МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

О. В. Щербаков
Ю. Е. Парфьонов
В. М. Федорченко

ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Навчальний посібник

Харків
ХНЕУ ім. С. Кузнеця
2019

УДК 004.43(075.034)
Щ61

Авторський колектив: канд. техн. наук, доцент О. В. Щербаков – підрозділи 2, 4, 5, 8; канд. техн. наук, доцент Ю. Е. Парфьонов – підрозділи 3, 6, 7, 10, 11; канд. техн. наук, доцент В. М. Федорченко – підрозділи 1, 9.

Рецензенти: доцент кафедри комп'ютерних систем, мереж та кібербезпеки Національного аерокосмічного університету ім. М. Є. Жуковського "ХАІ", канд. техн. наук *А. В. Шостак;* проректор з інформаційних технологій Харківського гуманітарного університету "Народна українська академія", канд. техн. наук, доцент *В. П. Козиренко.*

Рекомендовано до видання рішенням ученої ради Харківського національного економічного університету імені Семена Кузнеця.
Протокол № 9 від 27.05.2019 р.

Самостійне електронне текстове мережеве видання

Щербаков О. В.

Щ61 Основи об'єктно-орієнтованого програмування [Електронний ресурс] : навчальний посібник / О. В. Щербаков, Ю. Е. Парфьонов, В. М. Федорченко. – Харків : ХНЕУ ім. С. Кузнеця, 2019. – 237 с.
ISBN 978-966-676-759-5

Викладено основні елементи об'єктно-орієнтованої парадигми. Велику увагу приділено їх реалізації у сучасних об'єктно-орієнтованих мовах програмування C# і Java. Розглянуто призначення та функціональність головних бібліотечних класів .Net Framework і Java SE.

Рекомендовано для студентів, викладачів і користувачів, які вивчають основи об'єктно-орієнтованого програмування.

УДК 004.43(075.034)

ISBN 978-966-676-759-5

Вступ

Об'єктно-орієнтована технологія створення програмного забезпечення була задумана та розроблена як інструмент подолання складності користування. Вона успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування програмного забезпечення. Головним завданням об'єктно-орієнтованого підходу є забезпечення способу структурування програми та керування складними взаємозв'язками між великою кількістю компонентів системи.

У навчальному посібнику викладаються основи об'єктно-орієнтованого програмування на прикладі алгоритмічних мов C# і платформ Microsoft .NET, Java та платформи Java SE.

Компанія Microsoft позиціонує .NET як базову платформу розробки програмного забезпечення найближчими роками. Технологія .NET – це технологія для розробки Web-застосувань і програмного забезпечення, орієнтованого на операційну систему Windows. За допомогою .NET можна також розробляти програмне забезпечення для портативних комп'ютерів і мобільних телефонів.

Мова програмування C# створена корпорацією Microsoft спеціально для платформи .NET. У C# представлені найважливіші аспекти розробок у галузі обчислювальних систем: об'єктно-орієнтоване програмування, обробка графічних об'єктів, компоненти графічного інтерфейсу користувача (GUI), обробка виключень, багатопотокова обробка, мультимедіа (аудіо, зображення, анімація і відео), обробка файлів, підготовлені структури даних, робота з базами даних, розробка багатоланкових програмних застосунків для Інтернету і WWW, організація мереж, Web-служби та розподілені обчислення.

Java SE забезпечує основні функціональні можливості мови програмування Java. Вона визначає все, від базових типів і об'єктів мови програмування Java до класів високого рівня, які використовуються для створення мереж, забезпечення безпеки, доступу до бази даних, розробки графічного інтерфейсу користувача (GUI) і аналізу XML.

На додаток до основного API платформа Java SE складається з віртуальної машини, засобів розробки, технологій розгортання та інших бібліотек класів і наборів інструментів, зазвичай використовуваних в додатках Java.

Java вивільняє міць об'єктно-орієнтованої розробки додатків, поєднуючи простий і знайомий синтаксис з надійним і зручним в роботі середовищем розробки. Це дозволяє широкому колу програмістів швидко створювати нові програми та нові аплети, надаючи багатий набір класів об'єктів для ясного абстрагування багатьох системних функцій, використовуваних у роботі з вікнами, мережею і для введення-виведення. Ключова риса цих класів полягає в тому, що вони забезпечують створення незалежних від використовуваної платформи абстракцій для широкого спектра системних інтерфейсів.

Отже, сучасному програмістові необхідно володіти знаннями щодо технології об'єктно-орієнтованого програмування, а також архітектури та алгоритмічних мов платформ Microsoft .NET і Java SE.

У посібнику систематизовано головні прийоми програмування мовами C# .NET і Java: опис типів даних, оголошення змінних і констант, організацію галужень та циклів, опис і використання структурованих типів даних, підпрограм і модулів. Розглянуто реалізацію у Visual C# .NET і Java SE об'єктно-орієнтованої парадигми програмування.

На достатньо детальному рівні описано об'єктно-орієнтовану семантику об'єктно-орієнтованих мов програмування C# і Java, поняття об'єктно-орієнтованого аналізу, проектування та програмування, основних елементів ООП.

Структурно навчальний посібник містить навчальний матеріал щодо основ Microsoft .NET і Java SE, основ об'єктно-орієнтованих мов програмування C# і Java, понять об'єктно-орієнтованого аналізу, проектування та програмування, основних елементів ООП. Розглянуто використання таких важливих засобів програмування, як оброблення винятків і бібліотеки класів, об'єктно-орієнтоване програмування застосувань із графічним інтерфейсом користувача.

Навчальний посібник сприяє якісному вивченню дисципліни "Основи об'єктно-орієнтованого програмування" та набуття студентами компетенцій з:

використання основних концепцій об'єктно-орієнтованого програмування у розробці програм на алгоритмічних мовах C# і Java;

використання головних можливостей бібліотеки .NET і Java SE в розробці застосунків і графічного інтерфейсу користувача.

Розділ 1. Основи об'єктно-орієнтованої парадигми

1. Основи Microsoft .NET та Java SE

Мета теми: набуття знань щодо особливостей розроблення програм для платформ Microsoft .NET та Java SE.

Професійні компетентності: уявляти особливості розроблення застосувань для платформ Microsoft .NET і Java SE**.**

Основні питання:

1.1. Програмна платформа Microsoft .NET.

1.2. Програмна платформа Java SE.

Основні питання для опрацювання: програмні платформи Microsoft .NET та Java SE, архітектура, компіляція та виконання програм, система типізації, стандартні бібліотеки класів, інструментальні засоби розроблення програм**.**

Ключові слова: Microsoft .NET, Common Language Runtime, Framework Class Library, managed code, модуль, Intermediate Language, just-in-time компіляція, віртуальна машина, Java Development Kit.

1.1. Програмна платформа Microsoft .NET

Технологія Microsoft .NET надає універсальні засоби створення розподілених програмних систем, що підтримують високий ступінь сумісності пристроїв, служб і комп'ютерів. Ключовим елементом цієї технології є *програмна платформа* .NET Framework – компонентна модель програмного забезпечення, яка дає змогу спільно використовувати окремі програмні модулі, створені різними мовами програмування, у вигляді єдиної функціональної системи.

.NET Framework сформовано із *середовища виконання коду* – Common Language Runtime (CLR) і *бібліотеки класів* .NET Framework Class Library (FCL), розташованої над CLR.

CLR – це загальне *середовище виконання* застосунків для всіх .NET-мов (або *віртуальна машина* – програмна система, яка завантажує програму, надає їй усі необхідні служби та виконує її).

З CLR зв'язане також поняття *керованого коду* (managed code), який виконується у .NET. Програма, написана NET-мовою, і ресурси, що викори-

стовуються нею, *керуються* винятково середовищем CLR – звідси і назва. Керований код зберігається у спеціальному двійковому файлі (*складеному модулі* – assembly). Зазвичай складений модуль і двійковий файл – синоніми, однак трапляються випадки, коли модуль містить декілька двійкових файлів.

Для візуального подання складеного модуля створено *проміжну мову* – Intermediate Language (IL), яку інколи називають MSIL.

Код, який виконується безпосередньо під Windows, називають *некерованим*. Деякі мови, наприклад C++, можуть працювати без керівного середовища CLR і не будуть керованими.

Середовище виконання коду

Середовище виконання коду – Common Language Runtime (CLR) абстрагує сервіси операційної системи та виконує функції середовища виконання для *керованих програм* (managed applications) – програм, кожна дія яких контролюється і підтверджується CLR.

CLR архітектурно стоїть над операційною системою і надає віртуальне середовище для виконання керованих програм. Коли така програма починає своє виконання, CLR завантажує в оперативну пам'ять комп'ютера модуль, що потрібно виконати, та виконує код, який він містить.

Інструкції керованого коду компілюються *за потребами* використання у машинний код під час виконання. Такий процес компіляції називають just-in-time (JIT) компіляцією. Здебільшого кожен метод компілюється лише тоді, коли його вперше активізують. Далі метод кешується в оперативній пам'яті для того, щоб його можна було використати повторно без компілювання. Отож код, який ніколи не активізується, не компілюється.

Середовище CLR містить множину типів, яку розділяють на *типи-значення* (value types) і *типи-посилання* (reference types). Базовим класом усіх типів є System.Object. *Типи-значення* (або *розмірні* типи) походять від базового типу System.ValueType. Їх розподіляють на три категорії: *вбудовані* типи, тип *перелічення* (Enum) і тип *користувача*. Тип System.ValueType є прямим нащадком System.Object. Будь-який тип, похідний від System.ValueType, є *структурою*, отож їх ще називають *структурними* типами.

Типи-посилання розподіляють на *об'єктні* типи (object types), *інтерфейсні* типи (interface types), *вказівні* типи (pointer types) і тип-посилання *користувача*. Об'єктний тип аналогічний *класу* (class) у багатьох об'єктно-орієнтованих мовах програмування. Усі типи-посилання є прямими нащадками класу System.Object.

Типи можуть містити члени (members), які можуть бути полями (fields) чи *методами* (methods). *Властивості* (properties) і події (events) є спеціальними типами методів. Поля і методи можуть належати всьому типу (type) або якомусь *екземпляру* (instance).

Доступ до поля чи виклик методу, які належать усьому типу, можна здійснити навіть тоді, коли цей тип не має жодного екземпляра. Такі члени типу (поля/методи) іноді називають *статичними*.

Доступ до поля екземпляра можна отримати, тільки вказуючи його екземпляр, а метод екземпляра можна викликати тільки з зазначенням його екземпляра.

Для розміщення значень змінних типу посилання пам'ять виділяється у *купі*, а змінних розмірних типів – у *стекові*. У ході присвоєння одного *розмірного* типу іншому присвоюється не сам тип (як осередок пам'яті), а його двійкове зображення. У разі присвоєння одного типу *посилання* іншому створюється ще один *покажчик*, який отримує адресу осередка пам'яті, що займає об'єкт.

Змінні типу посилання можна порівнювати за ознаками ідентичності та рівності. *Ідентичність* (identity) двох посилань означає, що вони містять покажчики (адреси) на один і той самий об'єкт, а рівність (equality) – на два різні об'єкти з однаковими даними.

Система виконання є частиною середовища CLR, яка відповідає за завантаження збірок, керування потоком виконання і збирання сміття в купі.

Вбудовані типи-значення

У табл. 1.1 перелічені вбудовані типи-значення CLR. У першому стовпці наведено назву типу проміжною мовою (IL). У наступному стовпці наведено назву цього типу в бібліотеці класів платформи NET (FCL). Останній стовпець табл. 1.1 містить дані про сумісність типів із *загальномовними специфікаціями* (CLS).

Таблиця 1.1

Убудовані типи-значення в середовищі CLR

Назва в FCL	Опис	Підтримка CLS
System.Boolean	Логічне значення (true / false	Так
System.Char	Символ Unicode	Так
System.SByte	8-бітове ціле число зі знаком	Ні
System.Int16	16-бітове ціле число зі знаком	Так
System.Int32	32-бітове ціле число зі знаком	Так
System.Int64	64-бітове ціле число зі знаком	Так
System.Byte	8-бітове ціле число без знака	Так
System.UInt16	16-бітове ціле число без знака	Ні
System.UInt32	32-бітове ціле число без знака	Ні
System.UInt64	64-бітове ціле число без знака	Ні
System. Single	32-бітове число з плаваючою комою	Так
System.Double	64-бітове число з плаваючою комою	Так
System.IntPtr	Машинне ціле число зі знаком, (32 або 64 біта залежно від процесора)	Так
System.UIntPtr	Машинне ціле число без знака	Ні

У створенні власних типів у багатомовному середовищі варто обмежуватися типами, сумісними з CLS. Тоді користувацькі класи, інтерфейси та структури працюватимуть без проблем з будь-якою мовою NET.

Бібліотека класів платформи .NET

Бібліотека класів Framework .NET – Framework Class Library (FCL) містить понад 7 000 типів (класів, структур, інтерфейсів, перелічених типів

і делегатів), які поділено між "просторами імен", кожен з яких відповідає за служби з певної області. Простори імен, які автоматично додаються до нового проекту, наведені в табл. 1.2.

Таблиця 1.2

Основні простори імен .NET

Простір імен	Опис
System	Містить основні типи та класи
System.Collections	Кешовані таблиці, динамічні масиви та інші контейнери
System.Collections.Generic	Містить інтерфейси та класи, що визначають універсальні колекції, які дозволяють користувачам створювати строго типізовані колекції
System.Linq	Містить класи й інтерфейси, які підтримують LINQ (Language-Integrated Query)
System.Text	Містить класи для кодування символів і для управління рядками. Дочірній простір імен дозволяє обробляти текст з використанням регулярних виразів
System.Threading.Tasks	Містить класи, які спрощують роботу з написання паралельного й асинхронного коду

Простори імен забезпечують ієрархію класів, отож допускають функціонування двох різних класів з однаковими назвами, які знаходяться у різних просторах назв. Отже, простір імен – це не що інше, як область дії класу.

Програміст, що використовує вбудовані класи, отримує доступ до простору імен через директиву:

using [aliasname =] namespace.element

де aliasname – ідентифікатор, за допомогою якого можна посилатися усередині модуля на зазначений простір імен;

namespace – назва імпортованого простору;

element – назва елемента простору (клас, простір імен тощо).

Кожен модуль може налічувати довільне число директив Imports, однак усі вони повинні розташовуватися до будь-якого посилання на ідентифікатори. Назва простору є частиною докладної назви об'єкта, що має загалом синтаксис namespace.typename. Усі простори імен, які постачає корпорація Microsoft, розпочинаються словом System або словом Microsoft.

Платформа Java SE

Платформа Java є набором системного програмного забезпечення і специфікацій, який надає інструменти для розроблення прикладних програм та їхнього розгортання в крос-платформному обчислювальному середовищі. Вона з'явилася ще у 1996 р. і широко використовується у багатьох апаратно-програмних системах, від вбудованих пристроїв і мобільних телефонів до корпоративних серверів і суперкомп'ютерів. Сьогодні платформа Java підтримується компанією Oracle.

Основою платформи Java є віртуальна машина. Вона виконує так званий "байт-код" програми, який є однаковим незалежно від типу обладнання та операційної системи, з якими використовується програма, та мови програмування, якою було створено її вихідний код.

Віртуальна Java-машина містить JIT-компілятор. Під час виконання програми він перетворює її байт-код у вбудовані інструкції процесора.

Використання байт-коду як проміжної мови дозволяє Java-програмам запускатися на будь-якій платформі, якщо для неї є віртуальна машина. Java-програми можуть працювати з такими поширеними операційними системами як Windows, Linux та Mac OS.

Існує декілька випусків платформи Java:

Java Micro Edition (Java ME) визначає кілька різних наборів бібліотек для пристроїв з обмеженими можливостями зберігання, відображення та живлення. Часто використовується для розроблення застосунків для мобільних пристроїв, телеприставок і принтерів;

Java Standard Edition (Java SE) використовується у застосунках загального призначення на настільних ПК, серверах і подібних пристроях;

Java Enterprise Edition EE (Java EE) призначено для використання в багатоланкових корпоративних застосунках клієнт-сервер і веб-застосунках.

1.2. Програмна платформа Java SE

Java SE є платформою для розроблення та розгортання застосунків для настільних і серверних середовищ. Вона визначає широкий спектр API загального призначення і включає специфікацію мови Java та специфікацію віртуальної машини Java. Найбільш відомими реалізаціями Java SE є Java Development Kit (JDK) корпорації Oracle і вільне програмне забезпечення OpenJDK.

Із самого початку свого існування Java SE використовувала об'єктно-орієнтовану мову програмування Java як свою єдину цільову мову. Тобто програми, сумісні з платформою Java, можна було розробляти тільки мовою Java. Тому вони зазвичай вважалися єдиним модулем.

Однак багато сучасних мов програмування підтримують кілька парадигм програмування і зазвичай виходять за рамки терміну "платформа". Згодом специфікацію Java було перероблено щоб більш чітко розмежувати мову Java та віртуальну машину Java як окремі об'єкти. Це дозволяє незалежно вдосконалювати Java-платформу та цільові мови програмування. Сьогодні, крім Java, є кілька десятків таких мов програмування, наприклад: Clojure, Groovy, Scala, Kotlin, Jython та багато інших.

Платформа Java зазнала численних змін з моменту виходу JDK 1.0 у 1996 р. Зокрема, у Java SE 8, що була випущена 18.03.2014 р., з'явились зміни багато в чому більш глибокі, ніж будь-які інші в історії платформи Java. Вони значно полегшили працю розробника за рахунок того, що написання коду значно скоротилось. Так, з'явилися лямбда-вираз, посилання на методи, методи за замовчуванням, Stream API – для роботи з колекціями, технологія JavaFX для розроблення графічних інтерфейсів користувача та інше.

Поточною версією платформи Java є Java SE 12, випущена 19.03.2019 р.

Типи даних Java SE

Java SE містить **примітивні** та **посилальні типи даних**. Основна відмінність між ними – це спосіб зберігання їх значень в пам'яті.

Примітивні типи наведено в табл. 1.3.

Таблиця 1.3

Примітивні типи даних Java

Ім'я	Розрядність, байт	Діапазон значень
1	2	3
char	2	0…65536
byte	1	–128…+127
short	2	–32768…+32767
int	4	$-2^{31}\ldots+2^{31}-1$

Закінчення табл. 1.3

1	2	3
long	8	$-2^{63}\ldots+2^{63}-1$
float	4	$3{,}4\text{x}10^{-38}\ldots3{,}4\text{x}10^{38}$
double	8	$1{,}7\text{x}10^{-308}\ldots1{,}7\text{x}10^{308}$
boolean	1	true; false

Вбудовані посилальні типи наведено в табл. 1.4.

Таблиця 1.4

Вбудовані посилальні типи даних Java

Ім'я	Опис
java.lang.Object	Базовий тип для всіх посилальних типів
java.lang.String	Послідовність символів (рядок)

Примітивні типи зберігають своє фактичне значення в статичній області пам'яті (стек). Посилальні типи зберігають в стеці лише адресу змінної, а сама змінна зберігається в динамічній області пам'яті.

Бібліотека класів платформи Java SE

Бібліотека класів платформи Java SE містить попередньо визначені класі, що включені до JDK, які можна використовувати у розробленні програм. У ній є компоненти для керування введенням і виведенням даних, підтримки математичних функцій, програмування баз даних тощо.

Протягом тривалого часу бібліотека була поділена на пакети та класи. Починаючі з Java 9, вона також містить модулі, **Модуль** – це група логічно зв'язаних пакетів, ім'я якої унікальне. Модулі дозволяють використовувати у розроблюваних застосунках тільки підмножину платформи Java. Пакети використовують для логічного групування споріднених класів аналогічно просторам імен .NET Framework. Його можна уявити як папку в файловій системі.

Перелік основних пакетів модуля java.base, що часто використовуються у Java-програмах, наведено в табл. 1.5.

Таблиця 1.5

Головні пакети модуля java.base

java.lang	Класи, якими оперує практично кожна програма
java.io	Введення/виведення даних з використанням потоків
java.util	Класи колекцій, парсинга рядків, генерування випадкових чисел
java.nio	Введення/виведення даних з використанням буферів, каналів і кодування
java.time	Класи для роботи з датами та часом

Пакети допомагають уникнути конфліктів імен класів, тому що одному пакету можуть належати тільки класи з унікальними іменами.

Інструментальні засоби розроблення програм

Хоча вихідний код Java-програми можна створювати у будь-якому текстовому редакторі, процес її розроблення містить такі етапи, як налагодження, компіляція та, власне, виконання програми. Їх можна виконувати за допомогою відповідних утиліт командного рядка, що входять до складу JDK. Це досить трудомісткій процес, тому, як правило, використовують одне з інтегрованих середовищ програмування. Найбільш популярними з них є JetBrains Idea, Eclipse та NetBeans.

Питання для самопідготовки

1. Призначення віртуальної машини й основних стандартних бібліотек класів.

Запитання для самодіагностики

1. Для чого призначена віртуальна машина?
2. Як створюють проект консольного застосування в середовищі програмування?
3. Як відкрити проект у середовищі програмування?
4. Як додати файл до проекту в середовищі програмування?
5. Як виконати компіляцію та запуск програми на виконання в середовищі програмування?
6. Як виконати налагодження програми в середовищі програмування?

Література: [1; 2; 4; 9; 11].

2. Основи об'єктно-орієнтованої мови програмування

Мета теми: набуття знань щодо системи типізації, операцій, операторів, структури програми, базових елементів стандартної бібліотеки класів, синтаксису й особливостей використання масивів, синтаксису сигнатури та виклику методів.

Професійні компетентності: розробляти найпростіші програми мовою програмування Java, C#.

Основні питання:

2.1. Загальні відомості про мови C# і Java: алфавіт, типи даних, операції, оператори, структура програми, основи використання стандартних бібліотек класів Microsoft .NET і Java SE.

2.2. Одновимірні та багатовимірні масиви у C# і Java: створення, ініціалізація, оброблення, підтримка масивів у стандартних бібліотеках Microsoft .NET і Java SE.

2.3. Методи у C# і Java: визначення, механізм передавання параметрів, використання масиву як параметра, повертання масиву з методу, виклик методу.

Ключові слова: алфавіт, типи даних, операції, оператори, структура програми, бібліотека класів, байт-код, простори імен, масив, метод, параметри.

2.1. Загальні відомості про мови C# і Java: алфавіт, типи даних, операції, оператори, структура програми, основи використання стандартних бібліотек класів Microsoft .NET і Java SE

C# – це об'єктно-орієнтована мова програмування загального призначення, яка вперше з'явилася в 2000 р. як частина платформи Microsoft .NET. Вона була розроблена для спільної мовної інфраструктури (CLI) – відкритої специфікації, розробленої компанією Microsoft. C#-програми компілюються в байт-код, який може виконуватися на платформах, які реалізують CLI.

Java також є об'єктно-орієнтованою мовою програмування. Вона розроблена компанією Sun Microsystems у 1995 р. Мета мови Java: пишіть один раз, запускайте в будь-якому місці. Java-застосунки компілюються в байт-код, який може працювати на реалізаціях віртуальної машини Java (JVM).

Витоки як Java, так і C# тісно пов'язані з переходом від мов програмування типу C++ до більш високого рівня. Тепер вихідний код програми компілюються в байт-код, який можна запускати на віртуальній машині. Це надає низку переваг, найбільш важливою з яких є можливість написання коду, зрозумілого людині. Він може виконуватися на будь-якій апаратній архітектурі, на якій встановлена віртуальна машина.

C# і Java мають багато спільного:

безпечність типів даних. Коли деякий тип даних помилково призначається змінній іншому типу, можуть з'явитися непередбачувані побічні ефекти. У C# і Java несумісні перетворення типів даних виявляються здебільшого під час компіляції, у ході виконання така ситуація призводить до генерування винятку;

збирання "сміття". У традиційних мовах програмування керування пам'яттю може бути втомливим, тому що програмісту потрібно пам'ятати, про необхідність "правильно" видаляти об'єкти. У C# і Java використовується автоматичне збирання "сміття". Це допомагає запобігти витоку пам'яті, оскільки видаляються об'єкти, які більше не використовуються програмою;

одиночне успадкування. І в C#, і в Java будь-який клас може мати тільки одного предка. Це обмежує ненавмисні побічні ефекти, які можуть виникнути, коли існує кілька зв'язків між декількома базовими класами та похідними класами;

інтерфейси. Інтерфейс – це сутність, що містить абстрактні методи. Це методи, які не мають реалізації. Код таких методів повинні надавати класи, що реалізують інтерфейс. Результатом є чиста, лінійна ієрархія класів одного успадкування в поєднанні з деякою універсальністю багаторазового успадкування.

Хоча C# і Java мають багато спільного, мова, яку вибирає користувач, здебільшого залежить від цільової платформи проекту.

Сьогодні C# використовується головним чином у .NET Framework, Mono та Portable.NET. Якщо розроблюваний застосунок призначений для використання з операційною системою Microsoft Windows, то C# є найкращим вибором. Слід нагадати, що, якщо необхідно розробляти застосунки для Unix, Linux, Mac OS або інших платформ за межами платформи Microsoft, велика "екосистема" Java є доцільнішою. Сьогодні у світі існує біля дев'яті мільйонів Java-розробників. Java-спільнота постійно створює нові бібліотеки й інструменти. З'явилися також нові потужні мови (такі, як Scala,

Clojure і Groovy), які базуються на JVM. Більшість реалізацій JVM є відкритими та безкоштовними. Java також є основною мовою, яку компанія Google використовує в розробленні для Android – найбільш розповсюдженої мобільної операційної системи.

Алфавіт Java і C#

Оскільки ці мови є C-подібними, їх алфавіт аналогічний мовам C/C++. Він містить таки елементи:

1) літери (a-z; A-Z);
2) цифри (0 - 9);
3) спеціальні символи (~, %, *, (,), -, + тощо);
4) керівні символи (\n, \t тощо).

Основні типи даних Java і C#

Основні вбудовані типи мови C# подані в табл. 2.1.

Таблиця 2.1

Основні вбудовані типи мови C#

Назва типу C#	Назва типу .NET	Опис типу	Розмір у бітах	Діапазон значень
1	2	3	4	5
bool	Boolean	Логічний	8	false, true
byte	Byte	8-розрядний цілий без знаку	8	0 … 255
sbyte	SByte	8-розрядний знаковий цілий	8	–128 … +127
short	Int16	Короткий знаковий цілий	16	–32768 … +32787
ushort	UInt16	Короткий цілий без знаку	16	0 … 65535
int	Int32	Знаковий цілий	32	–2 147 483 648 +2 147 483 647
uint	UInt32	Цілий без знаку	32	0 +4 294 967 295
long	Int64	Довгий знаковий цілий	64	–9 223 372 036 854 775 808 +9 223 372 036 854 775 807
ulong	UInt64	Довгий цілий без знаку	64	0 +18 448 744 073 709 551 615

Закінчення табл. 2.1

1	2	3	4	5
float	Single	Дійсний	32	1,401298E-45 3,402823E+38
double	Double	Дійсний подвоєної точності	64	E-324 E+308
decimal	Decimal	Числовий для фінансових розрахунків	96	29 значущих розрядів
char	Char	Символьний	16	
string	String	Набір символів Unicode		

Типи даних мови Java відповідають типам платформи Java SE, наведеним у табл. 1.3, 1.4.

Мова C# – це мова статичної типізації. Це означає, що кожний об'єкт у програмі має належати до одного з визначених типів. Усі типи даних, що використовуються в C#, розподіляють на дві категорії: типи-значення (***value types***) і типи-посилання (***reference types***).

Визначення та ініціалізація змінних, область їх видимості

Для того щоб визначити змінну одного із стандартних типів C# або Java, досить указати її тип та ідентифікатор:

```
double f;
char c;
```

Можлива її ініціалізація в момент визначення константним значенням або значення виразу:

```
int i = 0;
bool b = true;(в Java – boolean b = true;)
```

Змінні можна визначати всередині будь-якого блоку, тобто в області програми, обмеженої парою фігурних дужок. Такі змінні створюються, коли виконання програми доходить до цього блоку, та зникають, коли блок виконаний.

Локальні змінні в програмі на мовах C# і Java (починаючи з Java 10) можна оголошувати без указівки конкретного типу. Ключове слово var указує, що компілятор повинен вивести тип змінної з виразу праворуч від

оператора ініціалізації. Виведений тип може бути: стандартним; анонімним; визначеним користувачем; визначеним в бібліотеці класів .NET Framework або Java SE.

Таблиця 2.2

Основні операції Java та C#

Арифметичні	+ - * / %
Логічні	&& || !
Відношення	== != < > <= >=
Конкатенація рядків	+
Інкремент, декремент	++ --
Присвоювання	= += -= *= /= %=
Індексація	[ ]
Створення об'єкта в динамічній пам'яті	new

Приклад оголошення змінних з використанням ключового слова var:

```
var x = 10;
var s = "Привіт";
var y = 4.5;
```

Змінна x отримає тип int, змінна s – тип string (String у Java), а змінна y – тип double.

Структура консольної програми

Мова програмування C# є об'єктно-орієнтованою, тому навіть найпростіша програма повинна мати як мінімум один клас, в якому має містити метод Main. Метод Main є "точкою входу в програму", оскільки саме з нього починається виконання будь-якої програми.

Саме тому під час створення нової консольної програми середовище програмування Visual Studio 2017 автоматично генерує такий програмний код:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

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

Перші п'ять рядків підключають до програми найбільш широко вживані простори імен, призначення яких було наведено в табл. 1.2. Не всі простори імен, які автоматично підключилися, повинні використовуватися в програмі. Для простих програм достатньо лише простору імен System. З іншого боку, якщо в програмі потрібно використовувати інші простори імен, достатньо додати цей простір, указавши його назву після using.

Далі необхідне оголошення нового простору імен FirstProgram, назву якого програміст указує для створення нової програми. Межі простору імен визначаються відповідними відкривальною та закривальною фігурними дужками.

Далі створюється оголошення нового класу з назвою Program, в якому оголошується єдиний поки що метод Main. Межі класу Program і межі методу Main також визначаються відповідними відкривальною та закривальною фігурними дужками.

У подальшому програміст може наповнювати метод Main програмним кодом, використовуючи конструкції мови C#, оголошувати нові методи в класі Program, нові класи тощо.

Структура консольної програми мовою Java є аналогічною C# з певними відмінностями:

```
package my;
import ……..;
……….
class Program {
      ……
      public static void main(string[ ] args) {
      ……..
      }
}
```

Тут package my – це пакет, до якого належить наведений код (my – ім'я пакета). **Пакет** – це аналог простіру імен у C++ та C#.

Інструкція import – призначена для імпорту необхідних бібліотек. Це аналогічно використанню using у C#. Їх може не бути взагалі, або файл вихідного коду може містити декілька інструкцій import.

Формат контенту інструкції import для підключення стандартних бібліотек має декілька варіантів, наприклад:

import java.io.*; – імпорт усіх елементів пакету вводу – виводу даних java.io

import java.io.File; – імпорт класу File з пакету java.io.

Оскільки Java – об'єктно-орієнтована мова програмування, в будь-якій програмі потрібен хоча б один клас. Тут клас Program містить метод main. **Метод main** – точка входу до програми. На відміну від C#, сигнатура цього методу завжди однакова.

Введення-виведення даних у консольній програмі

Консольна програма працює в консольному вікні, яке дозволяє здійснювати введення – виведення тексту за допомогою клавіатури. Простір імен System містить клас Console, в якому є методи, що забезпечують виведення текстової інформації у вікно консолі та зчитування тексту, який користувач програми вводить за допомогою клавіатури.

Для виведення тексту в консольне вікно використовують методи Write та WriteLine класу Console. Обидва методи виводять у консольне вікно рядок тексту, який передається як параметр методу та записується в круглих дужках після назви методу. У мові програмування C# текстом чи рядком є послідовність будь-яких символів, обмежених подвійними лапками. Навіть один символ, обмежений подвійними лапками, також є радком тексту.

Приклади рядків:

"Це рядок тексту"

"Рядок може містити цифри 123 та інші символи !№;% "

"a"

Щоб програма вивела в консольне вікно будь-який текст (наприклад, "C# – це мова програмування"), потрібно в методі Main написати такий програмний код:

```
class Program
  {
```

```
    static void Main(string[] args)
    {
       Console.WriteLine("C# - це мова програмування");
    }
  }
```

Різниця між методами Write та WriteLine полягає у тому, що метод Write виводить рядок у вікно та залишає курсор у поточній позиції. Тому наступний текст буде виведено у цьому ж рядку відразу після попереднього виведеного тексту. А метод WriteLine виводить текст у вікно й автоматично переводить курсор на наступний новий рядок.

Наведемо приклади використання цих методів і результати їх роботи.

Така програма

```
class Program
  {
    static void Main(string[] args)
    {
       Console.WriteLine("Текстовий рядок 1 ");
       Console.WriteLine("Текстовий рядок 2 ");
       Console.WriteLine("Текстовий рядок 3 ");
    }
  }
```

виведе у вікно такий текст:

Текстовий рядок 1
Текстовий рядок 2
Текстовий рядок 3

А така програма:

```
class Program
  {
    static void Main(string[] args)
    {
       Console.Write("Текстовий рядок 1 ");
       Console.Write("Текстовий рядок 2 ");
       Console.Write("Текстовий рядок 3 ");
    }
  }
```

виведе у вікно такий текст:

Текстовий рядок 1 Текстовий рядок 2 Текстовий рядок 3

Вводячи рядки, їх можна поєднувати один з одним або з числами за допомогою знаку "+". Така операція має назву **"конкатенація"**.

Наприклад, результатом роботи такої програми:

```
class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Харківський національний " + "економічний університет");
       Console.WriteLine("Студент Петренко отримав "+100+ " балів");
    }
  }
```

буде такий текст:

Харківський національний економічний університет
Студент Петренко отримав 100 балів

Введення або зчитування рядка в консольному вікні здійснюється за допомогою метода ReadLine класу Console. Результатом роботи цього методу буде рядковий тип даних, тобто string. Для того щоб введений рядок можна було використовувати в програму, потрібно оголосити змінну типу string і присвоїти їй рядок, який буде Введено методом ReadLine. Наприклад:

```
class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Як твоє ім'я ?");
      string name = Console.ReadLine();
      Console.WriteLine("Привіт, " + name + "!");
    }
  }
```

Якщо у відповідь на запитання про ім'я ввести, наприклад, рядок "Петро", то у вікні з'явиться текст "Привіт, Петро!".

Треба зазначити, що метод ReadLine завжди повертає результат типу "рядок", навіть коли користувач програми вводить число або один символ. У цьому разі потрібно виконати приведення рядка до відповідного

типу. Для цього в мові програмування C# є дві можливості. Перша – використання відповідних методів класу Convert. Друга – використання методу Parse, який має усі стандартні типи.

Наприклад, ціле число можна ввести так:

```
int x = Convert.ToInt32(Console.ReadLine());
```

або так:

```
int y = int.Parse(Console.ReadLine());
```

Таким самим чином можна ввести дані інших стандартних типів.

Для виводу на консоль можливо організувати форматування виводу, тобто отримати значення на екрані у зручному для сприйняття вигляді. Для цього у текстову константу – аргумент методу Console.WriteLine – треба помістити так звані **плейсхолдери** (число у фігурних дужках) з номером потрібного елемента списку виводу, причому нумерація починається з нуля. На місці кожного з таких плейсхолдерів на екрані з'явиться відповідне значення.

Наприклад, результатом роботи такої програми:

```
class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Студент Хоменко отримав {0} балів",90);
    }
  }
```

буде такий текст:

```
Студент Хоменко отримав 90 балів
```

На місце плейсходера може підставлятися значення константи, змінної або виразу.

```
class Program
  {
    static void Main(string[] args)
    {
      int x = 10;
```

```
        int y = 20;
        Console.WriteLine("{0} + {1} = {2}",x,y,x+y);
    }
}
```

Результат:
10 + 20 = 30

Більше того, можливе використання форматів виводу. Загальний вигляд плейсхолдеру у цьому випадку такий: {n:fk}. Тут n означає порядковий номер елемента у списку виводу (нумерація починається з нуля), f – це символ специфікації формату, число k задає точність. Основні символи специфікацій формату такі:

F або f – для виводу дійсних значень у форматі з фіксованою точністю;

E або e – для виводу дійсних значень у експоненціальному форматі;

G або g – загальний формат для виводу дійсних значень або у форматі з фіксованою точністю або у експоненціальному форматі;

N або n – формат для виводу дійсних значень з відокремленням трійок розрядів пробілами;

C або c – грошовий формат;

X або x – шістнадцятковий формат для виводу цілих типів.

Наступна програма виводить дійсне число 123,456 без форматування, з двома десятковими знаками та в експоненціальному форматі.

```
class Program
{
    static void Main(string[] args)
    {
        double x = 123.456;
        Console.WriteLine("{0}  {0:F2}  {0:E4}",x);
    }
}
```

Результат:
123,456 123,46 1,2346E + 002

Починаючи з версії мови C# 6.0, з'явився ще один спосіб виведення даних, який передбачає безпосередньо у плейсхолдері вказувати ім'я змінної, значення якої буде виводитися. Але для цього обов'язково потрібно вказати символ $ перед рядком форматування. Усі три варіанти виведення наведено у такій програмі.

```
class Program
  {
    static void Main(string[] args)
    {
      int x = 10;
      double y = 12.5;
      Console.WriteLine("x = " + x + "  y = " + y);
      Console.WriteLine("x = {0}  y = {1}", x, y);
      Console.WriteLine($"x = {x}  y = {y}");
    }
  }
```

Результат:
x = 10 y = 12,5
x = 10 y = 12,5
x = 10 y = 12,5

Для введення даних у Java існує декілька способів.

Найпростіший із них заснований на використанні класу Scanner з пакету java.util. Наприклад:

```
import java.util.Scanner;
……….
Scanner sc = new Scanner(System.in);
String s = sc.next(); //Введення рядка до пробілу
double str = sc.nextDouble(); // Введення дійсного числа
int str = sc.nextInt(); // Введення цілого числа
```

Scanner має декілька методів, подібних до nextDouble. Вони повертають значення відповідного типа даних, яке було введено з консолі.

Для виводу даних у Java є методи print (println), format і printf.

Методи print і println зазвичай використовують для неформатованого виводу. Різниця між ними полягає в тому, що println після виведення рядка символів переводить курсор на новий рядок консолі.

Приклад фрагменту програми:

```
double x = 125.12;
System.out.println("Значення = " + x);
```

У результаті, на консоль буде виведена така символьна послідовність:
Значення = 125.5

Методи format і printf призначені для форматованого виводу даних. Існує багато форматів для форматування даних різних типів.

Наприклад, виведення дійсного числа з двома знаками в його дробовій частині може виглядати таким чином:

```
double x = 125.1234567;
System.out.format("Значення = %1$.2f ", x);
```

Результат:
Значення = 125,12

Основні синтаксичні елементи Java і C#

Як зазначалося, об'єктно-орієнтовані мови програмування Java та C# за своїм синтаксисом дуже схожі, оскільки кожна з них створювалася на основі синтаксису класичної мови програмування C/C ++. Відповідність конструкцій цих двох мов програмування наведена в табл. 2.3.

Таблиця 2.3

Відповідність конструкцій мов програмування C# і Java

Java	**C#**
1	2
Структура програми	
package hello; public class HelloWorld { public static void **main**(String[] args) { String name = "Java"; System.out.println("Hello, " + name + "!"); } }	using System; namespace Hello { public class HelloWorld { public static void **Main**(string[] args) { string name = "C#"; Console.WriteLine("Hello, " + name + "!"); } } }
Коментарі	
// Single line */* Multiple* *line */* */** Javadoc documentation comments */*	*// Single line* */* Multiple* *line */* */// XML comments on a single line* */** XML comments on multiple lines */*

1	2
Типи даних	
Primitive Types boolean byte char short, int, long float, double *Reference Types* Object *(superclass of all other classes)* String *arrays, classes, interfaces* *Conversions* *// int to String* int x = 123; String y = Integer.toString(x); *// y is "123"* *// String to int* y = "456"; x = Integer.parseInt(y); *// x is 456* *// double to int* double z = 3.5; x = **(int)** z; *// x is 3 (truncates decimal)*	*Value Types* bool byte, sbyte char short, ushort, int, uint, long, ulong float, double, decimal *structures, enumerations* *Reference Types* object *(superclass of all other classes)* string *arrays, classes, interfaces, delegates* *Convertions* *// int to string* int x = 123; String y = x.ToString(); *// y is "123"* *// string to int* y = "456"; x = int.Parse(y); *// or x =* *//Convert.ToInt32(y);* *// double to int* double z = 3.5; x = **(int)** z; *// x is 3 (truncates decimal)*
Константи	
// May be initialized in a constructor **final** double PI = 3.14;	**const** double PI = 3.14; *// Can be set to a const or a variable. May* *//be initialized in a constructor.* **readonly** int MAX_HEIGHT = 9;
Перерахування	
enum Action {Start, Stop, Rewind, Forward}; *// Special type of class* **enum** Status { Flunk(50), Pass(70), Excel(90); private final int value; Status(int value) { this.value = value; } public int value() { return value; } };	**enum** Action {Start, Stop, Rewind, Forward}; **enum** Status {Flunk = 50, Pass = 70, Excel = 90}; *No equivalent.*

Продовження табл. 2.3

1	2
Action a = Action.Stop; if (a != Action.Start) System.out.println(a); *// Prints "Stop"* Status s = Status.Pass; System.out.println(s.value()); *// Prints "70"*	Action a = Action.Stop; if (a != Action.Start) Console.WriteLine(a); *// Prints "Stop"* Status s = Status.Pass; Console.WriteLine((int) s); *// Prints "70"*
Оператори	
omparison == < > <= >= != *Arithmetic* + - * / % *(mod)* / *(integer division if both operands are ints)* Math.Pow(x, y) *Assignment* = += -= *= /= %= &= |= ^= <<= >>= >>>= ++ -- *Bitwise* & | ^ ~ << >> >>>> *Logical* && || & | ^ ! **Note:** && and || perform short-circuit logical evaluations *String Concatenation* +	*Comparison* == < > <= >= != *Arithmetic* + - * / % *(mod)* / *(integer division if both operands are ints)* Math.Pow(x, y) *Assignment* = += - = *= /= %= &= |= ^= <<= >>= ++ -- *Bitwise* & | ^ ~ << >> *Logical* && || & | ^ ! **Note:** && and || perform short-circuit logical evaluations *String Concatenation* +
Вибір	
greeting = age < 20 **?** "What's up?" **:** "Hello"; **if** (x < y) System.out.println("greater"); **if** (x != 100) { x *= 5; y *= 2; } **else** z *= 6; int selection = 2; **switch** (selection) { *// Must be byte,*	greeting = age < 20 **?** "What's up?" **:** "Hello"; **if** (x < y) Console.WriteLine("greater"); **if** (x != 100) { x *= 5; y *= 2; } **else** z *= 6; string color = "red";

Закінчення табл. 2.3

1	2
short, //int, char, or enum **case** 1: x++; *// Falls through to next* *//case if no break* **case** 2: y++; **break;** **case** 3: z++; **break;** **default:** other++; }	**switch** (color) { *// Can be* *//any predefined type* **case** "red": r++; **break;** *// break is* *//mandatory; no fall-through* **case** "blue": b++; **break;** **case** "green": g++; **break;** **default:** other++; **break;** *// break* *//necessary on default* }
Цикли	
while (i < 10) i++; **for** (i = 2; i <= 10; i += 2) System.out.println(i); **do** i++; **while** (i < 10);	**while** (i < 10) i++; **for** (i = 2; i <= 10; i += 2) Console.WriteLine(i); **do** i++; **while** (i < 10);

Аналіз табл. 2.3 засвідчує однотипні підходи до синтаксису та семантики мов Java та C#.

2.2. Одновимірні та багатовимірні масиви у C# і Java: створення, ініціалізація, оброблення, підтримка масивів у стандартних бібліотеках Microsoft .NET і Java SE

Масиви є в обох мовах програмування і мають дуже багато спільного. Як в Java, так і в C#, **масив** – це обмежений набір даних одного типу, які мають одне спільне ім'я та розрізняються лише за індексом. За традицією, нумерація елементів масиву розпочинається з нуля.

У кожній з двох мов є стандартні класи для роботи з масивами. У Java – клас java.util.Arrays, а в C# – клас Array, які містять корисні методи для виконання типових дій з масивами. Наприклад, метод, який виконує упорядковування (сортувння) елементів масиву за зростанням.

Але є і певні відмінності. У Java можуть бути оголошені тільки одномірні масиви. Багатовимірний масив в Java – масив масивів. У C# є як справжні багатовимірні масиви, так і масиви масивів, які в C# зазвичай називають "нерівними", або "ступінчастими" (jagged). Багатовимірні масиви завжди

"прямокутні" (за двовимірною термінологією), тоді як масиви масивів можуть зберігати рядки різної довжини (знову-таки в двовимірному випадку, в багатовимірному – аналогічно). Багатовимірні масиви прискорюють доступ до пам'яті; нерівні масиви працюють повільніше, проте економлять пам'ять, коли не всі рядки заповнені. Багатовимірні масиви вимагають для свого створення лише одного виклику оператора new, а ступінчасті – явно виділяти пам'ять у циклі для кожного рядка.

Порівняння застосування масивів у мовах Java та C#, а також особливості їх використання наведені в табл. 2.4.

Таблиця 2.4

Порівняння застосування масивів у мовах Java та C#

Java	Масиви	C#
int nums[ ] = {1, 2, 3}; *or* int[ ] nums = {1, 2, 3}; for (int i = 0; i < nums.length; i++) System.out.println(nums[i]); String names[ ] = new String[5]; names[0] = "David"; for (int i : nums) *// foreach construct* sum += i;		int[ ] nums = {1, 2, 3}; for (int i = 0; i < nums.Length; i++) Console.WriteLine(nums[i]); string[ ] names = new string[5]; names[0] = "David"; foreach (int i in numArray) sum += i;
int twoD[ ][ ] = new int[4][5]; twoD[2][0] = 3; int[ ][ ] jagged = new int[3][ ]; jagged[0] = new int[5]; jagged[1] = new int[2]; jagged[2] = new int[3]; jagged[0][4] = 5; int[ ] numbers = {1, 23, 3, 8, 2, 4, 4}; // sorting Arrays.sort(numbers);		int [,] twoD = new int[4, 5]; twoD[2,0] = 3; int[ ][ ] jagged = new int[3][ ] { new int[5], new int[2], new int[3] }; jagged[0][4] = 5; int[ ] numbers = {1, 23, 3, 8, 2, 4, 4}; // sorting Array.Sort(numbers);

Також у кожній мові є оператор циклу, який дозволяє почергово перебирати всі елементи масиву й отримувати на кожній ітерації циклу значення поточного елемента.

2.3. Методи у C# і Java: визначення, механізми передавання параметрів, використання масиву як параметра, повертання масиву з методу, виклик методу

Якщо **змінні** в програмі зберігають дані, то методи містять оператори, які виконують певні дії (наприклад, обробку даних у відповідності до деякого алгоритму). Можна сказати, що **метод** – це іменований блок коду, призначений для виконання тієї чи іншої дії. Метод можна виконати в будь-якому місці програми в області видимості, вказавши ім'я методу. Метод можна виконувати кілька разів. Під час роботи програми в метод можна передавати дані й отримувати результат роботи методу.

У загальному вигляді оголошення методу виглядає так:

```
[модифікатори] тип_результату назва_методу([параметри])
{
  // тіло методу
}
```

Модифікатори та параметри є необов'язковими.

Метод може бути статичним, тобто мати модифікатор static.

У такому випадку метод можна викликати безпосередньо в класі без створення об'єкта класу.

Типом результату, який повертає метод, може бути будь-який стандартний тип даних або попередньо оголошений в програмі розробником. Якщо у якості типу даних використовується службове слово void, це означає, що метод не повертає значення.

Після назви методу в круглих дужках перераховуються параметри. Кожний параметр має тип та ім'я. Параметри відділяються один від одного комами. У списку параметрів можуть бути параметри різних типів. Приклади заголовків методів:

```
int Summa(int x, int y)
void Print (string s)
double Sqr(double a, double b, int h)
```

Круглі дужки після назви методу необхідно писати, навіть якщо метод не має параметрів. У тілі методу, яке в фігурних дужках розташовують відразу після заголовку методу, можна оголошувати локальні змінні методу, використовувати будь-які алгоритмічні конструкції – умовні

оператори, цикли та викликати інші методи. Якщо метод повертає значення (тобто метод має тип, відмінний від void), у тілі методу обов'язково повинна бути інструкція return з виразом, тип якого відповідає типу значення, що повертається.

Далі розглянемо використання методів у мові C#. У Java ці операції виконуються подібним чином.

Наведемо приклад двох методів, які обчислюють суму двох цілих чисел. Перший метод повертає результат типу int на місце виклику цього методу; і цей результат може бути присвоєний змінній відповідного типу. Другий метод відразу виводить результат у консольне вікно. Обидва методи викликаються в методі Main, але по-різному.

```
class Program
  {
    static int Summa1(int x, int y)
    {
      return x+y;
    }
    static void Summa2(int x, int y)
    {
      Console.WriteLine(x+y);
    }
    static void Main(string[] args)
    {
      int a = 10;
      int b = 5;
      int c = Summa1(a, b);
      Console.WriteLine(c);

      Summa2(a, b);
    }
  }
```

Є два способи передавання в метод параметрів – за значенням і за посиланням. У передаванні параметру за значенням метод отримує не саму змінну, а її копію. А з передаванням за посиланням метод отримує адресу змінної в пам'яті. Щоб передати параметр за посиланням, потрібно в списку параметрів указати перед параметром ключове слово ref. І якщо в методі змінюється значення параметра, який передається за посиланням, то змінюється і значення змінної, яка передається на його місце.

Розглянемо два приклади. У першому параметр у метод передається за значенням.

```
class Program
{
   // передавання за значенням
   static void IncrementVal(int x)
   {
      x++;
      Console.WriteLine($"Значення: {x}");
   }
   static void Main(string[] args)
   {
      int a = 5;
      Console.WriteLine($"Початкове значення змінної a = {a}");
      IncrementVal(a);
      Console.WriteLine($"Змінна a після виконання методу = {a}");
      Console.ReadKey();
   }
```

Результат:

1) початкове значення змінної a = 5;
2) значення : 6;
3) змінна a після виконання методу = 5.

Як бачимо, значення змінної "a" після виконання методу IncrementVal(a) не змінилося: в метод було передано її копію.

Повторимо ще раз, але тепер передамо змінну в метод за посиланням.

```
class Program
{
   // передавання за значенням
   static void IncrementVal(ref int x)
   {
      x++;
      Console.WriteLine($"Значення: {x}");
   }
   static void Main(string[] args)
   {
      int a = 5;
      Console.WriteLine($"Початкове значення змінної a = {a}");
```

```
        IncrementVal(ref a);
        Console.WriteLine($"Змінна a після виконання методу = {a}");
        Console.ReadKey();
    }
}
```

Результат:

1) початкове значення змінної a = 5;
2) значення : 6;
3) змінна a після виконання методу = 6.

У другому прикладі змінна "a" після виконання методу IncrementVal(a) змінила своє значення. Слід зазначити, що перед тим як передавати в метод змінну за посиланням, обов'язково потрібно присвоїти цій змінній значення.

Крім вхідних параметрів у метод можна передавати вихідні параметри (в Java немає такої можливості). Щоб параметр став вихідним, потрібно перед ним вказати модифікатор out. Використання вихідних параметрів розглянемо на прикладі методу, який обчислює площу та периметр прямокутника зі сторонами a та b.

```
class Program
    {
        static void Calc(int x, int y, out int area, out int perim)
        {
            area = x * y;
            perim = (x + y) * 2;
        }
        static void Main(string[] args)
        {
            int a = 10, b = 15;
            int area;
            int perimetr;
            Calc(a, b, out area, out perimetr);
            Console.WriteLine("Площа :  {0}", area);
            Console.WriteLine("Периметр : {0}", perimetr);
        }
    }
```

У C# 7.0 (Visual Studio 2017) можна визначати змінні безпосередньо під час виклику метода. У цьому випадку метод Main можна записати так:

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

```
    int a = 10, b = 15;
    Calc(a, b, out int area, out int perimetr);
    Console.WriteLine("Площа :  {0}", area);
    Console.WriteLine("Периметр : {0}", perimetr);
}
```

C# дозволяє використовувати необов'язкові параметри. Для таких параметрів під час оголошення методу необхідно вказати значення. Але якщо з оголошенням методу вказано необов'язковий параметр, то всі наступні параметри також повинні бути необов'язковими. Наприклад:

```
class Program
{
    static int Summa(int x, int y = 2, int z = 3)
    {
        return x + y + z;
    }
    static void Main(string[] args)
    {
        int sum1 = Summa (10);
        Console.WriteLine(sum1);
        int sum2 = Summa (10,15);
        Console.WriteLine(sum2);
        int sum3 = Summa (10, 20, 30);
        Console.WriteLine(sum3);
    }
}
```

Оскільки два останніх параметра були оголошені як необов'язкові, то можна один з них або обидва не передавати в метод. Тоді будуть використовуватися значення, які вказані в списку параметрів під час оголошення методу.

Не завжди можна наперед знати, скільки параметрів буде передаватися в метод. У такому випадку доцільно використовувати масив параметрів. Щоб оголосити масив параметрів, потрібно вказати модифікатор params. Наприклад:

```
class Program
    {
        static void Summa(params int[] integers)   {
            int result = 0;
            foreach (int x in integers)
```

```
            result += x;
        Console.WriteLine(result);
    }
    static void Main(string[] args)   {
        Summa(5, 2, 3, 1, 4);
        Summa(2, 3);
        Summa();
        int[] array = { 3, 2, 1, 4 };
        Summa(array);
    }
}
```

Результат: 15; 5; 0; 10.

У наведеному прикладі параметр методу Summa integers є масивом параметрів.

Питання для самопідготовки

1. Використання коментарів.
2. Налагодження програм у середовищі програмування.

Запитання для самодіагностики

1. Для чого призначено головний метод програми?
2. Для чого призначено посилання this та як його використовувати?
3. Які існують способи передавання параметрів методів?
4. Охарактеризуйте структуру програми на мовах C# і Java.
5. Перелічіть вбудовані типи даних C# і Java.
6. Яке призначення мають методи класу Math?

Література: [9; 11; 19; 20].

3. Поняття об'єктно-орієнтованого аналізу, проектування та програмування

Мета теми: набуття знань щодо змісту основних етапів об'єктно-орієнтованої технології.

Професійні компетентності: здатність до об'єктно-орієнтованого мислення.

Основні питання:

3.1. Об'єктно-орієнтована декомпозиція. Принципи об'єктно-орієнтованого підходу: абстракція, інкапсуляція, ієрархія, поліморфізм.

3.2. Поняття об'єкта. Характеристики об'єкта. Поняття класу. Співвідношення між класом та його об'єктом.

3.3. Об'єктно-орієнтований аналіз та його мета. Головні види вимог до програмної системи. Об'єктно-орієнтоване проектування. Об'єктно-орієнтоване програмування.

3.4. UML-діаграми класів. Відношення на діаграмі класів. CASE-засоби.

Основні питання для опрацювання: об'єктно-орієнтована декомпозиція, принципи об'єктно-орієнтованого підходу, поняття об'єкта та класа, об'єктно-орієнтований аналіз, об'єктно-орієнтоване проектування та програмування, UML-діаграми класів, CASE-засоби.

Ключові слова: абстракція, інкапсуляція, ієрархія, поліморфізм, об'єкт, клас, стан, об'єктно-орієнтований аналіз, UML, CASE-засоби.

3.1. Об'єктно-орієнтована декомпозиція. Принципи об'єктно-орієнтованого підходу: абстракція, інкапсуляція, ієрархія, поліморфізм

Програмна система означає програмне забезпечення, що розробляється. Це може бути крупна колекція з безлічі компонентів програмного забезпечення, одна програма або частина програми.

Очевидно, не всі програмні системи складні. Є багато програм, задуманих, розроблених, супроводжуваних і використовуваних однією людиною. Але вони, як правило, мають обмежену галузь застосування і коротку тривалість використання. Системи банківських розрахунків, резервування залізничних квитків, операційні системи та інші належать до індустріально-розробленого програмного забезпечення (ПЗ). Його створює і вдосконалює команда розробників. Воно використовується протягом тривалого часу та має багато користувачів. Суттєвою рисою такого ПЗ є його велика складність: одному розробникові практично неможливо охопити всі тонкощі системи. Таким чином, можна зробити висновок – складність притаманна програмному забезпеченню.

Можна виділити чотири основні причини складності ПЗ:

складність проблеми предметної області. Прикладні проблеми часто містять складні елементи, до яких ставиться багато різних, часто

суперечливих вимог. Ситуація погіршується тим, що розробники ПЗ можуть мати недостатню кваліфікацію в галузі проблеми, а замовники часто не до кінця уявляють, що, власне, їм потрібно. Додаткова складність зумовлюється зміною вимог до програми в процесі розроблення, оскільки вже наявність проекту системи змінює саму проблему та ступінь її розуміння. На жаль, розроблення не можна щоразу починати з самого початку, тому великі системи мають тенденцію до еволюції;

складність управління процесом розроблення. Головне завдання розробників полягає у створенні ілюзії простоти, яка захищатиме користувачів від складності процесів, які реалізує програма. Однак складність проблеми та велика кількість вимог до ПЗ вимагає колективної праці розробників. У цій ситуації важливим завданням є координація робіт і підтримання цілісності основної ідеї;

гнучкість програмного забезпечення. Програміст може самостійно забезпечити себе всіма необхідними елементами архітектури ПЗ, які належать до будь-якого рівня абстракції, починаючи з найнижчих. Ця спокуслива властивість та ще й відсутність єдиних стандартів на такі елементи часто призводить до того, що програми починають писати "з нуля". Тому розроблення ПЗ залишається дуже копіткою справою;

складність опису поведінки окремих підсистем. Програма для сучасного комп'ютера є системою зі скінченною кількістю дискретних станів, причому їх може бути дуже багато. Переходи системи з одного стану в інший не можна моделювати неперервними функціями, тому іноді (у випадку несприятливого збігу обставин) зміна стану однієї частини системи може призвести до катастрофічних змін у інших частинах. Адже всеохоплююче тестування великої програмної системи провести просто неможливо.

Прикладом складної системи може бути **персональний комп'ютер** (ПК). Основними складовими ПК є системний блок, монітор і клавіатура. Кожну з цих частин теж можна розділити на складові: наприклад, системний блок містить процесор, оперативну пам'ять, накопичувачі на магнітних дисках, блок живлення. Далі можна розглянути влаштування процесора і т. д. Так отримуємо *структурну ієрархію* ПК. Комп'ютер працює добре, коли злагоджено функціонують усі його частини. Кожна з частин є відносно незалежною від інших.

Складні системи є не просто ієрархічними: рівні ієрархії відображають різні рівні абстракції, що випливають один з одного, будучи деякою мірою автономними. Для кожної конкретної задачі ми розглядаємо відповідний

рівень. Наприклад, щоб підготувати за допомогою ПК деякий текст, достатньо лише уявляти загальну будову комп'ютера (найвищий рівень абстракції). Проте таких знань буде замало, щоб написати ефективну програму мовою асемблера.

Можна також навести приклад іншої ієрархії. Різні ПК бувають оснащені різними типами процесорів. Маючи спільну властивість виконувати певний набір команд, процесори бувають різної потужності, різних фірм-виробників тощо. У цьому випадку говорять про *ієрархію типів* процесорів.

Сучасне програмне забезпечення стає щораз складнішим. Зручний у використанні інтерфейс, розвинені можливості, керованість програми подіями, нові нетрадиційні сфери застосування – все це зумовлює ускладнення ПЗ. Можна сказати, що виготовлення ПЗ, зручного у використанні, робить його складним. Сьогодні, щоб побудувати сучасну програму, не достатньо просто об'єднати в послідовність певні машинні інструкції, оператори мови високого рівня чи навіть набори процедур і модулів. Головним стала проблема розроблення виразної структури програми, придатної до легкої модифікації, вільної від помилок, стійкої до змін. Як сказав Алан Кей, розробник мови програмування Smalltalk, зі зростанням складності архітектура домінує над матеріалами.

Об'єктно-орієнтована технологія створення ПЗ була задумана і розроблена як інструмент подолання складності. Вона успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування ПЗ. Головним завданням об'єктно-орієнтованого підходу є забезпечення способу структурування програми та керування складними взаємозв'язками між великою кількістю компонентів системи.

Об'єктно-орієнтоване структурування зменшує число зв'язків між компонентами. Воно заставляє об'єкти взаємодіяти через вузькі інтерфейси, що дає змогу легко ізолювати помилки та визначати, котрий з методів є відповідальним за помилку, що виникла.

Об'єкти захищають свої дані від несанкціонованого доступу. Оголошені *протоколи взаємодії* дозволяють компіляторові чи інтерпретаторові попереджувати користувача про несанкціонований доступ до даних і навіть не допустити його. Добре спроектовані інтерфейси класів дають змогу будувати добре модульовані, легко переміщувані компоненти програми.

Найбільшою зручністю технології є безпосереднє перетворення об'єктів прикладної області в об'єкти програми. Легше моделювати реальний світ у вигляді об'єктів, ніж проектувати його у вигляді процедур.

Об'єктно-орієнтована парадигма дозволяє полегшити повторне використання коду. Для того щоб мова програмування була об'єктно-орієнтованою, вона має використовувати класи й об'єкти, а також реалізовувати фундаментальні принципи об'єктно-орієнтованого підходу, головними з яких є абстракція, інкапсуляція, ієрархія та поліморфізм. Розглянемо їх більш детально.

Абстрагування

Абстрагування – це процес визначення лише найважливіших особливостей деякого об'єкта в певній предметній області, що роблять його унікальним.

Використовуючи абстрагування, програміст приховує всі дані про реальний об'єкт, крім необхідних для розроблення конкретної програми, щоб зменшити її складність. Таким чином, об'єкт, що залишається, є моделлю оригіналу, з якого вилучені несуттєві деталі. Цю модель можна назвати деяким *уявленням оригінального об'єкта* (абстракцією). Кожна *абстракція є іменованою сутністю*, що складається з певних атрибутів і поведінки, характерних для конкретного використання оригіналу. В об'єктно-орієнтованій парадигмі вона найчастіше буде називатися класом.

Розглянемо приклад використання принципу абстрагування в реальному житті. Наприклад, необхідно розробити дві програми, де важливим об'єктом предметної області є кішка.

Перша програма призначена для використання у ветеринарній клініці. Тому тут можуть бути важливими, такі параметри кішки, як кличка, порода, рік народження, вага, історія хвороби тощо, а також деякі варіанти її поведінки: кусати, приймати ліки, реагувати на біль.

Друга ж програма призначена для домашнього використання. Тому важливими параметрами кішки можуть бути її кличка, колір і довжина шерсті, характер, а варіантами поведінки – нявкати, стрибати, спати, приймати їжу та тому подібне.

Таким чином, об'єкти реального світу можуть мати будь-яку кількість абстракцій.

Інкапсуляція

Принцип інкапсуляції є однією з основних концепцій ООП. Він відповідає за приховування внутрішніх елементів класу. Екземпляри класів надають сервіси лише за допомогою їхнього "інтерфейсу", який, як правило,

визначається загально доступними варіантами поведінки, що були визначені в певному класі. Це полегшує оновлення або заміну внутрішніх елементів класів, не впливаючи на інший код програми.

Інкапсуляція дозволяє використовувати реальні чи "віртуальні" об'єкти без необхідності знати їхню внутрішню реалізацію. Хорошим прикладом є телевізор. Телеглядачу не потрібно знати внутрішній устрій телевізора, щоб використовувати його. Усе, що йому потрібно – це пульт дистанційного керування з невеликим набором кнопок (інтерфейс пульта), і він зможе дивитися телевізор. Але телемайстру потрібно знати внутрішній устрій телевізора, і він може використовувати приховані від телеглядача елементи управління, що є в його корпусі.

Ієрархія

Крім найпростіших ситуацій, розроблювана програмна система може мати дуже багато абстракцій. Для спрощення розуміння та розроблення таких систем доцільно знайти ієрархічні зв'язки між абстракціями.

Таким чином, принцип ієрархії відповідає за впорядкування абстракцій, розташування їх за рівнями.

Основні види ієрархічних відносин – це успадкування та агрегація.

Успадкування визначає відношення "загальне – приватне" між абстракціями. Дозволяє описати нову абстракцію на основі вже існуючої (батьківської). Для цього атрибути та функціональність батьківської абстракції запозичуються. Наприклад, усі котячі мають чотири лапи, є хижаками та полюють на свою здобич. Ця функціональність може бути визначена один раз у класі "Котячі"; всі хижаки з біологічної родини котячих (класи "Тигр", "Пума", "Рись" та ін.) можуть використовувати її.

Агрегація – відношення "ціле – частина" між абстракціями. Дозволяє описати нову абстракцію, комбінуючи існуючі. Приклад відношення агрегації наведений на рис. 3.1.

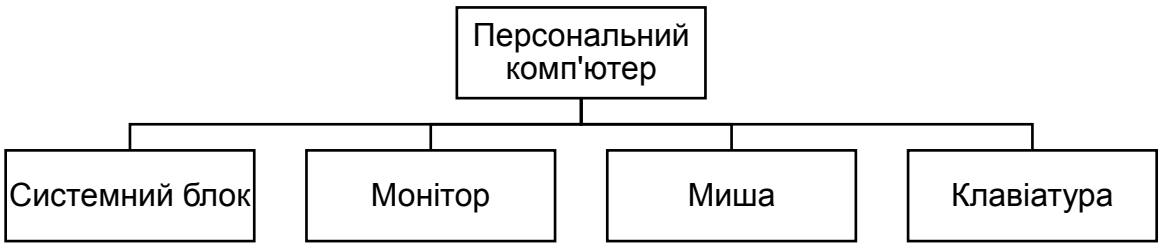

Рис. 3.1. **Відношення агрегації**

Тут "Персональний комп'ютер" – ціле, а всі інші абстракції – частки. Тобто ціле складається з незалежних частин.

Поліморфізм

Поліморфізм – це грецький термін, який означає здатність приймати більше однієї форми. Він дозволяє розглядати екземпляри похідної абстракції як екземпляри його батьківської абстракції.

Відношення спадкування між геометричними фігурами розглянуто на рис. 3.2.

Рис. 3.2. **Приклад використання поліморфізму**

Площу будь-якої геометричної фігури можна обчислити. Але площі трикутника, квадрата та кола обчислюються за різними формулами. Поліморфізм дозволяє нам поводитися з трикутником, квадратом чи колом, як з геометричною фігурою взагалі, та дати їй команду "обчислити площу". Цю команду "розуміють" усі конкретні фігури, але алгоритм її виконання залежить від її поточного типу. Завдяки принципу поліморфізму ми можемо обчислювати площу конкретної фігури, не перевіряючи, якою саме вона є. Також це значно полегшує додавання до ієрархії успадкування нових фігур.

3.2. Поняття об'єкта. Характеристики об'єкта. Поняття класу. Співвідношення між класом та його об'єктом

Що ж є об'єктом, а що ні? Здатністю до розпізнання об'єктів фізичного світу людина володіє із самого раннього віку. З точки зору сприйняття людиною об'єктом може бути:

відчутний і (або) видимий предмет;

щось, сприймане мисленням;

щось, на що спрямована думка або дія.

Отже, об'єкт моделює частину навколишньої дійсності й у такий спосіб існує в часі та просторі. Термін **об'єкт** у програмному забезпеченні вперше був уведений у мові Simula і застосовувався для моделювання реальності.

Об'єктами реального світу не вичерпуються типи об'єктів, цікаві для проектування програмних систем. Інші важливі типи об'єктів вводяться на етапі проектування; їхня взаємодія слугує механізмом відображення поводження більш високого рівня.

Існують такі об'єкти, для яких визначені явні концептуальні межі, але самі об'єкти становлять невловимі події або процеси. Наприклад, хімічний процес на заводі можна трактувати як об'єкт, тому що він має чітку концептуальну межу, взаємодіє з іншими об'єктами за допомогою впорядкованого та розподіленого в часі набору операцій і проявляє добре визначене поводження. Розглянемо систему просторового проектування CAD/CAM. Два тіла (наприклад, сфера та куб) мають, як правило, нерегулярне перетинання. Хоча ця лінія перетинання не існує окремо від сфери та куба, вона все-таки є самостійним об'єктом із чітко певними концептуальними межами.

Подібно тому, як людина, яка взяла у руки молоток, починає бачити у навколишньому світі тільки цвяхи, проектувальник з об'єктно-орієнтованим мисленням починає сприймати весь світ у вигляді об'єктів. Зрозуміло, такий погляд трохи спрощений, тому що існують поняття, що явно не є об'єктами. До їхнього числа належать *атрибути* – такі, як час, краса, колір, емоції (наприклад, любов або гнів). Однак потенційно все зазначене – це *властивості*, притаманні об'єктам. Можна, наприклад, стверджувати, що деяка людина (об'єкт) любить свою дружину (інший об'єкт) або що конкретний кіт (ще один об'єкт) – сірий.

Корисно розуміти, що об'єкт – це щось, що має чітко певні межі. Проте цього недостатньо, щоб відокремити один об'єкт від іншого або дати оцінку якості абстракції. На основі наявного досвіду можна дати таке визначення: *об'єкт має стан, поводження та ідентичність; структура та поводження схожих об'єктів визначає загальний для них клас; терміни "екземпляр класу" й "об'єкт" взаємозамінні.*

Стан

Розглянемо торговельний автомат, що продає напої. Поводження такого об'єкта полягає в тому, що після опускання в нього монети та натискання кнопки автомат видає обраний напій. Що відбудеться, якщо

спочатку буде натиснута кнопка вибору напою, а потім уже опущена монета? Більшість автоматів просто нічого не зроблять, тому що користувач порушив їхні основні правила. Інакше кажучи, автомат відігравав роль (очікування монети), яку користувач ігнорував, спочатку нажавши кнопку. Або припустимо, що користувач автомата не звернув увагу на попереджувальний сигнал "Киньте стільки дрібних грошей, скільки коштує напій" і опустив в автомат зайву монету. У більшості випадків автомати не дружні до користувача, тому радісно "заковтують" усі гроші.

У кожній з таких ситуацій бачимо, що поводження об'єкта визначається його історією: важлива послідовність учинених над об'єктом дій. Така залежність поводження від подій і від часу пояснюється тим, що об'єкт має внутрішній стан. Для торговельного автомата, наприклад, стан визначається сумою грошей, опущених до натискання кнопки вибору. Інша важлива інформація – це набір сприйманих монет і запас напоїв.

На основі цього прикладу можна дати таке визначення: с*тан об'єкта характеризується переліком (звичайно статичним) усіх властивостей даного об'єкта та поточними (звичайно динамічними) значеннями кожної із цих властивостей.*

Однією із властивостей торговельного автомата є здатність приймати монети. Це статична (фіксована) властивість у тому розумінні, що вона – істотна характеристика торговельного автомата. З іншого боку, цій властивості відповідає динамічне значення, що характеризує кількість прийнятих монет. Сума збільшується в міру опускання монет в автомат і зменшується, коли продавець забирає гроші з автомата. У деяких випадках значення властивостей об'єкта можуть бути статичними (наприклад, заводський номер автомата), тому в цьому визначенні використаний термін "звичайно динамічними". Усі властивості мають деякі значення. Ці значення можуть бути простими кількісними характеристиками, а можуть посилатися на інший об'єкт.

Поводження

Об'єкти не існують ізольовано, а піддаються впливу або самі впливають на інші об'єкти. **Поводження** – це те, як об'єкт діє і реагує; поводження виражається в термінах стану об'єкта та переданні повідомлень.

Іншими словами, поводження об'єкта – це його діяльність, що спостерігається і ззовні перевіряється. *Операцією* називають певний вплив одного об'єкта на іншій з метою викликати відповідну реакцію.

Природа класів

Поняття класу й об'єкта настільки тісно зв'язані, що неможливо говорити про об'єкт безвідносно до його класу. Однак існує важливе розходження цих двох понять. Тоді як об'єкт позначає конкретну сутність, певну в часі й у просторі, клас визначає лише абстракцію істотного в об'єкті. Таким чином, можна говорити, що клас "Ссавці" включає характеристики, загальні для всіх ссавців. Для вказівки на конкретного представника ссавців необхідно сказати "це – ссавець" або "то – ссавець". У контексті об'єктно-орієнтованого аналізу дамо таке визначення класу: **клас** – це якась безліч об'єктів, що мають загальну структуру та загальне поводження.

Будь-який конкретний об'єкт є просто екземпляром класу. Що не є класом? Об'єкт не є класом, хоча надалі побачимо, що клас може бути об'єктом. Об'єкти, не зв'язані спільністю структури та поводження, не можна об'єднати в клас, тому що за визначенням вони не зв'язані між собою нічим, крім того, що всі вони об'єкти.

3.3. Об'єктно-орієнтований аналіз та його мета. Головні види вимог до програмної системи. Об'єктно-орієнтоване проектування. Об'єктно-орієнтоване програмування

Розроблення програмного забезпечення, як правило, розбивається на декілька етапів. Ці етапи є абстрактними поняттями, які використовуються для відокремлення діяльності, що відбувається в рамках кожної фази процесу. Часто ці етапи можуть включати: аналіз вимог, планування, проектування, кодування, тестування, розгортання, технічне обслуговування тощо.

У випадку таких методологій розроблення, як метод "водоспаду", ці етапи є послідовними та призначені бути повністю відокремленими один від одного. Отже, розроблення програми з використанням цього методу накладає певні обмеження. Так, відкриття, зроблені під час фази тестування або розгортання, навряд чи можуть вплинути на рішення, вже прийняті протягом фази планування або проектування. Ці обмеження призвели до поширення ітеративних моделей, подібних до об'єктно-орієнтованого аналізу та проектування.

Об'єктно-орієнтований аналіз

Об'єктно-орієнтований аналіз (ООА) – це методологія, за якої вимоги до програмної системи сприймаються з точки зору класів і об'єктів, виявлених у предметній області.

На етапі ООА розробники працюють з майбутніми користувачами програми й експертами предметної області. На цьому етапі передбачається, що деталі впровадження будуть у більшій мірі або повністю ігноруватися.

Цілі проведення об'єктно-орієнтованого аналізу

Проведення ООА має вирішувати такі завдання:

1) зрозуміти проблеми, які повинна вирішити програмна система;
2) поставити значущі питання про проблему та про програмну систему;
3) забезпечити основу для відповідей на запитання про специфічні властивості проблеми та програмної системи;
4) визначити, що програмна система повинна та не повинна робити;
5) переконатися, що програмна система задовільнює потреби її користувачів;
6) забезпечити основу для розроблення програмної системи.

ООА – це ітеративний етап, призначений для моделювання вимог до програмного забезпечення. Типова фаза ООА складається з таких етапів**:**

1) знайти та визначити об'єкти;
2) організувати об'єкти;
3) описати, як об'єкти взаємодіють один з одним;
4) визначити зовнішню та внутрішню поведінку об'єктів.

Тобто ООА полягає в тому, щоб створити об'єктну модель програми, яка є незалежною від обмежень – таких, як конкретні технології. Окреслюються варіанти використання програмної системи й абстрактне визначення її найбільш важливих об'єктів за допомогою концептуальної моделі.

Об'єктна модель описує структуру об'єктів, що становлять систему, їх властивості, операції і взаємозв'язок з іншими об'єктами.

У ній мають бути відображені ті поняття і об'єкти реального світу, які важливі для розроблюваної системи. Об'єктна модель може описувати імена, відносини, характеристики кожного об'єкта системи. Ця інформація робить процес розроблення програмної системи набагато простішим.

Вимоги до програмного забезпечення

Опис вимог – це складне завдання, оскільки воно включає набір таких процесів, як виявлення, аналіз, специфікація, перевірка й управління.

Найпоширенішими типами вимог є:

бізнес-вимоги – залежать від основних бізнес-цілей організації, яка є замовником програмного продукту. Надаються, у вигляді списку, який зазвичай займає одну сторінку;

ринкові вимоги – вони дещо деталізують бізнес-вимоги, окреслюючи потреби ринку. Як правило, надаються у вигляді списку або таблиці обсягом до п'яти сторінок;

функціональні вимоги – детально розкривають функціональність програмного продукту, необхідну для того, щоб користувачі могли виконувати свої завдання. Один з найкращих способів документування функціональних вимог – це опис варіантів використання програмного продукту. Відповідний контент може займати декілька кілька сотень сторінок;

нефункціональні вимоги – не пов'язані з функціональними можливостями програмного продукту, але охоплюють такі показники, як надійність, масштабованість, безпека, інтеграція тощо. Також відомі як ***якісні атрибути програмного продукту.***

Отже, етап ООА визначає об'єкти, їх взаємозв'язок та поведінку, використовуючи концептуальну модель, тобто абстрактне визначення об'єктів.

Об'єктно-орієнтоване проектування

Етап об'єктно-орієнтованого проектування вдосконалює модель ООА, застосовуючи необхідні технології та інші обмеження щодо впровадження. Відмінність між аналізом і проектуванням часто характеризується як "що потрібно зробити" та "як ми це будемо робити". Вхідними даними для об'єктно-орієнтованого проектування є результати об'єктно-орієнтованого аналізу. Але аналіз і дизайн можуть відбуватися паралельно.

Об'єктно-орієнтоване проектування – це методологія проектування, що поєднує процес об'єктної декомпозиції та прийоми подання логічної, фізичної, статичної та динамічної моделей проектованої системи.

Під час об'єктно-орієнтованого проектування описують класи та їхнє відношення, взаємодію між об'єктами тощо.

Моделі, що розробляються на етапі об'єктно-орієнтованого проектування, зосереджені на описі об'єктів, їх атрибутів, поведінки та взаємодії. Ці моделі повинні містити всі необхідні відомості, щоб програмісти могли реалізувати проект певною мовою програмування.

Стандартною мовою об'єктно-орієнтованого моделювання, де-факто, є Unified Modeling Language (UML). **UML** – відкритий стандарт, який використовує графічні позначення для створення моделей системи. UML головним чином застосовується для визначення, візуалізації, проектування та документування програмних систем. UML не є мовою програмування, але на базі UML-моделей можлива генерація "каркасу" вихідного коду програми.

UML дозволяє розробляти моделі, використовуючи UML-діаграми. Вони показують систему з різних перспектив: зовнішньої (моделюється контекст або середовище системи), взаємодії (моделюється взаємодія між компонентами системи; між системою та іншими системами), структурної (моделюється організація системи або структура даних, що обробляються системою), поведінкової (моделюється динамічна поведінка системи і як вона реагує на події).

Це дещо аналогічно кресленням, що визначають конструкцію деякого виробу. Вони мають головний вигляд, який дає найповнішу уяву про форму та розміри виробу, відносно якого розташовують інші основні види: зверху, зліва, справа, знизу та позаду.

Найбільш часто використовують такі UML-діаграми:

діаграма варіантів використання – показує взаємодію системи та її середовища (користувачів або систем) у певній ситуації;

діаграма класів – показує різні об'єкти, їх відношення, їхню поведінку та атрибути;

діаграма послідовності – показує взаємодію між різними об'єктами в системі, а також між акторами й об'єктами в системі;

діаграма станів – показує, як система реагує на зовнішні та внутрішні події;

діаграма діяльності – показує потік даних між процесами в системі.

UML-діаграми можна створювати на папері або на дошці, принаймні на початкових етапах проекту. Але для цього також є багато застосунків.

Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування (ООП) – це методологія програмування, за використанням якої програми розглядають у вигляді

сукупності взаємодіючих об'єктів; будь-який об'єкт є екземпляром певного класу; класи утворюють ієрархію.

На етапі ООП розробляються структури даних, алгоритми, виконується створення та редагування вихідного коду програми, її трансляція, налагодження і тестування.

Вихідний код програми розробляється вибраною мовою програмування, яка підтримує об'єктно-орієнтовану парадигму. Найбільш поширеними з них сьогодні є C++, Java, C#, Python, JavaScript, PHP.

3.4. UML-діаграми класів. Відношення на діаграмі класів. CASE-засоби

Діаграма класів – один з найбільш використовуваних типів діаграм в UML, оскільки вони дозволяють планувати структуру конкретної системи, моделюючи її класи, ознаки, операції та співвідношення між об'єктами.

UML-діаграма класів призначена для подання статичної структури моделі програмної системи. Вона може відбивати, зокрема, різні взаємозв'язки між окремими сутностями предметної області (такими, як об'єкти та підсистеми), а також описує їхню внутрішню структуру та типи відносин.

Клас у мові UML слугує для позначення безлічі об'єктів, які мають однакову структуру, поводження і відношення з об'єктами інших класів. Клас може не мати екземплярів або об'єктів. У цьому випадку він називається *абстрактним класом*. Графічно клас зображується у вигляді прямокутника, що додатково може бути розділений горизонтальними лініями на секції. У цих секціях можуть вказуватися ім'я класу, атрибути й операції.

Обов'язковим елементом позначення класу є його ім'я. **Ім'я класу** повинне бути унікальним. Воно вказується в першій верхній секції прямокутника. Як імена класів рекомендується використовувати іменники, записані без пробілів. Необхідно пам'ятати, що саме імена класів утворять *словник предметної області*. Прикладами імен класів можуть бути такі іменники, як "Співробітник", "Компанія", "Керівник", "Клієнт", "Продавець", "Менеджер", "Офіс" та, ті, що мають безпосереднє відношення до предметної області та функціонального призначення проектованої системи.

У другій зверху секції прямокутника класу записуються його **атрибути**. Кожному атрибуту класу відповідає окремий рядок тексту, що складається із *квантора видимості* атрибута, імені атрибута, його кратності, типу значень атрибута та, можливо, його вихідного значення.

Квантор видимості може приймати одне із трьох можливих значень:

1) *загальнодоступний* (public). Атрибут із цією областю видимості доступний з будь-якого іншого класу пакета, у якому визначена діаграма;

2) *захищений* (protected). Атрибут із цією областю видимості недоступний для всіх класів, за винятком підкласів даного класу;

3) *закритий* (private). Атрибут із цією областю видимості недоступний для всіх інших класів без винятку.

Ім'я атрибута становить собою рядок тексту, що використовується як ідентифікатор відповідного атрибута і тому має бути унікальним у межах даного класу. Ім'я атрибута є єдиним обов'язковим елементом синтаксичного позначення атрибута.

Кратність атрибута характеризується загальною кількістю конкретних атрибутів даного типу, що входять до складу окремого класу.

Тип атрибута є виразом, семантика якого визначається мовою специфікації відповідної моделі. У нотації UML тип атрибута іноді визначається залежно від мови програмування, яку передбачається використовувати для реалізації даної моделі. У найпростішому випадку тип атрибута вказується рядком тексту, що має осмислене значення в межах пакета або моделі, до яких належить розглянутий клас.

У третій зверху секції прямокутника записуються *операції класу*. Операція становить деякий сервіс, що надає будь-який екземпляр класу на певну вимогу. Сукупність операцій характеризує *функціональний аспект поводження класу*.

Кожній операції класу відповідає окремий рядок, що складається із квантора видимості операції, імені операції, вираження типу, що повертається операцією.

Ім'я операції становить рядок тексту, що використовується як ідентифікатор відповідної операції і тому повинен бути унікальним у межах даного класу. Крім внутрішнього устрою або структури класів на відповідній діаграмі вказуються різні відношення між класами. Сукупність типів таких відношень зафіксована в мові UML і визначена семантикою цих типів відносин. Базовими відношеннями або зв'язками, в мові UML є:

1) відношення залежності;
2) відношення асоціації;
3) відношення узагальнення;
4) відношення реалізації;
5) відношення агрегації.

Відношення залежності в загальному випадку вказує деяке семантичне відношення між двома елементами моделі або двома множинами таких елементів, що не є відношенням асоціації, узагальнення або реалізації. Відношення залежності використовується в такій ситуації, коли деяка зміна одного елемента моделі може потребувати зміни іншого залежного від нього її елемента.

Відношення асоціації відповідає наявності деякого відношення між класами. Найбільш простий випадок такого відношення – бінарна асоціація. Вона зв'язує в точності два класи та, як виняток, може зв'язувати клас із самим собою.

Тернарна асоціація і асоціації більш високого порядку в загальному випадку називають N-арною асоціацією. Така асоціація зв'язує деяким відношенням три та більше класів; водночас один клас може брати участь в асоціації більше одного разу.

Відношення агрегації має місце між декількома класами в тому випадку, якщо один із класів становить деяку сутність, що як складові частини інші сутності. Це відношення має фундаментальне значення для опису структури складних систем, оскільки застосовується для подання системних взаємозв'язків типу "частина – ціле". Розкриваючи внутрішню структуру системи, відношення агрегації показує, з яких компонентів складається система і як вони зв'язані між собою. Це відношення описує декомпозицію складної системи на більш прості складові, які також можуть бути піддані декомпозиції, якщо в цьому надалі виникне необхідність. Окремим випадком відношення асоціації є відношення агрегації, яке, в свою чергу, теж має спеціальну форму – відношення композиції.

Відношення композиції як окремий випадок відношення агрегації слугує для виділення спеціальної форми відношення "частина – ціле", за якої частини в деякому змісті перебувають усередині цілого. Специфіка взаємозв'язку між ними полягає в тому, що частини не можуть виступати у відриві від цілого, тобто зі знищенням цілого знищуються і всі його складові.

Відношення узагальнення є відношенням між більш загальним елементом (батьком або предком) і більш частковим або спеціальним, елементом (дочірнім або нащадком). Стосовно до діаграми класів це відношення описує ієрархічну будову класів і спадкування їхніх властивостей і поводження. Передбачається, що клас-нащадок має всі властивості та поводження класу-предка, водночас маючи свої власні властивості та поводження, які відсутні у класа-предка.

Приклад UML-діаграми класів наведено на рис. 3.3.

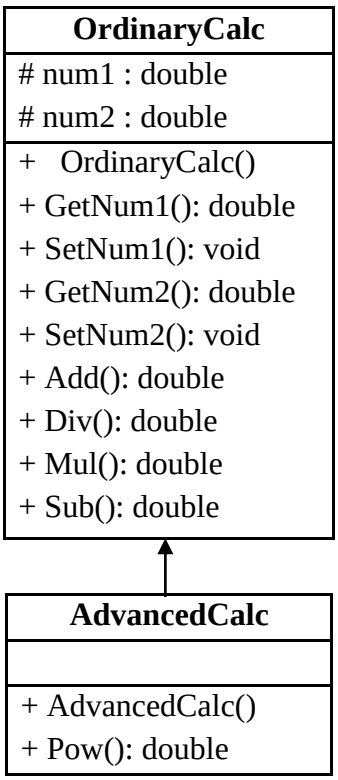

Рис. 3.3. **Діаграма класів**

Класи OrdinaryCalc та AdvancedCalc зв'язані відношенням узагальнення.

Клас OrdinaryCalc є батьківським. Він має атрибути num1 і num2 типу double та квантором видимості protected. У цьому класі визначено дев'ять операцій з квантором видимості public. Більшість операцій повертає деякий тип даних, а саме double або void.

Клас AdvancedCalc є нащадком класу OrdinaryCalc. Він визначає дві операції з квантором видимості public, одна з яких повертає тип даних double.

CASE-засоби

Computer-aided software engineering (CASE) – це методи та програмні засоби, що використовуються для проектування та реалізації застосунків. Вони використовуються менеджерами проектів програмного забезпечення, аналітиками та інженерами.

Використання CASE-засобів прискорює розроблення проекту для отримання бажаного результату та допомагає розкрити недоліки, перш ніж перейти до наступного етапу розроблення програмного забезпечення.

Є низка CASE-засобів для спрощення різних етапів процесу. Це інструменти аналізу, засоби проектування, інструменти управління проектами, інструменти документації тощо.

Питання для самопідготовки

1. Ознаки складних програмних систем.
2. Переваги об'єктно-орієнтованого підходу до розроблення програмних систем.

Запитання для самодіагностики

1. Охарактеризуйте прості та складні програмні системи.
2. Як можна боротися зі складністю програм?
3. У чому полягає сутність об'єктно-орієнтованої декомпозиції?
4. Які основні елементи має об'єктно-орієнтована технологія?
5. Яке співвідношення існує між класами й об'єктами?
6. Які характеристики має будь-який об'єкт?
7. Сформулюйте принципи об'єктно-орієнтованого підходу.

Література: [2; 7; 12; 18].

Розділ 2. Основні елементи об'єктно-орієнтованого програмування

4. Абстрагування даних та інкапсуляція

Мета теми: набуття знань щодо елементів класу, рівнів доступу до них, основ використання об'єктів, послідовності створення та ініціалізації об'єкта, особливостей застосування конструкторів і статичних членів класу.

Професійні компетентності: здатність до об'єктно-орієнтованого мислення.

Основні питання:

4.1. Абстрактні типи даних. Проектування абстрактного типу даних.

4.2. Класи та структури. Елементи класу. Особливості використання статичних елементів. Доступ до елементів класу, модифікатори доступу.

4.3. Поняття про створення, ініціалізацію та використання об'єктів класу. Посилання this.

4.4. Життєвий цикл об'єктів. Послідовність створення та ініціалізації об'єкта. Конструктори. Конструктор за замовчуванням. Основні властивості конструкторів. Перевантаження конструкторів. Звільнення пам'яті. Система "збирання сміття".

Питання для опрацювання: абстрактні типи даних, класи та структури, створення, ініціалізація та використання об'єктів класу, життєвий цикл об'єктів.

Ключові слова: абстрактні типи даних, інтерфейс, клас, структура, члени класу, поле, метод, властивість, константа, індексатор, подія, оператор, модифікатор доступу, ініціалізація, посилання, конструктор.

4.1. Абстрактні типи даних. Проектування абстрактного типу даних

Абстрактні типи даних (АТД) – типи даних, визначені набором операцій (інтерфейс). Цей підхід до визначення та використання нових типів є відмінним способом вирішення складності: ми зменшуємо її, розділяючи обов'язки, що пропонуються типом (що), від типу реалізації (як). Проблеми з реалізацією відтерміновуються або делегуються іншим, тому потрібно лише зосередити увагу на одному аспекті. Цей процес, як правило, називають *розподілом задач*. Оскільки ми визначаємо ці нові типи високого рівня з абстрактної точки зору, їх можна назвати абстрактними типами. Абстрактний тип означає сукупність об'єктів, що характеризуються переліком операцій, які можуть бути застосовані до них разом з точною специфікацією кожної з цих операцій. Зазвичай список операцій, що визначають тип та його специфікацію, називають *поведінкою типу, специфікацією типу* або *типом контракту*.

Сукупність сутностей, які поділяють операції, визначені для типу, називають *екземплярами* цього *типу*. Прикладом абстрактного типу може бути тип Student. Це абстрактний тип, прикладом якого є кожен з конкретних студентів (Joe, Ann ...). Його операціями можуть бути:

установлення імені студента;

зареєстрування учня на предмет;

отримання переліку предметів, у яких зарахований конкретний студент, тощо.

4.2. Класи та структури. Елементи класу. Особливості використання статичних елементів. Доступ до елементів класу, модифікатори доступу

Класи – серце кожної об'єктно-орієнтованої мови.

Синтаксис визначення класів на C# і Java простий. Помістивши перед ім'ям вашого класу ключове слово *class,* ви вставляєте члени класу, укладені у фігурні дужки. Наприклад:

```
class Employee {
private long employeeId; }
```

Як бачимо, цей найпростіший клас із ім'ям *Employee* містить єдиний член – *employeeId.*

Зауваження. Імені члена передує ключове слово private – це модифікатор доступу (access modifier).

Члени класу розрізняють за видами.

Поле. Так називається член-змінна, яка має деяке значення. В ООП поля іноді називають даними об'єкта. До поля можна застосовувати кілька модифікаторів залежно від характеру його використання. У число модифікаторів входять *static, readonly* та *const.*

Метод. Це реальний код, що впливає на дані об'єкта (або поля). Фактично методи – це функції, які визначають певні дії з даними.

Властивості (тільки C#). Їх іноді називають *"розумними" полями* (smart fields), тому що вони дійсно є методами, які клієнти класу сприймають як поля. Це забезпечує клієнтам більший ступінь абстрагування за рахунок того, що їм не потрібно знати, чи звертаються вони до поля прямо або через виклик метода-аксесора.

Константи. Як можна припустити, виходячи з імені, константа – це поле, значення якого змінити не можна. Нижче обговоримо константи та порівняємо їх із сутністю за назвою *незмінні* (readonly) поля.

Індексатори (тільки C#)**.** Якщо властивості – це "розумні" поля, то індексатори – це "розумні" масиви, тому що вони дозволяють індексувати об'єкти методами-аксесорами *get* і *set.* За допомогою індексатора легко проіндексувати об'єкт для установки або отримання значень.

Події. Подія викликає виконання деякого фрагмента коду. Наприклад, події виникають під час рухання миші, клацанні або зміні розмірів вікна.

Перевантажені оператори (тільки C#)**.** Використовуючи перевантаження операторів C#, можна додавати до класу стандартні математичні оператори, які дозволяють писати більш інтуїтивно зрозумілий код.

Модифікатори доступу

Ознайомимося з модифікаторами, що використовуються для задання ступеня "видимості" або доступності даного члена для коду, що перебуває за межами його власного класу. Вони називаються *модифікаторами доступу* (access modifiers), їх приклади наведені в табл. 4.1.

Таблиця 4.1

Модифікатори доступу

Модифікатор доступу	Опис
public	Член доступний поза визначенням класу й ієрархії похідних класів
protected	Член недоступний за межами класу, до нього можуть звертатися тільки похідні класи
private	Член недоступний за межами області видимості класу, в якому його визначили. Тому доступу до цих членів немає навіть у похідних класів
internal	Член доступний тільки в межах поточної одиниці компіляції. Модифікатор доступу *internals* у плані обмеження доступу є гібридом *public* і *protected,* залежним від місця розташування коду.

У табл. 4.1 наведені види види модифікаторів доступу мови C#.

4.3. Поняття про створення, ініціалізацію та використання об'єктів класу. Посилання this

Припустимо, що ми розробили деякий клас SimpleClass мовою Java (мовою C# – аналогічно):

```
class SimpleClass {
   private int x;
   public void setX(int value) { x = value; }
   public int getX(){
        return x;
   }
}
```

Тут метод setX призначений для ініціалізації поля x. Метод getX повертає поточне значення цього поля.

Для створення та використання об'єкта класу SimpleClass необхідний вихідний код, подібний до такого:

```
class MainClass {
   public static void main(String[] args) {
        SimpleClass r = new SimpleClass(); //1
        r.setX(25); //2
        System.out.println("Число = " + r.getX()); //3
   }
}
```

Рядок 1 відповідає за створення об'єкта класа SimpleClass. Тут r є посиланням на нього. Далі через це посилання викликаються методи setX та getX. Рядок 2 призначений для задання поточного значення поля x класу SimpleClass. Рядок 3 виводить на консоль текстове повідомлення та значення поля x, яке повертає метод getX.

Отже, після запуску програми на виконання на консолі з'явиться таке повідомлення: **Число = 25.**

Але, якщо закоментувати рядок 2, результат виконання програми змінюється: **Число = 0.**

Тобто, якщо забути викликати метод setX, значення поля x буде визначене за замовчуванням; для цілого та інших числових типів воно дорівнюватиме нулю. Це може бути певною проблемою. У C# і Java ця процедура вирішується за допомогою конструкторів.

Посилання this

Іноді потрібно, щоб метод посилався на об'єкт, що його викликав. Для цього в Java та C# треба визначити ключове слово this. Воно може використовуватися всередині будь-якого методу для посилання на поточний об'єкт та є його адресою в пам'яті.

this завжди є посиланням на об'єкт, для якого був викликаний метод. Ключове слово this можна використовувати всюди, де допускається посилання на об'єкт типу поточного класу.

Розглянемо використання ключового слова *this* в мові Java (ця інформація також релевантна і для C#).

Ключове слово *this* у Java можна використовувати:
для посилання на змінну екземпляра поточного класу;
для виклику методу екземпляра поточного класу;
для виклику конструктора поточного класу;
як аргумент у виклику методу;
як аргумент у виклику конструктора;
для повернення поточного екземпляра класу з методу.
Далі наведено приклади деяких типових випадків використання *this.*

Посилання на змінну екземпляра поточного класу

У наведеному прикладі параметри методу setFields і змінні екземпляра мають те саме ім'я.

```
class Apple {
        private String sort, color;
        private double diameter;

        public void setFields(String sort, String color, double diameter) {
                this.sort = sort;
        this.color = color;
        this.diameter = diameter;
        }
}
```

Отже, ми використовуємо ключове слово *this*, щоб відрізнити локальну змінну від змінної екземпляра.
Якщо це не так, немає потреби використовувати ключове слово *this*.

Виклик методу поточного класу

Можна викликати метод поточного класу за допомогою ключового слова *this*.

```
class MyClass {
    private void sayHello(String name) {
        System.out.println("Привіт, " + name);
  }
    public void printText(String text) { this.hello(text); }
  }
```

Якщо слово не застосоване, компілятор автоматично додає ключове слово *this* під час виклику методу.

Особливості посилання this

Розглядуване посилання діє в такому полі значень:

1) кожен новий об'єкт має прихований покажчик this;
2) this вказує на початок свого об'єкта в пам'яті комп'ютера;
3) this не треба додатково оголошувати;
4) this передається як прихований аргумент в усі нестатичні методи свого об'єкта;
5) this – це локальна змінна, яка недоступна за межами об'єкта.

4.4. Життєвий цикл об'єктів. Послідовність створення та ініціалізації об'єкта. Конструктори. Конструктор за замовчуванням. Основні властивості конструкторів. Перевантаження конструкторів. Звільнення пам'яті. Система "збирання сміття"

Конструктори

Одна з найбільших переваг мов ООП (таких, як C# і Java) полягає в тому, що можна визначати спеціальні методи, які викликаються щораз під час створення екземпляра класу. Ці методи називають **конструкторами** (constructors).

Гарантія ініціалізації об'єкта належним чином є ключовою вигодою від конструктора. Коли користувач створює екземпляр об'єкта, викликається його конструктор, що повинен повернути управління, перш ніж користувач зможе виконати над об'єктом іншу дію. Саме це допомагає забезпечувати цілісність об'єкта та зробити написання застосунків на об'єктно-орієнтованих мовах набагато надійнішим.

Конструктор повинен мати те ж ім'я, що і у самого класу. Наведемо простий клас C# із таким же простим конструктором:

```
class ConstructorApp {
ConstructorApp() {
Console.WriteLine("Я конструктор.");
}
public static void Main() {
```

```
ConstructorApp app = new ConstructorApp();
}
}
```

Конструктори не повертають значень. У разі спроби використання конструктором імені типу в ролі префіксу компілятор повідомить про помилку. Варто звернути увагу на спосіб створення екземпляра об'єкта. Ця операція здійснюється за допомогою ключового слова *new:*

<клас> <об'єкт> = new *<клас> (аргументи конструктора)*

У сучасному програмуванні на C# і Java велику роль відіграють використання збирання сміття в пам'яті та її звільнення. Це забезпечує раціональніше використання ресурсів задіяної пам'яті для програмного коду та застосунку. На початку необхідно з'ясувати, як платформи .NET і Java SE управляють уже розміщеними об'єктами за допомогою процесу, який називають ***збиранням сміття***. Програмістам, які використовують C# чи Java, не доводиться видаляти об'єкти з пам'яті "вручну".

Об'єкти розміщуються в області пам'яті, яку називають *керованою динамічною пам'яттю,* де "в деякий відповідний момент" вони будуть автоматично знищені збиральником сміття.

Клас – це своєрідний "шаблон" з описом того, як екземпляр даного типу повинен виглядати та поводитися в пам'яті. Розглянемо простий клас Car мовою C#:

```
public class Car
  {
    private int currSp;
    private string petName;
    public Car() { }
    public Car(String name, int speed)
    {
      petName = name;
      currSp = speed;
    }
    public override string ToString()
    {
      return string.Format("{0} має швидкість {1} км/ч", petName, currSp);
    }
}
```

Визначивши клас, ви можете розмістити в пам'яті будь-яке число відповідних об'єктів, використовуючи ключове слово new. Проте слід розуміти, що ключове слово new повертає посилання на об'єкт у динамічній пам'яті, а не сам реальний об'єкт: змінна з посиланням запам'ятовується в стеку для подальшого використання в застосунку.

```
class Program
  {
    static void Main(string[] args)
    {
      Car refToMyCar = new Car("Lanos", 50);
      Console.WriteLine(refToMyCar.ToString());
      Console.ReadLine();
    }
  }
```

Для виклику членів об'єкта до збереженого посилання слід застосувати операцію, що позначається крапкою.

Основні відомості про існування об'єктів

Для побудови C#-застосунків користувач має право припускати, що керована динамічна пам'ять оброблятиметься без його прямого втручання. Правило управління пам'яттю .NET є дуже простим – слід помістити об'єкт у керовану динамічну пам'ять за допомогою ключового слова new і "забути" про це.

Збиральник сміття знищить створений об'єкт, коли той більше не буде потрібний. Очевидне питання: "Як збиральник сміття визначає, що об'єкт більше не потрібний?". Спрощена відповідь полягає в тому, що збиральник сміття видаляє об'єкт з динамічної пам'яті тоді, коли об'єкт стає *недоступним* для всіх частин програмного коду. Припустимо, що ви маєте метод, який розміщує локальний об'єкт Car.

```
public static Void MakeACar () {
// Якщо myCar є єдиним посиланням на об'єкт Car,
//то об'єкт може бути знищений після повернення з методу.
Car myCar = new Car () ;
}
```

Посилання на об'єкт (myCar) було створене безпосередньо в методі MakeACar () і не передавалося за межі області видимості, що визначає

це посилання об'єкта ні у вигляді повертаного значення, ні у вигляді параметрів ref/out. Тому після завершення роботи викликаного методу посилання myCar стає недоступним, і відповідний об'єкт Car виявляється *кандидатом* для видалення в "сміття".

Проте слід зазначити, що не можна гарантувати негайне видалення цього об'єкта з пам'яті відразу ж після закінчення роботи MakeACar (). У цей момент можна гарантувати тільки те, що під час наступного збирання сміття в загальномовному середовищі виконання (CLR) об'єкт myCar може бути без побоювань знищений.

Програмування в оточенні, що забезпечує автоматичне збирання сміття, значно спрощує завдання розроблення застосунків. Програмісти, які використовують C++, знають: якщо в C++ забути вручну видалити розміщені в динамічній пам'яті об'єкти, може відбутися "витік пам'яті".

Насправді запобігання витіканню пам'яті є одним із найбільш трудомістких аспектів програмування мовами, які не є керованими. Доручивши збиральникові сміття знищення об'єктів, ви знімаєте з себе вантаж відповідальності за управління пам'яттю, перекладаючи його на CLR.

CIL-код для new

Коли компілятор C# виявляє ключове слово new, то генерує CIL-інструкцію newobj в рамках реалізації відповідного методу. Якщо виконати **компіляцію програмного** коду поточного прикладу і за допомогою ildasm.exe розглянути отриманий компонувальний блок, то в рамках методу MakeACar () з'являться такі CIL-оператори

```
.method public hidebusig static void MakeACar () cil managed
{
// Code size   7 (0x7)
.maxstack   1
.locals init ([0] class SimpleFinalize.Car c)
IL_0000 IL_0005 IL 0006
newobj instance void   SimpleFinalize.Car:
stloCpOret }
} // end of method Program:.MakeACar
```

Перш ніж обговорити точні правила, що визначають момент видалення об'єкта з керованої динамічної пам'яті, необхідно з'ясувати роль CIL-інструкції newobj.

Керована динамічна пам'ять є не просто випадковим фрагментом пам'яті, доступної для середовища виконання. Збиральник сміття .NET є зразково акуратним "двірником" у динамічній пам'яті – він (за необхідності) навіть стискає порожні блоки пам'яті з метою оптимізації. Щоб спростити завдання збирання сміття, керована динамічна пам'ять має покажчик (зазвичай званий *покажчиком на наступний об'єкт,* або *покажчиком на новий об'єкт),* який ідентифікує точне місце розміщення наступного об'єкта.

Інструкція newobj інформує середовище CLR про те, що необхідно виконати такі головні завдання:

обчислити загальний обсяг пам'яті, необхідної для розміщення об'єкта (включаючи пам'ять, необхідну для членів-змінних і базових класів типу);

перевірити керовану динамічну пам'ять, щоб гарантувати достатній обсяг пам'яті для розміщуваного об'єкта. Якщо пам'яті достатньо, викликається конструктор типу; стороні, яка викликала конструктор, повертається посилання на новий об'єкт у пам'яті, причому адреса цього об'єкта відповідатиме останній позиції покажчика на наступний об'єкт.

Нарешті, перед поверненням посилання стороні, яка викликала, необхідно змінити значення покажчика на наступний об'єкт, щоб покажчик відповідав початку вільної області керованої динамічної пам'яті.

Генерації об'єктів

Коли середовище CLR намагається знайти недоступні об'єкти, це не означає, що буде розглянутий буквально кожен об'єкт, розміщений у керованій динамічній пам'яті. Очевидно, що це вимагало б дуже багато часу, особливо в реальних (тобто великих) застосунках.

Щоб оптимізувати процес, кожен об'єкт у динамічній пам'яті приписується певній "генерації". Ідея достатньо проста: чим довше об'єкт існує в динамічній пам'яті, тим більша вірогідність того, що він там залишиться. Наприклад, об'єкт, що реалізовує Main (), знаходитиметься в пам'яті до тих пір, поки програма не закінчиться. З іншого боку, об'єкти, які недавно розміщені в динамічній пам'яті, найімовірніше стануть незабаром недосяжними (наприклад, об'єкти, створені в рамках області видимості методу). За цих припущень кожен об'єкт у динамічній пам'яті можна віднести до однієї з таких категорій:

генерація 0. Нові, тільки що розміщені об'єкти, які ще ніколи не призначалися для використання в процесі збирання сміття;

генерація 1. Об'єкти, які "пережили" одне збирання сміття (тобто були позначені для використання в процесі збирання сміття, але не були видалені з тієї причини, що в динамічній пам'яті опинилося достатньо місця);

генерація 2. Об'єкти, які "пережили" декілька збирань сміття.

Збиральник сміття спочатку розглядає об'єкти генерації 0. Якщо після виявлення непотрібних об'єктів і відповідного **ЧИЩЕННЯ** вільної пам'яті виявляється достатньо, об'єкти, що ще залишилися, відносяться до генерації 1.

Якщо всі об'єкти генерації 0 уже розглянуті, але пам'яті все одно ще не достатньо, то розглядається "досяжність" об'єктів генерації 1 і виконується збирання сміття серед цих об'єктів. Об'єкти генерації 1, що "вижили", переходять до генерації 2. Якщо збиральник сміття *все ще* вимагає додаткової пам'яті, тоді оцінюються об'єкти генерації 2. Об'єкт генерації 2, що "виживає" в процесі збирання сміття, зберігає приналежність до генерації 2, оскільки це межа для генерацій об'єктів.

Отже, за допомогою призначення ознаки генерації об'єктам у динамічній пам'яті новіші об'єкти (наприклад, локальні змінні) віддалятимуться швидше, тоді як старі об'єкти "турбуватимуться" значно рідше.

Тип System.GC

Бібліотеки базових класів пропонують тип класу System.GC, який дозволяє програмно взаємодіяти зі збиральником сміття, використовуючи безліч статичних членів указаного класу. Слід зазначити, що безпосередньо використовувати цей тип у програмному коді доводиться дуже рідко (якщо доводиться взагалі).

Основні методи класу System.GC:

Collect () – змушує GC виконати збирання сміття;

AddMemoryPressure (), RemoveMemoryPressure () – дозволяють указати числове значення, що характеризує "терміновість" виклику процесу збирання сміття;

CollectionCount() – повертає числове значення, що вказує, скільки разів "виживала" дана генерація у збираннях сміття;

GetGeneration() – повертає інформацію про генерацію, до якої зараз віднесено об'єкт;

GetTotalMemory() – повертає оцінку обсягу пам'яті (у байтах), виділеної зараз для керованої динамічної пам'яті. Логічний параметр указує,

чи повинен виклик очікувати початку збирання сміття, щоб повернути результат;

MaxGeneration – повертає максимум для числа генерацій, підтримуваних у системі. У Microsoft .NET 2.0. Передбачається існування трьох генерацій (0, 1 і 2);

SuppressFinalize() – установлює індикатор того, що даний об'єкт не повинен викликати свій метод Finalize ();

WaitForPendingFinalizers() – припиняє виконання поточного потоку, поки не будуть відпрацьовані всі об'єкти, які передбачають фіналізацію. Цей метод зазвичай викликається безпосередньо після виклику GC.Collect().

Розглянемо метод Main(), у якому ілюструється використання вказаних членів System.GC.

```
static void Main(string[] args)
    {
        // Виведення оцінки (у байтах) для динамічної пам'яті.
        Console.WriteLine("Оцінка об'єму пам'яті (у байтах) ; {0}",
GC.GetTotalMemory(false));
        // Відлік для MaxGeneration починається з нуля, // тому для зручності
додаємо 1.
        Console.WriteLine("Число генерацій для даної ОС: {0}", GC.MaxGeneration
+ 1);
        Car refToMyCar = new Car("Lanos", 100);
        Console.WriteLine(refToMyCar.ToString());
        // Виведення інформації про генерацію для об'єкта refToMyCar.
        Console.WriteLine("Генерація RefToMyCar: {0}",
GC.GetGeneration(refToMyCar));
 }
```

Найчастіше члени типу System.GC задіюють тоді, коли створюються типи, що використовують некеровані *ресурси.*

Активізація збирання сміття

Отже, збиральник сміття в .NET покликаний управляти пам'яттю за вас. Проте в окремих випадках буває вигідно програмно активізувати початок збирання сміття, використовуючи GC.Collect(), а саме:

перед входом застосунка в блок програмного коду, для якого небажано, щоб його виконання уривалося можливим збиранням сміття;

після закінчення розміщення дуже великого числа об'єктів, коли бажано вивільнити якомога більше пам'яті.

Якщо вигідніше виконати збирання сміття, то можна явно почати цей процес таким чином:

```
static void Main (string[] args) {
// Активізація збірки сміття і
// очікування завершення фіналізації об'єктів.
GC.Collect ();
GC. WaitForPendingFinalizers () ;
...
}
```

Для безпосередньої активізації збирання сміття необхідно викликати GC.WaitForPendingFinalizers(). Цей підхід забезпечує, що всі об'єкти, які передбачають фіналізацію, виконають необхідні завершальні процедури, перш ніж програма продовжить свою роботу. Методу GC.Collect () можна передати числове значення, яке вказує стару генерацію, для якої треба виконати збирання сміття. Наприклад, щоб повідомити CLR, що слід розглянути тільки об'єкти генерації 0, необхідно надрукувати:

```
static void Main(string [] args)
{
// Розглянути тільки об'єкти генерації 0.
GC.Collect(0);
GC.WaitForPendingFinalizers();
}
```

Подібно до будь-якого збирання сміття, виклик GC.Collect () підвищить статус генерацій, що вижили.

Питання для самопідготовки

1. Принципи об'єктно-орієнтованого підходу.
2. Основні властивості конструктора.
3. Конструктор за замовчуванням.
4. "Збирання сміття".

Запитання для самодіагностики

1. Який синтаксис опису класу?
2. Які специфікатори доступу до елементів класу є в мові програмування?
3. Який порядок ініціалізації об'єкта класу?

Література: [2; 4; 13; 16; 17; 18; 23; 24].

5. Повторне використання класів

Мета теми: набуття знань щодо варіантів повторного використання класів і відповідних відношень між класами та об'єктами.

Професійні компетентності: здатність використовувати відношення агрегації, композиції та спадкування у програмах.

Основні питання

5.1. Поняття про асоціацію. Відношення композиції та агрегації як види асоціації. Реалізація композиції та агрегації в C# і Java.

5.2. Відношення успадкування. Реалізація відношення успадкування в C# і Java. Ініціалізація об'єкта базового класу. Використання конструкторів під час успадкування. Варіанти використання успадкування. Перевизначення методів.

5.3. Раннє та пізнє зв'язування. Віртуальні методи. Реалізація принципу поліморфізму в C# і Java. Рядкове подання об'єкта. Абстрактні класи та методи. Реалізація поліморфної поведінки на базі абстрактного класу. Інтерфейси. Реалізація поліморфної поведінки на базі інтерфейсу.

Питання для опрацювання: поняття про асоціацію, відношення успадкування, раннє та пізнє зв'язування, віртуальні методи, реалізація принципу поліморфізму в C# і Java.

Ключові слова: асоціація, агрегація, композиція, успадкування, предок, нащадок, суперклас, підклас, базовий клас, похідний клас, абстрактний клас, віртуальні методи, поліморфізм, інтерфейс.

5.1. Поняття про асоціацію. Відношення композиції та агрегації як види асоціації. Реалізація композиції та агрегації в C# і Java

Між класами або об'єктами можуть існувати різні логічні відношення.

Логічним відношенням між об'єктами, що часто зустрічається, є **асоціація**. Асоціація показує, що об'єкти різних класів пов'язані між собою. Якщо між двома класами встановлено зв'язок, то можна переміщатися від об'єктів одного класу до об'єктів іншого. Асоціація, що зв'язує два класи, називається *бінарною*. Часто в моделюванні необхідно вказати, скільки об'єктів можна зв'язати за допомогою одного екземпляра асоціації. Це число називають *кратністю ролі асоціації*. Кратність, зазначена на одному з кінців асоціації, позначає, що на цьому кінці саме стільки об'єктів повинно

відповідати кожному об'єкту на протилежному кінці. Кратність можна задати дорівненою одиниці (1), можна вказати діапазон: "нуль або одиниця" (0..1), "багато" (0 .. *), "одиниця або більше" (1 .. *) і т.д. Асоціації може бути присвоєно ім'я, яке описує природу відношень. Приклад відношень асоціації показаний на рис. 5.1.

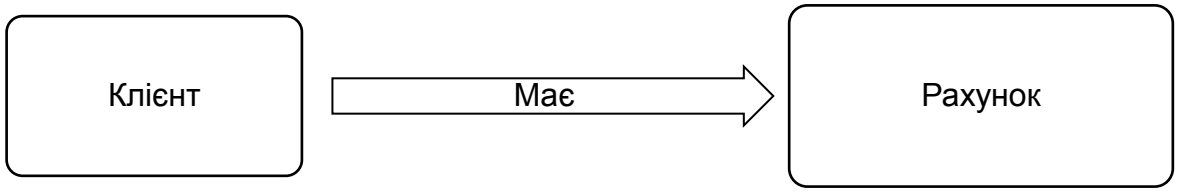

Рис. 5.1. **Приклад відношень асоціації**

Проста асоціація відображає структурне відношення між рівноправними сутностями, коли обидва класи перебувають на одному концептуальному рівні та жоден з них не є більш важливим, ніж інший.

Однак часто доводиться моделювати відношення типу "частина/ціле", в якому один з класів має більш високий ранг (ціле) та складається з декількох менших за рангом (частин). Відношення такого типу називають *агрегацією*; воно є окремим випадком асоціації і зображується у вигляді простої асоціації з не зафарбованим ромбом збоку "цілого".

Агрегація може означати фізичне входження одного об'єкта в інший, коли об'єкт-ціле не може існувати без об'єктів-частин. Наприклад, літак складається з крил, двигунів, шасі та інших частин. Тоді це сильніший варіант відношень агрегації – *композиція*. Композиція зображується у вигляді простої асоціації з зафарбовані ромбом збоку "цілого".

Відмінності між композицією і агрегацією

Наприклад, відношення між об'єктами класів "Університет", "Студент" і "Факультет" дещо відрізняються, хоча обидва є відношеннями агрегування. Відношення між об'єктами класів "Університет" і "Факультет" – це композиція, оскільки зі знищенням об'єкта "Університет" об'єкти факультетів, що належать цьому університету, також повинні бути знищені. Об'єкти класів "Студент" і "Університет" перебувають у відношенні агрегації тому, що об'єкт класу "Студент" може існувати після знищення цього університету, позаяк студент може вчитися в декількох навчальних закладах одночасно.

Агрегація (композиція) в Java та C#

Об'єкт, який є атрибутом іншого об'єкта (агрегату), має зв'язок зі своїм агрегатом. Через цей зв'язок агрегат може посилати йому повідомлення.

Синтаксично агрегація (композиція) – це використання посилання на об'єкт одного класу в якості поля іншого класу.

Приклад 1. Розглянемо приклад реалізації відношення агрегації (один об'єкт класу AppleList може містити кілька об'єктів класу Apple) мовою Java. У цьому прикладі об'єкти класу Apple створюються за межами класу AppleList, тобто об'єкти класу Apple можуть існувати і після знищення об'єкта класу AppleList.

```
class AppleList
{
      private Apple[ ] lst;

      //Конструктор
      public AppleList(int n)
      {
         lst = new Apple[n];
      }

      public void addApple(Apple apl, int index)
      {
         lst[index] = apl;
      }
      private double calculateAverageVolume()
      {
         double sum = 0;
         for (int i = 0; i < lst.length; i++)
            sum += lst[i].calculateVolume();
         return (sum/lst.length);
      }

      public void printTable()
      {
         System.out.println("Колір\tСорт\tДіаметр\tОб'єм");
         foreach (Apple A in lst)
         {
System.out.format("%1$\t%2$\t%3$\t%4$10.4f}", A.getSort(), A.getColor(),
A.getDiameter(), A.calculateVolume());
         }
```

```
            System.out.format("\nРазом:\t\t\t%1$6.2f}",
calculateAverageVolume());
        }
    }
    class Program
    {
        public static void main(String[ ] args)
        {
           java.util.Scanner sc = new java.util.Scanner(System.in);
            System.out.print("Введіть кількість яблук: ");
            int n = sc.nextInt();
            AppleList vedomost = new AppleList(n);
            String sort, color;
            double diameter;
     Apple apl;
            for (int i = 0; i < n; i++)
            {
               System.out.print("Введіть сорт яблука: ");
               sort = sc.next();
               System.out.print ("Введіть колір яблука: ");
               color = sc.next();
               System.out.print("Введіть діаметр яблука: ");
               diameter = sc.nextDouble();
               System.out.println();
        apl = new Apple(sort, color, diameter);
               vedomost.addApple(apl, i);
            }
     sc.close();
            System.out.println();
            vedomost.printTable();
            System.out.println();
        }
    }
```

Приклад 2. Розглянемо приклад реалізації відношення композиції (один об'єкт класу AppleList може містити кілька об'єктів класу Apple) мовою C#.

```
    class AppleList
    {
        private Apple [ ] lst;

        // Конструктор
        public AppleList (int n)
```

```
        {
            lst = new Apple [n];
        }

        public void AddApple (string sort, string color, double diameter, int
index)
        {
            lst [index] = new Apple (sort, color, diameter);
        }
        public double CalculateAverageVolume ()
        {
            double sum = 0;
            for (int i = 0; i <lst.Length; i ++)
                sum + = lst [i] .CalculateVolume ();
            return (sum / lst.Length);
        }

        public void PrintTable ()
        {
            Console.WriteLine ( "Колір\tСорт\tДіаметр\tОб'єм");
            foreach (Apple A in lst)
            {
                Console.WriteLine ( "{0}\t{1}\t{2}\t{3,10: f4}", A.GetSort (), A.GetColor
(), A.GetDiameter (), A.CalculateVolume ());
            }
           Console.WriteLine ( "\nРазом:\t\t\t{0,6: f2}", CalculateAverageVolume ());
        }
    }

    class Program
    {
        static void Main (string [] args)
        {
            Console.Write ( "Введіть кількість яблук:");
            int n = int.Parse (Console.ReadLine ());
            AppleList Vedomost = new AppleList (n);
            string sort, color;
            double diameter;
            for (int i = 0; i <n; i ++)
            {
                Console.Write ( "Введіть сорт яблука:");
                sort = Console.ReadLine ();
                Console.Write ( "Введіть колір яблука:");
                color = Console.ReadLine ();
```

```
            Console.Write ( "Введіть діаметр яблука:");
            diameter = double.Parse (Console.ReadLine ());
            Console.WriteLine ();
            Vedomost.AddApple (sort, color, diameter, i);
          }
          Console.WriteLine ();
          Vedomost.PrintTable ();
          Console.WriteLine ();
        }
    }
```

У цьому прикладі об'єкти класу Apple створюються всередині класу AppleList, тобто об'єкти класу Apple можуть існувати лише доти, доки існує об'єкт класу AppleList.

5.2. Відношення успадкування. Реалізація відношення успадкування в C# і Java. Ініціалізація об'єкта базового класу. Використання конструкторів під час успадкування. Варіанти використання успадкування. Перевизначення методів

Успадкування є одним з основних понять об'єктно-орієнтованого програмування. Воно дозволяє одному класу використовувати елементи іншого, "спорідненого" класу, не визначаючи їх знову. Отже, програма буде містити менше коду.

Для опису класів, які зв'язані відношенням успадкування, існують певні *варіанти термінології*: предок та нащадок; суперклас і підклас; базовий клас і похідний клас.

Відношення успадкування має деякі *особливості*:

похідні класи автоматично успадковують від базового класу всі відкриті (public) поля і методи;

через об'єкт похідного класу не можна безпосередньо звернутися до закритих (private) полів, визначених у базовому класі, за їхніми іменами;

конструктори не успадковуються, тому похідний клас має визначати власні конструктори;

після створення об'єкта похідного класу з його конструктора спочатку викликається конструктор базового класу, а тільки після цього виконується код конструктора цього (похідного) класу.

На UML-діаграмі класів відношення успадкування зображується лінією зі стрілкою, яка завжди спрямована на базовий клас. Це показує,

що завжди можна стверджувати, що похідний клас, як мінімум, такий самий, як базовий клас. Наприклад, автобус (похідне поняття) – це вид транспортного засобу (базове поняття).

Приклад відношення успадкування наведено на рис. 5.2.

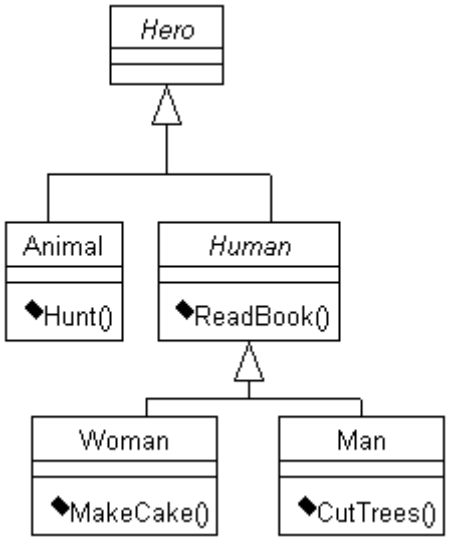

Рис. 5.2. **UML-діаграма класів**

Тут усі герої казки є нащадками класу Hero. Людина (клас Human) уміє читати книги, тварина (клас Animal) уміє полювати, жінка (клас Woman) – читати книги та пекти пироги, а чоловік (клас Man) – читати книги та рубати дерева.

Реалізація діаграми успадкування мовами C# і Java наведена в табл. 5.1.

Таблиця 5.1

Діаграми успадкування мовами C# та Java

C#	Java
class Hero { } class Animal : Hero { public void Hunt() { } } class Human : Hero { public void ReadBook() { } } class Woman : Human { public void MakeCake() { } } class Man : Human { public void CutTrees() { }	class Hero { } class Animal *extends* Hero { public void hunt() { } } class Human *extends* Hero { public void readBook() { } } class Woman *extends* Human { public void makeCake() { } } class Man *extends* Human { public void cutTrees() { } }

У табл. 5.1 наведені приклади успадкування на мовах C# та Java.

Використання конструкторів під час успадкування

Мова C#

У всіх конструкторах C#, крім *System.Object,* конструктори базового класу викликаються прямо перед виконанням першого рядка конструктора. Ці ініціалізатори конструкторів дозволяють задавати клас і конструктор, що підлягає виклику. Вони бувають двох видів:

ініціалізатор у вигляді *base(...)* – активізує конструктор базового класу поточного класу;

ініціалізатор у вигляді *this(...)* – дозволяє поточному базовому класу викликати інший конструктор. Це корисно, якщо ви надміру завантажили конструктори, проте маєте переконатись, що завжди викликатиметься конструктор за замовчуванням. *Перевантаженими* називають два та більше методи з однаковим ім'ям, але з різними списками аргументів.

Щоб побачити порядок подій у дії, зверніть увагу на код, який спочатку виконає конструктор класу A, а потім – конструктор класу B:

```
class A
  {
    public A()
    {
      Console.WriteLine("A");
    }
  }
class B : A
  {
    public B()
    {
      Console.WriteLine("B");
    }
  }
class DefaultInitializerApp
  {
    public static void Main()
    {
      B b = new B();
    }
  }
```

Цей код – функціональний еквівалент наступного, де є явний виклик конструктора базового класу:

```
class A
   {
      public A()
      {
         Console.WriteLine("A");
      }
   }
class B : A
   {
      public B()
         : base()
      {
         Console.WriteLine("B");
      }
   }
class BaseDefaultInitializerApp  {
      public static void Main()
      {
         B b = new B();
      }
   }
```

А тепер розглянемо більш вдалий приклад ситуації, коли вигідно використовувати ініціалізатори конструкторів. Розглянемо знову два класи: A і B. Цього разу в класі A два конструктори, один з них не вимагає аргументів, а інший приймає аргумент типу *int.* У класі B один конструктор, що приймає аргумент типу *int.* Зі створенням класу B виникає проблема. Якщо запустити наступний код, буде викликаний конструктор класу A, який не приймає аргументів:

```
class A
   {
      public A()
      {
         Console.WriteLine("A");
      }
      public A(int foo)
      {
         Console.WriteLine("A = {0}", foo);
```

```
    }
  }

class B : A
  {
    public B(int foo)
    {
      Console.WriteLine("B = {0}", foo);
    }
  }
class DerivedInitializer2App
  {
    public static void Main()
    {
      B b = new B(42);
    }
  }
```

Як же гарантувати, що буде викликаний саме потрібний конструктор класу A? Необхідно явно вказати компіляторові, який конструктор в ініціалізаторі повинен бути викликаний першим; наприклад, так:

```
class A
  {
    public A()
    {
      Console.WriteLine("A");
    }
    public A(int foo)
    {
      Console.WriteLine("A = {0}", foo);
    }
  }

class B : A
  {
    public B(int foo): base(foo)
    {
      Console.WriteLine("B = {0}", foo);
    }
  }
class DerivedInitializer2App  {
    public static void Main()   {
```

```
        B b = new B(42);
      }
    }
```

Мова Java

Розглянемо випадок використання конструкторів без параметрів.

Припустимо, що є ієрархія успадкування Art (мистецтво) – базовий клас, похідні класи VisualArt (образотворче мистецтво) та Painting (малювання фарбами). Тоді маємо такий вихідний код:

```
class Art {
  public Art () { System.out.println("Art");  }
}
class VisualArt extends Art {
  public VisualArt() { System.out.println("VisualArt"); }
}
public class Painting extends VisualArt {
  public Painting() {System.out.println("Painting"); }
  public static void main(String[ ] args) {
    Painting x = new Painting();
  }
}
```

У цьому прикладі після створення об'єкта класу Painting викликається його конструктор. У ньому спочатку автоматично викликається конструктор найближчого базового класу (тобто конструктор класу VisualArt). З конструктора класу VisualArt спочатку автоматично викликається конструктор найближчого базового класу (тобто конструктор класу Art). Створюється така послідовність викликів конструкторів:

Painting() → VisualArt() → Art().

У результаті виконання програми на консоль буде виведено три текстових повідомлення, кожне з яких починається з нового рядка:

```
Art
VisualArt
Painting
```

Якщо конструктори базових класів мають параметри, концептуально відбувається те саме, що і в разі конструкторів без параметрів. Конструк-

тор найближчого базового класу необхідно явно викликати за допомогою ключового слова super:

```
class Box {
   private double width;
   private double height;
   private double depth;
   public Box(double width, double height, double depth) {
      this.width = width;
      this.height = height;
      this.depth = depth;
   }
class WeightBox extends Box {
  private double weight;
  public WeightBox(double width, double height, double depth, double weight)
{   //   Виклик конструктора найближчого базового класу
    super (width, height, depth);
    this.weight = weight;
  }
}
public class BoxManager {
  public static void main(String[ ] args) {
    WeightBox box = new WeightBox(2.0, 3.0, 5.0, 125.5);
  }
}
```

У конструкторі класу WeightBox явно викликає конструктор класу пращура Box.

Головні варіанти використання успадкування

Це питання розглянемо з використанням фрагментів програм мовою Java. Реалізація мовою C# є аналогічною.

Приклад 3. "B – це A".

```
class A {
  private int field;
  public void f() { }
}
class B extends A { }
```

Цей варіант можна використовувати для клонування базового класу.

Приклад 4. "B схожий на A".

```
class A {
        private int field1;
        public void f() { … }
}
class B extends A {
  private int field2;
        public void g() { … }
}
```

Цей варіант є основним. У цьому прикладі об'єкт класу B має два метода: власний метод g та успадкований – f.

Приклад 5. Перевантаження метода базового класу.

```
class A {      // базовый класс
        private int field1;
        public void f() { … }
}
class B extends A {
        private int field2;
        public void f(int argument) { … }
}
```

Тут реалізація методу f, яка викликається для об'єкта похідного класу B, залежить від наявності його аргумента. Тобто метод з одним параметром викликається з класу B, а метод без параметрів – з базового класу A.

Приклад 6. Перевизначення метода базового класу.

```
class A
  private int field1;
  public void f() { //Алгоритм 1 }
}
class B extends A {
  private int field2;
  public void f() { //Алгоритм 2  }
}
```

Цей варіант є основою для реалізації принципа поліморфізму. Використовується для динамічного вибору варіанту методу, який буде викликано під час виконання. Рішення залежить від типу посилання та типу об'єкта, яким було ініціалізоване це посилання.

5.3. Раннє та пізнє зв'язування. Віртуальні методи. Реалізація принципу поліморфізму в C# і Java. Рядкове подання об'єкта. Абстрактні класи та методи. Реалізація поліморфної поведінки на базі абстрактного класу. Інтерфейси. Реалізація поліморфної поведінки на базі інтерфейсу

Поліморфізм (polymorphism) (від грецького polymorphos) – це властивість, яка дозволяє те саме ім'я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об'єктно-орієнтованого програмування, є використання одного імені для задання загальних для класу дій. Виконання кожної конкретної дії визначатиметься типом даних.

У більш загальному сенсі концепцією поліморфізму є ідея "один інтерфейс – безліч методів". Це означає, що можна створити загальний інтерфейс для групи близьких за сенсом дій. Перевагою поліморфізму є те, що він допомагає усунути складність програм, використовуючи той самий інтерфейс для задання єдиного класу дій. Вибір же конкретної дії, залежно від ситуації, покладається на компілятор. Програмістові не потрібно робити цей вибір самому. Слід тільки запам'ятовувати та використовувати загальний інтерфейс.

Поліморфізм може застосовуватися також і до операторів. Фактично поліморфізм обмежено застосовується у всіх мовах програмування (наприклад, в арифметичних операторах).

Ключовим у розумінні поліморфізму є те, що він дозволяє маніпулювати об'єктами різного ступеня складності шляхом створення загального для них стандартного інтерфейсу для реалізації схожих дій.

Основні поняття. Поняття поліморфізму

Поліморфізм є найістотнішою властивістю об'єктно-орієнтованого програмування.

Поліморфізм, за Г. Бучем, полягає в тому, що імена можуть відповідати різним класам об'єктів, що входять в один суперклас. Отже, один об'єкт, відмічений таким ім'ям, може по-різному реагувати на деяку безліч дій.

Приклад 7. Ілюстрацією може слугувати така конструкція:

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

```
using System.Text;

namespace Sample1_1
{
   class Starter
   {
      [STAThread]
      static void Main(string[] args)
      {
         OneWordWriter w1 = new OneWordWriter();
         TwoWordsWriter w2 = new TwoWordsWriter();
         ThreeWordsWriter w3 = new ThreeWordsWriter();
         w1.Write();
         ((OneWordWriter)w3).Write();
         // вывод Three words written
         ((OneWordWriter)w2).Write();
         // вывод Two words
      }
   }
   class OneWordWriter
   {
      public virtual void Write()
      {
         Console.WriteLine("One");
      }
   }

   class TwoWordsWriter : OneWordWriter
   {
      public override void Write()
      {
         Console.WriteLine("Two words");
      }
   }

   class ThreeWordsWriter : TwoWordsWriter
   {
      public override void Write()
      {
         Console.WriteLine("Three words written");
      }
   }
}
```

У розглянутому прикладі створені три класи, "суперклас" OneWordWriter і підкласи TwoWordsWriter і ThreeWordsWriter, кожний з яких має метод Write(). Навіть після приведення відповідних об'єктів у методі Main до "суперкласу" OneWordWriter кожний з них виконує виклик свого методу Write().

Статичне та динамічне зв'язування

Під поняттям "**поліморфізм**" розуміють здатність програми виконувати методи об'єктів у точній відповідності до конкретного типу об'єкта. Наприклад, метод "Їхати" об'єкта – екземпляра класу "Запорожець" напевно відрізняється від методу "Їхати" об'єкта класу "Мерседес". Водночас імена таких методів співпадають, що подеколи заплутує розробників.

Крім того, в змінній об'єктного типу (класу) допускається зберігання реального об'єкта іншого типу. Ці типи деяким чином позв'язані за ієрархією спадковості. Наприклад:

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

namespace Sample2
{
   class Program
   {
     class Car
     {
     public double Benzin; // бензин
     void Go( float dist )
     {
      Benzin -= dist / 10; // 1 літр бензину на 10 км
      Rasst += dist; // проехали
      }
      protected double Rasst; // пройденна відстань
     }

  class Truck : Car
  {
    public void Go( float dist )
    {
      Benzin -= dist / 5; // 1 літр бензину на 5 км
      Rasst += dist; // проїхали
    }
  }
```

```
        static void Main(string[] args)
        {
            Truck truct = new Truck();
            Car car2 = new Car();
            //Car car1 = new Truck();
            truct.Go(100);
         }
    }
}
```

Стандартні такі оголошення об'єктів:
Truck truct = new Truck();
Car car2 = new Car();
Однак принцип поліморфізму допускає і такий запис:
Car car1 = new Truck();

Які конкретно версії методу Go() будуть викликані – класу Car або Truck? Це визначає принцип поліморфізму. Відповідно до нього в мові C# визначені так звані статичне зв'язування і динамічне зв'язування.

Статичне зв'язування – це зв'язування методу з батьківським об'єктом на етапі компіляції. Оскільки компілятор не знає, як конкретно працюватиме програма, він завжди виходить з відомого і явно заданого типу об'єкта-власника методу. Якщо використовується змінна car класу Car, то буде викликаний метод Go() класу Car, хоча фізично в змінній car зберігається екземпляр класу Truck. Так само й у випадку із змінною truck – хоча ми примусово записали в неї посилання на об'єкт класу Car, компілятор про це не знає і виходить з типу змінної truck – викликає метод Go() класу Truck.

Динамічне зв'язування – це зв'язування методу з об'єктом, яке відбувається під час роботи програми. Для цього потрібний метод визначається вже не типом змінної-власника, а реальним типом об'єкта, посилання на який вона зберігає. Але таке зв'язування можливе тільки для так званих *віртуальних методів* – розділення типів методів потрібне, щоб підказати компілятору, де яке зв'язування реалізовувати.

Віртуальні функції

Базові класи можуть визначати та реалізовувати віртуальні методи, а похідні класи – перевизначати їх. Це означає, що вони надають свої власні визначення і реалізацію. Під час виконання, коли клієнтський код

викликає метод, середовище CLR шукає тип часу виконання об'єкта та викликає це перевизначення віртуального методу. Таким чином, у початковому коді можна викликати метод у базовому класі та викликати виконання методу з версією похідного класу.

Віртуальні методи дозволяють єдиним чином працювати з групами зв'язаних об'єктів. Наприклад, є застосунок малювання, яке дає можливість користувачу створювати на поверхні для малювання різні форми. Під час компіляції невідомо, які конкретні форми створюватиме користувач. Але застосунок повинен ураховувати всі різні типи створюваних форм і оновлювати їх у відповідь на дії миші користувача. Поліморфізм можна використовувати для вирішення цієї проблеми в два основних етапи:

1) створення ієрархії класів, в якій клас кожної конкретної форми проводиться від загального базового класу;

2) використання віртуального методу для виклику відповідного методу в деякому похідному класі одним викликом методу базового класу.

Приклад 8. Розглядувані процедури можна отримати так:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Sample3
{
   public class Shape
   {
      // Virtual method
      public virtual void Draw()
      {
         Console.WriteLine("Performing base class drawing tasks");
      }
   }
   class Circle : Shape
   {
      public override void Draw()
      {
         // Code to draw a circle...
         Console.WriteLine("Drawing a circle");
         base.Draw();
      }
   }
   class Rectangle : Shape
   {
      public override void Draw()
```

```
        {
            // Code to draw a rectangle...
            Console.WriteLine("Drawing a rectangle");
            base.Draw();
        }
    }
    class Triangle : Shape
    {
        public override void Draw()
        {
            // Code to draw a triangle...
            Console.WriteLine("Drawing a triangle");
            base.Draw();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            System.Collections.Generic.List<Shape> shapes = new
System.Collections.Generic.List<Shape>();
            shapes.Add(new Rectangle());
            shapes.Add(new Triangle());
            shapes.Add(new Circle());

            foreach (Shape s in shapes)
            {
                s.Draw();
            }
        }
    }
}
```

У цьому прикладі видно: в оновленні поверхні малювання використовується цикл foreach для ітерації списку та виклику методу Draw на кожному об'єкті Shape в списку. Хоча кожен об'єкт в списку має оголошений тип Shape, викликатиметься саме тип часу виконання (перевизначена версія методу в кожному похідному класі).

Запобігання перевизначенню віртуальних членів похідними класами

Віртуальні члени залишаються віртуальними без будь-яких обмежень відносно кількості класів, оголошених між віртуальним членом і класом, у якому він був спочатку оголошений.

Приклад 9. У класі A оголошується віртуальний член, клас B є похідним від класу A, а клас C є похідним від класу B. Тоді клас C успадковує віртуальний член і забезпечує можливість його перевизначення незалежно від того, чи було в класі B оголошене перевизначення цього члена.

Похідний клас може припинити спадковість, оголосивши перевизначення як sealed. Для цього потрібно додати ключове слово sealed перед ключовим словом override в оголошенні члена класу:

```
public class C : B
{
   public sealed override void DoWork() { }
}
```

Тобто метод DoWork більше не є віртуальним для будь-якого похідного від класу C класу. Він залишається віртуальним для екземплярів класу C, навіть якщо вони приведені до типу B або A.

Приклад 10. Закриті методи можна заміщати в похідних класах за допомогою ключового слова new:

```
public class D : C
{
   public new void DoWork() { }
}
```

У цьому випадку викликанням методу DoWork для типу D з використанням змінної типу D викликається новий метод DoWork. Якщо для доступу до екземпляра типу D використовується змінна типу C, B або A, то виклик методу DoWork підкорятиметься правилам віртуального спадкоємства, скеровуючи виклики реалізації методу DoWork у клас C.

У мові Java в такому контексті використовується ключове слово *final*.

Поліморфізм у мові Java

Реалізація поліморфної поведінки в Java аналогічна розглянутої для C#, проте з певними відмінностями. Головними з них є відсутність ключових слів, подібних *virtual* та *override* у C#. Наприклад:

```
class Shape { public void draw() { } }

class Circle extends Shape {
  public void draw() { System.out.println("Circle.draw()"); }
}
```

```
class Square extends Shape {
  puvoid draw() { System.out.println("Square.draw()"); }
  public static void main(String[ ] args) {
     Shape s;
     s = new Circle(); s. draw(); // викликається метод draw() з класу Circle
     s = new Square(); s. draw(); // викликається метод draw() з класу Square
  }
}
```

Методи Java є "віртуальними" за замовчуванням.

Рядкове подання об'єкта

Часто виникає необхідність перетворення об'єкта у деякий рядок, що містить контент щодо поточного стану об'єкта. Нагадаємо, що стан об'єкта визначається поточними значеннями полів класу, екземпляром якого він є.

Приклад 11. Розглянемо приклад виведення об'єкта на консоль мовою C#:

```
namespace Lecture3 {
   class Student {
          private int Age = 18;
          private string LastName = "Петренко";
          public static void Main() {
                 Student Obj = new Student();
                 Console.WriteLine(Obj);
                }
  }
}
```

Результат роботи програми буде таким:

Lecture3. Student

Чому це відбувається?

Відомо, що будь-який клас у C# неявно є нащадком класу System.Object, визначеного в .NET.Framework. Цей клас має віртуальний метод ToString, який повертає рядкове подання об'єкта. Таким поданням є повне ім'я його класу, тобто сукупність назви простору імен та імені класу, розділених крапкою.

Якщо потрібна власна реалізація методу ToString, яка повертає рядок у бажаному форматі, необхідно перевизначити цей метод. Тоді попередня програма буде мати такий вигляд:

```
namespace Lecture3 {
  class Student {
    private int Age = 18;
    private string LastName = "Петренко";
     public override string ToString() {
         return "Прізвище: " + LastName + ",   вік: " + Age;
     }
     public static void Main() {
        Student Obj = new Student();
        Console.WriteLine(Obj);
     }
  }
}
```

У результаті маємо на консолі:
Прізвище: Петренко, вік: 18

У мові Java розглянуті механізми є аналогічними з деякими відмінностями. Програма, що видає бажане рядкове подання об'єкта, може мати такий вигляд:

```
package tostringtest;
public class Student {
  private int age = 18;
  private String lastName = "Петренко";
  public String toString() {
    return "Прізвище: " + lastName + ",   вік: " + age; }
  public static void main(String[ ] s) {
    Student obj = new Student();
    System.out.println(obj);
  }
}
```

Зокрема, за замовчуванням метод toString класу java.lang.Object повертає рядок в форматі:
<ім'я пакету>.<ім'я класу>@<хеш-код об'єкта>

Абстрактні класи

Із успадкуванням тісно пов'язаний ще один важливий механізм проектування сімейств класів – механізм **абстрактних класів**.

Клас є *абстрактним*, якщо він має хоч би один абстрактний метод.

Метод називають *абстрактним*, якщо під час визначення методу задана його сигнатура, але не задана реалізація методу.

Абстрактні класи мають ряд *властивостей*:

такі класи не можуть бути sealed;

вони не можуть бути інстанційовані оператором new;

оголошення методу класу абстрактним означає, що він віртуальний; тому метод не може бути оголошений як virtual;

усі абстрактні члени повинні бути визначені в класі, що успадковує, за виключення випадку, коли клас-спадкоємець є теж абстрактний;

абстрактні методи класу не можуть бути оголошені як static.

У ході реалізації абстрактним класом інтерфейсу всі його члени повинні бути або реалізованими, або описані їх сигнатури й оголошені як abstract.

Абстрактний клас може мати як абстрактні, так і неабстрактні члени. Обмежень на кількість абстрактних членів немає, тобто їх може не бути зовсім, але тоді сенс оголошення класу абстрактним втрачається.

Оголошені абстрактними можуть бути методи, властивості, індексатори та події.

Абстрактний клас може реалізовувати декілька інтерфейсів, бути спадкоємцем класу та перевизначати віртуальні методи неабстрактного класу.

Клас, що успадковує під час перевизначення абстрактної властивості, зобов'язаний реалізовувати одного з аксесорів. Якщо перевизначається абстрактна властивість з тільки ним аксесором, то перевизначатися може тільки він. Якщо абстрактну властивість мають обидва аксесори, то перевизначати може як один, так і обидва.

Оголошення абстрактних методів і абстрактних класів повинне супроводжуватися модифікатором abstract. Оскільки абстрактні класи не є повністю певними класами, то не можна створювати об'єкти абстрактних класів. Абстрактні класи можуть мати нащадків, що частково або повністю реалізовують абстрактні методи батьківського класу. Абстрактний метод розглядається переважно як віртуальний, що перевизначається нащадком; тому до нього застосовується *стратегія динамічного скріплення*.

Абстрактні класи є одним з найважливіших інструментів об'єктно-орієнтованого проектування класів. У основу будь-якого класу закладена абстракція даних. Абстрактний клас описує цю абстракцію, не входячи в деталі реалізації та обмежуючись описом тих операцій, які можна виконувати

над даними класу. Так, проектування абстрактного класу Stack, що описує стек, може складатися з розгляду основних операцій над стеком і не визначати, як буде реалізований стек – списком або масивом. Два нащадки абстрактного класу – ArrayStack і ListStack – можуть бути вже конкретними класами, заснованими на різних представленнях стека.

Повний опис абстрактного класу Stack виглядає так:

```
public abstract class Stack
    {
       public Stack()
       { }
       public abstract void put(int item);
       public abstract void remove();
       public abstract int item();
       public abstract bool IsEmpty();
    }
```

Опис класу містить тільки сигнатури методів класу і їх специфікацію, задану тегами <summary>.

Приклад 12. Побудуємо один із нащадків класу, реалізація якого заснована на списковому поданні. Клас ListStack буде нащадком абстрактного класу Stack і клієнтом класу Linkable, що задає елементи списку.

Клас Linkable виглядає таким чином:

```
public class Linkable
    {
       public Linkable()
       { }
       public int info;
       public Linkable next;
    }
```

У ньому – два поля і конструктор за замовчуванням. Побудуємо тепер клас ListStack:

```
public class ListStack : Stack
    {
       public ListStack()
       {
          top = new Linkable();
       }
```

```
        Linkable top;
        public override void put(int item)
        {
           Linkable newitem = new Linkable();
           newitem.info = item;
           newitem.next = top;
           top = newitem;
        }
        public override void remove()
        {
           top = top.next;
        }
        public override int item()
        {
           return (top.info);
        }
        public override bool IsEmpty()
        {
           return (top.next == null);
        }
  }
```

Клас має одне поле top класу Linkable та методи, успадковані від абстрактного класу Stack.

Приклад 13. Розглянемо приклад роботи із стеком:

```
  static void Main(string[] args)
     {
        ListStack stack = new ListStack();
        stack.put(7); stack.put(9);
        Console.WriteLine(stack.item());
        stack.remove(); Console.WriteLine(stack.item());
        stack.put(11); stack.put(13);
        Console.WriteLine(stack.item());
        stack.remove(); Console.WriteLine(stack.item());
        if (!stack.IsEmpty()) stack.remove();
        Console.WriteLine(stack.item());
     }
```

***Зауваження*.** Створення об'єкта базового класу, подібного Shape (тобто якоїсь абстрактної геометричної фігури), не має особливого сенсу, оскільки не зрозуміло, як її малювати. Тому доцільно заборонити створення об'єктів таких класів. Це є однією з головних особливостей абстрактних класів.

Приклад 14. Розглянемо приклад реалізації поліморфної поведінки з абстрактним класом у мові Java:

```
abstract class Shape { abstract void draw(); }

class Circle extends Shape {
  void draw() { System.out.println("Circle.draw()"); }
}

class Square extends Shape {
  void draw() { System.out.println("Square.draw()"); }
  public static void main(String[ ] args) {
     //Shape s = new Shape();  // Помилка, створення об'єкту заборонено!
     Shape s; // Немає помилки
     s = new Circle(); s. draw(); // викликається draw() з Circle
     s = new Square(); s. draw(); // викликається draw() з Square
  }
}
```

У прикладі 14 показана реалізація принципу поліморфної поведінки на базі абстрактного класу **Shape.**

Інтерфейси

Поліморфізм може бути реалізований за допомогою **інтерфейсів**.

Властивості інтерфейсів:

клас може реалізовувати більше одного інтерфейсу;

інтерфейс не містить реалізації методів;

метод класу, що реалізує метод інтерфейсу, повинен бути не статичним, оголошеним із специфікатором доступу public, мати таку ж саму сигнатуру;

клас, що реалізовує інтерфейс, може не дотримуватися політики реалізації аксесорів властивостей інтерфейсу, за винятком ситуації, коли інтерфейс оголошений як explicit;

інтерфейс не може інстанціюватися оператором new;

інтерфейси можуть містити методи, властивості, індексатори і події.

Приклад 15. Розглянемо приклад реалізації інтерфейсу:

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

```
namespace Sample5
{
   class Program
   {
      interface IFileTypesProvider
      {
         void Open();
         void Save();
      }
      class CFileNewtype : IFileTypesProvider
      {
         public void Open()
         {
            Console.WriteLine("Open method for Newtype File");
         }

         public void Save()
         {
            Console.WriteLine("Save method for Newtype File");
         }
      }

      class CFileOldtype : IFileTypesProvider
      {
         public void Open()
         {
            Console.WriteLine("Open method for Oldtype File");
         }

         public void Save()
         {
            Console.WriteLine("Save method for Oldtype File");
         }
      }
      static void Main(string[] args)
      {
         CFileNewtype type = new CFileNewtype();
         type.Open();
         type.Save();
      }
   }
}
```

Коли ж варто застосовувати реалізацію поліморфізму через абстрактні класи, а коли через інтерфейси?

Реалізація декількох інтерфейсів допомагає розв'язати проблему, навіть за відсутності множинного спадкоємства.

Абстрактні класи не використовують types для value. Таким чином, поліморфізм реалізується для структур тільки через інтерфейси.

Абстрактні класи можуть містити реалізацію внутрішніх, не абстрактних методів без порушення правил спадкоємства. Проте додавання нового абстрактного методу порушить усі зв'язки із спадкоємцями. У цьому випадку допоможе додавання нового інтерфейсу.

Інтерфейс – це чергова синтаксична конструкція, яка здатна спростити вихідний код, зробивши його більш зрозумілим і логічним. Простим прикладом можуть бути розетка та вилка. У розетки є свій інтерфейс – це два отвори певного діаметру, розташовані на певній відстані один від одного. Якщо треба, щоб якийсь пристрій живився від мережі, необхідно реалізувати цей інтерфейс. Результатом такої реалізації є вилка з двома ніжками. Якщо інтерфейс буде реалізований неправильно (наприклад, відстань між ніжками менша ніж потрібно), то це призведе до помилки (пристрій не зможе живитися від мережі).

Зауваження. Інтерфейси відповідають на запитання "Як це повинно працювати?" або "Як це повинно виглядати?". Це не рекомендації, а жорсткі правила, недотримання яких призведе до неправильної роботи. Інтерфейси дуже схожі на абстрактні класи, але вони мають відмінності та призначені для інших цілей. Інтерфейс не дає відповіді на запитання "Як робити?".

Приклад 16. Необхідно більш детально розглянути механізм реалізації інтерфейсу:

```
public interface Car
  {
    int Speed{get; set;}
    void GetInfo();
  }
```

Створено інтерфейс Car (машина). Зверніть увагу на ключове слово interface. Отже інтерфейс "говорить" про те, що в класі, який буде реалізовувати цей інтерфейс, має бути властивість Speed (швидкість) і метод GetInfo (отримати інформацію). Звісно, інтерфейс не містить інформації, яка б демонструвала те, як будуть працювати даний метод і властивість. Слід зауважити, що властивість Speed доступна як для запису, так і для читання. Якщо необхідно зробити властивість тільки для читання, потрібно

прибрати метод set. Для того щоб створити властивість тільки для запису, потрібно прибрати метод get.

Отже, створено інтерфейс машини (грубо кажучи). Зараз необхідно реалізувати цей інтерфейс і відповісти на запитання "Як це працює?". Для цього потрібний клас ferrari (за маркою італійського автомобіля). Цей клас повинен реалізовувати інтерфейс Car (оскільки "Феррарі" теж машина). Зробити це можна так:

```
public class ferrari: Car
  {
    private int spd; //швидкість
    public int Speed
    {
      get
      {
        return spd;
      }
      set
      {
        spd = value;
      }
    }
    public void GetInfo()
    {
      Console.WriteLine("Це суперкар Ferrari".);
    }
  }
```

Тепер є клас, який реалізує інтерфейс Car. Реалізація інтерфейсу за синтаксисом дуже схожа на спадкування; якщо не знати, що Car – це інтерфейс, їх можна переплутати. Обов'язково зверніть увагу на той факт, що клас повинен реалізовувати всі без виключення властивості та методи інтерфейсу. Якщо ви забудете реалізувати будь-який метод або властивість, то під час компіляції вам буде показана помилка; наприклад, така: 'ConsoleApplication7.ferrari' does not implement interface member 'ConsoleApplication7.Car.GetInfo ()'.

Один клас може реалізовувати декілька інтерфейсів. Це цілком логічно. Така машина, як "Феррарі" може бути одночасно як розкішшю, так і засобом пересування. Тому доцільно створити ще один інтерфейс Luxury (розкіш):

```
public interface Luxury
  {
```

```
    int Price{get; set;}
    void GetInfo();
  }
```

Тепер треба переписати клас Ferrari таким чином, щоб він реалізовував обидва інтерфейси:

```
public class ferrari: Car, Luxury
  {
    private int spd; //швидкість
    private int prc; //ціна

    public int Speed
    {
      get
      {
        return spd;
      }
      set
      {
        spd = value;
      }
    }

    public void GetInfo()
    {
      Console.WriteLine("Це суперкар Ferrari".);
    }

    public int Price
    {
      get
      {
        return prc;
      }
      set
      {
        prc = value;
      }
    }
  }
```

Тепер клас Ferrari реалізує обидва інтерфейси. Інтерфейсів може бути як завгодно багато, їх необхідно подавати через кому.

Зауваження. У обох інтерфейсів є метод GetInfo (). У цьому прикладі реалізували його лише один раз. З синтаксичної точки зору тут все правильно. Але найімовірніше розробник інтерфейсів Car і Luxor очікував іншої поведінки. Як же зробити так, щоб клас Ferrari по-різному реалізовував метод GetInfo () для кожного інтерфейсу? Для цього достатньо чітко вказати, який інтерфейс реалізує задіяний метод:

```
void Car.GetInfo()
    {
      Console.WriteLine("Це суперкар Ferrari ".);
    }


void Luxury.GetInfo()
    {
      Console.WriteLine("Це Ferrari це коштує багато грошей!".);
    }
```

Зверніть увагу на відсутність модифікатора доступу (public). У розглядуваному випадку його використовувати не можна.

Як же виконати потрібний метод? Щоб відповісти на це запитання, розглянемо оператор is. Оператор дозволяє нам у процесі роботи програми перевірити, чи реалізує даний об'єкт той чи інший інтерфейс. Ця інформація дуже корисна: знаючи, який інтерфейс реалізує клас, можна з впевненістю сказати, що він реалізує і методи, які цей інтерфейс надає.

Приклад 17. Наведемо приклад роботи з класом Ferrari:

```
static void Main(string[] args)
    {
      Ferrari c = new Ferrari ();
      c.Price = 1000000;
      c.Speed = 450;
      if (c is Car)
      {
        Car tmp = (Car)c;
        tmp.GetInfo();
      }
      if (c is Luxury)
      {
        Luxury tmp2 = (Luxury)c;
        tmp2.GetInfo();
      }
      Console.ReadLine();
    }
```

Насамперед слід створити екземпляр класу Ferrari. Після цього присвоїти значення властивостям щойно створеного об'єкта. Перевіряємо, чи реалізує об'єкт с інтерфейс Car. Оскільки інтерфейс Car реалізується, то ця умова поверне позитивне значення – виконати. Наступний рядок є дуже важливим. Створюємо посилання на інтерфейс Car. За допомогою круглих дужок наводимо об'єкт с до об'єкта Car (тобто перетворюємо). З об'єктом с нічого не відбувається, проте тепер у нас з'явилося посилання на інтерфейс Car під назвою tmp. Тепер за допомогою цього посилання можна звернутися до потрібного методу, що виведе рядок "Це суперкар Ferrari". Далі слід діяти аналогічно.

Зауваження. Дізнатися, чи реалізує об'єкт той чи інший інтерфейс, можна не тільки за допомогою оператора is, але і за допомогою оператора as і звичайної конструкції if ().

Крім того, інтерфейси можуть узагальнювати кілька об'єктів. Адже якщо кілька об'єктів (маються на увазі примірники різних класів) реалізують той самий інтерфейс, то у них є загальні методи та властивості (правда, реалізовувати їх можна по-різному). З огляду на цей факт можна використовувати інтерфейс як параметр, який передається функції, а функція вже сама буде вирішувати, яку версію методу запускати.

Як і класи, інтерфейси можна наслідувати; успадкування інтерфейсів нічим не відрізняється від успадкування класів. Але, на відміну від класів, інтерфейси підтримують множинне успадкування, тобто один інтерфейс може наслідувати відразу від двох і більше не пов'язаних між собою інтерфейсів. Робиться це так само, як і реалізація класом декількох інтерфейсів, тобто успадковані інтерфейси пишуться через кому.

Класи в мові C# зазнали досить серйозних змін порівняно з мовою програмування C++: неможливість множинного спадкоємства; в C #, дозволено успадкування від декількох інтерфейсів.

Інтерфейсом у C# є тип посилань, що містить тільки абстрактні елементи, які не мають реалізації. Безпосередньо реалізація цих елементів повинна міститися в класі, похідному від цього інтерфейсу (не можна безпосередньо створювати екземпляри інтерфейсів). Інтерфейси C# можуть містити методи, властивості й індексатори. Проте, на відміну, наприклад, від Java, вони не можуть містити константні значення.

Приклад 18. Розглянемо найпростіший приклад використання інтерфейсів:

```
using System;
class CShape
{
bool IsShape() {return true;}
}
interface IShape
{
double Square();
}
class CRectangle: CShape, IShape
{
double width;
double height;
public CRectangle(double width, double height)
{
this.width = width;
this.height = height;
}
public double Square()
{
return (width * height);
}
}
class CCircle: CShape, IShape
{
double radius;
public CCircle(double radius)
{
this.radius = radius;
}
public double Square() {
return (Math.PI * radius * radius);
}
}
class Example_3 {
public static void Main()
{
CRectangle rect = new CRectangle(3, 4);
CCircle circ = new CCircle(5);
Console.WriteLine(rect.Square());
Console.WriteLine(circ.Square());
}
}
```

Обидва об'єкти, rect і circ, є похідними від базового класу CShape; тим самим вони успадковують єдиний метод IsShape (). Задавши ім'я інтерфейсу IShape в оголошеннях CRectangle і CCircle, ми вказуємо на те, що в цих класах міститься реалізація всіх методів інтерфейсу IShape.

Реалізація поліморфної поведінки з інтерфейсом в мові Java:

```
interface Shape { void draw(); }
class Circle implements Shape {
  void draw() { System.out.println("Circle.draw()"); }
}
class Square implements Shape {
  void draw() {System.out.println("Square.draw()");}
  public static void main(String[ ] args) {
     //Shape s = new Shape();  // Помилка
     Shape s; // Немає помилки
     s = new Circle(); s. draw(); // викликається draw() з Circle
     s = new Square(); s. draw();// викликається draw() з Square
  }
}
```

Крім того, члени інтерфейсів не мають модифікаторів доступу. Їх область видимості визначається безпосередньо класом, що реалізується.

Питання для самопідготовки

1. Пізнє та раннє зв'язування.
2. Використання конструкторів за успадкуванням

Запитання для самодіагностики

1. Що таке "агрегація" та "успадкування"?
2. Наведіть приклад синтаксису успадкування в C#.
3. Яка послідовність виклику конструкторів за успадкування?
4. Як перевизначити "базовий" метод?
5. У чому полягає принцип поліморфізму?
6. Які переваги має концепція поліморфізму?
7. Що таке "абстрактні класи" та яке призначення вони мають?
8. Які правила використання абстрактних класів існують?
9. Дайте характеристику поняття "інтерфейси" та їхнього призначення.
10. Які існують правила використання інтерфейсів?

Література: [9; 11; 12; 13; 16; 17; 18; 24].

6. Принципи об'єктно-орієнтованого проектування класів

Мета теми: набуття знань щодо принципів SOLID та основних шаблонів проектування.

Професійні компетентності: здатність використовувати принципи SOLID і базові шаблони проектування.

Основні питання:

6.1. Система принципів SOLID. Принцип єдиної відповідальності.

6.2. Загальні відомості про шаблони проектування. Застосування основних шаблонів проектування.

Питання для опрацювання: система принципів SOLID, загальні відомості про шаблони проектування.

Ключові слова: SOLID, SRP, OCP, LSP, DIP, шаблон проектування.

6.1. Система принципів SOLID. Принцип єдиної відповідальності

"Гнилий код" − будь-яка особливість вихідного коду програми, яка в поточному часі не заважає її функціонуванню, проте свідчить про слабкі місця дизайну, які можуть уповільнити процес розроблення або збільшити ризик появи помилок або збоїв в майбутньому.

Ознаки "гнилого коду"

Rigidity (жорсткість) − програму важко змінити; кожна зміна викликає каскад наступних змін у залежних модулях.

Fragility (крихкість) – зі зміною програма вона "ламається" в багатьох місцях.

Immobility (нерухомість) − неможливість повторного використання програмного коду з інших проектів або із частин цього ж проекту.

Viscosity (в'язкість) − зі зміною вимог до програми в неї важко внести зміни, що зберігають вихідний дизайн, але легко руйнують його.

Принципи об'єктно-орієнтованого проектування (SOLID)

Принцип єдиної відповідальності (Single Responsibility Principle – SRP).

Принцип відкриття – закриття (Open-Closed Principle – OCP).

Принцип заміщення Барбари Лісков (Liskov Substitution Principle – LSP).

Принцип ізоляції інтерфейсу (Interface Segregation Principle – ISP).

Принцип інверсії залежностей (Dependency Inversion Principle – DIP).

Принцип єдиної відповідальності: не повинно існувати більше однієї причини для зміни даного класу.

Клас повинен мати тільки одну відповідальність, і вона повинна бути повністю інкапсульована в класі.

Призначення класу однозначно інтерпретується, реалізація прихована в класі.

Об'єднання двох сутностей, що змінюються з різних причин і в різний час, вважається поганим проектним рішенням.

Приклад 1. Порушення принципу єдиної відповідальності

```
class Calculator {
  public void add(int x, int y) {
   System.out.println(x + y);
 }
  public void sub(int x, int y) {
   System.out.println(x - y);
 }
}
```

Найпростіший спосіб вирішення проблеми – розділити клас Calculator на два класи.

Приклад 2. Дотримання принципу єдиної відповідальності:

```
class Calculator {
  public int add(int x, int y) {
    return x + y;
 }
  public int sub(int x, int y) {
    return x - y;
}
class Printer {
  public static void print(int value)
{
   System.out.println(value);
 }
}
```

Принцип відкриття – закриття: програмні сутності (класи, модулі тощо) повинні бути відкриті для розширення, але закриті для модифікації.

Через це ви можете писати код, який не потрібно міняти кожен раз зі зміною вимог (рис. 6.1).

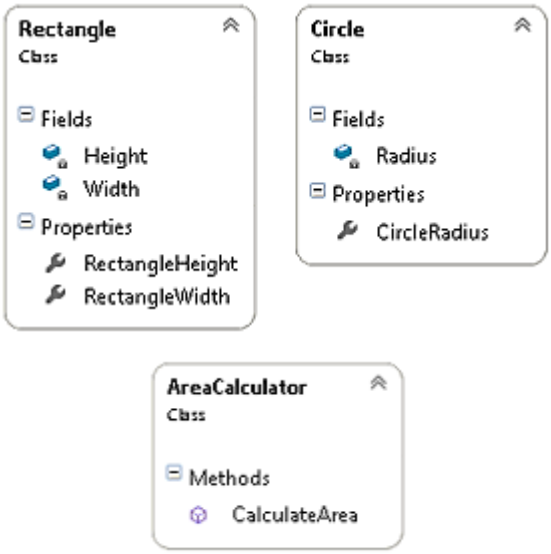

Рис. 6.1. **Порушення принципу відкриття-закриття**

Приклад 3. У цьому прикладі необхідно обчислювати сумарну площу множини прямокутників (Rectangle) та кіл (Circle), використовуючи метод CalculateArea класу AreaCalculator. Його вихідний код мовою C# може бути таким:

```
public double CalculateArea2(object[] shapes)
{
    double area = 0;
     foreach (object shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle)shape;
            area += rectangle.RectangleWidth * rectangle.RectangleHeight;
         }
         else
         {
            Circle circle = (Circle)shape;
            area += circle.CircleRadius * circle.CircleRadius * Math.PI;
          }
    }
    return area;
}
```

У цьому коді перед обчисленням площі геометричної фігури необхідно перевіряти її тип (прямокутник чи ні).

Якщо вимоги до програми зміняться так, щоб можна було додати ще декілька конкретних фігур та обчислювати їхню площу, код методу прийдеться також змінити. Чим більше типів геометричних фігур буде використовуватися, тим більше потрібно змін. Принцип відкриття – закриття дозволяє зробити, щоб код методу CalculateArea не залежав від цього (рис. 6.2).

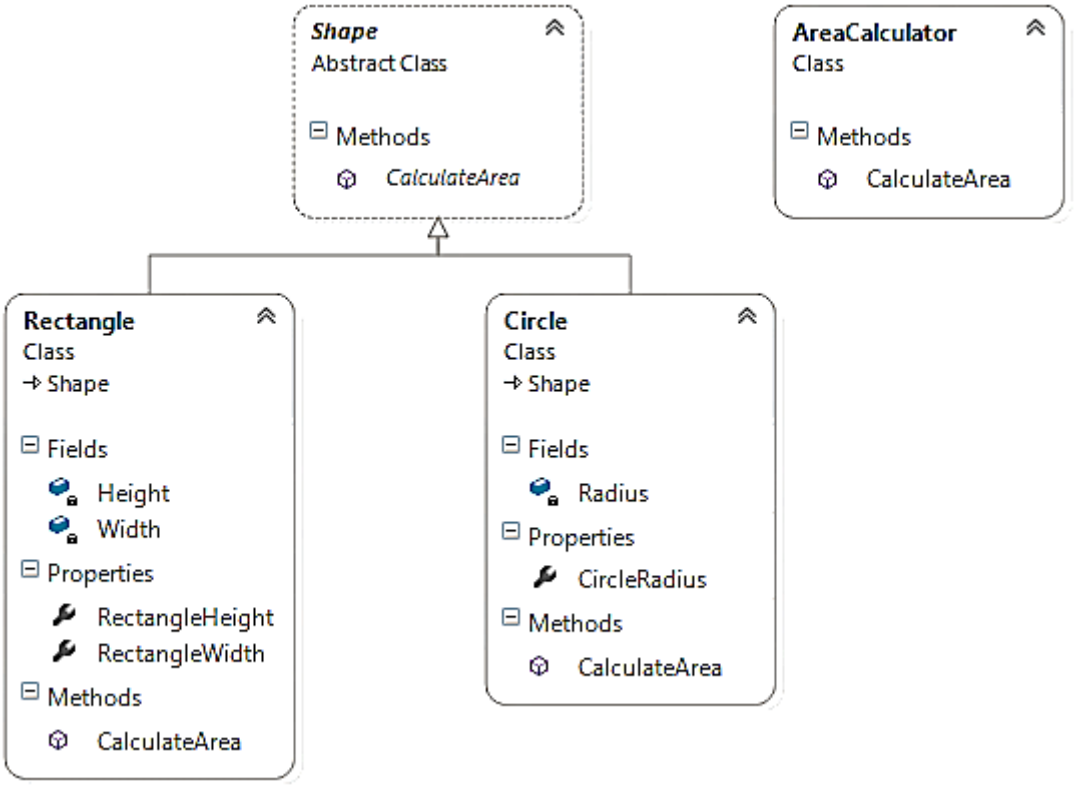

Рис. 6.2. **Дотримання принципу відкриття – закриття**

Тут у кожному класі, що описує деяку геометричну фігуру, є певна реалізація методу CalculateArea, яка перевизначає абстрактний метод CalculateArea абстрактного класу Shape. Тоді маємо таку реалізацію цього методу в класі AreaCalculator:

```
public double CalculateArea(Shape[] shapes)
{
   double area = 0;
   foreach (var shape in shapes)
   {
      area += shape.CalculateArea();
```

```
    }
    return area;
}
```

Принцип заміщення Барбари Лісков: методи, які використовують посилання на базові класи, повинні бути здатними використовувати об'єкти похідних класів, не "знаючи" про це.

Розглянемо спочатку діаграму класів на рис. 6.3.

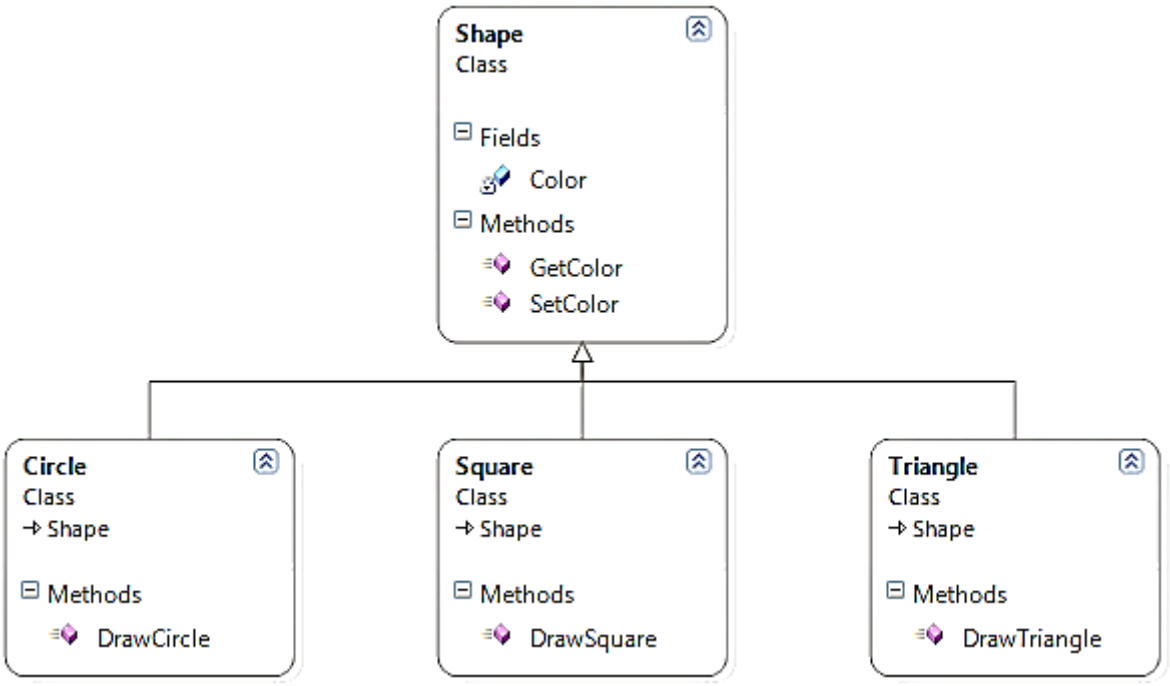

Рис. 6.3. **Порушення принципу заміщення Барбари Лісков**

Вона відбиває структуру успадкування класів, необхідних для розроблення програми, що "малює" геометричні фігури різних типів: кола (Circle), квадрати (Square) та трикутники (Triangle). Кожний з цих класів має свій метод "малювання": DrawCircle, DrawSquare та DrawTriangle.

Вихідний код метод Main такої програми має вигляд:

```
static void Main(string[] args)
      {
         Shape[ ] ShapeList = new Shape[3];
         Circle C = new Circle();
         C.SetColor("Red");
         Square S = new Square();
         S.SetColor("Green");
         Triangle T = new Triangle();
```

```
        T.SetColor("Yellow");
        ShapeList[0] = C;
        ShapeList[1] = S;
        ShapeList[2] = T;
        foreach (Shape s in ShapeList)
        {
           if (s is Square)
              ((Square)s).DrawSquare();
           else if (s is Circle)
              ((Circle)s).DrawCircle();
           else if (s is Triangle)
              ((Triangle)s).DrawTriangle();
        }
     }
```

У цьому прикладі є численні явні приведення типів, через це зі збільшенням кількості типів фігур код ускладнюється.

Кожний клас, що описує конкретну геометричну фігуру, має певну реалізацію методу "малювання" draw, яка перевизначає абстрактний метод draw абстрактного класу Shape.

Для подолання цієї проблеми треба змінити попередню діаграму класів (рис. 6.4). У методі Main використовуються поліморфні виклики цього методу:

```
static void Main(string[] args)
 {
    Shape[ ] ShapeList = new Shape[3];
    Circle C = new Circle();
    C.SetColor("Red");
    Square S = new Square();
    S.SetColor("Green");
    Triangle T = new Triangle();
    T.SetColor("Yellow");
    ShapeList[0] = C;
    ShapeList[1] = S;
    ShapeList[2] = T;
    foreach (Shape s in ShapeList)
       s.draw();
}
```

Принцип ізоляції інтерфейсу: клієнти не повинні залежати від програмних інтерфейсів, які вони не використовують. Приклад порушення розглядуваного принципу розміщений на рис. 6.5.

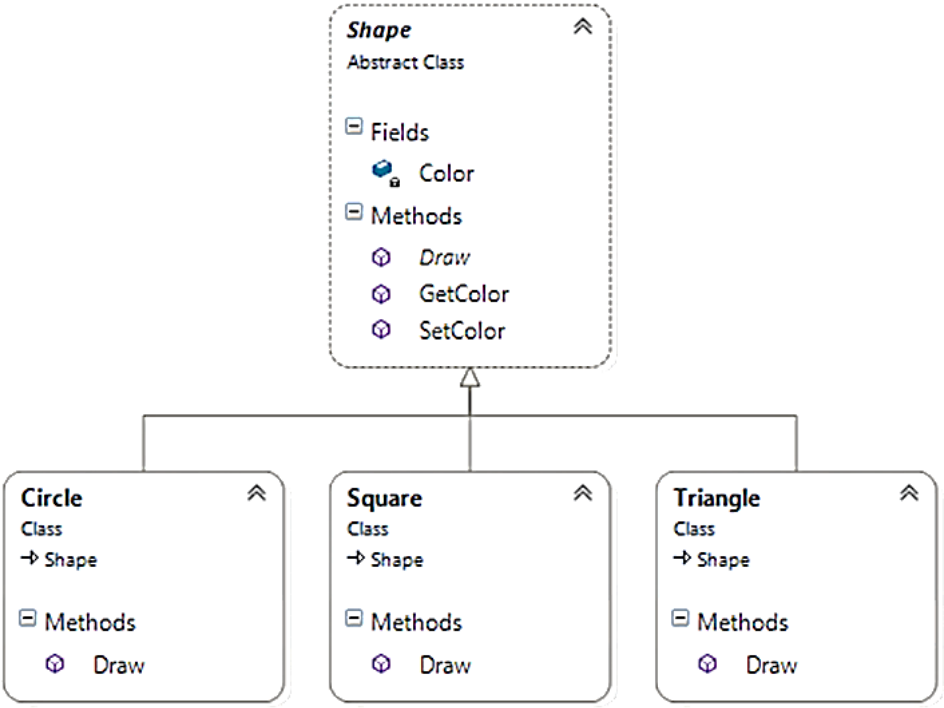

Рис. 6.4. **Дотримання принципу заміщення Барбари Лісков**

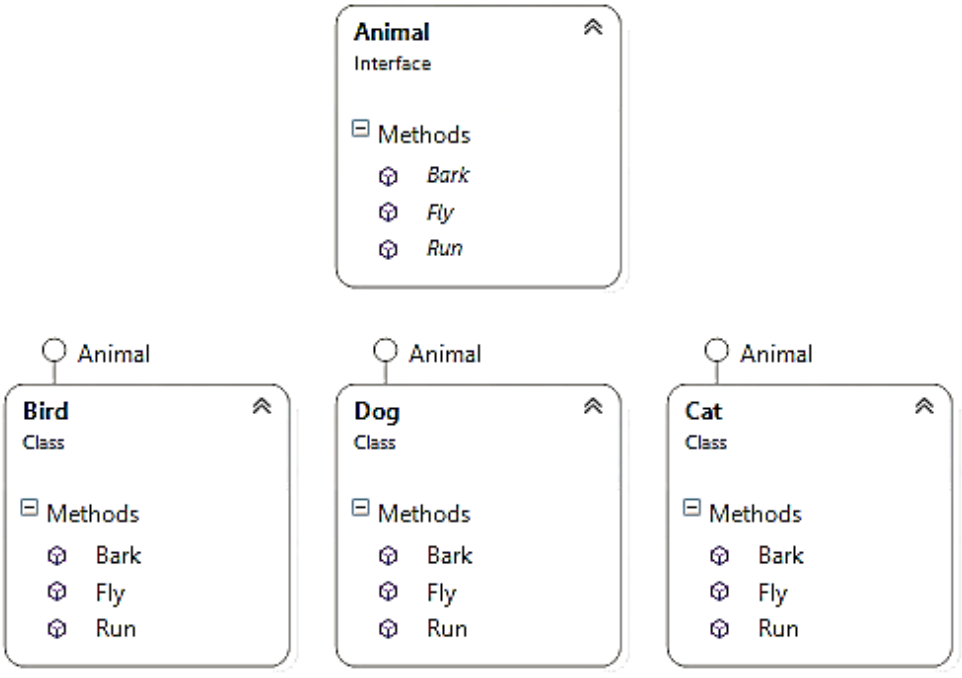

Рис. 6.5. **Порушення принципу ізоляції інтерфейсу**

На рис. 6.5 інтерфейс Animal має три абстрактних варіанти поведінки: гавкати (Bark), літати (Fly) та бігати (Run). Тому класи-нащадки Bird (птиця), Dog (собака), Cat (кішка) мають реалізувати всі методи цього інтерфейсу. Але не всі ці варіанти поведінки притаманні зазначеним тваринам.

Для подолання проблеми необхідно замість інтерфейсу Animal використати три інтерфейси (Barkable, Flyable та Runnable), в кожному з яких визначено тільки один варіант поведінки тварини (рис. 6.6).

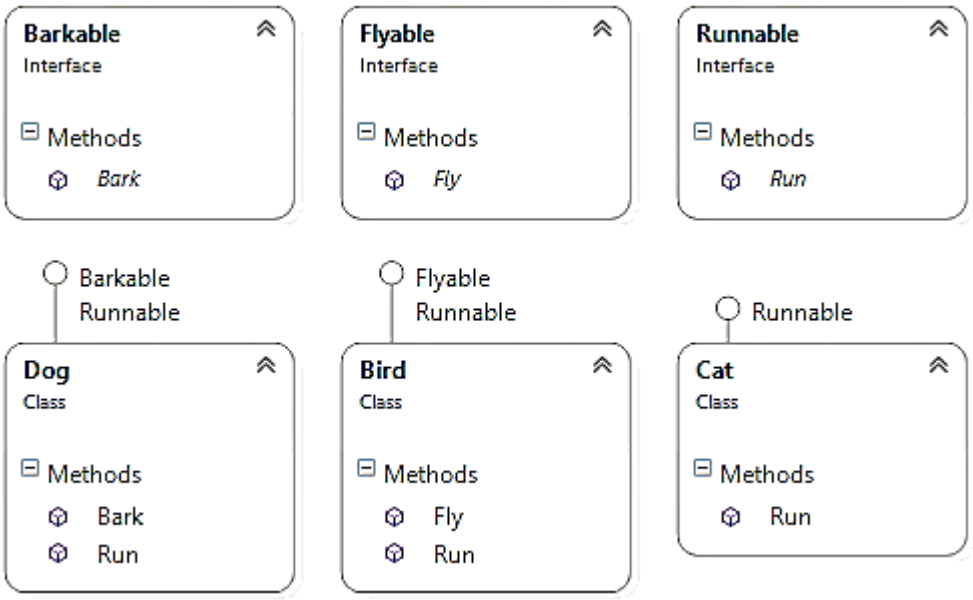

Рис. 6.6. **Дотримання принципу ізоляції інтерфейсу**

Принцип інверсії залежностей: модулі верхнього рівня не повинні залежати від модулів нижнього рівня.

Приклад 4. Порушення принципу інверсії залежностей.

```
class Copy {
        private KeyboardReader reader;
        private PrinterWriter writer;
    public Copy() {
            this.reader = new KeyboardReader();
            this.writer = new PrinterWriter(); }
        public void DoWork() {
          writer.Write(this.reader.Read());
        }
     }
```

Програма копіює символи, що вводяться з клавіатури, на принтер. Клас Copy безпосередньо використовує класи KeyboardReader і PrinterWriter, тому зі зміною вимог до вводу – виводу клас Copy доведеться змінити.

Приклад 5. Дотримання принципу інверсії залежностей.

```
class Copy {
        private IReader reader;
```

```
    private IWriter writer;
        public Copy(IReader reader, IWriter writer) {
      this.reader = reader);
      this.writer = reader;       }
    public void DoWork() {
      writer.Write(reader.Read());       }
  }
```

Усі модулі повинні залежати від абстракцій.

6.2. Загальні відомості про шаблони проектування. Застосування основних шаблонів проектування

Шаблон проектування − повторювана архітектурна конструкція, що є рішенням проблеми проектування в рамках деякого контексту, який часто виникає [1].

Зазвичай шаблон не є закінченим зразком, який може бути прямо перетворений в код; це лише приклад розв'язання задач, який можна використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони показують відношення і взаємодію між класами або об'єктами без визначення того, які кінцеві класи або об'єкти застосунку будуть використовуватися.

Як показує аналіз літератури [2 – 8], використання шаблонів має як переваги, так і певні недоліки.

Переваги використання шаблонів проектування

1. Кожен шаблон описує рішення цілого класу абстрактних проблем.
2. Дозволяє глибоко та всебічно опрацювати архітектуру розроблюваної системи.
3. Підвищує стійкість системи до зміни вимог.
4. Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову та знову.
5. Полегшує дискусію про абстрактні структури даних між розробниками, оскільки вони можуть посилатися на відомі іменовані шаблони.

Недоліки використання шаблонів проектування

1. Іноді шаблони консервують громіздку та малоефективну систему понять, розроблену вузькою групою.

2. Шаблони можуть пропагандувати недоцільні стилі розроблення застосунків.

3. Сліпе застосування шаблонів без осмислення причин і передумов виділення кожного окремого з них уповільнює професійне зростання програміста, оскільки підміняє творчу роботу механічною підстановкою шаблонів.

Атрибути шаблону

1. **Ім'я**. Важливе, тому що воно стає частиною словника проектування та підвищує рівень спілкування.

2. **Проблема**. Це опис ситуації застосовування певного шаблону.

3. **Рішення**. Визначає елементи шаблону та взаємозв'язок між ними. Воно є абстрактним, тобто має бути конкретизоване в конкретному застосунку.

4. **Наслідки**. Це результати та компроміси застосування шаблону, оскільки кожен шаблон має як переваги, так і недоліки.

Типи шаблонів проектування

Шаблони проектування розподіляють на такі категорії: породжувальні шаблони, структурні та поведінкові [2].

Породжувальні шаблони – абстрагують процес створення об'єктів.

Структурні шаблони – визначають різні складні структури, які змінюють інтерфейс вже існуючих об'єктів або його реалізацію, дозволяючи полегшити розроблення і оптимізувати програму.

Поведінкові шаблони – визначають взаємодію між об'єктами, тим самим збільшуючи його гнучкість.

Породжувальні шаблони

"*Абстрактна фабрика*" (Abstract Factory) – надає інтерфейс для створення сімейств, пов'язаних між собою, або незалежних об'єктів, конкретні класи яких невідомі.

1. "*Фабричний метод*" (Factory Method) – визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, об'єкт якого з них створювати.

2. "*Одинак*" (Singleton) – гарантує, що певний клас може мати тільки один екземпляр, і надає глобальну точку доступу до нього.

3. "*Будівельник*" (Builder) – відокремлює конструювання складного об'єкта від його подання, дозволяючи використовувати той самий процес конструювання для створення різних подань.

4. "*Прототип*" (Prototype) – описує види створюваних об'єктів за допомогою прототипу та створює нові об'єкти шляхом його копіювання

5. "*Пул об'єктів*" (Object Pool) – клас, який є програмним інтерфейсом для роботи з набором ініціалізованих і готових до використання об'єктів. Коли системі потрібний об'єкт, він не створюється, а береться з пулу. Коли об'єкт більше не потрібен, він не знищується, а повертається в пул.

На практиці найбільш часто використовують шаблони "Абстрактна фабрика", "Фабричний метод" та "Одинак". UML-діаграма класів шаблона "Одинак", наведена на рис. 6.7.

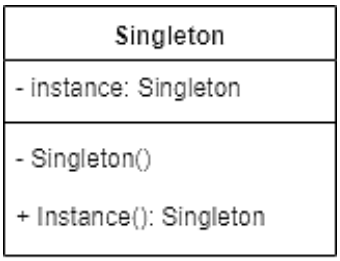

Рис. 6.7. **Шаблон "Одинак"**

Назви на рису. 6.7 є узагальненими. У конкретному застосунку вони, як правило, будуть іншими. Це стосується також усіх інших шаблонів.

Учасником розглядуваного шаблону є єдиний клас – Singleton. Він повинен мати приватне поле instance типу Singleton, приватний конструктор без параметрів, а також загальнодоступний метод Instance, який повертає посилання на об'єкт класу Singleton.

Приклад 6. Приклад вихідного коду програми мовою Java, що містить клас Earth, побудований відповідно до шаблону Singleton, та створює його об'єкт:

```
class Earth {
   private static Earth instance;
   private Earth() {
     System.out.println("Об'єкт створено...");
   }
   public static Earth GetInstance() {
     if (instance == null)
     {
         instance = new Earth();
     }
     return instance;
```

```
    }
}

class Program
{
    public static void main(String[] args) {
      Earth e1 = Earth.GetInstance();
    }
}
```

У методі GetInstance класа Earth перевіряється, чи був створений його об'єкт. Якщо цей метод викликано вперше, то створюється новий об'єкт класу Earth. Інакше цей метод повертає посилання на об'єкт класу Earth, що був створений під час першого виклику методу GetInstance. Приватний конструктор цього класу запобігає створенню його об'єкта іншими способами. Статичне поле instance дозволяє однократно ініціалізувати його, закладаючи в пам'ять єдину його копію.

Таким чином, гарантується, що в пам'яті програми буде існувати тільки один об'єкт класу Earth.

Структурні шаблони

1. "*Адаптер*" (Adapter) – перетворює інтерфейс класу в деякий інший інтерфейс, очікуваний клієнтами. Забезпечує спільну роботу класів, яка була б неможливою через несумісність їх інтерфейсів.

2. "*Декоратор*" (Decorator) – динамічно покладає на об'єкт нові функції. "Декоратори" застосовуються для розширення наявної функціональності; вони є гнучкою альтернативою породження підкласів.

3. "*Замісник*" (Proxy) – підміняє інший об'єкт для контролю доступу до нього, тобто є замінником іншого об'єкта. "Замісник" може застосовуватись у всіх випадках, коли виникає необхідність послатися на об'єкт більш витончено, ніж це можливо з використанням простого покажчика.

4. "*Компонувальник*" (Composite) – групує об'єкти в деревоподібні структури для подання ієрархії типу "частина – ціле". Дозволяє клієнтам працювати з одиничними об'єктами так само, як з групами об'єктів.

5. "*Міст*" (Bridge) – відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше.

6. "*Пристосуванець*" (Flyweight) – мінімізує витрати пам'яті за рахунок сумісного використання даних з великою кількістю інших подібних об'єктів.

7. "*Фасад*" (Facade) – надає уніфікований інтерфейс до безлічі інтерфейсів у деякій підсистемі. Визначає інтерфейс більш високого рівня, що полегшує роботу з підсистемою. Клієнти спілкуються з підсистемою, посилаючи запити "фасаду". Він переадресує їх відповідним об'єктам усередині підсистеми.

Найбільш часто використовують структурні шаблони "Адаптер", "Замісник", "Фасад".

Приклад 7. Розглянемо більш ретельно шаблон "Замісник".

"Замісник" (Proxy) створює об'єкт, що представляє інший об'єкт.

Його цілі:

забезпечити замінник або заповнювач для іншого об'єкта, щоб управляти доступом до нього;

використовувати додатковий рівень абстракції для підтримки розподіленого, керованого або інтелектуального доступу;

додати "обгортку" та делегацію для захисту реального компонента від зайвої складності.

UML-діаграма класів шаблона "Замісник" наведена на рис. 6.8.

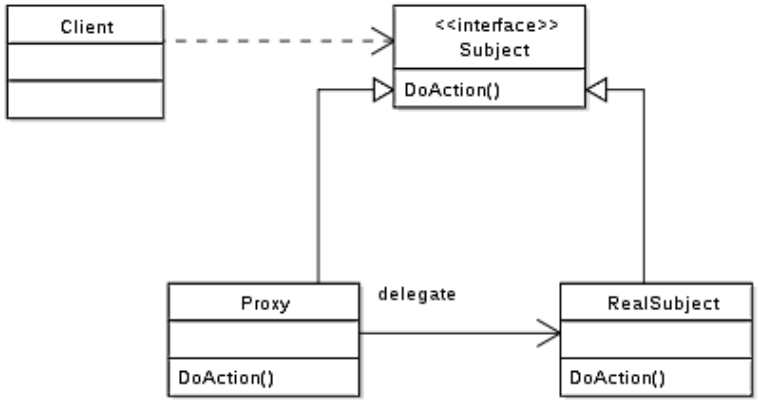

Рис. 6.8. **Шаблон "Замісник"**

Учасниками цього шаблону є класи RealSubject, Proxy, Client та інтерфейс Subject.

Інтерфейс Subject визначає абстрактний метод DoAction, призначений для виклику з коду класа Client. Цей метод перевизначають класи RealSubject і Proxy.

Запит від клієнтського коду на виконання певних дій спочатку надходить до об'єкта класа Proxy з використанням посилання типу Subject, а він передає його об'єкту класа RealSubject. Цей об'єкт виконує запит і повертає результат об'єкту класа Proxy.

Отже, об'єкт Proxy – це "обгортка" або об'єкт-агент, який викликається клієнтом для неявного доступу до реального об'єкта (RealSubject), що обслуговує запит. Об'єкт RealSubject є прихованим.

Приклад діаграми класів конкретної реалізації шаблона "Замісник" наведено на рис. 6.9.

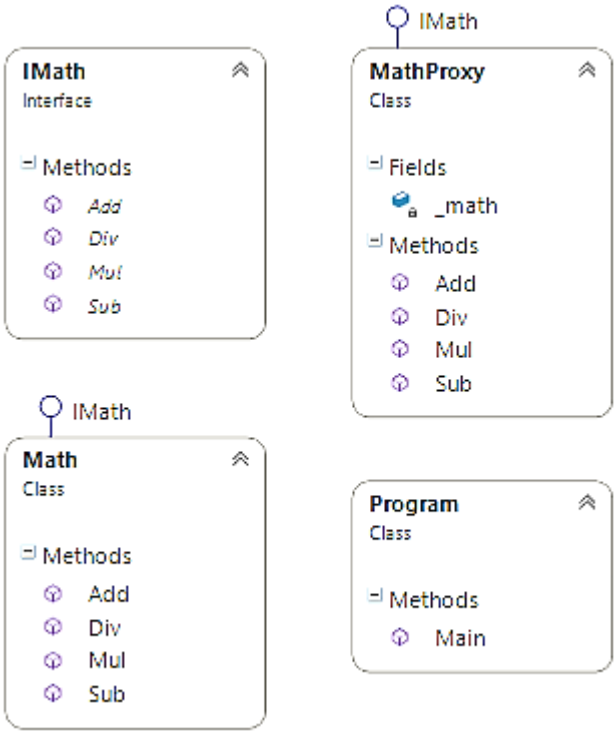

Рис. 6.9. **Приклад реалізації шаблона "Замісник"**

Інтерфейс IMath виконує роль "Subject", клас Math виконує роль "RealSubject", клас MathProxy – роль "Proxy", клас Program – роль "Client".

Add – метод, призначений для складання двох чисел; Div – метод для ділення двох чисел; Mul – метод для множення двох чисел; Sub – метод для віднімання двох чисел. Поле _math – посилання типу IMath.

Головний метод програми, який реалізує діаграму класів може виглядати таким чином:

```
public static void Main(string[ ] args)
{
   // Створення об'єкта класа MathProxy
   IMath proxy = new MathProxy();

   Console.WriteLine("4 + 2 = " + proxy.Add(4, 2));
   Console.WriteLine("4 - 2 = " + proxy.Sub(4, 2));
   Console.WriteLine("4 * 2 = " + proxy.Mul(4, 2));
```

```
    Console.WriteLine("4 / 2 = " + proxy.Div(4, 2));
}
```

Отже, у методі Main викликаються методи Add, Div, Mul і Sub екземпляру класу Math.

Поведінкові шаблони

1. "*Інтерпретатор*" (Interpreter) − для заданої мови визначає подання її граматики, а також інтерпретатор речень мови, що використовує це подання.

2. "*Шаблоновий метод*" (Template Method) − визначає "скелет" алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальної структури.

3. "*Ітератор*" (Iterator) − дає можливість послідовно обійти всі елементи складеного об'єкта, не розкриваючи його внутрішнього устрою.

4. "*Команда*" (Command) − інкапсулює запит у вигляді об'єкта.

5. "*Спостерігач*" (Observer) − визначає відношення типу "один − до − багатьох" між об'єктами: зі зміною стану одного об'єкта всі залежні від нього об'єкти отримують повідомлення і автоматично оновлюються.

6. "*Відвідувач*" (Visitor) − операція, яку треба виконати над елементами об'єкта. Дозволяє визначити нову операцію, не змінюючи класи елементів, до яких він застосовується.

7. "*Посередник*" (Mediator) − визначає об'єкт, в якому інкапсульоване знання про те, як взаємодіють об'єкти з деякої множини. Сприяє зменшенню числа зв'язків між об'єктами, дозволяючи їм працювати без явних взаємопосилань.

8. "*Стан*" (State) − дозволяє об'єкту варіювати свою поведінку у разі зміни внутрішнього стану. Водночас створюється враження, що змінився клас об'єкта.

9. "*Стратегія*" (Strategy) − визначає сімейство алгоритмів, інкапсулюючи їх всі та дозволяючи підставляти один замість іншого. Можна міняти алгоритм незалежно від клієнта, який ним користується.

10. "*Хранитель*" (Memento) − дозволяє, не порушуючи інкапсуляції, отримати та зберегти у зовнішній пам'яті внутрішній стан об'єкта, щоб пізніше його можна було відновити в такому самому стані.

11. "*Ланцюжок обов'язків*" (Chain of Responsibility) − дозволяє уникнути жорсткої залежності відправника запиту від його отримувача. Об'єкти-отримувачі зв'язуються в ланцюжок, і запит передається цим ланцюжком, поки деякий об'єкт його не обробить.

Найпопулярніші в користуванні шаблони "Команда", "Спостерігач" і "Стратегія".

Питання для самопідготовки

1. Принципи SOLID та їхнє використання.

Запитання для самодіагностики

1. У чому полягає сутність принципу єдиної відповідальності?
2. У чому полягає сутність принципу відкриття – закриття?
3. У чому полягає сутність принципу інверсії залежностей?
4. У чому полягає сутність принципу заміщення Барбари Лісков?
5. У чому полягає сутність принципу ізоляції інтерфейсу?
6. Для чого використовують шаблон "Абстрактна фабрика"?
7. Для чого використовують шаблон "Фабричний метод"?
8. Для чого використовують шаблон "Адаптер"?
9. Для чого використовують шаблон "Компонувальник"?
10. Для чого використовують шаблон "Фасад"?
11. Для чого використовують шаблон "Команда"?
12. Для чого використовують шаблон "Спостерігач"?
13. Для чого використовують шаблон "Стратегія"?

Література: [2; 3; 5; 6; 7; 10; 22].

7. Бібліотеки класів

Мета теми: набуття знань щодо принципів подання бібліотек класів на платформах Microsoft .NET і Java SE.

Професійні компетентності: здатність створювати та використовувати бібліотеки класів.

Основні питання:

7.1. Бібліотеки та їх використання. Статичні та динамічні бібліотеки.

7.2. Розроблення бібліотек на платформі Java SE. DLL-бібліотеки. Розроблення DLL-бібліотек на платформі Microsoft .NET.

Питання для опрацювання: бібліотеки та їхнє використання, розроблення бібліотек на платформах Java SE, Microsoft .NET.

Ключові слова: статичні бібліотеки, динамічні бібліотеки, апаратно-програмна платформа, JAR, інтегроване середовище, цільовий застосунок.

7.1. Бібліотеки та їх використання. Статичні та динамічні бібліотеки

Бібліотека − набір ресурсів, заздалегідь створених компаніями або незалежними розробниками, призначений для використання під час розроблення програм. Такі ресурси містять код і дані, що надають послуги розроблюваним програмам, а саме: типи даних, підпрограми, класи, шаблони повідомлень, конфігураційну інформацію тощо. Одна бібліотека може використовуватися в абсолютно різних програмах. Головне, щоб вона була сумісна з цільовою платформою розробки.

Бібліотеки можуть бути частиною мови програмування або програмної платформи, на якій розроблюється цільова програма. Прикладами таких стандартних бібліотек є бібліотека вводу – виводу та бібліотека колекцій платформи Java або MS .NET. Інші бібліотеки постачаються сторонніми розробниками.

Більшість розроблюваних програм можуть отримати дуже складні можливості, просто використавши готову бібліотеку замість самостійного розроблення відповідного коду.

За способом використання бібліотеки бувають статичними і динамічними.

Статичні бібліотеки можуть бути у вигляді вихідного коду або об'єктних файлів. Вони копіюються в цільовий застосунок на етапі компіляції. Як наслідок, цей застосунок буде мати весь необхідний код. Це робить програму автономною. Таким чином, її можна запускати на виконання на будь-якій апаратно-програмній платформі, для якої вона пристосована. Але використання статичних бібліотек призводить до збільшення розміру цільового програми.

Динамічні бібліотеки завантажуються операційною системою за запитом цільового застосунка в ході його виконання. Ту саму бібліотеку можна використати одночасно в декількох працюючих програмах. Динамічна бібліотека може бути плагіном, що розширює функціональність програми. Якщо деякі динамічні бібліотеки відсутні, то цільовий застосунок не буде працювати. Таким чином, розроблювану програму можна запускати на виконання на цільовій апаратно-програмній платформі за умови, що програма має доступ до необхідної динамічної бібліотеки.

В операційних системах Microsoft Windows використовується бібліотека динамічного компонування (Dynamic-link library). Файли таких бібліотек зазвичай мають розширення .dll.

7.2. Розроблення бібліотек на платформі Java SE. DLL-бібліотеки. Розроблення DLL-бібліотек на платформі Microsoft .NET

Бібліотека класів Java − це набір бібліотек, що можуть завантажуватися динамічно та викликатися програмою під час її виконання. Оскільки платформа Java не залежить від конкретної операційної системи, програми не можуть покладатися на платформно-залежні бібліотеки. Натомість платформа Java надає набір стандартних бібліотек класів, що містять функції, звичайні для сучасних операційних систем.

Java-бібліотеки постачаються у вигляді jar-файлів.

JAR (Java ARchive) − це формат архівного файлу, який зазвичай використовується для об'єднання багатьох файлів класів Java та пов'язаних метаданих і ресурсів в один файл для поширення прикладного програмного забезпечення або бібліотек на платформі Java. jar-файли базуються на форматі ZIP і мають розширення .jar.

Можна розробляти власні Java-бібліотеки. Створення бібліотеки в інтегрованому середовищі Eclipse доцільно розглянути на прикладі.

Приклад 1. Створимо просту математичну бібліотеку, що складається з одного класу. Клас має чотири метода для виконання арифметичних операцій, та два – для задання значень операндів. Вихідний код цього класу має такий вигляд:

```
package librarypackage;
public class LibraryClass {
    private double num1, num2;
    public LibraryClass() { }
    public void setNum1(double num1) {
       this.num1 = num1;
    }
    public void setNum2(double num2) {
       this.num2 = num2;
    }
    public double add() {
       return num1 + num2;
```

```
    }
    public double sub() {
        return num1 - num2;
    }
    public double mul() {
        return num1 * num2;
    }
    public double div() {
        return num1 / num2;
    }
}
```

Надалі необхідно провести таку послідовність процедур.

1. Створіть новий проект Java.
2. Додайте до нього клас LibraryClass (рис. 7.1).

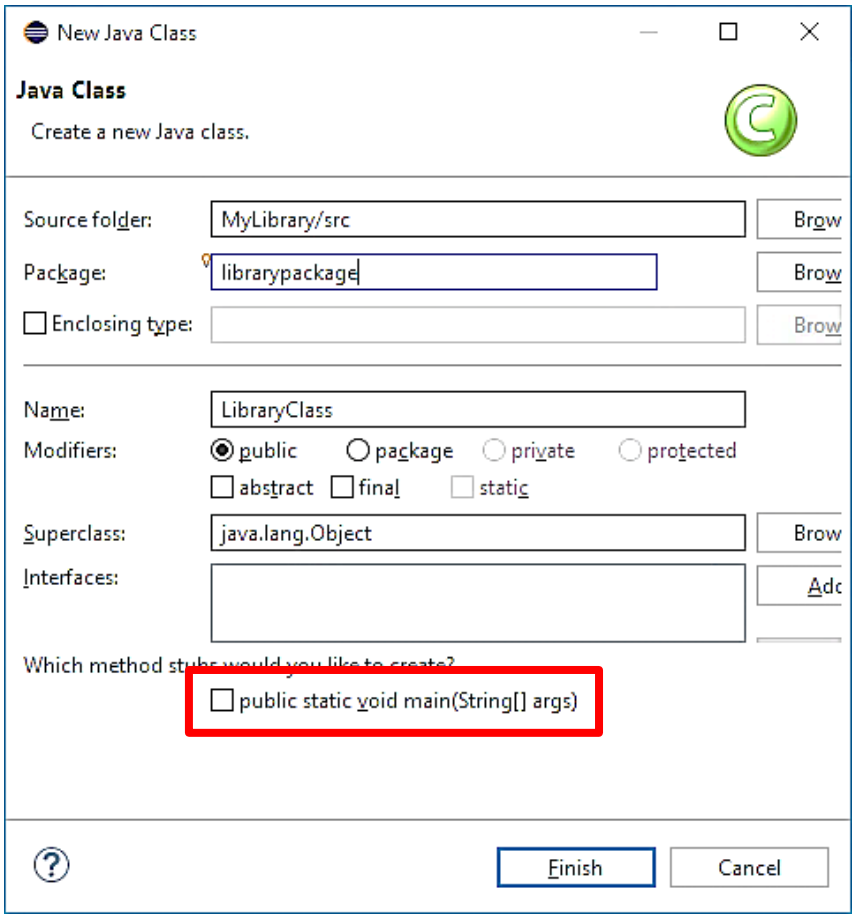

Рис. 7.1. **Створення класу бібліотеки**

3. Експортуйте проект бібліотеки в jar-файл (рис. 7.2).

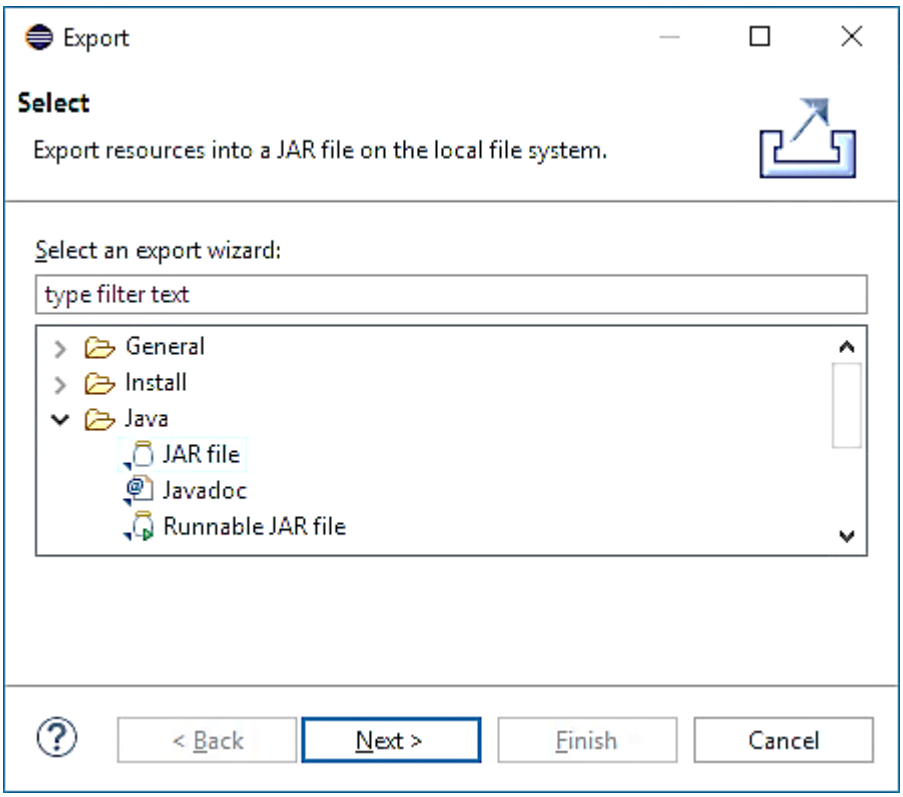

Рис. 7.2. **Вікно експорту**

4. Додайте посилання на jar-файл у цільовому застосунку (рис. 7.3).

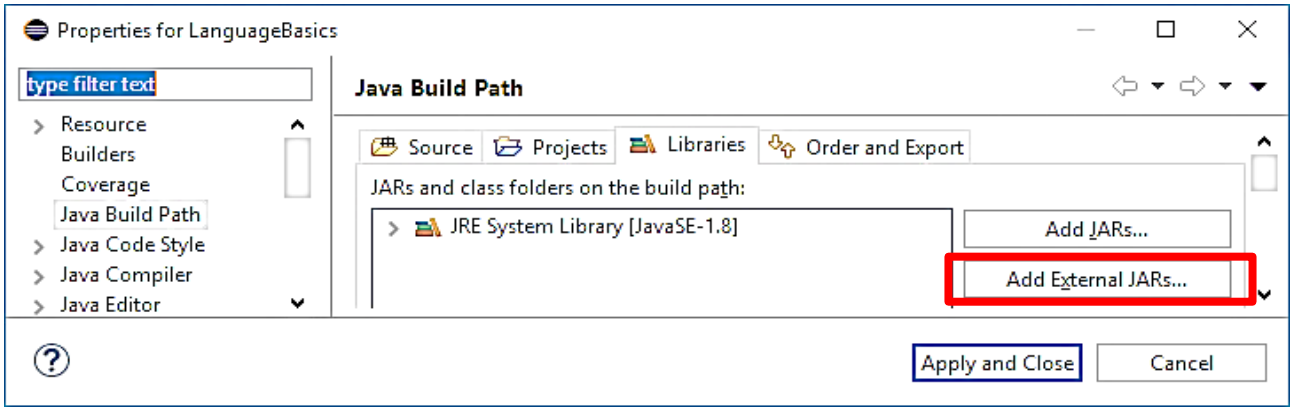

Рис. 7.3. **Вікно додавання на jar-файла**

Вихідний код цільового застосунку виглядає так:

```
package calculator;
import librarypackage.*;
public class Calculator {
   public static void main(String[ ] args) {
```

```
        LibraryClass cB = new LibraryClass();
        double n1 = 45.9, n2 = 4.67;
        cB.setNum1(n1);
        cB.setNum2(n2);
        System.out.format("Сума=%1$f5.2", cB.add());
    }
}
```

Приклад 2. Розглянемо створення dll-бібліотеки на платформі MS .NET у інтегрованому середовищі *Visual Studio 2017*.

Для цього необхідно створити новий проект типу "Class Library (.NET Framework)" (рис. 7.4). Потім додати до нього вихідний код класу LibraryClass мовою C#, що є аналогом відповідного вихідного коду мовою Java. Після компіляції цього коду отримаємо файл dll-бібліотеки.

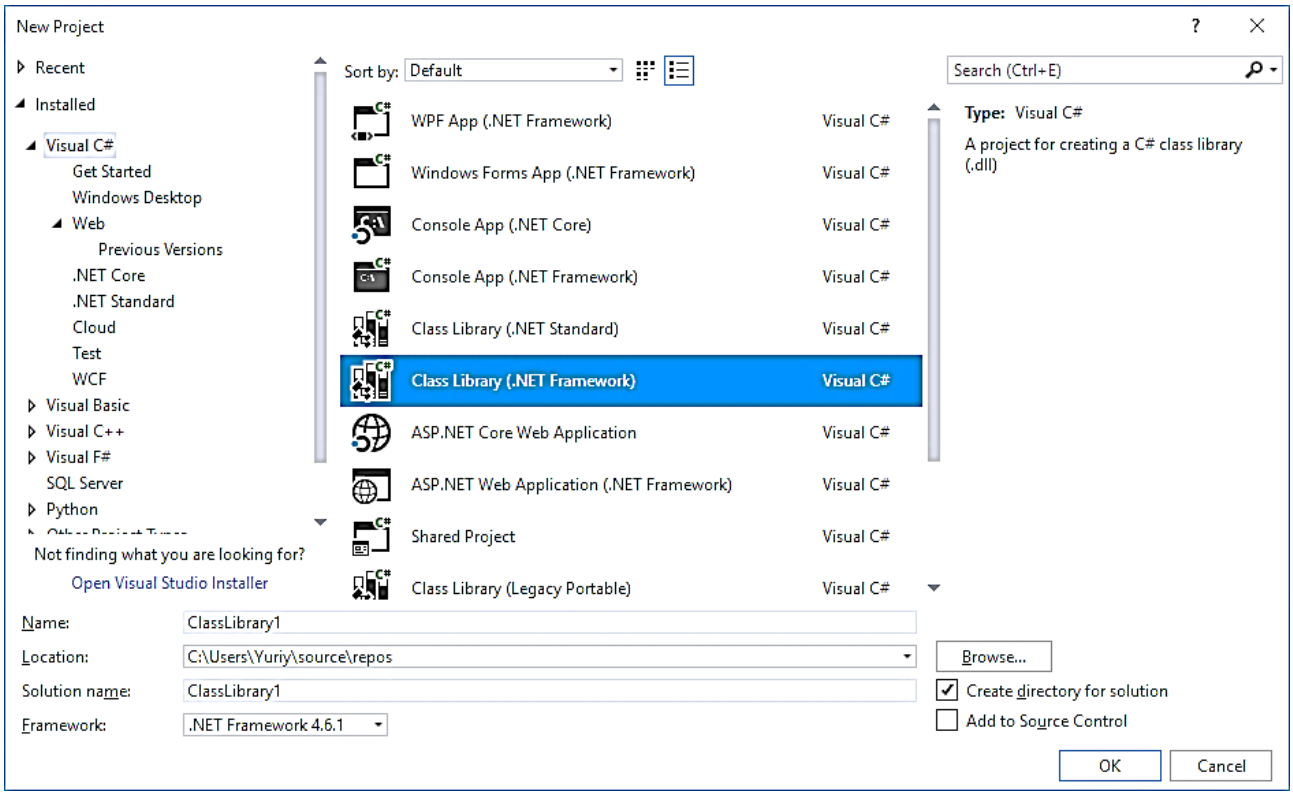

Рис. 7.4. **Вікно створення нового проекту**

У цільовому застосунку слід додати посилання на розроблену dll-бібліотеку (рис. 7.5).

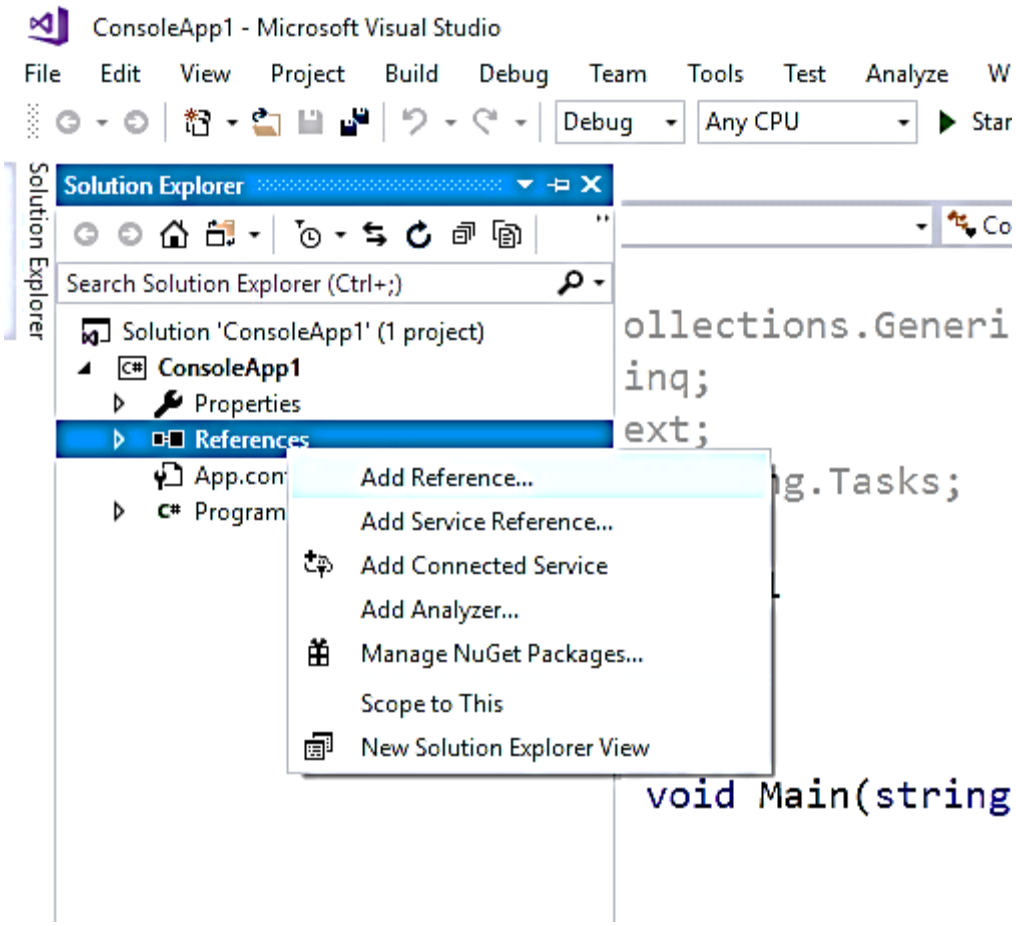

Рис. 7.5. **Додавання посилання на розроблену dll-бібліотеку**

У вихідний код цільового застосунку, аналогічний коду цільового застосунку мовою Java, потрібно додати інструкцію using для простору імен dll-бібліотеки.

Питання для самопідготовки

1. Проблема "Аду DLL" та напрями її вирішення

Запитання для самодіагностики

1. Як створити DLL-бібліотеку на платформі Microsoft. NET?
2. Як створити JAR-бібліотеку на платформі Java SE?

Література: [9; 10; 11; 23; 24].

Розділ 3. Оброблення винятків і бібліотеки класів

8. Оброблення виняткових ситуацій

Мета теми: набуття знань щодо структури основних класів винятків стандартної бібліотеки Microsoft.NET і Java SE, синтаксису оброблення винятків.

Професійні компетентності: здатність використовувати механізми оброблення винятків.

Основні питання:

8.1. Види помилок у програмах. Проблеми традиційного підходу до оброблення помилок.

8.2. Механізм оброблення винятків. Класи винятків стандартних бібіліотек Microsoft .NET і Java SE. Синтаксис оброблення винятків.

Питання для опрацювання: види помилок у програмах; розроблення бібліотек на платформах Java SE, Microsoft .NET; механізм оброблення винятків, синтаксис оброблення винятків**.**

Ключові слова: синтаксичні помилки, помилки часу виконання, логічні помилки, супроводження коду, модифікація коду, оброблення винятків, класи винятків, стандартне виключення, перехоплення винятків.

8.1. Види помилок у програмах. Проблеми традиційного підходу до оброблення помилок

Незважаючи на те, що кожний розробник намагається написати програмний код, який буде виконуватися надійно та завжди отримувати правильні результати, досить часто в програмах з'являються помилки. Причин цього явища досить багато. Це і брак досвіду або знань, це може бути так званий "людський фактор", коли розробник помиляється через неуважність або фізичну втому.

Серед помилок, які можуть з'являтися в програмах, виділяють синтаксичні помилки, помилки часу виконання, логічні помилки та інші.

Синтаксичні помилки полягають у порушенні правил написання тих чи інших формальних конструкцій алгоритмічної мови. Наприклад, пропущена літера в назві стандартного методу, відсутня крапка з комою в кінці оператора, різна кількість відкривальних і закривальних дужок тощо.

Такі помилки виявляються ще на етапі компіляції програми та швидко виправляються самим розробником.

Помилки часу виконання програми проявляються вже після того, як були виявлені та виправлені всі синтаксичні помилки. Програма успішно пройшла етап компіляції і запущена на виконання, проте в аварійному режимі достроково завершує свою роботу. Причинами такого роду помилок можуть бути, наприклад, математичні розрахунки з некоректними даними (ділення на нуль, спроба обчислити квадратний корінь або логарифм із від'ємного числа тощо), взаємодія з неіснуючими ресурсами (файлами) та інші.

Логічні помилки є найбільш складними для пошуку. У цьому випадку програма успішно компілюється, запускається, виконується і видає результат. Але цей результат або не відповідає дійсності, або не співпадає з очікуваним. Помилка може бути будь-де. Можливо, розробник помилився в кодуванні математичної формули, а можливо, хибним є увесь алгоритм розв'язання задачі. Інколи, щоб знайти причину такої помилки, потрібно витратити досить багато часу, перевірити кожний рядок програми, переглянути весь алгоритм від початку до завершення.

Зрозуміло, що проблема пошуку помилок у програмах та їх виправлення з'явилася разом появою самого програмування. Щоб локалізувати помилку, розробники у процедурах або функціях додавали, наприклад, додатковий код, який виводив інформацію про успішне, або навпаки, виконання підпрограми. З метою уникнення обробки некоректних даних в програму додавалися додаткові умовні оператори, які перевіряли допустимість вхідних даних на певні обмеження. Але такі дії, в свою чергу, приводили до збільшення об'єму програмного коду, що ускладнювало його подальше супроводження та модифікацію.

8.2. Механізм оброблення винятків. Класи винятків стандартних бібліотек Microsoft .NET і Java SE. Синтаксис оброблення винятків

Помилки далеко не завжди трапляються з вини того, хто кодує застосунок. Іноді програма генерує помилку через хибні дії кінцевого користувача. У будь-якому випадку користувач завжди повинен очікувати на виникнення помилок у своїх програмах, а програмісти, вбачаючи це, кодувати відповідно до цих очікувань.

У платформі .NET Framework передбачена розвинена система обробки помилок. Механізм, який влаштовано в C# для їх обробки, дозволяє

закодувати спеціальну процедуру оброблення для кожного типу помилкових умов, а також відокремити код, який потенційно може породжувати помилки, від коду, який провадить обробку останніх.

Незалежно від того, наскільки добре написано програмний код, розробники програми повинні вміти долати помилки будь-якого роду. Наприклад, посеред деякого процесу складної обробки програмний код може виявити, що не має прав доступу для читання файла, на який останній посилається, або ж під час передавання даних у мережі стався обрив зв'язку. У таких виняткових ситуаціях недостатньо, щоб метод повернув деякий код помилки. Для пересічного користувача це може стати критичним випадком, причини якого він не зможе зрозуміти, а продукт у його уявленні почне набувати недоброї слави.

Може статися, що до моменту виникнення надзвичайної ситуації в програмі виконано 15 – 20 вкладених викликів методів, проте потрібно "перестрибнути" назад через усі ці 15 або 20 викликів, щоб коректно вийти з завдання та прийняти відповідні заходи з оброблення ситуації. Мова C# має дуже гарний засіб для виправлення подібних ситуацій, а саме – механізм, відомий як **обробка винятків**.

На перевагу до інших мов програмування, винятки в C# відкривають для користувача новий "світ" можливостей для обробки помилок у програмах. Великий плюс, який отримала мова C# порівняно з C++, те що в мові C++ оброблення винятків може значно впливати на продуктивність процесору. У C# оброблення виняткових ситуацій побудована таким чином, що на продуктивність процесору зовсім не впливає.

Класи винятків

У мові програмування C# **виняток** – це об'єкт, створений з настанням певної помилкової ситуації. Цей об'єкт містить інформацію, яка може допомогти відстежити причину виникнення проблеми. Хоч в мові можливо створити свої власні класи винятків, .NET містить безліч визначених їх класів.

Усі класи є частиною простору імен System, за винятком IOException і класів, успадкованих від IOException, які становлять частину простору імен System.IO. Простір імен System.IO оперує читанням і записом даних у файли.

Зауваження. Не існує безумовного простору імен для винятків. Їхні класи мають бути поміщені в ті простори, де знаходяться класи, які можуть їх генерувати. Тому винятки, пов'язані з введенням – виведенням (IO), знаходяться в System.IO, а інші їх класи можна знайти в інших просторах імен базових класів.

Найбільш загальний клас винятків – System.Exception успадкований від System.Object. Програмісту не рекомендується створювати об'єкти винятків System.Exception у своєму коді, тому що вони не містять ніякої інформації щодо специфіки конкретної помилкової ситуації. Проте, якщо жоден із стандартних винятків .NET не відповідає потребам конкретного застосунку, можна створювати власні, похідні від класу System.Exception.

Від System.Exception в ієрархії успадковується важливий клас System.SystemException. Він описує винятки, які зазвичай генерує виконавча система .NET або ж які мають деяку загальну природу та можуть бути згенеровані майже будь-яким застосунком. Наприклад, StackOverflowException породжує виконавча система .NET, коли виявляє переповнення стеку. З іншого боку, у своєму коді програміст можете порушити ArgumentException або якийсь з його підкласів, якщо виявиться, що метод був викликаний з хибними аргументами. До підкласу System.SystemException належать винятки, що за характером є як фатальні, так і нефатальні помилки.

Серед інших корисних класів винятків варто розглянути:

StackOverflowException – цей виняток генерується, якщо область пам'яті, відведена для стека, переповнена. Переповнення стека може статися, якщо метод нескінченно рекурсивно викликає сам себе. Зазвичай це вважається фатальною помилкою, оскільки не дає програмі зробити нічого, окрім негайного завершення (у цьому випадку малоймовірно навіть, що виконається блок finally). Спроби обробити помилки подібного роду, як правило, неефективні, тому замість обробляння доцільно забезпечити коректний вихід із програми;

EndOfStreamException – зазвичай причиною цього винятку є спроба читання за кінцем файла. Потік (stream) є потоком даних між їх джерелами;

OverflowException – виникає зі спробою приведення змінної int, що має значення, скажімо, -40, до типу uint: у checked-контексті.

Ієрархія класів винятків дещо незвична, тому що більшість цих класів не додають ніякої функціональності до своїх базових класів. Однак у випадку їх обробки основною причиною долучення класів спадкоємців є вказівка більш специфічних помилкових умов, що часто не вимагає перевизначення

старих методів або додавання нових (хоча немає нічого незвичайного в додаванні додаткових властивостей для обслуговування додаткової інформації про умови помилки). Наприклад, програміст може мати базовий клас ArgumentException, призначений для виклику методів, коли їм передаються неправильні аргументи, і успадкований від нього клас ArgumentNullException, який обслуговує частковий випадок передавання в параметри значення null.

Перехоплення виняткiв

Беручи до уваги, що .NET Framework включає велику кількість уже існуючих класів винятків, розглянемо, як їх використовувати у своєму коді. Для того щоб подолати можливі помилкові ситуації в коді C#, програма зазвичай розділяється на блоки трьох різних типів:

try – блок, який інкапсулює код, що формує частину звичайних дій програми, й у якому потенційно може виникнути помилкова ситуація;

catch – блок, що інкапсулює код, який обробляє помилкові ситуації, що відбуваються в коді блоку try. Це також зручне місце для протоколювання помилок;

finally – блок, який інкапсулює код, який очищує будь-які ресурси або виконує інші дії, що зазвичай потрібно виконати в кінці блоків try або catch. Важливо розуміти, що цей блок виконується незалежно від того, було порушено виключення чи ні. Оскільки метою блоку finally є виконання коду очищення, який повинен бути виконаний у будь-якому випадку, компілятор видасть помилку, якщо ви помістите оператор return всередину блоку finally. Наприклад, у блоці finally ви можете закрити будь-яке з'єднання, яке було відкрито в блоці try. Однак блок finally необов'язковий. Якщо у вас немає вимог щодо очищення коду, то в цьому блоці немає необхідності.

Розглянемо рекомендації щодо об'єднання цих трьох блоків для перехоплення помилкових умов:

1) потік управління спочатку входить у блок try;

2) якщо у блоці try не виникає ніяких помилок, виконання триває нормально до кінця блоку та після завершення передається в блок finally, якщо такий присутній. Однак якщо в блоці try виникає помилка, потік управління переходить на блок catch;

3) помилкова умова обробляється в блоці catch;

4) наприкінці блоку catch управління автоматично передається в блок finally, якщо такий присутній;

5) виконується блок finally (якщо такий є).

У загальному вигляді синтаксис мовою C# буде мати такий вигляд:

```
try { // код при нормальному виконанні }
catch {  // обробка помилок }
finally { // очистка }
```

Існує кілька варіантів цієї конструкції:

1) програміст може пропустити блок finally, оскільки він не обов'язковий;

2) програміст може застосувати бажану кількість блоків catch для обробки специфічних типів помилок. Однак надмірна кількість блоків може знизити продуктивність створюваного застосунку;

3) можна пропустити всі блоки catch: синтаксис слугує не для ідентифікації винятків, а як спосіб гарантії виконання коду блоку finally, коли потік управління покине блок try. Це зручно, якщо в блоці try присутні кілька точок виходу.

Проте необхідно з'ясувати, коли виконання входить у блок try, як воно дізнається, що треба перемкнутися на блок catch, якщо виникне помилка? З виявленням помилки код активізує *збудження винятку*, створюючи екземпляр об'єкта винятку і активізуючи його:

```
throw new OverflowException();
```

Тут створюється екземпляр об'єкта класу винятку OverflowException. Досягши оператора throw усередині блоку try, комп'ютер негайно шукає блок catch, асоційований з блоком try. Якщо є більше одного блоку catch, асоційованого з цим try, він ідентифікує коректний catch-блок, перевіряючи, який клас винятків перехоплює кожен з них. Наприклад, коли генерується об'єкт OverflowException, потік виконання переходить на наступний блок catch:

```
catch (OverflowException ex)
        {/ / Тут йде обробка винятку }
```

Іншими словами, комп'ютер шукає такий блок catch, який приймає об'єкт-виняток відповідного класу (або його базового класу).

З урахуванням сказаного можемо розширити продемонстрований блок try. Припустимо, що в блоці try можуть виникнути дві серйозні помилки: переповнення і вихід за межі масиву. Припустимо, що наш код містить дві bool змінних – Overflow та OutOfBounds, які вказують на наявність цих умов. Для індикації переповнювання існує визначених клас винятків (OverflowException); аналогічно, на випадок виходу за межі масиву передбачений клас виключення IndexOutOfRangeException.

Тепер блок try буде виглядати так:

```
try
{// код для нормального виконання
        if  (Overflow==true)
        {
          throw new OverflowException();
        }
//подальша обробка
        if  (OutOfBounds==true)
        {throw new IndexOutOfRangeException();}
}
        //в протилежному випадку продовжується нормальна обробка
catch (OverflowException ex)
{
//обробка помилок переповнення
}
catch (IndexOutOfRangeException ex)
{
// обробка помилок виходу за межу масиву
}
finally
{
// очистка
}
```

C# є потужним і гнучким механізмом обробки помилок. Оператор throw може бути вкладений послідовно у кілька викликів методів усередині блоку try. Проте цей блок try залишається активним, навіть коли потік управління входить до вкладених методів. Якщо комп'ютер зустрічає оператор throw, то негайно передає управління назад – через усі вкладені виклики методів стеку, намагаючись відшукати кінець блоку try та запустити відповідний блок catch. Протягом цього процесу всі локальні змінні проміжних викликів методів коректно залишають контекст. Це робить

архітектуру try…catch придатною до ситуацій, коли помилка виникає всередині виклику методу, який має п'ятнадцятий або двадцятий рівень вкладеності. У цьому разі обробка повинна припинитися негайно.

Отже, блоки try можуть відігравати істотну роль у контролі потоку управління коду. Однак важливо розуміти, що виключення призначені для виняткових умов, про що свідчить їх назва. Не варто користуватися ними як способом управління виходом із циклів.

System.Exception

Часто бібліотечний код збуджує винятки, якщо стикається з якимись проблемами, або коли метод викликається неправильно, або передаються неправильні параметри. Однак бібліотечний код рідко намагається перехопити винятки. Це розглядається як сфера відповідальності клієнтського коду.

Під час налагодження може траплятись так, що виняток збуджується в бібліотеці базових класів. Процес налагодження в деяких випадках передбачає визначення причин виникнення винятків з тим, щоб позбутися їх. Тоді програміст має домогтися, щоб до моменту поставки коду споживачеві було гарантовано, що винятки виникатимуть у рідкісних випадках і, наскільки можливо, оброблятимуться у коді розумним чином.

У System.Exception є багато властивостей, що розглянуті в табл. 8.1.

Таблиця 8.1

Властивості System.Exception

Властивості	Опис
Data	Надає можливість додавати конструкції "ключ – значення" до виключень, які можуть бути використані для постачання винятків деякою додатковою інформацією
HelpLink	Зв'язок з довідковим файлом, в якому подана більш детальна інформація про виняток
InnerException	Якщо виняток було порушено всередині блоку catch, то InnerException містить об'єкт винятку, який був посланий в цей catch-блок
Message	Текст, що описує умова помилки
Source	Ім'я програми або об'єкта, який викликав виняток
StackTrace	Інформація про виклики методів у стек (для того щоб допомогти в пошуку методу, що збуджує виняток)
TargetSite	Об'єкт рефлексії .NET, що описує метод, який збудив виняток

З усіх цих властивостей тільки StackTrace і TargetSite застосовуються автоматично виконавчим середовищем .NET, якщо доступне трасування стеку. Source завжди заповнюється виконавчою системою .NET ім'ям збірки, в якій виникло виняток (хоча програміст може модифікувати цю властивість у своєму коді, щоб надати більш специфічну інформацію). Водночас Data, Message, HelpLink і InnerException повинні бути заповнені кодом, який був порушений винятком, шляхом установлення потрібних значень безпосередньо перед збудженням винятку.

Наприклад, код, який збуджує виняток, може виглядати так:

```
if (ErrorCondition == true)
{
Exception myException = new ClassmyException ("Help!!!");
myException.Source = ("ім'я мого застосунку");
myException.HelpLink = "MyHelpFile.txt";
myException.Data["ErrorDate"] = DateTime.Now;
myException.Data.Add("Додаткова інформація");
throw myException;
}
```

Тут ClassMyException – ім'я класу, який збуджується винятком. Можна зазначити, що класи винятків прийнято іменувати із закінченням Exception. Також зверніть увагу, що значення властивості Data може присвоюватися двома можливими способами.

Що відбувається з необробленими винятками?

Іноді виключення може бути порушено, але у вашому коді немає відповідного блоку catch, який би обробляв винятки подібного роду. Приклад SimpleExceptions може слугувати ілюстрацією сказаного. Припустимо, що ми опустили catch-блок для FormatException і catch-блок без параметрів, а залишили тільки блок, який бере IndexOutOfRangeException. Що відбудеться у випадку виникнення винятку FormatException? Відповідь: його перехопить виконавча система .NET. Це аналогічно вкладанню блоків try один в одний. Виконавча система .NET поміщає всю оброблювану програму всередину величезного блоку try, проводячи процедуру робить це з кожною програмою .NET. Блок try має оброблювач catch, який може перехопити виняток будь-якого типу. Якщо трапляється виняток, якого не обробляє ваша програма, то потік винятку передається з програми

та перехоплюється catch-блоком виконавчої системи .NET. Однак результат може бути незадовільним. Виконання такого коду буде перервано, і користувач побачить діалогове вікно зі вказівкою на те, що код не обробив виняток, і з інформацією про виняток, яку виконавча система .NET зможе витягти. Але важливо те, що виняток буде перехоплено.

Зауваження. Програміст, який пише програму, повинен намагатися перехопити максимальну кількість винятків і обробити їх осмисленим чином. Якщо ж він пише бібліотеку, то доцільніше не перехоплювати винятки. Їх перехопить і обробить викликаний програмою код. Проте краще перехоплювати винятки, визначені в .NET Framework, і створювати власні з більш специфічною інформацією клієнтського коду.

Оброблення винятків у мові Java

У Java SE також є багато стандартних класів винятків. Головні базові класи наведено на рис. 8.1.

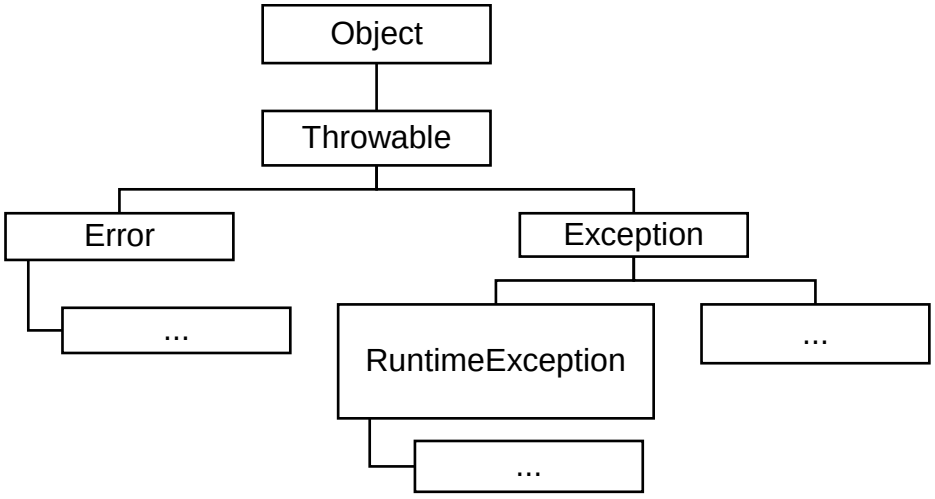

Рис. 8.1. **Базові класи системи обробки винятків Java SE**

Предком усіх класів помилок і винятків є *клас java.lang.Throwable*. Тільки об'єкти цього класу або його нащадків генеруються віртуальною машиною у разі виникнення помилкових ситуацій. Він містить елементи, які визначають загальні параметри та поведінку для всіх похідних класів. Основні методи класу java.lang.Throwable наведено в табл. 8.2.

Клас java.lang.Error – базовий для всіх помилок, які є зовнішніми для програми. Вони зазвичай виникають через несправне обладнання, наприклад, жорсткий диск тощо. Практично ніколи не обробляються в Java-програмах.

Таблиця 8.2

Основні методи класу java.lang.Throwable

Ім'я методу	Призначення
printStackTrace	Виводить об'єкт виняток і трасу стека до стандартного потоку помилок
getMessage	Повертає детальний рядок повідомлення про виняток
getLocalizedMessage	Створює локалізований опис проблеми
toString	Повертає короткий опис винятку

Клас java.lang.Exception є предком усіх класів винятків, у тому числі тих, що визначаються програмістом.

Винятки виконання програми – це нащадки класу *java.lang.RuntimeException*. Як правило, вони з'являються внаслідок багів у програмі. Програма може іноді обробляти таки винятки, але часто кращим рішенням є усунення проблем в її коді.

Усі винятки розподіляють на дві категорії: перевірювані та неперевірювані.

Неперевірювані винятки – це помилки та винятки виконання програми.

Перевірювані винятки є прямими нащадками класу java.lang.Exception. Здебільшого це помилкові ситуації, що можуть з'явитися під час виконання програми; їх дуже важко передбачити під час її розроблення. Їх рекомендується завжди обробляти в коді програми.

Механізми обробки винятків у Java SE (як і у .NET) передбачають використання ключових слів *try*, *catch* і *finally* в аналогічному контексті.

Проте існують деякі особливості:

1) синтаксис оброблення декількох типів винятків в одному блоці коду:

```
try {
……..……
} catch (NumberFormatException | FileNotFoundException caught) {
    System.out.print(caught.getMessage());
}
```

Тобто, якщо необхідно однаково обробляти винятки, можна використовувати символ "логічного і" замість декількох блоків *catch;*

2) передавання інформації про виняток до іншого контексту:

```
public void inputFile( String fName ) throws FileNotFoundException { }
```

Тут у визначенні методу можна зазначити, що в разі виникнення виняток буде переданий до викличного методу (ключове слово *throws*). Таким чином, для оброблення винятку в методі мають бути блоки *try* та *catch або* ключове слово *throws.*

Визначення власних винятків

Безпосередньо рекомендується, щоб класи таких винятків були похідними від класу java.lang.Exception.

Приклад 1. Розглянемо приклад програми, що визначає власний виняток CustomNumberFormatException та обробляє його:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

// Клас власного винятку
class CustomNumberFormatException extends Exception {

        public CustomNumberFormatException(Throwable t) {
                super(t);
        }
        // Превизначений метод класу Exception
        public String getLocalizedMessage() {
                return "Введіть ціле число!";
        }
}

public class MainClass {

        public static void main(String[] args) throws IOException {
                BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
                try {
                        try {
                                int num = Integer.parseInt(br.readLine());
                                System.out.format("Result=%1$5d", num);
                        } catch (NumberFormatException e) {
                                // Генерування власного винятку
                                throw new CustomNumberFormatException(e);
                        }
                } catch (CustomNumberFormatException e) {
                        // Обробка власного винятку
```

```
                System.out.println(e.getLocalizedMessage());
            }
        }
    }
```

У цій програмі визначено клас CustomNumberFormatException власного винятку. У ньому перевизначається метод getLocalizedMessage базового класу з метою задання повідомлення про виняток CustomNumberFormatException.

У методі main спочатку обробляється створений стандартний виняток NumberFormatException, якщо значення, введене користувачем, не є цілим числом. Обробка полягає в генеруванні винятку CustomNumberFormatException. У результаті викликається перевизначений у цьому класі метод getLocalizedMessage, який повертає бажане повідомлення.

Код створення власного винятку мовою C# є аналогічним.

Питання для самопідготовки

1. Розроблення власних класів винятків.
2. Генерування та перехоплення користувальницьких винятків.

Запитання для самодіагностики

1. Які проблеми має традиційний підхід до оброблення помилок під час виконання програми?
2. Які переваги має оброблення винятків порівняно із традиційним підходом до оброблення помилок?

Література: [9; 11; 13; 18; 23; 24].

9. Стандартні бібліотеки класів середовищ розробки програм

Мета теми: набуття знань щодо основних структур даних стандартної бібліотеки колекцій Microsoft .NET і Java SE, основних абстракцій стандартної бібліотеки введення та виведення.

Професійні компетентності: Уявляти особливості різних видів колекцій, використовувати потоки введення та виведення.

Основні питання:

9.1. Призначення та застосування регулярних виразів. Підтримка регулярних виразів на платформах Microsoft .NET і Java SE. Спеціальні символи, використовувані у регулярних виразах.

9.2. Загальні відомості про колекції. Основні структури даних стандартних бібліотек колекцій Microsoft .NET і Java SE. Типізовані колекції.

9.3. Джерела та споживачі даних. Загальні відомості про потоки введення – виведення даних. Алгоритми роботи потоків введення – виведення даних. Основні класи стандартних бібліотек Microsoft .NET і Java SE для підтримки введення – виведення даних. Серіалізація.

Питання для опрацювання: призначення та застосування регулярних виразів, загальні відомості про колекції, типізовані колекції, загальні відомості про потоки введення – виведення даних, алгоритми роботи потоків введення – виведення даних, серіалізація.

Ключові слова: рядок, регулярний вираз, шаблон, метасимволи, фрагмент, повторювачі, колекції, нумератор, інтерфейси колекції, список, геш-таблиця, стек, черга, динамічний масив, словник, множина, FIFO, LIFO, безпека типів, потік, серіалізація, форматування, байт, файл, ASCII, Unicode.

9.1. Призначення та застосування регулярних виразів. Підтримка регулярних виразів на платформах Microsoft .NET і Java SE. Спеціальні символи, використовувані у регулярних виразах

Стандартний клас string дозволяє виконувати над рядками різні операції, зокрема пошук, заміну, вставляння та видалення підрядків. Проте є класи завдань з обробки символьної інформації, де стандартних можливостей явно не вистачає. Щоб полегшити вирішення подібних завдань, у .Net Framework убудований могутній апарат роботи з рядками, заснований на регулярних виразах.

Регулярні вирази, призначені для обробки текстової інформації, забезпечують:

ефективний пошук у тексті за заданим шаблоном;

редагування тексту;

формування підсумкових звітів за наслідками роботи з текстом.

Детально розглянемо перші два аспекти застосування регулярних виразів.

Метасимволи в регулярних виразах

Регулярний вираз – це шаблон, за яким виконується пошук відповідного фрагмента тексту. На платформах .NET і Java SE використовується синтаксис мови регулярних виразів, який визначено в мові програмування Perl.

Мова опису регулярних виразів складається з символів двох видів: звичайних символів і метасимволів. **Звичайний символ** представляє у виразі сам себе, а **метасимвол** – деякий клас символів. Найбільш споживані метасимволи наведені в табл. 9.1.

Таблиця 9.1

Метасимволи в регулярних виразах

Класи символів	Опис	Приклад
.	Будь-який символ, окрім \n	Вираз c.t відповідає фрагментам: cat, cut, c#t, c{t і т. д.
[ ]	Будь-який одиночний символ з послідовності, записаної усередині дужок. Допускається використання діапазонів символів	Вираз c [aui]t відповідає фрагментам: cat, cut, cit. Вираз c[a-c]t відповідає фрагментам: cat, cbt, cct
[^]	Будь-який одиночний символ, що не входить у послідовність, записану всередині дужок. Допускається використання діапазонів символів	Вираз c [^aui]t відповідає фрагментам: cbt, cct, c2t і т. д. Вираз c[^a-c]t відповідає фрагментам: cdt, cet, c%t і т. д.
\w	Будь-який алфавітно-цифровий символ	Вираз c\wt відповідає фрагментам: cbt, cct, c2t і т. д., але не відповідає фрагментам c%t, c{t і т. д.
\W	Будь-який неалфавітно-цифровий символ	Вираз c\Wt відповідає фрагментам: c%t, c{t, c.t і т. д., але не відповідає фрагментам cbt, cct, c2t і т. д.
\s	Будь-який пробільний символ	Вираз \s\w\w\w\s відповідає будь-якому слову з трьох букв, оточеному пробільними символами
\S	Будь-який непробільний символ	Вираз \s\S\S\S\s відповідає будь-яким трьом непробільним символам, оточеним пробільними
\d	Будь-яка десяткова цифра	Вираз c\dt відповідає фрагментам: c1t, c2t, c3t і т. д.
\D	Будь-який символ, що не є десятковою цифрою	Вираз c\Dt не відповідає фрагментам: c1t, c2t, c3t і т. д.

Окрім метасимволів, що позначають класи символів, можуть застосовуватися *уточнювальні метасимволи*, які наведені в табл. 9.2.

Таблиця 9.2

Уточнювальні метасимволи в регулярних виразах

Уточнювальні символи	Опис
^	Фрагмент, що співпадає з регулярними виразами, слід шукати тільки на початку рядка
$	Фрагмент, що співпадає з регулярними виразами, слід шукати тільки в кінці рядка
\A	Фрагмент, що співпадає з регулярними виразами, слід шукати тільки на початку багаторядкового рядка
\Z	Фрагмент, що співпадає з регулярними виразами, слід шукати тільки в кінці багаторядкового рядка
\b	Фрагмент, що співпадає з регулярними виразами, починається або закінчується на межі слова, тобто між символами, відповідними метасимволам \w і \W
\B	Фрагмент, що співпадає з регулярними виразами, не повинен зустрічатися на межі слів

У регулярних виразах часто використовуються *повторювачі* (табл. 9.3) – метасимволи, які розташовуються безпосередньо після звичайного символу або групи символів і задають кількість його повторень у виразі.

Таблиця 9.3

Повторювачі

Повторювачі	Опис	Приклад
1	2	3
*	Нуль або більше за повторення попереднього елемента	Вираз ca*t відповідає фрагментам: ct, cat, caat, caaat і т. д.
+	Одне або більше за повторення попереднього елемента	Вираз ca+t відповідає фрагментам: cat, caat, caaat і т. д.
?	Не більше за одне повторення попереднього елементу	Вираз ca?t відповідає фрагментам: ct, cat
{n}	Рівно n повторень попереднього елемента	Вираз ca{3}t відповідає фрагменту: caaat. Вираз (cat){2} відповідає фрагменту: catcat

Закінчення табл. 9.3

1	2	3
{n,}	Принаймні n повторень попереднього елемента	Вираз ca{3,}t відповідає фрагментам: caaat, caaaat, caaaaaaat і т. д. Вираз (cat){2,} відповідає фрагментам: catcat, catcatcat і т. д.
{n, m}	От n до m повторень попереднього елемента	Вираз ca{2, 4}t відповідає фрагментам: caat, caaat, caaaat

Регулярний вираз записується у вигляді *рядкового літерала*, причому перед рядком необхідно ставити символ @. Це означатиме, що рядок потрібно буде розглядати і в тому випадку, якщо він займатиме на екрані декілька рядків. Проте символ @ можна не ставити, якщо використовується як шаблон без метасимволів.

Зауваження. Якщо потрібно знайти якийсь символ, який є метасимволом (наприклад, крапку), це можна зробити, захистивши його зворотним слешем. Тобто просто крапка означає будь-який одиночний символ, а \. означає просто крапку.

Приклади регулярних виразів:

1) слово ukr – @"ukr" або "ukr";

2) номер телефону у форматі xxx-xx-xx – @"\d\d\d-\d\d-\d\d" або @"\d{3}(-\d\d){2}";

3) номер автомобіля - @"[A-Z]\d{3}[A-Z]{2}\d{2,3}UKR".

Пошук у тексті за шаблоном

Простір імен бібліотеки базових класів System.Text.RegularExpressions містить усі об'єкти платформи .NET Framework, що стосуються регулярних виразів. Найважливішим класом, що підтримує регулярні вирази, є клас Regex – відкомпільовані незмінні регулярні вирази. Для опису регулярного виразу в класі визначено кілька *переобтяжених конструкторів*:

1) Regex() – створює порожній вираз;

2) Regex(String) – створює заданий вираз;

3) Regex(String, RegexOptions) – створює заданий вираз і задає параметри для його обробки за допомогою елементів перерахування RegexOptions (наприклад, розрізняти чи не розрізняти великі та малі букви).

Пошук фрагментів рядка, відповідних заданому виразу, виконується за допомогою методів IsMach, Mach, Matches класу Regex.

Метод ***IsMach*** повертає true, якщо фрагмент, відповідний виразу, в заданому рядку знайдений, і false – інакше.

Приклад 1. Спробуємо визначити, чи зустрічається в заданому тексті слово *собака*:

```
static void Main()
    {
       Regex r = new Regex("собака",RegexOptions.IgnoreCase);
       string text1 = "Кіт в будинку, собака в будці.";
       string text2 = "Котик в будинку, собачка в будці.";
       Console.WriteLine(r.IsMatch(text1));
       Console.WriteLine(r.IsMatch(text2));
     }
```

Зауваження. RegexOptions.IgnoreCase – означає, що регулярний вираз застосовується без урахування регістру символів.

Можна використовувати конструкцію вибору з декількох елементів. Варіанти вибору перераховуються за допомогою символу " ".

Приклад 2. Спробуємо визначити, чи зустрічаються в заданому тексті слова *собака* або *кіт*:

```
static void Main(string[] args)
    {
       Regex r = new Regex("собака кіт",RegexOptions.IgnoreCase);
       string text1 = "Кіт в будинку, собака в будці.";
       string text2 = "Котик в будинку, собачка в будці.";
       Console.WriteLine(r.IsMatch(text1));
       Console.WriteLine(r.IsMatch(text2));
    }
```

Приклад 3. Спробуємо визначити, чи є в заданих рядках номера телефону у форматі xx-xx-xx або xxx-xx-xx:

```
static void Main()
    {
       Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
       string text1 = "tel:123-45-67";
       string text2 = "tel:no";
       string text3 = "tel:12-34-56";
       Console.WriteLine(r.IsMatch(text1));
       Console.WriteLine(r.IsMatch(text2));
```

```
        Console.WriteLine(r.IsMatch(text3));
    }
```

Метод ***Match*** класу Regex не просто визначає, чи міститься текст, відповідний шаблону, а повертає об'єкт класу Match – послідовність фрагментів тексту, що збіглися з шаблоном.

Приклад 4. Необхідно знайти всі номери телефонів у вказаному фрагменті тексту:

```
static void Main()
    {
      Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
      string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-45
                      Контакти у Харкові tel:12-34-56; fax:12-56-45";
      Match tel = r.Match(text);
      while (tel.Success)
       {
         Console.WriteLine(tel);
         tel = tel.NextMatch();
       }
    }
```

Приклад 5. Необхідно підрахувати суму цілих чисел, що зустрічаються в тексті:

```
static void Main()
    {
       Regex r = new Regex(@"[-+]?\d+");
       string text = @"5*10=50 -80/40=-2";
       Match teg = r.Match(text);
       int sum = 0;
       while (teg.Success)
       {
          Console.WriteLine(teg);
          sum += int.Parse(teg.ToString());
          teg = teg.NextMatch();
       }
       Console.WriteLine("sum=" + sum);
   }
```

Метод ***Matches*** класу Regex повертає об'єкт класу MatchCollection – колекцію всіх фрагментів заданого рядка, що збіглися з шаблоном.

```
static void Main(string[] args)
        {
            string text = @"5*10=50 -80/40=-2";
            Regex theReg = new Regex(@"[-+]?\d+");
            MatchCollection theMatches = theReg.Matches(text);
            foreach (Match theMatch in theMatches)
            {
                Console.Write("{0} ", theMatch.ToString());

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

Водночас метод Matches багато разів запускає метод Match, кожного разу починаючи пошук з того місця, на якому закінчився попередній пошук.

Редагування тексту

Регулярні вирази можуть ефективно використовуватися для редагування тексту. Наприклад, метод Replace класу Regex дозволяє виконувати заміну одного фрагмента тексту іншим або видалення фрагментів тексту.

Приклад 6. Зміна номерів телефонів:

```
static Main(string[] args)
{
  string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-45.
            Контакти у Харкові tel:12-34-56; fax:11-56-45";
  Console.WriteLine("Старі дані\n"+text);
  string newText=Regex.Replace(text, "123-", "890-");
  Console.WriteLine("Нові дані\n" + newText);
}
```

Приклад 7. Видалення всіх номерів телефонів з тексту:

```
static void Main(string[] args)
{
  string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-45.
            Контакти у Харкові tel:12-34-56; fax:11-56-45";
  Console.WriteLine("Старі дані\n"+text);
  string newText=Regex.Replace(text @"\d{2,3}(-\d\d){2}", "");
  Console.WriteLine("Нові дані\n" + newText);
}
```

Приклад 8. Розбиття початкового тексту на фрагменти:

```
static void Main () {
  string text = @"Контакти у Києві tel:123-45-67, 123-34-56; fax:123-56-45.
                Контакти у Харкові tel:12-34-56; fax:11-56-45";
  string []newText=Regex.Split(text"[,.:;]+");
  foreach( string a in newText)
  Console.WriteLine(a);
}
```

У прикладі наведено методи використання регулярних виразів.

Регулярні вирази в Java

Для використання регулярних виразів у Java SE необхідно імпортувати в програму пакет java.util.regex. У ньому є класи Pattern та Matcher.

Регулярний вираз – це шаблон для пошуку, який є рядком символів мовою регулярних виразів. Спочатку він має скомпільований компілятором регулярних виразів об'єкт класу Pattern, який є результатом компіляції регулярного виразу в пам'яті програми. Об'єкти класу Matcher призначені для зіставлення цільової символьної послідовності з регулярним виразом. Головні методи цих класів наведено в табл. 9.4 і 9.5.

Таблиця 9.4

Головні методи класа Pattern

Сигнатура методу	Опис
Pattern compile(String regex)	Статичний метод. Компілює регулярний вираз regex у об'єкт Pattern та повертає його
Matcher matcher(CharSequence input)	Створює об'єкт класу Matcher для перевірки співпадіння текстової послідовності input із шаблоном регулярного виразу
boolean matches (String regex,CharSequence input)	Статичний метод. Компілює регулярний вираз regex і намагається перевірити співпадіння текстової послідовності input і шаблону регулярного виразу. Повертає булеве значення – результат перевірки
Matcher String[ ] split(CharSequence input)	Поділяє текстову послідовність input на частки відповідно шаблону регулярного виразу. Повертає масив часток

Таблиця 9.5

Головні методи класа Matcher

Сигнатура методу	Опис
boolean find()	Намагається знайти наступну підпослідовність, яка відповідає шаблону регулярного виразу, у вхідної послідовності. Повертає булевий результат пошуку
String group()	Повертає вхідну підпослідовність, яка відповідає попередньому співпадінню
int groupCount()	Повертає кількість груп вхідної послідовності, що відповідають шаблону регулярного виразу
int start()	Повертає початковий індекс попереднього співпадіння
int end()	Повертає the offset after the last character matched
Matcher reset()	Скидає об'єкт Matcher до початкового стану та повертає його

Розглянемо приклади Java-програм, які використовують регулярні вирази.

Приклад 9. Визначення приналежності вхідної символьної послідовності до телефонного номеру формату (0XX)XXX-XX-XX, де X – арабська цифра.

```
import java.util.Scanner;
import java.util.regex.Pattern;

public class PhoneNumber {
   public static void main(String[] s) {
      // телефонный номер
      String pattern = "^\\(0\\d{2}\\)[1-9](\\d{2}-){2}\\d{2}$";
      Scanner sc = new Scanner(System.in);
      for (;;) {
         System.out.println("Введіть номер телефона");
         String str = sc.next();
         if (Pattern.matches(pattern, str))
            System.out.println("Це телефонний номер\n");
        else
            System.out.println("Неприпустимий формат даних!\n");
      }
   }
}
```

Подвійний зворотній слєш використовується для екранування символів, які Java-компилятор вважає іскейп-послідовностями.

Приклад 10. Пошук у вхідній символьній послідовності всіх підпослідовностей, що складаються з одного чи кількох символів **<** або **>** або **{** або **}** та їх сполучень, за якими може слідувати один або декілька символів @, і закінчуються символом**;**

```
import java.io.*;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class CharSequenceMainClass {

	public static void main(String[ ] args) throws IOException {
		try (Scanner sc = new Scanner(System.in)) {
			sc.useDelimiter(System.getProperty("line.separator"));
			String pattern = "(<|>|\\{|\\})+@*;";
			for (;;) {
				System.out.println("Введите текст:");
				String str = sc.next();
				Pattern p = Pattern.compile(pattern);
				Matcher m = p.matcher(str);
				int i = 0;
				while (m.find()) {
					System.out.println("Элемент " + (i + 1) + ": " +
	m.group());
					i++;
				}
			System.out.println("Количество найденных элементов: " + i);
			}
		}
	}
}
```

У прикладі наведено методи використання регулярних виразів.

9.2. Загальні відомості про колекції. Основні структури даних стандартних бібліотек колекцій Microsoft .NET і Java SE. Типізовані колекції

Під **колекцією** розуміють групу об'єктів. Простір імен System.Collections містить безліч інтерфейсів і класів, які визначають і реалізують колекції різних типів. Колекції спрощують програмування, пропонуючи вже готові

рішення для побудови структур даних, розроблення яких "з нуля" відрізняється великою трудомісткістю. Мова йде про убудовані колекції, які підтримують, наприклад, функціонування стеків, черг і геш-таблиць.

Основна перевага колекцій полягає в тому, що вони стандартизують спосіб обробки груп об'єктів у прикладних програмах. Усі колекції розроблені на основі набору чітко певних інтерфейсів. Ряд вбудованих реалізацій таких інтерфейсів, як ArrayList, Hashtable, Stack і Queue, можна використовувати "як є". У кожного програміста також є можливість реалізувати власну колекцію, але в більшості випадків вистачає убудованих.

Платформа Microsoft .NET Framework підтримує три основних **типи колекцій**:

колекції загального призначення реалізують ряд основних структур даних, включаючи динамічний масив, стек і чергу. Сюди також відносять словники, призначені для зберігання пар ключ/значення. Колекції загального призначення працюють із даними типу object, тому їх можна використовувати для зберігання даних будь-якого типу;

колекції спеціального призначення орієнтовані на обробку даних конкретного типу або на обробку унікальним способом. Наприклад, існують спеціалізовані колекції, призначені тільки для обробки рядків або односпрямованого списку;

класи колекцій, орієнтованих на побітову організацію даних, слугують для зберігання груп бітів. Колекції цієї категорії підтримують такий набір операцій, що не характерний для колекцій інших типів. Наприклад, у біт-орієнтованої колекції Bit Array визначені такі побітові операції, як "і" та "або".

Основним для всіх колекцій є реалізація *перечислювача (нумератора),* що підтримується інтерфейсами IEnumerator і IEnumerable. Перечислювач забезпечує стандартизований спосіб поелементного доступу до вмісту колекції. Оскільки кожна колекція повинна реалізувати інтерфейс IEnumerable, до елементів будь-якого класу колекції можна отримати доступ за допомогою методів, визначених в інтерфейсі IEnumerator. Отже, після внесення невеликих змін код, що дозволяє циклічно опитувати колекцію одного типу, для циклічного опитування можна успішно використовувати колекції іншого типу.

Простір імен System.Collections містить класи й інтерфейси, які визначають різні колекції об'єктів. Основні класи й інтерфейси колекцій наведені в табл. 9.6.

Таблиця 9.6

Основні класи та інтерфейси колекцій

Класи	Опис
ArrayList	Слугує для реалізації інтерфейсу IList за допомогою масиву з динамічною зміною розміру на вимогу
BitArray	Управляє компактним масивом двійкових значень, представлених логічними величинами, де значення true відповідає 1, а значення false відповідає 0
CaseInsensitiveComparerpa	Перевіряє рівність двох об'єктів без урахування регістру рядків
CaseInsensitiveHashCodeProvider	Надає геш-код об'єкта, використовуючи алгоритм гешування, за якого не враховується регістр рядків
CollectionBase	Надає абстрактний базовий клас для колекції зі строгим типом
Comparer	Перевіряє рівність двох об'єктів з урахуванням регістра рядків
DictionaryBase	Надає абстрактний базовий клас для колекції пар "ключ - значення" зі строгим типом
Hashtable	Надає колекцію пар "ключ – значення", які впорядковані за геш-кодом ключа
Queue	Надає колекцію об'єктів, що обслуговується за принципом "першим прийшов – першим вийшов" (FIFO)
ReadOnlyCollectionBase	Надає абстрактний базовий клас для колекції зі строгим типом, що доступна тільки для читання
SortedList	Надає колекцію пар "ключ – значення", які впорядковані за ключами. Доступ до пар можна отримати за ключем і за індексом
Stack	Представляє колекцію об'єктів, що обслуговується за принципом "останнім прийшов – першим вийшов" (LIFO)
ICollection	Визначає розмір, перечислювачі та методи синхронізації для всіх колекцій
IComparer	Надає іншим застосункам метод для порівняння двох об'єктів
IDictionary	Надає колекцію пар "ключ – значення"
IDictionaryEnumerator	Здійснює нумерацію елементів словника
IEnumerable	Надає перечислювач, що підтримує просте переміщення колекцією
IEnumerator	Підтримує просте переміщення колекцією
IHashCodeProvider	Надає геш-код об'єкта, використовуючи користувальницьку геш-функцію
IList	Надає колекцію об'єктів, до яких можна отримати доступ окремо, за індексом

Інтерфейси колекцій

У просторі імен System.Collections міститься безліч інтерфейсів. Розглянемо інтерфейси колекцій, оскільки вони визначають функції, загальні для всіх класів колекцій.

Інтерфейс ICollection

Інтерфейс ICollection можна назвати фундаментом, на якому побудовані всі колекції. У ньому оголошені основні методи та властивості, які необхідні кожній колекції. Він успадковує інтерфейс IEnumerable. Не знаючи сутності інтерфейсу ICollection, неможливо зрозуміти механізм дії колекції.

В інтерфейсі ICollection визначені такі властивості:

int Count { get; } – кількість елементів колекції в поточний час;

bool isSynchronized { get; } – приймає значення true, якщо колекція синхронізована, та значення false у протилежному випадку.

За замовчуванням колекції не синхронізовані. Але для більшості колекцій можна отримати синхронізовану версію: object syncRoot { get; } – об'єкт, для якого колекція може бути синхронізована.

Властивість Count – найбільш затребувана, оскільки містить кількість елементів, збережених у колекції в поточний час. Якщо властивість Count дорівнює нулю, виходить, колекція порожня. В інтерфейсі ICollection визначений такий метод:

void CopyTo (Array *target,* mt *startIdx).*

Метод CopyTo () копіює вміст колекції в масив, заданий параметром *target,* починаючи з індексу, заданого параметром startIdx. Можна сказати, що метод CopyTo () забезпечував перехід від колекції до стандартного масиву.

Оскільки інтерфейс ICollection успадковує інтерфейс IEnumerable, він також включає його єдиний метод GetEnumerator ( ) :

IEnumerator GetEnumerator()

Цей метод повертає нумератор колекції.

Інтерфейс IList

Інтерфейс IList успадковує інтерфейс ICollection і визначає поводження колекції, доступ до елементів якої дозволений за допомогою індексу

з відліком від нуля. Крім методів, визначених в інтерфейсі ICollection, інтерфейс Ilist визначає власні методи. Деякі з них слугують для модифікації колекції. Якщо ж колекція призначена тільки для читання або має фіксований розмір, виклик цих методів призведе до генерування виключення типу NotSupportedException.

Інтерфейс IDictionary

Інтерфейс IDictionary визначає поводження колекції, що встановлює відповідність між унікальними ключами та значеннями. **Ключ** – це об'єкт, використовуваний для отримання відповідного йому значення. Отже, колекція, що реалізує інтерфейс IDictionary, призначена для зберігання пар "ключ – значення". Збережену один раз пару можна потім витягти за заданим ключем. Інтерфейс IDictionary успадковує інтерфейс ICollection.

Класи колекцій загального призначення

Класи колекцій розподіляють на три основних **категорії:** загального призначення, спеціалізовані й орієнтовані на побітову організацію даних. Класи загального призначення можна використовувати для зберігання об'єктів будь-якого типу (табл. 9.7). Бітові призначені для зберігання бітової інформації. Колекції спеціального призначення розробляються для обробки даних конкретного типу.

Таблиця 9.7

Класи колекцій загального призначення

Класи	Опис
Stack	Стек – окремий випадок односпрямованого списку, що діє за принципом "останнім прийшов – першим вийшов"
Queue	Черга – окремий випадок односпрямованого списку, що діє за принципом "першим прийшов – першим вийшов"
ArrayList	Динамічний масив, який за необхідності може збільшувати свій розмір
Hash table	Геш-кодування – таблиця для пар "ключ – значення"
SortedList	Відсортований список пар "ключ – значення"

Розглянемо зазначені колекції детальніше.

Зауваження. *Абстрактний тип даних (АТД) список – це послідовність елементів a_1, a_2, a_n ($n \geq 0$) одного типу. Кількість елементів n називають довжиною списку. Якщо $n > 0$, то a_1 називають першим елементом списку, а a_n – останнім елементом списку. У разі $n = 0$ маємо порожній список, який не містить елементів.*

Важлива властивість списку полягає в тому, що його елементи лінійно впорядковані відповідно до їх позиції в списку. Так, елемент a_i передує a_{i+1} для i = 1, 2 .n – 1 і a_i слідує за a_{i-1} для i = 2 .n. Список буде односпрямованим, якщо кожен його елемент містить посилання на наступний елемент. Якщо кожен елемент списку містить два посилання (одне – на наступний елемент в списку, друге – на попередній елемент), то такий список є двоспрямованим (двозв'язковим). Якщо останній елемент зв'язати покажчиком з першим, то створиться кільцевий список.

Клас Stack

АТД стек – це окремий випадок односпрямованого списку, додавання елементів в який і вибірка елементів з якого виконуються з одного кінця, званого *вершиною стека* (головою – head). Шляхом вибірки елемент виключається із стека. Інші операції із стеком не визначені. Говорять, що стек реалізує принцип обслуговування LIFO (last in – fist out, "останнім прийшов, – першим вийшов"). Стек найпростіше уявити собі у вигляді піраміди, на яку надягають кільця (рис. 9.1). Дістати перше кільце можна тільки після того, як будуть зняті всі верхні кільця.

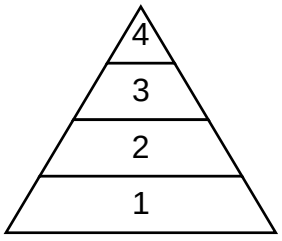

Рис. 9.1. **Подання стеку у вигляді піраміди**

У C# реалізації АТД стек представляє клас Stack, який реалізує інтерфейси ICollection, IEnumerable і ICloneable. Stack – це динамічна колекція, розмір якої змінюється.

У класі Stack визначені такі *конструктори:*

public Stack(); – створює порожній стек, початкова місткість якого дорівнює 10;

public Stack(int capacity); – створює порожній стек, початкова місткість якого дорівнює capacity;

public Stack(ICollection c); – створює стек, який містить елементи колекції, заданої параметром c.

Окрім методів, визначених в інтерфейсах, що реалізовуються класом Stack, у цьому класі визначені власні методи, які наведені в табл. 9.8.

Таблиця 9.8

Методи класу Stack

Методи	Опис
public virtual bool Contains(object v)	Повертає значення true, якщо об'єкт v міститься в стеку, що викликається, інакше – повертає значення false
public virtual void Clear()	Установлює властивість Count дорівненого нулю, тим самим очищаючи стек
public virtual object Peek()	Повертає елемент, розташований у вершини стека, але не витягуючи його із стека
public virtual object Pop()	Повертає елемент, розташований у вершини стека, і витягує його із стека
public virtual void Push(object v)	Поміщає об'єкт v у стек
public virtual object[] ToArray()	Повертає масив, який містить копії елементів стека, що викликається

Розглянемо декілька прикладів використання стека.

Приклад 11. Для заданого значення n запишемо в стек усі числа від 1 до n, а потім витягуватимемо із стека:

```
using System;
using System.Collections;
namespace ConsoleApplication
{
    class Program
    {
        public static void Main ()
        {
            Console.Write("n=");
            int n=int.Parse(Console.ReadLine());
            Stack intStack = new Stack();
            for (int i = 1; i <= n; i++)
                intStack.Push(i);
            Console.WriteLine("Розмірність стека " + intStack.Count);

            Console.WriteLine("Верхній елемент стека = " +
intStack.Peek());
            Console.WriteLine("Вміст стека " + intStack.Count);

            Console.Write("Вміст стека = ");
            while (intStack.Count != 0)
            Console.Write("{0} ", intStack.Pop());
```

```
                    Console.WriteLine("\nНова розмірність стека " +
                    intStack.Count);
                }
            }
        }
```

Приклад 12. У текстовому файлі міститься математичний вираз. Необхідно перевірити баланс круглих дужок у виразі.

```
        using System;
        using System.Collections;
        using System.IO;
        namespace MyProgram
        {
        class Program
        {
                public static void Main()
                {
                        StreamReader fileIn=new StreamReader("t.txt",);
                        string line=fileIn.ReadToEnd();
                        fileIn.Close();
                        Stack skobki=new Stack();
                        bool flag=true;
                    //перевіряємо баланс дужок
                        for ( int i=0; i<line.Length;i++)
                        {
        //якщо поточний символ дужка, що відкривається, то поміщаємо її в стек
                        if (line[i]== '(') skobki.Push(i);
          else if (line[i]== ')') // якщо поточний символ дужка, що //закривається, то
                        {
  //якщо стек порожній, то для дужки, що закривається, не вистачає парної, //що
відкривається
                        if (skobki.Count == 0)
           { flag = false; Console.WriteLine("Можливо в позиції " + i + "зайва )
дужка"); }
                          else skobki.Pop(); //інакше витягуємо парну дужку
                        }
                }
  //якщо після проглядання рядка стек виявився порожнім, то дужки
//збалансовані
      if (skobki.Count == 0) { if (flag) Console.WriteLine("дужки збалансовані"); }
                        else // інакше баланс дужок порушений
                        {
```

```
                    Console.Write("Можливо зайва ( дужка в позиції:");
                    while (skobki.Count != 0) {
                        Console.Write("{0} ", (int)skobki.Pop());
                    }
                    Console.WriteLine();
                }
            }
        }
        }
```

_____t.txt_____

(1+2)-4×(a-3)/(2-7+6)

У прикладі наведено використання контейнеру Stack.

Клас Queue

АТД черга – це окремий випадок односпрямованого списку, додавання елементів у який виконується в один кінець (хвіст), а вибірка проводиться з іншого кінця (голови). Інші операції з чергою не визначені. У ході вибірки елемент виключається з черги. Говорять, що чергу реалізує принцип обслуговування FIFO (fist in – fist out, "першим прийшов, – першим вийшов"). Чергу найпростіше зобразити у вигляді вузької труби, в один кінець якої кидають м'ячі, а з іншого кінця якої вони вилітають (рис. 9.2). Зрозуміло, що м'яч, який був кинутий в трубу першим, першим і вилетить з іншого кінця.

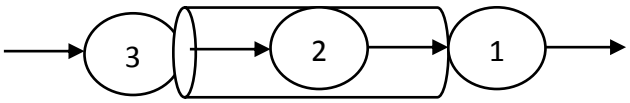

Рис. 9.2. **Подання черги**

У C# реалізацію АТД чергу представляє клас Queue, який також, як і стек, реалізує інтерфейси ICollection, IEnumerable і ICloneable. Queue – це динамічна колекція, розмір якої змінюється. За необхідності збільшення місткості черги відбувається з коефіцієнтом зростання за умовчанням, дорівненим 2.0.

У класі Queue визначені такі *конструктори*:

public Queue(); – створює порожню чергу, початкова місткість якої //дорівнює 32;

public Queue (int capacity); – створює порожню чергу, початкова місткість //якої дорівнює capacity;

public Queue (int capacity, float n); – створює порожню чергу, початкова //місткість якої дорівнює capacity, і коефіцієнт зростання //встановлюється параметром n;

public Queue (ICollection c); – створює чергу, яка містить елементи колекції, //заданої параметром c, і аналогічною місткістю.

Окрім методів, визначених в інтерфейсах, що реалізовуються класом Queue, в цьому класі визначені власні методи, які наведені в табл. 9.9.

Таблиця 9.9

Методи класу Queue

Методи	Опис
public virtual bool Contains (object v)	Повертає значення true, якщо об'єкт v міститься в черзі, що викликається, інакше – повертає значення false
public virtual void clear ()	Установлює властивість Count дорівненою нулю, тим самим очищаючи чергу
public virtual object Dequeue ()	Повертає об'єкт із початку черги, видаляючи його із черги
public virtual object Peek ()	Повертає об'єкт з початку черги, не видаляючи його із черги
public virtual void Enqueue(object v	Додає об'єкт v у кінець черги
public virtual object [ ] ToArray ()	Повертає масив, який містить копії елементів із черги
public virtual void TrimToSize()	Установлює властивість Capacity дорівненою значенню властивості Count

Розглянемо декілька прикладів використання стека.

Приклад 13. Для заданого значення n запишемо в чергу всі числа від 1 до n, а потім витягуватимемо їх із черги:

```
using System;
using System.Collections;
namespace MyProgram
{
class Program
{
        public static void Main()
        {
                Console.Write("n=");
                int n=int.Parse(Console.ReadLine());
                Queue intQ = new Queue();
                for (int i = 1; i <= n; i++)
```

```
                intQ.Enqueue(i);
            Console.WriteLine("Розмірність черги " + intQ.Count);

            Console.WriteLine("Верхній елемент черги = " + intQ.Peek());
            Console.WriteLine("Розмірність черги " + intQ.Count);

            Console.Write("Вміст черги = " );
            while (intQ.Count!=0)
                Console.Write("{0}  ", intQ.Dequeue());
            Console.WriteLine("\nНова розмірність черги " + intQ.Count);
        }
    }
    }
```

Приклад 14. У текстовому файлі через пропуск записана інформація про людей (прізвище, ім'я, по батькові, вік, вага). Необхідно вивести на екран спочатку інформацію про людей молодше 40 років, а потім інформацію про решту осіб:

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
class Program
{
    public struct one //структура для зберігання даних про одну людину
    {
        public string f;
        public string i;
        public string про;
        public int age;
        public float massa;
    }

    public static void Main()
    {
            StreamReader fileIn = new
            StreamReader("t.txt",Encoding.GetEncoding(1251));
            string line;
            Queue people = new Queue();
            one a;
```

```
                    Console.WriteLine("ВІК МЕНШЕ 40 РОКІВ");
    while ((line = fileIn.ReadLine()) != null) //читаємо до кінця файлу
                    {
                        string [] temp = line.Split(' ');
        //розбиваємо рядок на складові елементи
                        //заповнюємо структуру
                        a.f = temp[0];
                        a.i = temp ;
                        a.o = temp ;
                        a.age = int.Parse(temp );
                        a.massa = float.Parse(temp );
// якщо вік менше 40 років, то виводимо дані на екран, інакше поміщаємо //їх у
 чергу для тимчасового зберігання
            if (a.age<40)
Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t" + a.massa);
                        else people.Enqueue(a);
                    }
                    fileIn.Close();

                    Console.WriteLine("ВІК 40 РОКІВ І СТАРШЕ");
                    while (people.Count != 0) //витягуємо з черги дані  {
                        a = (one)people.Dequeue();
  Console.WriteLine(a.f + "\t"+ a.i + "\t"+ a.o + "\t"+a.age + "\t" + a.massa);
                    }
                }
            }
        }
```

_____t.txt_____

Іванов Сергій Миколайович 21 64
Петров Ігор Юрійович 45 88
Семенов Михайло Олексійович 20 70
Піманов Олександр Дмитрович 53 101

У прикладі наведено використання контейнеру Queue.

Клас ArrayList

У C# стандартні масиви мають фіксовану довжину, яка не може змінитися під час виконання програми. Клас ArrayList призначений для підтримки динамічних масивів, які за необхідності можуть збільшуватися або скорочуватися.

Об'єктом класу ArrayList є масив змінної довжини, елементами якого є об'єктні посилання. Будь-який об'єкт класу ArrayList створюється

з деяким початковим розміром. З перевищенням цього розміру колекція автоматично подвоюється. У разі видалення об'єктів масив можна скоротити.

Клас ArrayList реалізує інтерфейси ICollection, IList, IEnumerable і ICloneable. У класі ArrayList визначені такі *конструктори*:

```
//створює порожній масив з максимальною ємкістю рівної 16
елементам, //при поточній розмірності 0
public ArrayList()
public ArrayList(int capacity) // створює масив із заданою ємкістю
capacity, при поточній розмірності 0
public ArrayList(ICollection c)        // створює масив, який ініціалізувався
елементами колекції c
```

Окрім методів, визначених у інтерфейсах, які реалізує клас ArrayList, у ньому визначені власні методи, наведені в табл. 9.10.

Таблиця 9.10

Методи класу ArrayList

Методи	Опис
1	2
public virtual void AddRange (ICollection c)	Додає елементи з колекції c у кінець колекції, що викликається
public virtual int BinarySearch (object v)	У відсортованій колекції, що викликається, виконує пошук значення, заданого параметром v. Повертає індекс знайденого елемента. Якщо шукане значення не виявлене, повертає від'ємне значення
public virtual int BinarySearch (object v, IComparer comp)	У відсортованій колекції, що викликається, виконує пошук значення, заданого параметром v, на основі методу порівняння об'єктів, заданого параметром comp. Повертає індекс знайденого елемента. Якщо шукане значення не виявлене, повертає від'ємне значення
public virtual int BinarySearch (int startIdx nt count, object v, IComparer comp)	У відсортованій колекції, що викликається, виконує пошук значення, заданого параметром v, на основі методу порівняння об'єктів, заданого параметром comp. Пошук починається з елемента, індекс якого дорівнює значенню startIdx, і включає count елементів. Метод повертає індекс знайденого елемента. Якщо шукане значення не виявлене, повертає від'ємне значення
public virtual void CopyTo (Array ar, int startIdx)	Копіює вміст колекції, що викликається, починаючи з елемента, індекс якого дорівнює значенню startIdx, у масив, заданий параметром ar. Приймальний масив має бути одновимірним і сумісним за типом з елементами колекції

Продовження табл. 9.10

1	2
public virtual void CopyTo (int srcIdx, Array ar int destIdx, int count)	Копіює count елементів колекції, що викликається, починаючи з елемента, індекс якого дорівнює значенню srcIdx, у масив, заданий параметром ar, починаючи з елемента, індекс якого дорівнює значенню destIdx. Приймальний масив має бути одновимірним і сумісним за типом з елементами колекції
public virtual ArrayList GetRange(int idx int count)	Повертає частину колекції, що викликається, типу ArrayList. Діапазон повертаної колекції починається з індексу idx і включає count елементів. Повертаний об'єкт посилається на ті ж елементи, що і викликаний об'єкт
public static ArrayList FixedSize(ArrayList ar)	Перетворює колекцію ar на ArrayList-массив з фіксованим розміром і повертає результат
public virtual void InsertRange (int startIdx ICollection c)	Вставляє елементи колекції, заданої параметром з, у колекцію, що викликається, починаючи с індексу, заданого параметром startIdx
public virtual int LastIndexOf(object v)	Повертає індекс останнього входження об'єкта v у колекцію, що викликається. Якщо шуканий об'єкт не виявлений, повертає негативне значення
public static ArrayList Readonly(ArrayList ar)	Перетворює колекцію ar на ArrayList-массив, призначений тільки для читання
public virtual void ReraoveRange(int idx int count)	Видаляє count елементів із колекції, що викликається, починаючи з елемента, індекс якого дорівнює значенню idx
public virtual void Reverse()	Розташовує елементи колекції, що викликається, у зворотному порядку
public virtual void Reverse(int startIdx, int count)	Розташовує в зворотному порядку count елементів колекції, що викликається, починаючи з індексу startIdx
public virtual void SetRange(int startIdx ICollection c	Замінює елементи колекції, що викликається, починаючи з індексу startIdx, елементами колекції, заданої параметром с
public virtual void Sort()	Сортує колекцію за збільшенням
public virtual void Sort(IComparer comp)	Сортує колекцію, що викликається, на основі методу порівняння об'єктів, заданого параметром comp. Якщо параметр comp має нульове значення, для кожного об'єкта використовується стандартний метод порівняння
Public virtual void Sort (int startidx, int endidx icomparer comp)	Сортує частину колекції, що викликається, на основі методу порівняння об'єктів, заданого параметром comp. Сортування починається з індексу startidx і закінчується індексом endidx. Якщо параметр comp має нульове значення, для кожного об'єкта використовується стандартний метод порівняння

Закінчення табл. 9.10

1	2
public virtual object [ ] ToArray ()	Повертає масив, який містить копії елементів об'єкта, що викликається
public virtual Array ToArray (Type type)	Повертає масив, який містить копії елементів об'єкта, що викликається. Тип елементів у цьому масиві задається параметром type
public virtual void TrimToSize()	Установлює властивість Capacity дорівненою значенню властивості Count

Властивість Capacity дозволяє дізнатися або встановити ємкість динамічного масиву, що викликається, типу ArrayList. Ємкістю є кількість елементів, які можна зберегти в ArrayList-массиві без його збільшення. Якщо заздалегідь відомо, скільки елементів має міститися в ArrayList-масиві, то його розмірність можна встановити з використанням властивості Capacity, економлячи тим самим системні ресурси. Якщо потрібно зменшити розмір ArrayList-массива, то шляхом установки властивості Capacity його можна зменшити. Але встановлюване значення не має бути меншим значення властивості Count, інакше згенерується виключення ArgumentOutOfRangeException. Щоб зробити ємкість ArrayList-масиву рівною дійсній кількості елементів, що зберігаються в ньому в поточний час, треба встановити значення властивості Capacity дорівненим властивості Count. Того ж ефекту можна досягти, викликавши метод TrimToSize ().

Проаналізуємо декілька прикладів використання динамічного масиву.

Приклад 15. Розглянемо прості операції з динамічним масивом.

```
using System;
using System.Collections;

namespace MyProgram
{
  class Program
  {
            static void ArrayPrint(string s, ArrayList a)
            {
              Console.WriteLine(s);
              foreach (int i in a)
                    Console.Write(i + " ");
              Console.WriteLine();
            }

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

```
        {
          ArrayList myArray = new ArrayList();
          Console.WriteLine("Початкова ємкість масиву: " +
        myArray.Capacity);
Console.WriteLine("Початкова кількість елементів: " + myArray.Count);

          Console.WriteLine("\nДобавили 5 цифр");
          for (int i = 0; i < 5; i++) myArray.Add(i);
Console.WriteLine("Поточна ємкість масиву: " + myArra.Capacity);
Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
          ArrayPrint("Вміст масиву", myArray);

          Console.WriteLine("\nОптимізуємо ємкість масиву");
          myArray.Capacity=myArray.Count;
Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
          ArrayPrint("Вміст масиву", myArray);

          Console.WriteLine("\nДобавляємо елементи в масив");
          myArray.Add(10);
          myArray.Insert(1, 0);
          myArray.AddRange(myArray);
Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
          ArrayPrint("Вміст масиву", myArray);

          Console.WriteLine("\nВилучаємо елементи з масиву");
          myArray.Remove(0);
          myArray.RemoveAt(10);
Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
          ArrayPrint ("Вміст масиву", myArray);

          Console.WriteLine("\nУдаляем весь масив");
          myArray.Clear();
Console.WriteLine("Поточна ємкість масиву: " + myArray.Capacity);
Console.WriteLine("Поточна кількість елементів: " + myArray.Count);
          ArrayPrint("Вміст масиву", myArray);
          }
  }
}
```

Приклад 16. У текстовому файлі записана інформація про людей (прізвище, ім'я, по батькові, вік, вага через пропуск). Необхідно вивести на екран інформацію про людей, відсортовану за віком.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
  class Program
  {
  public struct one //структура для зберігання даних про одну людину
  {
        public string f;
        public string i;
        public string про;
        public int age;
        public float massa;
  }

  public class SortByAge : IComparer //реализация стандартного //інтерфейсу
  {
        int IComparer.Compare(object x, object y) //перевизначення методу
//Compare
        {
              one t1 = (one)x;
              one t2 = (one)y;
              if (t1.age > t2.age) return 1;
              if (t1.age < t2.age) return -1;
              return 0;
        }
  }

  static void ArrayPrint(string s, ArrayList a)
  {
        Console.WriteLine(s);
        foreach (one x in a)
              Console.WriteLine(x.f + "\t"+ x.i + "\t"+ x.o + "\t"+x.age + "\t" + x.massa);
  }

  static void Main(string[] args)
  {
              StreamReader fileIn = new
StreamReader("t.txt",Encoding.GetEncoding(1251));
              string line;
              one a;
              ArrayList people = new ArrayList();
              string[] temp = new string;
```

```
                while ((line=fileIn.ReadLine())!=null) //цикл для організації
//обробки файлу
                {
                        temp = line.Split(' ');
                        a.f = temp[0];
                        a.i = temp;
                        a.o = temp;
                        a.age = int.Parse(temp );
                        a.massa = float.Parse(temp );
                        people.Add(a);
                }
                fileIn.Close();

                ArrayPrint("Початкові дані: ", people);
                people.Sort(new Program.SortByAge());    //виклик сортування
                ArrayPrint("Відсортовані дані: ", people);
  }
}
}
```

_____t.txt_____

Іванов Сергій Миколайович 21 64
Петров Ігор Юрійович 45 88
Семенов Михайло Олексійович 20 70
Піманов Олександр Дмитрович 53 101

Зауваження. Зверніть увагу на те, що в даному прикладі був розроблений вкладений клас SortByAge, який реалізовує стандартний інтерфейс IComparer. У цьому класі був переобтяжений метод Compare, що дозволяє порівнювати між собою два об'єкти типу one. Створений клас використовувався для сортування колекції за заданим критерієм (за віком).

Клас Hashtable

Клас Hashtable призначений для створення колекції, в якій для зберігання об'єктів використовується геш-кодування-таблиця. У геш-таблиці для зберігання інформації застосовують механізм **гешування** (hashing). Сутність гешування полягає в тому, що для визначення унікального значення, яке називають геш-кодом, використовується інформаційний вміст відповідного йому ключа. Геш-кодування-код потім використовується як індекс, за яким у таблиці відшукуються дані, відповідні цьому ключу. Перетворення ключа в геш-кодування-код виконується автоматично, тобто саме геш-кодування-код ви навіть не побачите. Але перевага гешування

в тому, що воно дозволяє скорочувати час виконання таких операцій, як пошук, прочитування і запис даних, навіть для великих обсягів інформації.

У класі Hashtable, окрім властивостей, визначених у реалізованих ним інтерфейсах, визначено дві власні public-властивості:

```
public virtual ICollection Keys { get; } //дозволяє отримати колекцію ключів
public virtual ICollection Values { get; } //дозволяє отримати колекцію значень
```

Для додавання елементу в геш-таблицю необхідно викликати метод Add(), який приймає два окремі аргументи: ключ і значення. Важливо зазначити, що геш-таблиця не гарантує збереження порядку елементів, оскільки гешування зазвичай не застосовується до відсортованих таблиць.

Клас Hashtable реалізує стандартні інтерфейси IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback і ICloneable. Розмір геш-таблиці може динамічно змінюватися. Він збільшується тоді, коли кількість елементів перевищує значення, рівне твору місткості таблиці та її коефіцієнта заповнення, який може набувати значення на інтервалі від 0,1 до 1,0. За замовчуванням установлений коефіцієнт рівний 1,0.

У класі Hashtable визначено кілька конструкторів:

```
public Hashtable() // створює порожню геш-кодування-таблицю
public Hashtable(IDictionary c)// будує геш-таблицю, яка ініціалізувалася
//елементами колекції c
public Hashtable(int capacity)  // створює геш-таблицю з місткістю capacity
public Hashtable(int capacity, float n) //створює геш-таблицю місткістю
//capacity і коефіцієнтом заповнення n
```

Окрім методів, визначених в інтерфейсах, які реалізує клас Hashtable, в ньому визначені власні методи, які наведені в табл. 9.11.

Таблиця 9.11

Методи класу Hashtable

Методи	Опис
public virtual bool ContainsKey (object k)	Повертає значення true, якщо в геш-таблиці, що викликається, міститься ключ, заданий параметром k. Інакше – повертає значення false
public virtual bool ContainsValue (object v)	Повертає значення true, якщо в геш-таблиці, що викликається, міститься значення, задане параметром v. Інакше – повертає значення false
public virtual IDictionaryEnumerator GetEnumerator()	Повертає для геш-таблиці, що викликається, нумератор типу IDictionaryEnumerator

Наступний приклад демонструє використання Hashtable колекції.
Приклад 17. Розглянемо прості операції з геш-таблицею.

```
using System;
using System.Collections;

namespace MyProgram
{
class Program
{

        static void printTab(string s, Hashtable a)
        {
                Console.WriteLine(s);
                ICollection key = a.Keys; //Прочитали всі ключі
                foreach (string i in key) //використання ключа для набуття //значення
                {
                        Console.WriteLine(i+"\t"+a[i]);
                }
                Console.WriteLine();
        }
static void Main(string[] args)
        {
                Hashtable tab = new Hashtable();
                Console.WriteLine("Початкова кількість елементів: " + tab.Count);
                printTab("Вміст таблиці: ", tab);

                Console.WriteLine("Додали в таблицю записи");
                tab.Add("001","ПЕРШИЙ");
                tab.Add("002","ДРУГИЙ");
                tab.Add("003","ТРЕТІЙ");
                tab.Add("004", "ЧЕТВЕРТИЙ");
                tab.Add("005", "П'ЯТИЙ");
                Console.WriteLine("Поточна кількість елементів: " + tab.Count);
                printTab("Вміст заповненої таблиці", tab);
                tab["005"]= "НОВИЙ П'ЯТИЙ";
                tab["001"]= "НОВИЙ ПЕРШИЙ";
                printTab("Вміст зміненої таблиці", tab);
        }
}
}
```

Приклад 18. Розробимо простий записник, в який можна додавати та видаляти номера телефонів, а також здійснювати пошук номера телефону за прізвищем і прізвища – за номером телефону.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace MyProgram
{
class Program
{
static void printTab(string s, Hashtable a)
  {
        Console.WriteLine(s);
        ICollection key = a.Keys; //Прочитали всі ключі
        foreach (string i in key) //використання ключа для набуття значення
        {
                Console.WriteLine(i + "\t" + a[i]);
        }
  }

  static void Main(string[] args)
  {
                StreamReader fileIn = new
StreamReader("t.txt",Encoding.GetEncoding(1251));
                string line;
                Hashtable people = new Hashtable();
                while ((line = fileIn.ReadLine()) != null) //цикл для організації
//обробки файлу
                {
                        string [] temp = line.Split(' ');
                        people.Add(temp[0],temp );
                }
                fileIn.Close();
                printTab("Початкові дані: ", people);

                Console.WriteLine("Введіть номер телефону");
                line = Console.ReadLine();
                if (people.ContainsKey(line)) Console.WriteLine(line + "\t" + people[line]);
                else
                {
                        Console.WriteLine("Такого номера немає в
записнику.\nВведіть прізвище: ");
                        string line2=Console.ReadLine();
                        people.Add(line,line2);
                }
```

```
                printTab("Початкові дані: ", people);
                Console.WriteLine("Введіть прізвище для видалення");
                line = Console.ReadLine();
                if (people.ContainsValue(line))
                {
                        ICollection key =people.Keys; //Прочитали всі ключі
                        Console.WriteLine(line);
                        string del="";
                        foreach (string i in key) //використання ключа для набуття
значення
                                if (string.Compare((string)people[i], line) == 0)
                                {
                                        del = i;
                                        break;
                                }

                        Console.WriteLine(del + "\t" + people[del]+ "- дані видалені!!!");
                        people.Remove(del);
                        printTab("Змінені дані: ", people);
                }
                else Console.WriteLine("Такого абонента в записнику
        немає ");
                }
        }
}
```

_____t.txt_____

12-34-56 Іванов
78-90-12 Петров
34-56-78 Семенов
90-11-12 Піманов

Простір імен System.Collections.Generic містить інтерфейси та класи, що визначають універсальні колекції, які дозволяють користувачам створювати строго типізовані колекції, що забезпечують підвищену продуктивність і безпеку типів. Класи та інтерфейси, які входять до цього простору імен, наведені в табл. 9.12, 9.13.

Багато типізованих колекцій є прямими аналогами нетипізованих. Наприклад, колекція Dictionary <TKey, TValue> - це універсальна версія Hashtable, яка використовує структуру KeyValuePair <TKey, TValue> замість DictionaryEntry. List <T> - це типізована версія ArrayList. Є класи Queue <T> і Stack <T>, які відповідають нетипізованим версіям. Розглянемо декілька типізованих колекцій і почнемо з класу List <T>.

Таблиця 9.12

Класи

Класи	Опис
Comparer <T>	Представляє базовий клас для реалізацій універсального інтерфейсу IComparer <T>
Dictionary <TKey, TValue> .KeyCollection	Представляє колекцію ключів у Dictionary <TKey, TValue>. Цей клас не успадковується
Dictionary <TKey, TValue> .ValueCollection	Представляє колекцію значень у Dictionary <TKey, TValue>. Цей клас не успадковується
Dictionary <TKey, TValue>	Представляє колекцію ключів і значень
HashSet <T>	Представляє набір значень
KeyedByTypeCollection <TItem>	Надає колекцію, елементами якої є типи, які використовуються в якості ключів
LinkedList <T>	Представляє двоспрямований список
LinkedListNode <T>	Представляє вузол у LinkedList <T>. Цей клас не успадковується
List <T>	Представляє строго типізований список об'єктів, доступних за індексом. Підтримує методи для пошуку за списком, виконання сортування та інші операції зі списками
Queue <T>	Представляє колекцію об'єктів, засновану на принципі "першим надійшов - першим обслужений"
SortedDictionary <TKey, TValue> .KeyCollection	Представляє колекцію ключів у SortedDictionary <TKey, TValue>. Цей клас не успадковується
SortedDictionary <TKey, TValue> .ValueCollection	Представляє колекцію значень у SortedDictionary <TKey, TValue>. Цей клас не успадковується
SortedDictionary <TKey, TValue>	Представляє колекцію пар "ключ - значення", впорядкованих за ключем
SortedList <TKey, TValue>	Представляє колекцію пар "ключ - значення", упорядкованих за ключем на основі реалізації IComparer <T>
SortedSet <T>	Представляє впорядковану колекцію об'єктів
Stack <T>	Представляє колекцію змінного розміру примірників однакового заданого типу, яка обслуговується за принципом "останнім прийшов - першим вийшов" (LIFO)

Таблиця 9.13

Інтерфейси

Інтерфейси	Опис
ICollection <T>	Визначає методи для управління універсальними колекціями
IComparer <T>	Визначає метод, який реалізується типом для порівняння двох об'єктів
IDictionary <TKey, TValue>	Визначає універсальну колекцію пар "ключ – значення"
IEnumerable <T>	Надає нумератор, який підтримує простий перебір елементів у зазначеній колекції
IEnumerator <T>	Підтримує простий перебір елементів універсальної колекції
IEqualityComparer <T>	Визначає методи, що підтримують порівняння об'єктів на предмет рівності
IList <T>	Представляє колекцію об'єктів, доступ до яких можна отримати індивідуально за індексом
IReadOnlyCollection <T>	Повертає строго типізовану колекцію елементів, доступну тільки для читання
IReadOnlyDictionary <TKey, TValue>	Визначає універсальну колекцію пар "ключ – значення", доступну тільки для читання
IReadOnlyList <T>	Визначає доступну тільки для читання колекцію елементів, доступ до яких можна отримати за індексом
ISet <T>	Надає основний інтерфейс для абстракції наборів

Клас List <T> представляє найпростіший список однотипних об'єктів. Серед його методів можна виділити такі:

void Add (T item) – додавання нового елемента в список;

void AddRange (ICollection collection) – додавання в список колекції або масиву;

int BinarySearch (T item) – бінарний пошук елемента в списку. Якщо елемент знайдений, то метод повертає індекс цього елемента в колекції. Водночас список повинен бути відсортований;

int IndexOf (T item) – повертає індекс першого входження елемента в список;

void Insert (int index, T item) – вставляє елемент item у списку на позицію index;

bool Remove (T item) – видаляє елемент item зі списку, і якщо видалення пройшло успішно, то повертає true;

void RemoveAt (int index) – видалення елемента за вказаною індексу index;

void Sort () – сортування списку.

Приклад 19. Наведемо приклад використання колекції List, типізованої користувальницьким типом Student:

```
class Student
  {
    string fname;
    string lname;
    public Student(string fname, string lname)
    {
      this.fname = fname;
      this.lname = lname;
    }
    public override string ToString()
    {
      return fname+" "+lname;
    }
  }

  class Program
  {
    static void Main(string[] args)
    {
      List<Student> group = new List<Student>();
      group.Add(new Student("Ярослав", "Коваленко"));
      group.Add(new Student("Олена", "Петренко"));
      group.Add(new Student("Олеся", "Степаненко"));

      Console.WriteLine("Список студентів групи:");
      foreach (Student s in group)
        Console.WriteLine(s);

      Console.WriteLine("Кількість студентів в групі = {0}", group.Count);

    }
  }
```

Ще один поширений тип колекції становлять словники, які реалізує клас Dictionary<T, V>. Словник зберігає об'єкти, які представляють пару

"ключ – значення". Кожен такий об'єкт належить до структури KeyValuePair <TKey, TValue>. Завдяки властивостям Key і Value, які є у цій структурі, можна отримати ключ і значення елемента в словнику. Основні методи типізованої колекції Dictionary<T, V> наведені в табл. 9.14.

Таблиця 9.14

Інтерфейси

Інтерфейси	Опис
Add (TKey, TValue)	Додає зазначені ключ і значення в словник
Clear ()	Видаляє всі ключі та значення зі словника Dictionary <TKey, TValue>
ContainsKey (TKey)	Визначає, чи міститься вказаний ключ у словнику Dictionary <TKey, TValue>
ContainsValue (TValue)	Визначає, чи містить колекція Dictionary <TKey, TValue> указане значення
Equals (Object)	Визначає, чи дорівнює заданий об'єкт поточному об'єкту (Inherited from Object)
GetEnumerator ()	Повертає нумератор, який здійснює перебір елементів списку Dictionary <TKey, TValue>
GetObjectData (SerializationInfo, StreamingContext)	Реалізує інтерфейс ISerializable та повертає дані, необхідні для серіалізації примірника Dictionary <TKey, TValue>
OnDeserialization (Object)	Реалізує інтерфейс ISerializable та викликає подію де-серіалізації після завершення процесу десеріалізації
Remove (TKey)	Видаляє значення зі вказаним ключем з Dictionary <TKey, TValue>
ToString ()	Повертає рядок, що представляє поточний об'єкт
TryGetValue (TKey, TValue)	Повертає значення, пов'язане із заданим ключем

*Приклад 20. Я*к ключем, так і значенням може бути об'єкт будь-якого типу. Розглянемо приклад використання типізованої колекції Dictionary, де ключем буде тип string, а значенням – тип double.

```
class Program
  {
    static void Main(string[] args)
    {
      Dictionary<string, double> dict = new Dictionary<string, double>();
      dict.Add("Чай", 18.0);
      dict.Add("Хліб", 17.1);
```

```
            dict.Add("Молоко", 28.58);

            if (dict.ContainsKey("Чай"))
            {
                double val = dict["Чай"];
                Console.WriteLine("Ціна чаю: {0:c}.", val);

                dict.Remove("Чай");
            }

            foreach (KeyValuePair<string, double> entry in dict)
            {
                string k = entry.Key;
                double v = entry.Value;
                Console.WriteLine("{0} price: {1:c}", k, v);
            }

            foreach (string k in dict.Keys)
            {
                Console.WriteLine("Key: {0}", k);
            }
        }
    }
```

У прикладі наведено використання колекції Dictionary.

Колекції у Java

Java-колекції також, як і в C#, можуть бути нетипізованими або типізованими.

Однак один і той же клас може представляти як нетипізовану так і типізовану колекцію. Якщо зі створенням колекції вказується тип даних, що будуть в ній зберігатися, – це **типізована** колекція. Інакше – **нетипізована.**

Елементи бібліотеки колекцій містяться в пакеті java.util. ЇЇ головні компоненти – це інтерфейси та класи.

Інтерфейси визначають поведінку, притаманну всім класам колекцій, які реалізують ці інтерфейси.

Класи бібліотеки колекцій головним чином є конкретними реалізаціями таких структур даних, як динамічний масив, список, словник тощо.

Ієрархія інтерфейсів бібліотеки колекцій наведена на рис. 9.3.

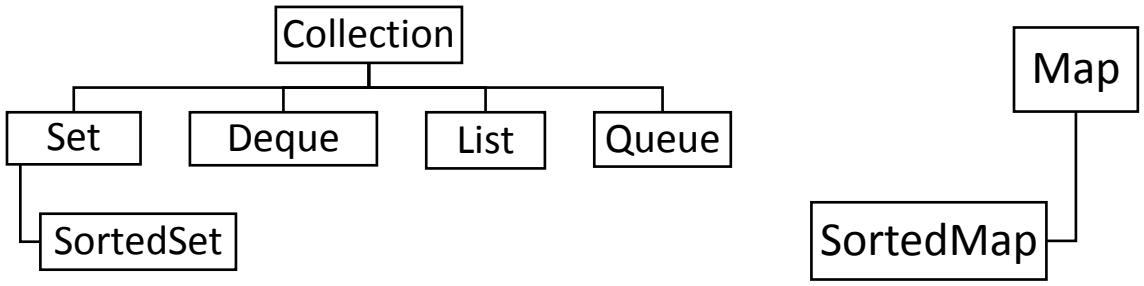

Рис. 9.3. **Ієрархія інтерфейсів бібліотеки колекцій**

Стисла характеристика інтерфейсів бібліотеки контейнерів Java SE є такою:

Collection – "кореневий" інтерфейс;

Set – "множина" (однакові елементи неприпустимі);

SortedSet – множина, елементи якої впорядковані за зростанням;

List – "список";

Queue, Dequeue – "черги";

Map – "словник" або "асоціативний масив";

SortedMap – "словник", ключі якого впорядковані за зростанням.

Основні класи реалізації наведено в табл. 9.15.

Таблиця 9.15

Основні класи реалізації інтерфейсів бібліотеки контейнерів Java SE

Ім'я класу	Опис
ArrayList	Динамічний масив, що реалізує інтерфейс List
ArrayDeque	Динамічний масив, що реалізує інтерфейси Queue та Dequeue. Клас можна використовувати в режимі черги або стеку
LinkedList	Двозв'язний список
HashSet	Множина на базі геш-таблиці
TreeSet	Множина на базі червоно-чорного дерева
HashMap	Словник на базі геш-таблиці
TreeMap	Словник на базі червоно-чорного дерева

У класах **ArrayList** та **ArrayDeque** для зберігання елементів використовується **масив**. Це забезпечує швидкі операції проходу за всім списком (ітерації) і отримання даних.

Приклад 21. Наведемо приклад використання класу **ArrayList**:

```
class Person {
        private String name;
        private int age;

        public Person(String name, int age) {
                this.name = name;
                this.age = age;
        }

        public String toString() {
                return (name + ", " + age);
        }
}
……..
ArrayList <Person> list = new ArrayList <Person>();
list.add(new Person("Іван", 99));
list.add(new Person("Мар'я", 24));
for (int i = 0; i < list.size(); i++) {
   System.out.println(list.get(i));
}
```

У цьому прикладі об'єкт класу ArrayList – це типізований динамічний масив для зберігання об'єктів класу Person.

Приклад 22. Розглянемо приклад використання класу ArrayDeque.

```
import java.util.*;

public class MainClass {

        public static void main(String[] args) {
                Person a1 = new Person("Олексій", 33);
                Person a2 = new Person("Вікторія", 19);
                Person a3 = new Person("Сергій", 3);

                // Режим черги
                Queue<Person> queue = new ArrayDeque<Person>();
                queue.add(a1);
                queue.add(a2);
                queue.add(a3);
                System.out.println("Черга:");
                while (!queue.isEmpty())
                        System.out.println(queue.remove());

                // Режим стека
                Deque<Person> stack = new ArrayDeque<Person>();
```

```
            stack.push(a1);
            stack.push(a2);
            stack.push(a3);
            System.out.println("\nСтек:");
            while (!stack.isEmpty())
                System.out.println(stack.pop());
        }
    }
```

Якщо створити об'єкт класу ArrayDeque та присвоїти його посиланню типу Queue, він буде виконувати роль черги. Із заміною типу посилання на Deque цей об'єкт буде виконувати роль стеку.

У класі **LinkedList** для зберігання елементів використовується двозв'язний список. Він забезпечує швидкі операції додавання елементів у початок (кінець) списку, проходу за всім списком (ітерації), видалення елементів із середини списку (постійний час). Цей клас також дозволяє створювати черги з дисципліною обслуговування First Input First Output (FIFO) та стеки (дисципліна обслуговування LIFO).

У класі **HashSet** для зберігання елементів використовується геш-таблиця. Це забезпечує постійний час виконання основних операцій з елементами множини.

У класі **TreeSet** для зберігання елементів використовується бінарне дерево. Дні впорядковані, максимальне число кроків для пошуку на дереві дорівнює висоті цього дерева.

Щоб зберігати об'єкти власного класу (наприклад, класу Person) у колекціях на базі бінарного дерева, необхідно спочатку визначити, як порівнювати ці об'єкти під час додавання в певний узел дерева. У цьому випадку клас Person може бути таким:

```
class Person implements Comparable<Person> {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        return (name + ", " + age);
    }
```

```
    public int compareTo(Person p) {
        if (p.age > this.age)
            return -1;
        else if (p.age < this.age)
            return 1;
        else
            return 0;
    }
}
```

Клас Person має реалізувати інтерфейс java.lang Comparable та перевизначити його метод compareTo, тобто визначити алгоритм порівняння об'єктів цього класу. Цей метод повинен повертати нуль, один чи мінус один залежно від результатів порівняння. У цьому коді порівняння об'єктів класу Person відбувається на базі значення поля age.

Приклад 23. Наведемо приклад використання класу **TreeSet**, що зберігає об'єкти класу Person:

```
import java.util.TreeSet;

public class MainClass {

    public static void main(String[] args) {
        TreeSet<Person> s = new TreeSet<Person>();
        s.add(new Person("Ivan", 33));
        s.add(new Person("Galya", 20));
        s.add(new Person("Shaun the sheep", 77));
        for (Person p : s)
            System.out.println(p);
    }
}
```

Такі колекції, як словники (асоціативні масиви), містять пари "ключ – значення". І ключ, і значення є об'єктами. Ключі повинні бути унікальними.

У Java є два основних класа словників: **HashMap і TreeMap**. Їхні внутрішні структури даних аналогічні відповідним реалізаціям множин.

Приклад 24. Розглянемо на прикладі об'єкт класу HashMap:

```
HashMap <String, Double> hm = new HashMap <String, Double>();
hm.put("Тугарин Змій", 3434.34);
hm.put("Альоша Попович", 123.22);
hm.put("Добриня Никитич", 1378.00);
```

```
for (String key : hm.keySet()) {
  System.out.println(key + " : " + hm.get(key));
}
```

У цьому прикладі об'єкт класу HashMap є словником, ключами якого є рядки об'єктного типу String, а значеннями – об'єкти класу Double. Double – це об'єктна "оболонка" для примітивного типу double.

Пакет java.util також містіть корисні класи, що не є колекціями. Одним з них є клас **Collections**, який містіть так звані "алгоритми колекцій" – статичні методи, які реалізують деякі розповсюджені алгоритми обробки даних. Це такі методи, як sort (сортування вмісту колекції), shuffle (перемішування вмісту колекції), min (пошук мінімального елемента колекції), max (пошук максимального елемента колекції) та інші.

9.3. Джерела та споживачі даних. Загальні відомості про потоки введення – виведення даних. Алгоритми роботи потоків введення – виведення даних. Основні класи стандартних бібліотек Microsoft .NET і Java SE для підтримки введення – виведення даних. Серіалізація

Програми мовами C# і Java виконують операції введення – виведення за допомогою потоків, які побудовані на ієрархії класів. **Потік** (stream) – це абстракція, яка генерує і приймає дані. За допомогою потоку можна читати дані з різних джерел (клавіатура, файл) і записувати в різні джерела (принтер, екран, файл). Попри те, що потоки зв'язуються з різними фізичними пристроями, характер поведінки всіх потоків однаковий. Тому класи та методи введення – виведення можна застосувати до багатьох типам пристроїв.

На найнижчому рівні ієрархії потоків введення – виведення знаходяться потоки, які оперують байтами. Це пояснюється тим, що багато пристроїв у виконанні операцій введення – виведення орієнтовані на байти. Проте для людини більш звично оперувати символами. Тому розроблені символьні потоки, які фактично є оболонками, що виконують перетворення байтових потоків у символьні, та навпаки. Окрім цього, реалізовані потоки для роботи з int-, double-, short- значеннями, які також є оболонкою для байтових потоків. Вони працюють не з самими значеннями, а з їх внутрішнім поданням у вигляді двійкових кодів.

Потоки працюють з джерелами та споживачами даних, роль яких можуть виконувати консоль, файл, оперативна пам'ять і мережеве з'єднання.

Центральну частину потокової C#-системи займає клас Stream простору імен System.IO. **Клас Stream** – це байтовий потік, який є базовим для решти всіх потокових класів. З класу Stream виведені такі байтові класи потоків, як:

FileStream – байтовий потік, розроблений для файлового введення – виведення;

BufferedStream – укладає в оболонку байтовий потік і додає буферизацію, яка у багатьох випадках збільшує продуктивність програми;

MemoryStream – байтовий потік, який використовує пам'ять для зберігання даних.

Програміст може вивести власні потокові класи. Проте для переважної більшості застосунків достатньо вбудованих потоків.

Детально розглянемо клас FileStream, класи StreamWriter і StreamReader, що є оболонками для класу FileStream і дозволяють перетворювати байтові потоки в символьні, а також класи BinaryWriter і BinaryReader, що є оболонками для класу FileStream і дозволяють перетворювати байтові потоки в двійкові для роботи з int-, double-, short- і так далі значеннями.

Байтовий потік

Щоб створити байтовий потік, пов'язаний з файлом, створюється об'єкт класу FileStream. Для цього в класі визначено декілька конструкторів. Найчастіше використовується конструктор, який відкриває потік для читання і/або запису:

```
FileStream(string filename, FileMode mode)
```

Розглядуваний конструктор має такі властивості:

1) параметр filename – визначає ім'я файла, з яким буде пов'язаний потік введення – виведення даних. Filename визначає або повний шлях до файла, або ім'я файла, який знаходиться в папці bin/debug вашого проекту;

2) параметр mode – визначає режим відкриття файла, який може приймати одне з можливих значень, визначених переліком FileMode:

а) FileMode.Append – призначений для додавання даних у кінець файла;

б) FileMode.Create – призначений для створення нового файла. Водночас файл з таким же ім'ям буде заздалегідь видалений;

в) FileMode.CreateNew – призначений для створення нового файла; файл з таким же ім'ям не повинен існувати;

г) FileMode.Open – призначений для відкриття існуючого файла;

д) FileMode.OpenOrCreate – якщо файл існує, то відкриває його; інакше – створює новий;

е) FileMode.Truncate – відкриває існуючий файл, але усікає його довжину до нуля.

Якщо спроба відкрити файл виявилася не успішною, то генерується одне з виключень:

FileNotFoundException – файл неможливо відкрити внаслідок його відсутності;

IOException – файл неможливо відкрити через помилку введення – виведення;

ArgumentNullException – ім'ям файла є null-значення;

ArgumentException – некоректний параметр mode;

SecurityException – користувач не володіє правами доступу;

DirectoryNotFoundException – некоректно заданий каталог.

Інша версія конструктора дозволяє обмежити доступ тільки читанням або тільки записом:

FileStream (string filename, FileMode mode, FileAccess how)

Цій версії конструктора притаманні такі параметри:

1) filename та mode – мають те ж призначення, що і в попередній версії конструктора;

2) how – визначає спосіб доступу до файла; може приймати одне із значень, визначених переліком FileAccess:

а) FileAccess.Read – тільки читання;

б) FileAccess.Write – тільки запис;

в) FileAccess.ReadWrite – і читання, і запис.

Після встановлення зв'язку байтового потоку з фізичним файлом внутрішній покажчик потоку встановлюється на початковий байт файла.

Для читання чергового байта з потоку, пов'язаного з фізичним файлом, використовується метод ReadByte (). Після прочитання чергового байта внутрішній покажчик переміщається на наступний байт файлу. Якщо досягнуто кінця файла, то метод ReadByte() повертає значення – 1.

Для побайтового запису даних у потік використовується метод WriteByte ().

Розглянемо приклад використання класу FileStream, для копіювання одного файла в інший. Але спочатку створимо текстовий файл text.txt

у папці bin/debug поточного проекту. Внесемо до нього довільну інформацію, наприклад:

12 456
Hello!
23,67 4: Message

```
using System;
using System.Text;
using System.IO;  //для роботи з потоками
namespace MyProgram {
    class Program {
        static void Main()
        {
            try
            {
            FileStream fileIn = new FileStream("text.txt",
            FileMode.Open, FileAccess.Read);
            FileStream fileOut = new FileStream("newText.txt",
            FileMode.Create, FileAccess.Write);
            int i;
            while ((i = fileIn.ReadByte())!=-1)
            {
        //запис чергового файла в потік, пов'язаний з файлом fIleOut
                fileOut.WriteByte((byte)i);
            }
            fileIn.Close();
            fileOut.Close();
            }
            catch (Exception EX)
            {
                Console.WriteLine(EX.Message);
            }
        }
    }
}
```

Після закінчення роботи з файлом його необхідно закрити. Для цього достатньо викликати метод Close (). Із закриттям файла звільняються системні ресурси, раніше виділені для цього файла, що дає можливість використовувати їх для роботи з іншими файлами.

Символьний потік

Щоб створити символьний потік потрібно помістити об'єкт класу Stream (наприклад, FileStream) "всередину" об'єкта класів StreamWriter або StreamReader. У цьому випадку байтовий потік автоматично перетворюватиметься в символьний.

Клас StreamWriter призначений для організації вихідного символьного потоку. У ньому визначено кілька конструкторів. Один із них записується таким чином:

StreamWriter(Stream stream);

Тут параметр stream визначає ім'я вже відкритого байтового потоку.

Наприклад, створити екземпляр класу StreamWriter можна таким чином:

StreamWriter fileOut=new StreamWriter (new FileStream ("text.txt", FileMode.Create, FileAccess.Write));

Цей конструктор генерує виняток типу ArgumentException, якщо потік stream не відкритий для виводу, і виключення типу ArgumentNullException, якщо потік має null-значення.

Інший вид конструктора дозволяє відкрити потік відразу через звернення до файлу:

StreamWriter(string name);

де параметр name визначає ім'я файла, що відкривається.

Наприклад, звернутися до цього конструктора можна таким чином:

StreamWriter fileOut=new StreamWriter("c:\temp\t.txt");

І ще один варіант конструктора StreamWriter:

StreamWriter(string name, bool appendFlag);

де параметр name визначає ім'я файлу, що відкривається;

параметр appendFlag може набувати значення true – якщо потрібно додавати дані в кінець файла, або false – якщо файл необхідно перезаписати таким, наприклад, чином:

StreamWriter fileOut=new StreamWriter ("t.txt", true);

Тепер для запису даних у потік fileOut можна звернутися до методу WriteLine. Це можна зробити таким чином:

```
fileOut.WriteLine("test");
```

У цьому випадку в кінець файлу t.txt буде дописано слово test.

Клас StreamReader призначений для організації вхідного символьного потоку. Один з його конструкторів виглядає таким чином:

StreamReader (Stream stream);

де параметр stream визначає ім'я вже відкритого байтового потоку.

Цей конструктор генерує виключення типу ArgumentException, якщо потік stream не відкритий для введення.

Наприклад, створити екземпляр класу StreamWriter можна таким чином:

StreamReader fileIn = new StreamReader (new FileStream ("text.txt", FileMode.Open, FileAccess.Read));

Як і у випадку з класом StreamWriter, StreamReader має інший вид конструктора, який дозволяє відкрити файл безпосередньо:

StreamReader (string name);

де параметр name визначає ім'я файла, що відкривається.

Звернутися до цього конструктора можна таким чином:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt");

У C# символи реалізуються кодуванням Unicode. Для того щоб можна було обробляти текстові файли, що містять символи кирилиці (створені, наприклад, у Блокноті), рекомендується викликати такий вид конструктора StreamReader:

StreamReader fileIn=new StreamReader ("c:\temp\t.txt", Encoding.GetEncoding(1251));

Параметр Encoding.GetEncoding(1251) говорить про те, що виконуватиметься перетворення з коду Windows-1251 (одна з модифікацій коду ASCII, що містить символи кирилиці) в Unicode. Encoding.GetEncoding(1251) реалізований в просторі імен System.Text.

Тепер для читання даних із потоку fileIn можна скористатися методом ReadLine. Треба знати, що якщо буде досягнутий кінець файла, то метод ReadLine поверне значення null.

Приклад 25. Розглянемо приклад, в якому дані з одного файла копіюються в інший, але вже з використанням класів StreamWriter і StreamReader.

```
static void Main ()
{
        StreamReader fileIn = new StreamReader ("text.txt",
        Encoding.GetEncoding(1251));
        StreamWriter fileOut=new StreamWriter ("newText.txt", false);
        string line;
        while ((line=fileIn.ReadLine())!=null) //поки потік не порожній
        {
                fileOut.WriteLine(line);
        }
        fileIn.Close();
        fileOut.Close();
}
```

Отже, такий спосіб копіювання одного файла в інший дасть той же результат, що і з використанням байтових потоків. Проте його робота буде менш ефективною, оскільки витрачатиметься додатковий час на перетворення байтів у символи.

```
static void Main()
{
        StreamReader fileIn = new StreamReader ("text.txt");
        StreamWriter fileOut=new StreamWriter ("newText.txt", false);
        string text=fileIn.ReadToEnd();
        Regex r= new Regex (@"[-+]?\d+");
        Match integer = r.Match(text);
        while (integer.Success)
        {
fileOut.WriteLine(integer);
                integer = integer.NextMatch();
        }
        fileIn.Close();
        fileOut.Close();
}
```

Але у символьних потоків є свої переваги. Наприклад, можна використовувати регулярні вирази для пошуку заданих фрагментів тексту у файлі.

Двійкові потоки

Двійкові файли зберігають дані в тому ж вигляді, в якому вони містяться в оперативній пам'яті, тобто у внутрішньому поданні. Двійкові файли не застосовують для перегляду людиною, вони використовуються тільки для програмної обробки.

Вихідний потік BinaryWriter підтримує довільний доступ, тобто є можливість виконувати запис у довільну позицію двійкового файла. Найбільш важливі методи потоку BinaryWriter наведені в табл. 9.16.

Таблиця 9.16

Основні методи потоку BinaryWriter

Члени класу	Опис
BaseStream	Визначає базовий потік, з яким працює об'єкт BinaryWriter
Close	Закриває потік
Flush	Очищає буфер
Seek	Установлює позицію в поточному потоці
Write	Записує значення в поточний потік

Найбільш важливі методи вихідного потоку BinaryReader наведені в табл. 9.17.

Таблиця 9.17

Основні методи потоку BinaryReader

Члени класу	Опис
BaseStream	Визначає базовий потік, з яким працює об'єкт BinaryReader
Close	Закриває потік
PeekChar	Повертає наступний символ потоку без переміщення внутрішнього покажчика в потоці
Read	Прочитує черговий потік байтів або символів і зберігає в масиві, передаваному у вхідному параметрі
ReadBoolean, ReadByte, ReadInt32 і т. д	Прочитує з потоку дані певного типу

Двійковий потік відкривається на основі базової протоки (наприклад, FileStream). Водночас двійковий потік перетворюватиме байтовий потік у значення int-, double-, short- і т. д.

Приклад 26. Розглянемо приклад формування двійкового файла:

```
static void Main ()
{
        //відкриваємо двійковий потік
        BinaryWriter fOut=new BinaryWriter (new FileStream
        ("t.dat",FileMode.Create));
        //записуємо дані в двійковий потік
        for (int i=0; i<=100; i+=2)
        {
            fOut.Write(i);
        }
        fOut.Close(); //закриваємо двійковий потік
}
```

Спроба проглянути двійковий файл за допомогою текстового редактора є неінформативною. Двійковий файл можна переглянути програмним шляхом, наприклад, таким чином:

```
static void Main ()
 {
     FileStream f=new FileStream ("t.dat",FileMode.Open);
        BinaryReader fIn=new BinaryReader (f);
long n=f.Length/4; //визначаємо кількість чисел у двійковому //потоці
        int a;
        for (int i=0; i<n; i++)
        {
            a=fIn.ReadInt32();
            Console.Write(a+" ");
        }
        fIn.Close();
        f.Close();
 }
```

Двійкові файли є файлами з довільним доступом; нумерація елементів в двійковому файлі ведеться з нуля. Довільний доступ забезпечує метод Seek. Розглянемо його синтаксис:

Seek (long newPos, SeekOrigin pos)

Параметр newPos визначає нову позицію внутрішнього покажчика файлу в байтах щодо вихідної позиції покажчика, яка визначається параметром pos. У свою чергу, параметр pos має бути заданий одним із значень перерахування SeekOrigin, які наведені в табл. 9.18.

Таблиця 9.18

Значення перерахування SeekOrigin

Значення	Опис
SeekOrigin.Begin	Пошук від початку файлу
SeekOrigin.Current	Пошук від поточної позиції покажчика
SeekOrigin.End	Пошук від кінця файлу

Після виклику методу Seek подальші операції читання або запису виконуватимуться з нової позиції внутрішнього покажчика файла.

Приклад 27. Розглянемо приклад організації довільного доступу до двійкового файла (на прикладі файла t.dat):

```
static void Main()
{
        //зміна даних у двійковому потоці
        FileStream f=new FileStream ("t.dat",FileMode.Open);
        BinaryWriter fOut=new BinaryWriter (f);
        //визначаємо кількість байт у байтовому потоці
        long n=f.Length;
        int a;
        //сдвиг на дві позиції, оскільки тип int займає 4 байти
        for (int i=0; i<n; i+=8)
        {
                fOut.Seek(i,SeekOrigin.Begin);
                fOut.Write(0);
        }
        fOut.Close();
        //читання даних з двійкового потоку
        f=new FileStream ("t.dat",FileMode.Open);
        BinaryReader fIn=new BinaryReader (f);
        // визначаємо кількість чисел у двійковому потоці
        n=f.Length/4;
        for (int i=0; i<n; i++)
        {
                a=fIn.ReadInt32();
                Console.Write(a+" ");
        }
        fIn.Close();
        f.Close();
}
```

Потік BinaryReader не має методу Seek. Проте, використовуючи можливості потоку FileStream, можна організувати довільний доступ читанні двійкових файлів. Розглянемо наступний приклад:

```
static void Main () {
   //Записуємо у файл t.dat цілі числа від 0 до 100
   FileStream f=new FileStream ("t.dat",FileMode.Open);
   BinaryWriter fOut=new BinaryWriter (f);
   for (int i=0; i<100; ++i) {
      fOut.Write(i);;
   }
   fOut.Close();
   //Об'єкти f і fIn пов'язані з одним і тим же файлом
   f=new FileStream ("t.dat",FileMode.Open);
   BinaryReader fIn=new BinaryReader (f);
   long n=f.Length; // визначаємо кількість байт потоці
   //Читаємо дані з файла t.dat, переміщаючи внутрішній покажчик на 8
//байт, тобто на два цілі числа
   for (int i=0; i<n; i+=8)
   {
      f.Seek(i,SeekOrigin.Begin);
      int a=fIn.ReadInt32();
      Console.Write(a+" ");
   }
   fIn.Close();
   f.Close();
}
```

У прикладі наведено використання методу класу FileStream.

Переспрямування стандартних потоків

Трьома стандартними потоками, доступ до яких здійснюється через властивості Console.Out, Console.In і Console.Error, можуть користуватися всі програми, що працюють у просторі імен System. Властивість Console.Out належить до стандартного вихідного потоку. За замовчуванням це консоль. Наприклад, з викликом методу Console.WriteLine() інформація автоматично передається в потік Console.Out. Властивість Console.In віднесена до стандартного вхідного потоку, джерелом якого, за заумовчуванням, є клавіатура. Наприклад, з введенням даних із клавіатури інформація автоматично передається потоку Console.In, до якого можна звернутися за допомогою

методу Console.ReadLine(). Властивість Console.Error відносять до помилок у стандартному потоці, джерелом якого також за замовчуванням, є консоль. Проте ці потоки можуть бути переспрямовані на будь-який сумісний пристрій введення – виведення (наприклад, на роботу з фізичними файлами).

Переспрямувати стандартний потік можна за допомогою методів SetIn (), SetOut () і SetError (), які є членами класу Console:

static void SetIn (TextReader input)
static void SetOut (TextWriter output)
static void SetError (TextWriter output)

Приклад 28. Приклад переспрямування потоків проілюстрований програмою, в якій двовимірний масив вводиться з файла input.txt, а виводиться у файл output.txt:

```
static void Main ()
{
        try
        {
                int [,] MyArray;
                StreamReader file=new StreamReader ("input.txt");
                // перенаправляємо стандартний вхідний потік на file
                Console.SetIn(file);
                string line=Console.ReadLine();
                string []mas=line.Split(' ');
                int n=int.Parse(mas[0]);
int m=int.Parse(mas );
                MyArray = new int [n,m];
                for (int i = 0; i < n; i++)
                {
                        line = Console.ReadLine();
                        mas = line.Split(' ');
                        for (int j = 0; j < m; j++)
                        {
                                MyArray [i,j]= int.Parse(mas [j]);
                        }
                }
                PrintArray ("початковий масив:", MyArray, n, m);
                file.Close();
}
```

```
static void PrintArray (string a, int [,] mas, int n, int m) {
        // перенаправляємо стандартний вхідний потік на file
        StreamWriterfile=new StreamWriter ("output.txt");
        Console.SetOut(file);
        Console.WriteLine(a);
        for (int i = 0; i < n; i++) {
                for (int j=0; j<m; j++) Console.Write("{0} ", mas [i,j]);
                Console.WriteLine();
        }
        file.Close();
}
```

___input.txt_____

3 4

1 4 2 8

4 9 0 1

5 7 4 2

За необхідності відновити початковий стан потоку Console.In можна таким чином:

```
// спочатку зберігаємо початковий стан вхідного потоку
TextWriter str = Console.In;
// за необхідності відновлюваний початковий стан вхідного потоку
Console.SetIn(str);
```

Аналогічним чином можна відновити початковий стан потоку Console.Out:

```
// спочатку зберігаємо початковий стан вихідного потоку
TextWriter str = Console.Out;
// за необхідності відновлюємо початковий стан вихідного потоку
Console.SetOut(str);
```

У прикладах наведено методи, що визначають стан потоків вводу та виводу.

Збереження та відновлення стану об'єктів у .NET

Серіалізація – запис структури, класу або даних у потік. Це може бути передання структури за протоколом TCP / IP для якої-небудь гри. Або може бути простий запис структури / дамп у файл, у бінарному / двійковому

форматі. Це може бути звичайний файл налаштувань для програми, файл збереження гри (save file) або власний варіант бази даних.

На відміну від програм на некерованому коді, застосунки. NET Framework не обов'язково виконуються у вигляді окремих процесів. Вони можуть існувати в межах одного процесу операційної системи у своїх власних областях, які називають *доменами програми*. Такі області можна розглядати як деякі логічні процеси віртуальної машини CLR. Використання керованого коду дозволяє гарантувати ізоляцію програм у межах своїх областей. Для передавання між доменами застосунків деякого об'єкта для його класу повинна бути визначена *процедура серіалізації*. Така процедура дозволяє зберегти стан об'єкта в деякому зовнішньому сховищі (наприклад, у файлі, або в повідомленні транспортного протоколу) за допомогою потоків вводу – виводу. Необхідна також *процедура десеріалізації*, що створює копію об'єкта. Слід зазначити, що в загальному випадку це можуть бути об'єкти різних класів і навіть створені в різних системах розробки застосунків.

Задача серіалізації об'єкта, що включає тільки поля з елементарних типів значень (нащадків класу System.ValueType) і рядків, не становить принципових труднощів. Для такого об'єкта в ході серіалізації в потік записуються самі значення всіх полів об'єкта. Однак у загальному випадку об'єкт містить посилання на інші об'єкти, які, в свою чергу, можуть взаємно посилатися, утворюючи так званий *граф об'єктів (object graph).* Самі посилання не можуть бути збережені в потоці вводу – виводу, тому основне питання серіалізації – це спосіб заміни посилань.

Класи, що виробляють серіалізацію і десеріалізацію в .NET Framework, називають *класами форматування* (formatters). У .NET Framework виділяють три різних незалежних класи форматування: XmlSerializer, SoapFormatter, BinaryFormatter. Вони використовуються як для запису та читання об'єктів з потоків вводу – виводу, так і для побудови розподілених систем:

технологія веб-служб ASP.NET використовує XmlSerializer;

технологія Remoting використовує SoapFormatter, BinaryFormatter або створений користувачем клас;

для роботи з повідомленнями MSMQ використовується XmlSerializer (через XMLMessageFormatter), або BinaryFormatter (через BinaryMessageFormatter), або створений користувачем клас;

технологія Enterprise Services заснована на Remoting і використовує BinaryFormatter.

Можна провести класифікацію можливих методів серіалізації за такими основними ознаками:

1) класифікація за видом оброблюваного графа:

а) універсальні методи: граф довільного виду з циклами;

б) довільний ациклічний граф;

в) дерево;

2) класифікація за форматом зберігання інформації в сховищі:

а) бінарні методи, які використовують двійковий формат зберігання даних, який не придатний для читання людиною без використання спеціальних засобів;

б) текстові методи використовують XML або інші текстові формати, придатні для читання або редагування людиною з використанням текстового редактора;

3) класифікація за специфікацією формату даних, отриманого в результаті серіалізації:

а) закриті методи: специфікація задається тільки за допомогою інтерфейсів усіх класів, об'єкти яких утворюють граф. У цьому випадку обидві вилучені компоненти повинні бути створені на одній мовній платформі, причому обидві сторони повинні мати як мінімум опис інтерфейсу серіалізованих класів;

б) відкриті методи: специфікація може бути задана у вигляді загальноприйнятого формату (наприклад, схеми XSD).

За цією класифікацією XmlSerializer реалізує відкритий текстовий неуніверсальний метод, BinaryFormatter – закритий двійковий універсальний метод, SoapFormatter – текстовий відкритий метод. Додатковою особливістю кожного з класів, за винятком BinaryFormatter, є обмеження на класи, які підлягають серіалізації.

Клас серіалізації Soap.SoapFormatter використовується винятково в середовищі .NET Remoting. Клас System.Runtime.Serialization.Formatters. Binary.BinaryFormatter може також використовуватися в середовищі *MSMQ* замість *XMLSerializer*. Обидва класу форматування за наведеною класифікацією є універсальними. Клас форматування BinaryFormatter реалізує двійковий закритий метод серіалізації, клас SoapFormatter – текстовий і відкритий, заснований на специфікації кодування SOAP-RPC.

Клас SoapFormatter не підтримує одне з важливих нововведень – параметризованих типів даних *(generic types).*

Обидва зазначених класи в найпростішому випадку у ході серіалізації зберігають усі поля класу (але не його властивості), незалежно від їх видимості Поля, що мають атрибут System.NonSerializeAttribute, ігноруються. Клас повинен мати атрибут System.SerializableAttribute. У ході серіалізації класу форматування використовують методи класу System.Runtime.Serialization.FormatterServices. Серіалізований клас повинен містити конструктор без параметрів, який викликається шляхом створення нового об'єкта в ході десеріалізації.

Якщо ж обробляється клас, що реалізує інтерфейс ISerializable, то він серіалізується викликом методу GetObjectData (SerializationInfo info, StreamingContext context) цього інтерфейсу, всередині якого зазвичай викликаються методи FormatterServices. Десеріалізація таких класів здійснюється викликом конструктора ISerializable (SerializationInfo info, StreamingContext context), заповнює поля об'єкта значеннями з info.

Про завершення своєї десеріалізації об'єкт може отримати повідомлення, реалізувавши інтерфейс System.Runtime.Serialization. IDeserializationCallback з єдиним методом OnDeserialization.

Отриманий таким чином на першому кроці серіалізації об'єкт класу SerializationInfo містить імена та значення серіалізованих полів. Розглянуті класи форматування, що реалізують інтерфейс IFormatter, перетворюють ці імена в деякий вид, який передається між доменами програми через потоки вводу – виводу.

Приклад 29. Розглянемо приклад створення класу з інтерфейсом ISerializable і власним механізмом серіалізації.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters;
using System.Runtime.Serialization.Formatters.Binary;
using System.Reflection;

[Serializable]
public class Person: ISerializable {

    public String name;

public Person ()    {
public void GetObjectData (SerializationInfo info, StreamingContext context)
{
```

```
Type thisType = this.GetType ();
     MemberInfo [] serializableMembers =
FormatterServices.GetSerializableMembers (thisType, context);
foreach (MemberInfo serializableMember in serializableMembers)
      {
// Не обробляти поля з атрибут NonSerializedAttribute
          if (! (Attribute.IsDefined (serializableMember,
typeof (NonSerializedAttribute))))
         {
info.AddValue (serializableMember.Name,
                  ((FieldInfo) serializableMember). GetValue (this));
            }
      }
}

protected Person (SerializationInfo info, StreamingContext context)
   {
Type thisType = this.GetType ();
MemberInfo [] serializableMembers =
            FormatterServices.GetSerializableMembers (thisType, context);
foreach (MemberInfo serializableMember in serializableMembers)
 {
FieldInfo fieldInformation = (FieldInfo) serializableMember;
          if (! (Attribute.IsDefined (serializableMember,
                              typeof (NonSerializedAttribute))))
         {
           fieldInformation.SetValue (this,
                    info.GetValue (fieldInformation.Name,
fieldInformation.FieldType));
         }
      }
   }
}// Person
```

Метод GetObjectData використовується на першому кроці серіалізації класу. У ході його роботи в об'єкт класу SerializationInfo додається інформація про поля класу, що підлягають серіалізації. Для отримання метаданих про поля класу використовується статичний метод GetSerializableMembers класу FormatterServices.

Для проведення десеріалізації клас містить конструктор спеціального виду, заповнюються поля класу значеннями з об'єкта класу SerializationInfo.

Приклад 30. Розглянемо приклад використання створеного класу Person.

```
public class SampleApp
 {
   public static void Main ()
{
using (Stream stream = new MemoryStream ())
 {
        IFormatter formatter = new BinaryFormatter ();

Person person = new Person ();
Console.WriteLine ( "Збережено: (0)", person.name);
formatter.Serialize (stream, person);

stream.Position = 0;
Person personRestored = (Person) formatter.Deserialize (stream);
Console.WriteLine ( "Відновлено: (0)", personRestored.name);
                    }
            }
}
```

Класи форматування мають механізм, що дозволяє змінити процедури серіалізації і десеріалізації для об'єктів певного класу і його нащадків. Це необхідно, зокрема, для використання віддалених об'єктів, які маршрутизуються за посиланням або не перетинають межі домену програми. Такі об'єкти знаходяться на сервері. На боці клієнта для їх використання повинен бути створений певний посередник, що реалізує весь інтерфейс віддаленого об'єкта, включаючи доступ до його полів і властивостей. Для реалізації маршрутизації за посиланням до об'єкта форматування через поле SurrogateSelector можна приєднати клас, який реалізує інтерфейс System.Runtime.Serialization.ISurrogateSelector. Він повинен пов'язувати тип віддаленого об'єкта зі спеціальною процедурою його серіалізації і десеріалізації. Використання цього механізму в .NET Remoting призводить до того, що нащадки класу MarshalByRefObject не покидають свого домену програми. З використанням ж BinaryFormatter у середовищі *MSMQ* нащадки MarshalByRefObject серіалізуються звичайним чином.

Використання класу BinaryFormatter є найбільш ефективним і універсальним, але найбільш закритим способом серіалізації. Цей клас дозволяє передавати між доменами програми довільний граф об'єктів, але з його використанням розподілена система втрачає властивість відкритості.

У разі застосування цього класу компоненти, що взаємодіють, можуть бути створені тільки на платформі CLI, причому обом сторонам необхідно мати збірку з серіалізованим типом. З використанням в якості параметрів типів зі стандартної бібліотеки чи класів, що їх використовують, бажано, щоб обидві сторони були реалізовані на одній версії CLI. Тому для передавання складних типів найкраще використовувати XML. Однак стандартний клас System.Xml.XmlDocument не може бути серіалізований класами BinaryFormatter і SoapFormatter, оскільки цей клас не має атрибута Serializable. Для серіалізації об'єктів класу XmlDocument найпростіше перетворити його в рядок, а потім серіалізувати його. Можна так само створити спадкоємця XmlDocument, який буде реалізовувати інтерфейс ISerializable.

Слід навести приклад допоміжного класу з двома статичними методами, що перетворить об'єкт класу XmlDocument у рядок, і навпаки. Оскільки метод XmlDocument.ToString(), проти очікувань, не повертає текст XML-документа і у нього немає методу, зворотного LoadXml, то слід використовувати клас StringWriter.

Приклад 31. Перетворення Xml файлу в рядок та навпаки:

```
using System;
using System.IO;
using System.Xml;

namespace Seva.Xml
 {
public static class XmlUtils
 {
      public static String XmlToString (XmlDocument xml)
           {
           StringWriter xmlLine = new StringWriter ();
           XmlTextWriter xw = new XmlTextWriter (xmlLine);
           Xml.WriteTo (xw);
           return xmlLine.ToString ();
           }

public static XmlDocument XmlFromString (String xmlLine) {
XmlDocument xml = new XmlDocument ();
xml.LoadXml (xmlLine);
return xml;
                    }
           }
}
```

Приклад 32. Двійкова серіалізація.

```
using System;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
class MouseEvent : Event
{
   //...
}

 [Serializable]
abstract class Event
{
   public void Save()
   {
      BinaryFormatter ser = new BinaryFormatter();
      Stream f = new StreamWriter("Test.gaga", false).BaseStream;

      ser.Serialize(f, this);
      f.Close();
   }
}
BinaryFormatter ser = new BinaryFormatter();
Stream f =
new StreamReader("Test.gaga", System.Text.Encoding.Default).BaseStream;
Event ev = (Event)ser.Deserialize(f);
 [Serializable]
class MouseEvent : Event, ISerializable
{
   int XCoordinate;
   public void GetObjectData(SerializationInfo info, StreamingContext context)
   {
      info.SetType(typeof(MouseEvent));
      info.AddValue("XCoordinate", XCoordinate, XCoordinate.GetType());
   }
   public MouseEvent(SerializationInfo info, StreamingContext context)
   {
      XCoordinate = (int)info.GetValue("XCoordinate", typeof(int));
   }
   //…
}
```

```
 [field: NonSerializedAttribute()]
int YCoordinate;
[OnDeserializedAttribute()]
private void RunThisMethod(StreamingContext context)
{
   YCoordinate = 777;
}
```

Усі класи серіалізації бібліотеки .NET Framework мають свої особливості й обмеження. Це може викликати значні зміни в програмному коді з переходом з одного проміжного середовища на інше. Один із способів боротьби з цією проблемою полягає у відмові від серіалізації нетривіальних класів (містять що-небудь, крім примітивних типів-значень і рядків) і особливо – складних облікових структур. Замість них, ймовірно, варто використовувати набори даних (клас System.Data.Dataset) або документи XML (клас System.Xml.XMLDocument). Хоча такий спосіб може бути не зовсім зручним для розробників, він гарантує створення незалежного від класу форматування програмного коду.

Для серіалізації об'єктів у формат JSON у просторі System.Runtime.Serialization.Json визначено клас DataContractJsonSerializer. Щоб задіяти цей клас, у проект треба додати збірку System.Runtime.Serialization.dll. Для запису об'єктів в json-файл у цьому класі є метод WriteObject (), а для читання раніше серіалізованих об'єктів – метод ReadObject (). Розглянемо їх застосування.

```
using System;
using System.IO;
using System.Runtime.Serialization.Json;
using System.Runtime.Serialization;

// В проект треба додати System.Runtime.Serialization.dll.

namespace Serialization_Json
{
   [DataContract]
   public class Person
   {
      [DataMember]
      public string Name { get; set; }
      [DataMember]
      public int Age { get; set; }
```

```
        public Person(string name, int year)
        {
            Name = name;
            Age = year;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // об'єкт для серіалізації
            Person person1 = new Person("Петро", 29);
            Person person2 = new Person("Степан", 25);
            Person[] people = new Person[] { person1, person2 };

            DataContractJsonSerializer jsonFormatter =
                new DataContractJsonSerializer(typeof(Person[]));

            using (FileStream fs = new FileStream("people.json",
                FileMode.OpenOrCreate))
            {
                jsonFormatter.WriteObject(fs, people);
            }

            using (FileStream fs = new FileStream("people.json",
                FileMode.OpenOrCreate))
            {
                Person[] newpeople =
                    (Person[])jsonFormatter.ReadObject(fs);
                foreach (Person p in newpeople)
                {
                    Console.WriteLine("Ім'я: {0} --- Вік: {1}", p.Name, p.Age);
                }
            }
            Console.ReadLine();
        }
    }
}
```

Щоб позначити, що клас Person можна серіалізувати, до нього застосовується атрибут DataContract, а до всіх його властивостей, які серіал узуються, – атрибут DataMember.

Метод WriteObject () приймає два параметри: файловий потік FileStream і об'єкт, який треба серіалізувати (в цьому випадку – масив

об'єктів Person). А метод ReadObject () приймає як параметр файловий потік, який представляє файл у форматі JSON.

Введення – виведення даних у Java

Основна інфраструктура вводу – виводу Java визначено в пакеті java.io. Java має два типи потоків: байтові та символьні.

Байтові потоки забезпечують зручні засоби для обробки вхідних і вихідних даних у вигляді байтів.

Символьні потоки призначено для обробки вхідних і вихідних даних у вигляді символів. Вони використовують Unicode і тому можуть бути інтернаціоналізовані.

Байтові потоки

Вони визначаються за допомогою двох базових абстрактних класів: InputStream і OutputStream. Нащадки InputStream – це потоки вводу даних, а нащадки OutputStream – потоки виводу даних.

Важливі класи байтових потоків наведено в табл. 9.19.

Таблиця 9.19

Класи байтових потоків у Java

Класи байтового потоку	Опис
InputStream	Абстрактний клас для опису потока вводу
OutputStream	Абстрактний клас для опису потока виводу
BufferedInputStream	Потік вводу з підтримкою буферизації
BufferedOutputStream	Потік виводу з підтримкою буферизації
DataInputStream	Призначено для читання даних стандартних типів
DataOutputStream	Призначено для запису даних стандартних типів
FileInputStream	Потік вводу для читання даних із файлу
FileOutputStream	Потік виводу для запису даних у файл
PrintStream	Потік виводу, що містить методи print() і println()

Ці класи мають низку методів. Найбільш важливими з них є read і write, які призначено, відповідно, для читання або запису байтів.

Символьні потоки

Їх визначають за допомогою двох базових абстрактних класів: Reader і Writer. Нащадки Reader – це потоки вводу даних, а нащадки Writer – потоки виводу даних.

Важливі класи символьних потоків наведено в табл. 9.20.

Таблиця 9.20

Класи символьних потоків

Класи символьного потоку	Опис
Reader	Абстрактний клас для опису потока вводу
Writer	Абстрактний клас для опису потока виводу
BufferedReader	Потік вводу з підтримкою буферизації
BufferedWriter	Потік виводу з підтримкою буферизації
FileReader	Потік вводу для читання даних із файлу
FileWriter	Потік виводу для запису даних у файл
InputStreamReader	Потік вводу, який транслює байти в символи
OutputStreamWriter	Потік виводу, який транслює символи в байти
PrintWriter	Потік виводу, що містить методи print() і println()

Найбільш важливими методами цих класів є read і write, які призначено, відповідно, для читання або запису символів.

Слід розглянути деякі приклади використання потоків вводу – виводу Java.

Приклад 33. Введення даних із консолі.

```
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());
```

Тут System.in – стандартний байтовий потік вводу; readLine-метод для порядкового читання даних.

Приклад 34. Посимвольне файлове введення-виведення текстових даних.

```
File inputFile = new File("in.txt");
File outputFile = new File("out.txt");
FileReader in = new FileReader(inputFile);
FileWriter out = new FileWriter(outputFile);
int c;
while ((c = in.read()) != -1)
    out.write(c);
in.close();
out.close();
```

Тут дані зчитуються посимвольно з файла in.txt і записуються до файлу out.txt.

Також у Java SE є пакет java.nio. Він визначає порядок вводу – виводу даних з використанням буферів у пам'яті.

Основними абстракціями цього пакету є буфери та кодировки.

Буфери є контейнерами для даних різних типів. Наприклад, є класи IntBuffer, DoubleBuffer, CharBuffer тощо.

Кодировки – це визначені співвідношення між послідовностями байтів і символів. Основний клас – Charset.

Пакет java.nio містіть декілька корисних класів, наприклад: java.nio.file.Files та java.nio.file.Paths. Клас java.nio.file.Files містіть статичні методи для роботи з файлами та каталогами. Клас java.nio.file.Paths надає статичні методи для створення шляхів до файлів або каталогів.

Слід навести приклади використання класів пакету java.nio.

Приклад 35. Читання даних з текстового файлу невеликого розміру в кодировці UTF-8.

```
// Повне ім'я файлу: C:/tutorial/ wiki.txt
Path wiki_path = Paths.get("C:/tutorial", "wiki.txt");
Charset charset = Charset.forName("UTF-8");
// читання контенту текстового файлу з початку до кінця
 List<String> lines = Files.readAllLines(wiki_path, charset);
// читання контенту бінарного файлу з початку до кінця
byte[ ] fileArray = Files.readAllBytes(wiki_path);
```

Приклад 36. Використання буферизованого потоку для порядкового читання даних із файлу в кодировці ASCII.

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader =  Files.newBufferedReader(file, charset)) {
 String line = null;
 while ((line = reader.readLine()) != null) {
          System.out.println(line);
  } }
catch (IOException x) {
      System.err.format("IOException: %s%n", x); }
```

У прикладах наведено принципи використання буферизації.

Питання для самопідготовки

1. Геш-таблиці та геш-функції.
2. Сортування вмісту контейнера.
3. Розроблення користувальницьких контейнерів.
4. Засоби форматування рядків.

Запитання для самодіагностики

1. Охарактеризуйте склад бібліотеки контейнерів Java SE та Microsoft .NET.
2. Які основні інтерфейси існують у бібліотеці контейнерів Java SE та Microsoft .NET?
3. Дайте стислу характеристику структури даних "динамічний масив" і її реалізації в Java SE та Microsoft .NET.
4. Дайте стислу характеристику структури даних "двозв'язний список" і її реалізації в Java SE та Microsoft .NET.
5. Що таке "геш-таблиця" і "геш-функція"?
6. Дайте стислу характеристику структури даних "асоціативний масив" і її реалізації в Java SE та Microsoft .NET.
7. Які засоби розбору рядків на лексеми є у Java SE та Microsoft .NET?
8. Які особливості реалізації рядкового типу даних є у Java SE та Microsoft .NET?
9. Охарактеризуйте класи стандартної бібліотеки Java SE та Microsoft .NET для подання рядків.
10. До яких елементів програми можливе застосування атрибутів?
11. Для чого використовують атрибути умовної компіляції?
12. Як створити клас, об'єкти якого можна серіалізувати?
13. Які ви знаєте формати серіалізації?
14. Укажіть базові класи байтових і символьних потоків введення – виведення Java SE та Microsoft .NET.
15. Для чого призначено основні класи символьних потоків введення – виведення Java SE та Microsoft .NET?
16. Для чого призначено основні класи байтових потоків введення – виведення Java SE та Microsoft .NET?

Література: [9; 11; 17; 19; 20; 24].

Розділ 4. Об'єктно-орієнтоване програмування застосувань із графічним інтерфейсом користувача

10. Основи розроблення графічних інтерфейсів користувача

Мета теми: набуття знань щодо послідовності оброблення подій, основних елементів XML та основ використання FXML для опису графічного інтерфейсу користувача.

Професійні компетентності: здатність до використання сучасних технологій розроблення графічних інтерфейсів користувача.

Основні питання:

10.1. Загальні відомості про події. Генерування подій. Обробники подій.

10.2. Огляд сучасних технологій розроблення застосунків із графічним інтерфейсом користувача на платформах Microsoft .NET і Java SE. Основи мови XML. Структура XML-документа. Простори імен XML. XML-схеми.

10.3. Загальні відомості про FXML. Використання FXML для опису графічного інтерфейсу користувача.

Питання для опрацювання: загальні відомості про події, технології розроблення застосунків із графічним інтерфейсом користувача, XML, загальні відомості про мову FXML.

Ключові слова: подія, графічний інтерфейс, видавець, підписник, мова розмітки, XML, Windows Forms, Swing, макет інтерфейсу, Windows Presentation Foundation, JavaFX, елементи управління, CSS, Scene Builder, дескриптор, атрибут, вузол.

10.1. Загальні відомості про події. Генерування подій. Обробники подій

Найчастіше події використовуються в застосунках з графічним інтерфейсом користувача, в яких елементи управління (такі, як кнопка або інші) можуть реагувати на деяку подію. У відповідь може з'являтися діалогове вікно повідомлення. Генерування події викликається натисканням мишею на кнопку.

Однак використання подій не обмежене програмами з графічним інтерфейсом – вони можуть бути дуже корисними в звичайних консольних програмах.

Інфраструктура подій часто реалізується на базі шаблона проектування "Спостерігач".

Події працюють таким чином. Для об'єкта, який має реагувати на виникнення події, в коді програми реєструють обробник цієї події. **Обробник події** – це деяка підпрограма або блок програмного коду. Якщо очікувана подія відбувається, викликаються всі зареєстровані обробники.

Подія – деякий спеціальний стан, у якому може опинитися об'єкт класу. Такі стани дозволяють об'єкту повідомляти інші об'єкти про виникнення будь-яких ситуацій.

Клас, що відправляє (або викликає) подію, називають **видавцем**, а класи, які приймають (або обробляють) подію, – **підписниками**. Видавець визначає момент виклику події, підписники визначають відповідну дію. У події може бути кілька підписників. Підписник може обробляти кілька подій від декількох видавців. Події, що не мають підписників, ніколи не виникають.

10.2. Огляд сучасних технологій розроблення застосунків із графічним інтерфейсом користувача на платформах *Microsoft .NET* і *Java SE*. Основи мови XML. Структура XML-документа. Простори імен XML. XML-схеми

Технології розроблення застосунків із графічним інтерфейсом користувача на платформах Microsoft .NET і Java SE

Історично на платформах Microsoft .NET і Java SE використовувалось декілька технологій та відповідних бібліотек для розроблення графічного інтерфейсу користувача.

Так, з 2002 р. для Microsoft .NET існує технологія Windows Forms. Хоча на сучасному етапі вона вважається застарілою, можливість створення застосунків Windows Forms є й у сучасному інтегрованому середовищі розробки *Microsoft Visual Studio 2017.*

Щодо платформи Java SE, то з 1995 р. для створення графічного інтерфейсу користувача використовувався Abstract Window Toolkit (AWT), на зміну якому в 1998 р. прийшов Swing. Ці технології також застарілі.

Спільною рисою всіх зазначених технологій є те, що графічний інтерфейс можна було розробляти тільки з використанням деякої мови програмування, наприклад, C# у випадку Windows Forms і Java для AWT та Swing. Це ускладнювало взаємодію між дизайнерами графіки та розробниками програмного коду.

Типова послідовність такої взаємодії могла бути такою:

дизайнер створює макет інтерфейсу користувача в графічному редакторі;

цей дизайн реалізується програмістом деякою мовою програмування;

дизайн, реалізований програмістом, не схожий на початковий;

скандали, доопрацювання тощо.

Очікування користувачів щодо зовнішнього вигляду та функціональних можливостей графічного інтерфейсу постійно зростають. Отже, з розвитком веб- і мобільних технологій, появленням нових типів пристроїв цей процес значно прискорився.

Тому стало очевидно, що необхідні нові сучасні технології розроблення графічного інтерфейсу користувача, які будуть задовільнювати вимогам користувачів.

Таким чином, з'явилися технології Windows Presentation Foundation (платформа Microsoft .NET, 2006 р.) і JavaFX (платформа Java SE, 2008 р.). Однією з їхніх особливостей є те, що тепер інтерфейс користувача розробляється XML-подібною декларативною мовою XAML (для Windows Presentation Foundation) або FXML (для JavaFX), а логіка програми – мовою програмування, сумісною з платформою Microsoft .NET або Java SE.

Тепер співпраця дизайнера та розробника набула нової змістовності:

дизайнер створює макет інтерфейсу користувача в Microsoft Blend, Microsoft Visual Studio (для Windows Presentation Foundation) чи Scene Builder (для JavaFX);

отриманий XAML- або FXML-файл передається програмісту;

програміст додає до застосунку обробники подій та інший код, розроблені деякою мовою програмування (наприклад, C# або Java);

усі щасливі.

Розглянемо більш ретельно технологію JavaFX.

JavaFX – це нова технологія розроблення графічного інтерфейсу користувача на платформі Java SE. Вона є набором графічних і мультимедійних

API, які можна використовувати для створення та розгортання клієнтських програм з "насиченим" інтерфейсом.

Поточна версія – JavaFX 11. Розробляється та постачається компанією Gluon Solutions (https://gluonhq.com) у вигляді окремого модуля.

JavaFX дозволяє розробникам швидко створювати крос-платформні програми. Технологія підтримує сучасні графічні процесори за допомогою апаратно-прискореної графіки, поєднуючи в застосунку графіку, анімацію та елементи управління графічного інтерфейсу користувача.

Код JavaFX написано на Java, а код програми JavaFX може посилатися на API з будь-якої бібліотеки Java.

Зовнішній вигляд застосунків JavaFX можна налаштувати за допомогою каскадних таблиць стилів (CSS), аналогічних тим, що використовуються на веб-сторінках. Ними можуть користуватись як дизайнери графіки, так і програмісти.

Графічний інтерфейс програми зазвичай створюється XML-подібною мовою FXML. Це можна робити вручну або з використанням візуального засобу Scene Builder.

Розроблення графічного інтерфейсу JavaFX ручним способом є досить трудомістким процесом, за якого існує велика ймовірність помилок.

Сьогодні Scene Builder також підтримується компанією Gluon Solutions. *Поточна версія* – Scene Builder for Java 11. У цьому середовищі інтерфейс користувача створюється за допомогою перетягування. У результаті мовою FXML автоматично створюється відповідний код, який зберігається у файлі з розширенням .fxml. Згодом його можна перенести до інтегрованого середовища розробки, наприклад Eclipse, щоб розробники могли додати бізнес-логіку застосунку.

На практиці ці два способи часто використовують сумісно.

Для кодування логіки JavaFX-програми застосовують мову програмування Java або іншу, сумісну з платформою Java SE.

JavaFX має багато сучасних елементів управління для розроблення графічних інтерфейсів користувача будь-якої складності. Зокрема, елемент управління WebView призначено для створення веб-інтерфейсу. Він може відображати веб-сторінки, підтримує JavaScript, веб-сокети, веб-працівники та веб-шрифти. Він також дозволяє друкувати веб-сторінки.

JavaFX підтримує 3D-графіку, малювання на базі Canvas API, операції мультитач і дисплеї з високою роздільною здатністю.

Для розроблення застосунків JavaFX 11 попередньо необхідно:

1) завантажити OpenJDK 11 з https://jdk.java.net/11/ і встановити його;

2) завантажити з веб-сайту компанії Gluon Solutions (https://gluonhq.com /products/javafx/) програмне забезпечення JavaFX SDK 11 і виконати його інсталяцію.

Розробляти JavaFX-програми можна в будь-якому текстовому редакторі, а запускати їх на виконання – в командній оболонці цільової операційної системи. Але для більшої зручності рекомендується використовувати Scene Builder for Java 11 та інтегроване середовище розробки (таке, як Eclipse, IntelliJ IDEA або Apache NetBeans).

Мова XML

Розширювана мова розмітки (Extensible Markup Language – XML) становить технологію, яка охоплює широке коло застосувань – від опису конфігурації застосунків до передання інформації між веб-службами. **XML** – це спосіб зберігання даних у простому текстовому форматі. Це означає, що він може бути прочитаний майже на будь-якому комп'ютері, роблячи його неперевершеним форматом для передавання даних через Internet.

XML надзвичайно важлива в світі .NET, оскільки використовується як формат за замовчуванням для передавання даних, а тому важливо розуміти його основи.

Повний набір даних у XML відомий як документ XML. *Документом* XML може бути фізичний файл на комп'ютері або просто рядок у пам'яті. Проте він має бути завершений і повинен слідувати певним правилам. Документ XML складається з безлічі різних частин. Найбільш важливі з них – елементи XML, які містять саме дані документа.

Елементи XML складаються з відкривального дескриптора (імені елемента, що міститься в кутових дужках, наприклад: <myElement>), даних усередині елемента та закривального дескриптора (подібного до відкривального дескриптора, але з провідним слешем після відкривальної дужки: </myElement>).

Приклад 1. Визначити елемент для зберігання заголовка книги можна таким чином:

```
<book>Tristram Shandy</book>
```

Синтаксис мови XML дуже схожий на синтаксис HTML. Головна відмінність у тому, що XML не має зумовлених елементів – користувач сам вибирає імена для своїх елементів, тому ніщо не обмежує їх кількість. Найбільш важливий момент, про який потрібно пам'ятати, полягає в тому, що попри назву XML – зовсім не мова. Це скоріше стандарт для визначення мов (відомих як XML-застосунки). Кожна мова має свій, відмінний від інших словник – певний набір елементів, які можуть застосовуватися в документі, і структуру, яку можуть приймати ці елементи. Водночас можна явно обмежувати допустимі елементи в документі XML та альтернативно – дозволити будь-які елементи, а самій програмі, що використовує документ, – визначати структуру.

Імена елементів залежать від регістра, тому <book> і <ВООК> трактуються як різні елементи. Це означає, що зі спробою закрити елемент <book> з використанням закривального дескриптора, записаного в іншому регістрі (наприклад, </ВООК>), XML-документ не буде правильним. Програми, що читають XML-документи й аналізують їх індивідуальні елементи, відомі як XML-аналізатори (XML parsers). Вони відхиляють будь-який документ, що містить неправильний XML.

Елементи також можуть містити в собі інші елементи. Тому можна модифікувати елемент <book> так, щоб включити інформацію про автора в заголовку книги в двох піделементах:

```
<book>
<title>Tristram Shandy</title>
<author>Lawrence Sterne</author>
</book>
```

Перекриття елементів не допускається, тому піделементи повинні закриватися перед появою закривального дескриптора батьківського елемента. Наприклад, не можна зробити так:

```
<book>
<title>TristramShandy
<author>LawrenceSterne
</title></author>
</book>
```

Це некоректно, оскільки елемент <author> відкритий усередині елементу <title>, але закривальний дескриптор </title> знаходиться перед закривальним дескриптором </autho>.

Існує виняток із правила, що вимагає, щоб усі елементи мали закривальний дескриптор. Допускаються "порожні" елементи, які не мають вкладених даних або тексту. У цьому випадку можна просто додавати закривальний дескриптор відразу після відкривального, як показано в попередньому коді, або використовувати скорочений синтаксис, додаючи слеш закривального дескриптора в кінець відкривального:

<book />

Це ідентично такому синтаксису:

<book></book>

Разом із зберіганням даних у тілі елемента можна також зберігати їх усередині атрибутів, які додаються до дескриптора, що відкриває елемент. Атрибути мають таку форму:

name="value"

Тут значення атрибуту має бути поміщене в одиночні або подвійні лапки. Наприклад:

<booktitle="TristramShandy"></book>

або

<booktitle='TristramShandy'></book>

Обидва наведених варіанти правильні, а наступний – ні:

<book title=Tristram Shandy></book>

Навіщо потрібно два способи зберігання даних у XML? У чому різниця між наступними двома способами запису?

<book>
<title>Tristram Shandy</title>
</book>

і

<book title="Tristram Shandy"></book>

Фактично між ними немає якоїсь фундаментальної відмінності. Жодний зі способів не переважає над іншим. **Елементи** – більш вдалий вибір,

якщо необхідно додавати інформацію про цю частину даних пізніше. Завжди можна додати піделемент або атрибут до елемента, чого не можна робити з атрибутами. Елементи більш читанні й елегантніші (але це – справа особистого смаку). На противагу, атрибути споживають менше смуги пропускання, коли документ пересилається мережею без стискування (зі стискуванням різниця незначна). Вони зручні для зберігання інформації, яка не істотна для кожного користувача документа.

Оголошення XML

На додаток до елементів і атрибутів XML-документи можуть містити множину інших складових. Ці індивідуальні частини документа XML відомі як вузли (nodes); елементи, текст усередині елементів і атрибути. Усе це вузли документа XML.

Деякі елементи використовуються досить рідко, проте один тип вузла з'являється майже в кожному документі XML – це оголошення XML. Якщо ви його включаєте, воно повинне знаходитися в першому вузлі документа.

Оголошення XML за своїм форматом подібне до елемента, але має усередині знаки запитань і кутові дужки. Воно завжди має ім'я xml і завжди забезпечено атрибутом на ім'я version; єдине допустиме значення для нього – 1.0. Проста можлива форма XML-оголошення така:

<?xml version="1.0"?>

Додаткове оголошення XML також містить:

атрибути encoding зі значенням, що вказує символьний набір, який має бути використаний для прочитання документа – такий, як "UTF-16". Він показує, що документ використовує 16-бітовий символьний набір Unicode;

standalone із значеннями "yes" або "no", щоб указати, чи залежить XML-документ від будь-яких інших файлів.

Проте ці атрибути не обов'язкові. І ймовірно, доцільно включити тільки атрибут version у ваші XML-файли.

Структура документа XML

Одна з найбільш важливих характеристик XML полягає в тому, що він надає спосіб структуризації даних, який істотно відрізняється від реляційних баз даних. Більшість сучасних систем управління базами даних зберігають інформацію в таблицях, які зв'язані між собою через значення певних стовпців. Таблиці зберігають дані в рядках і стовпцях: кожен рядок

є окремим записом, а кожен стовпець в ній – певним елементом даних у цьому рядку. На відміну, дані XML структуровані ієрархічно подібно до організації папок і файлів у Windows Explorer. Кожен документ повинен мати єдиний кореневий елемент, усередині якого містяться всі елементи та текстові дані. Якщо на вершині документа знаходиться більше одного елемента, то такий документ не вважається за правильний документ XML. Проте можна включити інші вузли XML у верхній рівень – особливе оголошення XML. Правильний документ XML виглядає так:

```
<?xml version=1.0"?>
<books>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
</books>
```

А цей фрагмент не є правильним документом XML:

```
<?xml version="1.0"?>
<book>Tristram Shandy</book>
<book>Moby Dick</book>
<book>Ulysses</book>
```

Під кореневим елементом надана значна свобода відносно структури даних. На відміну від реляційних даних, в яких кожний рядок має однакову кількість стовпців, тут немає обмежень на число піделементів, які може містити елемент. Документи XML часто структуровані подібно до реляційних даних, проте документи XML з елементом для кожного запису не потребують ніяких зумовлених структур. Це одна з основних відмінностей між традиційними реляційними базами даних і XML. Тоді як реляційні бази даних завжди визначають структуру інформації перед додаванням яких-небудь даних, інформація може бути збережена в XML без початкових накладних витрат, що дуже зручно для зберігання невеликих блоків даних. Цілком можливо подати структуру вашого XML (на відміну від реляційних баз даних) без указівки цієї структури явно.

Простори імен XML

Як відомо, будь-хто може визначати власні класи C# та елементи XML. Це призводить до виникнення очевидної проблеми: як дізнатися, які

елементи до якого словника належать? Відповідь – за допомогою просторів імен. Точно так, як визначаються простори імен для організації ваших типів C#, можна використовувати простори імен XML для визначення власних словників XML. Це дозволяє включати елементи безлічі різних словників усередині єдиного документа XML, не ризикуючи неправильно інтерпретувати їх, тому що (наприклад) два різних словники визначають елемент <customer>.

Простори імен XML можуть бути достатньо складними, проте базовий синтаксис простий. Певні елементи або атрибути асоційовані з певним простором імен за допомогою префікса, за яким слідує двокрапка. Наприклад, <wrox:book> представляє елемент <book>, що знаходиться в просторі імен wrox. Як ви дізнаєтеся, який простір імен представляє wrox? Щоб цей підхід працював, ви повинні гарантувати унікальність кожного простору імен. Простий спосіб домогтися цього – відобразити префікси на щось, унікальність чого відома. Десь у вашому документі XML ви повинні асоціювати будь-які префікси простору імен з універсальним ідентифікатором ресурсів (Uniform Resource Identifier – URI). URI існують у декількох іпостасях, але найбільш поширений вигляд – проста адреса веб, наприклад, www. wrox. com.

Щоб ідентифікувати префікс із певним простором імен, використовуйте атрибут xmlns:prefix усередині елемента. Для цього встановіть це значення в унікальний URI, який ідентифікує простір імен. Префікс може застосовуватися в будь-якому місці елемента, включаючи будь-які вкладені дочірні елементи:

```
<?xml version=1.0"?>
<books>
<book xmlns:wrox="http://www.wrox.com">
<wrox: title>Beginnmg C#</wrox: title>
<wrox:author>Karli Watson</wrox:author>
</book>
</books>
```

Тут можна використовувати префікс wrox: з елементами <title> і <author>, тому що вони знаходяться усередині елемента <book>, де визначений префікс. Проте, якщо ви спробуєте додати цей префікс до елемента <books>, то XML стане неправильним, оскільки префікс для цього елемента не визначений.

Для елемента можна також визначити простір імен за замовчуванням, використовуючи атрибут xmlns:

```
<?xml version=1.0"?>
<books>
<book xmlns="http://www.wrox.com">
<title>Beginning Visual C# </title>
<author>Karli Watson </author>
<html:img src="begvcsharp.gif"
xmlns:html="http://www.w3.org/1999/xhtml" />
</book>
</books>
```

Тут простір імен за замовчуванням для елемента <book> визначений як "http://www.wrox.com". Тому все, що знаходиться усередині цього елемента, буде належати до цього простору імен. За умови, що ви явно не специфікуєте протилежне, додавши префікс іншого простору імен, як робите це для елемента <img> (коли ви встановлюєте простір імен, використовуваний XML-сумісними документами HTML).

Правильно оформлений і дійсний XML

До цих пір говорилося про правильний (legal) XML. Фактично XML створює відмінність між двома формами правильності. Документи, відповідні всім правилам стандарту XML, вважаються правильно оформленими (well-formed). Якщо документ XML не є правильно оформленим, то аналізатори не зможуть інтерпретувати його коректно та відхилять такий документ. Щоб бути правильно оформленим, документ може відповідати таким вимогам:

мати один, і лише один кореневий елемент;

мати закривальні дескриптори для кожного елемента (за винятком скороченого синтаксису);

не мати перекривальних елементів. Усі дочірні елементи мають бути повністю поміщені усередині батьківського;

мати всі атрибути, поміщені в дужки.

Це не повний перелік вимог, але він висвічує найбільш поширені помилки, що допускаються програмістами – новачками в XML. Проте документи XML можуть відповідати всім цим правилам і, проте, не бути дійсними (valid). Як зазначалось, XML не є мовою, а швидше стандартом для

визначення застосунків XML. Правильно оформлені документи XML просто відповідають стандарту XML. Щоб бути дійсними, вони також повинні відповідати всім правилам, специфікованим для застосунків XML. Не всі аналізатори перевіряють дійсність документів; ті, що виконують цю функцію, називають *перевіряльними аналізаторами*. Щоб переконатись, чи відповідає документ правилам застосунку, потрібен спосіб визначення цих правил.

XML підтримує два шляхи визначення того, які елементи й атрибути можуть бути поміщені в документ і який порядок визначення типів документів (Document Type Definitions – DTD) і схеми. DTD використовують відмінний від XML синтаксис, успадкований від батька XML, і поступово замінюються схемами. DTD не дозволяють специфікувати типи даних елементів і атрибути, а тому вони не гнучкі та не набули широкого використання в контексті .NET Framework. На противагу цьому, схеми використовуються часто; вони дозволяють специфікувати типи даних і написані в XML-сумісному синтаксисі. На жаль, схеми дуже складні; існують різні форми для їх визначення, навіть усередині технології .NET.

Відомі два формати схем, що підтримуються .NET: мова визначення схем XML (XML Schema Definition – XSD) і скорочені схеми XML-даних (XML-Data Reduced – XDR). Визначення схем XDR – старіший зі стандартів, які належать Microsoft. Зазвичай він не використовується і не розпізнається аналізаторами, не притаманними Microsoft. XSD – відкритий стандарт, рекомендований W3C, і тому його визначення присутнє тут.

Приклад 2. Розглянемо просту схему XSD для простого документа XML, що містить базові деталі про пару книг Wrox на тему C#:

```
<?xml version=1.0"?>
<books>
<book>
<title>Beginning Visual C#</title>
<author>Karli Watson</author>
<code>7582</code>
</book>
<book>
<title>Professional C# 2nd Edition</title>
<author>Simon Robinson</author>
<code>7043</code>
</book>
</books>
```

Схеми можуть як включатися всередину XML-документа, так і зберігатися в окремому файлі. Треба добре ознайомитись з XML, перш ніж приступати до написання схем. Необхідно навчитись розпізнавати головні елементи схеми.

Схеми XSD

Елементи в схемах XSD повинні належати до простору імен http://www.w3.org/2001/XMLSchema. Якщо цей простір імен не буде включений, елементи схеми не будуть розпізнані.

Щоб асоціювати документ XML з схемою XSD в іншому файлі, необхідно додати елемент schemalocation до кореневого елемента:

```
<?xml version=1.0"?>
<books schemalocation="file://C:\Chapter 25\books.xsd">
</books>
```

Розглянемо приклад схеми XSD:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
<element name="books">
<complexType>
<choice maxOccurs="unbounded">
<element name="book">
<complexType>
<sequence>
<element name="title" />
<element name="author" />
<element name="code" />
</sequence>
</complexType>
</element>
</choice>
<attribute name="schemalocation" />
</complexType>
</element>
</schema>
```

Слід зауважити, що простір імен за замовчуванням установлено в простір. Це повідомляє аналізатор, що елементи документа належать до схеми. Якщо ви не специфікуєте цей простір імен, то аналізатор вважатиме, що це просто нормальні елементи XML, і не зрозуміє, що він повинен застосовувати їх для перевірки.

Уся схема міститься усередині елемента під назвою <schema>. Кожен елемент, який може з'явитися в документі, має бути представлений елементом <element>. Цей елемент має атрибут name, що вказує ім'я елемента. Якщо елемент повинен містити в собі дочірні елементи, треба передбачити для них дескриптори <element> усередині елемента <contextType>. Усередині цього ви специфікуєте, як саме дочірні елементи повинні з'являтися.

Наприклад, ви використовуєте елемент <choice> (щоб специфікувати будь-який вибір дочірніх елементів, що допускається) або <sequence> (щоб указати, що дочірні елементи повинні з'являтися в тому ж порядку, як вони зазначені в схемі). Якщо елемент може з'являтися більше одного разу (як елемент <book>), потрібно помістити атрибут maxOccurs усередину батьківського елемента. Установка його в "unbounded" означає, що елемент може з'являтися необмежену кількість разів. І нарешті, будь-які атрибути мають бути представлені елементами <attribute>, включаючи ваш атрибут schemalocation, який повідомляє аналізатору, де шукати схему. Помістіть його після закінчення списку дочірніх елементів.

10.3. Загальні відомості про мову FXML. Використання FXML для опису графічного інтерфейсу користувача

Мова розмітки інтерфейсу користувача (МРІК) – це мова розмітки, яка призначена для опису й відтворення графічного інтерфейсу та елементів управління. Сьогодні існує біля двох десятків таких мов. Багато з них є діалектами мови XML.

Типова МРІК визначає код програми, що часто використовується повторно, у вигляді розмітки. Це полегшує фокусування на дизайні інтерфейсу користувача замість фокусування на розроблені програмного коду, призначеного для створення та налагодження елементів візуального інтерфейсу.

Під час виконання програми код розмітки перетворюється в програмний код для цільової апаратно-програмної платформи. У XML-подібних МРІК розмітка подається в пам'яті у вигляді дерева вузлів, яким можна керувати під час виконання коду програми.

FXML – це XML-подібна МРІК, створена компанією Oracle для визначення користувальницького інтерфейсу JavaFX-програм.

Приклад 3. Розглянемо спрощений приклад FXML для створення простого графічного інтерфейсу JavaFX:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.AnchorPane?>

<AnchorPane>
  <children>
    <Button text="Очистити" />
    <Button text="Повідомлення" />
    <Label text="Тут буде текст" />
  </children>
</AnchorPane>
```

Цей приклад визначає AnchorPane, що містить три дочірніх елемента управління: два з них мають тип Button, а третій – Label. Компонент AnchorPane є панеллю компонування JavaFX і визначає відносне розташування дочірніх елементів. Button – це кнопка. Label просто показує текст у графічному інтерфейсі. Властивість text цих елементів визначає текст, що буде в них відображений.

Перший рядок документа FXML є стандартним для XML-документів. Наступні чотири рядки є операторами імпорту. Вони призначені для імпорту необхідних класів JavaFX і Java SE, які ви хочете використовувати.

Таким чином, маємо деревоподібне подання графічного інтерфейсу в пам'яті, зображене на рис. 10.1.

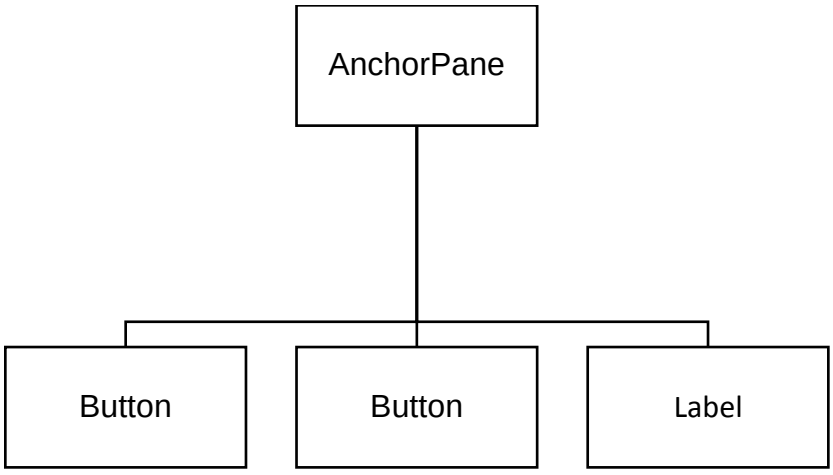

Рис. 10.1. **Подання графічного інтерфейсу JavaFX у пам'яті**

Тут AnchorPane – корневий вузол, Button та Label – підпорядковані вузли одного рівня. Після завантаження FXML-файла в пам'ять JavaFX-програми вони створюються об'єкти відповідних класів.

Наостанок зазначимо, що FXML дозволяє створювати складні інтерфейси з використанням усіх існуючих елементів управління JavaFX.

Питання для самопідготовки

1. Декларативний опис елементів графічних інтерфейсів користувача.
2. Інтерфейси зворотного виклику.
3. Події та їх оброблення.

Запитання для самодіагностики

1. Що таке "подія"?
2. Дайте характеристику механізму оброблення подій.
3. Обґрунтуйте використання слухачів подій у Java SE.
4. Назвіть основні елементи синтаксису мови XML.

Література: [9; 14; 15; 21].

11. Розроблення графічних інтерфейсів користувача на платформі Java SE

Мета теми: набуття знань щодо основних елементів JavaFX.

Професійні компетентності: здатність до розроблення сучасних графічних інтерфейсів користувача.

Основні питання:

11.1. Загальна структура застосунку JavaFX. Форми. Події рівня форми. Використання форм і базових елементів управління JavaFX.

11.2. Оброблення подій від миші, клавіатури й елементів управління. Використання основних елементів управління JavaFX.

Питання для опрацювання: загальна структура застосунку JavaFX, використання форм і базових елементів управління JavaFX, оброблення подій, використання основних елементів управління JavaFX.

Ключові слова: форма, елемент управління, обробник події, життєвий цикл застосунку, сцена, епізод, контролер, діалогове вікно, панель компонування.

11.1. Загальна структура застосунку JavaFX. Форми. Події рівня форми. Використання форм і базових елементів управління JavaFX

Застосунки з графічним інтерфейсом мають деякі особливості:

1) застосунок, як правило, має не менше одного вікна (форми). **Форма,** видима поверхня, на якій відображається інформація для користувача;

2) на форму можна помістити елементи управління. **Елемент управління** – це окремий елемент інтерфейсу користувача, призначений для відображення або введення даних;

3) форми й елементи управління є об'єктами деяких класів;

4) під час виконання користувачем якої-небудь дії з формою або одним з її елементів управління генерується подія;

5) застосунок може реагувати на події шляхом виклику методу – обробника події;

6) у С-подібному застосунку також є метод **main**, у якому звичайно створюється об'єкт форми.

Це у повній мірі стосується застосунків на базі JavaFX.

API Java FX 8 і старші містять кілька сотень класів та інтерфейсів. Але є деякі класи, що прямо або опосередковано використовуються в JavaFX-застосунку будь-якої складності. Це таки класи, як Application, Stage, Scene, Control.

Клас Application призначений для керування життєвим циклом застосунка. Це абстрактний клас, визначений у пакеті javafx.application. Його головні методі наведено в табл. 11.1.

Таблиця 11.1

Клас Application

Ім'я методу	Опис
init	Метод ініціалізації застосунку – "заглушка"
start	"Точка входу" для застосунку. Абстрактний метод, має бути перевизначений у похідному класі
stop	Викликається перед завершенням програми "заглушка"
launch	Запускає застосунок шляхом виклику перевизначеного методу start. Має викликатися тільки один раз

Життєвий цикл застосунку JavaFX містить кілька етапів:

1) створення об'єкта, похідного від javafx.application.Application;

2) виклик метода init;

3) виклик метода start;

4) очікування завершення застосунку. Застосунок завершується, якщо був викликаний метод exit класу javafx.application.Platform або користувач закрив останнє вікно за стосунку;

5) виклик метода stop.

Таким чином, у загальному випадку, в застосунку JavaFX потрібно перевизначити тільки метод start класу javafx.application.Application, а для його запуску викликати метод launch.

Для опису вікон і контенту, який в них міститься, JavaFX використовує театральну термінологію. Сам **фрейм** – це сцена, а **контент вікна** – це один або декілька послідовних епізодів.

Клас javafx.stage.Stage ("сцена") – відповідає за фрейм, а *клас javafx.scene.Scene* ("епізод") – за контент вікна. Ці класи мають деякі важливі елементи (табл. 11.2, 11.3).

Таблиця 11.2

Важливі елементи класу javafx.stage.Stage

Ім'я властивості	Опис	Методи для роботи з властивістю
Iconified	Вікно згорнуто чи ні	setIconified, isIconified
maximized	Вікно розгорнуто чи ні	setMaximized, isMaximized
Resizable	Чи може користувач змінити розмір вікна	setResizable, isResizable
Title	Заголовок вікна	setTitle, getTitle
Focused	Чи знаходиться вікно у фокусі вводу	isFocused, setFocused
x, y	Координати лівого верхнього кута вікна	setX, getX, setY, getY
Scene	"Епізод", який відображається на "сцені"	setScene, getScene

Таблиця 11.3

Важливі елементи класу javafx.scene.Scene

Ім'я властивості	Опис	Методи для роботи з властивістю
Cursor	Визначає вигляд курсору миші для "епізоду"	setCursor, getCursor
Fill	Фон "епізоду"	setFill, getFill
Root	Кореневий вузол графу вузлів "епізоду"	setRoot, getRoot
x, y	Координати лівого верхнього угла "епізоду"	setX, getX, setY, getY
onMouseClicked	Визначає функцію, яка викликається, якщо користувач натисне кнопку миші	setOnMouseClicked

Для застосунку, графічний інтерфейс якого мовою FXML наведено в п. 10.3, головний клас має такий вигляд:

```
import java.io.IOException;
import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class MainClass extends Application {

        public void start(Stage primaryStage) throws IOException {
            Parent root =
          FXMLLoader.load(getClass().getResource("FXMLDocument.fxml"));
            Scene sc = new Scene(root);
            primaryStage.setScene(sc);
            primaryStage.show();
        }

        public static void main(String[] args) {
            launch(args);
        }
}
```

Програма працює таким чином:

1) для запуску її на виконання викликається метод launch. Це призводить до виклику методу start;

2) у методі start за допомогою класу FXMLLoader завантажується fxml-файл графічного інтерфейсу користувача;

3) далі створюється об'єкт класу Scene, який буде відтворювати епізод, що був завантажений з fxml-файлу;

4) наступний крок – це додавання епізоду до фрейму, що визначений посиланням primaryStage на об'єкт класу Stage. Цей об'єкт створюється автоматично під час запуску програми;

5) останнім етапом є відображення вікна програми за допомогою виклика методу show. Вікно має вигляд, наведений на рис. 11.1.

Зауваження. Натискання на кнопки "Повідомлення" або "Очистити" не дає жодного видимого результату. Щоб у відповідь з'являлася якась реакція, для цих кнопок необхідно додати обробники події "Натискання".

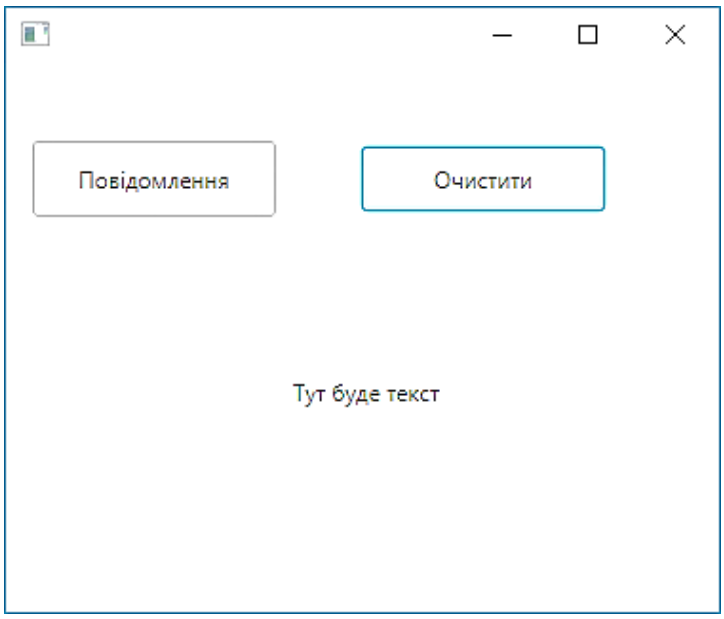

Рис. 11.1. **Головне вікно програми**

У головному вікні є три елемента управління: дві кнопки та текстова мітка. Вони є об'єктами класів javafx.scene.control.Button та javafx.scene.control.Label, відповідно. Ці класи, як і всі інші класи візуальних елементів управління, є похідними від *базового класу Control.*

Елементи управління JavaFX є спеціалізованими вузлами графу вузлів, які придатні для повторного використання в різних програмах. Їх візуальне подання можуть налаштовувати як дизайнери, так і програмісти. **Базові елементи управління** – це кнопки (клас Button), текстові мітки (клас Label) і текстові поля (клас TextField).

Оскільки елементи управління є вузлами графу, вони можуть вільно змішуватися з іншими типами вузлів: зображеннями, медіа, геометричними фігурами тощо.

У застосунках JavaFX також часто використовують класи Shape, Parent, ImageView, MediaView.

Клас Shape є базовим, що визначає загальні властивості класів – геометричних фігур Circle (коло), Line (пряма), PolyLine (ламана), Polygon (багатокутник) та інших.

Клас ImageView призначений для відтворення зображень, що завантажуються за допомогою класу Image.

Клас MediaView відображає Media, що відтворюється за допомогою MediaPlayer.

Клас Parent – це базовий клас для вузлів, які можуть мати підпорядковані вузли. Це вузли Control, Group, Region та інші.

Модель обробки подій в JavaFX:

1) **подія** – це безпосередня дія користувача або зміна стану будь-якого компонента інтерфейсу;

2) кожна подія – об'єкт деякого класу, що містить інформацію про тип події, його джерело та споживачів;

3) щоб отримувати в програмі повідомлення про деяку подію в компоненті, необхідно зареєструвати в ньому **обробник** цієї події – метод або деяку ділянку коду.

У JavaFX існує багато класів подій. Базовим для всіх подій є клас javafx.event.Event.

Вікно JavaFX-програми може реагувати на події від миші, клавіатури, маніпуляції з фреймом тощо. Вони є об'єктами класів javafx.stage. WindowEvent, javafx.scene.input.MouseEvent і javafx.scene.input.KeyEvent, відповідно.

Подія типу WindowEvent виникає, якщо вікно відображується або приховується.

Подія типу MouseEvent пов'язана з використанням миші (натискання кнопок, переміщення миші тощо).

Подія типу KeyEvent виникає з натисканням чи відпусканням клавіші на клавіатурі комп'ютера.

Якщо графічний інтерфейс користувача створений у fxml-файлі, для того щоб програма реагувала на деяку подію, необхідно створити клас контролера та зареєструвати його та метод – обробник події в цьому файлі.

Контролер – це звичайний Java-клас, деякі методи якого є обробниками подій, а **поля** здебільшого призначені для подання елементів графічного інтерфейсу в коді Java.

Приклад 1. Контролер, що обробляє подію типу MouseEvent, для вікна застосунку JavaFX, у якому немає елементів управління, може виглядати таким чином:

```
import javafx.fxml.FXML;
import javafx.scene.input.MouseButton;
import javafx.scene.input.MouseEvent;

public class SampleController {
        @FXML
        public void handleMouse(MouseEvent e) {
                if (e.getButton() == MouseButton.PRIMARY)
                        System.out.println("Привіт!");
        }
}
```

Тут handleMouse – обробник події MouseEvent, призначений для реагування на натискання лівої кнопки миші (MouseButton.PRIMARY). Анотація @FXML показує, що цей обробник викликається у відповідь на дії користувача застосунку.

FXML-файл має вигляд:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.BorderPane?>
<BorderPane onMouseClicked="#handleMouse"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8.0.141"
fx:controller="SampleController">
</BorderPane>
```

У ньому BorderPane – кореневий вузол графічного інтерфейсу користувача за замовчуванням;

fx:controller="SampleController" – визначає, що клас SampleController є контролером;

onMouseClicked="#handleMouse" – показує, що клацанням кнопкою миші буде викликатися метод handleMouse класу SampleController.

Діалогові вікна

Діалогові вікна мають таке призначення:

1) використовуються для збирання відомостей, що вводяться користувачами;

2) можна створювати свої власні діалогові вікна або використовувати стандартні;

3) модальне діалогове вікно має бути закрите перед продовженням роботи з іншими формами;

4) немодальні діалогові вікна дозволяють переходити між формами без необхідності закривати їх;

5) діалогові вікна, в яких відображаються важливі повідомлення, завжди повинні бути модальними.

Стандартні діалогові вікна JavaFX можуть бути об'єктами класів; javafx.scene.control.Alert; класу javafx.scene.control.ChoiceDialog; класу javafx.scene.control.TextInputDialog. Це підкласи класу Dialog.

Клас Alert надає підтримку для деяких заздалегідь визначених типів стандартних діалогових вікон, щоб нагадати користувачу про помилку, попередження тощо. Тому клас Alert часто є найкращим рішенням.

Якщо користувачеві потрібно нагадати про введення тексту або вибір зі списку варіантів, краще використовувати класи TextInputDialog і ChoiceDialog.

Деякі стандартні вікна повідомлень JavaFX на базі класу javafx.scene.control.Alert

Типи розглядуваних вікон розрізняють за характером виконуваних ними функцій.

1. Інформаційне вікно (рис. 11.2).

Java-код:

```
Alert mB = new Alert(AlertType.INFORMATION); mB.setTitle("Information Dialog");
        mB.setHeaderText("Look, an Information Dialog");
        mB.setContentText("I have a great message for you!");
        mB.showAndWait();
```

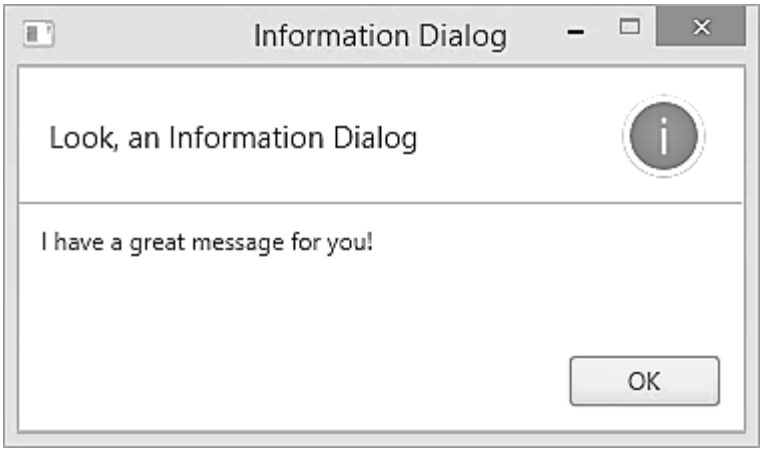

Рис. 11.2. **Зовнішній вигляд інформаційного вікна**

2. Вікно попередження (рис. 11.3).

Java-код:

```
Alert mB = new Alert(AlertType.WARNING);
mB.setTitle("Warning Dialog");
        mB.setHeaderText("Look, an Warning Dialog");
        mB.setContentText("Careful with the next step!");
        mB.showAndWait();
```

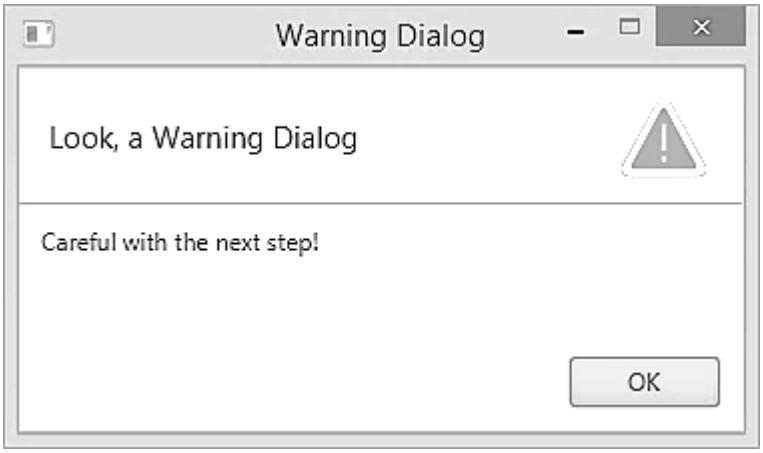

Рис. 11.3. **Зовнішній вигляд вікна попередження**

3. Вікно оповіщення про помилку (рис. 11.4).
Java-код:

```
Alert mB = new Alert(AlertType.ERROR);
mB.setTitle("Error Dialog");
mB.setHeaderText("Look, an Error Dialog");
mB.setContentText("Ooops, there was an error!");
mB.showAndWait();
```

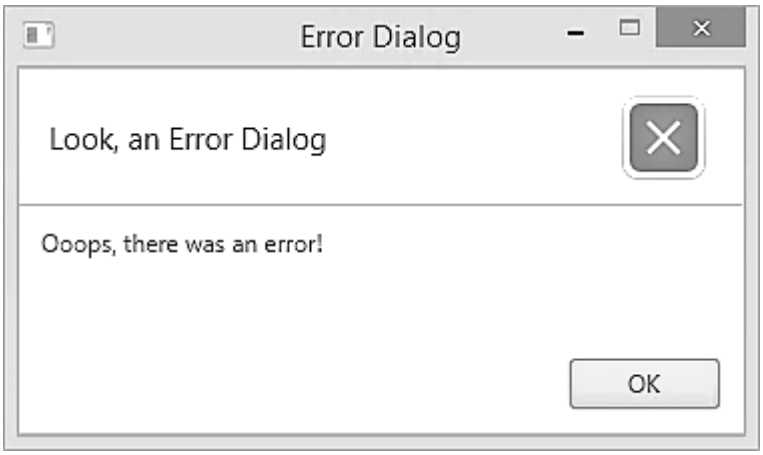

Рис. 11.4. **Зовнішній вигляд вікна оповіщення про помилку**

4. Вікно підтвердження (рис.11.5).
Java-код:

```
Alert mB = new Alert(AlertType.CONFIRMATION);
mB.setTitle("Confirmation Dialog");
mB.setHeaderText("Look, a Confirmation Dialog");
mB.setContentText("Are you ok with this?");
```

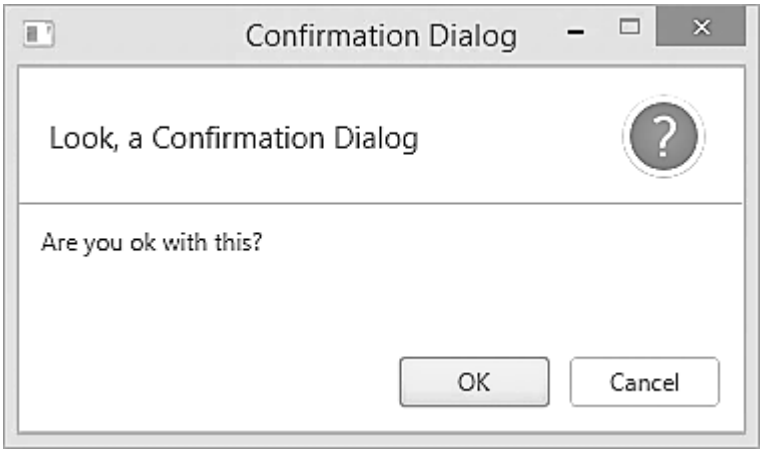

Рис. 11.5. **Зовнішній вигляд вікна підтвердження**

Далі необхідно додати код, що буде виконуватися натисканням кнопки "OK" чи "Cancel". Він може бути таким:

```
Optional<ButtonType> result = mB.showAndWait();
 if (result.get() == ButtonType.OK){
// Код 1}
else { // Код 2 }
```

Посилання result типу Optional зберігає вибір користувача.

Якщо користувач натиснув кнопку "OK", виконується Код 1. Інакше (натиснута кнопка "Cancel") виконується код 1.

Для створення власного діалогового вікна в проекті JavaFX необхідно:

1) додати у проект FXML-файл, де буде зберігатися "дизайн" діалогового вікна, додати потрібні елементи управління в SceneBuilder і налаштувати властивості цього вікна;

2) додати код у клас контролера діалогового вікна;

3) написати код для завантаження діалогового вікна з FXML-файла і його відображення.

Розміщення компонентів у контейнерах

Традиційне рішення – абсолютне позиціонування компонентів. Воно виконується в деякій системі координат, наприклад декартовій.

Сучасне рішення – відносне позиціонування компонентів (Swing, JavaFX, WPF, Windows 10 тощо). У JavaFX для цього застосовують панелі компонування.

Кожна панель, похідна від класу javafx.scene.layout.Pane, має **менеджер компонування**, що гарантує бажаний розмір кожного компонента,

та певне відносне розташування компонентів. Панель компонування викликає менеджер компонування з першою появою на екрані та з кожною зміною її розмірів. **Компонування** описує процес вимірювання і розстановки членів колекції children елемента Pane і їх подальшого відображення на екрані.

Основні стандартні панелі компонування JavaFX

HBox (VBox) – дозволяє розташувати елементи управління горизонтально (вертикально) в один рядок (стовпчик). Ці панелі компонування забезпечують доцільний розмір елементів управління;

BorderPane – елементи управління будуть знаходитися в п'яти областях: Center (центр), Top (верх), Bottom (низ), Left (ліворуч), Right (праворуч). Верхня та нижня області мають кращий розмір на вертикалі та займають усю ширину вікна. Ліва та права області мають кращий розмір на горизонталі та займають усю висоту вікна. Центральна область займає весь вільний простір у центрі вікна;

StackPane – елементи управління розташовуються в стеку в порядку від заднього до переднього;

AnchorPane – дозволяє "прикріпляти" елементи управління до одної або декількох сторін фрейму вікна;

FlowPane – елементи управління розташовуються один за одним у вигляді потоку. Потік може бути горизонтальним або вертикальним. Компоненти, які не вміщуються в одному рядку або стовпчику, автоматично "перетікають" на іншій;

TilePane – елементи управління знаходяться в комірках таблиці. Усі комірки мають однаковий розмір;

GridPane – розташовує елементи управління в комірках таблиці. Елемент управління може займати одну чи декілька комірок на горизонталі або вертикалі.

11.2. Оброблення подій від миші, клавіатури й елементів управління. Використання основних елементів управління JavaFX

Елементи управління JavaFX, як і форми, можуть реагувати на різноманітні події. Зокрема, це можуть бути події від мишки. Хоча програма може їх обробляти, вони вважаються нізкорівневими та використовуються не дуже часто.

Зазвичай для елементів управління віддають перевагу так званим "семантичним" подіям. Така подія містить декілька подій від миші; напри-

клад: натискання та відпускання кнопки миші, розташування курсору миші над елементом управління тощо.

Прикладом семантичної події є ActionEvent, що визначає деяку дію з вузлом графу JavaFX, яким може бути деякий елемент управління. Для кнопки це означає клацання мишею.

Приклад 2. Код контролера для оброблення події ActionEvent кнопкою:

```
import javafx.fxml.FXML;
import javafx.event.ActionEvent;
import javafx.scene.control.Button;

public class AnotherSampleController {

@FXML
private Button btn;

@FXML
        public void handle(ActionEvent e) {
                btn.setText(e.toString());
}
}
```

У табл. 11.4 наведено перелік основних елементів управління JavaFX (пакет javafx.scene.control).

Таблиця 11.4

Елементи управління JavaFX

Назва елемента	Клас JavaFX	Примітки	Головна подія
1	2	3	4
Кнопка	Button	–	ActionEvent
Прапорець	CheckBox	–	ActionEvent
Список прапорців	ChoiceBox	–	–
Комбінований список	ComboBox	Вибір визначеного або введення власного варіанту	ActionEvent
Вибір кольору	ColorPicker	Вибір кольору зі стандартної палітри або визначення свого кольору	ActionEvent
Контекстне меню	ContextMenu	За замовчуванням з'являється після клацання правою кнопкою миші	ActionEvent

Продовження табл. 11.4

1	2	3	4
Вибір дати	DatePicker	Введення дати або її вибір із випадаючого календаря	ActionEvent
HTML-редактор	HTMLEditor	–	-
Гіперпосилання	HyperLink	–	ActionEvent
Текстова мітка	Label	–	–
Список	ListView	Вибір зі списку визначеного варіанта	–
Меню	Menu	–	ActionEvent, MenuValidationEvent
Поле вводу пароля	PasswordField	Символи, що вводяться, замінюються на ехо-символи	ActionEvent
Панель прогресу	ProgressBar	–	–
Індикатор прогресу	ProgressIndicator	–	–
Радіо-кнопка	RadioButton	–	ActionEvent
Роздільник	Separator	Використовується для візуального розділення елементів управління горизонтальною або вертикальною лінією	–
Полоса прокрутки	ScrollBar	–	–
Панель прокрутки	ScrollPane	–	–
Слайдер	Slider	–	–
Панель із роздільником	SplitPane	Має дві або більше дочірніх панелей з вертикальним або горизонтальним роздільником	–
Панель із вкладками	TabPane	–	–
Таблиця	TableWiew	–	–

Закінчення табл. 11.4

1	2	3	4
Текстова область	TextArea	–	–
Текстове поле	TextField	–	ActionEvent
Підказка	Tooltip	Відображується з наведенням курсора миші на елемент управління	–
Дерево	TreeView	–	–

Аналіз табл. 11.4 показує, що за допомогою додаткових елементів управління JavaFX зручно розробляти насичені графічні застосування.

Питання для самопідготовки

1. Використання "колекції" візуальних елементів управління формою.
2. Застосування обробників подій рівня форми.
3. Розміщення візуальних елементів управління на формі.
4. Використання SceneBuilder для розроблення графічних інтерфейсів користувача.

Запитання для самодіагностики

1. Назвіть основні елементи синтаксису мови FXML.
2. Обґрунтуйте використання Eclipse для розроблення застосувань JavaFX.
3. Обґрунтуйте використання SceneBuilder для розроблення графічного інтерфейсу користувача.

Література: [15; 18; 21].

Рекомендована література

1. Блинов И. Н. Java. Методы программирования : учеб.-метод. пособ. / И. Н. Блинов, В. С. Романчик. – Минск : Изд-во "Четыре четверти", 2013. – 768 с.

2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч, Р. Максимчук, М. Энгл и др. ; пер. с англ. – Москва : ИД "Вильямс", 2008. – 720 с.

3. Влиссидес Дж. Применение шаблонов проектирования. Дополнительные штрихи. / Дж. Влиссидес. – Москва : ИД "Вильямс", 2003. – 423 с.

4. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – Санкт-Петербург : Питер, 2007. – 366 с.

5. Гранд М. Шаблоны проектирования в JAVA. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи UML / М. Гранд. – Новое знание, 2004. – 560 с.

6. Кериевски Дж. Рефакторинг с использованием шаблонов (паттернов проектирования) / Дж. Кериевски – Москва : ИД "Вильямс", 2006. – 400 с.

7. Ларман К. Применение UML 2.0 и шаблонов проектирования / К. Ларман. – Москва : ИД "Вильямс", 2006. – 736 с.

8. Леоненков А. Самоучитель UML 2 / А. Леоненков. – Санкт-Петербург : BHV, 2007. – 576 с.

9. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0 / Э. Троелсен ; пер. с англ. – Москва : ИД "Вильямс", 2011. – 1392 с.

10. Фаулер М. Шаблоны корпоративных приложений / М. Фаулер. – Москва : ИД "Вильямс", 2009. – 544 с.

11. Шилдт Г. Java. Руководство для начинающих / Г. Шилдт ; пер. с англ. – Москва : ИД "Вильямс", 2012. – 624 с.

12. Эмблер С. В. Рефакторинг баз данных: эволюционное проектирование / С. В. Эмблер, П. Дж. Садаладж. – Москва : ИД "Вильямс", 2007. – 368 с.

13. Michaelis M. Essential C# 6.0 / M. Michaelis, E. Lippert. – Boston : Addison-Wesly, 2016. – 1004 p.

14. Pro JavaFX 8: A Definitive Guide to Building Desktop, Mobile and Embedded Java Clients / J. Vos, W. Gao, S. Chin [et al.]. – New York : Apress, 2014. – 588 p.

15. Sharan K. Learn JavaFX 8 / K. Sharan. – New York : Apress, 2015. – 1200 p.

16. Wu C. Th. An introduction to object-oriented programming with Java / C. Th. Wu. – New York : McGraw-Hill, 2010. – 987 p.

17. Парфьонов Ю. Е. Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни "Основи об'єктно-орієнтованого програмування" [Електронний ресурс] / Ю. Е. Парфьонов. – Режим доступу : http://www.ikt.hneu.edu.ua/course/view.php?id=879.

18. Парфьонов Ю. Е. Основи об'єктно-орієнтованого програмування : презентації лекцій [Електронний ресурс] / Ю. Е. Парфьонов. – Режим доступу : http://www.ikt.hneu.edu.ua/course/view.php?id=879.

19. Программирование на Java для детей, родителей, дедушек и бабушек [Электронный ресурс]. – Режим доступа : http://myflex.org /books/java4kids/JavaKid8x11_ru.pdf.46.

20. Уроки Java для начинающих [Электронный ресурс]. – Режим доступа : http://cybern.ru/category/java/begin-java.

21. Учебник по JavaFX 8 [Электронный ресурс]. – Режим доступа : http://code.makery.ch/library/javafx-8-tutorial/ru.

22. Шаблон проектирования [Электронный ресурс]. – Режим доступа : https://ru.wikipedia.org/wiki/Шаблон_проектирования.

23. Programming Tutorials and Source Code Examples [Electronic resource]. – Access mode : http://www.java2s.com.

24. The Java Tutorials [Electronic resource]. – Access mode : http://download.oracle.com/javase/tutorial.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

Щербаков Олександр Всеволодович
Парфьонов Юрій Едуардович
Федорченко Володимир Миколайович

ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Навчальний посібник

Самостійне електронне текстове мережеве видання

Відповідальний за видання *О. Г. Руденко*

Відповідальний редактор *М. М. Оленич*

Редактор *Н. І. Ганцевич*

Коректор *Н. І. Ганцевич*

План 2019 р. Поз. № 11-ЕНП. Обсяг 237 с.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру
ДК № 4853 від 20.02.2015 р.