«ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ»
МІНІСТЕРСТВА ОСВІТИ І НАУКІ УКРАЇНИ

С.Ю. Борю

Посібник із самостійного вивчення об'єктно-орієнтованого програмування мовою C# Об'єкто орієнтоване програмування

навчально-методичний посібник для студентів
природничих спеціальностей

Затверджено вченою радою
ЗНУПротокол № ___ від _____

Запоріжжя 2025

УДК: 004.4з (075.8)

ББК: з-973.2 - D18 я7з
Б848

Борю С.Ю. Посібник із самостійного вивчення об'єктно-орієнтованого програмування мовою C#. Навчально-методичний посібник для студентів природничих спеціальностей – Запоріжжі; ЗНУ, 2025, – 111 с.

Розглядаються оператори мови та методи їх застосування для реалізації об'єктно-орієнтованого програмування мовою C#. Наводиться велика кількість прикладів та фрагментів програм. Посібник орієнтований на самостійну роботу студентів.

Рецензент – Гоменюк С.І.

Відповідальний за випуск – Матвейшена Н.В.

ЗМІСТ

Вступ 5
1. Класи 5
1.1. Класи та об'єкти 5
1.1.1. Ініціалізатори об'єктів 8
1.1.2. Часткові класи 9
1.2. Модифікатори доступу 9
1.3. Константи та поля для читання 12
1.3.1. Константи 12
1.3.2. Поля для читання 13
1.4. Властивості та інкапсуляція 13
1.4.1. Інкапсуляція 16
1.4.2. Автоматичні властивості 17
1.4.3. Ініціалізація автовластивостей 17
1.5. Перевантаження методів та операторів 18
1.5.1. Перевантаження операторів 18
1.6. Статичні члени та модифікатор static 22
1.6.1. Статичні члени класу 22
1.6.2. Статичний конструктор 24
1.6.3. Статичні класи 24
1.7. успадкування 25
1.7.1. Доступ до членів базового класу із класу-спадкоємця 26
1.7.2. Ключове слово base 27
1.8. Поліморфізм та перевизначення методів 28
1.8.1. Ключове слово base 30
1.8.2. Заборона перевизначення методів 31
1.9. Абстрактні класи 31
1.10. Клас System.Object та його методи 34
1.10.1. ToString 34
1.10.2. Метод GetHashCode 35
1.10.3. Отримання типу об'єкта та метод GetType 35
1.10.4. Метод Equals 36
1.11. Перетворення типів 36
1.12. Узагальнені типи 40
1.12.1. Значення за замовчуванням 42
1.12.2. Статичні поля узагальнених класів 42
1.12.3. Обмеження універсальних типів 42
1.12.4. Використання кількох універсальних параметрів 44
1.12.5. Узагальнені методи 44
1.12.6. Наслідування узагальнених типів 45
1.13. Створення класів винятків та клас Exception 45
1.14. Анонімні типи 48
1.15. Методи розширення 49
1.16. Простір імен, псевдоніми та статичний імпорт 50

1.16.1. Простір імен 50
1.16.2. Псевдоніми 51
1.17. Контрольні питання 53
2. Інтерфейси 53
2.1. Введення в інтерфейси 53
2.1.1. Інтерфейси у перетвореннях типів 56
2.1.2. Узагальнені інтерфейси 56
2.1.3. Явне застосування інтерфейсів 57
2.2. Клонування об'єктів. Інтерфейс ICloneable 58
2.3. Сортування об'єктів. Інтерфейс IComparable 60
2.3.1. Застосування компаратора 61
2.4. Коваріантність та контраваріантність узагальнених інтерфейсів 62
2.5. Контрольні питання 65
3. Делегати, події та лямбди 65
3.1. Делегати 65
3.2. Події 72
3.3. Анонімні методи 76
3.4. Лямбди 77
3.5 Коваріантність та контраваріантність делегатів 80
3.5.1 Коваріантність та контраваріантність в узагальнених делегатах 81
3.6 Делегати Action, Predicate та Func 83
3.6.1 Action 83
3.6.2 Predicate 83
3.6.3 Func 84
3.7 Вбудовані методи (Expression-Bodied) 85
3.8. Контрольні питання 86
4. null та типи Nullable 87
4.1. Значення null та Nullable-типи 87
4.1.1. Рівність об'єктів 88
4.1.2. Перетворення типів Nullable 88
4.1.3. Оператор ?? 89
4.2. Елвіс оператор 89
4.3 Контрольні питання 91
5. Практичний приклад 91
5.1. Створення проекту бібліотеки класів 91
5.2. Створення класів. Частина 1 94
5.3. Створення класів. Частина 2 99
5.4. Створення головного проекту 103
Література 108

Вступ

Сьогодні мова програмування C# вважається однією з найпотужніших мов, що швидко розвиваються і затребуваних в IT-галузі. На ньому пишуться різні програми: від невеликих десктопних програм до великих веб-порталів і веб-сервісів, що обслуговують щодня мільйони користувачів.

Порівняно з іншими мовами C# досить молодий, але в той же час він уже пройшов великий шлях. Перша версія мови вийшла разом із релізом Microsoft Visual Studio .NET у лютому 2002 року. Поточною версією мови є версія C# 12, яка підтримується в рамках платформи .NET 8. Версія C# 12 була випущена разом із .NET 8 14 листопада 2023 року.

Мова C# є об'єктно-орієнтованою і в цьому плані вона багато перейняла з мов Java і C++. Наприклад, C# підтримує поліморфізм, інкапсуляцію, успадкування, навантаження операторів, статичну типізацію. Об'єктно-орієнтований підхід дозволяє вирішити завдання з побудови великих, але в той же час гнучких, масштабованих і додатків, що розширюються. C# продовжує активно розвиватися і з кожною новою версією з'являється все більше цікавих функціональностей, як, наприклад, лямбди, динамічне зв'язування, асинхронні методи і т.д. C# є мовою з подібним синтаксисом і близький, у цьому відношенні, до C++ і Java.

В даний час технології. Текст даної роботи заснований на матеріалах сайтів https://msdn.microsoft.com/library, http://metanit.com, http://www.intuit.xu та, в першу чергу, призначений для самостійного вивчення мови програмування C# на платформі .NET.

Нижче не формально описуються оператори мови, які допомагають програмісту реалізовувати у своїх програмах ідеї об'єктно-орієнтованого програмування. Передбачається, що читач знайомий із базовими основами мови програмування C#.

1. Класи

1.1. Класи та об'єкти

C# є повноцінною об'єктно-орієнтованою мовою. Це означає, що програму на C# можна у вигляді взаємозалежних взаємодіючих між собою об'єктів.

Описом об'єкта є клас, а об'єкт є екземпляр цього класу. Можна провести таку аналогію. У нас у всіх є деяке уявлення про людину – наявність двох рук, двох ніг, голови, травної, нервової системи, головного мозку тощо. Є певний шаблон – цей шаблон можна назвати класом. Реально існуюча людина (фактично екземпляр даного класу) є об'єктом цього класу. Часто клас уподібнюють і вважають програмною моделлю деякої предметної області. А реалізація цієї моделі з конкретними параметрами та властивостями – це об'єкт класу.

Клас визначається за допомогою ключового слова class, наприклад:

```
class Book {

}
```

Вся функціональність класу представлена його членами – полями (полями називаються змінні класу), властивостями, методами та подіями. Створимо клас Book:

```
class Book {
public string  name;
public string  author;
public int year;

public void  Info() {
Console.WriteLine("Книга '{0}' (автор {1}) була "+"
видана в {2} році", name, author, year);
}
}
```

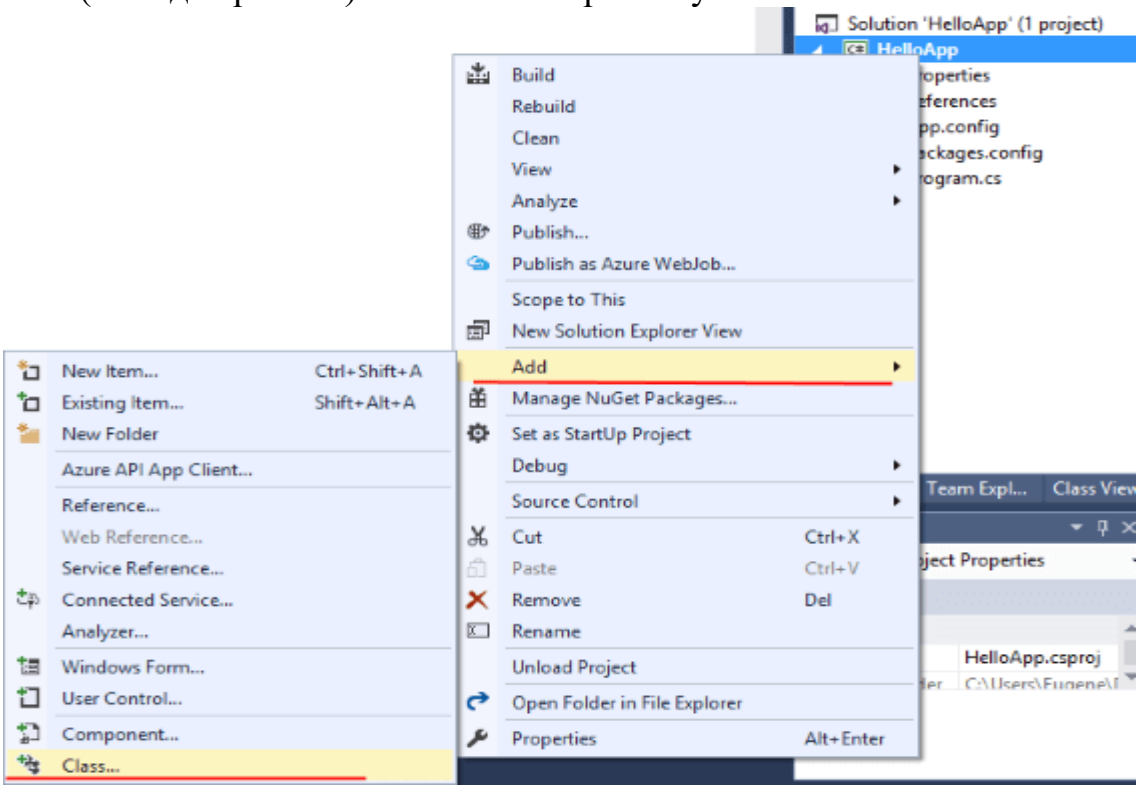
Крім звичайних методів у класах використовуються також і спеціальні методи, що називаються**конструкторами**. Конструктори викликаються під час створення нового об'єкта цього класу. Відмінною рисою конструктора є те, що його назва має співпадати з назвою класу:

```
class Book {
public string  name;
public string  author;
public int year;
public Book() {

}

Public Book(string  name, string  author, int year) {
this.name = name;
this.author = author;
this.year = year;
}
public void  Info() {
Console.WriteLine("Книга '{0}' (автор {1}) "+
                          "була видана в {2} році",
                                  name, author, year);
}
}
```

Одне з призначень конструктора – початкова ініціалізація членів класу. В даному випадку використовувалися два конструктори. Один «порожній» - немає вхідних параметрів та коду цього методу. Другий конструктор заповнює поля класу початковими значеннями, які передаються через параметри.

Оскільки імена параметрів і імена полів (name, author, year) у разі збігаються, ми використовуємо ключове слово this. Це ключове слово є посиланням на поточний клас. Тому у виразі this.name = name; перша частина this.name означає, що name - це поле поточного класу, а чи не назва параметра name. Якби параметри та поля називалися по-різному, то використовувати слово це було б необов'язково.

Тепер використовуємо клас у новій програмі. Створимо новий проект. Потім натиснемо правою кнопкою миші на назву проекту у вікні Solution Explorer (Оглядач рішень) і в меню виберемо пункт Class .



У діалоговому вікні, що з'явилося, дамо новому класу ім'я Book і натиснемо кнопку Add (Додати). До проекту буде додано новий файл Book.cs, що містить клас Book.

Змінимо в цьому файлі код класу Book на наступний код:

```
class Book {
public string  name;
public string  author;
public int year;

public Book() {
name = "невідомо";
author = "невідомо";
        year = 0;
}
Public Book(string  name, string  author, int year) {
this.name = name;
this.author = author;
this.year = year;
```

```
}
public void  GetInformation() {
Console.WriteLine("Книга '{0}' (автор {1})"+ " була
видана в {2} році",
                                        name, author, year);
}
}
```

Тепер перейдемо до коду файлу Program.cs та змінимо метод Main класу Program:

```
class  Program {
static void Main(string [] args) {
Book b1 = new Book("Війна і мир", "Л. Н. Толстой",
1869);
b1.GetInformation();
      Book b2 = New Book();
b2.GetInformation();
Console.ReadLine();
}
}
```

Якщо виконати цю програму, то на консоль виведеться інформація про книги b1 та b2. Зверніть увагу, щоб створити новий об'єкт з використанням конструктора, необхідно використовувати ключове слово new. Оператор new створює об'єкт класу і виділяє йому область в оперативної пам'яті (де зберігаються значення параметрів об'єкта).

1.1.1. Ініціалізатори об'єктів

Для класу Book (див. попередній приклад) можна встановити послідовно значення для всіх трьох полів (параметрів) класу (оскільки ці поля оголошені як public):

```
Book b1 = New Book();
b1.name = "Війна та мир";
b1.author = "Л. Н. Толстой";
b1.year = 1869;

b1.GetInformation();
```

Але можна також використовувати**ініціалізатор об'єктів**:

```
Book b2 = New Book();
b2 = new Book { name = "Батьки та діти", author = "І. С.
Тургенєв",
year = 1862};
b2.GetInformation();
```

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

1.1.2. Часткові класи

Часткові класи (partial class) дають можливість записати функціонал одного класу у кількох файлах. Наприклад, у попередньому прикладі весь код класу Book знаходиться в одному файлі Book.cs. Але можна помістити код цього класу в різних файлах. У цьому випадку необхідно перед визначенням класу записати ключове слово partial.

Допустимо, в одному файлі буде код:

```
partial class Book {
public string  name;
public string  author;
public int year;
}
```

А в іншому файлі буде:

```
partial class Book {
Public Book(string  name, string  author, int year) {
this.name = name;
this.author = author;
this.year = year;
}

public void  GetInformation() {
Console.WriteLine("Книга '{0}' (автор {1}) була "+"
видана в {2} році", name, author, year);
}
}
```

Так можна «розділити» код методів та властивостей одного класу.

1.2. Модифікатори доступу

Усі члени класу – поля, методи, властивості – всі вони мають модифікатори доступу. Модифікатори доступу дозволяють встановити допустиму область видимості для членів класу. Тобто контекст, у якому можна використовувати цю змінну чи спосіб. Наприклад, у попередній темі поля класу Book оголошували публічними (тобто модифікатором public).

У C# застосовуються такі модифікатори доступу:

- **public**: публічний, загальнодоступний клас чи член класу. Такий член класу доступний з будь-якого місця в коді, а також інших програм і збірок.
- **private**: закритий клас або член класу Представляє повну протилежність модифікатору public. Такий закритий клас чи член класу доступний лише з коду у тому класі чи контексті.
- **protected**: такий член класу доступний з будь-якого місця в поточному класі або похідних (успадкованих) класах.

- **internal**: клас і члени класу з подібним модифікатором доступні з будь-якого місця коду в тій самій збірці, однак він недоступний для інших програм та збірок (як у випадку з модифікатором public).

protected internal: поєднує функціонал двох модифікаторів Класи та члени класу з таким модифікатором доступні з поточної збірки та похідних класів.

Оголошення полів класу без модифікатора доступу рівнозначне їхньому оголошення з модифікатором private. Класи, оголошені без модифікатора, за промовчанням мають доступ до internal.

Розглянемо приклад – класу State:

```
public class  State {
int a; // однаково, що private int a;
private int b; // поле доступне тільки з поточного
                    // класу
protected int c; // доступно з поточного класу та //
похідних класів
internal int d; // Доступно в будь-якому місці програми
protected internal int e; // доступно у будь-якому місці
// програми та з //класів-спадкоємців
public int f; // Доступно в будь-якому місці програми, //
а також для інших // програм та збірок
private void  Display_f() {
Console.WriteLine("Змінна f = {0}", f);
}

public void  Display_a() {
Console.WriteLine("Змінна a = {0}", a);
}

internal void  Display_b() {
Console.WriteLine("Змінна b = {0}", b);
}

protected void  Display_e() {
Console.WriteLine("Змінна e = {0}", e);
}
}
```

Так як клас State оголошений з модифікатором public, він буде доступний з будь-якого місця програми, а також інших програм і збірок. Клас State має п'ять полів кожного рівня доступу. Плюс одна змінна без модифікатора, яка закрита за замовчуванням.

Також є чотири методи, які виводитимуть значення полів класу на консоль. Зверніть увагу, що оскільки всі модифікатори дозволяють використовувати члени класу всередині даного класу, то і всі змінні класу, в тому числі закриті, доступні всім його методам (всі друзі знаходиться в контексті класу State).

Тепер розглянемо, як можна використовувати змінні цього класу в програмі (тобто в методі Main класу Program):

```
class  Program {
static void Main(string [] args) {
State state1 = New State();

// Присвоїти значення змінної a у нас не вийде,
// оскільки вона закрита і клас Program її бачить
// І цей рядок середа підкреслить як неправильну

state1.a = 4; //Помилка, отримати доступ не можна

// те саме стосується і змінної b
state1.b = 3; // Помилка, отримати доступ не можна

        // присвоїти значення змінної з тим же не
//вийти,
// Оскільки клас Program не є класом - // спадкоємцем
класу State
state1.c = 1; // Помилка, отримати доступ не можна

        // змінна d з модифікатором internal //доступна з
будь-якого місця програми
        // тому «спокійно» надає їй значення
state1.d = 5;

// змінна e також доступна з будь-якого місця // програми
state1.e = 8;

// Змінна f загальнодоступна
state1.f = 8;

// Спробуємо вивести значення змінних

// Оскільки цей метод оголошений як private,
        // його можна використовувати лише // усередині
класу State
state1.Display_f() ; // Помилка, отримати // доступ не
можна

// Так як цей метод оголошений як protected, // а клас
Program не є спадкоємцем // класу State
state1.Display_e(); // Помилка, отримати доступ // Не
можна
```

```
// Загальнодоступний метод
state1.Display_a();

// Метод доступний з будь-якого місця програми
state1.Display_b();

Console.ReadLine();
}
}
```

Таким чином, допустимо використовувати тільки змінні d, e та f, тому що їх модифікатори дозволяють їх використовувати в даному контексті. Доступні лише два методи: state1.Display_a() та state1.Display_b(). Однак, оскільки значення змінних a і b не були встановлені, ці методи виведуть нулі, так як значення змінних типу int за замовчуванням ініціалізуються нулями.

Незважаючи на те, що модифікатори public і internal схожі на свою дію, але досить сильно відрізняються. Класи та члени класу з модифікатором public також будуть доступні й іншим програмам, якщо дані класу помістити в динамічну бібліотеку dll і потім її використовувати в цих програмах.

Завдяки такій системі модифікаторів доступу можна приховувати деякі моменти реалізації класу з інших частин програми. Таке приховування часто називають інкапсуляцією.

1.3. Константи та поля для читання

Полями класу називаються звичайні змінні рівня класу. Раніше розглядалися змінні – їх оголошення та ініціалізацію. У мові передбачені поля, що містять константи та поля для читання.

1.3.1. Константи

Особливістю констант і те, що й значення можна встановити лише один раз. Наприклад, якщо у програмі є деякі змінні, які не повинні змінювати значення (наприклад, число PI, число e і т.д.), їх можна оголосити їх константами. Для цього використовується ключове слово const:

```
const double PI = 3.1415926;
const double E = 2.71;
```

При використанні констант треба пам'ятати, що оголосити їх можна лише один раз і при цьому їх треба обов'язково ініціалізувати (задати значення).

```
class MathLib {
public const double  PI = 3.141;
public const double  E = 2.81;
public const double  K;
public MathLib() {
K = 2.5;
      // !!! помилка - константа має бути // визначена
до компіляції
```

```
} // тобто. «опис» та «завдання значення» //в одному
операторі
}
class  Program {
static void Main(string [] args) {
MathLib.E=3.8; // !!! помилка
// константу не можна встановити // кілька разів
}
}
```

Також зверніть увагу на синтаксис звернення до константи. Так як це статичне поле (за замовчуванням), то його використання не треба створювати об'єкт класу. До таких полів потрібно звертатися, використовуючи ім'я класу.

1.3.2. Поля для читання

Поля для читання схожі на константи: їх також не можна встановити двічі, проте їх можна встановлювати під час виконання програми, наприклад, у конструкторі, що з константами є неприпустимим.

Поле для читання оголошується із ключовим словом readonly:

```
class MathLib {
public readonly double  K;

public MathLib(double  _k) {
K = _k; // поле читання може бути // визначено після
компіляції
}
}

class  Program {
static void Main(string [] args) {
        MathLib MathLib = New MathLib (3.8);
Console.WriteLine(mathLib.K); // 3.8
mathLib.K = 7.6; // Помилка - поле для читання
                            // не можна встановити поза //
свого класу
Console.ReadLine();
}
}
```

1.4. Властивості та інкапсуляція

Окрім звичайних методів у мові C# передбачені спеціальні методи доступу, які називають властивості. Вони забезпечує простий доступ до полів класу, дозволяють дізнатися їх значення, виконати їхню установку або зміну.

Стандартний опис властивості має наступний синтаксис:

```
[модифікатор_доступу] повертається_тип довільна_назва {
```

```
                    // код якості
}
```

Наприклад:

```
class Person {
private string  name;

public string  Name {
get {
return name;
}

set {
name = value;
}
}
}
```

У цьому прикладі є закрите поле name і є загальнодоступною властивістю Name. Хоча вони мають практично однакову назву за винятком регістру, але це не більше ніж стиль, назви у них можуть бути довільними і не обов'язково повинні збігатися.

Через це властивість можна керувати доступом до змінної name. Стандартне визначення властивості містить блоки get та set. У блоці get повертається значення поля, а блоці set його значення встановлюється. Параметр value представляє значення, що передається.

Приклад використання:

```
Person p = new Person();

   // Встановлюємо властивість – спрацьовує блок set
   // значення "Tom" і є передане // у властивість value
p.Name = "Tom";

   // Отримуємо значення якості та
   // привласнюємо його змінною -
   // Спрацьовує блок get
string personName = p.Name;
```

Можливо, може виникнути питання, навіщо потрібні властивості, якщо можливо у цій ситуації обходитися звичайними полями класу? Але властивості дозволяють запрограмувати додаткову логіку, яка може бути необхідна для ухвалення рішення про присвоєння змінної класу будь-якого значення. Наприклад, перевірка за віком:

```
class Person {
private int age;

public int Age {
set {
```

```
if (value < 18) {
Console.WriteLine("Вік має бути"+ "більше 18");
} else {
age = value;
}
}
get { return age; }
}
}
```

Блоки set та get не обов'язково одночасно повинні бути присутніми у властивості. Наприклад, можна закрити властивість від установки або зміни, але не читання (значення властивості можна отримувати). Для цього достатньо не кодувати блок set. І, навпаки, можна видалити блок get, тоді можна буде встановити значення властивості, але не можна значення отримати:

```
class Person {
private string  name;
// властивість лише читання
public string  Name {
get {
return name;
}
}

private int age;
// властивість лише запису
public int Age {
set {
age = value;
}
}
}
```

Можливе застосування модифікаторів доступу не тільки до всієї властивості, але і до окремих блоків - або get, або set. При цьому якщо модифікатор застосовується до одного з блоків, то до іншого його застосовувати не можна:

```
class Person {
private string  name;

public string  Name {
get {
return name;
}

private set {
```

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

Тепер закритий блок ми зможемо використовувати тільки в даному класі - у його методах, властивостях, конструкторі, але ніяк не в іншому класі:

```
Person p = new Person("Tom", 24);

// Помилка - set оголошений з модифікатором private
p.Name = "John";

Console.WriteLine(p.Name);
```

1.4.1. Інкапсуляція

Вище було розглянуто, що через характеристики можна встановлюється доступ до приватних змінних класу. Подібне приховування стану класу від втручання ззовні часто називають механізмом інкапсуляції. Використання модифікаторів доступу типу private захищає змінну від зовнішнього доступу. Для управління доступом у багатьох мовах програмування використовуються спеціальні методи, які часто називають гетерами і сеттерами (або аллокаторами). У C# їх роль, як правило, виконують властивості.

Наприклад, є певний клас Account, у якому визначено поле sum, що становить суму:

```
class Account {
public int sum;
}
```

Оскільки змінна sum є публічною, то в будь-якому місці програми можна отримати до неї доступ і змінити її значення. У тому числі встановити якесь неприпустиме значення, наприклад, негативне. Навряд чи подібна поведінка є бажаною. Тому застосовується інкапсуляція для обмеження доступу до змінної sum та приховання її всередині класу:

```
class Account {
private int sum;
public int Sum {
get {return sum;}
set {
if (value > 0) {
sum = value;
}
}
}
```

```
}
```

1.4.2. Автоматичні властивості

Властивості керують доступом до полів класу. Але якщо в класі передбачено досить багато полів, то визначати кожне поле та писати для нього однотипну властивість досить незручно. Тому з версії .NET 4.0 у фреймворк було додано автоматичні властивості. Вони мають скорочене оголошення:

```
class Person {
public string  Name { get; set; }
public int Age {get; set; }

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

У цьому прикладі створюються поля властивостей Name і Age. Тільки їх створює не програміст у коді, а компілятор автоматично.

З одного боку, автоматичні характеристики досить комфортні. З іншого боку, стандартні властивості мають ряд переваг - наприклад, можуть інкапсулювати додаткову логіку перевірки значення. Також не можна створити автоматичну властивість тільки для запису або читання, як у випадку зі стандартними властивостями.

1.4.3. Ініціалізація автовластивостей

У C# 6.0 (Visual Studio 2015) було додано таку функціональність, як ініціалізація автовластивостей:

```
class Person {
public string  Name { get; set; } = "Tom";
public int Age {get; set; } = 23;
}
class  Program {
static void Main(string [] args) {
        Person person = new Person();
Console.WriteLine(person.Name); // Tom
Console.WriteLine(person.Age); // 23
Console.Read();
}
}
```

Як видно з прикладу, якщо не вказати для об'єкта person значення властивостей Name та Age, то діятимуть значення за замовчуванням.

Ще одна зміна торкнулася визначення автовластивостей. Наприклад, якщо в C# 5.0 захотілося зробити автовластивість доступною для встановлення тільки з класу, то треба було вказувати private set:

```
class Person {
public string  Name { get; private set; }
```

```
Public Person(string  n) {
Name = n;
}
}
```

Крім як із класу Person цю властивість неможливо встановити.

C# 6.0 писати private set необов'язково:

```
class Person {
public string  Name {get;}
Public Person(string  n) {
Name = n;
}
}
```

1.5. Перевантаження методів та операторів

Іноді виникає необхідність створити той самий метод, але з різним набором параметрів. І залежно від існуючих параметрів застосовувати певну версію методу. Допустимо, є наступний клас State:

```
class State {
public string  Name { get; set; } // Назва
public int Population { get; set; } // населення
public double  Area { get; set; } // площа
}
```

Припустимо, що необхідно визначити метод для нападу на іншу державу – метод Attack. Перша реалізація методу прийматиме як параметр об'єкт State - тобто держава, на яку нападаємо:

```
publicvoid  Attack (State enemy) {}
```

Але припустимо, що необхідно визначити також версію даного методу, де вказуватиме не лише держава, а й кількість військ, за допомогою яких здійснюється напад на ворога. Тоді можна просто додати другу версію цього методу:

```
publicvoid  Attack (State enemy) {
// тут код методу
}
publicvoid  Attack (State enemy, int army) {
// тут код методу
}
```

1.5.1. Перевантаження операторів

Поруч із методами, можливо, також перевантажувати оператори.

При перевантаженні оператора вказуються модифікатори public static, оскільки оператор, що перевантажується, буде використовуватися для всіх об'єктів даного класу. Далі йде назва типу, що повертається, і після нього ключове слово operator. Потім назва оператора та параметри:

```
publicstatic повертається_тип operator оператор(параметри)
```

Подивимося на прикладі. Нехай є клас State чи держава. Але раптом знадобиться завдання об'єднувати держави. Це можна вирішити застосовуючи навантаження оператора "+". Крім того, зазвичай «перевантажують» так само оператори порівняння - ">" і "<", за допомогою яких порівнюватимемо дві держави:

```
class State {
public string  Name { get; set; } // Назва
public int Population { get; set; } // населення
public double  Area { get; set; } // площа

public static State operator +(State s1, State s2) {
string  name=s1.Name;
int people = s1.Population+s2.Population;
double  area = s1.Area+s2.Area;

// Повертаємо нову об'єднану державу
return new State { Name = name, Area = area, Population =
people };
}

public static bool operator <(State s1, State s2) {
if (s1.Area < s2.Area) {
return true;
} else {
return false;
}
}

public static bool operator >(State s1, State s2) {
if (s1.Area > s2.Area) {
return true;
} else {
return false;
}
}
}
```

Оскільки всі перевантажені оператори – бінарні – тобто проводяться над двома операндами (об'єктами), то для кожного методу перевантаження передбачено по два параметри. Унарні оператори мають один параметр. Використовуємо перевантажені оператори у програмі:

```
static void Main(string [] args) {
    State s1 = new State {Name = "State1", Area = 300,
Population = 100};

    State s2 = new State {Name = "State2", Area = 200,
```

```
Population = 70};

if (s1 > s2) {
Console.WriteLine("Держава s1 більше" + "держави s2");
} else if (s1 < s2) {
Console.WriteLine("Держава s1 менша" + "держави s2");
} else {
Console.WriteLine("Держави s1 і s2 рівні");
}

State s3 = s1 + s2;
Console.WriteLine("Назва держави: {0}", s3.Name);
Console.WriteLine("Площа держави: {0}", s3.Area);
Console.WriteLine("Населення держави: {0}",
s3.Population);

Console.ReadLine();
}
```

При навантаженні операторів треба враховувати, що не всі оператори допускають навантаження.

Оператори	Можливість навантаження
+,-,!,~,++,-,true,false	Ці унарні оператори можна перевантажити.
+,-,*,/,%,&,|,^,<<,>>	Ці бінарні оператори можна перевантажити.
==,!=,<,>,<=,>=	Оператори порівняння можуть бути перевантажені (але див. примітку після цієї

	таблиці).
`&&, ||`	Умовні логічні оператори неможливо знайти перевантажені, але вони оцінюються з допомогою & і |, які можна перевантажені.
`[]`	Оператор індексування масиву не може бути перевантажений, але можна визначити індексатори.
`(T) x`	Оператор приведення типів не може бути перевантажений, але можна визначити нові оператори перетворення (див. explicit та implicit).

`+=,-=,*=,/=,%=,&=, |=,^=,<<=,>>=`	Оператори присвоювання не можуть бути перевантажені, але +=, наприклад, оцінюється за допомогою +, який може бути перевантажений.
`=,.,?:,??,->,=>,f(x),as,checked,unchecked,default,delegate,is,new,sizeof,typeof`	Ці оператори не можуть бути перевантажені.
Примітка:При перевантаженні операторів порівняння вони повинні перевантажуватись парами; тобто якщо оператор == перевантажується, оператор != теж повинен перевантажуватися.	

Для перевантаження оператора в класі користувача потрібно створити метод (в цьому класі) з правильною сигнатурою. Метод повинен мати ім'я operator X, де X — ім'я або символ оператора для перевантаження. Унарні оператори мають один параметр, а бінарні оператори мають два параметри. У кожному випадку один параметр повинен бути того ж типу, що і клас або структура, що оголошує його.

1.6. Статичні члени та модифікатор static

1.6.1. Статичні члени класу

У загальному випадку, щоб використовувати якийсь клас, встановлювати та отримувати його поля, використовувати його методи, необхідно було створити його об'єкт. Однак, якщо цей клас має статичні методи, то, щоб отримати до них доступ, необов'язково створювати об'єкт цього класу. Наприклад, створимо новий клас Algorithm і додамо до нього дві функції (методу) для обчислення числа Фібоначчі та факторіалу:

```
class Algorithm {
public static double  pi = 3.14;
```

```
public static int Factorial(int x) {
if (x == 1) {
return 1;
} else {
return x * Factorial (x - 1);
}
}
public static int Fibonachi(int x) {
if (x == 0) {
return 1;
}
if (x == 1) {
return 1;
} else {
return Fibonachi(x - 1) + Fibonachi(x - 2);
}
}
}
```

Ключове слово static щодо змінної і методів показує, що ці члени будуть доступні всього класу, тобто статичними. Тепер використовуємо їх у програмі:

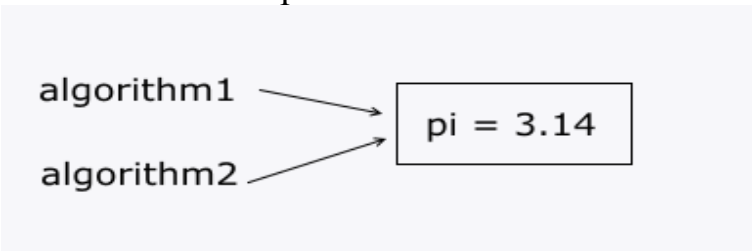
```
intnum1 = Algorithm.Factorial(5);
intnum2 = Algorithm.Fibonachi(5);

Algorithm.pi = 3.14159;
```

При використанні статичних членів класу не треба створювати об'єкт класу, а потрібно звернутися до них «безпосередньо», вказуючи як ім'я об'єкта – ім'я класу.

Для статичних полів створюватиметься ділянка в пам'яті, яка буде спільною для всіх об'єктів класу. Наприклад, якщо створити два об'єкти класу Algorithm:

```
Algorithm algorithm1 = новий Algorithm();
Algorithm algorithm2 = новий Algorithm();
```

Обидва ці об'єкти будуть зберігати посилання на одну ділянку в пам'яті, де зберігатися значення pi:



Нерідко статичні поля використовуються для зберігання лічильників. Наприклад, нехай є клас State, і необхідно мати лічильник, який дозволяв би дізнатися скільки об'єктів State створено:

```
class State {
private static int counter = 0;
public State() {
```

```
counter++;
}

public static void  DisplayCounter() {
Console.WriteLine("Створено {0} об'єктів State",
counter);
}
}
class  Program {
static void Main(string [] args) {
        State state1 = New State();
        State state2 = New State();
        State state3 = New State();
        State state4 = new State();
        State state5 = New State();

State.DisplayCounter(); // 5

Console.Read();
}
}
```

1.6.2. Статичний конструктор

Крім звичайних конструкторів, у класу також можуть бути статичні конструктори. Статичні конструктори виконуються при першому створенні об'єкта даного класу або першому зверненні до його статичних членів (якщо такі є):

```
class State {
static State() {
Console.WriteLine("Створено першу державу");
}
}
class  Program {
static void Main(string [] args) {
State s1 = New State(); // тут спрацює //статичний
конструктор
State s2 = New State();

Console.ReadLine();
}
}
```

1.6.3. Статичні класи

Статичні класи оголошуються з модифікатором static і можуть містити лише статичні поля та методи. Наприклад, у прикладах вище клас Algorithm мав лише статичні поля та методи. Потрібно створювати об'єкт цього класу,

щоб використати його функціональність. Тому цей клас можна оголосити як статичний:

```
static class Algorithm {
//............
}
```

1.7. успадкування

Спадкування (inheritance) є одним із ключових моментів ООП. Його сенс у тому, що ми можемо розширити функціональність вже існуючих класів з допомогою додавання нового функціоналу чи зміни старого. Або "зменшити рівень абстракції". Нехай є наступний клас Person, який описує окрему людину:

```
class Person {
private string  _firstName;
private string  _lastName;

public string  FirstName {
get { return _firstName; }
set {_firstName = value; }
}
public string  LastName {
get { return _lastName; }
set {_lastName = value; }
}

public void  Display() {
Console.WriteLine(FirstName + " " + LastName);
}
}
```

Створимо клас, який описує співробітника підприємства – клас Employee. Оскільки цей клас реалізовуватиме той самий функціонал, як і клас Person (оскільки співробітник - це і людина), було б раціонально зробити клас Employee похідним (чи спадкоємцем) від класу Person. У такій ситуації клас Person називається базовим класом або класом батьком:

```
class Employee : Person {

}
```

*Після імені класу записується двокрапка*та вказується ім'я базового класу для даного класу. Для класу Employee базовим є Person, і тому клас Employee успадковує ті самі властивості, методи, поля, які є у класі Person.

Таким чином, успадкування реалізує відношення "є", об'єкт класу Employee також є об'єктом класу Person:

```
static void Main(string [] args) {
    Person p = new Person { FirstName = "Bill", LastName
= "Gates" };
```

```
p.Display();
p = new Employee { FirstName = "Denis", LastName =
"Ritchi"};
p.Display();
Console.Read();
}
```

І оскільки об'єкт Employee є і об'єктом Person, можна визначити змінну: Person p = new Employee().

Усі класи за умовчанням можуть успадковуватись. Однак тут є низка обмежень:

- Чи не підтримується множинне спадкування, клас може успадковуватися тільки від одного класу. (Проблема множинного успадкування реалізується за допомогою концепції інтерфейсів, про які буде сказано нижче).
- При створенні похідного класу треба враховувати тип доступу до базового класу - тип доступу до похідного класу повинен бути таким самим, як і у базового класу, або більш суворим. Тобто якщо базовий клас має тип доступу internal, то похідний клас може мати тип доступу internal або private, але не public.
- Якщо клас оголошений з модифікатором sealed, то цього класу не можна успадковувати і створювати похідні класи. Наприклад, наступний клас не допускає створення спадкоємців:

```
sealed class Admin {
. . . . . .
}
```

1.7.1. Доступ до членів базового класу із класу-спадкоємця

Повернемося до розгляду класів Person та Employee. Хоча Employee успадковує весь функціонал від класу Person, але:

```
class Employee : Person {
public void  Display() {
Console.WriteLine(_firstName);
}
}
```

цей код буде помилковим, оскільки змінна _firstName оголошена з модифікатором private і тому доступ до неї має тільки клас Person. Але в класі Person визначено загальнодоступну властивість FirstName, яку можна використовувати. Тому наступний код «працюватиме нормально»:

```
class Employee : Person {
public void  Display() {
Console.WriteLine(FirstName);
}
}
```

Таким чином, похідний клас може мати доступ тільки до членів базового класу, які визначені з модифікаторами public, internal, protected і protected internal.

1.7.2. Ключове слово base

Тепер додамо до наших класів конструктори:

```
class Person {
public string  FirstName {get; set; }
public string  LastName {get; set; }

Public Person(string  fName, string  lName) {
FirstName = fName;
LastName = lName;
}

public void  Display() {
Console.WriteLine(FirstName + " " + LastName);
}
}

class Employee : Person {
public string  Company { get; set; }

Public Employee(string  fName, string  lName, string
comp) : base(fName, lName) {
Company = comp;
}
}
```

Клас Person має стандартний конструктор, який встановлює дві властивості. Оскільки клас Employee успадковує та встановлює ті ж властивості, що й клас Person, то логічно було б не писати ще раз код установки, а викликати відповідний код класу Person. (У загальному випадку властивостей і параметрів, які треба встановити, може бути набагато більше).

За допомогою ключового слова base можна звернутися до базового класу. В цьому випадку в конструкторі класу Employee треба встановити ім'я, прізвище та компанію. Але ім'я та прізвище можна передати на встановлення в конструктор базового класу, тобто в конструктор класу Person, за допомогою виразу base(fName, lName).

Приклад використання розглянутих класів:

```
static void Main(string [] args) {
    Person p = new Person("Bill", "Gates");
p.Display();
    Employee emp = new Employee ("Tom", "Simpson",
"Microsoft");
emp.Display();
```

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

1.8. Поліморфізм та перевизначення методів

Поліморфізм є третім ключовим аспектом об'єктно-орієнтованого програмування та передбачає здатність до зміни функціоналу, успадкованого від базового класу.

Поліморфізм передбачає визначення поліморфного інтерфейсу в базовому класі – набір членів класу, які можуть бути перевизначені у класі-спадкоємці.

Методи, які можна перевизначити, у базовому класі позначаються модифікатором virtual. Такі методи називають віртуальними. Вони представляють поліморфний інтерфейс (також частиною поліморфного інтерфейсу можуть бути абстрактні члени класу).

При визначенні класу-спадкоємця та успадкування методів базового класу, можна вибрати одну з таких стратегій:

- Звичайне успадкування всіх членів базового класу у класі-спадкоємці;
- Перевизначення членів базового класу у класі-спадкоємці;
- Приховування членів базового класу у класі-спадкоємці;

Перша стратегія є досить простою. Припустимо, є наступна пара класів Person та Employee:

```
class Person {
public string  FirstName {get; set; }
public string  LastName { get; set; }
Public Person(string  lName, string  fName) {
FirstName = fName;
LastName = lName;
}

public virtual void  Display() {
Console.WriteLine(FirstName + " " + LastName);
}
}

class Employee : Person {
public string  Company { get; set; }
Public Employee(string  lName, string  fName, string
comp) : base(fName, lName) {
Company = comp;
}
}
```

У базовому класі Person метод Display() визначено модифікаторами virtual, тому цей метод може бути перевизначений. Але клас Employee успадковує його без зміни:

```
class  Program {
static void Main(string [] args) {
        Person p1 = new Person("Bill", "Gates");
p1.Display(); // виклик методу Display // із класу Person

        Person p2 = New Employee("Tom", "Johns",
"UnitBank");
p2.Display(); // виклик методу Display // із класу Person

        Employee p3 = New Employee("Sam", "Toms",
"CreditBank");
p3.Display(); // виклик методу Display // із класу Person
Console.Read();
}
}
```

Консольний висновок:

```
Bill Gates
Tom Johns
Sam Toms
```

Друга стратегія - перевизначення методів базового класу в класі-спадкоємці передбачає використання ключового слова override:

```
class Employee : Person {
public string  Company { get; set; }
Public Employee(string  lName, string  fName, string
comp) : base(fName, lName) {
Company = comp;
}

public override void  Display() {
        Console.WriteLine(FirstName + " " + LastName + "
працює у компанії " + Company);
}
}
```

Клас Person залишається тим самим, у ньому метод Display оголошується як віртуальний. У цьому випадку поведінка об'єкта Employee зміниться:

```
Person p1 = new Person("Bill", "Gates");
p1.Display(); // виклик методу Display із класу Person

Person p2 = New Employee("Tom", "Johns", "UnitBank");
p2.Display(); // виклик методу Display із класу Employee

Employee p3 = New Employee("Sam", "Toms", "CreditBank");
p3.Display(); // виклик методу Display із класу Employee
```

Консольний висновок:

```
Bill Gates
```

```
Tom Johns працює у компанії UnitBank
Sam Toms працює в компанії CreditBank
```

При третій стратегії можна просто визначити в класі-спадкоємці метод з тим самим ім'ям, без перевизначення за допомогою слова override:

```
class Employee : Person {
public string  Company { get; set; }
Public Employee(string  lName, string  fName, string
comp) : base(fName, lName) {
Company = comp;
}

public new void  Display() {
Console.WriteLine(FirstName + " " + LastName +
                        "працює в компанії" + Company);
}
}
```

У цьому випадку метод Display() у Employee приховує метод Display() із класу Person. Щоб явно приховати метод із базового класу, використовується ключове слово new, хоча в принципі воно необов'язкове, за замовчуванням це робиться неявно.

Використання у програмі:

```
Person p1 = new Person("Bill", "Gates");
p1.Display(); // виклик методу Display із класу Person

Person p2 = New Employee("Tom", "Johns", "UnitBank");
p2.Display(); // виклик методу Display із класу Person

Employee p3 = New Employee("Sam", "Toms", "CreditBank");
p3.Display(); // виклик методу Display із класу Employee
```

Консольний висновок:

```
Bill Gates
Tom Johns
Sam Toms працює в компанії CreditBank
```

Зверніть увагу на відмінності у другому випадку (p2.Display()) за різних стратегій.

1.8.1. Ключове слово base

За допомогою ключового слова base можна звертатись до інших членів базового класу. Виклик base.Display(); буде викликати метод Display() із класу Person:

```
class Employee : Person {
public string  Company { get; set; }

Public Employee(string  lName, string  fName,
```

```
                        string  comp) : base(fName, lName) {
Company = comp;
}

public override void  Display() {
base.Display();
Console.WriteLine("Місце роботи:" + Company);
}
}
```

1.8.2. Заборона перевизначення методів

За необхідності можна заборонити перевизначення методів та властивостей у класах спадкоємців. Для цього використовується модифікатор sealed:

```
class Employee : Person {
public string  Company { get; set; }

Public Employee(string  lName, string  fName, string
comp) : base(fName, lName) {
Company = comp;
}

public override sealed void  Display() {
base.Display();
Console.WriteLine("Місце роботи:" + Company);
}
}
```

При створенні методів з модифікатором sealed треба враховувати, що sealed застосовується в парі з override, тобто тільки в методах, що перевизначаються.

І в цьому випадку не можна перевизначити метод Display у класі, успадкованому від Employee.

1.9. Абстрактні класи

Крім звичайних класів у C# є абстрактні класи. Анотація клас схожий на звичайний клас. Він може мати змінні, методи, конструктори, властивості. І він зазвичай також містить лише опис деякого методу без його реалізації. Тому не можна створити об'єкт або екземпляр абстрактного класу. Абстрактні класи лише надають базовий функціонал для класів-спадкоємців. А похідні класи вже реалізують цей функціонал.

Для визначення абстрактних класів використовується ключове слово abstract:

```
abstract class Human {
public int Length { get; set; }
public double  Weight { get; set; }
}
```

Крім звичайних методів, абстрактний клас може мати абстрактні методи. Подібні методи визначаються за допомогою ключового слова abstract і не мають жодного функціоналу (реалізації):

```
publicabstract void  Display();
```

У цьому похідний клас повинен перевизначити і продати всі абстрактні способи і якості, які у базовому абстрактному класі. При перевизначенні у похідному класі такий метод оголошується з модифікатором override. Також слід врахувати, що якщо клас має хоча б одну абстрактну властивість або метод, він повинен бути визначений як абстрактний.

Абстрактні методи, як і віртуальні, є частиною поліморфного інтерфейсу. Але якщо у випадку з віртуальними методами говорять, що клас-спадкоємець успадковує реалізацію, то у випадку з абстрактними методами говорять, що успадковується інтерфейс, представлений цими абстрактними методами.

Для чого потрібні абстрактні класи? Допустимо, програмі для банківського сектора визначимо три класи:

- Person, який описує людину;
- Employee, який описує співробітника банку;
- Client, який представлятиме клієнта банку.

Очевидно, що класи Employee та Client будуть похідними від класу Person. І оскільки всі об'єкти будуть представляти або співробітника банку, або клієнта, безпосередньо від класу Person створювати об'єкти не можна. Тому є сенс зробити класу Person абстрактним.

```
abstract class Person {
public string  FirstName {get; set; }
public string  LastName {get; set; }

Public Person(string  lName, string  fName) {
FirstName = fName;
LastName = lName;
}

public abstract void  Display();
}

class Client : Person {
public string  Bank {get; set; }

Public Client(string  lName, string  fName, string
comp) : base(fName, lName) {
Bank = comp;
}

public override void  Display() {
Console.WriteLine(FirstName + " " +
            LastName + "має рахунок у банку" + Bank);
```

```
}
}
```

Іншим поширеним прикладом є система геометричних фігур. Насправді немає геометричної постаті як такої. Є коло, прямокутник, квадрат і т.д., але фігури немає. Однак і коло, і прямокутник мають щось спільне і є фігурами:

```
// абстрактний клас фігури
abstract class Figure {
float x; // x-координата точки
float y; // y-координата точки

public Figure(float x, float y) {
this.x=x;
this.y=y;
}

// абстрактний метод отримання периметра
public abstract float Perimeter();
// абстрактний метод отримання площі
public abstract float Area();
}
// Похідний клас прямокутника
class Rectangle : Figure {
public float Width { get; set; }
public float Height {get; set; }

// конструктор зі зверненням до // конструктора класу
Figure
public Rectangle(float x, float y, float width, float
height) : base(x, y) {
this.Width = width;
this.Height = height;
}
// Перевизначення отримання периметра
public override float Perimeter() {
return Width*2+Height*2;
}
// Переопреляння отримання площі
public override float Area() {
return Width * Height;
}
}
```

1.10. Клас System.Object та його методи

На початку всього ланцюжка успадкування класів знаходиться клас System.Object. Всі інші класи, навіть ті, які ми додаємо до свого проекту, а також базові типи, такі як System.Int32, є неявно похідними від класу Object. Навіть якщо не вказувати клас Object як базовий, за умовчанням неявно клас Object все одно стоїть на вершині наслідування ієрархії. Тому всі типи та класи можуть реалізувати ті методи, які визначені у класі Клас System.Object. Розглянемо ці способи.

1.10.1. ToString

Метод ToString служить отримання рядкового представлення даного об'єкта. Для базових типів просто виводитиметься їх рядкове значення:

```
int i = 5;
Console.WriteLine(i.ToString ()); // виведе число 5

double d = 3.5;
Console.WriteLine(d.ToString ()); // виведе число 3,5
```

Для класів цей метод виводить повну назву класу із зазначенням простору імен, у якому визначено цей клас. Можна перевизначити цей метод. Подивимося на прикладі:

```
class Human {
public string  Name { get; set; }

public override string  ToString () {
return Name;
}
}

class Animal { }

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

        Human human = new Human();
human.Name = "Том";
        Console.WriteLine(human.ToString ()); // виведе
// ім'я Том

        Animal anim = new Animal();
        Console.WriteLine(anim.ToString ()); // виведе //
назву класу Animal

Console.ReadLine();
}
```

```
}
```

Для класу Human метод ToString () перевизначено за допомогою ключового слова override та виводить значення властивості Name. А для класу Animal спрацьовує стандартна реалізація цього методу, яка виводить назву класу.

У разі задіяні обидві реалізації. Зверніть увагу, що якщо для об'єкта людського не буде встановлено ім'я, то буде виведено порожній рядок. Однак можна змінити реалізацію методу ToString , щоб у такому разі виводилося ім'я класу:

```
class Human {
public string  Name { get; set; }

public override string  ToString () {
if (Name == "" || Name==null) {
return base.ToString ();
} else {
return Name;
}
}
```

1.10.2. Метод GetHashCode

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

```
class Human {
public string  Name { get; set; }

public override int GetHashCode() {
return base.GetHashCode();
}
}
```

1.10.3. Отримання типу об'єкта та метод GetType

Метод GetType дозволяє отримати тип об'єкта:

```
Animal anim = new Animal();
Console.WriteLine(anim.GetType());
```

Наприклад, нехай ієрархія класів, де класи Employee і Client успадковуються від загального класу Person. Є кілька об'єктів класу Person, і потрібно звернутися лише до всіх клієнтів. У цьому випадку необхідно перевірити їх тип:

```
Person[] persons = new Person[]{new Client("Tom",
"Johnes", "SberBank", 200, 20),
new Employee("Bill",
                                                "Gates",
```

```
"Microsoft")
                                    };
foreach(Person p in persons) {
if (p.GetType() == typeof(Client)) {
Console.WriteLine("Це об'єкт класу Client");
}
}
```

За допомогою ключового слова**typeof**можна отримати тип класу об'єкта та порівняти його з типом об'єкта (метод GetType()). Якщо цей об'єкт є типом Client, то виконуємо певні дії.

1.10.4. Метод Equals

Метод Equals дозволяє порівняти два об'єкти на рівність:

```
class Human {
public string  Name { get; set; }

public override bool Equals(object obj) {
if (obj.GetType() != this.GetType()) return false;

Human human2 = (Human) obj;
return (this.Name == human2.Name);
}
}
```

Метод Equals приймає в якості параметра об'єкт будь-якого типу, який потім приводиться до поточного, якщо вони є об'єктами одного класу. Потім об'єкти порівнюються полем «ім'я». Якщо імена дорівнюють, повертається true - об'єкти рівні або false-об'єкти не рівні.

1.11. Перетворення типів

Раніше йшлося про перетворення об'єктів найпростіших типів. Розглянемо перетворення типів об'єктів класів. Припустимо, є така ієрархія класів:

```
abstract class Person {
public string  FirstName {get; set; }
public string  LastName { get; set; }

Public Person(string  lName, string  fName) {
FirstName = fName;
LastName = lName;
}

public abstract void  Display();
}

class Employee : Person {
```

```
public string  Company { get; set; }

Public Employee(string  lName, string  fName, string
comp) : base(fName, lName) {
Company = comp;
}

public override void  Display() {
        Console.WriteLine(FirstName + " " + LastName + "
працює у компанії " + Company);
}
}

class Client : Person {
int _sum; // Змінна для зберігання суми
public string  Bank {get; set; }
Public Client(string  lName, string  fName, string  comp,
int sum) : base(fName, lName) {
Bank = comp;
_sum = sum;
}

public override void  Display()
{
Console.WriteLine(FirstName + " " + LastName + " має
рахунок у банку " + Bank + " у сумі " + _sum.ToString
());
}
}
```

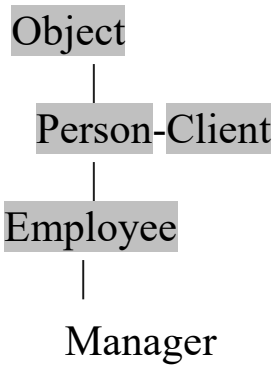
У цій ієрархії класів є наступний ланцюг успадкування: Object -> Person -> Employee | Client.

Розглянемо можливі перетворення типів:

```
// Об'єкт Employee також представляє тип object
object emp = new Employee("Bill", "Gates",
"Microsoft");
// об'єкт Client також представляє тип Person
Person cl = new Client ("Tom", "Johnes", "SberBank",
200);

//У класу Object немає методу Display, // тому
приводимо до класу Employee
((Employee) emp).Display();

// Клас Person має метод Display
cl.Display();
```

```
// клас Person не має властивості Bank,// тому
приводимо до класу Client
string bank = ((Client) cl). Bank;
```

Тут спочатку створюються дві змінні типів object та Person, які зберігають посилання на об'єкти Employee та Client відповідно. У разі «працює» неявне перетворення. Змінні представляють класи Object і Person, тому допустимо*неявне висхідне перетворення*- Перетворення до типів, що знаходяться вгорі ієрархії класів:



Однак, при застосуванні цих змінних необхідно використовувати явне перетворення. Оскільки змінна emp зберігає посилання об'єкт типу Employee, можна перетворити її до цього типу: ((Employee)emp).Display(). Те саме і зі змінною cl.

У той самий час не можна привести змінну emp до об'єкта Client, наприклад, так: ((Client)emp).Display();, Клас Client немає у ієрархії класів між Employee і Object.

Додамо ще клас Manager, який буде похідним від Employee (перебувати в самому «низу» ієрархії класів):

```
class Manager : Employee {
public Manager(string  lName, string  fName, string
comp) : base(lName,fName,comp) { }

public override void  Display() {
Console.WriteLine(FirstName + "" + LastName + "менеджер
компанії" + Company);
}
}
```

Так як об'єкт класу Manager в той же час є і співробітником компанії (тобто об'єктом Employee), то можна написати таке:

```
Employee man = new Manager("Джон", "Томпсон",
"Google");
man.Display(); // приведення не потрібне, // так як у
класі Employee є // метод Display
```

У цьому випадку здійснюється "висхідне" перетворення Manager до Employee. І оскільки метод Display є у класу Employee, то не треба виконувати перетворення змінної типу Manager.

Тепер подивимося на протилежну ситуацію: якщо застосувати «низхідні» перетворення неявно, під час виконання програми виникне помилка:

```
// об'єкт Employee може не представляти об'єкт Manager
Manager man = new Employee("Джон", "Томпсон", "Google");
man.Display();
```

У цьому випадку спроба неявно перетворити об'єкт Emloyee до типу Manager. Але якщо Manager є об'єктом типу Emloyee, об'єкт Emloyee не є об'єктом типу Manager.

Є кілька способів уникнути помилки під час виконання програми.

Перший спосіб (as) -використовувати ключове слово. За допомогою нього програма намагається перетворити вираз до певного типу, причому не викидає виняток. У разі невдалого перетворення вираз міститиме значення null:

```
Employee emp = new Employee("Джон", "Томпсон", "Google");
Manager man = emp as Manager;
if(man == null) {
Console.WriteLine("Перетворення пройшло невдало");
}else {
man.Display();
}
```

Другий спосіб (try)– «перехоплення» виключення**InvalidCastException**, що виникне в результаті спроби помилкового перетворення:

```
Employee emp = new Employee("Джон", "Томпсон",
"Google");
try{
Manager man = (Manager)emp;
man.Display();
}catch (InvalidCastException ex) {
Console.WriteLine(ex.Message);
}
```

Третій спосіб (is)полягає у перевірці допустимості перетворення за допомогою ключового слова is:

```
Employee emp = new Employee("Джон", "Томпсон", "Google");
if(emp is Manager) {
((Manager)emp).Display();
}else {
Console.WriteLine("Перетворення не допустимо");
}
```

Вираз emp is Manager перевіряє, чи є змінна emp об'єктом типу Manager. Але так як у даному випадку явно не є, то така перевірка поверне значення false, і перетворення не виконуватиметься.

1.12. Узагальнені типи

З виходом версії 2.0 фреймворк .NET став підтримувати узагальнені типи (generics), і навіть створення узагальнених методів і делегатів. Щоб розібратися особливо цього нововведення, спочатку подивимося на проблему, що могла виникнути до появи узагальнених типів. Подивимося на прикладі:

```
intx = 44;

System.Collections.ArrayList array = New
System.Collections.ArrayList();
// Упаковка значення x тип Object
array.Add(x);

// Розпакування значення типу int першого // елемента
колекції
inty = (int) array [0];

Console.WriteLine(y);
```

Тут використовується клас ArrayList, який представляє колекцію об'єктів. Щоб помістити об'єкт у колекцію, використовується метод Add. І хоча до колекції додаються числа та рядки, але по суті ArrayList містить колекцію значень типу Object. Це означає, що у виклику array.Add(x) значення змінної x спочатку буде "запаковано" (перетворено) у значення типу Object, а потім при отриманні елементів з колекції - навпаки, "розпаковано" у потрібний тип.

Упаковка (boxing) передбачає перетворення об'єкта значущого типу (наприклад, типу int) типу object. При упаковці загальномовне середовище CLR обгортає значення об'єкт типу System.Object і зберігає їх у керованій купі (хіпі). Розпакування (unboxing), навпаки, передбачає перетворення об'єкта типу object до значного типу. Упаковка та розпакування ведуть до зниження продуктивності, тому що системі треба здійснити необхідні перетворення.

Крім того, існує інша проблема – проблема безпеки типів. У наступній програмі виникає помилка часу виконання програми:

```
intx = 44;

System.Collections.ArrayList array = New
System.Collections.ArrayList();
// Упаковка значення x тип Object
array.Add(x);
string s = "hello";
// Додавання s до колекції ArrayList
array.Add(s);
// за індексом 1 у колекції array рядок s
inty = (int) array [1];
```

Слідом за числом x до колекції array додається рядок s. І спроба отримати число за індексом 1, яким зберігається рядок "hello", викличе помилку.

Ці проблеми були покликані усунути узагальнені типи. Узагальнені типи дозволяють вказати тип, який буде використовуватися. Наприклад, використовуємо узагальнений варіант класу ArrayList – клас List:

```
intx = 44;
string s = "hello";

List<int> array = new List<int>();

array.Add(x);
// Розпакування вже не потрібне
inty = array[0];

// тут буде помилка компіляції, оскільки можна //
додавати лише об'єкти int
array.Add(s);
```

Оскільки клас List є узагальненим, необхідно в «гострокутних» дужках вказувати тип даних - <тип>, котрій цей клас буде застосовуватися. Далі додається число та рядок до колекції. Число буде додано до колекції array, тому що колекція типізована типом int. Але для рядка array.Add(s) виникне помилка часу компіляції і програміст повинен видалити цей рядок. Таким чином, використовуючи узагальнений варіант класу, знижується час виконання програми і зменшується кількість потенційних помилок, що виявляються на етапі компіляції.

приклад. Створимо узагальнений клас Bank:

```
class Bank<T> {
T[] clients;

public Bank() {
}
public Bank(T[] _clients) {
this.clients = _clients;
}
}
```

Використання літери T в описі class Bank<T> дозволяє вказати, що даний «збагачений» тип буде використовуватися цим класом. У класі створюється масив об'єктів цього. Невідомо, що це буде тип (це може бути будь-який тип). Параметр T у кутових дужках часто називають універсальним параметром типу, оскільки замість нього можна підставити будь-який тип даних.

Наприклад, замість параметра T можна використовувати об'єкт int, тобто число, яке представляє номер рахунку. Це також може бути об'єкт string , або будь-який інший клас або структура:

```
Bank<int>bank = new Bank<int>(new int[] {1, 2, 4, 5,
6});
Bank<string > bank2 = new Bank<string >(new string []
{"13433", "342454", "21432"});
```

1.12.1. Значення за замовчуванням

Іноді виникає необхідність присвоїти змінним універсальних параметрів деяке початкове значення, у тому числі null. Але наприєму його привласнити не можна:

```
T id = null; //!!!! помилка
```

Для цього потрібно використовувати оператор default(T). Він надає посиланням типу як значення - значення null, а простим типам значень - значення 0:

```
class Bank<T> {
    T id = default(T);
}
```

1.12.2. Статичні поля узагальнених класів

При типізації узагальненого класу певним типом створюватиметься свій набір статичних членів. Наприклад, у класі Bank визначено таке статичне поле:

```
class Bank<T> {
public static T session;
public Bank() {
}
}
Тепер типизуємо клас двома типами int та string :
Bank<string > bank1 = new Bank<string >();
Bank<string >.session = "45245";

Bank<int>bank2 = new Bank<int>();
Bank<int>.session = 5452;

Console.WriteLine(Bank<string >.session); // "45245"
```

У результаті для Bank<**string** > і для Bank<**int**> буде створено свою змінну session.

1.12.3. Обмеження універсальних типів

Допустимо, необхідно, щоб клас Bank зберігав набір рахунків, представлених об'єктами деякого класу Account:

```
class Account {
public int Id {get; private set;} // номер рахунку
public Account(int _id) {
Id = _id;
}
}
```

Але цей клас може мати багато спадкоємців: DepositAccount (депозитний рахунок), DemandAccount (рахунок до запитання) тощо. Однак у загальному випадку невідомо, який вид рахунку в банку буде використовуватися.

Можливо, банк прийматиме лише депозитні рахунки. І в цьому випадку як універсальний параметр можна встановити тип Account:

```
class Bank<T> where T : Account {
T[] accounts;

public Bank(T[] accs) {
this.accounts = accs;
}
// виведення інформації про всі облікові записи
public void  AccountsInfo() {
foreach(Account acc in accounts) {
Console.WriteLine(acc.Id);
}
}
}
```

За допомогою виразу where T : Account вказується, що тип T обов'язково повинен бути класом Account або його спадкоємцем.

Тепер застосуємо клас Bank у програмі:

```
class  Program {
static void Main(string [] args) {
        Account[] accounts = new Account[] {
new Account(1857), new Account(2225), new Account(33232)
};
        Bank<Account>bank=new Bank<Account>(accounts);
bank.AccountsInfo();

Console.ReadLine();
}
}
```

Можна встановити безліч обмежень, записуючи їх типи через кому:

```
class Bank<T> where T : Client, Account {
}
```

Крім того, можна вказати обмеження - використовувати лише структури:

```
class Bank<T> where T : struct {
}
```

або класи:

```
class Bank<T> where T : class  {
}
```

А також можна задати як обмеження клас чи структуру, які реалізують конструктор за замовчуванням – використовуючи службове слово new:

```
class Bank<T> where T : new() {
}
```

1.12.4. Використання кількох універсальних параметрів

Можна задати відразу кілька універсальних параметрів та обмеження до кожного з них:

```
class Operation<T, U>
where U : class
where T : Account, new() {
}
```

1.12.5. Узагальнені методи

Крім узагальнених класів можна також створювати узагальнені методи, які також використовуватимуть універсальні параметри. Наприклад:

```
class  Program {
static void Main(string [] args) {
        Client client2 = new Client("Том", "Сімпсон");
Display<Client>(client2);
Console.ReadLine();
}

private static void  Display<T>(T людина) where T :
Person {
person.Display();
}
}
abstract class Person {
public string  FirstName {get; set; }
public string  LastName { get; set; }

Public Person(string  lName, string  fName) {
FirstName = fName;
LastName = lName;
}

public abstract void  Display();
}
class Client : Person {
Public Client(string  lName, string  fName)
: base(fName, lName) {
}

public override void  Display() {
Console.WriteLine(FirstName + " " + LastName);
}
}
```

1.12.6. Наслідування узагальнених типів

Один узагальнений клас може бути успадкований від іншого узагальненого класу:

```
class Transaction<T> {
T inAccount;
T outAccount;

public Transaction(T inAcc, T outAcc) {
inAccount = inAcc;
outAccount = outAcc;
}
}
class UniversalTransaction<T> : Transaction<T> {
Public UniversalTransaction(T inAcc, T outAcc) :
base(inAcc, outAcc) {

}
}
```

При цьому похідний клас необов'язково має бути узагальненим. Наприклад:

```
class String Transaction : Transaction<string > {
public String Transaction(string  inAcc, string  outAcc)
: base(inAcc, outAcc) {
}
}
```

Тепер у похідному класі як тип використовуватиметься тип string .

Також похідний клас може бути типизований параметром зовсім іншого типу, відмінного від типу базового класу:

```
class IntTransaction<T> : Transaction<int> {
    T operationCode = default(T);
Public IntTransaction(int inAcc, int outAcc, T operCode)
                                : base(inAcc, outAcc) {
this.operationCode = operCode;
}
}
```

1.13. Створення класів винятків та клас Exception

Вище у темі «обробка винятків» йшлося про обробку винятків. У межах об'єктно-орієнтованого підходу досить легко створювати свої класи винятків.

Базовим класом всім винятків є клас Exception. Він зберігає всю інформацію про виняток, що відбувся. Ця інформація доступна через властивості класу.

Деякі властивості класу Exception:

- `Helplink: зберігає адресу інтернет-ресурсу на якому можна знайти всю інформацію про помилку;`

- InnerException: об'єкт класу**Exception**зберігає інформацію про виключення, що спричинило поточний виняток;
- Message: зберігає повідомлення про виключення, текст помилки
- Source: зберігає ім'я об'єкта або складання, яке викликало виняток
- StackTrace: повертає строкове представлення стека викликів, які призвели до виключення
- TargetSite: повертає метод, у якому було викликано виняток

приклад. Допустимо, у програмі необхідно організувати обмеження за віком у процесі отримання нового об'єкта класу Person:

```
class  Program {
static void Main(string [] args) {
try {
                Person p = new Person();
p.Age = 17;
} catch (Exception ex) {
Console.WriteLine("Помилка:" + ex.Message);
Console.WriteLine("Метод:" + ex.TargetSite);
                      // Void  set_Age (Int32)

}
Console.ReadLine();
}
}

class  Person {
private string  name;
public string  Name {
get { return name; }
set {name = value; }
}
private int age;
public int Age {
get { return age; }
set {
if (value < 18) {
throw new Exception("Особам до 18" + "реєстрація
заборонена");
} else {
age = value;
}
}
```

```
}
}
```

У класі Person (а конструкторі) під час встановлення віку відбувається перевірка. Якщо вік менше 18, то викидається виняток - оператор throw. Клас Exception приймає в конструкторі як параметр рядок, який потім передається до його властивості Message.

Іноді зручніше використовувати свої класи винятків. Наприклад, у якійсь ситуації необхідно обробити певним чином лише ті винятки, які належать до класу Person. Для цих цілей створюємо спеціальний клас PersonException, наслідуючи клас Exception:

```
class PersonException : Exception {
Public PersonException(string  message)
                                  : base(message) {
    }
}
```

Цей клас, крім «порожнього» конструктора, нічого не має. У конструкторі відбувається звернення до конструктора базового класу Exception, передаючи до нього рядок message. Але тепер можна змінити клас Person так, щоб він «викидав» виняток саме цього типу і відповідно в основній програмі обробляти цей виняток:

```
class  Program {
static void Main(string [] args) {
try {
               Person p = new Person();
p.Age = 17;
} catch (PersonException ex) {
Console.WriteLine("Помилка:" + ex.Message);

}
Console.ReadLine();
}
}

class  Person {
private int age;
public int Age {
get { return age; }
set {
if (value < 18) {
throw new PersonException("Особам до 18" + "реєстрація
заборонена");
} else {
age = value;
}
```

```
}
}
}
```

1.14. Анонімні типи

Анонімні типи дозволяють створити об'єкт із деяким набором властивостей без визначення класу. Анонімний тип визначається за допомогою ключового слова var та ініціалізатора об'єктів:

```
varuser = new { Name = "Tom", Age = 34};
Console.WriteLine(user.Name);
```

В даному випадку user - це об'єкт анонімного типу, який має дві властивості Name і Age. Ці властивості можна використовувати як властивості звичайних об'єктів класів. Але з обмеженням - властивості анонімних типів доступні лише читання.

Компілятор сам буде створювати для анонімного типу ім'я типу і використовувати це ім'я при зверненні до об'єкта. Зазвичай анонімним типам компілятром надаються імена на кшталт "<>f__AnonymousType0'2".

Для середовища CLR анонімні типи будуть також як і класи, представляти посилальний тип.

Якщо у програмі використовуються кілька об'єктів анонімних типів з однаковим набором властивостей, то для них компілятор створить одне визначення анонімного типу:

```
varuser = new { Name = "Tom", Age = 34};
varstudent = new { Name = "Alice", Age = 21};
varmanager = new { Name = "Bob", Age = 26, Company =
"Microsoft"};

Console.WriteLine( user.GetType().Name); //
<>f__AnonymousType0'2
Console.WriteLine( student.GetType().Name); //
<>f__AnonymousType0'2
Console.WriteLine( manager.GetType().Name); //
<>f__AnonymousType1'3
```

Тут user і student будуть мати те саме визначення анонімного типу. Проте подібні об'єкти не можна перетворити на інший тип, наприклад до класу, навіть якщо він має подібний набір властивостей.

Навіщо потрібні анонімні типи? Іноді виникає завдання використовувати один тип в одному вузькому контексті або навіть один раз. Створення класу для такого типу може бути надлишковим. Якщо необхідно в процесі розробки програми додати нову властивість, його легко додати в анонімний клас. У випадку класу доведеться змінювати ще й клас, який може більше ніде не використовуватися. Типова ситуація - отримання результату вибірки з бази даних: об'єкти використовуються лише отримання вибірки і часто більше ніде

не використовуються (і класи їм створювати було зайве). А ось анонімний об'єкт чудово підходить для тимчасового зберігання вибірки.

1.15. Методи розширення

Методи розширення (extension methods) дозволяють додавати нові методи вже існуючі типи без створення нового похідного класу. Ця функціональність буває особливо корисною, коли необхідно додати до певного типу новий метод, але сам тип (клас чи структуру) змінити небажано (або неможливо).

Наприклад, необхідно додати для типу string новий метод:

```
class  Program {
static void Main(string [] args) {
string  s = "Привіт світ";
char c = 'i';
int i = s.WordCount(c);
Console.WriteLine(i);
Console.ReadLine();
}
}

publicstatic class  String Extension {
public static int WordCount(ці string  str, char c) {
int counter = 0;
for (int i = 0; i<str.Length; i++) {
if (str[i] == c) counter++;
}
return counter;
}
}
```

Щоб створити метод розширення, спочатку треба створити статичний клас, який і міститиме цей метод. У разі це клас String Extension. У цьому вся класі створюється статичний метод. У цьому вся прикладі суть методу розширення - підрахунок кількості певних символів у рядку.

Власне метод розширення - це звичайний статичний метод, який як перший параметр завжди приймає таку конструкцію:

```
this имя_типу назва_параметра
```

Тобто в нашому випадку this string str. Оскільки метод розширення буде належати до типу string , то тут і використовується цей тип.

Тепер для всіх рядків можна назвати цей метод: int i = s.WordCount(c);. Причому за виклику цього не треба вказувати перший параметр. Значення для інших параметрів передаються у звичайному порядку.

Застосування методів розширення дуже зручно, але при цьому треба пам'ятати, що метод розширення ніколи не буде викликаний, якщо він має ту саму сигнатуру, що і метод, визначений у типі.

Також слід враховувати, що методи розширення діють лише на рівні простору імен. Тобто, якщо додати проект інший простір імен, то метод не буде застосовуватися до рядків, і в цьому випадку для його використання необхідно підключити простір імен методу через директиву using.

1.16. Простір імен, псевдоніми та статичний імпорт

1.16.1. Простір імен

Усі зумовлені класи немає самі собою, а, зазвичай, полягають у спеціальні контейнери - простору імен. Клас Program, що створюється за умовчанням, вже знаходиться в просторі імен, який зазвичай збігається з назвою проекту:

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

Простір імен визначається за допомогою ключового слова namespace, після якого йде назва простору. Так, у даному випадку повна назва класу Program буде HelloApp.Program.

Клас Program "бачить" всі класи, які оголошені в тому ж просторі імен:

```
namespaceHelloApp {
class  Program {
static void Main(string [] args) {
Account account = new Account(4);
}
}
class  Account {
public int Id {get; private set;} // номер рахунку
public Account(int _id) {
Id = _id;
}
}
}
```

Але щоб задіяти класи з інших просторів імен, ці простори треба підключити за допомогою директиви using:

```
usingSystem;
namespaceHelloApp {
class  Program {
static void Main(string [] args) {
Console.WriteLine("hello");
}
}
}
```

Тут підключається простір імен System, у якому визначено клас Console. Якщо цього не робити, необхідно вказувати повний шлях до класу Console:

```
static void Main(string [] args)
{
System.Console.WriteLine("hello");
}
```

Простори імен можуть бути визначені всередині інших просторів:

```
usingHelloApp.AccountSpace;
namespaceHelloApp {
class  Program {
static void Main(string [] args) {
Account account = new Account(4);
}
}

namespace AccountSpace {
class  Account {
public int Id {get; private set;}
public Account(int _id) {
Id = _id;
}
}
}
}
```

У цьому випадку для підключення простору вказується повний шлях з урахуванням зовнішніх просторів імен: using HelloApp.AccountSpace;

1.16.2. Псевдоніми

Для різних класів можна задавати псевдоніми. Потім у програмі замість назви класу використовується його псевдонім.

Наприклад, для виведення рядка на екран застосовується метод Console.WriteLine(). Але якщо задати для класу Console псевдонім printer, замість System.Console.WriteLine можна кодувати printer.WriteLine:

```
Usingprinter = System.Console;
class  Program {
static void Main(string [] args) {
printer.WriteLine("Hello from C#");
printer.Read();
}
}
```

За допомогою виразу using printer=System.Console вказуємо, що псевдонімом для класу System.Console буде ім'я printer. Цей вираз не має нічого спільного з підключенням просторів імен на початку файлу, хоча використовує оператор using. У цьому фактично використовується повне ім'я класу з урахуванням простору імен, у якому клас визначено.

І ще приклад. Визначимо клас і для нього псевдонім:

```
usingPerson = HelloApp.User;
usingPrinter = System.Console;
namespaceHelloApp {
class  Program {
static void Main(string  [] args) {
            Person person = new Person();
person.name = "Tom";
Printer.WriteLine(person.name);
Printer.Read();
}
}

class  User {
public string  name;
}
}
```

Клас називається User, але у програмі йому використовується псевдонім Person.

Починаючи з версії C# 6.0 (Visual Studio 2015) до мови було додано можливість імпорту функціональності класів. Наприклад, знову ж таки візьмемо клас Console і застосуємо нову функціональність:

```
using staticSystem.Console;
namespaceHelloApp {
class  Program {
static void Main(string [] args) {
WriteLine("Hello from C# 6.0");
Read();
}
}
}
```

Вираз using static включає в програму всі статичні методи та властивості, а також константи. І після цього можна не вказувати назву класу під час виклику методу.

Подібним чином можна визначати свої класи та імпортувати їх:

```
using staticSystem.Console;
using staticSystem.Math;
using staticHelloApp.Geometry;
namespaceHelloApp {
class  Program {
static void Main(string [] args) {
double  radius = 50;
double  result = GetArea(radius); // Geometry.GetArea
WriteLine(result); //Console.WriteLine
Read(); // Console.Read
```

```
}
}
class  Geometry {
public static double  GetArea(double  radius) {
return PI*radius*radius; // Math.PI
}
}
}
```

1.17. Контрольні питання

1. Поняття класу та методу.
2. Концепція об'єкта класу.
3. Спеціальні методи класу.
4. Призначення цього ключового слова.
5. Призначення оператора new.
6. Як ініціалізуються початкові значення об'єктів?
7. Що таке часткові класи?
8. Основні модифікатори доступу.
9. Програмування констант та полів для читання.
10. Призначення та програмування блоків get і set.
11. Програмування та використання статичних членів класу.
12. Призначення та застосування ключового слова base.
13. Які методи називаються віртуальними?

2. Інтерфейси

2.1. Введення в інтерфейси

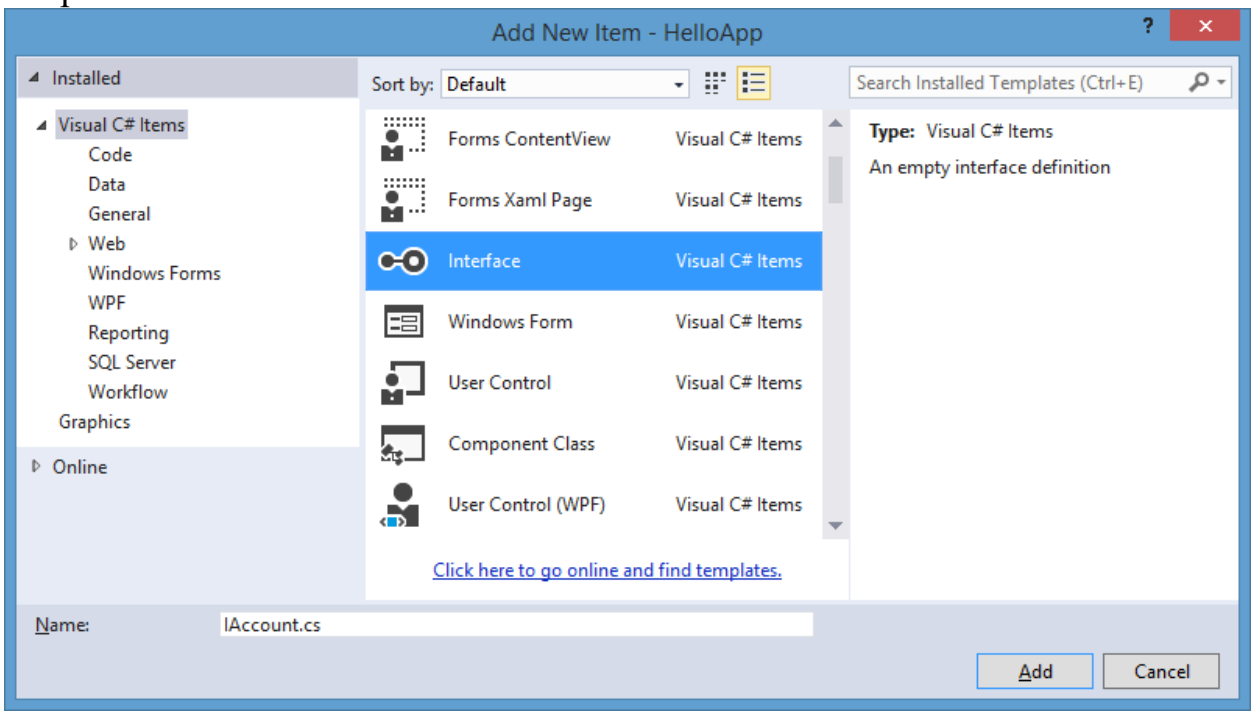
Використовуючи механізм успадкування можна доповнювати та перевизначати загальний функціонал базових класів у класах-спадкоємцях. У C# можна успадковуватись тільки від одного класу, на відміну, наприклад, від мови C++, де дозволено множинне успадкування.

У мові C# "проблема" множинного успадкування вирішується через інтерфейси. Інтерфейси дозволяють визначити деякий функціонал, який не має конкретної реалізації. Потім цей функціонал реалізують класи, які використовують дані інтерфейси.

Для визначення інтерфейсу використається ключове слово interface. Як правило, назви інтерфейсів у C# починаються з великої літери I, наприклад, IComparable, IEnumerable (так звана угорська нотація), проте це не обов'язкова вимога, а вимога стилю програмування. Інтерфейси так само, як і класи, можуть містити властивості, методи та події, тільки без конкретної реалізації.

Визначимо наступний інтерфейс IAccount, який міститиме методи та властивості для роботи з рахунком клієнта. Для додавання інтерфейсу в проект можна натиснути правою кнопкою миші на проект і в контекстному меню

вибрати Add-> New Item і в діалоговому вікні додавання нового компонента вибрати Interface:



Змінимо порожній код інтерфейсу IAccount на наступний:

```
interface IAccount {
// Поточна сума на рахунку
int CurrentSum {get; }
// Покласти гроші на рахунок
void  Put(int sum);
    // Взяти з рахунку
void  Withdraw(int sum);
    // Відсоток нарахувань
int Percentage {get; }
}
```

У інтерфейсу методи та властивості не мають реалізації, у цьому вони схожі на абстрактні методи абстрактних класів.

Сутність даного інтерфейсу проста: він визначає дві властивості для поточної суми грошей на рахунку та ставки відсотка за вкладами та два методи для додавання грошей на рахунок та вилучення грошей.

Ще один момент в оголошенні інтерфейсу: всі його члени - методи та властивості не мають модифікаторів доступу, але фактично за умовчанням доступ public, оскільки мета інтерфейсу - визначення функціоналу для реалізації його класом. Тому весь функціонал має бути відкритим для реалізації.

Застосування інтерфейсу аналогічне до успадкування класу:

```
class Client : IAccount {
// Реалізація методів та властивостей інтерфейсу
}
```

Тепер же реалізуємо інтерфейс у класі Client, оскільки клієнт має рахунок:

```
class Client : IAccount {
```

```
int _sum; // Змінна для зберігання суми
int _percentage; // Змінна для зберігання відсотка

public string  Name { get; set; }
Public Client(string  name, int sum, int percentage) {
Name = name;
_sum = sum;
_percentage = percentage;
}

public int CurrentSum {
get {return_sum; }
}

public void  Put(int sum) {
_sum + = sum;
}

public void  Withdraw(int sum) {
if (sum <= _sum) {
_sum -= sum;
}
}

public int Percentage {
get {return_percentage; }
}
public void  Display() {
Console.WriteLine("Клієнт" + Name + "має рахунок на суму
"+_sum);
}
}
```

Як і у випадку абстрактних методів абстрактного класу, клас Client реалізує всі методи інтерфейсу. Оскільки всі методи та властивості інтерфейсу є публічними, то за їх реалізації можна застосовувати лише модифікатор public. Тому, якщо клас повинен мати метод із якимось іншим модифікатором, наприклад, protected, то використання інтерфейсу неможливе.

Застосування класу у програмі:

```
Client client = new Client("Tom", 200, 10);
client.Put(30);
Console.WriteLine(client.CurrentSum); //230
client.Withdraw(100);
Console.WriteLine(client.CurrentSum); //130
```

Інтерфейси, як і класи, можуть успадковуватися:

```
interfaceIDepositAccount : IAccount {
```

```
void  GetIncome(); // нарахування відсотків
}
```

При застосуванні цього інтерфейсу клас Client повинен буде реалізувати як методи та властивості інтерфейсу IDepositAccount, так і методи та властивості базового інтерфейсу IAccount.

2.1.1. Інтерфейси у перетвореннях типів

Все сказане вище щодо перетворення типів класів можна застосувати і для інтерфейсів. Оскільки клас Client реалізує інтерфейс IAccount, змінна типу IAccount може зберігати посилання об'єкт типу Client:

```
IAccount client1 = new Client("Том", 200, 10);
client1.Put(200);
Console.WriteLine(client1.CurrentSum); // 400
// Інтерфейс не має методу Display, // необхідне явне
приведення
((Client)client1).Display();
```

Якщо необхідно звернутися до методів класу Client, які не визначені в інтерфейсі IAccount, а визначені в самому класі Client або в його базовому класі, треба явно виконати перетворення типів: ((Client)client1).Display();

2.1.2. Узагальнені інтерфейси

Як і класи, інтерфейси можуть бути узагальненими:

```
interfaceIAccount<T> {
void  SetSum (T_sum);
void  Display();
}

class Client<T> : IAccount<T> {
T sum = default (T);
public void  SetSum(T _sum) {
this.sum = _sum;
}
public void  Display() {
Console.WriteLine(sum);
}
}
```

Інтерфейс IAccount типизований параметром T, який при реалізації інтерфейсу використовується у класі Client. Зокрема змінну суму визначено як T, що дозволяє нам використовувати для суми різні числові типи.

Визначимо дві реалізації: одна як параметр буде використовувати тип int, а інша - тип double :

```
IAccount<int> intClient = new Client<int>();
intClient.SetSum(300);
intClient.Display();
```

```
IAccount<double > double Client = новий Client<double
>();
double Client.SetSum(500.09);
double Client.Display();
```

2.1.3. Явне застосування інтерфейсів

Може скластися ситуація, коли клас застосовує кілька інтерфейсів, але вони мають один і той же метод з одним і тим же повертається результатом і одним і тим же набором параметрів. Наприклад:

```
class Person : ISchool, Iuniversity {
public void  Study() {
Console.WriteLine("Навчання в школі або в університеті");
}
}

interfaceIschool {
void  Study();
}

interfaceIuniversity {
void  Study();
}
```

Клас Person визначає один метод Study(), створюючи одну загальну реалізацію обох застосованих інтерфейсів. І незалежно від того, чи розглядатимемося об'єкт Person як об'єкт типу ISchool чи IUniversity, результат методу буде той самий.

Однак нерідко буває необхідно розмежувати реалізовані інтерфейси. У цьому випадку треба явно застосувати інтерфейс:

```
class Person : ISchool, Iuniversity {
void  ISchool.Study() {
Console.WriteLine("Навчання в школі");
}
void  IUniversity.Study() {
Console.WriteLine("Навчання в університеті");
}
}
```

При явній реалізації вказується назва методу разом із назвою інтерфейсу, при цьому не можна використовувати модифікатор public, тобто ці методи закриті. У цьому випадку при використанні методу Study у програмі нам треба об'єкт Person привести до типу відповідного інтерфейсу:

```
static void Main(string [] args) {
    Person p = new Person();

((ISchool) p). Study ();
((IUniversity) p). Study ();
```

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

2.2. Клонування об'єктів. Інтерфейс ICloneable

Оскільки класи в C# представляють типи посилань, то це накладає деякі обмеження на їх використання. Зокрема:

```
class  Program {
static void Main(string [] args) {
        Person p1 = new Person { Name="Tom", Age = 23};
Person p2 = p1;
p2.Name = "Alice";
        Console.WriteLine(p1.Name); //Alice

Console.Read();
}
}

class Person
{
public string  Name { get; set; }
public int Age {get; set; }
}
```

В даному випадку об'єкти p1 і p2 будуть вказувати на той самий об'єкт у пам'яті, тому зміни властивостей у змінній p2 торкнуться також і змінну p1.

Щоб змінна p2 вказувала на новий об'єкт, але зі значеннями p1, можна застосувати клонування за допомогою реалізації інтерфейсу ICloneable:

```
publicinterface Icloneable {
object Clone();
}
```

Реалізація інтерфейсу в класі Person могла б виглядати так:

```
class Person : Icloneable {
public string  Name { get; set; }
public int Age {get; set; }
public object Clone() {
return new Person { Name = this.Name, Age = this.Age };
}
}
```

Використання:

```
class  Program {
static void Main(string [] args) {
        Person p1 = new Person { Name="Tom", Age = 23};
Person p2 = (Person) p1.Clone ();
p2.Name = "Alice";
```

```
Console.WriteLine(p1.Name); // Tom

Console.Read();
}
}
```

Тепер все нормально - зміни у властивостях p2 не позначаються на властивостях p1.

Для скорочення коду копіювання можна використовувати спеціальний метод MemberwiseClone(), який повертає копію об'єкта:

```
class Person : Icloneable {
public string  Name { get; set; }
public int Age {get; set; }
public object Clone() {
return this.MemberwiseClone();
}
}
```

Цей метод реалізує поверхневе («неглибоке» – побите) копіювання. Таке копіювання може призвести до несподіваних результатів. Наприклад, нехай клас Person містить посилання на об'єкт Company:

```
class Person : Icloneable {
public string  Name { get; set; }
public int Age {get; set; }
public Company Work { get; set; }

public object Clone() {
return this.MemberwiseClone();
}
}

class Company {
public string  Name { get; set; }
}
```

У цьому випадку при копіюванні нова копія вказуватиме на той самий об'єкт Company:

```
Person p1 = New Person {Name = "Tom", Age = 23, Work =
New Company {Name = "Microsoft"}};
Person p2 = (Person) p1.Clone ();
p2.Work.Name = "Google";
p2.Name = "Alice";
Console.WriteLine(p1.Name); // Tom
Console.WriteLine(p1.Work.Name); // Google - а має //бути
Microsoft
```

Поверхневе копіювання працює тільки для властивостей, що представляють примітивні типи, але не для складних об'єктів (не для посилань). У разі «посилальних» об'єктів треба застосовувати**глибоке копіювання**:

```
class Person : Icloneable {
public string  Name { get; set; }
public int Age {get; set; }
public Company Work { get; set; }

public object Clone() {
Company company = new Company {Name = this.Work.Name};
return new Person {
            Name = this.Name,
            Age = this.Age,
Work = company
};
}
}

class Company {
public string  Name { get; set; }
}
```

2.3. Сортування об'єктів. Інтерфейс IComparable

Більшість вбудованих у .NET класів колекцій та масиви підтримують сортування. За допомогою одного методу, який, як правило, називається Sort(), можна відразу відсортувати за зростанням весь набір даних. Наприклад:

```
int[]numbers = new int[] {97, 45, 32, 65, 83, 23, 15};
Array.Sort(numbers);
foreach(int n in numbers)
Console.WriteLine(n);
```

Однак метод Sort за промовчанням працює тільки для наборів примітивних типів, як int або string . Для сортування наборів складних об'єктів використовується інтерфейс IComparable. Він має лише один метод:

```
publicinterface IComparable {
int CompareTo(object o);
}
```

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

- Менше за нуль. Отже, поточний об'єкт повинен перебувати перед об'єктом, який передається як параметр
- дорівнює нулю. Отже, обидва об'єкти рівні
- Більше нуля. Отже, поточний об'єкт повинен перебувати після об'єкта, що передається як параметр

Наприклад, є клас Person:

```
class Person : IComparable {
public string  Name { get; set; }
public int Age {get; set; }
public int CompareTo(object o) {
        Person p = o as Person;
if (p! = null)
return this.Name.CompareTo(p.Name);
else
throw new Exception("Неможливо порівняти два об'єкти");
}
}
```

Тут як критерій порівняння вибрано властивість Name об'єкта Person. Тому при порівнянні тут фактично іде порівняння значення властивості Name поточного об'єкта та властивості Name об'єкта, переданого через параметр. Якщо раптом об'єкт не вдасться привести до типу Person, то викидається виняток.

Застосування:

```
Person p1 = new Person {Name = "Bill", Age = 34};
Person p2 = new Person {Name = "Tom", Age = 23};
Person p3 = new Person { Name = "Alice", Age = 21};

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people);

foreach(Person p in people) {
Console.WriteLine("{0} - {1}", p.Name, p.Age);
}
```

Інтерфейс IComparable має узагальнену версію, тому можна скоротити і спростити його застосування в класі Person:

```
class Person : IComparable<Person> {
public string  Name { get; set; }
public int Age {get; set; }
public int CompareTo(Person p) {
return this.Name.CompareTo(p.Name);
}
}
```

2.3.1. Застосування компаратора

Крім інтерфейсу IComparable, платформа .NET надає інтерфейс IComparer:

```
interfaceIComparer {
int Compare(object o1, object o2);
}
```

Метод Compare призначений для порівняння двох об'єктів o1 та o2. Він також повертає три значення, залежно від результату порівняння:

- якщо перший об'єкт більший за другий, то повертається число більше 0,
- якщо перший об'єкт менший за другий, то число менше нуля;
- якщо обидва об'єкти дорівнюють, повертається нуль.

Створимо компаратор об'єктів Person. Нехай він порівнює об'єкти залежно від довжини рядка – значення властивості Name:

```
class PeopleComparer : IComparer<Person> {
public int Compare(Person p1, Person p2) {
if (p1.Name.Length > p2.Name.Length)
return 1;
else if (p1.Name.Length < p2.Name.Length)
return -1;
else
return 0;
}
}
```

У цьому випадку використовується узагальнена версія інтерфейсу IComparer, щоб не робити зайвих перетворень типів. Застосування компаратора:

```
Person p1 = new Person {Name = "Bill", Age = 34};
Person p2 = new Person {Name = "Tom", Age = 23};
Person p3 = new Person { Name = "Alice", Age = 21};

Person[] people = new Person[] { p1, p2, p3 };
Array.Sort(people, new PeopleComparer());

foreach(Person p in people) {
Console.WriteLine("{0} - {1}", p.Name, p.Age);
}
```

Об'єкт компаратора вказується як другий параметр методу Array.Sort(). При цьому не важливо, чи реалізує клас Person інтерфейс IComparable чи ні. Правила сортування, встановлені компаратором, матимуть більший пріоритет. На початку будуть йти об'єкти Person, у яких імена менші, а в кінці - у яких імена довші:

```
Tom - 23
Bill - 34
Alice - 21
```

2.4. Коваріантність та контраваріантність узагальнених інтерфейсів

Поняття коваріантності та контраваріантності пов'язані з можливістю використовувати в додатку замість деякого типу інший тип, який знаходиться нижче або вище в ієрархії спадкування.

Є три можливі варіанти поведінки:

- **Коваріантність**: дозволяє використовувати конкретніший тип, ніж заданий спочатку;
- **Контраваріантність**: дозволяє використовувати більш універсальний тип, ніж заданий спочатку;
- **Інваріантність**: дозволяє використовувати тільки заданий тип

Починаючи з .NET 4.0 в C# було додано можливість створення коваріантних та контраваріантних узагальнених інтерфейсів. Ця функціональність підвищує гнучкість під час використання узагальнених інтерфейсів у програмі. За умовчанням усі узагальнені інтерфейси, наприклад, IAccout<T> є інваріантними.

Узагальнені інтерфейси можуть бути підступними, якщо до універсального параметра застосовується ключове слово out. Наприклад:

```
class Account {
static Random rnd = new Random();

public void  DoTransfer() {
int sum = rnd.Next (10, 120);
Console.WriteLine("Клієнт поклав на рахунок {0}"+
"доларів", sum);
}
}
class DepositAccount : Account {
}

interfaceIBank<out T> where T : Account {
T DoOperation();
}

class Bank: IBank<DepositAccount> {
public DepositAccount DoOperation() {
        DepositAccount acc = new DepositAccount();
acc.DoTransfer();
return acc;
}
}
```

Визначення інтерфейсу IBank<out T> вказує, що цей інтерфейс буде підступним. У програмі його можна використовувати так:

```
static void Main(string [] args) {
    IBank<DepositAccount> depositBank = new Bank();
depositBank.DoOperation();

IBank<Account> ordinaryBank = depositBank;
ordinaryBank.DoOperation();
```

```
Console.ReadLine();
}
```

Тобто можна присвоїти більш загальному типу IBank<Account> об'єкт більш конкретного типу IBank<DepositAccount>.

У той же час, якби не використовувалося ключове слово out:

```
interfaceIBank<out T> where T : Account
```

то компілятор Visual Studio відзначив би рядок:

```
IBank<Account> ordinaryBank = depositBank;
```

як хибну. Оскільки в цьому випадку неможливо було б привести об'єкт IBank<DepositAccount> до типу IBank<Account>.

При створенні коваріантного інтерфейсу треба враховувати, що універсальний параметр може використовуватися тільки як тип значення, що повертається методами інтерфейсу. Але не може використовуватися як тип аргументів методу або обмеження методів інтерфейсу.

Для створення контраваріантного інтерфейсу необхідно використовувати ключове слово in. Для його застосування змінимо інтерфейс IBank та клас Bank:

```
interfaceIBank<in T> where T : Account {
void  DoOperation(T account);
}

class Bank<T>: IBank<T> where T : Account {
public void  DoOperation(T account) {
account.DoTransfer();
}
}
```

Класи Account та DepositAccount залишаються без змін. Тепер застосуємо інтерфейс IBank та класи у програмі:

```
static void Main(string [] args) {
    Account account = new Account();
    IBank<Account> ordinaryBank = new Bank<Account>();
ordinaryBank.DoOperation(account);

    DepositAccount depositAcc = new DepositAccount();
IBank<DepositAccount> depositBank = ordinaryBank;
depositBank.DoOperation(depositAcc);

Console.ReadLine();
}
```

Оскільки інтерфейс IBank використовує універсальний параметр із ключовим словом in, то він є контраваріантним, тому в коді можливо об'єкт Bank<Account> привести до типу IBank<DepositAccount>:

```
IBank<DepositAccount> depositBank = ordinaryBank;
```

Тобто об'єкт інтерфейсу з універсальним типом наводиться до об'єкта інтерфейсу з більш конкретним типом.

2.5. Контрольні питання

1. Яке ключове слово використовується для визначення інтерфейсу?
2. Чи можуть інтерфейси містити властивості та методи?
3. Як можна додати інтерфейс до проекту?
4. Що таке узагальнені інтерфейси?
5. Що таке «поверхневе» та «глибоке» копіювання об'єктів?
6. Як використовувати інтерфейс IComparable?
7. Що таке інваріантність, підступність і контраваріантність узагальнених інтерфейсів?

3. Делегати, події та лямбди

3.1. Делегати

Крім властивостей та методів класи та інтерфейси можуть містити делегати та події. Делегати представляють такі об'єкти, які вказують інші методи. Тобто делегати – це покажчики на методи. За допомогою делегатів можна викликати певні методи у відповідь на деякі дії, що відбулися. Тобто, по суті, делегати надають програмісту функціонал – «функції зворотного виклику».

Методи, на які посилаються делегати, повинні мати ті ж параметри і той самий тип значення, що повертається. Створимо два делегати:

```
delegate intOperation(int x, int y);
delegate void GetMessage();
```

Для оголошення делегата використовується ключове слово delegate, після якого йде тип, назва і параметри, що повертається. Перший делегат посилається на функцію, яка як параметри приймає два значення типу int і повертає деяке int число. Другий делегат посилається на метод без параметрів, який нічого не повертає.

Щоб "працювати" з делегатом, треба "створити" його об'єкт за допомогою конструктора. У конструктор передається «адреса методу», що викликається делегатом.

Щоб викликати метод, який вказує делегат, треба використовувати його метод Invoke. Крім того, делегати можуть виконуватися в асинхронному режимі, при цьому не треба створювати другий потік, лише замість методу Invoke використовувати пару методів BeginInvoke/EndInvoke:

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

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

```
        GetMessagedel; // 2. Створюємо змінну делегата
if (DateTime.Now.Hour < 12) {
del= GoodMorning; // 3. Привласнюємо цю // змінну //
адресу методу
} else {
del= GoodEvening;
}
del.Invoke(); // 4. Викликаємо метод
Console.ReadLine();
}
private static void  GoodMorning() {
Console.WriteLine("Good Morning");
}
private static void  GoodEvening() {
Console.WriteLine("Good Evening");
}
}
```

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

Подивимося з прикладу іншого делегата:

```
class  Program {
delegate int Operation(int x, int y);

static void Main(string [] args) {
// Надання адреси методу в контрукторі
Operationdel= New Operation (Add); // делегат //
свідчить про метод Add
int result = del.Invoke(4,5);
Console.WriteLine(result);

del = Multiply; // Тепер делегат вказує // на метод
Multiply
result = del.Invoke(4, 5);
Console.WriteLine(result);

Console.Read();
}
private static int Add(int x, int y) {
return x+y;
}
private static int Multiply (int x, int y) {
return x * y;
}
```

```
}
```

Тут показаний спосіб привласнення делегату "адреси методу" при його конструюванні (через конструктор). І оскільки пов'язаний метод, як і делегат, має два параметри, то при виклику делегата метод Invoke йому передаються два параметри. З іншого боку, оскільки метод повертає значення типу int, можна присвоїти результат роботи методу Invoke який-небудь цілої змінної.

Метод Invoke() при виклику делегата можна опустити та використати скорочену форму:

```
del = Multiply; // Тепер делегат вказує на метод //
Multiply
result = del(4, 5);
```

Тобто, делегат можна викликати як звичайний метод, передаючи йому аргументи.

Також делегати можуть бути параметрами методів:

```
class  Program {
delegate void  GetMessage();

static void Main(string [] args) {
if (DateTime.Now.Hour < 12) {
Show_Message(GoodMorning);
} else {
Show_Message(GoodEvening);
}
Console.ReadLine();
}
private static void  Show_Message(GetMessage _del) {
_del.Invoke();
}
private static void  GoodMorning() {
Console.WriteLine("Good Morning");
}
private static void  GoodEvening() {
Console.WriteLine("Good Evening");
}
}
```

Дані приклади не показують «справжньої сили делегатів», оскільки необхідні методи у разі можна викликати і без будь-яких делегатів. Однак найбільш сильна сторона делегатів полягає в тому, що вони (використовуючи «функціонал методів зворотного виклику») можуть повідомляти інші об'єкти про події, що відбулися.

Розглянемо приклад. Нехай має клас, що описує рахунок у банку:

```
class Account {
int _sum; // Змінна для зберігання суми
int _percentage; // Змінна для зберігання відсотка
```

```
public Account(int sum, int percentage) {
_sum = sum;
_percentage = percentage;
}

public int CurrentSum {
get {return_sum; }
}

public void  Put(int sum) {
_sum + = sum;
}

public void  Withdraw(int sum) {
if (sum <= _sum) {
_sum -= sum;
}
}

public int Percentage {
get {return_percentage; }
}
}
```

Припустимо, у разі виведення грошей за допомогою методу Withdraw необхідно якось повідомляти про це самого клієнта і, можливо, інші об'єкти. Для цього створимо делегат AccountStateHandler. Щоб використати делегат, треба створити змінну цього делегата, а потім привласнити йому метод, який викликатиметься делегатом.

Отже, додамо до класу Account наступні рядки:

```
class Account {
    // Оголошуємо делегат
public delegate void  AccountStateHandler(string
message);
// Створюємо змінну делегата
AccountStateHandler del;

// Реєструємо делегат
public void  RegisterHandler(AccountStateHandler _del)
{
del = _del;
}

// Далі решта рядків класу Account
```

Тут фактично робляться ті ж кроки, що були показані в прикладі вище, крім виклику делегата. У разі делегат приймає параметр типу string . Тепер змінимо метод Withdraw в такий спосіб:

```
publicvoid  Withdraw(int sum) {
if (sum <= _sum) {
_sum -= sum;

if (del! = null)
del("Сума" + sum.ToString () + "знята з рахунку");
} else {
if (del! = null)
del("Недостатньо грошей на рахунку");
}
}
```

Тепер при знятті грошей через метод Withdraw спочатку перевіряється, чи делегат має посилання на будь-який метод (інакше він має значення null). І якщо метод «встановлений», він викликається, передаючи відповідне повідомлення як його параметра.

Тепер протестуємо клас в основній програмі:

```
class  Program {
static void Main(string [] args) {
        // створюємо банківський рахунок
        Account account = new Account(200, 6);
// Додаємо в делегат посилання метод Show_Message
// а сам делегат передається як параметр методу //
RegisterHandler
account.RegisterHandler(new
Account.AccountStateHandler(Show_Message));
// Двічі поспіль намагаємось зняти гроші
account.Withdraw(100);
account.Withdraw(150);
Console.ReadLine();
}
private static void  Show_Message(String  message) {
Console.WriteLine(message);
}
}
```

При виконанні програми отримаємо два різні повідомлення:

```
Суму 100 знято з рахунку
Недостатньо грошей на рахунку
```

Таким чином створюється механізм зворотного виклику для класу Account, який спрацьовує у разі зняття грошей.

Оскільки делегат оголошений всередині класу Account, то отримати доступ до нього можна так: Account.AccountStateHandler.

Знову ж таки може виникнути питання: чому б до коду методу Withdraw() не виводити повідомлення про зняття грошей? Навіщо треба задіяти якийсь делегат?

Справа в тому, що не завжди програміст має доступ до коду класів. Наприклад, частина класів може створюватися і компілюватися однією людиною, яка не знатиме, як ці класи використовуватимуться. А використовуватиме ці класи інший розробник.

У прикладі виводиться повідомлення на консоль. Однак для класу Account не важливо, як це повідомлення виводиться. Класу Account навіть невідомо, що взагалі робитиметься внаслідок списання грошей. Він просто надсилає повідомлення про це через делегат.

В результаті, якщо створюється консольна програма, то можливо через делегат виводити повідомлення на консоль. Якщо ж створюється графічна програма Windows Forms або WPF, можна виводити повідомлення у вигляді графічного вікна. Також можна не просто виводити повідомлення, а наприклад, записати при списанні інформацію про цю дію у файл або надіслати повідомлення на електронну пошту. У випадку - будь-якими способами обробити виклик делегата. Ці методи обробки не залежатимуть від класу Account.

У прикладі делегат приймав «адресу» однією метод, насправді може вказувати відразу кілька методів. Крім того, за потреби можна видалити посилання на адреси певних методів, щоб вони не викликалися під час виклику делегата.

Змінимо в класі Account метод RegisterHandler і додамо новий метод UnregisterHandler, який видалятиме методи зі списку методів делегата:

```
// Реєструємо делегат
publicvoid  RegisterHandler(AccountStateHandler _del) {
                  Delegate mainDel =
                  System.Delegate.Combine(_del,del);
del= mainDel як AccountStateHandler;
}
// Скасування реєстрації делегата
publicvoid  UnregisterHandler( AccountStateHandler _del)
{
Delegate mainDel = System.Delegate.Remove(del, _del);
del = mainDel як AccountStateHandler;
}
```

У першому методі метод Combine об'єднує делегати _del і del один, який потім присвоюється змінної del. У другому методі метод Remove повертає делегат, зі списку викликів якого вилучено делегат _del.

Основна програма:

```
class  Program {
static void Main(string [] args) {
        Account account = new Account(200, 6);
        Account.AccountStateHandler colorDelegate = новий
```

```
Account.AccountStateHandler(Color_Message);

        // Додаємо в делегат посилання на методи
account.RegisterHandler(new
Account.AccountStateHandler(Show_Message));
account.RegisterHandler(colorDelegate);
        // Двічі поспіль намагаємось зняти гроші
account.Withdraw(100);
account.Withdraw(150);

        // Видаляємо делегат
account.UnregisterHandler(colorDelegate);
account.Withdraw(50);

Console.ReadLine();
}
private static void  Show_Message(String  message) {
Console.WriteLine(message);
}
private static void  Color_Message(string  message) {
        // Встановлюємо червоний колір символів
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine(message);
        // Скидаємо налаштування кольору
Console.ResetColor();
}
}
```

Тут створено ще один метод - Color_Message, який виводить те саме повідомлення тільки червоним кольором. Для першого делегата створюється окрема змінна. Але великої різниці між передачею обох у метод account.RegisterHandler немає. У першому випадку відразу передається об'єкт, створюваний конструктором

account.RegisterHandler(new Account.AccountStateHandler(Show_Message));

У другому випадку створюється змінна, і її передаємо метод account.RegisterHandler(colorDelegate);.

Виклик account.UnregisterHandler(colorDelegate) видаляє зі списку викликів делегат colorDelegate, тому цей метод більше не спрацьовуватиме. Консольний висновок матиме таку форму (у дужках вказано – колір повідомлення):

```
Суму 150 знято з рахунку      (червоний)
Суму 150 знято з рахунку          (чорний)
Недостатньо грошей на рахунку     (червоний)
Недостатньо грошей на рахунку     (чорний)
Суму 50 знято з рахунку           (чорний)
```

Також можна використовувати скорочену форму додавання та видалення делегатів:

```
// Реєструємо делегат
publicvoid  RegisterHandler(AccountStateHandler _del) {
del + = _del; // додаємо делегат
}
// Скасування реєстрації делегата
publicvoid  UnregisterHandler(AccountStateHandler _del){
del - = _del; // видаляємо делегат
}
```

3.2. Події

Вище було розглянуто, як із допомогою делегатів можна створювати «механізм зворотних викликів у програмі». Однак C# для тієї ж мети надає зручніші і простіші конструкції під назвою події, які сигналізують про те, що відбулося певну дію.

Події оголошуються у класі за допомогою ключового слова event, після якого йде назва делегата:

```
// Оголошуємо делегат
publicdelegate void  AccountStateHandler(string
message);
// Подія, що виникає під час виведення грошей
publicevent AccountStateHandler Withdrowed;
```

Зв'язок із делегатом означає, що метод, який обробляє цю подію, повинен приймати самі параметри, як і делегат, і повертати той самий тип, як і делегат.

Розглянемо приклад. Візьмемо клас Account з минулої теми та змінимо його наступним чином:

```
class Account {
    // Оголошуємо делегат
    public delegate void  AccountStateHandler(string
message);
    // Подія, що виникає під час виведення грошей
    public event AccountStateHandler Withdrowed;
    // Подія, що виникає при додаванні на рахунок
    public event AccountStateHandler Added;

    int _sum; // Змінна для зберігання суми
    int _percentage; // Змінна для зберігання відсотка

    public Account(int sum, int percentage) {
        _sum = sum;
        _percentage = percentage;
    }
```

```
    public int CurrentSum {
        get {return_sum; }
    }

    public void  Put(int sum) {
        _sum + = sum;
        if (Added! = null) Added ("На рахунок надійшло" +
sum);
    }
    public void  Withdraw(int sum) {
        if (sum <= _sum) {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed("Сума" + sum + "знята з
рахунку");
        } else {
            if (Withdrowed != null)
                Withdrowed("Недостатньо грошей на "+"
рахунку");
        }
    }

    public int Percentage {
        get {return_percentage; }
    }
}
```

Тут визначено дві події: Withdrowed та Added. Обидві події оголошені як екземпляри делегата AccountStateHandler, тому для обробки цих подій буде потрібний метод, який приймає рядок як параметр.

Потім у методах Put та Withdraw викликаються ці події. Перед викликом перевіряється, чи закріплені за цими подіями обробники:

(if (Withdrowed! = null)).

Так як ці події представляють делегат AccountStateHandler, що приймає як параметр рядок, то і при виклик подій передається рядок - пояснює, що сталося.

Тепер використовуємо події в основній програмі:

```
class  Program {
    static void Main(string [] args) {
        Account account = new Account(200, 6);
        // Додаємо обробники події
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;

        account.Withdraw(100);
        // Видаляємо обробник події
        account.Withdrowed -= Show_Message;
```

```
        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void  Show_Message(string  message) {
        Console.WriteLine(message);
    }
}
```

Для прикріплення оброблювача події до певної події використовується операція += («плюс одно») і для відкріплення - операція -= («мінус одно»):

```
подія += метод_обробника_події
подія -= метод_обробника_події
```

Знову ж таки звернемо увагу, що метод оброблювача повинен мати такі ж параметри, як і делегат події, і повертати той же тип.

Консольний висновок прикладу:

```
Суму 100 знято з рахунку
На рахунок надійшло 150
```

Крім використаного вище способу прикріплення обробників є й інший з використанням делегату. Обидва способи рівноцінні:

```
account.Added += Show_Message;
// Клас даних події AccountEventArgs
account.Added += new
Account.AccountStateHandler(Show_Message);
```

У графічних програмах (створених за допомогою Windows Forms або WPF) застосовуються обробники подій. Вони як параметр приймають аргумент типу EventArgs. Наприклад, обробник натискання кнопки:

```
private void button1_Click(object sender,
System.EventArgs e){}
```

Параметр e, будучи об'єктом класу EventArgs, містить усі дані події. Додамо і в програму такий клас. Назвемо його AccountEventArgs і додамо до нього наступний код:

```
class AccountEventArgs {
    // Повідомлення
    public string  message;
    // Сума, яку змінився рахунок
    public int sum;

    public AccountEventArgs(string  _mes, int _sum) {
        message=_mes;
        sum = _sum;
    }
}
```

Цей клас має два поля: message - для зберігання повідомлення, що виводиться, і sum - для зберігання суми, на яку змінився рахунок.

Тепер застосуємо клас AccoutEventArgs, змінивши клас Account таким чином:

```
class Account {
    // Оголошуємо делегат
    public delegate void  AccountStateHandler(object
sender, AccountEventArgs e);
    // Подія, що виникає під час виведення грошей
    public event AccountStateHandler Withdrowed;
    // Подія, що виникає при додаванні на рахунок
    public event AccountStateHandler Added;

    int _sum; // Змінна для зберігання суми
    int _percentage; // Змінна для зберігання відсотка

    public Account(int sum, int percentage) {
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum {
        get {return_sum; }
    }

    public void  Put(int sum)
    {
        _sum + = sum;
        if (Added! = null)
            Added(this, new AccountEventArgs("На рахунок
"+" надійшло sum", sum));
    }
    public void  Withdraw(int sum) {
        if (sum <= _sum) {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed(this, new
AccountEventArgs("Сума" + sum + "знята з рахунку", sum));
        } else {
            if (Withdrowed != null) Withdrowed(this, new
AccountEventArgs( "Недостатньо грошей на рахунку", sum));
        }
    }

    public int Percentage {
```

```
        get {return_percentage; }
    }
}
```

Порівняно з попередньою версією класу Account, тут змінилася лише кількість параметрів у делегата і відповідно кількість параметрів під час виклику події. Тепер вони також приймають об'єкт AccountEventArgs, який зберігає інформацію про подію, яку отримує через конструктор.

Тепер змінимо основну програму:

```
class  Program {
    static void Main(string [] args) {
        Account account = new Account(200, 6);
        // Додаємо обробники події
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;

        account.Withdraw(100);
        // Видаляємо обробник події
        account.Withdrowed -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void  Show_Message( object sender,
AccountEventArgs e){
     Console.WriteLine("Сума транзакції: {0}", e.sum);
     Console.WriteLine(e.message);
    }
}
```

Порівняно з попереднім варіантом тут змінено кількість параметрів та сутність їх використання в обробникуShow_Message.

3.3. Анонімні методи

Із делегатами тісно пов'язане поняття анонімних методів.**Анонімні методи**представляють скорочений запис методів «без імені». Наприклад, у минулій темі створювали подію та обробник до неї:

```
private static void Show_Message(object sender,
AccountEventArgs e) {
Console.WriteLine("Сума транзакції: {0}", e.sum);
Console.WriteLine(e.message);
}
```

Іноді такі методи потрібні для обробки однієї події і більше цінності не становлять і ніде не використовуються. Анонімні методи дозволяють вбудувати код там, де він викликається, наприклад:

```
Account account = new Account(200, 6);
// Додаємо обробники події
account.Added += delegate(object sender, AccountEventArgs
e) {
Console.WriteLine("Сума транзакції: {0}", e.sum);
Console.WriteLine(e.message);
}; //!!!! «крапка з комою» обов'язкова
```

Фактично цей той самий спосіб, тільки вбудований в код. Вбудовування відбувається за допомогою ключового слова delegate, після якого йде список параметрів і далі код анонімного методу. Підсумковий результат буде такий самий, начебто був підключений обробник події Show_Message.

Важливо, що на відміну від блоку методів або умовних і циклічних конструкцій, блок анонімних методів повинен закінчуватися крапкою з комою після фігурної дужки, що закриває.

Якщо для анонімного методу не потрібні параметри, то після його імені «порожні» дужки не записуються:

```
delegate void GetMessage();
static void Main(string [] args) {
    GetMessage message = delegate {
Console.WriteLine("анонімний делегат");
};
message();

Console.Read();
}
```

3.4. Лямбди

Лямбда-вираження представляють спрощений запис анонімних методів. Лямбда-вираження дозволяють створити ємні лаконічні методи, які можуть повертати деяке значення і які можна передати як параметри в інші методи.

Ламбда-вирази мають наступний синтаксис: ліворуч від лямбда-оператора => визначається список параметрів, а праворуч блок виразів, який використовує ці параметри:

(Список_параметрів) => вираз.

Наприклад:

```
class  Program {
delegate int Square(int x); // оголошуємо делегат, //
приймає int // і повертає int
static void Main(string [] args) {
Square squareInt = i => i * i; // об'єкту делегата
//надається //лямбда-вираз
```

```
int z = squareInt(6); // використовуємо делегат
Console.WriteLine(z); // виводить число 36
Console.Read();
}
}
```

Тут i => i * i є лямбда-вираз, де i - це параметр, а i * i - вираз.

При використанні необхідно враховувати, що кожен параметр у лямбда-вираженні неявно перетворюється на відповідний параметр делегата, тому типи параметрів повинні бути однаковими. Крім того, кількість параметрів має бути такою самою, як і у делегата. Значення лямбда-виразів, що повертається, повинно бути тим же, що і у делегата.

Модифікуємо клас Account з минулої теми, переписавши прикріплення обробника події за допомогою лямбди-виразу:

```
Account account = new Account(200, 6);

account.Added += (sender, e) => {
Console.WriteLine("Сума транзакції: {0}", e.sum);
Console.WriteLine(e.message);
};
```

Оскільки тут використовується кілька параметрів, вони полягають у дужки. І оскільки в тілі лямбда-вирази застосовується кілька виразів, то вони полягають у блок із фігурних дужок.

Якщо параметри не потрібні, замість параметра в лямбда-вираженні використовуються порожні дужки:

```
class  Program {
delegate void  message(); // делегат без параметрів
static void Main(string [] args) {
        message GetMessage = () => {
Console.WriteLine("Лямбда-вираз");
                                    };

GetMessage();

Console.Read();
}
}
```

Лямбда-вираз необов'язково має приймати блок операторів та виразів. Воно може також приймати посилання на метод:

```
class  Program {
delegate void  message(); // делегат без параметрів
static void Main(string [] args) {
message GetMessage = () => Show_Message();

GetMessage();
```

```
}
private static void  Show_Message() {
Console.WriteLine("Привіт!");
}
}
```

Як і делегати, лямбда-вирази можна передавати як параметри методу. Нерідко лямбди використовуються як параметри:

```
class  Program {
delegate bool IsEqual(int x);

static void Main(string [] args) {
int[] integers = { 1, 2, 3, 4, 5, 6, 7, 8, 9};

// знайдемо суму чисел більше 5
int result1 = Sum(integers, x => x > 5);
Console.WriteLine(result1); // 30

        // Знайдемо суму парних чисел
int result2 = Sum(integers, x => x % 2 == 0);
Console.WriteLine(result2); //20

Console.Read();
}

private static int Sum (int[] numbers, IsEqual func) {
int result = 0;
foreach(int i in numbers) {
if (func(i)) result += i;
}
return result;
}
}
```

Метод Sum приймає як параметр масив чисел і делегат IsEqual, а повертає суму чисел масиву як об'єкта int. У циклі приходить складання значень елементів масиву. Причому складаються ті числа, котрим делегат IsEqual func повертає true. Тобто делегат IsEqual тут задає умову, якій мають відповідати значення масиву. Але на момент написання методу Sum ця умова невідома.

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

```
intresult1 = Sum(integers, x => x > 5);
```

Тут параметр x - значення, яке передається до делегата:

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

А вираз x > 5 є умовою, якій має відповідати це значення. Якщо x відповідає цій умові, то лямбда-вираз повертає true, а передане значення складається з іншими числами.

Подібно працює другий виклик методу Sum, тільки тут виконується перевірка значення на парність, тобто якщо залишок від поділу на 2 дорівнює нулю:

```
intresult2 = Sum(integers, x => x % 2 == 0);
```

то підсумовування виконується.

3.5 Коваріантність та контраваріантність делегатів

Делегати так само, як класи та інтерфейси, можуть бути узагальненими. І так само, як і узагальнені інтерфейси, вони можуть бути підступними і контраваріантними. Завдяки підступності можна привласнити делегату метод, який повертається типом якого є тип, похідний від того типу, який повертається делегатом.

Припустимо, є така структура класів:

```
class Person {
    public string  Name { get; set; }

    Public Person(string  name) {
        Name = name;
    }
    public void  Display() {
        Console.WriteLine(Name);
    }
}
class Client : Person {
    Public Client(string  name) : base(name){

    }
}
```

У цьому випадку підступність делегата може виглядати так:

```
delegatePerson PersonFactory(string  name);
static void Main(string [] args) {
    PersonFactory personDel;
    personDel = BuildClient; // підступність
    Person p = personDel("Tom");
    p.Display();
    Console.Read();
}
private staticClient BuildClient(string  name) {
    return new Client(name);
}
```

Завдяки контраваріантності можна привласнити делегату метод, тип параметра якого є базовим класом по відношенню до класу параметра, який використовується делегатом. Наприклад:

```
delegate void ClientInfo(Client client);
```

```
static void Main(string [] args) {
    // контраваріантність
    ClientInfo clientInfo = GetPersonInfo;
    Client client = new Client("Alice");
    clientInfo(client);
    Console.Read();
}
private static void GetPersonInfo(Person p) {
    p.Display();
}
```

Незважаючи на те, що делегат як параметр приймає об'єкт Client, йому можна присвоїти метод, який приймає як параметр об'єкт базового типу Person.

3.5.1 Коваріантність та контраваріантність в узагальнених делегатах

Узагальнені делегати, як і узагальнені інтерфейси, можуть бути ковариантными і контравариантными. Наприклад, візьмемо ту саму ієрархію класів:

```
class Person {
    public string  Name { get; set; }

    Public Person(string  name) {
        Name = name;
    }

    public virtual void  Display() {
        Console.WriteLine("Person: "+Name);
    }
}
class Client : Person {
    Public Client(string  name) : base(name) {
    }
    public override void  Display() {
        Console.WriteLine("Client:" + Name);
    }
}
```

Тепер оголосимо підступний узагальнений делегат:

```
class  Program {
    delegate T Builder<out T>(string  name);
    static void Main(string [] args) {
        Builder<Person> personBuilder = GetPerson;
        Builder<Client> clientBuilder = GetClient;

        personBuilder = clientBuilder; // підступність

        Person p = personBuilder("Tom"); // виклик //
```

```
делегата
        p.Display(); // Client: Tom

        Console.Read();
    }
    private static Person GetPerson(string  name) {
        return new Person(name);
    }
    private static Client GetClient(string  name) {
        return new Client(name);
    }
}
```

Завдяки використанню out можна присвоїти делегату типу Builder <Person> делегат типу Builder <Client>.

Розглянемо контраваріантний делегат:

```
class  Program {
    delegate void  GetInfo<in T>(T item);
    static void Main(string [] args) {
        GetInfo<Person> personInfo = PersonInfo;
        GetInfo<Client> clientInfo = ClientInfo;

        clientInfo = personInfo; // контраваріантність

        Client client = new Client("Tom");
        clientInfo(client); // Client: Tom

        Console.Read();
    }
    private static void  PersonInfo(Person p) {
        p.Display();
    }

    private static void  ClientInfo(Client cl) {
        cl.Display();
    }
}
```

Використання ключового слова**in**дозволяє присвоїти делегат з більш універсальним типом(GetInfo<Person>) делегату з похідним типом (GetInfo<Client>).

3.6 Делегати Action, Predicate та Func

У .NET є кілька вбудованих делегатів, які використовуються у різних ситуаціях. До них зазвичай відносять делігати Action, Predicate та Func.

3.6.1 Action

Делегат Action є узагальненим, він приймає параметри, але нічого не повертає:

```
publicdelegate void  Action<T>(T obj)
```

Цей делегат має ряд перевантажених версій. Кожна версія приймає різну кількість параметрів:

```
від Action<in T1> до Action<in T1, in T2, .... in T16>.
```

Таким чином, можна передати до 16 значень метод.

Як правило, цей делегат передається як параметр методу і передбачає виклик певних дій у відповідь на дії або події, що відбулися. Наприклад:

```
static void Main(string [] args) {
Action<int, int> op;
op = Add;
Operation(10, 6, op);
op = Substract;
Operation(10, 6, op);

Console.Read();
}

static void Operation(int x1, int x2, Action<int, int>
op) {
if (x1 > x2) op(x1, x2);
}

static void Add(int x1, int x2) {
Console.WriteLine("Сума чисел:" + (x1 + x2));
}

static void Substract(int x1, int x2) {
Console.WriteLine("Різниця чисел:" + (x1 - x2));
}
```

3.6.2 Predicate

Делегат Predicate<T>, як правило, використовується для порівняння деякого об'єкта T певній умові. Як вихідний результат повертається значення true, якщо умова дотримана, і false, якщо не дотримано:

```
Predicate<int> isPositive = delegate (int x) { return
x > 0; };
```

```
Console.WriteLine(isPositive(20));
Console.WriteLine(isPositive(-20));
```

В даному випадку повертається true або false в залежності від того, чи більше нуля вхідний параметр чи ні.

3.6.3 Func

Делегат Func повертає результат дії та може приймати параметри. Він також має багато різних форм:

від Func<out T>(), де T - тип значення, що повертається,

до Func<in T1, in T2,...in T16, out TResult>(),

тобто може приймати до 16 параметрів.

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

Цей делегат також часто використовується як параметр у методах:

```
static void Main(string [] args) {
Func<int, int> retFunc = Factorial;
int n1 = GetInt(6, retFunc);
Console.WriteLine(n1); // 720

int n2 = GetInt (6, x => x * x);
Console.WriteLine(n2); // 36

Console.Read();
}

static intGetInt(int x1, Func<int, int> retF) {
int result = 0;
if (x1 > 0)
result = retF(x1);
return result;
}
static intFactorial(int x) {
int result = 1;
for (int i = 1; i <= x; i++) {
result *= i;
}
return result;
}
```

Метод GetInt() як параметр приймає делегат Func<int, int>, тобто посилання на метод, який приймає число int і повертає значення int.

При першому виклику методу GetInt() йому передається посилання метод обчислення факторіалу. У другому випадку передається лямбда-вираз x => x * x, тобто знову ж таки вираз, який приймає параметр int x і повертає результат x * x.

3.7 Вбудовані методи (Expression-Bodied)

Починаючи з версії C# 6.0 (Visual Studio 2015), з'явилася нова функціональність під назвою*вбудовані методи*або*Expression–Bodied Methods*. Ця функціональність дозволяє застосовувати лямбда-вирази для скороченого написання методів в один рядок. Наприклад, візьмемо пару класів із методами:

```
class Person {
    public string  Name {get;set;}
    public void  Display() {
        Console.WriteLine(this.Name);
    }
}

class Circle {
    public static double  GetArea(double  radius) {
        return radius * radius * 3.14;
    }
}
```

Дія кожного методу визначається у вигляді одного виразу або одного рядка. Тепер застосуємо Expression-Bodied методи:

```
class Person {
    public string  Name {get;set;}
    public void  Display() => Console.WriteLine(Name);
}

class Circle {
    public static double  GetArea(double  radius) =>
radius * radius * 3.14;
}
```

Метод Display() просто виводить значення властивості Name, а метод GetArea() обчислює площу кола, використовуючи переданий метод радіус. Дані методи можуть використовуватися так само, як і будь-які інші методи:

```
class  Program {
    static void Main(string [] args) {
        Person person = new Person { Name = "Tom"};
```

```
        person.Display();

        double  area = Circle.GetArea(50);
        Console.WriteLine("Площа кола з радіусом "+" в 50
дорівнює {0}", area);
        Console.Read();
    }
}
```

Також можна застосовувати лямбди для спрощення написання властивостей, які містять лише блок get і які повертають певне значення:

```
class Person {
    // еквівалентно public string  Name {get {return
name; } }
    public string  Name => name;
    private readonly string  name;

    Public Person(string  _name) {
        name = _name;
    }
    public void  Display() => Console.WriteLine(Name);
}
class  Program {
    static void Main(string [] args) {
        Person person = new Person("Tom");
        person.Display();
        Console.Read();
    }
}
```

3.8. Контрольні питання

1. Що таке делегати. Як вони використовуються для написання програми?
2. Чи можуть бути делегати параметрами методів?
3. Що таке механізм зворотного виклику?
4. Що таке події? Як вони використовуються для написання програми?
5. Яким ключовим словом «оголошуються» події?
6. Яка використовується операція для прикріплення події обробника до певної події?
7. Що таке анонімні методи?
8. Що таке лямбда-вирази?
9. Що таке лямбда-оператор?

1. Що таке коваріантність та контраваріантність делегатів?

4. null та типи Nullable

4.1. Значення null та Nullable-типи

Одна з відмінностей «посилальних типів» від «типів значень» полягає в тому, що об'єкти типів посилань можуть приймати значення null. Наприклад, об'єкт d = null. "Значені типи" так задати не можна - якщо написати щось на зразок int x = null, то отримаємо помилку компіляції.

Однак у різних ситуаціях буває зручно, щоб об'єкти числових типів даних мали значення null, тобто не визначені. Стандартний приклад - робота з базою даних, яка може містити значення null як значення деяких полів. В цьому випадку заздалегідь невідомо, що за значення витягується з бази даних на запит - якесь певне значення або null.

Для "присвоєння" "типу значення" значення null необхідно використовувати знак питання? після типу значень. Наприклад:

```
int?z = null;
bool?enabled = null;
```

Але фактично запис? є спрощеною формою використання структури System.Nullable<T>. Параметр T у кутових дужках є універсальним параметром, замість якого в конкретній задачі підставляється конкретний тип даних. Наступні види визначення змінних будуть еквівалентні та допустимі:

```
int? z1 = 5;
bool? enabled1 = null;
Double ? s1 = 3.3;

Nullable<int> z2 = 5;
Nullable<bool> enabled2 = null;
Nullable <System.Double > s2 = 3.3;
```

Зазначимо, що структура Nullable застосовується тільки для типів значень, оскільки типи посилань «за визначенням» можуть мати значення null:

```
class  Program {
static void Main(string [] args) {
        Nullable<State> state = null;
if(state.HasValue)
Console.WriteLine(state.Value.Name);
        state = new State() { Name = "Аргентина"};
if(state.HasValue)
            Console.WriteLine(state.Value.Name);

// Nullable<Country> country = null; так не можна
```

```
Console.ReadLine();
}
}
class Country {
public string  Name { get; set; }
}
struct State{
public string  Name { get; set; }
}
```

Щоб отримати доступ до значення Nullable, необхідно використовувати властивість Value: state.Value. Щоб перевірити, чи має структура якесь значення, потрібно застосувати властивість HasValue: if (state.HasValue).

Структура Nullable за допомогою методу GetValueOrDefault() дозволяє використовувати значення за промовчанням (для числових типів це 0), якщо значення не задано:

```
int?figure = null;
Console.WriteLine(figure.GetValueOrDefault());
// За замовчуванням використовується 0

Console.WriteLine(figure.GetValueOrDefault(10));
// За замовчуванням використовується 10
```

4.1.1. Рівність об'єктів

При перевірці об'єктів на рівність слід враховувати, що вони рівні не тільки коли вони мають ненульові значення, які збігаються, але і коли обидва об'єкти рівні null:

```
int?x1 = null;
int?x2 = null;
if(x1 == x2)
Console.WriteLine("об'єкти рівні");
else
Console.WriteLine("об'єкти не рівні");
```

Якщо ж один об'єкт має значення null, а інший ні, або обидва об'єкти мають різні значення, то вони не рівні.

4.1.2. Перетворення типів Nullable

Розглянемо приклади можливих перетворень:

- **явне перетворення від T? до T**

```
int?x1 = null;
if(x1.HasValue)
int x2 = (int) x1;
Console.WriteLine(x2);
}
```

- **неявне перетворення від T до T?**

```
intx1 = 4;
int? x2 = x1;
Console.WriteLine(x2);
```

- **неявні розширюючі перетворення від V до T?**

```
intx1 = 4;
long? x2 = x1;
Console.WriteLine(x2);
```

- **явні звуження перетворення від V до T?**

```
long x1 = 4;
int? x2 = (int?) x1;
```

- **перетворення від V? до T?**

```
long? x1 = 4;
int? x2 = (int?) x1;
```

- **явні перетворення від V? до T**

```
long? x1 = 4;
intx2 = (int) x1;
```

4.1.3. Оператор ??

Оператор ?? називається оператором null-об'єднання. Він застосовується для встановлення значень за умовчанням для типів значень та типів посилань, які допускають значення null. Оператор ?? повертає лівий операнд, якщо цей операнд не дорівнює null. Інакше повертається правий операнд. У цьому лівий операнд повинен приймати null.

Розглянемо приклад:

```
int?x = null;
inty = x ?? 100; // одно 100, оскільки x дорівнює null

int?z = 200;
intt = z ?? 44; // одно 200, оскільки z не дорівнює null
```

Але не можна:

```
intx = 44;
inty = x ?? 100;
```

Тут змінна x не є nullable-типом і не може набувати значення null, тому в якості лівого операнда в операції ?? вона використовуватись не може.

4.2. Елвіс оператор

Починаючи з C# 6.0 (Visual Studio 2015), у мові з'явився елвіс-оператор або «оператор умовного null» (Null-Conditional Operator). Він дозволяє спростити перевірку об'єктів значення null в умовних конструкціях.

Наприклад, нехай є наступна система класів:

```
class User {
    public Phone Phone { get; set; }
}

class Phone {
```

```
    public Company Company { get; set; }
}

class Company {
    public string  Name { get; set; }
}
```

Об'єкт User містить посилання на об'єкт Phone, а об'єкт Phone містить посилання на об'єкт Company, тому теоретично можна отримати з об'єкта User назву компанії, наприклад:

```
User user = new User();
user.Phone=new Phone { Company = new Company { Name =
"Samsung" } };
string companyName = user.Phone.Company.Name;
```

Однак цілком може скластися ситуація, що при отриманні назви компанії властивість Phone або властивість Company або сам об'єкт користувача буде невизначений і мати значення null. У загальному випадку для визначення цього необхідно використовувати умовну конструкцію:

```
User user = new User();

if(user!=null) {
    if(user.Phone!=null) {
        if (user.Phone.Company! = null) {
            string  companyName =
user.Phone.Company.Name;
            Console.WriteLine(companyName);
        }
    }
}
```

Виходить багатоповерхова конструкція, яку можна скоротити:

```
if(user!=null && user.Phone!=null &&
user.Phone.Company!=null) {
    string  companyName = user.Phone.Company.Name;
    Console.WriteLine(companyName);
}
```

Якщо user не дорівнює null, то перевіряється такий вираз user.Phone! = null тощо. Конструкція набагато простіша, але все одно виходить досить великою. І щоб її спростити, C# 6.0 можна використовувати**елвіс-оператор**:

```
string companyName = user?.Phone?.Company?.Name;
```

Вираз?. та представляє елвіс-оператор. Тут послідовно перевіряється чи об'єкт user і вкладені об'єкти значення null. Якщо ж на якомусь етапі один з об'єктів виявиться рівним null, то companyName матиме значення за умовчанням, тобто порожній рядок.

І в цьому випадку можна «піти далі» і застосувати операцію ?? для встановлення значення за замовчуванням ("не встановлено"), якщо рядок є порожнім:

```
User user = new User();
string companyName = user?.Phone?.Company?.Name??"не встановлено";
Console.WriteLine(companyName);
```

4.3 Контрольні питання

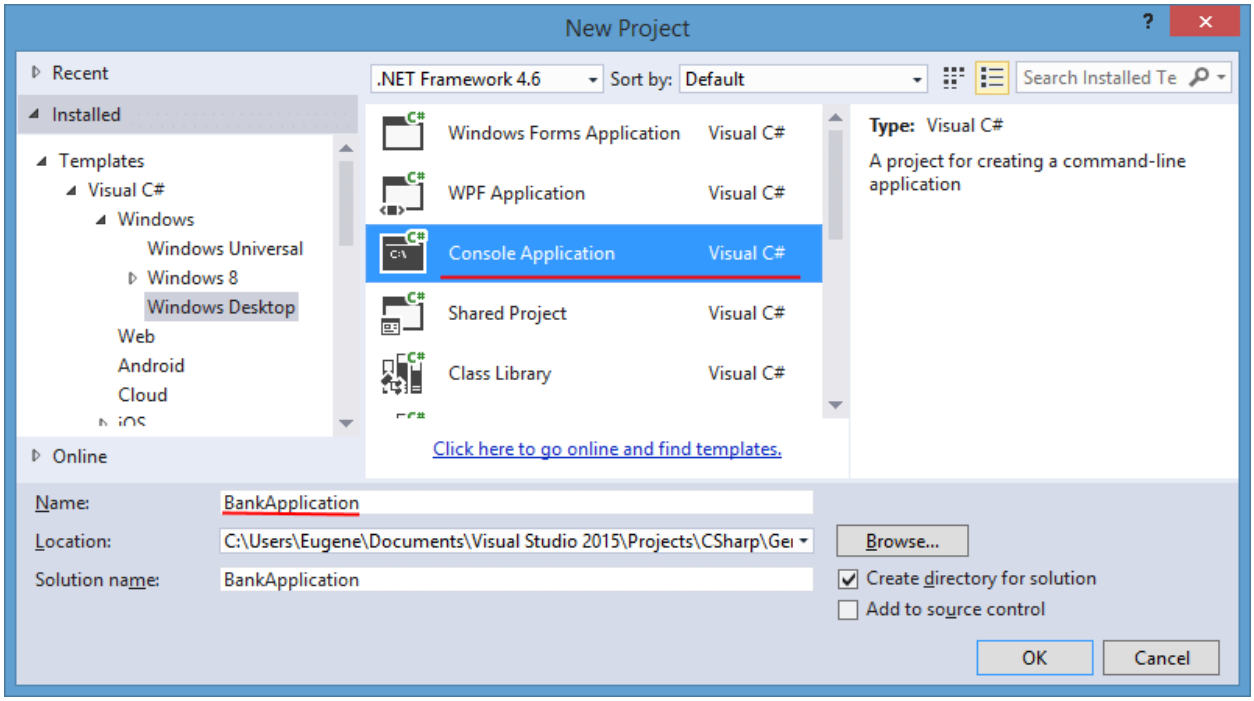
1. Що таке значення null та Nullable-типи?
2. Що таке оператор null-об'єднання?
3. Що таке елвіс-оператор?

5. Практичний приклад

5.1. Створення проекту бібліотеки класів

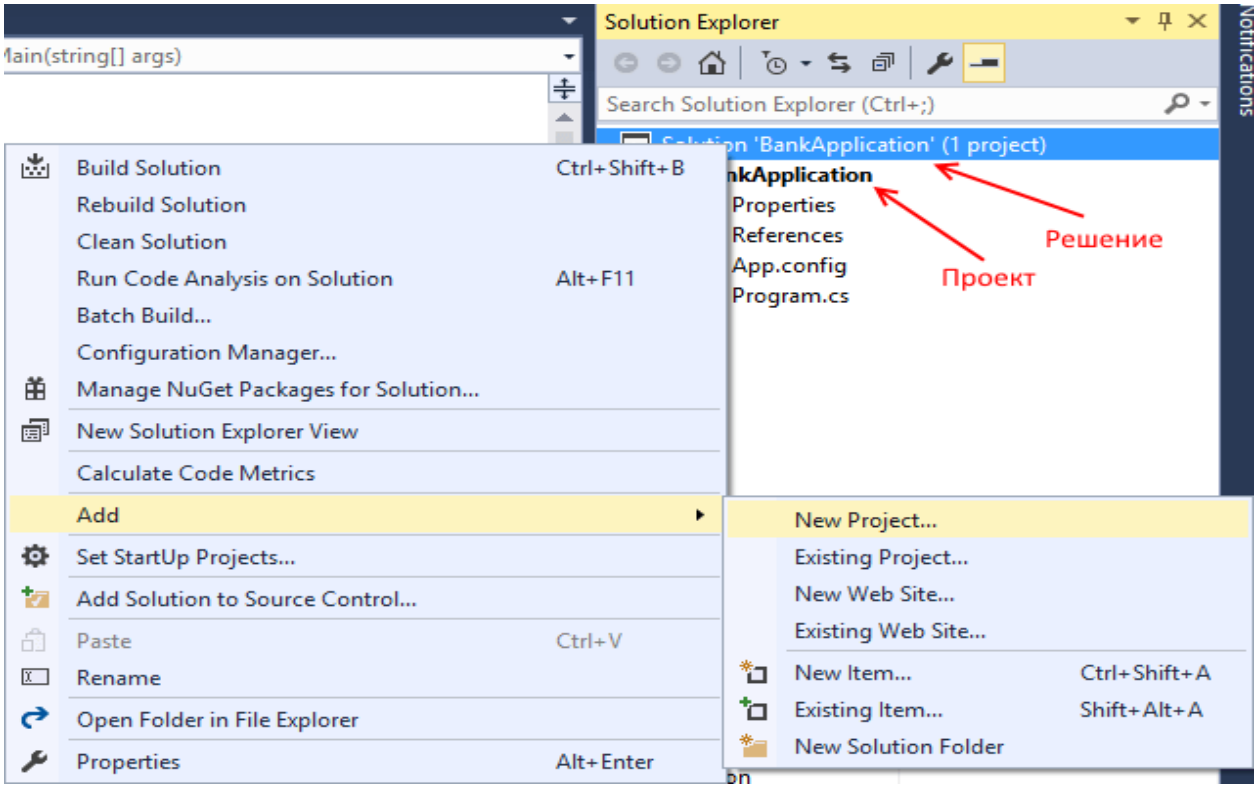
У попередніх розділах було розглянуто основні концепції об'єктно-орієнтованого програмування. Далі розглядається розробка програми, в якій на практичному прикладі демонструються ці концепції (за матеріалами сайту http://metanit.com).

Спочатку створимо новий проект типу Console Application, який назвемо BankApplication:

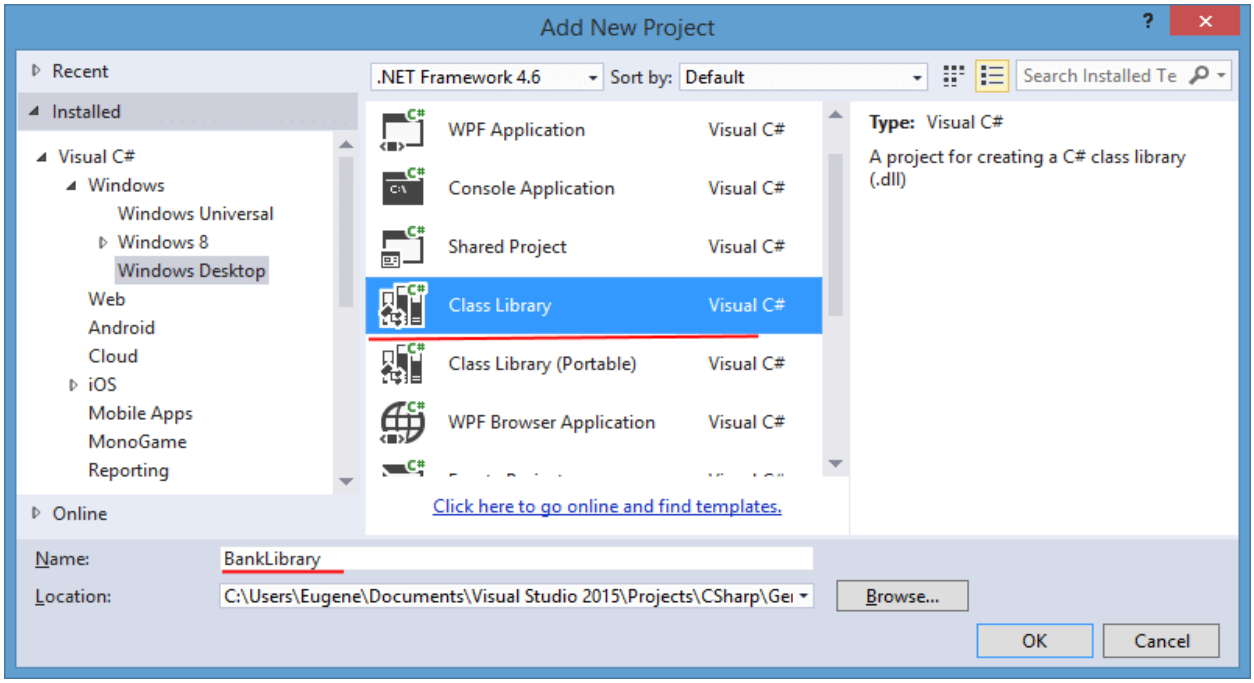


І після цього створюється стандартний порожній проект із класом Program та методом Main. Це буде головний проект програми.

Але для зберігання класів та інтерфейсів нерідко створюються окремі проекти, в рамках яких усі класи компілюються у файл бібліотеки dll, яка потім підключається до головного проекту. Тому додамо до рішення новий проект. Для цього натиснемо правою кнопкою миші на рішення та виберемо в контекстному меню Add -> New Project...:



Як тип нового проекту виберемо шаблон Class Library (Бібліотека класів) і назвемо новий проект BankLibrary:



Після цього до рішення додається новий проект, який за умовчанням має один файл Class 1.cs. Він не потрібний, тому видалимо цей файл.

Цей проект міститиме всі класи, які використовуватимуть головний проект.

Додаток імітуватиме роботу банку. І перш ніж розпочати роботу над додатком, виділимо сутності, які використовуватимемо, а також відносини між сутностями. Виділимо такі сутності, як сам банк, банківський рахунок. Рахунки бувають різних видів, наприклад, рахунки до запитання та депозити. Відповідно буде кілька сутностей рахунків.

Додамо до проекту BankLibrary новий інтерфейс, який описуватиме функціональність банківського рахунку.

При проектуванні інтерфейсу слід пам'ятати, що він визначає загальний функціонал, який повинен бути реалізований у класах, що застосовують даний інтерфейс. Причому, всі члени цього інтерфейсу є публічними або загальнодоступними. Тобто якщо необхідно визначити та використовувати в класі якісь властивості, методи, події, які не повинні мати модифікатор public, то інтерфейс для визначення подібних властивостей та методів не підходить.

Отже, додамо до проекту BankLibrary інтерфейс IAccount:

```
public interface IAccount {
// Покласти гроші на рахунок
void  Put(decimal sum);
    // Взяти з рахунку
decimal Withdraw(decimal sum);
}
```

Даний інтерфейс визначаємо два методи: покласти на рахунок та зняти (взяти) кошти з рахунку.

Для реакцію зміни стану рахунки будемо використовувати подієву модель, тобто оброблятися різні зміни рахунки через події. Для цього додамо до

проекту BankLibrary новий файл AccountStateHandler.cs, у якому визначимо делегат та допоміжний клас:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BankLibrary
{
public delegate void  AccountStateHandler(object sender,
AccountEventArgs e);

public class  AccountEventArgs {
        // Повідомлення
public string  Message { get; private set;}
// Сума, яку змінився рахунок
public decimal Sum { get; private set;}

public AccountEventArgs(string  _mes, decimal _sum) {
Message = _mes;
Sum = _sum;
}
}
}
```

Делегат AccountStateHandler буде використовуватися для створення подій. Для обробки подій визначимо клас AccountEventArgs, який має дві властивості доступні для читання: повідомлення про подію та сума, на яку змінився рахунок.

Тепер визначимо основний клас програми Account.

5.2. Створення класів. Частина 1

Додамо до проекту BankLibrary новий клас Account, який матиме таке визначення:

```
public abstract class  Account : IAccount {
    //Подія, що виникає під час виведення грошей
    protected internal event AccountStateHandler
Withdrawed;
    // Подія, що виникає при додаванні на рахунок
    protected internal event AccountStateHandler Added;
    // Подія, що виникає при відкритті рахунку
    protected internal event AccountStateHandler Opened;
    // Подія, що виникає при закриття рахунку
    protected internal event AccountStateHandler Closed;
```

```
    // Подія, що виникає при нарахуванні відсотків
    protected internal event AccountStateHandler
Calculated;

    protected int _id;
    static int counter = 0;

    protected decimal _sum; // Змінна для // зберігання
суми
    protected int _percentage; // Змінна для //
зберігання відсотка

    protected int _days = 0; // час з моменту //
відкриття рахунку

    public Account(decimal sum, int percentage) {
        _sum = sum;
        _percentage = percentage;
        _id=++counter;
    }

    // Поточна сума на рахунку
    public decimal CurrentSum {
        get {return_sum; }
    }

    public int Percentage {
        get {return_percentage; }
    }

    public int Id {
        get {return_id; }
    }
    //виклик подій
    private void  CallEvent(AccountEventArgs e,
AccountStateHandler handler) {
        if (handler != null && e!=null) handler(this, e);
    }
    // Виклик окремих подій. // Для кожної події
визначається
    / / Свій витуальний метод
    protected virtual void  OnOpened(AccountEventArgs e)
{
        CallEvent(e, Opened);
    }
```

```
    protected virtual void  OnWithdrawed(AccountEventArgs
e) { CallEvent(e, Withdrawed);
    }
    protected virtual void  OnAdded(AccountEventArgs e) {
        CallEvent(e,Added);
    }
    protected virtual void  OnClosed(AccountEventArgs e)
{
        CallEvent(e, Closed);
    }
    protected virtual void  OnCalculated(AccountEventArgs
e) {
        CallEvent(e, Calculated);
    }

    public virtual void  Put(decimal sum) {
        _sum + = sum;
      OnAdded(new AccountEventArgs("На рахунок надійшло"
+ sum, sum));
    }
    public virtual decimal Withdraw(decimal sum) {
        decimal result = 0;
        if (sum <= _sum) {
            _sum -= sum;
            result = sum;
            OnWithdrawed(new AccountEventArgs("Сума" +
sum + "знята з рахунку" + _id, sum));
        } else {
            OnWithdrawed(new AccountEventArgs(
"Недостатньо грошей на рахунку" + _id, sum));
        }
        return result;
    }
    //відкриття рахунку
    protected internal virtual void Open() {
        OnOpened(new AccountEventArgs( "Відкрито новий
депозитний рахунок!Id рахунки: " +
                                    this._id, this._sum));
    }
    //закриття рахунку
    protected internal virtual void  Close() {
        OnClosed(new AccountEventArgs("Рахунок " + _id +
" закритий. Загальна сума: " + CurrentSum, CurrentSum));
    }
```

```
    protected internal void  IncrementDays() {
        _days++;
    }
    // нарахування відсотків
    protected internal virtual void Calculate() {
        decimal increment = _sum * _percentage/100;
        _sum = _sum + increment;
        OnCalculated(new AccountEventArgs( "Нараховані
відсотки у розмірі: " + increment, increment));
    }
}
```

Оскільки «як такого» банківського рахунку немає, а є конкретні рахунки – депозит, до запитання тощо, то цей клас є абстрактним. У той самий час він реалізує інтерфейс IAccount.

Він визначає ряд подій, що викликаються при зміні стану. Для визначення подій застосовується раніше створений делегат AccountStateHandler.

Кожен рахунок має унікальний ідентифікатор id - унікальний номер. Для його одержання застосовується статичний лічильник counter.

Майже всі методи є віртуальними. Насамперед, визначаються методи генерації. Метод CallEvent() викликає подію, яку представляє делегат AccountStateHandler і передає йому аргументи події - об'єкт AccountEventArgs. Для виклику окремих подій за допомогою методу CallEvent передбачені методи OnAdded, OnCalculated, OnClosed, OnOpened, OnWithdrawed. Подібна організація виклику подій дозволить простіше перевизначити модель виклику в класах-спадкоємцях та передати різні для кожного класу-спадкоємця об'єкти AccountEventArgs.

Звернення до цих методів іде всередині методів, які відповідають за функціональність рахунку. Наприклад, у методі Put зберігається логіка із зарахування коштів на рахунок. І в цьому методі через виклик OnAdded() буде генеруватися подія Added.

Також слід зазначити, що більшість членів класу мають модифікатор protected internal, тобто вони будуть видні лише всередині проекту BankLibrary.

Абстрактний клас Account визначає поліморфний інтерфейс, який успадковується чи перевизначається похідними класами. Тепер додамо в проект новий клас, який представлятиме рахунок до запитання і який буде називатися DemandAccount:

```
public class  DemandAccount : Account {
    public DemandAccount(decimal sum, int percentage) :
base(sum, percentage) {
    }

    protected internal override void  Open() {
        base.OnOpened(new AccountEventArgs( "Відкритий
новий рахунок до запитання! Id рахунки: " + this._id,
this._sum));
```

```
    }
}
```

У цьому класі перевизначено один метод Open. При необхідності можна перевизначити і більше функціоналу, але обмежимося в цьому випадку цим.

Додамо другий клас-спадкоємець DepositAccount:

```
public class  DepositAccount : Account {
    public DepositAccount(decimal sum, int percentage) :
base(sum, percentage){
    }
    protected internal override void  Open() {
        base.OnOpened(new AccountEventArgs( "Відкритий
новий депозитний рахунок!Id рахунки: " + this._id,
this._sum));
    }

    public override void  Put(decimal sum) {
        if (_days % 30 == 0) base.Put(sum);
        else
            base.OnAdded( new AccountEventArgs( "На
рахунок можна покласти тільки "+ "після 30-ти денного
періоду", 0));
    }

    public override decimal Withdraw(decimal sum) {
        if (_days % 30 == 0) return base.Withdraw(sum);
        else
            base.OnWithdrawed( new AccountEventArgs(
"Вивести кошти можна тільки"+ "після 30-ти денного
періоду", 0));
        return 0;
    }

    protected internal override void  Calculate() {
        if (_days % 30 == 0) base.Calculate();
    }
}
```

Депозитні рахунки мають особливість: вони оформляються на тривалий період, що накладає певні обмеження. Тому тут перевизначаються ще три методи. Припустимо, що депозитний рахунок має строк у 30 днів, в межах якого клієнт не може ні додати на рахунок, ні вивести частину коштів з рахунку, крім закриття всього рахунку. Тому при всіх операціях перевіряємо кількість минулих днів для цього рахунку:

```
if(_days % 30 == 0)
```

У разі порівнюємо залишок розподілу кількості днів на 30 днів. Якщо залишок від поділу дорівнює 0, то пройшов черговий 30-денний період, по

закінченню якого відбувається нарахування відсотків, можливе виведення коштів або їх додавання.

Власне, це всі класи банківських рахунків, які будуть використовуватися в програмі.

5.3. Створення класів. Частина 2

Як правило, банківські рахунки існують не власними силами, а всередині банку, який виступає деяким контейнером рахунків і виконує функції з управління ними. Тому додамо до проекту BankLibrary новий клас Bank:

```
public class  Bank<T> where T : Account {
    T[] accounts;

    public string  Name { get; private set; }

    public Bank(string  name) {
        this.Name = name;
    }
    //Метод створення рахунку
    public void  Open(AccountType accountType, decimal
sum,
        AccountStateHandler addSumHandler,
AccountStateHandler withdrawSumHandler,
           AccountStateHandler calculationHandler,
AccountStateHandler closeAccountHandler,
                AccountStateHandler openAccountHandler) {
        T newAccount = null;

        switch (accountType) {
           case AccountType.Ordinary:
             newAccount = new DemandAccount(sum, 1) as T;
                break;
           case AccountType.Deposit:
           newAccount = new DepositAccount(sum, 40) as T;
                break;
        }

        if (newAccount == null)
         throw newException("Помилка створення рахунку");
        // додаємо новий рахунок до масиву рахунків
        if (accounts == null)
            accounts = new T[] { newAccount };
        else {
           T[] tempAccounts = новий T[accounts.Length +
1];
```

```
            for (int i = 0; i < accounts.Length; i++)
                tempAccounts[i] = accounts[i];
            tempAccounts[tempAccounts.Length - 1] =
                                                NewAccount;
            accounts = tempAccounts;
        }
        // встановлення обробників подій рахунку
        newAccount.Added += addSumHandler;
        newAccount.Withdrawed += withdrawSumHandler;
        newAccount.Closed += closeAccountHandler;
        newAccount.Opened += openAccountHandler;
        newAccount.Calculated += calculationHandler;

        newAccount.Open();
    }
     //додавання коштів у рахунок
    public void  Put(decimal sum, int id) {
        T account = FindAccount(id);
        if (account == null)
            throw new Exception("Рахунок не знайдений");
        account.Put(sum);
    }
    // виведення коштів
    public void  Withdraw(decimal sum, int id) {
        T account = FindAccount(id);
        if (account == null)
            throw new Exception("Рахунок не знайдений");
        account.Withdraw(sum);
    }
    // Закриття рахунку
    public void  Close(int id) {
        int index;
        T account = FindAccount(id, out index);
        if (account == null)
            throw new Exception("Рахунок не знайдений");

        account.Close();

        if (accounts.Length <= 1)
            accounts = null;
        else {
            // зменшуємо масив рахунків, // видаляючи із
нього закритий рахунок
            T[] tempAccounts = новий T[accounts.Length -
1];
```

```
            for (int i = 0, j = 0; i < accounts.Length;
i++) {
                if (i! = index)
                    tempAccounts[j++] = accounts[i];
            }
            accounts = tempAccounts;
        }
    }

    // нарахування відсотків за рахунками
    public void  CalculatePercentage() {
        if (accounts == null) // якщо масив не створено,
// виходимо з методу
            return;
        for (int i = 0; i < accounts.Length; i++) {
            T account = accounts[i];
            account.IncrementDays();
            account.Calculate();
        }
    }

    // пошук рахунку за id
    public T FindAccount(int id) {
        for (int i = 0; i < accounts.Length; i++) {
            if (accounts[i].Id == id)
                return accounts[i];
        }
        return null;
    }
       // перевантажена версія пошуку рахунку
    public T FindAccount(int id, out int index) {
        for (int i = 0; i < accounts.Length; i++) {
            if (accounts[i].Id == id) {
                index = i;
                return accounts[i];
            }
        }
        index = -1;
        return null;
    }
}
// тип рахунку
publicenum AccountType { Ordinary, Deposit }
```

Клас банку є узагальненим. У цьому параметр T має обмеження: він має представляти клас Account чи його спадкоємців. Тому у будь-якого об'єкта T будуть доступні методи та властивості класу Account.

Всі рахунки у класі зберігаються у масиві рахунків. На момент проектування класу невідомо, якими саме рахунками управлятиме банк. Можливо, це будуть будь-які рахунки, а можливо лише депозитні, тобто об'єкти DepositAccount. Тому використання узагальнень дозволяє додати більше гнучкості при розробці та проектуванні програми.

Створення нового рахунку відбувається у методі Open. До нього передається ряд параметрів, зокрема, тип рахунку, який описується переліком:

```
publicenum AccountType {
    Ordinary, Deposit
}
```

Цей перелік можна визначити після класу Bank. Залежно від типу рахунку створюється об'єкт DemandAccount або DepositAccount і потім додається до масиву accounts. Оскільки масиви не розширюються автоматично, фактично створюється новий масив зі збільшенням елементів на одиницю і в кінець нового масиву додається новий елемент.

При цьому параметризація (тобто застосування та/або створення узагальнених класів) вимагає виконання «приведення» - створений об'єкт необхідно приводити до типу T:

```
newAccount = new DemandAccount(sum, 1) as T;
```

Таке приведення дозволить уникнути помилок, наприклад, якщо типізується класBank не типом Account, а типом DepositAccount, то перетворення newAccount = DemandAccount(sum, 1) as T поверне значення null. Це легко перевіряється:

```
if(newAccount == null)
```

У метод Open також передаються обробники всім подій класу Account, які встановлюються після створення об'єкта Account.

На завершення «створення» Account викликається метод OnOpened(), який генерує подію Account.Opened, завдяки чому ззовні можна отримати повідомлення про цю подію.

Для пошуку рахунку в масиві по id визначається метод FindAccount(). Його перевантажена версія дозволяє отримувати індекс знайденого елемента через вихідний параметр index:

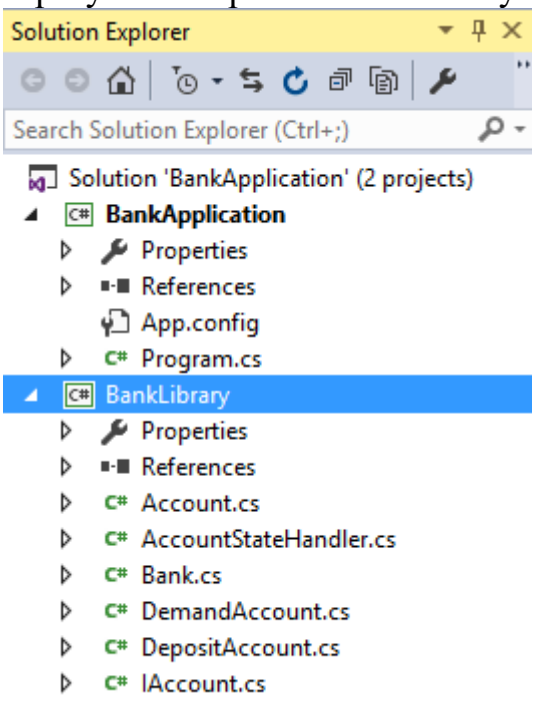
```
public T FindAccount(int id, out int index)
```

У методах Put, Withdraw і Close використовуються метод FindAccount() для отримання рахунку, додавання або виведення коштів, а також закриття рахунку. При закритті рахунку методом Close() створюється новий масив без одного елемента - рахунки, який треба видалити. Таким чином відбувається видалення рахунку.

У методі CalculatePercentage() «пробігаємо» по всіх елементах масиву рахунків, збільшуємо у кожного рахунку лічильник днів і робимо нарахування відсотків.

Загалом, клас Bank є обгорткою, через яку з головного проекту здійснюватиметься робота з об'єктами Account.

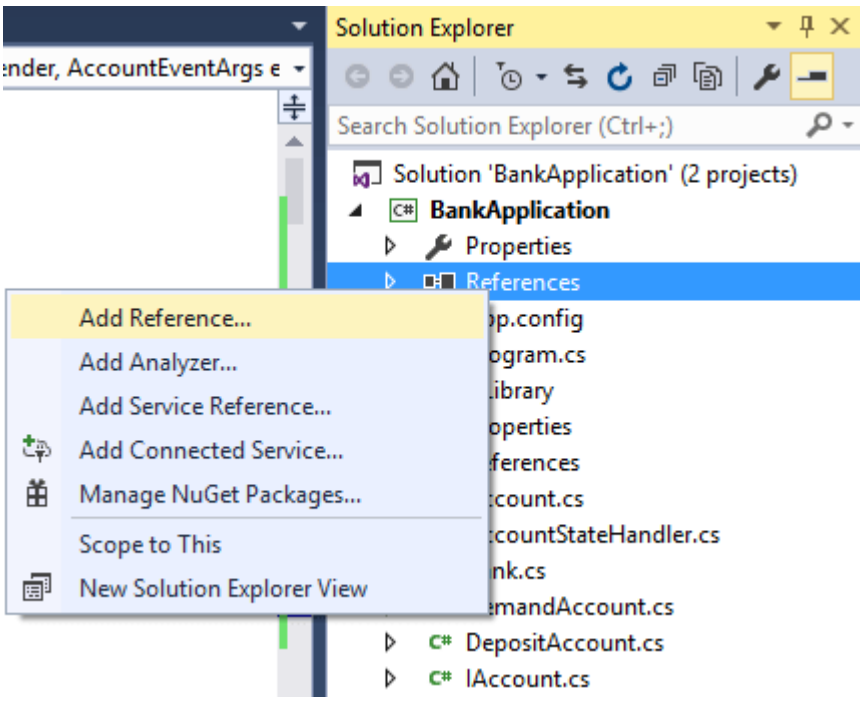
У результаті проект BankLibrary має виглядати так:



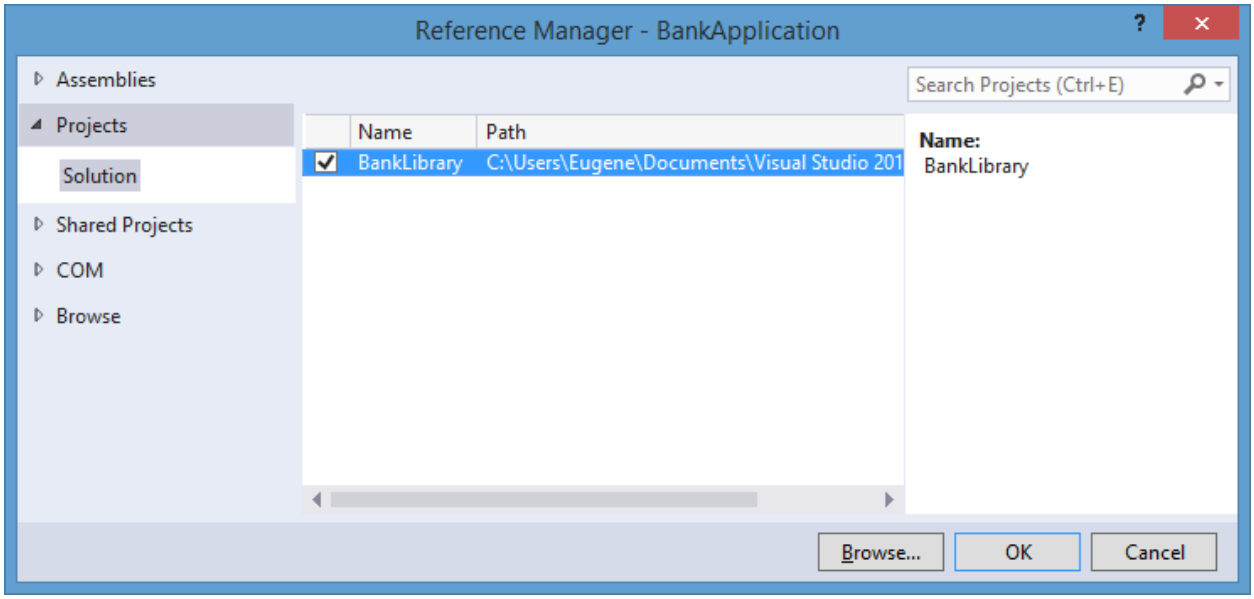
І тепер збудуємо проект. Для цього натиснемо на назву проекту у вікні Solution Explorer (Обозреватель рішень) правою кнопкою миші і в контекстному меню виберемо пункт Build. Після цього у проекті у папці*bin/Debug*буде створено файл бібліотеки класів із розширенням dll.

5.4. Створення головного проекту

Спочатку підключимо скомпільовану раніше бібліотеку класів. Для цього в головному проекті BankApplication натиснемо на пункт References правою кнопкою миші і в меню виберемо пункт Add Reference...:



Потім у вікні відзначимо пункт BankLibrary, який представлятиме нашу бібліотеку класів, і натиснемо на OK.



Після цього до проекту буде додано посилання на бібліотеку. І якщо розкриємо вузол Reference, зможемо побачити її серед підключених бібліотек.

Тепер змінимо файл Program.cs у головному проекті таким чином:

```
using System;
using BankLibrary;

namespace BankApplication {
    class  Program {
        static void Main(string [] args) {
            Bank<Account> bank = new
Bank<Account>("ЮнітБанк");
            bool alive = true;
```

```
            while (alive) {
             ConsoleColor color =
Console.ForegroundColor;
             // Виводимо список команд зеленим кольором
             Console.ForegroundColor =
ConsoleColor.DarkGreen;
             Console.WriteLine("1. Відкрити рахунок \t"+
"2. Вивести кошти \t"+ "3. Додати на рахунок");
             Console.WriteLine("4. Закрити рахунок \t"+
"5. Пропустити день \t"+ "6. Вийти з програми");
              Console.WriteLine("Введіть номер пункту:");
              Console.ForegroundColor = color;
              try {
                   int command =
Convert.ToInt32(Console.ReadLine());
                  switch (command) {
                        case 1:
                            OpenAccount(bank);
                            break;
                        case 2:
                            Withdraw(bank);
                            break;
                        case 3:
                            Put(bank);
                            break;
                        case 4:
                            CloseAccount(bank);
                            break;
                        case 5:
                            break;
                        case 6:
                            alive = false;
                            continue;
                  }
                  bank.CalculatePercentage();
                }
                catch (Exception ex) {
           // виводимо повідомлення про помилку червоним
кольором
                   color = Console.ForegroundColor;
                   Console.ForegroundColor =
ConsoleColor.Red;
                    Console.WriteLine(ex.Message);
                    Console.ForegroundColor = color;
                }
```

```
            }
        }

        private static void  OpenAccount(Bank<Account>
bank) {
            Console.WriteLine( "Вкажіть суму для
створення рахунку:");
            decimal sum =
Convert.ToDecimal(Console.ReadLine());
            Console.WriteLine("Виберіть тип рахунку: "+
"1. До запитання "+"2. Депозит");
            AccountType accountType;
            int type =
Convert.ToInt32(Console.ReadLine());
            if (type == 2) accountType =
AccountType.Deposit;
            else accountType = AccountType.Ordinary;
            bank.Open(accountType,
                sum,
               // Обробник додавання коштів на рахунок
               AddSumHandler,
               // Обробник виведення коштів
               WithdrawSumHandler,
               // обробник нарахувань відсотків // як
лямбда-выражения
               (o, e) => Console.WriteLine(e.Message),
                CloseAccountHandler, // обробник //
закриття рахунку
                OpenAccountHandler); // Обробник //
Відкриття рахунку
        }

        private static void  Withdraw(Bank<Account>
bank){
            Console.WriteLine( "Вкажіть суму для
виведення з рахунку:");
            decimal sum =
Convert.ToDecimal(Console.ReadLine());
            Console.WriteLine("Введіть id рахунки:");
            int id = Convert.ToInt32(Console.ReadLine());
            bank.Withdraw(sum, id);
        }
        private static void  Put(Bank<Account>bank) {
            Console.WriteLine("Вкажіть суму, щоб "+"
покласти на рахунок:");
```

```
            decimal sum =
Convert.ToDecimal(Console.ReadLine());
            Console.WriteLine("Введіть Id рахунки:");
            int id = Convert.ToInt32(Console.ReadLine());
            bank.Put(sum, id);
        }

        private static void  CloseAccount(Bank<Account>
bank){
            Console.WriteLine("Введіть id рахунки, "+"
який треба закрити:");
            int id = Convert.ToInt32(Console.ReadLine());
            bank.Close(id);
        }

        // Обробники подій класу Account
        // Обробник відкриття рахунку
        private static void  OpenAccountHandler(object
sender, AccountEventArgs e) {
            Console.WriteLine(e.Message);
        }

        // Обробник додавання грошей на рахунок
        private static void  AddSumHandler(object sender,
AccountEventArgs e) {
            Console.WriteLine(e.Message);
        }

        // Обробник виведення коштів
        private static void  WithdrawSumHandler(object
sender, AccountEventArgs e) {
            Console.WriteLine(e.Message);
            if (e.Sum > 0)
                Console.WriteLine("Йдемо витрачати
гроші");
        }

        // обробник закриття рахунку
        private static void  CloseAccountHandler(object
sender, AccountEventArgs e) {
            Console.WriteLine(e.Message);
        }
    }
}
```

На початку файлу підключається бібліотека:

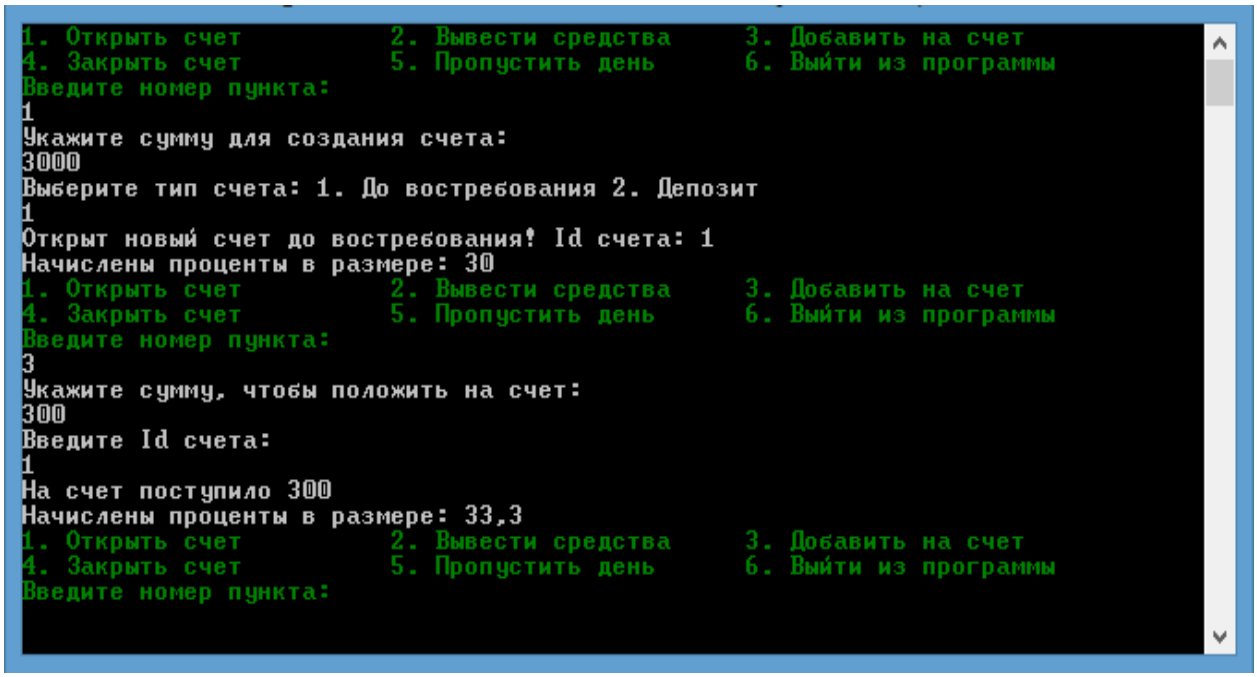
```
usingBankLibrary;
```

У методі Main створюється об'єкт Bank, типизований класом Account і через нього відбувається взаємодія з об'єктами Account.

У циклі while виводиться список команд, який має обрати користувач. Після вибору однієї з них у конструкції switch виконується відповідна команда. Кожна команда представляє отримання введення від користувача, його перетворення за допомогою класу Convert та передання аргументів методам об'єкта Bank.

Кожна ітерація циклу while відповідає одному дню. Тому наприкінці циклу викликається метод bank.CalculatePercentage(), який збільшує об'єкти Account лічильник днів і здійснює нарахування відсотків.

У результаті вийде наступна програма, що імітує роботу банку та взаємодію з користувачем:



Література

1. Зеленський О. С., Лисенко В. С. Розробка Windos-додатків на мові C# : навч. посіб. Ч. 1. Кривий Ріг : Держ. ун-т економіки і технологій, 2023. 160 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058684.pdf.
2. Зеленський О. С., Лисенко В. С. Розробка Windos-додатків на мові C# : навч. посіб. Ч. 2. Кривий Ріг : Держ. ун-т економіки і технологій, 2023. 357 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058674.pdf.
3. Бичков О. С., Іванов Є. В. Об'єктно-орієнтоване програмування мовою C# : підручник. Київ : Київ. ун-т, 2018. 208 с.

4. Решевська К. С., Лісняк А. О., Борю С. Ю. Об'єктно-орієнтоване програмування : навч. посіб. Запоріжжя : ЗНУ, 2020. 94 с. URL: http://ebooks.znu.edu.ua/files/metodychky/2020/06/0045039.doc.
5. Бобков В. Б., Грудзинський Ю. Є., Крилов К. В. Програмування - 2. Об'єктно-орієнтоване програмування : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2023. 77 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi84/0063780.pdf.
6. Гришанович Т. О., Глинчук Л. Я. Основи об'єктно-орієнтованого програмування : навч. посіб. / ВНУ ім. Лесі Українки. Луцьк : ВНУ ім. Лесі Українки, 2022. 120 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051132.pdf.
7. Дібрівний О. А., Гребенюк В. В. Вступ до об'єктно орієнтованого програмування C# : навч. посіб. Київ : Державний університет телекомунікацій, 2018. 190 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi73/0053671.pdf.
8. Олефіренко Н. В., Курганський А. Р., Остапенко Л. П., Пономарьова Н. О. Об'єктно-орієнтоване програмування мовою C# : навч. посіб. Харків : ХНПУ ім. Г. С. Сковороди, 2024. 254 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi84/0063778.pdf.