(obj is MyVector))
            return false;
        return this == (obj as MyVector);
    }
    //public IEnumerator GetEnumerator() { return data.GetEnumerator(); }
    //public IEnumerator GetEnumerator()
    //{
    //    for(int i = 0; i < Length; i++)
    //        yield return data[i];
    //}
    public IEnumerator GetEnumerator() { return new MyIterator(data); }
}

class MyIterator: IEnumerator
{
    double[] data;
    int position = -1;
    public MyIterator(double[] d) { data = d; }
    public bool MoveNext() { position++; return position < data.Length; }
    public void Reset() { position = -1; }
    public object Current
    {
        get
```

```
            {
                try {return data[position];}
                catch(IndexOutOfRangeException)
                {throw new InvalidOperationException();}
            }
        }
    }
    static void Main()
    {
        double[] x = { 2, 5, 7, 1 };
        MyVector a = new MyVector(x);
        MyVector b = new MyVector(4, 2);
        Console.WriteLine("a*b={0}\na:", a*b);
        PrintCollection(a);
        x = (double[])b;
        Console.WriteLine("b + x:");
        foreach (double y in b + x)
            Console.Write("{0} ", y);
        a -= 3 * b;
        Console.WriteLine("\na:");
        PrintCollection(a);
        b = (MyVector)a.Clone(); b[0] = 10;
        MyVector[] v = new MyVector[3];
        v[0] = b; v[1] = a;
        x[0] = 0; v[2] = x;
        Array.Sort(v);
        Console.WriteLine("v[2]:");
        PrintCollection(v[2]);
    }
```

Універсальні класи (generics)

Універсальні класи (узагальнення) в .NET являють собою концепцію параметричних типів, які дозволяють розробляти класи та методи, специфікація типів для яких визначається лише у момент оголошення методів та створення екземплярів класів клієнтським кодом.

Визначення узагальнених класів

```
class MyGenericClass<T1, T2, T3>
{
}
```

Особливості опису узагальнених класів

```
class MyGenericClass<T>
{
    T value;
    public MyGenericClass() { value = default(T); }
    public static bool Compare(T a, T b) {return a == b;} // помилка
}
static void Main()
{
    bool a = MyGenericClass<int>.Compare(3, 5);
}
```

Обмеження типів для узагальнених класів

```
class MyGenericClass<T1, T2> where T1 : обмеження1 where T2 : обмеження2
{
```

```
}
  class MyClass
  {
      public int value = 1;
      public static bool operator ==(MyClass a, MyClass b) { return a.value == b.value; }
      public static bool operator !=(MyClass a, MyClass b)
      { return !(a.value==b.value); }
  }

  class MyGenericClass<T> where T : MyClass
  {
       public static bool Compare(T a, T b) {return a == b;}
  }

  static void Main()
  {
      bool a = MyGenericClass<MyClass>.Compare(new MyClass(), new MyClass());
  }
```

Типи обмежень:

1) `struct` – тип має бути типом-значенням;

2) `class` – тип-посилання;

3) `base-class` – тип має бути типом base-class або його нащадком;

4) `interface` – тип має являти собою або має реалізовувати заданий інтерфейс;

5) `new()` – тип має мати загальнодоступний конструктор без параметрів.

```
class MyGenericClass<T1, T2> where T2 : T1  // допустимо
```

```
class ShapeList<T> : IEnumerable<T> where T : Shape
{
    private List<T> innerList = new List<T>();
    public List<T> Shapes {get {return innerList;}}
    public IEnumerator<T> GetEnumerator()
    {
     return innerList.GetEnumerator();
  }
   IEnumerator IEnumerable.GetEnumerator() { return innerList.GetEnumerator(); }
}
static void Main()
{
    ShapeList<Rectangle> list = new ShapeList<Rectangle>();
    list.Shapes.Add(new Rectangle(100, 200));
}
```

Наслідування від узагальнених типів

```
class RectList<T> : ShapeList<T> where T : Rectangle { }  // коректно
class RectList<T> : ShapeList<T> where T : class { }      // помилка
class RectList : ShapeList<Rectangle>, ICloneable { }     // коректно
class RectList : ShapeList<T>, ICloneable { }             // помилка
```

Перевизначення операцій

```
    public static ShapeList<T> operator +(ShapeList<T> a, ShapeList<T> b)
    {
        ShapeList<T> result = new ShapeList<T>();
        foreach (var x in a)
```

```
            result.Shapes.Add(x);
        foreach (var x in b)
            result.Shapes.Add(x);
        return result;
    }
    ShapeList<Rectangle> list1 = new ShapeList<Rectangle>();
    list1.Shapes.Add(new Rectangle(100, 200));
    ShapeList<Rectangle> list2 = new ShapeList<Rectangle>();
    list2.Shapes.Add(new Rectangle(150, 100));
    ShapeList<Rectangle> list = list1 + list2;
```

Узагальнені методи

```
public class Defaulter  // Реалізація у звичайному класі
{
  public T GetDefault<T>()
  {
      return default(T);
  }
}
public class Defaulter<T>  // warning
{
  public T GetDefault<T>()
  {
      return default(T) ;
  }
}
```

Узагальнені делегати

```
public delegate Tl MyDelegate<Tl, T2>(T2 opl, T2 op2) where Tl : T2;
```

Приклад. Реалізація однозв'язного параметризованого списку

```
class MyList<T> : ICollection<T>
{
    class Link<T>
    {
        public Link(T v) : this(v, null) { }
        public Link(T v, Link<T> n) { Value = v; Next = n; }
        public Link<T> Next { get; set; }
        public T Value { get; set; }
    }
    Link<T> head = null;
    int count = 0;
    public int Count { get { return count; } }
    public bool IsReadOnly { get { return false; } }
    public void Clear() { count = 0; head = null; }
    public bool Contains(T value)
    {
        foreach (T x in this)
            if (Comparer<T>.Default.Compare(x, value) == 0)
  // Direct comparison x == value causes error
                return true;
        return false;
    }
    public void CopyTo(T[] array, int index)
    {
        foreach (T x in this)
            array[index++] = x;
    }
    public bool Remove(T item)
    {
        if (count == 0)
            return false;
       if (count > 0 && Comparer<T>.Default.Compare(head.Value, item) == 0)
```

```
    {
        head = head.Next;
        count--;
        return true;
    }
    Link<T> currLink = head;
    while (currLink.Next != null)
       if (Comparer<T>.Default.Compare(currLink.Next.Value, item) == 0)
        {
            currLink.Next = currLink.Next.Next;
            count--;
            return true;
        }
        else
            currLink = currLink.Next;
    return false;
}
public void Add(T v)
{
    Link<T> newLink = new Link<T>(v, head);
    head = newLink;
    count++;
}
public IEnumerator<T> GetEnumerator()
{
    if (count > 0)
    {
        Link<T> currLink = head;
        do
        {
            yield return currLink.Value;
            currLink = currLink.Next;
        } while (currLink != null);
    }
    else
        yield break;
}
IEnumerator IEnumerable.GetEnumerator()
```

```
        {
            Link<T> currLink = head;
            while (currLink != null)
            {
                yield return currLink.Value;
                currLink = currLink.Next;
            }
        }
    }
    static void Main()
    {
        MyList<int> myList = new MyList<int>();
        myList.Add(10);
        myList.Add(5);
        myList.Add(3);
        myList.Remove(10);
        PrintCollection(myList);
        Console.WriteLine(myList.Contains(5));
    }
```

Методи розширень

Методи розширень дозволяють розширяти функціональні можливості типів без внесення змін у самі типи. Це робиться шляхом написання нових методів (методів розширень), які можна викликати через екземпляри згаданих типів. Кроки

1) створити не узагальнений статичний невкладений клас;
2) додати створений метод у вигляді статичного методу, перший параметр якого увляє собою екземпляр того типу, для якого буде викликатися метод розширень і має вигляд `this <extended_class> instance`;

3) викликати метод розширень із використанням звичайного синтаксису;

```
static class MyExtensionClass
{
    public static string ToTitleCase(this string inputString)
    {
        if (inputString == "")
            return "";
        string result = "";
        string[] stringArray = inputString.Split(' ');
        foreach (string s in stringArray)
          if (s.Length > 0)
          result += s.Substring(0,1).ToUpper()+s.Substring(1,s.Length-1).ToLower()+" ";
          else
            result += " ";
        return result.Substring(0, result.Length - 1);
    }
}
    static void Main()
    {
        string s1 = "It is a testing string";
        Console.WriteLine(s1.ToTitleCase());
    }
```

Обробка подій

```
static int counter = 0;
static string stringToDisplay = "This string will appear one letter by a time. ";
static void WriteChar(object source, ElapsedEventArgs e)
{ Console.Write(stringToDisplay[counter++ % stringToDisplay.Length]); }
static void Main()
{
    Timer myTimer = new Timer(100);
    myTimer.Elapsed += new ElapsedEventHandler(WriteChar);
    myTimer.Start();
    Console.ReadKey();
}
```

Приклад визначення події

```
public delegate void MessageHandler(string messageText);
public class Connection
{
    public event MessageHandler MessageArrived;
    private Timer pollTimer;
    public Connection()
    {
        pollTimer = new Timer(200);
        pollTimer.Elapsed += new ElapsedEventHandler(CheckForMessage);
    }
    public void Connect() { pollTimer.Start(); }
    public void Disconnect() { pollTimer.Stop(); }
    private static Random random = new Random();
    private void CheckForMessage(object source, ElapsedEventArgs e)
    {
        Console.WriteLine("Checking for new messages.");
        if (MessageArrived != null && random.Next(9) == 0)
            MessageArrived("Hello");
    }
}
static void DisplayMessage(string message)
{
```

```
    Console.WriteLine("Message: {0}", message);
}
static void Main()
{
    Connection myConnection = new Connection();
    myConnection.MessageArrived += Program.DisplayMessage;
    myConnection.Connect();
    Console.ReadKey();
}
```

Мова структурованих запитів LINQ

Лямбда-вирази

Лямбда-вираз складається із трьох частин:

1) список параметрів, які можуть бути нетипізованими;

2) операція =>;

3) тіло виразу – оператори мови C#.

Приклад 1.

```
delegate double BinaryOperation(double x, double y);
static void PrintResultTable(int n, BinaryOperation op)
{
    for(int i = 1;  i <= n; i++)
    {
        for (int j = 1; j <= n; j++)
            Console.Write("{0,3} ", op(i, j));
        Console.WriteLine();
    }
}
static void Main()
{
    Console.WriteLine("Addition table");
    PrintResultTable(5, (x, y) => x + y);
    Console.WriteLine("Multiplication table");
    PrintResultTable(5, (x, y) => x * y);
```

```
        Console.WriteLine("Division table");
        PrintResultTable(5, (x, y) => x / y);
    }
```

Приклад 2

```
    class Point
    {
        public double X;
        public double Y;
    }
    static void Main(string[] args)
    {
        double[] numbers = { 2, 3.5, 4, -2.3, 1, 5, 7.2, 3 };
        var cnt = numbers.Count(x => Math.Truncate(x) == x);
        Console.WriteLine("Count of positive numbers = {0}", cnt);
        var points = new List<Point>
            {
                new Point(){X = 2,  Y = -3},
                new Point(){X = -1, Y = -7},
                new Point(){X = 4,  Y = 5}
            };
        var sum = points.Sum(p => p.X);
        var avg = points.Average(x => x.Y);
        Console.WriteLine("Sum of abscissae = {0}\nAverage ordinate = {1}",
sum, avg);
    }
```

Приклад 3.

```
string[] s = { "one", "two", "three", "four", "five" };
List<string> myList = new List<string>(s);
var t = myList.FindAll(x => x.EndsWith("e"));
PrintCollection(t);
```

Приклад 4.

```
// public static void Sort<T>(T[] array, Comparison<T> comparison)
// public delegate int Comparison<T>(T x, T y)
```

```
Array.Sort(s,(x,y) => x.Length-y.Length != 0 ? x.Length-y.Length:x.CompareTo(y));
```

Метод розширення Aggregate – акумулююча функція

1) `public static T Aggregate<T>(this IEnumerable<T> source, Func<T, T, T> func);`

Перший елемент послідовності `source` служить початковим значенням статистичної операції. Результат `func` заміняє попереднє підсумкове значення. Метод `Aggregate` повертає результат останнього виконання func.

```
string str = s.Aggregate((x,y)=> x + "\n" + y); // конкатенація
Console.WriteLine(str);
```

2) `public static TAccumulate Aggregate<T, TAccumulate> (this IEnumerable<T> source, TAccumulate seed, Func<TAccumulate, T, TAccumulate> func);`

```
а) int total = s.Aggregate<string, int>(0, (x, y) => x + y.Length);
Console.WriteLine("total length of array is {0}", total);
б) string str = s.Aggregate<string, string>("", (x,y)=> x + y[0]);
Console.WriteLine(str);
```

3) `public static TResult Aggregate<T, TAccumulate, TResult> (this IEnumerable<T> source, TAccumulate seed, Func<TAccumulate, T, TAccumulate> func, Func<TAccumulate, TResult> resultSelector);`

```
const string message = "United nation organization";
var acronym = message.Split(' ').Aggregate<string, string, string>("", (x, y) => y != "" ? x + y[0] : x, x => x.ToUpper());
```

Використання LINQ to Objects. Синтаксис запитів та методів

LINQ to Objects призначений для оперування колекціями даних, які підтримують узагальнений інтерфейс `IEnumerable<T>`.

Основні операції запитів LINQ наведені у наступній таблиці:

Таблиця операцій LINQ

Операція	**Опис**
`from, in`	Вказує найменування даних та джерело даних, з якої вони вибераються. Загальна форма: `from` *змінна* `in` *джерело*
`where`	Вказує обмеження, яким повинні задовольняти вибрані елементи. Синтаксис: `where` *логічний_вираз*
`select`	Вказує конкретний тип елементів даних, які повертаються у якості результату запиту. Синтаксис: `select` *вираз*
`orederby, ascending, descending`	Впорядковує результуючий набір. Синтаксис: `orderby` *елемент* [`ascending|descending`]
`group, by, into`	Групування по заданому критерію. Синтаксис: `group` *змінна* `by` *ключ*
`join, on, equals, into`	Виконує з’єднання джерел на основі значень ключів. Синтакис: `from` *змінна_діапазону_A* `in` *джерело_A* `join` *змінна_діапазону_B* `in` *джерело_B* `on` *змінна_діапазону_A.властивість* `equals` *змінна_діапазону_B.властивість*

Усі запити **мають закінчуватися операціями** `select` або `group`!

Застосування запитів LINQ для вибору даних із звичайних масивів

1. Отримати перелік додатних елементів масиву

```
int[] nums = { 1, -2, 3, 0, -4, 5 };
var posNums = from n in nums
              where n > 0
              select n;
PrintCollection(posNums);
```

Для запитів використовується механізм *відкладеного виконання*. Вони виконуються лише тоді, коли відбувається ітерація по результуючій послідовності. Запити можуть виконуватися неодноразово. При цьому використовується *поточний стан джерела даних*. Тобто змінна даних зумовлює змінну результату запиту.

```
nums[2] = -10;
Console.WriteLine("After change:");
PrintCollection(posNums);
```

2. У масиві зберігаються ціни товарів. Вивести у порядку спадання усі ціни, які потрапляють у проміжок від 200 до 300 грн.

```
int[] prices = { 200, 500, 300, 1000, 150, 400, 230, 600 };
var query = from x in prices
    where x >= 200 && x <= 300
orderby x descending
```

```
select x;
PrintCollection(query);
```

3. Вивести максимальну ціну тих товарів, ціна яких не вища за середню ціну.

```
var result = (from x in prices
    where x <= prices.Average()
    select x).Max();
Console.WriteLine(result);
```

Використання вкладених операцій `from`

У запиті операція `from` може зустрічатися неодноразово, якщо результат запиту формується на основі даних кількох джерел.

Приклад. Сформувати усі пари елементів двох цілочислових масивів, перший елемент яких менший за другий:

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3, 4};
var query = from x in a
    from y in b
    where x < y
    select string.Format("({0}, {1})", x, y);
PrintCollection(query);
```

Групування даних за допомогою операції `group`

Операція групування дає змогу згрупувати результати запиту за ключами. Результатом виконання операції групування є послідовність, яка складається з елементів типу `IGrouping<TKey, TElement>`, тобто типом результату запиту є змінна типу `IEnumerable<IGrouping<TKey, TElement>>`. У інтерфейсі `IGrouping` передбачена властивість `Key` — значення ключа для відповідної групи елементів.

Приклад. Згрупувати веб-адреси за їх доменом верхнього рівня, наприклад `.com` чи `.org`.

```
string[] websites =
    {
        "hsNameA.com", "hsNameD.com", "hsNameG.tv",
        "hsNameB.net", "hsNameE.org", "hsNameH.net",
        "hsNameC.net", "hsNameF.org", "hsNamel.tv"
    };
var webAddress = from addr in websites
                 where addr.LastIndexOf(".") != -1
                 group addr by addr.Substring(addr.LastIndexOf("."));
foreach (var sites in webAddress)
{
    Console.WriteLine("Domain name: {0}", sites.Key);
    foreach (var site in sites)
    {
```

```
                Console.WriteLine("\t{0}", site);
            }
        }
```

Продовження запитів із використанням `into`

Оператор `into` використовується збереження результату запиту у "тимчасову" змінну, яка надалі використовується для формування кінцевого результату. Синтаксис:

`into` *змінна тіло_запиту*

Приклад. Вивести групи доменних імен, які містять більше двох веб-адресів

```
var query = from addr in websites
            where addr.LastIndexOf(".") != -1
            group addr by addr.Substring(addr.LastIndexOf(".")) into wa
            where wa.Count() > 2
            orderby wa.Key
            select wa;
```

Операція `let`

Операція `let` використовується для тимчасового збереження значень величин усередині тіла запиту.

Приклад. Заданий масив рядків. Вивести усі символи, які зустрічаються у рядках та їх частоту входження, у порядку зростання частот.

```
string[] words = {"alpha", "beta", "gamma"};
var chars = from word in words
    let charArray = word.ToCharArray()
    from ch in charArray
    group ch by ch into charGroup
    orderby charGroup.Count()
    select new{Character = string.Format("'{0}'", charGroup.Key),
               Reps = charGroup.Count()};
PrintCollection(chars);
```

З'єднання двох послідовностей за допомого `join`

Оператор `join` застосовується у випадку необхідності з'єднувати дані із кількох джерел. Операція `join` виконує з'єднання на основі співпадання значень ключів за допомогою ключового слова `equals`. Формат результату з'єднання залежить від його типу. Основні типи з'єднань:

- Внутрішнє з'єднання.
- Групове з'єднання.
- Ліве зовнішнє з'єднання.

Розглянемо приклад з'єднання по імені наступних масивів даних:

```
var personnel = new[] {
    new {Name = "Smith", Job = "driver", Salary = 5000},
    new {Name = "Carter", Job = "turner", Salary = 4000},
```

```
        new {Name = "Miller", Job = "driver", Salary = 6000},
        new {Name = "Jackson", Job = "driver", Salary = 5500},
        new {Name = "Simpson", Job = "engineer", Salary = 6500},
    };
    var bonuses = new[] {
        new {Name = "Carter", Amount = 2000},
        new {Name = "Smith", Amount = 1500},
        new {Name = "Jackson", Amount = 500},
        new {Name = "Smith", Amount = 200},
        new {Name = "Simpson", Amount = 1000},
        new {Name = "Smith", Amount = 200},
        new {Name = "Carter", Amount = 2000}
    };
```

Внутрішнє з'єднання

```
    var query = from person in personnel
                join bonus in bonuses on person.Name equals bonus.Name
                select new { person.Name, person.Salary, bonus.Amount };
```

Кожний запис про працівника у масиві `personnel` з'єднується із записом про премію працівника з масиву `bonuses`. Результат:

```
{ Name = Smith, Salary = 5000, Amount = 1500 }
{ Name = Smith, Salary = 5000, Amount = 200 }
{ Name = Smith, Salary = 5000, Amount = 200 }
{ Name = Carter, Salary = 4000, Amount = 2000 }
{ Name = Carter, Salary = 4000, Amount = 2000 }
{ Name = Jackson, Salary = 5500, Amount = 500 }
{ Name = Simpson, Salary = 6500, Amount = 1000 }
```

Групові з’єднання

Результатом групового з’єднання є ієрархічна послідовність, яка пов’язує елементи лівого джерела даних з послідовністю (одним чи кількома елементами) відповідних йому даних правого джерела. Якщо елементу лівого джерела не відповідає жодний елемент правого джерела, то join створює порожню послідовність. У групових з’єднаннях використовується ключове слово `into` для того, щоб вказати ім’я послідовності елементів правого джерела.

Приклад 1. Вивід прізвищ працівників та їх премій:

```
var query = from person in personnel
            join bonus in bonuses on person.Name equals bonus.Name
                  into bonusGroup
            select new { person.Name, bonusGroup};
foreach (var item in query)
{
    Console.WriteLine(item.Name + ":");
    foreach (var bonus in item.bonusGroup)
    {
        Console.WriteLine("\t {0}", bonus.Amount);
    }
}
```

Приклад 2. Знаходження прізвищ працівників та їх премій, які не менші за 1000 грн.:

```
var query = from person in personnel
            join bonus in bonuses on person.Name equals bonus.Name
```

```
                    into bonusGroup
                select new {person.Name,
               Bonuses = from b in bonusGroup
                    where b.Amount >= 1000
                    select b.Amount};
```

Приклад 3. Вивід прізвищ працівників та грошових сум, які вони отримали (зарплата + премії):

```
var query = from person in personnel
            join bonus in bonuses on person.Name equals bonus.Name
                  into bonusGroup
            select new { person.Name,
           Income = person.Salary + bonusGroup.Sum(x => x.Amount) };
PrintCollection(query);
```

Ліві зовнішні з’єднання

У лівому зовнішньому з’єднанні повертаються усі елементи лівої послідовності (навіть якщо їм не відповідає жоден елемент правої послідовності). Для виконання цього типу з’єднань потрібно використовувати метод `DefaultIfEmpty`.

Приклад.

```
var query = from person in personnel
            join bonus in bonuses on person.Name equals bonus.Name
           into bonusGroup
             from b in bonusGroup.DefaultIfEmpty(new {Name =
              person.Name, Amount = 0})
             select new { person.Name, B = b };
foreach (var item in query)
{
```

```
            Console.WriteLine("{0} {1}", item.Name, item.B.Amount);
        }
```

LINQ. Синтаксис методів

Проекції. Побудова нових об’єктів

```
var list = personnel.Select(x => new { Name = x.Key, Salary = x.Value.salary });
PrintCollection(list.Where(x => x.Salary > 4000));
```

Метод OrderBy

```
var list = personnel.OrderByDescending(x=>x.Value.job);
PrintCollection(list.Select(x=>new{x.Key, x.Value.job}));
var list = personnel.OrderBy(x=>x.Value.job).ThenBy(x=>x.Key);
```

Метод Distinct

```
var list = personnel.Select(x => x.Value.job).Distinct();
PrintCollection(list);
```

Any та All

```
bool result1 = personnel.Any(x=>x.Key.StartsWith("M"));
bool result2 = personnel.All(x => x.Value.job != "plumber");
```

Агрегатні функції

```
var query = personnel.Select(x => new{x.Key, x.Value.salary})
.Where(x=> x.salary >= personnel.Values.Average(y=>y.salary));
```

Групування даних з використанням методу GroupBy

```
var query = personnel.GroupBy(x=>x.Value.job);
```

```
        PrintCollection(query.Select(x=>new{Job = x.Key, TotalSalary = 
x.Sum(y=>y.Value.salary)}));
```

або

```
        var query = personnel.GroupBy(x => x.Value.job);
        foreach (var x in query)
            Console.WriteLine("{0} {1}", x.Key, x.Sum(y => y.Value.salary));
```

Приклади запитів

```
        List<Entry> employees = new List<Entry>
        {
            new Entry { name = "Smith", job = "doctor", salary = 4000 },
            new Entry { name = "Smith", job = "worker", salary = 3000 },
            new Entry { name = "Taylor", job = "doctor", salary = 2000 },
            new Entry { name = "Adams", job = "manager", salary = 5000 },
            new Entry { name = "Carter", job = "worker", salary = 2500 }
        };

    var result = employees.Select((x, id) => new { index = id,  name = 
    x.name, job = x.job}).Where(x => x.job == "worker");

      var result = employees.Where(x => x.salary <= employees
                    .Where(y => y.job == x.job).Average(y => y.salary))
                    .Select(x => new {x.name, x.job});
       Print(result);

      var result = employees.Where(x => employees
                    .Any(y => y.name == x.name && !y.Equals(x)));
      var result = employees.GroupBy(x => x.job)
                  .Select(x => new {job = x.Key, avgSalary = 
```

```
                x.Average(y => y.salary)}).OrderBy(x => x.avgSalary);
```

Take i Skip

```
var query = personnel.OrderBy(x => x.Value.salary).Take(3);
PrintCollection(query.Select(x => new {x.Key, x.Value.salary}));
```

First, FirstOrDefault i Last

```
var query = personnel.Last(x => x.Value.job == "driver");
Console.WriteLine(query.Key);
```

З’єднання таблиць

Join – операція зв’язування даних двох таблиць на основі спільного ключа

```
var bonusArray = new[]
{
    new{name = "Miller", bonus = 1000},
    new{name = "Carter", bonus = 2000},
    new{name = "Smith",  bonus = 1500},
};
var query = personnel.Join(bonusArray, x => x.Key, x=>x.name,
    (x,y) => new{name = x.Key, wage = x.Value.salary + y.bonus});
PrintCollection(query);
```

Групові з’єднання GroupJoin

```
var bonusArray = new[]
{
    new{name = "Miller", bonus = 1000},
```

```
            new{name = "Smith",  bonus = 1500},
            new{name = "Miller", bonus = 1500},
        };
        var query = personnel.GroupJoin(bonusArray, x => x.Key, x => x.name,
            (x, y) => new { name = x.Key, wage = x.Value.salary + y.Sum(z =>
z.bonus)});
        PrintCollection(query);
```

Робота з документами XML в .NET Framework

XML (Extensible Markup Language) — це технологія збереження даних у простому текстовому форматі. XML підтримує ієрархічну (деревоподібну) модель даних. XML часто використовується для передачі даних між клієнтською та серверною машинами.

Елементи XML

Елемент XML складається відкриваючого дескриптора (ім'я елемента у кутових дужках, наприклад `<myElement>`), даних усередині елемента та закриваючого дескриптора з іменем елемента, наприклад `</myElement>`.

Наприклад, для збереження назв книги можна використати наступний елемент:

```
<book>The Murders in the Rue Morgue</book>
```

Елементи можуть містити у собі інші елементи:

```
<book>
    <title>The Murders in the Rue Morgue</title>
    <author>Edgar Allan Poe</author>
</book>
```

Атрибути. Дані у тілі елементів можуть зберігатися всередині атрибутів, які мають наступну форму:

```
name = “value”
```

Наприклад

```
<book title="The Murders in the Rue Morgue" pages = “60”></book>
```

Структура документа XML

Кожний елемент повинен містити єдиний кореневий елемент. Наступний документ XML містить оголошення та кореневий елемент books

```
<?xml version=”1.0”?>
<books>
    <book>Moby Dick</book>
    <book>Ulysses</book>
</books>
```

Засоби роботи з XML у .NET Framework

Засоби підтримки опрацювання документів XML зосередженні у просторах імен System.Xml та System.Xml.Linq. Використання `LINQ to XML` дає змогу писати більш компактний код. Основними класами у `LINQ to XML` є класи `XNode` (абстрактний), `XDocument, XElement, XText` та `XAttribute`.

Приклад. Нехай у документі `writers.xml` містяться наступні дані

```
<writers>
    <writer>
        J.F. Cooper
        <book title="The Pathfinder" pages="520" price="70" />
        <book title="The Pioneers" pages="364" price="50.50" />
    </writer>
    <writer>
        J.R.R. Tolkien
        <book title="The Fellowship of the Ring" pages="404" price="60" />
        <book title="The Two Towers" pages="346" price="50" />
        <book title="The Hobbit" pages="420" price="65" />
    </writer>
 </writers>
```

Завантаження та збереження

Наступний фрагмент коду дає змогу вивести вміст XML-документа

```
XDocument xDoc = XDocument.Load("../../writers.xml");
Console.WriteLine(xDoc);
xDoc.Root.Element("writer").Save("Cooper.xml");
```

Для виводу усіх атрибутів документа можна використати наступну функцію:

```
static void showAttributes(XNode node)
{
    if (node.NodeType == XmlNodeType.Element)
    {
        XElement element = node as XElement;
        foreach (var a in element.Attributes())
            Console.WriteLine("{0} = {1}", a.Name, a.Value);
        foreach (var item in element.Nodes())
```

```
            showAttributes(item);
        }
    }
```

Для виклику потрібно додати рядок

```
        showAttributes(xDoc.Root);
```

Добавлення, зміна та видалення елементів у документі XML

```
        // Добавлення нового автора
        XElement jackLondon = new XElement("writer", "J. London",
            new XElement("book",
                new XAttribute("title", "White Fang"),
                new XAttribute("pages", "30"),
                new XAttribute("price", "20")
                ),
            new XElement("book",
                new XAttribute("title", "The Call of the Wild"),
                new XAttribute("pages", "25"),
                new XAttribute("price", "20")
                )
            );
        xDoc.Root.Add(jackLondon);
       // Добавлення інформації про нову книгу
        XElement newBook = new XElement("book",
            new XAttribute("title", "The Return of the King"),
            new XAttribute("pages", "430"),
            new XAttribute("price", "60"),
            new XElement("yearOfBirth", ""),
            new XElement("yearOfDeath", "1973")
            );
        foreach(var node in xDoc.DescendantNodes())
            if (node.NodeType == XmlNodeType.Text &&
               (node as XText).Value.Contains("Tolkien")){
                node.Parent.AddFirst(newBook);
                break;
            }
        XElement x = newBook.Element("yearOfBirth");
```

```
    x.Name = "Birth";
    x.Value = "1892";
    x.NextNode.Remove();
    foreach(var node in newBook.FirstNode.Ancestors())
        Console.WriteLine(node.Name); // Вивід ланцюжка вузлів-предків
    Console.WriteLine(xDoc);
```

Приклади

1. Сумісне використання `LINQ to Objects` та `LINQ to XML`:

```
List<Record> personnel = new List<Record>{
    new Record{name = "Smith", job = "driver", salary = 5000},
    new Record{name = "Carter", job = "turner", salary = 4000},
    new Record{name = "Miller", job = "driver", salary = 5500},
    new Record{name = "Jackson", job = "driver", salary = 5500 }
};
var bonuses = new[]
{
    new{name = "Miller", bonus = 1000},
    new{name = "Carter", bonus = 2000},
    new{name = "Miller", bonus = 500},
    new{name = "Smith",  bonus = 1500},
    new{name = "Miller", bonus = 600},
    new{name = "Smith",  bonus = 400},
};
XElement drivers = new XElement("drivers",
    from p in personnel
    where p.job == "driver"
    join b in bonuses on p.name equals b.name into bonusGroup
    select new XElement("driver",
        new XAttribute("name", p.name),
        new XAttribute("job", p.job),
        from item in bonusGroup
        select new XElement("bonus", item.bonus),
        new XElement("totalBonus"), bonusGroup.Sum(x => x.bonus)));
```

2. Формування переліку книг:

```
XElement writers = XElement.Load("../../writers.xml");
```

```
XElement books = new XElement("books",
    from book in writers.Descendants("book")
    select new XElement("book",
        new XElement("author", ((XText)book.Parent.FirstNode).Value),
        new XAttribute("pages", book.Attribute("pages").Value),
        new XAttribute("price", book.Attribute("price").Value)
        )
    );
Console.WriteLine(books);
```

3. Формування XML-документу, який містить дані про кількість виданих книг та списки цих книг, згруповані по роках.

```
var xDoc = XDocument.Load("../../writers2.xml");
var element = new XElement("Books",
    from book in xDoc.Descendants("book")
    group book by book.Attribute("year").Value into bookGroup
    select new XElement ("published",
        new XAttribute("year", bookGroup.Key),
        new XAttribute("count", bookGroup.Count()),
        new XElement("books",
            from item in bookGroup
                select new XElement("book",
                        new  XText(item.Attribute("title").Value)))
        );
Console.WriteLine(element);
```

Література

1. Брауде Э. Технология разработки программного обеспечения.— СПб.: Питер, 2004.
2. Гагарина Л. Г. и др. Технология разработки программного обеспечения. — М.: ИНФРА-М, 2008.
3. Лавріщева К. М. Програмна інженерія. — К. : Академперіодика, 2008.
4. Лаврищева Е. М. Методы программирования. Теория, инженерия, практика. — К.: Наукова думка, 2006.
5. Лаврищева Е. М., Грищенко В. Н. Сборочное программирование. — К.: Наукова думка, 2009.
6. Орлик С. Программная инженерия. Конструирование программного обеспечения, 2006.
7. Соммервилл И. Инженерия программного обеспечения. 6 издание. — М.: Вильямс, 2002.
8. Уотсон К., Нейгел К, Педерсен Я.Х. и др. Visual C# 2008. Базовый курс. — М.: "Вильямс", 2011. — 1216 с.
9. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4. — М.: "Вильямс", 2010. — 1392 с.
10. Фленов М. Библия C#. 2-е издание. — СПб.: БХВ, 2011. — 560 с.
11. Шилдт Г. C# 4.0. Полное руководство. — М.: "Вильямс", 2011. — 1056 с.