МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Л. В. Іванов, А. А. Пашнєв

МОВА І ПЛАТФОРМА JAVA В ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЯХ

Навчальний посібник
з дисциплін «Основи програмування»,
«Основи програмування Java» та
«Об’єктно-орієнтоване програмування»
для студентів з галузі знань «Інформаційні технології»
усіх форм навчання

Під редакцією проф. Годлевського М. Д.

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»,
протокол № 1 від 28.01.22

Харків
НТУ “ХПІ”
2024

УДК 004.432.2 (075)
І 20

Рецензенти:
С. Г. Семенов, д-р техн. наук, професор ХНЕУ
Д. В. Гриньов, канд. техн. наук (комп. «Epam»)

Іванов Л. В.

І 20 Мова і платформа JAVA в інформаційних технологіях : навч. посіб. / Іванов Л. В., Пашнєв А. А. – Харків : НТУ «ХПІ», 2024. – 167 с.

ISBN 978-617-05-0437-1

Розглянуто основи застосування мови програмування Java і Java-платформи в сучасних комп'ютерних інформаційних технологіях. Розглянуто базовий синтаксис мови Java, засади об'єктно-орієнтованого програмування мовою Java, принципи роботи з колекціями та потоками введення і виведення. Викладені матеріали відображають різні рівні використання мови Java і Java-платформи, а саме використання базових засобів мови Java, роботу з масивами та рядками, створення класів, успадкування та поліморфізм, узагальнене програмування та колекції, роботу з винятками та файлами. Кожен з розділів має запитання для самоперевірки.

Для студентів з галузі знань «Інформаційні технології» усіх форм навчання.

Іл. 1. Табл. 11. Бібліогр. 11 назв

УДК 004.432.2(075)

ISBN 978-617-05-0437-1

ВСТУП

На сучасному етапі розвитку суспільства зростає потреба у фахівцях з галузі інформаційних та програмних технологій, підвищуються вимоги до їхньої кваліфікації. Майбутні фахівці повинні володіти фундаментальними знаннями засад комп'ютерних наук і програмування, а також навичками самостійного навчання, постійного вдосконалення свого професійного рівня. Одночасно зростає потреба у підручниках, які можуть забезпечити можливість як засвоєння основ, так і подальшого самостійного поглиблення знань і вмінь.

З метою підготовки професіональних фахівців, затребуваних на ринку високих технологій, при вивченні курсів ««Основи програмування», «Основи програмування Java» та «Об'єктно-орієнтоване програмування» для студентів з галузі знань «Інформаційні технології» пропонується цей навчальний посібник.

Посібник присвячено вивченню синтаксису та засвоєнню практичних навичок програмування мовою Java. Ця мова є досить поширеною у сучасному світі програмування. Мова є одночасно потужною і легкою для вивчення й розуміння, підтримує об'єктно-орієнтовану парадигму, а також узагальнене й функціональне програмування, одночасно пропонує імперативний і декларативний підходи.

З іншого боку, мова Java є основною під час створення застосунків для однойменної платформи. В свою чергу, можливості платформи Java охоплюють практично всі напрямки сучасної розробки програмних систем, які працюють під керівництвом різних операційних систем. Підтримується різна архітектура застосунків.

Мова Java також є однією з найбільш популярних для створення застосунків для платформи Android.

Навчальний посібник «Мова і платформа Java в інформаційних технологіях» складається з п'яти розділів.

У першому розділі розглядаються основні базові елементи мови програмування Java і особливості Java-платформи, наведені засоби мови Java, які дозволяють створювати програми в межах процедурної парадигми.

Другий розділ присвячений роботі з типами-посиланнями у Java на прикладі масивів і користувацьких класів. Крім того, розглянуто роботу з найбільш поширеними стандартними класами для роботи з числовими даними і рядками.

У третьому розділі акцентовано увагу на успадкуванні й поліморфізмі, крім того, наводяться відомості про використання функціонального підходу, зокрема, застосування лямбда-виразів.

Четвертий розділ присвячено засадам узагальненого програмування, а також роботі з контейнерними класами й інтерфейсами.

П’ятий розділ містить відомості про механізми генерації та обробки винятків, а також про технології роботи з потоками введення-виведення.

Кожне наступне питання, що розглядається, базується на попередніх знаннях, містить велику кількість прикладів, для яких наведено пояснення, контрольні запитання й додаткові практичні завдання.

1. ВИКОРИСТАННЯ БАЗОВИХ ЗАСОБІВ МОВИ JAVA

У розділі розглядаються основні базові елементи мови програмування Java і особливості Java-платформи. Наведені засоби мови Java, які дозволяють створювати програми в межах процедурної парадигми.

1.1. Мова програмування Java і Java-платформа

1.1.1. Загальні концепції

Під Java розуміють:

– мову програмування;

– програмну платформу.

Робота над мовою Java почалася з 1990 році. Програміст компанії Sun Microsystems Патрік Ноутон (Patrick Naughton) сформулював проблеми адаптації програмного забезпечення до різних платформ та пристроїв, які існували в компанії. Патріка Ноутона було включено в автономну групу, якою керував Джеймс Гослінг (James Gosling). Першою назвою мови була Oak. Назва Java з'явилася в 1995 році.

Одночасно в світі виникла та стрімко розвивалася глобальна мережа Internet. У 1995 році компанія представила власний браузер HotJava з підтримкою Java-аплетів.

23 січня 1996 року вийшов офіційний реліз JDK 1.0.

У 1997 році Microsoft намагалася включити підтримку істотно зміненої Java у Visual Studio 97. Після судового розгляду у 2001 році Microsoft втратила право надалі використовувати торговельну марку Java.

Основними властивостями мови є такі:

– крос-платформна (синтаксис мови та набір стандартних засобів не залежить від певної операційної системи та типу комп'ютера);

– стандартизована;

– об'єктно-орієнтована;

– підтримує строгу типізацію (обмежена кількість примітивних типів, удосконалені правила перетворення типів);

– легка в засвоєнні та розробці;

– забезпечує підвищену надійність завдяки відсутності вказівників та обов'язковій перевірці віртуальною машиною коду класів, що завантажуються, і дій, які виконує програма;

– пристосована до розробки Інтернет-застосунків.

Для запобігання багатьох типових помилок, притаманних програмам на C++, запропоновано механізм автоматичного збирання сміття (garbage collection). Цей механізм автоматично підраховує кількість посилань на кожний об'єкт Java. Коли на об'єкт більше не вказує жодне посилання, він вважається непотрібним та може бути видалений з пам'яті, звільняючи ресурси для програми.

Синтаксис Java багато в чому схожий з C++. Разом з тим, Java має властивість крос-платформності (транспортабельності), застосовує інтерпретатор байт-коду, використовує іншу об'єктну модель (аналогічну C#), тобто іншу сукупність об'єктно-орієнтованих концепцій і набору засобів опису класів і зв'язків між ними, а також механізмів створення об'єктів, та ін.

На відміну від багатьох інших мов і систем програмування, сирцевий код Java компілюється не безпосередньо в машинні команди, а в так званий байт-код. Набір команд цього коду повинен бути інтерпретований для виконання на конкретному комп'ютері під керуванням конкретної операційної системи. Іншими словами, потрібна спеціальна програмна платформа.

Програмна платформа у загальному випадку включає набір програмних засобів, що забезпечують виконання інших програм. Програмні засоби, що складають програмну платформу, як правило, мають стандартний набір функцій, технічно по-різному реалізованих на різних апаратних платформах і в різних операційних системах. Java-платформу складають віртуальна машина Java і стандартні класи.

Віртуальна машина Java (Java Virtual Machine) – це програма, яка інтерпретує і виконує байт-код, попередньо створений з сирцевого тексту програми компілятором.

На сьогодні, крім мови Java, для розробки Java-застосунків можна використовувати такі мови, як C, Object Pascal, Python, Scala, Kotlin та багато інших. Кількість мов, орієнтованих на JVM дуже швидко розширюється. Продовжують створюватися нові мови з підтримкою JVM.

Концептуально платформа Java реалізована на різних рівнях, залежно від призначення програмних застосунків:

– Java Card – технологія, яка дозволяє безпечно запускати невеликі Java-програми (аплети) на смарт-картах та подібних пристроях малої пам’яті;

– Java ME (Micro Edition) – визначає кілька різних наборів бібліотек для пристроїв з обмеженими можливостями; її часто використовують для розробки додатків для мобільних пристроїв, КПК, телевізорів та принтерів;

– Java Platform, Standard Edition, (Java SE) – стандартна версія платформи Java, призначена для створення і виконання застосунків, розрахованих на індивідуальне користування;

– Java Platform, Enterprise Edition, скорочено Java EE – платформа, яка реалізує серверну архітектуру для задач середніх і великих підприємств.

Технічно платформа Java SE також реалізована на двох рівнях (до Java 8 включно):

– Java Runtime Environment (JRE) – середовище виконання Java-програм; фізично це набір програмних засобів, що включають віртуальну машину Java і бібліотеку класів;

– Java Development Kit (JDK) – комплект розробника застосунків мовою Java, що включає в себе компілятор Java (javac), стандартні бібліотеки класів Java, приклади, документацію, різні утиліти і середовище виконання Java-програм (JRE).

Кожна нова версія Java має нові можливості в порівнянні з попередньою.

Java версії 1.2 (4 грудня 1998 р.) настільки перевершувала платформу 1.1, що, починаючи з неї, усі наступні версії назвали платформою Java 2.

Починаючи з JDK 1.5 (29 вересня 2004 р.) прийнято говорити про Java 5. У цій версії істотно розширено синтаксис і засоби стандартних бібліотек.

Версія Java 6 (JDK 1.6, 11 грудня 2006 р.) відрізняється від попередньої переважно розширеними бібліотеками класів.

Версія Java 7 (28 липня 2011 р.), окрім розширених бібліотек класів, надає нові зручні синтаксичні конструкції.

Java 8 (18 березня 2014 р.) завдяки розширенням можливостям інтерфейсів, наявності лямбда-виразів, а також появі Stream API надає можливість реалізації функційного та декларативного програмування. Це істотно вплинуло на стиль і виразність Java-програмування.

Починаючи з 2017 року, версії Java доповнюються кожні півроку.

На жаль, у версіях, починаючи з дев'ятої, не повністю реалізовано принцип зворотної сумісності. Код, створений у попередніх версіях Java, не може бути виконаний без додаткової переробки та перекомпіляції. Через ці проблеми Java 8 залишається поширеною версією серед розроблювачів і далі підтримується компанією Oracle. Подальші приклади в тексті орієнтовані на використання Java 21. Як і Java 8, Java 21 є версією, для якої компанія Oracle здійснює подальшу підтримку.

1.1.2. Інсталяція Java 8

Для того щоб застосунок Java 8 можна було виконати на конкретному комп'ютері, на ньому необхідно встановити JRE або JDK. Встановлення JRE достатнє в більшості випадків, якщо мова йде не про розробку, а тільки про запуск застосунків.

Якщо робота здійснюється під управлінням операційної системи Windows, з сайту java.com можна завантажити програму-інсталятор. З головної сторінки цього сайту слід перейти за посиланням `Download (Завантажити)`. На першій сторінці слід погодитися з умовами ліцензійної угоди, натиснувши відповідну кнопку, після чого починається завантаження програми встановлення

Java для автоматично визначеної операційної системи. У багатьох випадках варіант інсталяції слід вибирати вручну. Для цього використовується посилання `See all Java downloads`.

Перед встановленням JRE доцільно закрити браузери та інші програми, які можуть використовувати Java. Якщо на комп'ютері раніше були встановлені старі версії JRE, інсталятор запропонує їх видалити. Попередні версії можна вибрати зі списку. Зазвичай після інсталяції в браузері відкривається сторінка, яка дозволяє перевірити версію Java.

Після завершення можна перевірити успішність встановлення, набравши в командному рядку команду `java-version`. У консольному вікні буде виведена версія JRE.

Починаючи з версії Java 9, Oracle не надає інсталяції JRE. Замість цього, зі сторінки `Java SE Downloads` можна завантажити JDK різних версій.

На початку засвоєння мови Java і Java-платформи окрема інсталяція JDK може виявитися зайвою, оскільки для роботи деяких інтегрованих середовищ, наприклад Eclipse, достатньо засобів Java, вбудованих в інсталяцію IDE.

1.1.3. Інтегровані середовища розробки для Java-програмування

У перші роки існування Java програмісти користувалися засобами компіляції, які завантажуються з командного рядку. Разом з тим, для інших мов програмування існували потужні середовища розробки, такі як Visual C++, Borland Delphi тощо. Візуальні засоби середовищ для автоматизації створення та налагодження програм істотно підвищували продуктивність праці програмістів. Загалом існують десятки комерційних і безкоштовних IDE для Java. Нижче наведені найбільш популярні середовища, а також ті, які історично найбільше вплинули на розвиток Java-розробки.

Історично першим було Visual Café for Java (1996 р.) – середовище, яке поширювалось на комерційних засадах. Проєкт проіснував до 2002 року.

Випуск комерційного середовища Microsoft Visual Studio 97 (1997 р.) містив підтримку Visual J++ – "Java від Microsoft". Потім як мова NET-розробки підтримувалася мова Visual J# (до 2007 р.).

JBuilder – комерційне середовище, створене у 1997 році компанією Borland. Остання версія CodeGear JBuilder 2008 вийшла у 2009 році.

Oracle JDeveloper – безкоштовне середовище розробки (з 2005 р.), перша версія IDE 1998 року була комерційною. Середовище в першу чергу орієнтовано на використання технологій Oracle. Остання стабільна версія вийшла у 2016 році.

BlueJ – інтегроване середовище розробки Java із відкритим кодом, яке застосовується з метою засвоєння програмування початківцями, а також в невеличких проєктах.

NetBeans – безкоштовне середовище розробки декількома мовами програмування (в першу чергу Java). Проєкт NetBeans IDE підтримується і спонсорується компанією Oracle. NetBeans успішно конкурує з найбільш розповсюдженими середовищами (Eclipse і IntelliJ IDEA), але поступається у гнучкості й продуктивності.

Eclipse IDE – найбільш популярне в світі крос-платформне безкоштовне середовище розробки. Спочатку Eclipse розроблялося фірмою IBM як наступник середовища розробки IBM VisualAge. Eclipse характеризується гнучкістю й можливістю нарощування функціональності через механізм плагінів.

JetBrains IntelliJ IDEA – найбільш потужне середовище професійної розробки Java-застосунків. IDE розповсюджується у двох варіантах. Безкоштовна версія Community Edition дозволяє створювати застосунки Java SE, повна версія Ultimate Edition, яка дозволяє, зокрема, розробляти серверні рішення, є комерційною.

1.2. Інтегроване середовище розробки IntelliJ IDEA

1.2.1. Встановлення IDE IntelliJ IDEA і створення першого проєкту

Інтегроване середовище розробки IntelliJ IDEA – застосунок, повністю написаний на Java. Для роботи IDE Java повинна бути заздалегідь встановлена

на комп’ютері. Для компіляції класів IntelliJ IDEA необхідно встановити JDK.

Існує два варіанти IDE, які підтримують різні підмножини технологій Java. Безкоштовний варіант Community Edition забезпечує повній набір засобів розробки для платформи Java SE мовами Java, Kotlin, Groovy та Scala, засоби управління проєктом, роботи з репозиторієм, тестування та налагодження програм. Варіант Ultimate Edition підтримує повний набір технологій, додатково включаючи Java EE, Spring Framework, спеціальні засоби роботи з базами даних, підтримку розробки на JavaScript і TypeScript тощо.

Програму встановлення середовища програмування IntelliJ IDEA можна скачати зі сторінки завантажень сайту компанії JetBrains. Обираємо варіант `Community Edition`. Після завантаження інсталятора його потрібно запустити на виконання. Натискаючи кнопку `Next`, проходимо по сторінках майстра встановлення:

– на сторінці `Choose Install Location` вибираємо теку для встановлення (можна залишити без змін);

– на сторінці `Installation Options` опції також можна залишити без змін;

– на сторінці `Choose Start Menu Folder` вибираємо групу в меню `Пуск` і натискаємо кнопку `Install`;

– після встановлення IntelliJ IDEA на останній сторінці майстра можна відразу вибрати опцію `Run IntelliJ IDEA Community Edition` і натиснути `Finish`.

Після першого запуску середовища, після погодження з умовами ліцензії нам пропонують вікно, в якому можна вибрати варіант імпорту установок попередньої версії. Якщо IntelliJ IDEA встановлюється вперше, ніякі опції імпортувати не потрібно.

У правій частині вікна `Welcome to IntelliJ IDEA` можна знайти чотири позиції меню: `Projects`, `Customize`, `Plugins` та `Learn`. Зокрема, позиція `Customize` дозволяє здійснити загальні налаштування стилів середовища:

кольорова тема (Color theme) – Dracula, High Contrast і IntelliJ Light. Тема вибирається на свій розсуд. Можна також змінити розмір шрифту.

У вікні Welcome to IntelliJ IDEA усталено обрано позицію Projects. З цієї позиції логічно починати під час першого запуску. Вибираємо створення нового проєкту (кнопка New Project). На сторінці майстра для нового проєкту вибрані найбільш розповсюджені опції, зокрема, позиція Java. Середовище намагається знайти встановлені JDK. Якщо це не вдалося, можна вибрати функцію Add JDK... і вказати шлях до раніше встановленого JDK.

Вибір опції Add sample code дозволить автоматично отримати клас з функцією main().

Якщо розкрити додаткові опції (Advanced Settings), можна уточнити назву модуля (Module name), корінь проєкту (Content root), а також теку розташування модуля (Module file location). Далі натискаємо кнопку Create. Відкривається головне вікно IDE з областю редагування.

На закладці Main.java можна побачити такий код.

Приклад

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

У тілі функції main() замість привітання "Hello, World". можна розмістити необхідній програмний код.

Для запуску програми на виконання можна скористатися функцією головного меню Run | Run 'Main', зеленою стрілкою в панелі інструментів або клавішною комбінацією Shift+F10. У разі успішної компіляції здійснюється виконання програми.

У нижній частині головного вікна з’являється спеціальна область, що імітує роботу в командному вікні. Саме там можна побачити привітання `"Hello, World"`.

1.2.2. Елементи графічного інтерфейсу користувача IDE IntelliJ IDEA. Використання шаблонів коду та гарячих клавіш

Головне вікно IDE включає як традиційні для всіх сучасних інтегрованих середовищ елементи користувацького інтерфейсу (меню, панель інструментів, головне вікно редактора, рядок стану), так і специфічний для середовища IntelliJ IDEA набір інструментальних вікон (`Tool Windows`). Інструментальні вікна мають кнопки виклику, розташовані по периметру робочої області, з піктограмою, підписом і числовим позначенням (останнє не обов’язково). Якщо на ці кнопки натиснути, поруч з ними відкриються віконця з деякою допоміжною функціональністю.

У стандартному головному меню наочно видно відсутність функції збереження. Автоматичне збереження відбувається під час запуску програми або під час виходу з програми.

Функція головного меню `File | Save All (Ctrl-S)` дозволяє зберегти всі відкриті файли одночасно.

Підменю головного меню пропонують потужні засоби управління кодом (`Navigate`, `Code`, `Analyze` і `Refactor`). У таблиці 1.1 наведені деякі функції для роботи з кодом та відповідні гарячі клавіші.

Таблиця 1.1

Функція меню	Опис	Клавіатурна комбінація
`Navigate | Back`	Перейти до попереднього місця перегляду або редагування	`Ctrl+Alt+Left`
`Navigate | Forward`	Перейти до наступного місця перегляду або редагування	`Ctrl+Alt+Right`
`Navigate | Previous Highlighted Error`	Перейти до попередньої знайденої помилки в коді	`Shift+F2`

Продовження таблиці 1.1

Функція меню	Опис	Клавіатурна комбінація
`Navigate | Next Highlighted Error`	Перейти до наступної знайденої помилки в коді	`F2`
`Code | Override methods`	Перевизначити метод базового класу	`Ctrl+O`
`Code | Implement methods`	Реалізувати метод абстрактного класу або інтерфейсу	`Ctrl+I`
`Code | Generate...`	Згенерувати фрагмент коду (конструктор, геттери і сеттери, перевизначені методи тощо)	`Alt+Insert`

Редактор IntelliJ IDEA підтримує велику кількість клавішних комбінацій для редагування. Крім стандартної роботи з буфером, підтримується, наприклад, видалення рядка (`Ctrl-Y`), дублювання рядка або блоку (`Ctrl-D`), перехід до точок переривання (`Ctrl+номер_точки_переривання`), коментування блоку (`Ctrl+/`) тощо.

Комбінація `Ctrl-пропуск` дозволяє отримати список можливих елементів об’єкта, параметрів методу тощо. Клавішна комбінація `Ctrl-Shift-пропуск` фільтрує список, залишивши тільки варіанти очікуваного типу.

Використовуючи комбінацію `Ctrl+Alt+L`, можна відформатувати код.

Корисна клавішна комбінація `Alt-Enter` дозволяє отримати набір варіантів виправлення помилки, яка виникла (наприклад, додати директиву `import`, згенерувати порожній метод тощо).

Перелік усіх клавішних комбінацій можна отримати у вікні `Settings`, далі `Keymap` (`File | Settings...`).

У рядку, розташованому безпосередньо після рядка меню, показаний шлях до файлу, який ми редагуємо (всередині проєкту), а також, найбільш вживані функції (вибір програми для запуску, кнопки запуску та налагодження, отримання структури проєкту та пошуку).

Середовище IntelliJ IDEA істотно полегшує введення сирцевого коду завдяки використанню шаблонів коду (Live Templates). Викликати відповідну

функцію можна через головне меню (`Code | Insert Live Template...`) або за допомогою клавішної комбінації `Ctrl-J`. З'являється список шаблонів коду. Список залежить від контексту (розташування курсору у вікні редактора). Найбільш корисні шаблони наведені у таблиці 1.2.

Таблиця 1.2

Послідовність символів	Програмний код
psvm	`public static void main(String[] args)`
St	`String`
psf	`public static final`
fori	`for (int i = 0; i < ; i++)`

itar	`for (int i = 0; i < args.length; i++) {`
	`  String arg = args[i];`
	
	`}`

ifn	`if ( == null) {`
	
	`}`

iter	`for (Object o : ) {`
	
	`}`

sout	`System.out.println();`

Шаблони можна також просто набирати в коді відповідної послідовністю літер. Після появи віконця підказки підтверджуємо введення і отримуємо відповідний текст у вікні редактора.

1.2.3. Використання налагоджувача

Запуск програми для налагодження може здійснюватися за допомогою функції головного меню `Run | Debug 'Main'`, кнопкою із зображенням зеленого жука в панелі інструментів або клавішною комбінацією `Shift+F9`. У найпростішому випадку для налагодження програми досить вказати точку переривання (кликнути мишею на вертикальній сірій смузі ліворуч від необхідного рядка) і запустити програму на налагодження. Виконання програми зупиниться на вибраному рядку, після чого проміжні значення змінних можна подивитися в області виведення `Variables` або просто

розмістивши курсор миші над змінною в програмному коді. У підменю `Run | Debugging Actions` під час налагодження доступні різні варіанти виконання програми по кроках:

– `Step into (F7)` – з заходженням у функції, що викликаються;

– `Step over (F8)` – без заходження у функції, що викликаються.

1.2.4. Структура проєкту

Для кожного проєкту створюється окремий каталог (з ім'ям проєкту, далі `Проєкт`). Файли кореневої теки проєкту містять інформацію про модулі. Проєкт може включати кілька окремих модулів. Інформація про конкретний модуль міститься у файлі з розширенням `.iml`.

Тека `src` містить пакети з сирцевим кодом проєкту. Тека `out` містить результат компіляції сирцевого коду. Всередині теки `out` знаходиться підкаталог `production`, в ньому – `Проєкт`, далі структура тек повторює аналогічну структуру тек підкаталогу `src`.

1.3. Базові засоби мови Java

1.3.1. Загальна структура програми мовою Java. Особливості виконання програми

У Java немає глобальних змінних, функцій чи процедур. Це зроблено з метою запобігання конфліктів імен. Програма складається з одного чи більше описів класів з полями (елементами даних) і методами (функціями-елементами). Класи в Java можуть мати модифікатор public. Такі класи мають назву відкритих (публічних). Один із відкритих класів повинен визначати статичний метод `main()`, з якого починається виконання програми. Програмний код визначення класів слід зберігати у текстових файлах з розширенням .java. Ім'я класу (без розширення) повинне збігатися з іменем публічного класу, який міститься у цьому файлі. Великі й маленькі літери відрізняються.

Кожен клас компілюється в окремий файл з розширенням `.class`, що містить двійковий код. Це не команди процесора, а байт-коди.

Скомпільована програма може бути виконана на комп'ютері, лише якщо на ньому встановлена так звана віртуальна машина Java (JVM, Java Virtual Machine) – спеціальна програма, яка здійснює інтерпретацію байт-кодів. Такий підхід дозволяє створювати програми, які можна переносити з одного операційного середовища (операційної системи, апаратної платформи) на інше без додаткової перекомпіляції. На відміну від C і C++, Java не передбачає статичного компонування коду у файл, який може бути виконано (`.exe`). Компонування (отримання байт-коду з різних файлів з розширенням `.class` або зі спеціальних архівів та інтерпретація коду) здійснюється динамічно.

Починаючи з Java 9, запропоновано модульну структуру коду. Спеціальний файл `module-info.java`, який можна створити в проєкті, містить таку інформацію:

– визначення модулів, від яких залежить цей модуль;

– інформацію про пакети, які експортує цей модуль;

– список сервісів, які надає модуль.

Модульна структура підвищує ефективність завантаження і виконання великих Java-застосунків.

1.3.2. Ідентифікатори, ключові та зарезервовані слова. Коментарі

Як і в більшості мов програмування, у Java відрізняються великі та маленькі літери.

Сирцевий код Java складається з лексем. Лексема (token) – це послідовність символів, що мають певне сукупне значення. Проміж окремими лексемами розташовують розділювачі – пропуск, табуляцію, новий рядок тощо.

Лексеми поділяються на такі групи:

– ключові (зарезервовані) слова;

– ідентифікатори;

– літерали (константи);

– знаки операцій.

Ключове слово – це лексема, яка має єдиний наперед визначений сенс в програмному коді. Усі ключові слова є зарезервованими. Зарезервовані слова

не можна використовувати як ідентифікатори. У Java також існують зарезервовані слова, які не є ключовими. Їх взагалі не можна використовувати. Це **`const`** та **`goto`**.

Починаючи з Java 10, у мові з'явилися контекстно-залежні ключові слова, такі як **`var`** або **`record`** (у Java 14); ці слова не є зарезервованими, але мають визначений сенс у певному контексті.

Ідентифікатори використовуються для іменування змінних, функцій та інших програмних об'єктів. Першим символом повинна бути літера чи символ підкреслення (`"_", underscore character`). Далі можуть використовуватися також цифри. Використання символу підкреслення на початку імені не є бажаним.

Доцільно використовувати змістовні імена, які відображають природу об'єкта або функції. Не можна використовувати пропуски всередині ідентифікатора. Тому, якщо необхідно створити ідентифікатор з кількох слів, ці слова пишуть злито, починаючи друге та інші слова з великої літери – так звана "верблюжа нотація". Наприклад, можна створити таке ім'я змінної: `thisIsMyVariable`.

Для змістовних імен доцільно використовувати англійську мнемоніку. Імена класів та інтерфейсів слід починати з великої літери, інші імена – тільки з маленької. Імена статичних констант складаються з великих літер та символу підкреслення, наприклад: `PI`, `CONST_VALUE` тощо.

Коментарі – це текст всередині сирцевого коду, який не обробляє компілятор. Мова Java підтримує три види коментарів:

– у стилі C++ (`//` до кінця рядка);

– у стилі C (`/* */`);

– коментарі Javadoc (`/** */`).

Спеціальний засіб `javadoc.exe`, що входить у JDK (Java Development Kit), використовує коментарі третього виду для автоматичної генерації документації. Для цього такі коментарі повинні бути відформатовані за стандартом Javadoc. Коментарі Javadoc повинні бути розташовані перед

відповідними фрагментами коду. Кожен коментар складається з опису і тегів. У коментарі javadoc можна включати теги форматування HTML. Не рекомендується включати теги, що забезпечують виведення жирним шрифтом, курсивом тощо. Є спеціальний набір тегів Javadoc, таких як `@author`, `@param`, `@return` тощо. Ці теги забезпечують стандартне представлення документації, що стосується класів і методів. Наприклад, такі коментарі перед методом

```
/**
* Повертає ім'я файлу з відповідним індексом.<br>
* Імена файлів зберігаються в масиві рядків.
* @param i індекс
* @return ім'я файлу.
* */
public String getFileName(int i) {
    return names[i];
}
```

забезпечують генерацію фрагмента HTML-файлу, наведеного на рис. 1.1.

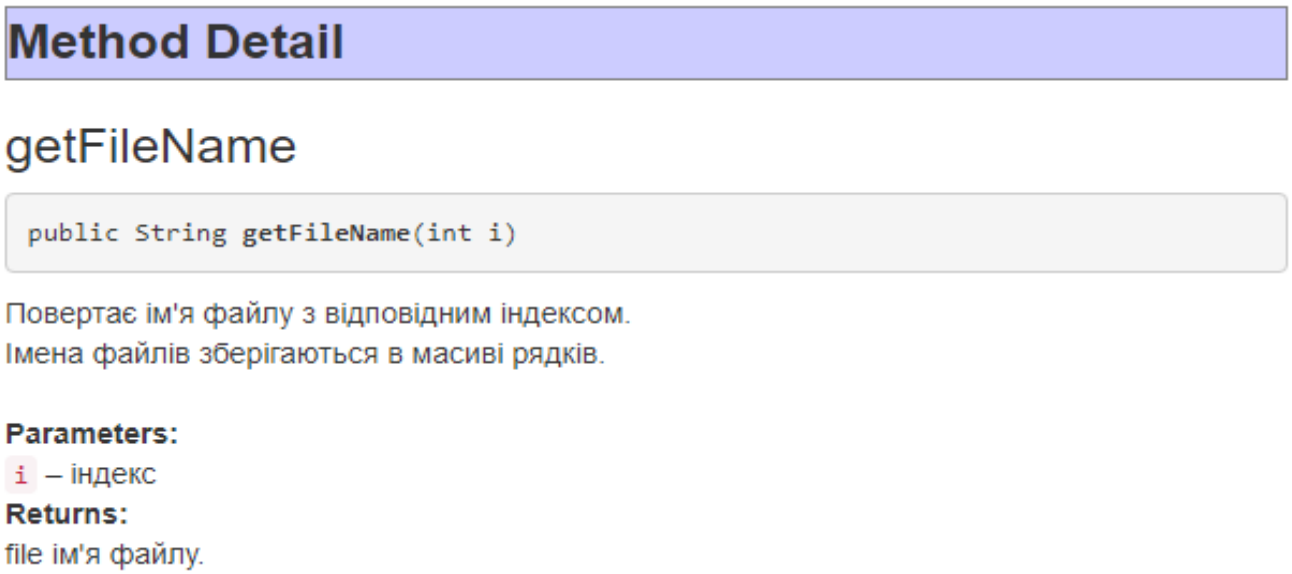

Рисунок 1.1

1.3.3. Визначення локальних змінних. Примітивні типи

Локальні змінні визначаються (створюються) всередині методів. Опис локальних змінних у Java здійснюється аналогічно C++.

Приклад

```
int i = 11;
double d = 0, x;
```

```
float f;
int j, k;
```

Локальні змінні можуть бути визначені в будь-якому місці всередині тіла функції, а також у вкладеному блоці. У Java не можна у внутрішньому блоці визначати імена, вже описані в зовнішньому блоці.

Приклад

```
{
    int i = 0;
    {
        int j = 1; // Змінна j визначена у внутрішньому блоці
        int i = 2; // Помилка! Змінна i вже визначена
    }
}
```

Ключове слово **final** стосовно до імен змінних означає, що вони не можуть бути змінені.

Приклад

```
final int h = 0;
```

Слово **const** зарезервоване, але не використовується.

Примітивні, чи базові, типи поділяються на цілі типи, типи з крапкою, що плаває, символьні і булеві. Примітивні типи приведені в таблиці 1.3.

Таблиця 1.3

Ключове слово	Опис	Розміри
Цілі типи		
byte	маленьке ціле (від –128 до 127)	8 біт
short	коротке ціле (від –32768 до 32767)	16 біт
int	ціле (від –2147483648 до 2147483647)	32 біта
long	довге ціле (від –9223372036854775808 до 9223372036854775807)	64 біта
Типи з плаваючою крапкою		
float	дійсне число звичайної точності	32 біта
double	дійсне число подвійної точності	64 біта

Продовження таблиці 1.3

Ключове слово	Опис	Розміри
Інші типи		
char	символ у кодах Unicode	16 біт
boolean	булеве значення (**true** або **false**)	

У багатьох інших мовах (наприклад, у C та у C++) формат і розміри примітивних типів даних залежать від платформи (DOS, Win 32 тощо). У Java розміри і формат стандартизовані і не залежать від платформи.

Константи цілого типу записуються як послідовності десяткових цифр. Усталений тип константи – **int**. Він може бути уточнений додаванням наприкінці константи літер `L` чи `l` (тип **long**). Цілі константи можуть записуватися у вісімковій системі числення, у цьому випадку першою цифрою повинна бути цифра 0, число може містити тільки цифри 0...7. Цілі константи можна записувати й у шістнадцятковій системі числення, у цьому випадку запис константи починається із символів `0x` чи `0X`. Для позначення цифр понад 9 використовуються латинські літери `a`, `b`, `c`, `d`, `e` та `f` (великі або маленькі).

Приклад

```
int octal = 023; // 19
int hex = 0xEF;  // 239
```

У Java 7 з'явилася можливість визначати також двійкові константи (з використанням префіксів 0B або 0b).

Приклад

```
int m = 0b110011; // 51
```

Крім того, групи розрядів у константах можна розділяти знаком підкреслення.

Приклад

```
int n = 1_048_576;
```

Константи типу **char** беруть в одиночні лапки (апострофи), значення константи задається або знаком з поточного набору символів, або цілою константою, якій передує зворотна коса риска (символ із заданим кодом). Є ряд спеціальних символів, що можуть використовуватись як значення константи типу **char** (такі подвійні символи називаються керуючими послідовностями):

```
'\n' – новий рядок,
'\t' – горизонтальна табуляція,
'\r' – переведення на початок рядку,
'\'' – одиночні лапки (апостроф),
'\"' – подвійні лапки,
'\\' – зворотна коса риска (backslash).
```

Для зберігання даних символьного типу в пам’яті використовується таблиця Unicode.

Константи дійсних типів можуть записуватись у формі з крапкою або в експонентному форматі та усталено мають тип **double**. При необхідності тип константи можна уточнити, записавши наприкінці суфікс `f` чи `F` для типу **float**, суфікс `d` чи `D` для типу **double**.

Приклад

```
1.5f    // 1.5   типу float
2.4E-2d // 0.024 типу double
```

Значення з плаваючою крапкою не можна присвоювати цілим змінним. Перетворення до більш вузького типу (дійсного до цілого, числа з подвійною точністю до числа з одинарною точністю) пов’язано з ризиком утрати даних і повинне здійснюватися явно.

Приклад

```
int   i  = 10;
float f  = i;         // Таке перетворення допускається
long  l  = f;         // Помилка!
long  l1 = (long) f; // Явне приведення типів
```

Числа без десяткової крапки інтерпретуються як цілі (типу int). Числа з плаваючою крапкою мають тип **double**. Для приведення їх до більш вузьких типів використовується явне приведення типів.

Приклад

```
float f  = 10.5;          // Помилка!
float f1 = (float) 10.5; // Явне приведення типів
float f2 = 10.5f;         // Уточнення типу. Помилки немає
```

У Java немає беззнакових (unsigned) цілих типів.

Булевим змінним можна присвоювати тільки константи **true** та **false**. Змінні типу boolean не можна неявно чи явно приводити до інших типів і навпаки.

Константа-рядок складається із символів, які беруть у подвійні лапки.

Приклад

```
"Це рядок"
```

Результат додавання рядка до змінної іншого типу забезпечує перетворення значення у представлення у вигляді рядка. Зокрема, такий підхід застосовують для виведення значень декількох змінних.

Приклад

```
int k = 1;
double d = 2.5;
System.out.println(k + " " + d); // 1 2.5
```

Починаючи з версії мови 10, Java надає можливість створення неявно типізованих локальних змінних. Для опису таких змінних застосовують контекстно-залежне ключове слово **var**. Такі змінні обов'язково повинні бути проініціалізовані. Тип змінної компілятор визначає відповідно до типу ініціалізуючого виразу.

Приклад

```
var i = 1;
var s = "Hello";
```

Незважаючи на те, що тип не вказано явно, компілятор створює змінну певного типу. Після того, як змінна створена, не можна змінювати її тип.

Приклад

```
var k = 1;
k = "Hello"; // Помилка!
```

У Java не можна оголошувати змінні без їхнього створення.

1.3.4. Вирази та операції

Всередині методів класів, з яких складається програма, можуть міститись твердження (оператори), які, в свою чергу, складаються з виразів. Вираз складається з однієї чи кількох операцій. Об'єкти операцій мають назву операндів. Операції бувають унарними (один операнд), бінарними (два операнди) і тернарними (три операнди).

До арифметичних операцій належать `+`, `-` (бінарні й унарні), `*`, `/` а також операція отримання залишку від ділення `%` (тільки до цілих). Якщо `/` застосовується до цілих, результатом ділення буде теж ціле, а залишок відкидається. Якщо хоча б один операнд – типу з плаваючою крапкою (дійсний), ми отримаємо дійсний результат.

Приклад

```
System.out.println(1 / 2);   // 0
System.out.println(1.0 / 2); // 0.5
```

До операцій відношення належать перевірка на рівність == і на нерівність !=, а також перевірки > (більше) >= (більше або дорівнює) < (менше) <= (менше або дорівнює). До логічних операцій належать логічне І (`&&`), АБО (`||`) та НІ (`!`). Операції відношення та логічні повертають значення типу boolean. Операції І та АБО можна застосовувати побітові & і |. У такому випадку завжди здійснюється повне обчислення значень обох операндів, тоді як обчислення значення другого операнду операцій && і || може не здійснюватися, якщо результат вже встановлено.

До операцій присвоювання належать операції простого і складеного присвоювання. Результатом операції простого присвоювання є значення того

виразу, що присвоюється лівому операнду. Складене присвоювання можна представити в загальному вигляді в такий спосіб: `a op= b`.

У цьому випадку `op` – арифметична чи побітова операція: `+`, `-`, `*`, `/`, `%`, `|`, `&`, `^`, `<<` та `>>`. Кожна складена операція еквівалентна присвоюванню: `a=(a)op(b)`. Наприклад,

```
x += 5;
```

що еквівалентно

```
x = x + 5.
```

Операція інкременту `++` забезпечує збільшення цілої змінної на одиницю. Вона має дві форми префіксну і постфіксну. Префіксна форма забезпечує збільшення змінної до того, як значення операції буде використано, а постфіксна – після. Операція декременту -- забезпечує зменшення змінної на одиницю і правила її використання аналогічні.

На відміну від C++, операція "кома" може бути застосована тільки в заголовках циклів.

Приклад

```
int i, j;
for (i = 0, j = 0; i < 10; i++, j += 2) {
    System.out.println(i + " " + j);
}
```

Java надає можливість використання так званих побітових операцій, які працюють з окремими бітами цілих чисел. Операнди повинні бути цілими числами.

Унарна операція побітового заперечення, або доповнення (~) встановлює нулі в одиниці, а одиниці в нулі.

Приклад

```
byte bits = 23;           // 0 0 0 1 0 1 1 1
bits = (byte)~bits;       // 1 1 1 0 1 0 0 0 або 232
System.out.println(bits); // -24
```

Останній результат виглядає дещо дивним. Справа в тому, що для представлення від'ємних чисел, використовують їх бінарне додавання до 2 у ступені кількості бітів, у цьому випадку – до 256.

```
232 - 256 = -24
```

Працюючи з побітовим запереченням та іншими аналогічними операціями, необхідно явне приведення до типу **byte**, оскільки побітові операції усталено повертають ціле значення.

Бінарні операції зсуву ліворуч (<<) та праворуч (>>) призначені для переміщення бітів першого операнду на кількість розрядів, визначених другим операндом. У той же час біти, які виходять за межі, відкидаються, а замість раніше відсутніх додаються нулі.

Приклад

```
byte bits = 1;                // 0 0 0 0 0 0 0 1
bits = (byte) (bits << 3);    // 0 0 0 0 1 0 0 0
System.out.println(bits);     // 8
bits = (byte) (bits >> 2);    // 0 0 0 0 0 0 1 0
System.out.println(bits);     // 2
```

Операція >> зберігає знак величини. Спеціальна операція зсуву >>> зсуває всі біти, в тому числі старший.

Приклад

```
byte bits = -24;           // 1 1 1 0 1 0 0 0
int i = bits >> 1;
System.out.println(i);     // -12
i = bits >>> 1;
System.out.println(i);     // 2147483636
```

Операція ТА (&, один амперсанд, на відміну від логічної операції), яка застосована до двох бітів, повертає 1, якщо обидва біти мають значення 1, або повертає 0, якщо один або обидва біти дорівнюють 0. Аналогічно, операція АБО (|, одна вертикальна риска, на відміну від логічного АБО), яка застосована до двох бітів, повертає 0, якщо обидва біти мають значення 0, або повертає 1, якщо один або обидва біти дорівнюють 1. Операція ВИКЛЮЧНЕ АБО (^) для

двох однакових бітів встановлює результат в одиницю, якщо біти різні та нуль, якщо вони однакові.

Приклад

```
byte result;
byte b1 = 101;              //  0 1 1 0 0 1 0 1
byte b2 = 47;               //  0 0 1 0 1 1 1 1
result = (byte) (b1 & b2);  //  0 0 1 0 0 1 0 1
System.out.println(result); //  37
result = (byte) (b1 | b2);  //  0 1 1 0 1 1 1 1
System.out.println(result); //  111
result = (byte) (b1 ^ b2);  //  0 1 0 0 1 0 1 0
System.out.println(result); //  74
```

Умовна операція (тернарна) має такий вигляд: `умова ? вираз1 : вираз2`.

Спочатку обчислюється значення `умови`. Якщо воно істинне, то обчислюється `вираз1` і його значення повертається умовною операцією. Якщо значення умови хибне, то обчислюється `вираз2` і повертається його значення. У наведеному нижче прикладі обчислюється мінімальне з двох чисел.

Приклад

```
min = a < b ? a : b;
```

Порядок застосування унарних операцій та операцій присвоєння "справа наліво", а всіх інших операцій – "зліва направо".

Пріоритет додавання і віднімання буде нижчим ніж множення і ділення, пріоритет присвоєння буде нижчим ніж арифметичні операції і т.д. Для зміни послідовності операцій слід використовувати дужки.

У Java не допускається безпосередня робота з вказівниками, отже, немає операцій розіменування та узяття адреси (`*` та `&`). Операція `->` замість вибору елемента за вказівником застосовується для створення лямбда-виразів.

У Java немає операції **`sizeof()`**, оскільки її головне призначення в C++ – з'ясовувати розміри тих чи інших даних під час перенесення сирцевого коду на іншу платформу. У Java розміри всіх типів стандартизовані.

1.3.5. Твердження (інструкції). Керування виконанням програми

Твердження (або інструкція, іноді оператор, англ. statement) – найменша автономна частина мови програмування. Програма являє собою послідовність інструкцій.

Порожня інструкція складається з однієї крапки з комою.

Інструкція-вираз є повний вираз, який закінчується крапкою з комою.

Приклад

```
k = i + j + 1;  // присвоєння
Arrays.sort(a); // виклик функції
return;         // вихід з функції
```

Складена інструкція – це послідовність тверджень, укладена у фігурні дужки. Складену інструкцію часто іменують блоком. Після фігурної дужки, яка закриває блок, крапка з комою не ставиться. Синтаксично блок може розглядатися як окрема інструкція, однак вона також має значення у визначенні видимості і часу життя ідентифікаторів. Ідентифікатор, оголошений всередині блоку, має область видимості від точки визначення до фігурної дужки, що закривається. Блоки можуть необмежено вкладатися один в одного.

Інструкції вибору – умовна інструкція та перемикач. Умовна інструкція застосовується у двох видах:

```
if (вираз-умова)
    інструкція1
else
    інструкція2
```

або

```
if (умова)
    інструкція1
```

Під час виконання цієї інструкції обчислюється вираз-умова і, якщо це істина, то виконується інструкція1 а інакше – інструкція2. На відміну від C++, вираз-умова може бути лише типу boolean. Тому наведений нижче код

```
int k;
// ... отримання k
```

```
if (k) { // перевірка, чи k не дорівнює нулю
}
```

призведе до синтаксичної помилки. Слід писати:

```
if (k != 0) {
}
```

Перемикач дозволяє вибрати одну з кількох можливих гілок обчислень і будується за схемою:

```
switch (цілий_вираз)
    блок
```

Блок має такий вигляд:

```
{
    case константа_1: інструкції
    case константа-2: інструкції
    ...
    default: інструкції
}
```

Виконання перемикача полягає в обчисленні керуючого виразу і переході до групи інструкцій, позначених **case**-міткою, значення якої дорівнює керуючому виразу. Якщо такої мітки немає, виконуються інструкції після мітки **default** (яка може бути відсутня). Під час виконання перемикача відбувається перехід на інструкції з обраною міткою, і далі інструкції виконуються у нормальному порядку. Для того щоб не виконувати інструкцій, які залишилися у тілі перемикача, необхідно використовувати оператор **break**.

Починаючи з Java 7, окрім цілих та символів, у конструкції **switch** () можна використовувати рядки. В цьому випадку константи варіантів (після **case**) повинні теж бути рядками.

Приклад

```
case "some text":
```

Інструкції циклу реалізовані в трьох варіантах: цикл із передумовою, цикл із постумовою і цикл із параметром. (існує четверта форма, яка стосується масивів та колекцій).

Цикл із передумовою будується за схемою:

```
while (вираз-умова)
    інструкція
```

На кожному повторенні циклу обчислюється вираз-умова, і якщо значення цього виразу дорівнює true, виконується інструкція – тіло циклу.

Цикл із постумовою будується за схемою:

```
do
    інструкція
while (вираз-умова);
```

Вираз-умова обчислюється і перевіряється після кожного повторення інструкції – тіла циклу; цикл повторюється, поки умова виконується. Тіло циклу в циклі з постумовою виконується принаймні один раз.

Цикл із параметром будується за схемою:

```
for (вираз1; вираз2; вираз3)
    інструкція
```

Тут `вираз1` і `вираз3` – вирази скалярного типу, `вираз2` – вираз типу **`boolean`**. Цикл з параметром реалізується за таким алгоритмом:

– обчислюється `вираз1` (зазвичай цей вираз виконує підготовку до початку циклу);

– обчислюється `вираз2`, і якщо він дорівнює **`false`**, виконується перехід до наступної інструкції програми (вихід з циклу);

– якщо `вираз2` дорівнює **`true`**, виконується інструкція – тіло циклу;

– обчислюється `вираз3` – виконується підготовка до повторення циклу, після чого знову виконується вираз `вираз2`.

Слід пам’ятати, що в усіх циклах перевірка умови продовження вимагає результату типу **`boolean`**.

У наведеному нижче прикладі сума `y = 1 + 2 + 3 + ... + n` знаходиться за допомогою трьох різних циклічних інструкцій.

За допомогою циклу **`while`**.

Приклад

```
int y = 0;
int i = 1;
while (i <= n) {
    y += i;
    i++;
}
```

За допомогою циклу **`do ... while`**.

Приклад

```
int y = 0;
int i = 1;
do {
    y += i;
    i++;
}
while (i <= n);
```

За допомогою циклу **`for`**.

Приклад

```
int y = 0;
for (int i = 1; i <= n; i++) {
    y += i;
}
```

Циклічні конструкції можна вкладати одну в іншу.

У сполученні з інструкціями циклу використовуються інструкції переходу – оператор **`break`**, який дозволяє перервати виконання найвнутрішнішого з циклів, оператор **`continue`**, який перериває поточну ітерацію найвнутрішнішого з циклів **`while`**, **`do`** або **`for`**. Найчастіше **`break`** використовують у такій конструкції:

```
if (умова_дострокового_завершення_циклу)
    break;
```

У Java після ключових слів **`break`** та **`continue`** можна розташувати мітку, що передує одному з вкладених циклів. У цьому випадку твердження стосуються не до найвнутрішнішого, а до позначеного циклу.

Приклад

```
  int a;
  . . .
  double b = 0;
label:
  for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
          if (i + j + a == 0) {
              break label;
          }
          b += 1.0 / (i + j + a);
      }
  }
```

Твердження **`goto`** не використовується в мові Java, але **`goto`** є зарезервованим словом. Це слово ніяк не може бути використано.

1.4. Пакети та функції

У Java немає заголовних файлів. Для того щоб запобігти можливим конфліктам імен, уся глобальна область видимості розділяється на пакети.

Пакет – це набір взаємозалежних класів та інших елементів коду, які здійснюють взаємодію один з одним через загальний доступ до функцій і даних і через загальний простір імен. Усі сирцеві файли одного пакету повинні міститися в одній теці. Пакети можуть вкладатися один в інший. Кожен клас Java знаходиться в будь-якому пакеті. Імена пакетів повинні збігатися з іменами тек, у яких знаходяться файли сирцевого тексту (чи скомпільовані файли).

У першому рядку сирцевого файлу (не враховуючи порожніх рядків і коментарів) найчастіше знаходиться заголовок пакету, до якого належать класи, визначені у файлі: **`package`** ім'я_пакету.

Якщо ім’я пакету пропущене, вважається, що код належить до безіменного пакета (усталеного пакета), межі якого визначаються поточною

текою. До класів та інших синтаксичних конструкцій такого пакета не можна звернутися ззовні. Такий варіант не є бажаним.

Бажано використовувати більш змістовні імена пакетів. Можна створювати вкладені пакети, відповідно створюючи вкладені теки. Крім того, бажано забезпечити унікальність імен пакетів. Для забезпечення унікальності можна використовувати обернене доменне ім'я. Наприклад, якщо доменне ім'я автора author.com, то слід використовувати пакет `com`, далі підпакет `author` і далі підпакети, відповідно до різних задач.

У Java немає глобальних функцій. Аналогом глобальної функції є статичний метод класу. Опис статичної функції у найпростішому випадку має таку структуру:

```
static тип_результату ім'я(список_формальних_параметрів) тіло
```

На відміну від C++, у Java не треба і не можна створювати прототипи функцій (оголошення функцій без визначення).

Параметри функції, що вказуються в списку у визначенні функції, називаються формальними. Параметри, що вказуються під час виклику функції, називаються фактичними параметрами, або аргументами. Під час виклику функції виділяється пам'ять під її формальні параметри, потім кожному формальному параметру присвоюється значення фактичного параметра.

Тіло функції являє собою складений оператор (блок). Визначення статичної функції, що обчислює суму двох цілих чисел, буде таким.

Приклад

```
static int sum(int a, int b) {
    int c = a + b;
    return c;
}
```

Можна взагалі обійтися без змінної `c`.

Приклад

```
static int sum(int a, int b) {
    return a + b;
```

```
}
```

Виклик функції може бути здійснений у виразі в описі або в тілі іншої функції. Під час виклику функції вказується її ім’я і список фактичних параметрів без зазначення їхніх типів.

Приклад

```
int x = 4;
int y = 5;
int z = sum(x, y);
int t = sum(1, 3);
```

Функція може бути без параметрів.

Приклад

```
int zero() {
    return 0;
}
```

Викликаючи таку функцію, також необхідно використовувати дужки.

Приклад

```
System.out.println(zero());
```

Інструкція **`return`** у тілі функції забезпечує завершення роботи функції. Значення виразу після **`return`** стає значенням функції, яке ця функція повертає.

Функція може не повертати ніякого результату. Для позначення цього використовується тип **`void`**.

Приклад

```
void hello() {
    System.out.println("Hello!");
}
```

У цьому випадку в тілі функції **`return`** може бути відсутнім. Якщо інструкція **`return`** присутня, то після неї не повинно бути ні якого виразу. Таку функцію можна викликати тільки окремою інструкцією `hello()`.

Параметри передаються до функцій за значенням, тобто значення фактичних параметрів копіюються в пам’ять, відведену для формальних

параметрів. При цьому значення, з якими працює функція – це її власні локальні копії фактичних параметрів, і їхня зміна на ці параметри не впливає. Під час передачі за значенням вміст фактичних параметрів не змінюється.

Приклад

```
static void f(int k) {
    k++;        // k = 2;
}

public static void main(String[] args) {
    int k = 1;
    f(k);
    System.out.println(k); // k = 1;
}
```

На відміну від багатьох інших мов програмування Java не надає механізму передачі параметрів типів-значень за посиланням.

У Java можна перевантажувати імена функцій. Якщо списки формальних параметрів двох функцій розрізняються числом чи їхніми типами параметрів, то ці дві функції вважаються перевантаженням однієї функції і вони повинні мати різні визначення.

Якщо списки формальних параметрів збігаються, але типи значень, що повертаються, різні, компілятор повідомляє про помилку.

Перевантажені функції використовуються в тих випадках, якщо кілька функцій виконує схожі дії над об’єктами різних типів і зручно дати однакові імена всім цим функціям.

Приклад

```
// Вибір максимального з двох цілих чисел:
static int max(int a, int b) { }
// Вибір максимального з трьох дійсних чисел:
static int max(double x, double y, double z) { }
```

Для того щоб визначити, яку саме функцію варто викликати, порівнюються кількість і типи фактичних параметрів, з кількістю і типами формальних параметрів всіх описів функцій з таким ім’ям. Викликається та

функція, у якої формальні параметри щонайкраще зіставилися з параметрами виклику, чи видається помилка, якщо такої функції не знайшлося.

Усталені параметри функції у Java не підтримуються.

Статичні функції в межах класу викликаються із застосуванням лише імені функції та списку фактичних параметрів. Для того щоб викликати статичну функцію іншого класу, необхідно вказувати його ім'я і далі через точку ім'я функції. В такий спосіб можна звертатись до імен у межах пакету, а також до класів пакету `java.lang`. Цей пакет містить класи з дуже корисними функціями. Наприклад, клас `Math` надає велику кількість математичних функцій. У таблиці 1.4 наведені деякі математичні функції класу `Math`.

Таблиця 1.4

Функція	Зміст	Приклад виклику
double pow(double a, double b)	Обчислення ab	Math.pow(x, y)
double sqrt(double a)	Обчислення квадратного кореня	Math.sqrt(x)
double sin(double a)	Обчислення синуса	Math.sin(x)
double cos(double a)	Обчислення косинуса	Math.cos(x)
double tan(double a)	Обчислення тангенса	Math.tan(x)
double asin(double a)	Обчислення арксинуса	Math.asin(x)
double acos(double a)	Обчислення арккосинуса	Math.acos(x)
double atan(double a)	Обчислення арктангенса	Math.atan(x)
double exp(double a)	Обчислення e^x	Math.exp(x)
double log(double a)	Обчислення натурального логарифма	Math.log(x)
double abs(**double** a) **int** abs(**int** a)	Знаходження модуля числа	Math.abs(x)
long round(**double** a)	Округлення цілого	Math.round(x)

Крім математичних функцій, клас `Math` надає такі корисні константи, як `Math.PI`, `Math.E`.

Якщо класи знаходяться в інших пакетах (не у `java.lang`), для виклику цих функцій слід або застосовувати префікс – `ім'я_пакету.ім'я_класу`. Якщо необхідні класи вкладених пакетів, префікс буде ще більш складним. Для того щоб запобігти використанню повних імен, застосовується конструкція **`import`**. Конструкції import розташовують на початку файлу з сирцевим кодом, або безпосередньо після конструкції **`package`**.

Для забезпечення доступу до класу чи інтерфейсу з іншого пакету є три варіанти:

– імпорт класу чи інтерфейсу (**`import`** `ім'я_пакету.ім'я_типу`);

– імпорт усього пакету (**`import`** `ім'я_пакету.*`);

– статичний імпорт – імпорт статичних елементів зазначеного класу (**`import static`** `ім'я_пакету.*`).

Наведений нижче приклад демонструє перші два варіанти.

Приклад

```
import java.io.FileReader;  // Імпорт класу (інтерфейсу)
import java.util.*;         // Імпорт усього пакету

public class TestClass {
    public static void main(String[] args) {
        java.io.FileWriter fw;  // Повне ім'я
        FileReader fr; // Доступ до імпортованого імені
        ArrayList al;  // ArrayList входить у пакет java.util
        . . .
    }
}
```

Не слід включати в програму твердження **`import`** `java.lang.*`. Цей пакет імпортується автоматично.

Додаткова форма імпорту, яка з'явилася у версії Java 5, дозволяє імпортувати тільки статичні елементи зазначеного класу. У програмах, які реалізують математичні обчислення, доцільно додавати статичний імпорт елементів класу `Math`.

Приклад

```
import static java.lang.Math.*;
```

Тепер статичні елементи можна використовувати без додаткового кваліфікатора.

Приклад

```
double d = sin(1);
```

1.5. Консольне введення та виведення

Для забезпечення консольного введення Java надає декілька варіантів:

– безпосереднє використання потоку введення System.in, зокрема його функції `read()`;

– використання класу `java.io.Console`;

– використання класу `java.util.Scanner`.

Перший варіант – низькорівневий і вимагає великої кількості "ручної роботи", тому зараз майже не застосовується. Другий варіант (клас `java.io.Console`, починаючи з JDK 1.6) теж має певні недоліки, зокрема, інтегровані середовища Eclipse і IntelliJ IDEA не підтримують роботу з об'єктами цих класів у своїх консольних вікнах.

Найбільш зручним засобом уведення даних є клас `java.util.Scanner`. Клас `Scanner` надає функції для читання даних з різних джерел, наприклад, з файлів. У нашому випадку ми вказуємо на необхідність читання з клавіатури (стандартний потік введення `System.in`). Об'єкт-сканер можна застосувати для читання даних різних типів. Наприклад, функція `next()` повертає наступне прочитане значення типу `String` (рядок), `nextInt()` і дозволяє отримати ціле, `nextDouble()` повертає прочитане число типу **`double`**. Є також функції `nextBoolean()`, `nextByte()`, `nextShort()`, `nextLong()`, `nextFloat()` тощо. Усі функції обумовлюють зупинку виконання програми, яке поновлюється після введення з клавіатури відповідного значення.

Для можливості роботи з класом `Scanner` на початку сирцевого коду слід додати **`import`** `java.util.Scanner`.

Це дозволить використовувати ім'я `Scanner` без додаткового префіксу.

Спочатку треба створити об’єкт цього класу за допомогою операції new (докладно ця операція буде розглянута пізніше). Зв’язуємо об’єкт-сканер зі стандартним System.in. Далі можна читати дані різних типів.

Приклад

```
import java.util.Scanner;

public class ScannerTest {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Читання рядка:
        String s = scanner.next();
        // Читання дійсного числа:
        double d = scanner.nextDouble();
        // Читання цілого числа:
        int i = scanner.nextInt();
        // ... використання введених даних
    }
}
```

Великою зручністю класу є можливість довільного розташування даних у вхідному потоці: окремі дані можна розділяти пропусками, табуляцією, або переведенням рядка. Єдиний виняток – використання функції `nextLine()`, яка читає дані до кінця рядка.

Виведення у консольне вікно здійснюється за допомогою функцій об’єкта `out` класу `System`. Функція `print()` дозволяє вивести результат без переходу на новий рядок, println() здійснює перехід на новий рядок після виведення.

Можна також скористатися функцією `printf()` для форматованого виведення. Використання цього методу аналогічне використанню відповідної функції мови С. Перший параметр – так званий рядок форматування. Далі можна вказати довільну кількість параметрів, значення яких слід вивести на консоль.

Приклад

```
double d = 3.5;
int i = 12;
System.out.printf("%f %d%n", d, i);
```

Використовують специфікатори формату, наведені в табл.1.5.

Таблиця 1.5

Специфікатор формату	Форматування, яке здійснюється
%a	Шістнадцяткове значення з плаваючою точкою
%b	Логічне (булеве) значення аргументу
%c	Символьне представлення аргументу
%d	Десяткове ціле значення аргументу
%e	Експоненціальне подання аргументу
%f	Десяткове значення з плаваючою точкою
%g	Вибирає більш коротке представлення з двох: `%e` або `%f`
%o	Вісімкове ціле значення аргументу
%n	Вставка символу нового рядка
%x	Шістнадцяткове ціле значення аргументу
%%	Вставка символу %

Після символу `%` можна вказувати ширину поля.

Є також можливість виведення повідомлень у потік System.err (стандартний потік повідомлень про помилки). Виведення в цей потік має більш високий пріоритет, ніж виведення в System.out, тому іноді повідомлення про помилки випереджають "нормальне" виведення.

1.6. Запуск Java-застосунків з командного рядка

Скомпільовані програми можуть бути запущені в командному рядку. Для запуску командного рядка в Windows використовується команда `cmd`. У командному рядку запускається віртуальна машина Java (команда `java`), далі через пропуск указується параметр `-classpath` (чи `-cp`), після якого через пробіл вказується повний шлях до каталогу проєкту. Далі вказується ім’я пакета і через крапку ім’я класу, що містить функцію `main()`.

Припустимо, проєкт `First` створений у робочому просторі, зв’язаному з текою `D:\workspace`, а функція `main()` реалізована в класі `TestClass` пакету `first`. Для запуску програми в командному рядку варто набрати

```
java -cp D:\workspace\First first.TestClass
```

Вправи для самостійної роботи

1. Увести два дійсних числа. Знайти та вивести середнє геометричне – квадратний корінь з їхнього добутку. Якщо корінь не можна обчислити, вивести повідомлення про помилку.

2. Увести ціле додатне число. Перевірити, чи є воно простим. Вивести відповідно "yes" або "no". Число називається простим, якщо у нього рівно два дільника – 1 і визначене число.

3. Увести дійсне число та цілий показник ступеня (додатний або від'ємний). Обчислити та вивести ступінь.

4. Створити рекурсивну функцію обчислення добутку синусів перших *n* натуральних чисел.

Контрольні запитання

1. Що таке Java-платформа?

2. Що таке віртуальна машина Java?

3. Що таке байт-код?

4. Які інтегровані середовища використовують для створення Java-застосунків?

5. Що таке IntelliJ IDEA?

6. Як створити консольний застосунок у середовищі IntelliJ IDEA?

7. Які файли містять сирцевий код на Java і які містять результат компіляції?

8. Як створюється вбудована документація?

9. У чому полягають особливості типу boolean?

10. Чи завжди потрібна мітка при використанні break?

11. Чи можна звертатися до імен з іншого пакета, не використовуючи операції import?

2. РОБОТА З МАСИВАМИ ТА РЯДКАМИ. СТВОРЕННЯ КЛАСІВ

У розділі розглядаються роботи з типами-посиланнями у Java на прикладі масивів і користувацьких класів. Крім того, розглянуто роботу з найбільш розповсюдженими стандартними класами для роботи з числовими даними і рядками.

2.1. Посилання

У Java немає типів вказівників. Імена змінних непримітивних типів по суті є іменами посилань на відповідні об’єкти. Усі непримітивні типи мають назву типів-посилань (reference types). Розіменування не потрібне: звернення до примітивних типів завжди здійснюється за значенням, а до непримітивних – за посиланням. Типи-посилання ніколи не можуть бути приведені до примітивних і навпаки. У Java немає операцій розіменування (`*` та `->`). Java також не підтримує адресної арифметики.

Спеціальне ключове слово **`null`** використовують для того, щоб показати, що змінна типу-посилання ні на що не посилається. Константа **`null`** може бути присвоєна змінній будь-якого типу-посилання.

Об’єкти, на які вказують посилання, повинні бути розміщені в динамічній пам’яті за допомогою операції **`new`**.

Приклад

```
SomeType st = new SomeType();
```

Присвоювання значення одного посилання іншому не забезпечує копіювання об’єктів. Після присвоювання два посилання посилаються на один об’єкт.

Приклад

```
SomeType a = new SomeType();
SomeType b = new SomeType();
a = b;
```

Об’єкт, на який раніше посилалося `a`, загублений.

На відміну від C++, звільнення пам’яті від непотрібних об’єктів не потрібно. У Java немає операції **`delete`**. Для звільнення пам’яті використовується спеціальний механізм, який має назву збирання сміття. Цей механізм базується на підрахунку посилань на об’єкти. Кожен об’єкт має свій лічильник посилань. Коли посилання копіюється в нову змінну типу-посилання, лічильник збільшується на одиницю. Коли посилання виходить з області видимості, чи перестає вказувати на об’єкт, лічильник зменшується на одиницю. Збирач сміття переглядає список об’єктів і видаляє з пам’яті всі об’єкти, для яких кількість посилань дорівнює 0.

Операція ==, застосована до змінних-посилань здійснює порівняння адрес, а не вмісту об’єктів.

Аргументи функцій типу-посилання передаються у функції за посиланням. Всередині функції створюється нове посилання на той самий об’єкт. Цей об’єкт можна змінити у функції та після повернення з функції використовувати його значення.

2.2. Масиви

2.2.1. Опис і використання масивів

Масив (array) – це набір комірок зберігання даних, кожна з яких має той же тип. Окрема комірка має назву елемента масиву (array item). На відміну від інших складених типів даних, масиви завжди розташовуються в цілісному блоці пам’яті. Оскільки всі елементи займають однакові за розміром комірки пам’яті, адреса конкретного елемента завжди може бути обчислена за його номером. Номери елементів називаються індексами. Звернення до конкретних елементів здійснюють через індекс.

Як і в C++, масиви в Java індексуються починаючи з нуля. Під час опису масиву квадратні дужки ставляться після імені типу, а не імені змінної. Розміщення квадратних дужок після імені змінної допускається, але не рекомендується.

Приклад

```
int[] a;         // посилання на масив
int n = 10;      // можна використовувати змінну
a = new int[n]; // створення масиву
```

У цьому прикладі описується посилання на масив, який створюється пізніше. Розмір масиву не є частиною типу, як це реалізовано в C++. Це дозволяє визначити масив необхідного розміру під час виконання програми.

Приклад

```
int[] numbers;  // посилання на масив цілих довільної довжини
numbers = new int[10]; // numbers складається з 10 елементів
numbers = new int[20]; // numbers складається з 20 елементів
```

У Java підтримуються одно- і багатовимірні масиви (масиви масивів). У прикладі наведено, як можна визначити посилання на двовимірний масив.

Приклад

```
byte[][] scores;
```

Наведений нижче приклад показує, як створити різні масиви.

Приклад

```
int[] numbers = new int[5];           // одновимірний масив
double[][] matrix = new double[5][4]; // прямокутний масив
// Двовимірний масив з різною довжиною рядків:
byte[][] scores = new byte[5][];
for (int i = 0; i < 5; i++) {
    scores[i] = new byte[i + 4];
}
```

Останній приклад показує як можна створити двовимірний масив з різною довжиною рядків. Розмір масиву може бути визначений як за допомогою констант, так і за допомогою змінних і виразів, що дають цілий результат.

У Java масиви містять елементи разом з їх кількістю. Кількість елементів масиву завжди можна отримати за допомогою спеціального поля `length`. Це поле доступне тільки для читання.

Приклад

```
int[] a = new int[10];
System.out.println(a.length); // 10
```

У Java передбачений простий спосіб ініціалізації масиву списком початкових значень.

Приклад

```
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
```

Можна опустити операцію **new**.

Приклад

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

Аналогічно ініціалізуються багатовимірні масиви.

Приклад

```
int[][] b = { { 1, 2, 3 },
              { 0, 0, 1 },
              { 1, 1, 11},
              { 0, 0, 0 } };
```

У наведеному прикладі створюється двовимірний масив з чотирьох рядків і трьох стовпців.

Так виглядає типовий цикл для обходу масиву.

Приклад

```
for (int i = 0; i < a.length; i++) {
    a[i] = 0;
}
```

Аналогічно обхід усіх елементів двовимірного масиву зазвичай здійснюється так.

Приклад

```
for (int i = 0; i < c.length; i++) {
    for (int j = 0; j < c[i].length; j++) {
```

```
            c[i][j] = 0;
        }
    }
```

Як видно з останнього прикладу, для отримання розміру і-го рядка двовимірного масиву використовують конструкцію `c[i].length`.

Альтернативна форма циклу **`for`** (починаючи з JDK 1.5) дозволяє спростити повний обхід масивів. Наприклад, замість

```
int[] nums = { 1, 2, 3, 4, 5, 6 };
for (int i = 0; i < nums.length; i++) {
    System.out.println(nums[i]);
}
```

можна написати

```
int[] nums = { 1, 2, 3, 4, 5, 6 };
for(int n : nums) {
    System.out.println(n);
}
```

Альтернативна форма може бути застосована. лише для читання значень елементів, а не для їхньої модифікації.

Масиви читають з клавіатури поелементно. Наведений нижче приклад демонструє читання кількості та значень елементів з клавіатури.

Приклад

```
public class ArrayTest {
    public static void main(String[] args) {
        System.out.println("Уведіть кількість елементів:");
        java.util.Scanner s =
            new java.util.Scanner(System.in);
        int size = s.nextInt();
        double[] a = new double[size];
        System.out.println("Уведіть елементи масиву:");
        for (int i = 0; i < a.length; i++) {
            a[i] = s.nextDouble();
        }
        // Робота з масивом
    }
}
```

Присвоювання імені масиву іншому імені масиву приводить тільки до копіювання посилання, але не самого масиву. Для копіювання елементів масиву необхідно організувати цикл або скористатися стандартними функціями.

2.2.2. Масиви як параметри та результат функцій

Масиви-параметри передаються у функції за посиланням. Після повернення з функції елементи можуть містити змінені значення.

Приклад

```
public class SwapElements {

    static void swap(int[] a) {
        int z = a[0];
        a[0] = a[1];
        a[1] = z;
    }

    public static void main(String[] args) {
        int[] b = { 1, 2 };
        swap(b);
        System.out.println(b[0]); // 2
        System.out.println(b[1]); // 1
    }

}
```

Якщо параметр описано як одновимірний масив, то можна фактичним параметром вказати рядок двовимірного масиву. Також можна використовувати параметри типу багатовимірних масивів.

Приклад

```
public class ArraysTest {

    static double sum(double[] arr1D) {
        double s = 0;
        for (double elem : arr1D) {
            s += elem;
        }
        return s;
    }

    static double sum(double[][] arr2D) {
        double s = 0;
        for (double[] line : arr2D) {
            for (double elem : line) {
                s += elem;
            }
        }
        return s;
    }
```

```
    public static void main(String[] args) {
        double[][] arr = { { 1, 2 },
                           { 3, 4 },
                           { 5, 6 } };
        System.out.printf(
           "Сума елементів рядка з індексом 1: %f%n",
           sum(arr[1]));
        System.out.printf("Сума всіх елементів: %f%n",
           sum(arr));
    }
}
```

У Java 1.5 з'явилася додаткова можливість створення функцій зі змінним числом параметрів визначеного типу (Variable Arguments List, varargs). Всередині функції такі параметри інтерпретуються як масив.

Приклад

```
static void printIntegers(int... a) {
    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }
}
```

Викликати таку функцію можна у два способи: передаючи список аргументів типу елемента масиву, або передаючи масив цілком.

Приклад

```
public static void main(String[] args) {
    int[] arr = {4, 5};
    printIntegers(arr);
    printIntegers(1, 2, 3);
}
```

Такий параметр може бути лише один і обов'язково розташований останнім в списку.

Функція може повертати посилання на масив. Найчастіше такий масив створюється всередині функції. Створення масиву цілих із заповненням одиницями, наведено у прикладі.

Приклад

```
static int[] arrayOfOnes(int n) {
    int[] arr = new int[n];
    for (int i = 0; i < arr.length; i++) {
```

```
            arr[i] = 1;
        }
        return arr;
    }
```

Тепер можна створити масив за допомогою нашої функції.

Приклад

```
int[] a = arrayOfOnes(6);
```

Якщо потрібен тільки один елемент, для нього можна не створювати масив з ім'ям а поєднати створення масиву з його використанням.

Приклад

```
System.out.println(arrayOfOnes(2)[0]);
```

Можна також повертати посилання на багатовимірні масиви.

Посилання на масиви передаються за значенням. Створення функції, яка змінює розмір існуючого масиву наведено у прикладі.

Приклад

```
static void resize(int[] arr, int newSize) {
    // збереження існуючих елементів
    arr = new int[newSize];
    // копіювання
}
```

Нове посилання слід повертати, інакше новий масив буде втрачено.

Приклад

```
static int[] resize(int[] arr, int newSize) {
    // збереження існуючих елементів
    arr = new int[newSize];
    // копіювання
    return arr;
}
```

2.2.3. Стандартні функції для роботи з масивами

Клас `System` надає найпростіший шлях копіювання одного масиву в інший – використання статичного методу `arraycopy()`.

Приклад

```
System.arraycopy(a, a_from, b, b_from, size);
```

Це еквівалентно циклу, який представлено у прикладі.

Приклад

```
for (int i = a_from, j = b_from; i < size + a_from; i++, j++)
    b[j] = a[i];
```

Масив, у який здійснюється копіювання, повинен мати необхідні розміри. Функція `arraycopy()` не створює нового масиву. Весь масив `a` можна скопіювати в `b` викликом, наведеним у прикладі.

Приклад

```
System.arraycopy(a, 0, b, 0, a.length);
```

Для роботи з масивами можна використовувати статичні методи класу `Arrays`, реалізованого в пакеті `java.util`. Ці методи наведені в табл. 2.1.

Таблиця 2.1

Статичний метод класу Arrays	Опис
`void fill(`*тип*`[] a,` *тип* `val)`	заповнює масив вказаними значеннями
`void fill(`*тип*`[] a, int fromIndex, int toIndex,` *тип* `val)`	заповнює частину масиву вказаними значеннями
`String toString(`*тип*`[] a)`	повертає список елементів масиву у вигляді рядка
`String deepToString(`*тип*`[][] a)`	подає багатовимірний масив у вигляді рядка
`тип[] copyOf(`*тип*`[] a, int len)`	створює новий масив довжини len з копіями елементів
`тип[] copyOfRange(`*тип*`[] a, int from, int to)`	створює новий масив з копіями елементів діапазону
`boolean equals(`*тип*`[] a,` *тип*`[] a2)`	перевіряє еквівалентність елементів двох масивів
`boolean deepEquals(`*тип*`[][] a, тип[][] a2)`	перевіряє еквівалентність елементів багатовимірних масивів

Продовження таблиці 2.1

Статичний метод класу Arrays	Опис
`void sort(`*тип*`[] a)`	здійснює сортування елементів за зростанням
`void sort(`*тип*`[] a, int fromIndex, int toIndex)`	здійснює сортування частини елементів за зростанням

У таблиці *тип* означає один з фундаментальних (примітивних) типів або тип `Object`.

Перший варіант функції `fill()` використовується для заповнення всього масиву, другий – частини, при чому елемент з номером `toIndex` не включається в послідовність.

Приклад

```
public class FillArray {

    public static void main(String[] args) {
        int[] a = new int[6];
        java.util.Arrays.fill(a, 0, 4, 12);
            // Інші елементи дорівнюють 0
        for (int x : a) {
            System.out.print(x + " ");
        }
        System.out.println();
        // Всі елементи дорівнюватимуть 100:
        java.util.Arrays.fill(a, 100);
        for (int x : a) {
            System.out.print(x + " ");
        }
        System.out.println();
    }

}
```

У попередньому прикладі для виведення елементів масиву на екран був використаний цикл. Альтернативний спосіб – використання функції `toString()` класу `Arrays`. Ця функція повертає представлення масиву у вигляді рядка, яке є зручним для більшості застосувань.

Приклад

```
java.util.Arrays.fill(a, 100);
```

```
System.out.println(Arrays.toString(a))
    // [100, 100, 100, 100, 100, 100];
```

Примітка. Для виведення в рядок багатовимірних масивів слід використовувати функцію `deepToString()`.

Клас `Arrays` надає альтернативний шлях копіювання масивів. Функція `copyOf()` створює новий масив копій елементів. Перший параметр – вихідний масив, другий параметр – довжина результуючого масиву. Елементи, які не помістилися, відкидаються, відсутні заповнюються нулями. Функція `copyOfRange(`*тип*`[] a, int from, int to)` копіює в новий масив частину масиву, включаючи початок інтервалу і не включаючи кінця інтервалу.

Приклад

```
import java.util.Arrays;

public class CopyOfTest {

    public static void main(String[] args) {
        int[] a = { 1, 2, 3, 4 };
        int[] b = Arrays.copyOf(a, 3);
        System.out.println(Arrays.toString(b));// [1, 2, 3]
        int[] c = Arrays.copyOf(a, 6);
        System.out.println(Arrays.toString(c));
                                    // [1, 2, 3, 4, 0, 0]
        int[] d = Arrays.copyOfRange(a, 1, 3);
        System.out.println(Arrays.toString(d));   // [2, 3]
    }
}
```

Порівняти два масиви чи частину їх можна за допомогою функцій групи `equals()`. Масиви порівнюються поелементно. Два масиви також вважаються еквівалентними, якщо обидва посилання – **`null`**.

Приклад

```
import java.util.Arrays;

public class ArraysComparison {
    public static void main(String[] args) {
        double[] a = null, b = null;
        System.out.println(Arrays.equals(a, b)); // true
        a = new double[] { 1, 2, 3, 4 };
        b = new double[4];
        System.out.println(Arrays.equals(a, b)); // false
```

```
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println(Arrays.equals(a, b)); // true
        b[3] = 4.5;
        System.out.println(Arrays.equals(a, b)); // false
    }
}
```

Є також метод `deepEquals()`, використання якого аналогічне. Різниця є істотною для багатовимірних масивів. Здійснюється більш "глибока" перевірка.

Приклад

```
int[][] a1 = { { 1, 2 } , { 3, 4 } };
int[][] a2 = { { 1, 2 } , { 3, 4 } };
System.out.println(Arrays.equals(a1, a2));    // false
System.out.println(Arrays.deepEquals(a1, a2));// true
```

За допомогою функції sort() можна здійснити сортування масиву чисел за зростанням.

Приклад

```
import java.util.Arrays;

public class ArraySort {

    public static void main(String[] args) {
        int[] a = new int[] { 11, 2, 10, 1 };
        Arrays.sort(a);            // 1 2 10 11
        for (int x : a) {
            System.out.print(x + " ");
        }
        System.out.println();
    }

}
```

Функція `sort()` реалізована для масивів усіх примітивних типів та рядків. Рядки впорядковуються за алфавітом. Можна також сортувати частину масиву. Як і для функції `fill()`, вказується початковий і кінцевий індекси послідовності, яку слід відсортувати. Кінцевий індекс не включається в послідовність.

Приклад

```
int[] a = { 7, 8, 3, 4, -10, 0 };
java.util.Arrays.sort(a, 1, 4); // 7 3 4 8 -10 0
```

У відсортованих масивах можна виконати пошук за допомогою методів класу `Arrays`. Група функцій `binarySearch()`, реалізована для всіх примітивних типів і типу `Object`, повертає індекс знайденого елемента або від'ємне значення, якщо елемент відсутній.

2.3. Визначення класів

2.3.1. Поля і методи

Клас – це структурований тип даних, набір елементів даних різних типів і функцій для роботи з цими даними. Опис класу складається зі специфікаторів (наприклад, **`public`**, **`final`**), імені, імені базового класу, списку інтерфейсів і тіла у фігурних дужках.

Тіло класу містить поля (їм відповідають елементи даних у C++) і методи (функції-елементи в C++). Поля і методи разом іменуються елементами (членами) класу. Нижче наводиться приклад опису класу.

Приклад

```
class Rectangle {
    double width;
    double height;

    double area() {
        return width * height;
    }
}
```

Після останньої фігурної дужки, що закривається, не слід ставити крапку з комою. Методи завжди реалізуються всередині визначення класу.

Під час створення об'єкта класу поля ініціалізуються усталеними значеннями (нулями або null для посилань). Java допускає ініціалізацію полів початковими значеннями.

Приклад

```
class Rectangle {
    double width = 10;
    double height = 20;
    double area() {
```

```
        return width * height;
    }
}
```

Можна створити спеціальний блок ініціалізації всередині тіла класу. Такий блок виконуватиметься щораз під час створення нового об’єкта.

Приклад

```
class Rectangle {
    double width;
    double height;
    {
        width = 10;
        height = 20;
    }
    double area() {
        return width * height;
    }
}
```

Для того, щоб працювати з полями і методами класу, необхідно створити об’єкт. Для цього спочатку створюють посилання на об’єкт, а потім за допомогою операції **new** створюють сам об’єкт шляхом виклику конструктора. Ці дії можна поєднати. Після цього можна викликати методи і використовувати поля.

Приклад

```
Rectangle rect = new Rectangle();//rect - посилання на об'єкт
double a = rect.area();          //a = 200
rect.width = 15;                 //зміна значення поля
double b = rect.area();          //b = 300
```

Під час виклику методів аргументи передаються за значенням.

Ключове слово **this** використовується як посилання на об’єкт, для якого викликаний метод. Усі нестатичні методи неявно отримують посилання на об’єкт для якого вони використані. Ключове слово **this** використовується явно, наприклад, коли треба повернути з функції посилання на поточний об’єкт, або запобігти конфлікту імен.

2.3.2. Специфікатори доступу. Інкапсуляція

Java підтримує закритий (**private**), пакетний, захищений (**protected**) і відкритий (**public**) рівні доступу. Сам клас може бути оголошений як **public**. Java вимагає окремої специфікації доступу для кожного елемента, або групи полів одного типу.

Приклад

```
public class Rectangle {
    private double width = 10;
    private double height = 20;

    public void setWidth(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getHeight() {
        return height;
    }

    public double area() {
        return width * height;
    }
}
```

Доступ до закритих (**private**) елементів класу обмежений методами всередині класу. У Java немає ключового слова **friend**, яке у C++ забезпечує доступ до закритих елементів ззовні класу.

Відкриті (**public**) елементи відкритого класу можуть бути доступні з будь-якої функції будь-якого пакету.

Елементи класу без атрибутів доступу мають пакетну видимість. Такий доступ ще називають "дружнім". Всі інші класи цього пакета мають доступ до таких елементів як до відкритих. Ззовні пакету такі елементи взагалі недоступні.

Доступ до захищеного (**protected**) елемента класу, визначеного в деякому пакеті, обмежений методами цього класу і похідних класів будь-яких пакетів, а також класів цього пакету.

Інкапсуляція (приховування даних) – одна з трьох парадигм об'єктно-орієнтованого програмування. Зміст інкапсуляції полягає у приховуванні від зовнішнього користувача деталей реалізації об'єкту. Зокрема доступ до даних (полів), які зазвичай описані з модифікатором **private**, здійснюється через відкриті функції доступу. Як правило, це так звані сеттери та геттери. Якщо поле має ім'я `name`, відповідні функції доступу мають імена `setName` та `getName`.

2.3.3. Конструктори

Екземпляр класу створюється шляхом застосування операції `new` до конструктора.

Конструктор – це функція, яка здійснює ініціалізацію даних об'єкта. Ім'я конструктора збігається з ім'ям класу. Не можна вказувати тип результату конструктора. У класі може бути визначено кілька конструкторів. Якщо жоден конструктор явно не визначений, автоматично створюється усталений конструктор (без параметрів). Такий конструктор ініціалізує всі поля усталеними початковими значеннями. Після визначення принаймні одного конструктора усталений конструктор автоматично не створюється.

Один конструктор можна викликати з іншого з використанням слова **this**, після якого випливають необхідні аргументи. Спочатку здійснюється ініціалізація в місці опису, після якої значення можуть бути перевизначені в блоці ініціалізації, а потім перевизначені в конструкторі.

Приклад

```
public class Rectangle {
    private double width = 10;
    private double height = 20;
    {
        width = 30;
        height = 40;
    }
```

```
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public Rectangle() {
        this(50, 60); // виклик іншого конструктора
    }
}
. . .
Rectangle rectangle = new Rectangle(); // width=50, height=60
```

У Java немає конструкторів копіювання і деструкторів. Можна створити спеціальний метод `finalize()`, який викликається збирачем сміття перед ліквідацією об'єкта. У деяких випадках об'єкт може бути не вилучений збирачем сміття ніколи (пам'яті вистачало до кінця програми), отже метод `finalize()` може бути ніколи не викликаний.

2.3.4. Статичні елементи. Константи

Методи і поля можуть бути оголошені з ключовим словом `static`. Звернення до таких полів і методів може здійснюватися без створення екземпляра класу. Статичні поля є альтернативою відсутнім у Java глобальним змінним. На відміну від C++, статичні елементи даних не потрібно окремо визначати в глобальній області видимості. Статичні поля можуть бути проініціалізовані під час створення.

Приклад

```
class SomeClass {
    static double x = 10;
    static int    i = 20;
}
```

Можна створити окремий блок статичної ініціалізації.

Приклад

```
class SomeClass {
    static double x;
    static int i;
    static {
        x = 10;
        i = 20;
```

```
    }
}
```

На відміну від нестатичної ініціалізації, створення й ініціалізація статичних полів здійснюється під час першого звернення до класу (створенні екземпляра класу чи зверненні до статичних елементів). Java не створює статичних полів для класів, які не використовуються.

Статичні методи не отримують посилання на об’єкт і не можуть використовувати посилання **this**.

Звернення до статичних елементів може здійснюватися як через ім’я класу, так і через посилання на об’єкт.

Приклад

```
SomeClass.x = 30;
SomeClass s = new SomeClass();
s.x = 40;
```

Всередині класів можна визначати константи. Константи можуть бути двох видів – статичні й нестатичні. Статичну константу створює компілятор. За угодою її ім’я має містити лише великі літери.

Приклад

```
public static final double PI = 3.14159265;
```

Значення нестатичної константи слід визначити, причому один раз – у місці визначення, в блоці ініціалізації (тоді це значення буде однаковим для всіх екземплярів), або в конструкторі.

Приклад

```
public class ConstDemo {
    public final int one = 1;
    public final int two;
    {
        two = 2;
    }

    public final int other;

    public ConstDemo(int other) {
        this.other = other;
    }
```

```
}
```

Цілком безпечно визначати константи як **`public`**, оскільки компілятор не дозволить змінити їх значення.

2.4. Використання стандартних класів

Раніше вже використовувалися статичні засоби стандартних класів `System` (поля-потоки `in` і `out`), `Math` (стандартні математичні функції, реалізовані у вигляді статичних методів) і `Arrays`. Крім того, створювався об'єкт класу `java.util.Scanner` з викликом його методів.

Для тестування програмного забезпечення, статистичного моделювання, в задачах криптографії тощо використовують випадкові та псевдовипадкові значення, які можна отримати за допомогою класу `java.util.Random`.

Практично жодна програма на Java не може обійтися без об'єктів класу `String`: функція `main()` описується з параметром типу масиву рядків, дані читаються з потоків у рядки і записуються з рядків у потоки, рядки використовуються для представлення даних у візуальних компонентах графічного інтерфейсу користувача тощо. Для модифікації вмісту рядків використовуються стандартні класи `StringBuffer` і `StringBuilder`. Для поділу рядку на лексеми використовують клас `StringTokenizer`.

Класи-обгортки `Integer`, `Double`, `Boolean`, `Character`, `Float`, `Byte`, `Short` і `Long` використовують для зберігання даних примітивних типів у об'єктах, з якими можна працювати, як з посиланнями. Крім того, ці класи надають ряд корисних методів для перетворення даних.

2.5. Робота з випадковими величинами

Іноді під час тестування виникає необхідність у заповненні масивів випадковими (псевдовипадковими) значеннями. Це можна здійснити за допомогою функції `random()` класу `Math` і за допомогою спеціального класу `java.util.Random`. Перший варіант дає випадкове число у діапазоні від 0 до 1.

Приклад

```
import java.util.Arrays;

public class MathRandomTest {
    public static void main(String[] args) {
        double[] a = new double[5];
        for (int i = 0; i < a.length; i++) {
            a[i] = Math.random() * 10;
                // випадкові значення від 0 до 10
        }
        System.out.println(Arrays.toString(a));
    }
}
```

Використання класу `Random` дозволяє отримати більш різноманітні результати. Конструктор класу `Random` без параметрів ініціалізує генератор псевдовипадкових чисел так, що послідовності випадкових значень практично не повторюються. Якщо для налагодження нам необхідно кожного разу отримувати однакові випадкові значення, слід скористатися конструктором з цілим параметром. Параметром може бути будь-яке ціле, яке ініціалізує генератор випадкових чисел.

У таблиці 2.2 представлені функції класу `java.util.Random`, що дозволяють отримати різні псевдовипадкові значення.

Таблиця 2.2

Функція	Опис
`nextBoolean()`	повертає наступне рівномірно розподілене значення типу boolean
`nextDouble()`	повертає наступне значення типу double, рівномірно розподілене на інтервалі від 0 до 1
`nextFloat()`	повертає наступне значення типу float, рівномірно розподілене на інтервалі від 0 до 1
`nextInt()`	повертає наступне рівномірно розподілене значення типу int
`nextInt(int n)`	повертає наступне значення типу int , рівномірно розподілене від 0 до n (не включаючи n)
`nextLong()`	повертає наступне рівномірно розподілене значення типу long

Продовження таблиці 2.2

Функція	Опис
`nextBytes(byte[] bytes)`	заповнює масив цілих типу byte випадковими значеннями
`nextGaussian()`	повертає наступне значення типу double, розподілене на інтервалі від 0 до 1 за нормальним законом

У наведеному нижче прикладі ми отримуємо псевдовипадкові цілі значення в діапазоні від 0 (включно) до 10.

Приклад

```
public class UtilRandomTest {

    public static void main(String[] args) {
        int[] a = new int[15];
        Random rand = new Random(100);
        for (int i = 0; i < a.length; i++) {
            a[i] = rand.nextInt(10);
                // випадкові значення від 0 до 10
        }
        System.out.println(Arrays.toString(a));
    }

}
```

Після кожного запуску програми ми будемо отримувати однакову послідовність псевдовипадкових значень. Якщо ми хочемо, щоб значення були дійсно випадковими, треба скористатися конструктором `Random()` без параметрів.

2.6. Рядки

2.6.1. Використання класу String

Рядки в Java – це екземпляри класу `java.lang.String`. Об’єкти цього класу містять символи Unicode. Об’єкт-рядок може бути створений під час опису посилання шляхом присвоювання йому рядкового літералу.

Приклад

```
String s = "Перший рядок";
```

Рядок можна також створити за допомогою різних конструкторів. Клас `String` у Java надає 15 конструкторів, які дозволяють визначити початкове значення рядка. Отримання рядка з масиву символів наведено у прикладі.

Приклад

```
char[] chars = { 'Т', 'е', 'к', 'с', 'т' };
String s1 = new String(chars); // Текст
```

Можна створити масив байтів і отримати з нього рядок. Перетворення байтів у символи здійснюється згідно з усталеною таблицею кодування, прийнятою в системі.

Приклад

```
byte[] bytes = { 49, 50, 51, 52 };
String s2 = new String(bytes);
System.out.println(s2); // 1234
```

Можна створити рядок з іншого рядка. Слід відрізняти створення нового посилання від створення нового рядка.

Приклад

```
String s = "текст";
String s1 = s; // s і s1 посилаються на один рядок
String s2 = new String(s); // s2 посилається на новий рядок -
                           // копію s
```

Якщо один з операндів – рядок, а інший – ні, то цей операнд подається у вигляді рядка. Альтернативний спосіб перетворення числових даних у рядки – застосування статичних функцій `valueOf()`. Відповідні функції реалізовані для аргументів числових типів, а також типів `Object`, `char`, `boolean` і масивів символів.

Приклад

```
double d = 1.1;
String sd = String.valueOf(d); // "1.1"
```

У таблиці 2.3 наведені методи класу `String`, які використовують найбільш часто.

Таблиця 2.3

Метод	Аргументи	Повертає	Опис
`length`	`()`	**`int`**	Повертає кількість символів у рядку
`concat`	`(String str)`	`String`	Додає вказаний рядок у кінець поточного рядка
`charAt`	`(int index)`	**`char`**	Повертає символ, що відповідає певному індексу
`compareTo`	`(String value)`	**`int`**	Порівнює поточний рядок з аргументом-рядком
`compareToIgnoreCase`	`(String str)`	**`int`**	Працює як compareTo(), але ігнорує регістр
`equals`	`(String value)`	**`boolean`**	Порівнює поточний рядок з аргументом-рядком. Повертає true, якщо рядки збігаються
`equalsIgnoreCase`	`(String str)`	**`boolean`**	Порівнює поточний рядок з аргументом-рядком з ігноруванням регістрів
`indexOf`	`(String substring)`	**`int`**	Повертає індекс, який відповідає першому входженню підрядка у рядок
`indexOf`	`(char ch)`	**`int`**	Повертає індекс, який відповідає першому входженню символу в рядок. Якщо символ відсутній – повертає –1
`lastIndexOf`	`(String substring)`	**`int`**	Повертає індекс, який відповідає останньому входженню підрядка у рядок
`lastIndexOf`	`(`**`char`** `ch)`	**`int`**	Повертає індекс, який відповідає останньому входженню символу
`substring`	`(`**`int`** `beginindex,` **`int`** `endindex)`	`String`	Повертає новий рядок, що є підрядком вихідного рядку

Продовження таблиці 2.3

Метод	Аргументи	Повертає	Опис
`toLowerCase`	`()`	`String`	Повертає рядок, усі символи якого переведені в нижній регістр
`toUpperCase`	`()`	`String`	Повертає рядок, усі символи якого переведені у верхній регістр
`regionMatches`	`(int toffset, String other, int ooffset, int len)`	**`boolean`**	Перевіряє, чи збігаються дві послідовності символів у двох рядках
`regionMatches`	`(`**`boolean`** `ignoreCase,` **`int`** `toffset, String other,` **`int`** `ooffset,` **`int`** `len)`	**`boolean`**	Перевіряє, чи збігаються дві послідовності символів у двох рядках. Додатково можна встановлювати можливість ігнорування регістра під час перевірки
`toCharArray`	`()`	**`char`**`[]`	Перетворює поточний рядок у новий масив символів
`getChars`	`(int srcBegin, int srcEnd, char[] dst, int dstBegin)`	**`void`**	Копіює символи поточного рядка у результуючий масив символів
`getBytes`	`()`	**`byte`**`[]`	Повертає масив байтів, що містять коди символів з урахуванням таблиці кодів
`trim`	`()`	`String`	Повертає копію рядка, у якій початкові і кінцеві пропуски опущені
`startsWith`	`(String prefix)`	**`boolean`**	Перевіряє, чи починається рядок із зазначеного префікса
`endsWith`	`(String suffix)`	**`boolean`**	Перевіряє, чи закінчується рядок зазначеним суфіксом

Наведений нижче приклад демонструє використання методів обробки рядків.

Приклад

```
String s1 = new String("Hello World.");
int i = s1.length();   // i = 12
char c = s1.charAt(6); // c = 'W'
i = s1.indexOf('e');   // i = 1 (індекс 'e' у "Hello World.")
String s2 = "abcdef".substring(2, 5); // s2 = "cde"
int k = "AA".compareTo("AB");         // k = -1
s2 = "abc".toUpperCase();             // s2 = ABC
```

Одна з найбільш типових операцій з рядками – зшивання. Для зшивання двох рядків можна застосувати функцію `concat()`.

Приклад

```
String s3 = s1.concat(s2);
s3 = s3.concat("додаємо текст");
```

Але найчастіше замість функції `concat()` застосовують операцію +.

Приклад

```
String s1 = "first";
String s2 = s1 + " and second";
```

Якщо один з операндів – рядок, а інший – ні, то цей операнд приводиться до рядкового представлення.

Приклад

```
int n = 1;
String sn = "n дорівнює " + n; // "n дорівнює 1"
double d = 1.1;
String sd = d + ""; // "1.1"
```

Можна також використовувати операцію "+=" для дошивання в кінець рядка.

Можна створювати масиви рядків. Як і для інших типів-посилань, масив зберігає не рядки безпосередньо, а посилання на них. Функція `sort()` класу `java.util.Arrays` реалізована також для масивів рядків. Рядки впорядковуються за алфавітом.

Приклад

```
String[] a = { "dd", "ab", "aaa", "aa" };
java.util.Arrays.sort(a); // aa aaa ab dd
```

Для читання рядка з потоку з використанням класу `java.util.Scanner` рядок можна отримати за допомогою методу `next()` (до роздільника) або `nextLine()` (до кінця рядка).

Екземпляр класу `String` не може бути змінений після створення. Робота деяких методів та операцій ззовні нагадує модифікацію об'єкту, однак насправді створюється новий рядок.

Приклад

```
String s = "ab";// У пам'яті один рядок
s = s += "c";   // У пам'яті три рядки:
                // "ab", "c" та "abc". На "abc" посилається s
// Зайві рядки потім будуть видалені збирачем сміття
```

2.6.2. Використання класів StringBuffer і StringBuilder

Існує спеціальний клас `StringBuffer`, що дозволяє модифікувати вміст рядкового об'єкта. Починаючи з Java 5, замість `StringBuffe`r можна використовувати `StringBuilder`. У програмах, які не створюють окремих потоків виконання, функції цього класу виконуються більш ефективно.

Створити об'єкт типу `StringBuilder` можна з існуючого рядка. Після модифікації можна створити новий об'єкт класу `String`, використовуючи об'єкт класу `StringBuilder`.

Приклад

```
String s = "abc";
StringBuilder sb1 = new StringBuilder(s); //Виклик
                                          // конструктора
StringBuilder sb2 = new StringBuilder("cd");
// модифікація sb1 та sb2
// ...
String s1 = new String(sb1); // Виклик конструктора
String s2 = sb2 + "";        // Перетворення типів
```

Окрім деяких типових для класу `String` функцій, таких як `length()`, `charAt()`, `indexOf()`, `substring()`, клас `StringBuilder` надає низку методів для модифікації вмісту. Це такі методи, як `append()`, `delete()`,

deleteCharAt(), insert(), replace(), reverse(), та setCharAt(). Розглянемо використання цих функцій на наведеному нижче прикладі.

Приклад

```
public class StringBuilderTest {
    public static void main(String[] args) {
        String s = "abc";
        StringBuilder sb = new StringBuilder(s);
        sb.append("d");          // abcd
        sb.setCharAt(0, 'f');    // fbcd
        sb.delete(1, 3);         // fd
        sb.insert(1, "gh");      // fghd
        sb.replace(2, 3, "mn"); // fgmnd
        sb.reverse();            // dnmgf
        System.out.println(sb);
    }
}
```

Використання StringBuilder може підвищити ефективність роботи програми у випадках, коли певний рядок зазнає багаторазових модифікацій протягом роботи програми. Але важливо пам’ятати, що декілька посилань вказують на один об’єкт типу StringBuilder. Тому, коли ми його змінюємо, усі посилання вказуватимуть на змінений рядок.

2.6.3. Поділ рядка на лексеми

Існує кілька способів поділу рядка на лексеми. Найпростіший спосіб – використання класу java.util.StringTokenizer. Об’єкт цього класу створюється за допомогою конструктора з параметром типу String, який визначає рядок, що підлягає поділу на лексеми.

Приклад

```
StringTokenizer st = new StringTokenizer(someString);
```

Після створення об’єкта можна отримати загальну кількість лексем за допомогою методу countTokens(). Клас реалізує внутрішній "поточний вказівник", який вказує на наступне слово. Функція nextToken() повертає наступну лексему з рядка. Функції nextToken() можна задати альтернативний роздільник лексем як параметр. За допомогою функції hasMoreTokens() можна

перевірити, чи є ще лексеми. У наведеному нижче прикладі всі слова рядка виводяться у окремих рядках.

Приклад

```
import java.util.*;

public class AllWords {

    public static void main(String[] args) {
        String s = new Scanner(System.in).nextLine();
        StringTokenizer st = new StringTokenizer(s);
        while (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }

}
```

Більш сучасний спосіб розбиття на лексеми – використання методу `split()` класу `String`. Параметр цього методу – так званий регулярний вираз, який визначає роздільники. Регулярні вирази дозволяють визначати шаблони для рядків. Наприклад, `"\\s"` – це будь-який символ-роздільник. Однак у найпростішому випадку можна використовувати безпосередньо розділовий символ – пропуск.

Приклад

```
String s = "aa bb ccc";
String[] a = s.split(" ");
System.out.println(Arrays.toString(a)); // [aa, bb, ccc]
```

2.7. Класи Integer, Double, Boolean, Character, Float, Byte, Short і Long

Класи `Integer`, `Double`, `Boolean`, `Character`, `Float`, `Byte`, `Short` і `Long` дозволяють представити числові та булеві значення в об’єктах. Тому ці класи також називають класами-обгортками. Додатково ці класи надають набір методів для перетворення арифметичних значень у представлення рядком і навпаки, й інші засоби для зручної роботи з цілими і дійсними числами.

Статичний метод `Double.parseDouble()` повертає дійсне число за представленням у вигляді рядку.

Приклад

```
String s = "1.2";
double d = Double.parseDouble(s);
```

Аналогічно працюють функції `Integer.parseInt()`, `Long.parseLong()`, `Float.parseFloat()`, `Byte.parseByte()`, `Short.parseShort()` та `Boolean.parseBoolean()`.

У версії Java 5 (JDK 1.5) об'єкти типу `Integer` можна ініціалізувати виразами цілого типу, використовувати у виразах для одержання значень (автоматичне упакування / розпакування). Цілі значення (константи) можна заносити в списки й інші контейнери. Автоматично будуть створюватися і заноситися в контейнер об'єкти типу `Integer`. У попередній версії Java (JDK 1.4) необхідно було писати, як наведено у прикладі.

Приклад

```
Integer m = new Integer(10); // ініціалізуємо об'єкт
                             // типу Integer
int k = m.intValue() + 1;// використовуємо значення у виразі
// створюємо масив:
Integer[] a = {new Integer(1),
               new Integer(2),
               new Integer(3)};
a[2] = new Integer(4); // заносимо об'єкт з новим цілим
    // значенням, отримаємо об'єкт типу Integer
    // з масиву і використовуємо значення:
int i = a[1].intValue() + 2;
```

Тепер усе простіше.

Приклад

```
Integer m = 10; // ініціалізуємо об'єкт типу Integer
int k = m + 1;  // використовуємо значення у виразі
// створюємо масив:
Integer[] a = {1, 2, 3};
a[2] = 4;       // заносимо об'єкт з новим цілим значенням
// отримаємо об'єкт типу Integer з масиву
// і використовуємо значення:
int i = a[1] + 2;
```

Автоматичне упакування і розпакування – це операція, яка вимагає додаткових ресурсів, зокрема, неявного створення об'єктів. Тому, наприклад

операція `m++` для змінної `m` типу `Integer` у циклі виконуватиметься вкрай неефективно.

Те ж саме стосується інших типів-обгорток. Фактично об'єкти цих типів можуть бути використані замість змінних відповідних примітивних типів. Недоліком використання типів-обгорток є зменшення ефективності за рахунок додавання операцій розміщення у динамічній пам'яті. Але перевагою є можливість використання значення null. Якщо значення функції не може бути обчислене, функція може повернути null, як наведено у прикладі.

Приклад

```
import java.util.Scanner;

public class Reciprocal {

    // Зворотна величина:
    static Double reciprocal(double x) {
        if (x == 0) {
            return null;
        }
        return 1 / x;
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        double x = s.nextDouble();
        Double y = reciprocal(x);
        if (y == null) {
            System.out.println("Помилка");
        }
        else {
            System.out.println(y);
        }
    }
}
```

Клас `Character` – це, в першу чергу, оболонка для зберігання значення примітивного типу **`char`** (символ) в об'єкті. Об'єкт типу `Character` містить одне поле типу **`char`**. Крім того, цей клас надає кілька методів для визначення категорії символу (малі літери, цифри і т.д.) і для перетворення символів з верхнього регістру в нижній і навпаки.

Є можливість переводити окремий символ в верхній (або нижній) регістр.

Приклад

```
char c1 = 'a';
char c2 = Character.toUpperCase(c1); // 'A'
char c3 = Character.toLowerCase(c2); // 'a'
```

Є функції, що дозволяють перевіряти властивості символів. Наприклад, метод `Character.isLetter()` повертає **`true`**, якщо символ є літерою в англійській, українській, російській, китайській, німецькій, арабській або іншій мові. Нижче наведені деякі найбільш корисні методи порівняння символів:

– `isDigit()` повертає **`true`**, якщо символ є цифрою;

– `isLetter()` повертає **`true`**, якщо символ є літерою;

– `isLetterOrDigit()` повертає **`true`**, якщо символ є літерою або цифрою;

– `isLowerCase()` повертає **`true`**, якщо символ є літерою в нижньому регістрі;

– `isUpperCase()` повертає **`true`**, якщо символ є літерою у верхньому регістрі;

– `isSpaceChar()` повертає **`true`**, якщо символ є роздільником – символом пробілу, нового рядка або символом табуляції.

2.8. Використання аргументів командного рядку

У Java можна організувати читання аргументів з командного рядку (окремих слів, набраних в командному рядку після імені головного класу)., Наведена у прикладі нижче програма виводить перший аргумент командного рядку на екран.

Приклад

```
public static void main(String[] args) {
    System.out.println(args[0]);
}
```

Аргументи командного рядку у програмі представлені у вигляді масиву рядків. Для того, щоб отримати числове значення, яке представлено рядком,

використовують функції класів `Integer` та `Double`. У наведеному нижче прикладі програма здійснює читання з командного рядку цілого та дійсного значень та знаходить їхню суму.

Приклад

```
public class TestArgs {

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        double x = Double.parseDouble(args[1]);
        double y = n + x;
        System.out.println(y);
    }

}
```

Кількість аргументів, які були уведені і командному рядку, можна отримати за допомогою виразу `args.length`.

Вправи для самостійної роботи

1. Увести з клавіатури кількість елементів та елементи одновимірного масиву цілих чисел. Знайти добуток максимального і мінімального елементів.

2. Увести з клавіатури рядок (`String`), змінити порядок символів на зворотний та вивести рядок на екран.

3. Створити клас з конструктором для опису точки в тривимірному просторі.

4. Створити клас з конструктором для опису користувача (зберігаються ім’я та пароль).

Контрольні запитання

1. Чим типи-посилання відрізняються від типів-значень?

2. Що є результатом присвоєння одного посилання іншому?

3. У чому полягає "збирання сміття"?

4. Як визначити кількість стовпців двовимірного масиву?

5. Чим відрізняється застосування двох різних конструкцій **for** для обходу елементів масиву?

6. Як у Java створити функцію зі змінною кількістю аргументів?

7. З яких основних елементів складається опис класу?

8. Чим відрізняються статичні та нестатичні елементи класу?

9. У чому полягає зміст інкапсуляції та як вона реалізована в Java?

10. Як можна використовувати посилання **this**?

11. Що таке конструктор?

12. Чому в Java немає деструкторів?

13. Чи можна змінити вміст раніше створеного рядка?

14. Як перевести число в його рядкове представлення і навпаки?

3. УСПАДКУВАННЯ ТА ПОЛІМОРФІЗМ

У розділі розглядаються особливості побудови програми, за допомогою реалізації окремих підпрограм і подальшої їх інтеграції. Матеріал розділу зосереджений довкола таких конструкцій мови програмування С++, як функції та посилання. Також вводяться в розгляд складені типи даних і принципи роботи з ними.

3.1. Композиція класів

Під композицією класів розуміють створення нових класів з використанням об’єктів інших класів як полів. Java не дозволяє розміщення об’єктів всередині інших об’єктів, можна тільки описувати посилання. Композиція класів у цьому випадку припускає створення об’єктів безпосередньо (у тілі класу при оголошенні полів) чи в конструкторах.

Приклад

```
class X {
}

class Y {
}

class Z {
    X x = new X();
    Y y;
    Z() {
        y = new Y();
    }
}
```

Можна також створити внутрішній об’єкт безпосередньо перед його першим використанням.

Відношення, що моделюється композицією, часто називають відношенням "has-a".

Агрегування – це різновид композиції, який передбачає, що сутність (екземпляр) міститься в іншій сутності, яка не може бути створена та існувати без сутності, яка її охоплює. При цьому сутність, що охоплює, може існувати

без внутрішньої, тобто час життя зовнішньої та внутрішньої сутностей може не збігатися. Більш строге трактування композиції (власне композиція) передбачає, що час життя зовнішньої та внутрішньої сутностей збігається. На рівні Java агрегування передбачає можливість створення внутрішнього об'єкта перед його використанням, тоді як строга композиція передбачає створення внутрішнього об'єкта в тілі класу, в блоці ініціалізації або в конструкторі.

3.2. Успадкування

Механізм успадкування полягає в породженні похідних класів від базових. Якщо один клас (похідний) є нащадком іншого (базового), то спадкоємець має можливість безпосередньо користуватися неприватними даними і функціями, визначеними в базовому класі. Відносини між класами і підкласами (нащадками) називаються ієрархією спадкування класів.

На відміну від C++, у Java дозволяється тільки одиничне успадкування класів – клас може мати тільки один базовий клас. Успадкування завжди відкрите. У Java також немає захищеного і закритого успадкування. Успадкування має такий синтаксис:

```
class DerivedClass extends BaseClass {
    // тіло класу
}
```

Функції похідного класу мають доступ до елементів, описаних як **`public`** і **`protected`** (захищені). Члени класу, оголошені як захищені, можуть використовуватися класами-нащадками, а також у межах пакета. Закриті (приватні, private) члени класу недоступні навіть для його нащадків.

Усі класи Java безпосередньо чи опосередненo походять від класу `java.lang.Object`. Цей клас надає набір корисних методів, таких як `toString()` для отримання даних будь-якого об'єкта у вигляді рядка тощо. Базовий клас `Object` не вказують явно.

Клас успадковує всі елементи базового класу, крім конструкторів. До початку виконання конструктора похідного класу викликається конструктор базового класу (усталений конструктор, якщо явно не викликано іншого).

Ключове слово **super** використовують для доступу до елементів базового класу з похідного класу, зокрема:

– для виклику перекритого методу базового класу;

– для передачі параметрів конструктору базового класу.

Приклад

```
class BaseClass {
    int i, j;
    BaseClass(int i, int j) {
        this.i = i;
        this.j = j;
    }
}

class DerivedClass extends BaseClass {
    int k;
    DerivedClass(int i, int j, int k) {
        super(i, j);
        this.k = k;
    }
}
```

Доступ до базового класу з використанням **super** дозволений тільки в конструкторах і нестатичних методах.

Класи можуть бути визначені з модифікатором **final** (фінальний). Фінальні класи не можуть використовуватися як базові. Методи з модифікатором **final** не можуть бути перевизначені.

Приклад

```
final class A {
    void f() { }
}

class B {
    final void g() { }
}

class C extends A { // Помилка! Не можна успадкувати від A
```

```
}

class D extends B {
    void g() { }    // Помилка! g() не можна перекрити
}
```

Посилання на похідний клас неявно приводяться до посилання на базовий клас. Об’єкти похідних класів завжди можна використовувати там, де потрібен об’єкт базового класу.

Приклад

```
class Base {
    static void f(Base b) { }
}

class Derived extends Base {

    public static void main(String[] args) {
        Base b;
        b = new Derived(); // Неявне приведення
        Derived d = new Derived();
        f(d);              // Неявне приведення
    }
}
```

Зворотне приведення необхідно робити явно.

Приклад

```
Base b = new Base();
Derived d = (Derived) b;
```

3.3. Анотації (метадані)

Анотації дозволяють включити в програмний код додаткову інформацію, яка не може бути визначена за допомогою засобів мови. У тексті програми анотації починаються із символу `@`. Типовий приклад анотації – `@Override`. Завдяки цій анотації компілятор може перевірити, чи дійсно відповідний метод був оголошений у базових класах.

Приклад

```
public class MyClass {

    @Override
```

```
    public String toString() {
        return "My overridden method!";
    }

}
```

Можна навести інші приклади анотацій:

– `@SuppressWarnings("ідентифікатор_попередження")` – попередження компілятора повинні бути замовчані в анотованому елементі;

– `@Deprecated` – використання анотованого елемента не є більше бажаним.

Java дозволяє визначати власні анотації.

3.4. Поліморфізм

3.4.1. Загальні концепції

Поліморфізм часу виконання – це властивість класів, згідно з якою поведінка об’єктів класу може визначатися не на етапі компіляції, а на етапі виконання. Класи, що надають ідентичний інтерфейс, але реалізовані під конкретні специфічні вимоги, мають назву поліморфних класів.

Підключення тіла функції до точки її виклику має назву зв’язування. Якщо воно відбувається до початку виконання програми, мова йде про раннє зв’язування. Цей тип зв’язування притаманний мовам процедурного типу, таким як C чи Pascal. Пізнє зв’язування означає, що підключення відбувається під час виконання програми та в об’єктно-орієнтованих мовах залежить від типів об’єктів. Пізнє зв’язування ще називають динамічним, або зв’язуванням часу виконання. Для реалізації поліморфізму використовується механізм пізнього зв’язування.

У мовах об’єктно-орієнтованого програмування пізнє зв’язування реалізоване через механізм віртуальних функцій. Віртуальна функція (віртуальний метод, virtual method) – це функція, визначена в базовому класі, та перевизначена (перекрита) у похідних, так, що конкретна реалізація функції для виклику визначатиметься під час виконання програми. Вибір реалізації віртуальної функції залежить від реального (а не оголошеного під час опису)

типу об’єкта. Оскільки посилання на базовий тип може містити адресу об’єкта будь-якого похідного типу, поведінка раніше створених класів може бути змінена пізніше шляхом перевизначення віртуальних методів. Перевизначення передбачає відтворення імені, списку параметрів та специфікатора доступу. Фактично поліморфними є класи, які містять віртуальні функції.

У C++ для позначення віртуальної функції використовують модифікатор virtual. У Java всі методи є віртуальними, за винятком конструкторів, статичних (**`static`**), фінальних (**`final`**) і закритих (**`private`**) методів. На відміну від C++, слово `virtual` не використовується.

Починаючи з Java 5, перед перевизначеними віртуальними методами розміщують анотацію `@Override`, яка дозволяє компілятору здійснити додаткову перевірку синтаксису – відповідність сигнатури нової функції сигнатурі перекритої функції базового класу. Використання `@Override` є бажаним, але не обов’язковим.

Усі класи Java є поліморфними, оскільки таким є клас `java.lang.Object`. Зокрема, завдяки поліморфізму кожен клас може визначити свою віртуальну функцію `toString()`, яка буде викликана для автоматичного отримання даних про об’єкт у вигляді рядку.

У Java є ключове слово **`instanceof`**, яке дозволяє перевірити, чи є об’єкт екземпляром певного типу (або похідних типів). Вираз `об'єкт` **`instanceof`** `клас` повертає значення типу `boolean`, яке може бути використане для перевірки, чи можна викликати метод цього класу.

Приклад

```
if (x instanceof SomeClass)
    ((SomeClass)x).someMethod();
```

3.4.2. Абстрактні класи та методи

Іноді класи створюються для представлення абстрактних концепцій, а не для створення екземплярів. Такі концепції можуть бути представлені абстрактними класами. У Java для цього використовується ключове слово **`abstract`** перед визначенням класу.

Приклад

```
abstract class SomeConcept {
    . . .
}
```

Абстрактний клас може містити абстрактні методи, такі, для яких не приводиться реалізація. Такі методи не мають тіла функції. Їхнє оголошення аналогічне оголошенню функцій-елементів у C++, але оголошенню повинне передувати ключове слово **abstract**.

Наприклад, абстрактний клас `Shape` (геометрична фігура) реалізує поля і методи, що можуть бути використані різними похідними класами. До таких полів можна, наприклад, віднести поточну позицію і метод переміщення по екрану `moveTo()`. У класі `Shape` також оголошені абстрактні методи, такі як `draw()`, що повинні бути реалізовані у всіх похідних класах, але по-різному. Усталена реалізація не має сенсу.

Приклад

```
abstract class Shape {
    int x, y;
    . . .
    void moveTo(int newX, int newY) {
        . . .
    }
    abstract void draw();
}
```

Конкретні класи, створені від `Shape`, такі як `Circle` або `Rectangle`, визначають реалізацію методу `draw()`.

Приклад

```
class Circle extends Shape
{
    void draw() {
        . . .
    }
}

class Rectangle extends Shape {
```

Абстрактні методи аналогічні суто віртуальним функціям у C++.

Від абстрактного класу не вимагають обов'язкової наявності абстрактних методів. Але кожен клас, у якому є хоч один абстрактний метод, чи хоча б один абстрактний метод базового класу не був визначений, повинен бути оголошений як абстрактний (з використанням ключового слова **abstract**).

3.5. Загальні відомості про інтерфейси. Упорядкування об'єктів

У Java використовується поняття інтерфейсів. Інтерфейс може розглядатися як чисто абстрактний клас, але на відміну від абстрактних класів інтерфейс ніколи не містить даних, тільки методи. Ці методи усталено вважаються публічними.

Приклад

```
interface SomeFunctions {
    void f();
    int g(int x);
}
```

Кожен клас може бути створений тільки від одного базового класу, але при цьому реалізовувати один чи кілька інтерфейсів. Клас, що реалізує інтерфейс, повинен забезпечити реалізацію всіх методів, оголошених в інтерфейсі. В іншому випадку такий клас буде абстрактним і повинен бути оголошений зі специфікатором **abstract**.

Інтерфейси формально можуть містити поля, але вони є фінальними і статичними (константами часу компіляції). Вони повинні бути ініціалізовані під час створення. Інтерфейси не можуть містити конструкторів, оскільки немає ніяких даних окрім статичних констант.

Для того, щоб вказати, що клас реалізує інтерфейс, ім'я інтерфейсу вказують у списку реалізованих інтерфейсів. Такий список розташовують у заголовку класу після ключового слова **implements**. Методи, визначені в інтерфейсі, є абстрактними і відкритими. У класі, що реалізує інтерфейс, такі методи повинні бути оголошені як **public**.

Приклад

```
interface SomeFunctions {
    void f();
    int g(int x);
}

class SomeClass implements SomeFunctions {
    @Override
    public void f() {

    }

    @Override
    public int g(int x) {
        return x;
    }
}
```

Примітка. Використання анотації `@Override` є бажаним, але не обов'язковим.

Інтерфейс може мати кілька базових інтерфейсів. Множинне успадкування інтерфейсів є безпечним з точки зору дублювання даних і конфліктів імен.

Приклад

```
interface SomeFunctions {
    void f();
    int g(int x);
}

interface AnotherFunction {
    void h(int z);
}

interface AllFunctions extends SomeFunctions,
                               AnotherFunction {
    int g(int x); // оголошення може повторюватися
}
```

Тепер клас, який реалізує інтерфейс `AllFunctions`, повинен визначати три функції.

Приклад

```
class Implementation implements AllFunctions {
    @Override
    public void f() {

    }

    @Override
    // Одна реалізація використовується для базового і
похідного інтерфейсів:
    public int g(int x) {
        return x;
    }

    @Override
    public void h(int z) {

    }
}
```

Клас може реалізувати кілька інтерфейсів. Це – більш розповсюджений шлях, ніж створення похідного інтерфейсу.

Приклад

```
class AnotherImplementation implements SomeFunctions,
                                       AnotherFunction {
    @Override
    public void f() {

    }

    @Override
    public int g(int x) {
        return x;
    }

    @Override
    public void h(int z) {

    }
}
```

Дуже часто в програмі створюють посилання на інтерфейс, яке ініціалізують об’єктом класу. Такий підхід є належною практикою, оскільки він дозволяє легко замінити одну реалізацію інтерфейсу іншою.

Інтерфейси не походять від класу `java.lang.Object`. Не можна створювати новий об’єкт типу інтерфейсу. Навіть для порожнього інтерфейсу треба створити клас, який його реалізує.

Приклад

```
interface Empty {

}

class EmptyImplementation implements Empty {

}

...

public static void main(String[] args) {
    Empty empty1 = new Empty(); // Помилка
    Empty empty2 = new EmptyImplementation();
        // Коректне створення об'єкта
}
```

JDK надає велику кількість стандартних інтерфейсів. Розглянемо застосування інтерфейсів `Comparable` і `Comparator` для сортування масивів.

У найпростішому випадку сортування всього масиву за зростанням здійснюється за допомогою функції `sort()` з одним параметром – посиланням на відповідний масив. Статична функція `sort()` класу `java.util.Array` реалізована для масивів усіх примітивних типів. Аналогічно можна реалізувати сортування об’єктів класів, для яких визначене натуральне порівняння, тобто реалізований інтерфейс `Comparable`. Єдиний метод цього інтерфейсу - `compareTo()`:

```
public int compareTo(Object o)
```

Метод повинен повернути від’ємне значення (наприклад, -1), якщо об’єкт, для якого викликаний метод, менше об’єкта `o`, нульове значення, якщо об’єкти рівні, і додатне значення в протилежному випадку.

До класів, що реалізують інтерфейс `Comparable`, відносяться класи оболонки `Double`, `Integer`, `Long` і т.д., а також `String`. У такий спосіб можна розсортувати масив об’єктів типу Integer.

Приклад

```
public class SortIntegers {

    public static void main(String[] args) {
        Integer[] a = {7, 8, 3, 4, -10, 0};
        java.util.Arrays.sort(a);
        System.out.println(java.util.Arrays.toString(a));
    }

}
```

Примітка. Ім'я інтерфейсу `Comparable` – це приклад найбільш коректного імені інтерфейсу. Бажано, щоб імена інтерфейсів закінчувалися на `-able` (`Comparable`, `Runnable` тощо). Але це правило дуже часто порушується навіть для стандартних інтерфейсів.

У Java 5 `Comparable` – це узагальнений інтерфейс. Узагальнення в Java за своїм синтаксисом та використанням схожі на шаблони C++, але реалізовані повністю в інший спосіб. Створення та використання узагальнень буде розглянуто пізніше. Завдяки узагальненням у функціях, які оголошені в інтерфейсі, замість параметрів типу `Object`, можна використовувати параметри інших типів. В нашому випадку функція `compareTo()` повинна приймати аргумент типу елементу масиву.

Можна самостійно створити клас, що реалізує інтерфейс `Comparable`. У прикладі, масив прямокутників сортується за площею.

Приклад

```
class Rectangle implements Comparable<Rectangle> {
    private double width, height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public double area() {
        return width * height;
    }

    public double perimeter() {
        return 2 * (width + height);
    }
```

```
    @Override
    public int compareTo(Rectangle rect) {
        return Double.compare(area(), rect.area());
    }

    @Override
    public String toString() {
        return "[" + width + ", " + height + ", area = "
            + area() + ", perimeter = " + perimeter() + "]";
    }

}

public class SortRectangles {

    public static void main(String[] args) {
        Rectangle[] a = {new Rectangle(2, 7),
            new Rectangle(5, 3), new Rectangle(3, 4)};
        java.util.Arrays.sort(a);
        System.out.println(java.util.Arrays.toString(a));
    }

}
```

У наведеному прикладі використовується статична функція `compare()` класу `Double`. Ця функція повертає значення, необхідні методу `sort()`.

Якщо ми не хочемо (чи не можемо) визначити функцію `compareTo()`, можна створити клас, що реалізує інтерфейс `Comparator`. Посилання на об’єкт такого класу передаються в якості другого (четвертого) параметру функції `sort()`.

Приклад

```
// Опис функції у Java 2:
public static void sort(Object[] a, Comparator c)
public static void sort(
    Object[] a, int fromIndex, int toIndex, Comparator c)
```

Інтерфейс містить опис методу `compare()` з двома параметрами. Функція повинна повернути від’ємне число, якщо перший об’єкт під час сортування необхідно вважати меншим, чим інший, нульове значення, якщо об’єкти еквівалентні, і додатне число в протилежному випадку.

Примітка. Починаючи з Java 5 `Comparator` – це також узагальнений інтерфейс. Під час його реалізації після його імені слід вказувати в кутових дужках тип об’єктів, які ми

порівнюємо. Якщо використовувати узагальнення, функція `compare()` повинна приймати два аргументи типу параметра узагальнення.

Завдяки використанню класу, який реалізує інтерфейс `Comparator`, можна додатково здійснити сортування за периметром прямокутників.

Приклад

```
class CompareByPerimeter
                implements java.util.Comparator<Rectangle>
{

    @Override
    public int compare(Rectangle r1, Rectangle r2) {
        return Double.compare(r1.perimeter(),
               r2.perimeter());
    }
}

public class SortRectangles {

    public static void main(String[] args) {
        Rectangle[] a = {new Rectangle(2, 7),
                         new Rectangle(5, 3),
                         new Rectangle(3, 4)};
        java.util.Arrays.sort(a); // сортування за площею
        System.out.println(java.util.Arrays.toString(a));
        // Сортування за периметром:
        java.util.Arrays.sort(a, new CompareByPerimeter());
        System.out.println(java.util.Arrays.toString(a));
    }
}
```

3.6. Вкладені класи

3.6.1. Загальні концепції

Визначення класу може бути розміщене всередині іншого класу. В такий спосіб можуть бути створені вкладені класи, які можуть бути статичними вкладеними або внутрішніми. Вкладені класи можуть використовуватися як всередині обхопного класу, так і поза ним.

Приклад

```
class Outer {
    class Inner {
        int i;
    };
```

```
        Inner inner = new Inner();
    }

    class Another {
        Outer.Inner i;
    }
```

Вкладені класи можуть бути оголошені зі специфікаторами **public**, **private** або **protected**.

Локальні класи створюють всередині блоків. Існує також спеціальний різновид локальних класів – безіменні класи.

Окрема категорія – статичні вкладені класи, використання яких аналогічне вкладеним класам в C++ і C#.

3.6.2. Внутрішні класи

Нестатичні вкладені класи називають також внутрішніми. Головною відмінністю внутрішніх класів у Java є те, що об'єкти цих класів отримують посилання на об'єкт обхопного класу. З цього факту випливає два важливих висновки:

– об'єкти внутрішніх класів мають прямий доступ до даних об'єкта обхопного класу;

– для створення об'єкта внутрішнього класу обов'язково мати в наявності об'єкт обхопного класу.

У зв'язку з цим в Java запропонований спеціальний механізм створення об'єктів внутрішніх класів. Цей механізм проілюстрований на наведеному нижче прикладі.

Приклад

```
class Outer {
    int k = 100;

    class Inner {
        void show() {
            System.out.println(k);
        }
    }

}
```

```
public class Test {

    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.show();
    }

}
```

Нестатичні внутрішні класи не можуть містити статичних елементів.

Слід пам’ятати, що об’єкт внутрішнього класу автоматично не створюється. Створення об’єкта може бути передбачене в конструкторі чи у будь-якому методі обхопного класу, а також поза ним (якщо цей клас не оголошений як `private`). Можна також створити масив об’єктів внутрішнього класу. Кожен з таких об’єктів матиме доступ до посилання на обхопний об’єкт.

Внутрішні класи можуть мати свої базові класи. В такий спосіб за допомогою внутрішніх класів можна змоделювати відсутній у Java механізм множинного спадкування.

Приклад

```
class FirstBase {
    int a = 1;
}

class SecondBase {
    int b = 2;
}

class Outer extends FirstBase {
    int c = 3;

    class Inner extends SecondBase {
        void show() {
            System.out.println(a);
            System.out.println(b);
            System.out.println(c);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
```

```
        inner.show();
    }
}
```

Наведений приклад має суто теоретичний сенс, оскільки множинне успадкування класів незалежно від способів його реалізації є небезпечним з точки зору можливого конфлікту імен.

3.6.3. Локальні й безіменні класи

До внутрішніх класів також відносяться локальні. До таких класів не можна звернутися ззовні блоку, у якому вони визначені. Локальні класи найчастіше поміщають у тіло функції

Приклад

```
void f() {
    class Local {
        int j;
    }
    Local l = new Local();
    l.j = 100;
    System.out.println(l.j);
}
```

Можна також розміщати локальні класи всередині окремих блоків.

Безіменний клас може реалізовувати певний інтерфейс, перекривати абстрактні функції базового класу чи розширювати його. Для створення об’єкта безіменного класу здійснюється виклик конструктора базового класу, або вказується ім’я інтерфейсу з круглими дужками, після чого розташовують тіло безіменного класу.

Приклад

```
new Object() {
    // Додавання нового методу:
    void hello() {
        System.out.println("Привіт!");
    }
}.hello();

System.out.println(new Object() {
    // Перевизначення методу:
    @Override public String toString() {
        return "Це безіменний клас.";
```

```
    }
});
```

Безіменні класи не можуть бути абстрактними. Безіменний клас завжди є внутрішнім класом; він не може бути статичним. Безіменні класи автоматично є фінальними (**final**). У наведеному нижче прикладі безіменний клас створюється для визначення способу сортування масиву рядків.

Приклад

```
    void sortByABC(String[] a) {
        Arrays.sort(a, new Comparator<String>() {
            public int compare(String s1, String s2) {
                return (s1).compareTo(s2);
            }
        });
    }
```

У безіменних класів не може бути явних конструкторів. Разом з тим, завжди створюється усталений безіменний конструктор. Якщо в базового класу немає конструктора без параметрів, необхідні параметри конструктора вказуються в дужках під час створення об’єкта.

Приклад

```
abstract class Base {
    int k;

    Base(int k) {
        this.k = k;
    }

    abstract void show();
}

public class Test {

    static void showBase(Base b) {
        b.show();
    }

    public static void main(String[] args) {
        showBase(new Base(10) {
            void show() {
                System.out.println(k);
            }
        });
```

```
    }
}
```

Можна також використовувати блоки ініціалізації.

Для того, щоб безіменні класи мали доступ до локальних елементів зовнішніх блоків, ці елементи повинні бути описані як **`final`**.

3.6.4. Статичні вкладені класи

Статичні вкладені класи мають доступ тільки до статичних елементів обхопних класів. Об’єкти таких класів можуть бути створені без створення об’єктів обхопних класів.

Приклад

```
class Outer {
    int k = 100;
    static int m = 200;

    static class Inner {
        void show() {
            // k недоступно
            System.out.println(m);
        }
    }
}

public class Test {

    public static void main(String[] args) {
        Outer.Inner inner = new Outer.Inner();
        inner.show();
    }
}
```

Статичні вкладені класи можуть містити свої статичні елементи, у тому числі свої вкладені статичні і нестатичні класи.

Класи можна створювати всередині інтерфейсів. Такі класи автоматично є статичними. Всередині класів також можна створювати інтерфейси, що є також статичними.

3.7. Усталена реалізація методів інтерфейсів

Версія Java 8 надає нову можливість усталеної реалізації для методів, оголошених в інтерфейсі. Для цього перед відповідною функцією слід розмістити ключове слово **`default`**, після чого функцію можна реалізувати всередині інтерфейсу. Можна запропонувати інтерфейс, який представляє деяку функцію, а також реалізацію цієї функції як наведено у прикладі.

Приклад

```
public interface Greetings {
    default void hello() {
        System.out.println("Hello everybody!");
    }
}
```

Клас, який реалізує інтерфейс, може бути порожнім. Можна залишити усталену реалізацію методу `hello()`.

Приклад

```
public class MyGreetings implements Greetings {

}
```

Під час тестування отримаємо усталене вітання.

Приклад

```
public class GreetingsTest {

    public static void main(String[] args) {
        new MyGreetings().hello(); // Hello everybody!
    }

}
```

Те ж саме можна отримати, використавши безіменний клас. Його тіло також буде порожнім.

Приклад

```
public class GreetingsTest {

    public static void main(String[] args) {
        new Greetings() { }.hello(); // Hello everybody!
```

```
    }
}
```

Наявність методів з усталеною реалізацією робить інтерфейси ще більш схожими на абстрактні (і навіть на неабстрактні) класи. Але зберігається принципова відмінність: інтерфейс не можна безпосередньо застосовувати для створення об’єктів. Усі класи безпосередньо або опосередковано походять від базового типу `java.lang.Object`, який містить дані й функції, необхідні для функціонування всіх, навіть найпростіших об’єктів. Інтерфейси не є класами і не походять від `java.lang.Object`. Інтерфейс – це лише декларація певної поведінки, яка може бути доповнена допоміжними засобами (методами з усталеною реалізацією). Поля, описані в інтерфейсі – це не власне дані об’єкту, а константи часу компіляції. Для виконання методів з усталеною реалізацією необхідний об’єкт класу, який реалізує інтерфейс. Саме через це в останньому прикладі створюється об’єкт безіменного класу

```
new Greetings() { }.hello();
```

а не інтерфейсу

```
new Greetings().hello(); // Синтаксична помилка!
```

Метод з усталеною реалізацією можна перевизначити.

Приклад

```
public class MyGreetings implements Greetings {

    @Override
    public void hello() {
        System.out.println("Hello to me!");
    }

}
```

Тепер, створивши об’єкт цього класу, ми отримаємо нове привітання.

Якщо з перевизначеного методу необхідно викликати усталений метод інтерфейсу, можна скористатися ключовим словом **`super`**.

Приклад

```
Greetings.super.hello();
```

Можна запропонувати такий приклад. Припустимо, необхідно надрукувати значення функції на деякому інтервалі з визначеним кроком. Створюємо інтерфейс з одним абстрактним методом (обчислення деякої функції) і одним методом з усталеною реалізацією.

Приклад

```
public interface FunctionToPrint {
    public double f(double x);
    default void print(double x) {
        System.out.printf("x = %7f f(x) = %7f%n", x, f(x));
    }
}
```

У класі `PrintValues` створюємо метод друку таблиці `printTable()`. Цей метод використовує створений раніше інтерфейс.

Приклад

```
public class PrintValues {

    static void printTable(double from, double to,
                           double step, FunctionToPrint func) {
        for (double x = from; x <= to; x += step) {
            func.print(x);
        }
        System.out.println();
    }

    // у функції main() створюємо об'єкт безіменного класу:
    public static void main(String[] args) {
        printTable(-2, 2, 0.5, new FunctionToPrint() {
            @Override
            public double f(double x) {
                return x * x * x;
            }
        });
    }

}
```

Припустимо, нас не влаштувала точність значень. У цьому випадку в безіменному класі можна також перевизначити метод `print()`.

Приклад

```
public static void main(String[] args) {
    printTable(-2, 2, 0.5, new FunctionToPrint() {
        @Override
        public double f(double x) {
            return x * x * x;
        }

        @Override
        public void print(double x) {
            System.out.printf("x = %9f f(x) = %9f%n",x,f(x));
            }
        });
    }
```

Головна перевага інтерфейсів з усталеною реалізацією – можливість розширення інтерфейсів від версії до версії із забезпеченням збереження сумісності зі старим кодом. Припустимо, раніше в деякій бібліотеці був описаний інтерфейс:

```
public interface SomeInterface {
    void f();
}
```

Цей інтерфейс реалізовувався деяким класом:

```
public class OldImpl implements SomeInterface {
    @Override
    public void f() {
        // реалізація
    }
}
```

Тепер під час оновлення бібліотеки ми створили нову версію інтерфейсу, додавши в нього новий метод:

```
interface SomeInterface {
    void f();
    default void g() {
        // реалізація
    }
}
```

Цей метод буде реалізований новими класами, як наведено у прикладі.

Приклад

```
public class NewImpl implements SomeInterface {

    @Override
    public void f() {
        // реалізація
    }

    @Override
    public void g() {
        // реалізація
    }
}
```

Без усталеної реалізації не компілюватиметься код, побудований на попередній версії.

Під час успадкування інтерфейсу, який містить метод з усталеною реалізацією, цей метод також успадковується з реалізацією, проте його також можна знов оголосити та зробити абстрактним, або перевизначити і запропонувати іншу реалізацію.

У Java 8 інтерфейси також можуть містити реалізацію статичних методів. Логічно всередині інтерфейсу визначати методи, що мають відношення до цього інтерфейсу (наприклад, вони можуть одержувати посилання на інтерфейс як параметр). Найчастіше, це допоміжні методи. Як і всі елементи інтерфейсу, такі статичні методи є публічними. Можна вказати **public** явно, але в цьому немає необхідності.

У наведеному раніше прикладі функцію `printTable()` можна було б розмістити всередині інтерфейсу.

Приклад

```
public interface FunctionToPrint {
    public double f(double x);
    default void print(double x) {
        System.out.printf("x = %9f f(x) = %9f%n", x, f(x));
    }
    static void printTable(double from, double to,
            double step, FunctionToPrint func) {
        for (double x = from; x <= to; x += step) {
            func.print(x);
        }
        System.out.println();
```

```
        }
}
```

Виклик функції слід здійснювати через ім’я інтерфейсу.

3.8 .Робота з функціональними інтерфейсами в Java 8

3.8.1. Лямбда-вирази і функціональні інтерфейси

Дуже часто інтерфейси в Java містять оголошення однієї абстрактної функції (без усталеної реалізації). Такі інтерфейси отримали назву функціональних інтерфейсів. Їх повсюдно використовують для реалізації механізмів зворотного виклику, обробки подій і т.д. Не дивлячись на простоту, для їхньої реалізації, тим не менш, потрібен окремий клас – звичайний, вкладений або безіменний. Навіть використовуючи безіменний клас ми отримуємо громіздкий синтаксис, який погано читається. Скоротити необхідність безіменних класів у сирцевому коді дозволяють лямбда-вирази, які з’явилися у версії Java 8.

У мовах програмування є поняття функціонального об’єкта – об’єкта, який можна використовувати як функцію. Лямбда-вираз – це спеціальний синтаксис опису функціонального об’єкта всередині методу. Іншими словами, лямбда-вираз – це спосіб опису функції всередині іншої функції.

Термін "лямбда- вираз" пов’язаний з математичною дисципліною – лямбда-численням. Лямбда-числення – це формальна система, розроблена американським математиком Алонсо Черчем для формалізації й аналізу поняття обчислюваності. Лямбда-числення стало формальною основою мов функційного програмування (Lisp, Scheme тощо).

Лямбда-вираз у Java має такий синтаксис:

– список формальних параметрів, розділених комами й розміщених у круглих дужках; якщо параметр один, дужки можна опустити; якщо параметрів немає, потрібна порожня пара дужок;

– стрілка (->);

– тіло, що складається з одного виразу або блоку; якщо використовується блок, всередині нього може бути твердження **return**.

Приклад функції з одним параметром.

Приклад

```
k -> k * k
```

Той самий приклад з дужками та блоком.

Приклад

```
(k) -> { return k * k; }
```

Приклад функції з двома параметрами.

Приклад

```
(a, b) -> a + b
```

Приклад функції без параметрів.

Приклад

```
() -> System.out.println("First")
```

Приклад функціонального інтерфейсу.

Приклад

```
public interface SomeInt {
    int f(int x);
}
```

Під час виклику деякої функції потрібен параметр типу функціонального інтерфейсу. Традиційно можна створити безіменний клас.

Приклад

```
someFunc(new SomeInt() {
    @Override
    public int f(int x) {
        return x * x;
    }
});
```

Можна створити змінну типу об’єкта, що реалізує інтерфейс, і використовувати її замість безіменного класу.

Приклад

```
SomeInt func = k -> k * k;
someFunc(func);
```

Можна також створити безіменний об’єкт під час виклику функції з параметром-функціональним інтерфейсом.

Приклад

```
someFunc(x -> x * x);
```

Оскільки кожен лямбда-вираз пов’язаний з певним функціональним інтерфейсом, типи параметрів і результату визначаються автоматично через зіставлення з відповідним функціональним інтерфейсом.

Програму в прикладі з таблицею значень функції можна реалізувати з використанням лямбда-виразів. Маємо попередньо створений інтерфейс. Він є функціональним, оскільки в ньому оголошено саме один абстрактний метод.

Приклад

```
public interface FunctionToPrint {
    public double f(double x);
    default void print(double x) {
        System.out.printf("x = %9f f(x) = %9f%n", x, f(x));
    }
    static void printTable(double from, double to,
            double step, FunctionToPrint func) {
        for (double x = from; x <= to; x += step) {
            func.print(x);
        }
        System.out.println();
    }
}
```

Використання функціонального інтерфейсу з застосуванням лямбда-виразу.

Приклад

```
public class PrintWithLambda {
    public static void main(String[] args) {
        FunctionToPrint.printTable(-2.0, 2.0, 0.5,
                                   x -> x * x * x);
    }
}
```

3.8.2. Використання посилань на методи

Дуже часто все тіло лямбда-виразу складається лише з виклику існуючого методу. У цьому випадку замість лямбда-виразу можна використовувати посилання на цей метод. Існує кілька варіантів опису посилань на методи.

Наприклад, є такі функціональні інтерфейси.

Приклад

```
interface IntOperation {
    int f(int a, int b);
}

interface StringOperation {
    String g(String s);
}
```

Можна створити деякий клас.

Приклад

```
class DifferentMethods
{
    public int add(int a, int b) {
        return a + b;
    }

    public static int mult(int a, int b) {
        return a * b;
    }
}
```

Викликаємо методи.

Приклад

```
public class TestMethodReferences {

    static void print(IntOperation op, int a, int b) {
        System.out.println(op.f(a, b));
    }
```

```
    static void print(StringOperation op, String s) {
        System.out.println(op.g(s));
    }

    public static void main(String[] args) {
        DifferentMethods dm = new DifferentMethods();
        print(dm::add, 3, 4);
        print(DifferentMethods::mult, 3, 4);
        print(String::toUpperCase, "text");
    }

}
```

3.8.3. Стандартні функціональні інтерфейси

Замість того, щоб створювати нові функціональні інтерфейси, в більшості випадків достатньо скористатися стандартними узагальненими інтерфейсами, які описані в пакеті `java.util.function`.

Крім перелічених, функціональними інтерфейсами також є узагальнений інтерфейс `Comparator`, інтерфейс `Runnable`, який використовують у багатопотоковому програмуванні, а також багато інших.

3.8.4. Композиція лямбда-виразів

Можна здійснювати композицію лямбда-виразів (використовувати лямбда-вирази як параметри). З цією метою інтерфейси пакету `java.util.function` надають методи з усталеною реалізацією, що забезпечують виконання деякої функції, переданої як параметр до або після даного методу. Зокрема, в інтерфейсі `Function` визначені такі методи:

```
// Виконується функція before, а потім функція, що викликає:
Function compose(Function before)
// Функція after виконується після функції, що викликає:
Function andThen(Function after)
```

Використання цих методів та їх відмінність розглянемо на такому прикладі. Є клас зі статичною функцією `calc()`, що приймає функціональний інтерфейс і аргумент типу `Double`. Можна здійснити композицію лямбда-виразів.

Приклад

```
import java.util.function.Function;

public class ComposeDemo {

    public static Double calc(
            Function<Double , Double> operator, Double x) {
        return operator.apply(x);
    }

    public static void main(String[] args) {
        Function<Double , Double> addTwo = x -> x + 2;
        Function<Double , Double> duplicate = x -> x * 2;
        System.out.println(calc(addTwo.compose(duplicate),
            10.0)); // 22.0
        System.out.println(calc(addTwo.andThen(duplicate),
            10.0)); // 24.0
    }

}
```

Композиція може бути більш складною.

Приклад

```
System.out.println(calc(addTwo.andThen(duplicate).andThen(
    addTwo), 10.0)); // 26.0
```

3.9. Клонування об’єктів і перевірка еквівалентності

Іноді виникає необхідність в створенні копії деякого об’єкта, наприклад, для виконання з копією дій, що не порушують даних про оригінал. Просте присвоювання призводить тільки до копіювання посилань. Якщо нам необхідно поелементно скопіювати деякий об’єкт, необхідно використовувати механізм так званого клонування.

У базовому класі `java.lang.Object` є функція `clone()`, усталене використання якої дозволяє скопіювати об’єкт поелементно. Ця функція також визначена для масивів, рядків і інших стандартних класів. Можна отримати копію існуючого масиву і працювати з цією копією як наведено у прикладі.

Приклад

```
import java.util.Arrays;

public class ArrayClone {
```

```
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4 };
        int[] a2 = a1.clone(); // Копія елементів
        System.out.println(Arrays.toString(a2)); // [1,2,3,4]
        a1[0] = 10; // змінюємо перший масив
        System.out.println(Arrays.toString(a1));
            // [10,2,3,4]
        System.out.println(Arrays.toString(a2)); // [1,2,3,4]
    }

}
```

Для того, щоб можна було клонувати об'єкти користувальницьких класів, ці класи повинні реалізовувати інтерфейс `Cloneable`. Цей інтерфейс не оголошує жодного методу. Він всього лише вказує, що об'єкти даного класу можна клонувати. В іншому випадку виклик функції `clone()` призведе до генерації винятку типу `CloneNotSupportedException`.

Примітка. Механізм обробки винятків багато в чому схожий на відповідний механізм мови C++. У Java в заголовку методів, які генерують винятки, слід перелічувати можливі винятки за допомогою ключового слова **`throws`**. Механізм обробки винятків буде розглянуто пізніше.

Припустимо, нам потрібно клонувати об'єкти класу `Human`, що включає два поля типу `String` – `name` і `surname`. Додаємо до опису класу реалізацію інтерфейсу `Cloneable`, генеруємо конструктор з двома параметрами, для зручності виведення вмісту полів перекриваємо функцію `toString()`. У функції `main()` здійснюємо тестування клонування об'єкта.

Приклад

```
public class Human implements Cloneable {
    private String name;
    private String surname;

    public Human(String name, String surname) {
        super();
        this.name = name;
        this.surname = surname;
    }

    @Override
    public String toString() {
        return name + " " + surname;
    }
```

```
    public static void main(String[] args)
                            throws CloneNotSupportedException {
        Human human1 = new Human("John", "Smith");
        Human human2 = (Human) human1.clone();
        System.out.println(human2); // John Smith
        human1.name = "Mary";
        System.out.println(human1); // Mary Smith
        System.out.println(human2); // John Smith
    }

}
```

Як видно з прикладу, після клонування у вихідний об’єкт можна вносити зміни. При цьому копія не зміниться.

Для зручності використання функції `clone()` її можна перекрити, змінивши її тип результату і зробивши відкритою. Завдяки наявності цієї функції спроститься клонування (не потрібно буде кожен раз приводити тип).

Приклад

```
@Override
public Human clone() throws CloneNotSupportedException {
    return (Human) super.clone();
}
. . .
Human human2 = human1.clone();
```

Стандартне клонування, реалізоване в класі `java.lang.Object`, дозволяє створювати копії об’єктів, поля яких – типи значення і тип `String` (а також класи-обгортки). Якщо поля об’єкта – посилання на масиви або інші типи, необхідно застосовувати так зване "глибоке" клонування. Припустимо, певний клас `SomeCloneableClass` містить два поля типу `double` та масив цілих. "Глибоке" клонування забезпечить створення окремих масивів для різних об’єктів.

Приклад

```
import java.util.Arrays;

public class SomeCloneableClass implements Cloneable {
    private double x, y;
    private int[] a;
```

```
    public SomeCloneableClass(double x, double y, int[] a) {
        super();
        this.x = x;
        this.y = y;
        this.a = a;
    }

    @Override
    protected SomeCloneableClass clone()
              throws CloneNotSupportedException {
        SomeCloneableClass scc = // копіюємо x і y
            (SomeCloneableClass) super.clone();
        scc.a = a.clone();
            // тепер два об'єкти працюють з різними масивами
        return scc;
    }

    @Override
    public String toString() {
        return " x=" + x + " y=" + y + " a=" +
            Arrays.toString(a);
    }

    public static void main(String[] args)
                           throws CloneNotSupportedException {
        SomeCloneableClass scc1 = new SomeCloneableClass(
                            0.1, 0.2, new int[] { 1, 2, 3 });
        SomeCloneableClass scc2 = scc1.clone();
        scc2.a[2] = 4;
        System.out.println("scc1:" + scc1);
        System.out.println("scc2:" + scc2);
    }

}
```

Для того, щоб переконатися, що клоновані об’єкти однакові, не завадило б мати можливість автоматичного порівняння всіх полів. Посилальна модель об’єктів Java не дозволяє порівнювати вміст об’єктів за допомогою операції порівняння (==), оскільки при цьому порівнюються посилання. Для порівняння даних доцільно використовувати функцію `equals()`, визначену в класі `java.lang.Object`. Для класів, полями яких є типи-значення, метод класу `Object` забезпечує поелементне порівняння. Якщо ж полями є посилання на об’єкти, необхідно явно перевизначити функцію `equals()`. Типова реалізація методу `equals()` передбачає перевірку посилань (чи вони збігаються), далі перевірку об’єкта, який ми порівнюємо, на значення **`null`**, потім – перевірку

типу, наприклад, за допомогою **instanceof**. Якщо типи збігаються, здійснюється перевірка значень полів.

Наведемо повний приклад з класом `Human`.

Приклад

```
public class Human implements Cloneable {
    private String name;
    private String surname;

    public Human(String name, String surname) {
        super();
        this.name = name;
        this.surname = surname;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null || !(obj instanceof Human)) {
            return false;
        }
        Human h = (Human) obj;
        return name.equals(h.name)
                && surname.equals(h.surname);
    }

    @Override
    public Human clone() throws CloneNotSupportedException {
        return (Human) super.clone();
    }

    @Override
    public String toString() {
        return name + " " + surname;
    }

    public static void main(String[] args)
           throws CloneNotSupportedException {
        Human human1 = new Human("John", "Smith");
        Human human2 = human1.clone();
        System.out.println(human2);
        human1.name = "Mary";
        System.out.println(human1);
        System.out.println(human2);
        human2.name = new String("Mary");
        System.out.println(human2);
        System.out.println(human1.equals(human2)); // true
    }
```

```
}
```

Якби метод `equals()` не було визначено, останнє порівняння дало б **`false`**.

Для порівняння двох масивів доцільно викликати статичну функцію `equals()` класу `Arrays`. Ця функція порівнює елементи масивів (викликає метод `equals()`):

```
Arrays.equals(array1, array2);
```

Вправи для самостійної роботи

1. Створити ієрархію класів Книга та Підручник. Реалізувати конструктори та функції доступу. Перекрити функцію `toString()`. У функції `main()` створити масив, який містить елементи різних типів. Вивести елементи на екран.

2. Створити ієрархію класів Місто та Столиця. Реалізувати конструктори та функції доступу. Перекрити функцію `toString()`. У функції `main()` створити масив, який містить елементи різних типів. Вивести елементи на екран.

3. Створити клас для представлення іменованої матриці з полем типу String – іменем матриці і полем, що представляє двовимірний масив. Реалізувати методи клонування, перевірки еквівалентності та отримання подання у вигляді рядку.

Контрольні запитання

1. У чому полягає сенс успадкування?

2. Де і для чого можна застосовувати ключове слово **`super`**?

3. Чи допускається множинне успадкування класів?

4. Які можливості надає використання поліморфізму?

5. Чим віртуальна функція відрізняється від невіртуальної?

6. Як у Java вказати, що функція віртуальна?

7. У чому перевага інтерфейсів у порівнянні з абстрактними класами?

8. Чи допускається множинне успадкування інтерфейсів?

9. Яким вимогам повинен відповідати клас, який реалізує інтерфейс?

10. Чи може клас реалізовувати кілька інтерфейсів?

11. Чим відрізняються статичні вкладені класи від внутрішніх?

12. Чому не можна створити явний конструктор безіменного класу?

13. Що таке лямбда-вираз?

14. Що таке функціональний інтерфейс?

15. Для чого використовуються посилання на методи?

4. УЗАГАЛЬНЕННЯ ТА КОЛЕКЦІЇ

4.1. Узагальнення (Generics)

4.1.1. Концепція узагальненого програмування

Часто виникає необхідність у створенні так званих класів-контейнерів – таких, які містять об’єкти довільних типів. При цьому над елементами контейнерів необхідно виконувати деякі однотипні дії. Код для обробки об’єктів різних типів може виглядати практично однаково. Наприклад, якщо для різних типів даних потрібно реалізувати алгоритми на кшталт швидкого сортування або способи обробки таких структур даних, як зв’язаний список або бінарне дерево. У таких випадках код однаковий для всіх типів об’єктів.

Парадигма узагальненого програмування передбачає опис правил зберігання даних і алгоритмів у загальному вигляді незалежно від конкретних типів даних. Конкретні типи даних, над якими виконуються дії, специфікуються пізніше. Механізми розділення структур даних і алгоритмів, а також формування абстрактних описів вимог до даних, визначаються по-різному в різних мовах програмування. Спочатку можливості узагальненого програмування були представлені в сімдесяті роки XX століття мовами CLU і Ада (узагальнені функції), пізніше були реалізовані в мові ML (параметричний поліморфізм).

Найбільш повно і гнучко ідея узагальненого програмування реалізована у мові C++ через механізм шаблонів. Шаблон (template) у C++ – це фрагмент коду, який узагальнено описує роботу з деяким абстрактним типом, заданим як параметр шаблону. Цей фрагмент коду (клас або функція) остаточно компілюється тільки після інстанціювання шаблону конкретним типом, тобто після підстановки конкретного типу замість параметра. На використанні шаблонних функцій і параметризованих класів побудована Стандартна бібліотека шаблонів (STL), що включає опис стандартних контейнерних класів і незалежних від них алгоритмів.

Для реалізації узагальненого програмування в Java використовуються узагальнення – спеціальна мовна конструкція, яка з'явилася у синтаксисі мови починаючи з версії Java 5.

4.1.2. Проблеми створення універсальних контейнерів у Java 2

Припустимо, нам необхідно створити контейнер для зберігання пари об'єктів одного типу. Можна запропонувати клас `Pair` (пара). Він містить два посилання на клас `Object`.

Приклад

```
public class Pair {
    Object first, second;

    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }

}
```

Оскільки клас `Object` є базовим для усіх типів-посилань, новий клас можна застосувати для зберігання пари рядків.

Приклад

```
Pair p = new Pair("Прізвище", "Ім\'я");
```

Такий підхід має певні недоліки:

– для читання об'єктів необхідно застосувати явне перетворення типів:

```
String s = (String) p.first; // Замість String s = p.first;
```

– немає впевненості, що у парі зберігаються об'єкти саме того типу, який нас цікавить:

```
Integer i = (Integer) p.second; // Помилка часу виконання
```

– не можна гарантувати, що обидва поля будуть одного типу:

```
// Жодного повідомлення про помилку:
Pair p1 = new Pair("Прізвище", new Integer(2));
```

Аналогічні проблеми в Java 2 виникали зі стандартними контейнерними класами. Наслідком реалізованого таким чином підходу стали потенційні помилки часу виконання, які не могли бути знайдені під час компіляції коду.

4.1.3. Синтаксис узагальнень

Зазначені раніше проблеми розв'язує синтаксична конструкція Java 5, так зване узагальнення – конструкція, що включає в себе параметр класу або функції, який містить додаткову інформацію про тип елементів та інших даних. Цей параметр беруть у кутові дужки. Узагальнення надають можливість створення та використання структур даних, безпечних з точки зору типів. Класи, опис яких містить такий параметр, мають назву узагальнених. Під час створення об'єкта узагальненого типу у кутових дужках вказують імена реальних типів. Можна використовувати тільки типи-посилання. Попередній приклад можна реалізувати із застосуванням узагальнень, як наведено нижче.

Приклад

```
public class Pair<T> {
    T first, second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public static void main(String[] args) {
        Pair<String> p = new Pair<String>
            ("Прізвище", "Ім\'я");
        String s = p.first;
            // Отримуємо рядок без приведення типів
        Pair<Integer> p1 = new Pair<Integer>(1, 2);
            // Можна використовувати цілі константи
        int i = p1.second;
            // Отримуємо ціле значення без приведення типів
    }
}
```

Примітка. Java версії 7 і вище дозволяє не повторювати фактичний параметр узагальнення після імені конструктора.

Приклад

```
Pair<Integer> p1 = new Pair<>(1, 2);
```

Якщо ми намагаємось додати до пари дані різних типів, компілятор згенерує помилку. Помилковою є також спроба явно перетворити тип.

Приклад

```
Pair<String> p = new Pair<String>("1", "2");
Integer i = (Integer) p.second; // Помилка компіляції
```

Тип даних з параметром у кутових дужках (наприклад, `Pair<String>`) має назву параметризованого типу.

Узагальнення по зовнішньому представленню і використанню аналогічні шаблонам C++. Але на відміну від шаблонів C++, існує не декілька різних типів `Pair`, а один. Фактично у полях класу зберігаються посилання на `Object`. Інформація про тип параметрів використовується компілятором для контролю та автоматичного приведення типів у вихідному тексті.

Окрім узагальнених класів, можна створювати узагальнені інтерфейси. Параметр може бути використаний в описі функцій, оголошених в інтерфейсі. Інтерфейси можуть бути реалізовані як узагальненими, так і неузагальненими класами. Під час їх реалізації замість параметра узагальнення використовують деякий тип-посилання.

Приклад

```
interface Function<T> {
    T func(T x);
}

class DoubleFunc implements Function<Double> {

    @Override
    public Double func(Double x) {
        return x * 1.5;
    }
}

class IntFunc implements Function<Integer> {

    @Override
    public Integer func(Integer x) {
        return x % 2;
    }
}
```

Узагальнені класи та інтерфейси в Java не є ані новими типами, ані шаблонами. Це лише механізм, який вказує компілятору на необхідність подальшої перевірки типів і додавання перетворення типів.

Java також дозволяє створювати узагальнені функції всередині як узагальнених, так і звичайних (неузагальнених) класів.

Приклад

```
public class ArrayPrinter {
    public static<T> void printArray(T[] a) {
        for (T x : a) {
            System.out.print(x + "\t");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        String[] as = {"First", "Second", "Third"};
        printArray(as);
        Integer[] ai = {1, 2, 4, 8};
        printArray(ai);
    }
}
```

Як видно з прикладу, виклик узагальненої функції не вимагає явного визначення типу. Іноді таке визначення необхідне, наприклад, коли в функції немає параметрів узагальненого типу. Якщо це статична функція, необхідно явно вказувати її клас.

Приклад

```
public class TypeConverter {
    public static <T>T convert(Object object) {
        return (T) object;
    }

    public static void main(String[] args) {
        Object o = "Some Text";
        String s = TypeConverter.<String>convert(o);
        System.out.println(s);
    }
}
```

Рекомендованими іменами формальних параметрів є імена з однієї великої літери. Узагальнення може мати два і більше параметрів. У наведеному нижче прикладі пара може містити посилання на об’єкти різних типів.

Приклад

```
public class PairOfDifferentObjects<T, E> {
    T first;
    E second;

    public PairOfDifferentObjects(T first, E second) {
        this.first = first;
        this.second = second;
    }

    public static void main(String[] args) {
        PairOfDifferentObjects<Integer, String> p =
            new PairOfDifferentObjects<Integer,
                String>(1000, "thousand");
        PairOfDifferentObjects<Integer, Integer> p1 =
            new PairOfDifferentObjects<Integer,
                Integer>(1, 2);
        //...
    }
}
```

Над даними типу параметру узагальнення можна здійснювати тільки дії, дозволені для об’єктів класу `Object`. Іноді для розширення функціональності бажаною є конкретизація типу. Наприклад, ми хочемо викликати методи, оголошені у певному класі або інтерфейсі. Тоді можна застосувати такий синтаксис опису параметру: `<T extends SomeBaseType>` або `<T extends FirstType & SecondType>` тощо. Слово **`extends`** використовують як для класів, так і для інтерфейсів.

Наприклад, можна створити узагальнену функцію обчислення середнього арифметичного в масиві деяких числових значень. Стандартні класи `Double`, `Float`, `Integer`, `Long` та інші класи-обгортки числових даних мають загальний абстрактний базовий клас `java.lang.Number`, що декларує, зокрема, метод `doubleValue()`, який дозволяє, отримати число, що зберігається в об’єкті, у вигляді значення типу **`double`**. Цей факт можна використовувати для

обчислення середнього арифметичного. Створена функція може працювати з масивами чисел різних типів.

Приклад

```
public class AverageTest {

    public static<E extends Number> double average(E[] arr) {
        double result = 0;
        for (E elem : arr) {
            result += elem.doubleValue();
        }
        return result / arr.length;
    }

    public static void main(String[] args) {
        Double[] doubles = { 1.0, 1.1, 1.5 };
        System.out.println(average(doubles)); // 1.2
        Integer[] ints = { 10, 20, 3, 4 };
        System.out.println(average(ints));    // 9.25
    }
}
```

Синтаксис узагальнень передбачає використання так званих масок (wildcard, символ '?'). Маска застосовується, наприклад, для опису посилань на поки невідомий тип. Використання масок робить узагальнені класи та функції більш сумісними. Маска надає альтернативний спосіб створення узагальнених функцій. Такі функції не є узагальненими, але містять аргументи узагальнених типів.

Наведемо приклад створення узагальненого класу, який зберігає масив елементів певного типу. Можна також додати статичну функцію виведення на консоль елементів масиву довільного типу, яка демонструє використання масок.

Приклад

```
public class MyArray<T> {
    private T[] arr;

    public MyArray(T... arr) {
        this.arr = arr;
    }

    public int size() {
```

```
        return arr.length;
    }

    public T get(int i) {
        return arr[i];
    }

    public void set(int i, T t) {
        arr[i] = t;
    }

    public static void printGenericArray(MyArray<?> a) {
        for (int i = 0; i < a.size(); i++) {
            System.out.print(a.get(i) + "\t");
        }
        System.out.println();
    }

}
```

В іншому класі здійснюємо тестування.

Приклад

```
public class MyArrayTest {
    public static void main(String[] args) {
        MyArray<String> arr1 = new MyArray<>
            ("First", "Second", "Third");
        MyArray.printGenericArray(arr1);
        MyArray<?> arr2 = new MyArray<>(1, 2, 3);
        MyArray.printGenericArray(arr2);
    }
}
```

Не можна створювати масиви об’єктів узагальнених типів.

Приклад

```
T arr = new T[10]; // Помилка!
```

У нашому прикладі цю проблему можна розв’язати за допомогою посилань на клас `Object`. Корисною також буде функція додавання елемента в кінець масиву і видалення останнього елемента. Замість використання статичної функції `printGenericArray()` можна перевизначити метод `toString()`. Альтернативна реалізація може бути такою.

Приклад

```
import java.util.Arrays;

public class MyArray<T> {
    private Object[] arr = {};

    public MyArray(T... arr) {
        this.arr = arr;
    }

    public MyArray(int size) {
        arr = new Object[size];
    }

    public int size() {
        return arr.length;
    }

    public T get(int i) {
        return (T)arr[i];
    }

    public void set(int i, T t) {
        arr[i] = t;
    }

    public void add(T t) {
        Object[] temp = new Object[arr.length + 1];
        System.arraycopy(arr, 0, temp, 0, arr.length);
        arr = temp;
        arr[arr.length - 1] = t;
    }

    public void remove(int i) {
        Object[] temp = new Object[arr.length - 1];
        System.arraycopy(arr, 0, temp, 0, i);
        System.arraycopy(arr, i + 1, temp, i, arr.length-i-
1);
        arr = temp;
    }

    @Override
    public String toString() {
        return Arrays.toString(arr);
    }
}
```

В новому класі здійснюємо тестування.

Приклад

```
public class TestClass {

    public static void main(String[] args) {
        MyArray<String> a = new MyArray<>("1", "2");
```

```
        String s = a.get(a.size() - 1);
        System.out.println(s);      // 2
        a.set(1, "New");
        System.out.println(a);      // 1 New
        MyArray<Double> b = new MyArray<>(3);
        b.set(0, 1.0);
        b.set(1, 2.0);
        b.set(2, 4.0);
        b.remove(2);
        b.add(8.0);
        System.out.println(b);      // [1.0, 2.0, 8.0]
    }

}
```

Функціональність класу можна розширити методами додавання нового елемента всередині масиву, видалення всіх елементів тощо.

Можна обмежити використання типу параметра функції певними похідними класами, наприклад, `MyArray<? super String>`. Тоді використання списку `MyArray<Integer>` неможливе.

4.2. Контейнерні класи та інтерфейси. Робота зі списками

4.2.1. Загальні відомості

Засоби Java для роботи з колекціями надають уніфіковану архітектуру для представлення та управління наборами даних. Ця архітектура дозволяє працювати з колекціями незалежно від деталей їхньої внутрішньої організації. Засоби для роботи з колекціями включають більше десятка інтерфейсів, а також стандартні реалізації цих інтерфейсів і набір алгоритмів для роботи з ними.

Колекція – це об’єкт, який представляє групу об’єктів. Використання колекцій забезпечує підвищення ефективності програм завдяки використанню високоефективних алгоритмів, а також сприяє створенню коду, придатного для повторного використання. Основними елементами засобів роботи з колекціями є такі:

– інтерфейси;

– стандартні реалізації інтерфейсів;

– алгоритми;

– утиліти для роботи з масивами.

Крім того, Java 8 підтримує контейнери Java 1.1. Це `Vector`, `Enumeration`, `Stack`, `BitSet` і деякі інші. Наприклад, клас `Vector` надає функціональність, аналогічну `ArrayList`. У першій версії Java ці контейнери не забезпечували стандартизованого інтерфейсу. Вони також не дозволяють користувачеві відмовитися від надмірної синхронізації, яка актуальна лише в багатопотоковому оточенні, і отже, недостатньо ефективні. Внаслідок цього вони вважаються застарілими і не рекомендовані для використання. Замість них слід використовувати відповідні узагальнені контейнери Java 5.

Стандартні контейнерні класи Java дозволяють зберігати в колекції посилання на об'єкти, класи яких походять від класу `Object`. Контейнерні класи реалізовані в пакеті java.util. Починаючи з Java 5, усі контейнерні класи реалізовані як узагальнені.

Існує два базових інтерфейси, в яких декларована функціональність контейнерних класів: `Collection` (похідний від `Iterable`) і `Map`. Інтерфейс `Collection` є базовим для інтерфейсів `List` (список), `Set` (множина), `Queue` (черга) і `Deque` (черга з двома кінцями).

Інтерфейс `List` (список) описує пронумеровану колекцію (послідовність), елементи якої можуть повторюватися.

Інтерфейс `Set` реалізований класами `HashSet` і `LinkedHashSet`. Інтерфейс `SortedSet`, похідний від `Set`, реалізований класом `TreeSet`. Інтерфейс `Map` реалізований класом `HashMap`. Інтерфейс `SortedMap`, похідний від `Map`, реалізований класом `TreeMap`.

Класи `HashSet`, `LinkedHashSet` і `HashMap` використовують для ідентифікації елементів так звані хеш-коди. Хеш-код – це унікальна послідовність бітів фіксованої довжини. Для кожного об'єкта ця послідовність вважається унікальною. Хеш-коди забезпечують швидкий доступ до даних за деяким ключем. Усі об'єкти Java можуть генерувати хеш-коди: метод `hashCode()` визначений для класу `Object`.

Для більшості колекцій існують як "звичайні" реалізації, так і реалізації, безпечні з точки зору потоків управління, наприклад `CopyOnWriteArrayList`, `ArrayBlockingQueue` тощо.

4.4.2. Інтерфейс Collection

Інтерфейс `Collection` є базовим для багатьох інтерфейсів `Collection` Framework і оголошує найбільш загальні методи колекцій, наведені в табл. 4.1.

Таблиця 4.1

Метод	Опис
`int size()`	Повертає розмір колекції
`boolean isEmpty()`	Повертає **`true`**, якщо колекція порожня
`boolean contains(Object o)`	Повертає **`true`**, якщо колекція містить об’єкт
`Iterator<E> iterator()`	Повертає ітератор – об’єкт, який послідовно вказує на елементи
`Object[] toArray()`	Повертає масив посилань на `Object`, який містить копії всіх елементів колекції
`<T> T[] toArray(T[] a)`	Повертає масив посилань на `T`, який містить копії всіх елементів колекції
`boolean add(E e)`	Додає об’єкт у колекцію. Повертає **`true`**, якщо об’єкт доданий
`boolean remove(Object o)`	Видаляє об’єкт з колекції
`boolean containsAll(Collection<?> c)`	Повертає **`true`** якщо колекція містить іншу колекцію
`boolean addAll(Collection<? extends E> c)`	Додає об’єкти в колекцію. Повертає **`true`**, якщо об’єкти додані
`boolean removeAll(Collection<?> c)`	Видаляє об’єкти з колекції
`boolean retainAll(Collection<?> c)`	Залишає об’єкти, присутні в іншій колекції
`void clear()`	Видаляє всі елементи з колекції

Примітка. В таблиці не вказані методи з усталеною реалізацією, які були додані у Java 8.

Наведений нижче приклад демонструє роботу методів інтерфейсу. В прикладі використовується клас `ArrayList` як найпростіша реалізація інтерфейсу `Collection`.

Приклад

```
import java.util.*;

public class CollectionDemo {

    public static void main(String[] args) {
        Collection<Integer> c =
            new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
        System.out.println(c.size());      // 5
        System.out.println(c.isEmpty());   // false
        System.out.println(c.contains(4)); // true
        c.add(6);
        System.out.println(c); // [1, 2, 3, 4, 5, 6]
        c.remove(1);
        System.out.println(c); // [2, 3, 4, 5, 6]
        Collection<Integer> c1 =
            new ArrayList<>(Arrays.asList(3, 4));
        System.out.println(c.containsAll(c1));// true
        c.addAll(c1);
        System.out.println(c);  // [2, 3, 4, 5, 6, 3, 4]
        Collection<Integer> c2 = new ArrayList<>(c); // копія
        c.removeAll(c1);
        System.out.println(c);  // [2, 5, 6]
        c2.retainAll(c1);
        System.out.println(c2); // [3, 4, 3, 4]
        c.clear();
        System.out.println(c);  // []
    }
}
```

4.2.3. Робота зі списками

Інтерфейс `List` описує упорядковану колекцію (послідовність). Цей інтерфейс реалізують стандартні класи `ArrayList` і `LinkedList`. Клас `ArrayList` реалізує список за допомогою масиву змінної довжини. Клас `LinkedList` зберігає об’єкти за допомогою так званого зв’язаного списку.

Існує також стандартна абстрактна реалізація списку – клас `AbstractList`. Цей клас, похідний від `AbstractCollection`, надає низку корисних засобів. Практично в тій чи іншій формі реалізовані всі методи, крім `get()` і `size()`. Проте, для конкретної реалізації списку більшість функцій

слід перекрити. Клас `AbstractList` – базовий для `ArrayList`. Клас `AbstractSequentialList`, похідний від `AbstractList`, є базовим для класу `LinkedList`.

Створити порожній список посилань на об'єкти деякого типу (`SomeType`) можна за допомогою усталеного конструктора.

Приклад

```
List<SomeType> al = new ArrayList<SomeType>();
```

Можна також одразу описати посилання на `ArrayList`.

Приклад

```
ArrayList<SomeType> al = new ArrayList<SomeType>();
```

Другий варіант іноді не є бажаним, оскільки в такому випадку знижується гнучкість програми. Перший варіант дозволить легко замінити реалізацію списку `ArrayList` на будь-яку іншу реалізацію інтерфейсу `List`, яка більше відповідає вимогам конкретної задачі. У другому випадку є спокуса викликати методи, специфічні для `ArrayList`, тому перехід на іншу реалізацію буде ускладнено.

Об'єкт класу `ArrayList` містить масив `elementData` елементів типу `Object`. Фізичний розмір масиву (capacity), якщо не викликати конструктор з явним зазначенням цього розміру, усталено дорівнює 10. Кожне додавання елемента передбачає виклик внутрішнього методу `ensureCapacity()`, який в разі заповнення масиву здійснює створення нового масиву з копіюванням в нього наявних елементів. Розмір нового масиву обчислюється за формулою (старий_розмір * 3) / 2 + 1.

Під час видалення елементів фізичний розмір масиву не зменшується. Для заощадження пам'яті після багаторазового видалення елементів доцільно викликати метод `trimToSize()`.

Створивши порожній список, у нього можна додавати елементи за допомогою функції `add()`. Метод `add()` з одним аргументом типу-посилання

додає елемент у кінець списку. Якщо цю функцію викликати з двома аргументами, новий елемент буде додано в список у позиції, зазначеній першим параметром.

Приклад

```
List<String> al = new ArrayList<String>();
al.add("abc");
al.add("def");
al.add("xyz");
al.add(2, "ghi"); // Додавання нового рядка перед "xyz"
```

До списку можна додати всі елементи іншого списку (або іншої колекції) за допомогою функції `addAll()`.

Можна створити новий список із використанням існуючого. Новий список містить посилання на копії елементів.

Приклад

```
List<String> a11 = new ArrayList<String>(al);
```

За допомогою статичної функції `asList()` класу `java.util.Arrays` можна створити список з існуючого масиву. Масив можна також створити безпосередньо у списку параметрів функції.

Приклад

```
String[] arr = {"one", "two", "three"};
List<String> a2 = Arrays.asList(arr);
List<String> a3 = Arrays.asList("four", "five");
```

Для списків перевантажена функція `toString()` дозволяє, наприклад, вивести всі елементи масиву на екран без використання циклів. Елементи виводяться у квадратних дужках.

Приклад

```
System.out.println(a3); // [four, five]
```

Списки дозволяють роботу з окремими елементами. Метод `size()` повертає кількість елементів, що містяться в списку. Як і в масивах, доступ до елементів списків може здійснюватися за індексом, але не через операцію `[]`, а

за допомогою методів `get()` і `set()`. Метод `get()` класу `ArrayList` повертає елемент із зазначеним індексом (позиція в списку). Як і елементи масивів, елементи списків пронумеровані з нуля. У наведеному нижче прикладі рядки виводяться за допомогою функції `println()`.

Приклад

```
List<String> al = new ArrayList<String>();
al.add("abc");
al.add("def");
al.add("xyz");
for (int i = 0; i < al.size(); i++) {
    System.out.println(al.get(i));
}
```

Метод `set()` дозволяє змінити об'єкт, що зберігається в зазначеній позиції. Метод `remove()` видаляє об'єкт у зазначеній позиції.

Приклад

```
al.set(0, "new");
al.remove(2);
System.out.println(al); // [new, def]
```

Функція `subList(fromIndex, toIndex)` повертає список, складений з елементів починаючи з елемента з індексом `fromIndex` і не включаючи елемент з індексом `toIndex`.

Приклад

```
System.out.println(al.subList(1, 3)); // [def, xyz]
```

Метод `removeRange(m, n)` видаляє всі елементи, індекс яких між `m` включно та не включаючи `n`. Усі елементи колекцій видаляються за допомогою методу `clear()`. Функція `contains()` повертає **`true`**, якщо список містить зазначений елемент.

Приклад

```
if (al.contains("abc")) {
    System.out.println(al);
}
```

Функція `toArray()` повертає посилання на масив копій об’єктів, посилання на які зберігаються в списку.

Приклад

```
Object [] a = al.toArray();
System.out.println(a[1]);    // def
(al.toArray()) [2] = "text"; // Зміна елементів нового масиву
```

Для збереження цілих і дійсних значень у колекціях часто використовують класи-оболонки `Integer` та `Double` відповідно.

У тих випадках, коли частіше, ніж вибір довільного елемента, застосовують операції додавання і видалення елементів у довільних місцях, доцільно використовувати клас `LinkedList`, що зберігає об’єкти за допомогою зв’язаного списку. Зв’язний список, реалізований у Java контейнером `LinkedList`, є двобічно зв’язаним списком, де кожен елемент містить посилання на попередній і наступний елементи.

Для зручної роботи додані також методи `addFirst()`, `addLast()`, `removeFirst()` і `removeLast()`.

Приклад

```
LinkedList<String> list = new LinkedList<String>();
list.addLast("last"); // Те саме, що list.add("last");
list.addFirst("first");
System.out.println(list); // [first, last]
list.removeFirst();
list.removeLast();
System.out.println(list); // []
```

Ці специфічні функції додані саме в `LinkedList`, оскільки вони не можуть бути ефективно реалізовані в ArrayList із застосуванням масивів.

Зв’язані списки у Java підтримують роботу з окремими елементами, але ці функції не можуть бути реалізовані ефективно.

4.2.4. Ітератори

Для проходження по колекції (списку) об’єктів використовується ітератор – спеціальний допоміжний об’єкт. Як і самі контейнери, ітератори базуються на

інтерфейсі. Інтерфейс `Iterator`, визначений у пакеті `java.util`. Будь-який ітератор має три методи:

```
boolean hasNext();//перевіряє, чи є ще елементи в контейнері,
       // та повертає true, якщо ще є елементи послідовності.
Object  next();    // повертає посилання на наступний елемент
void    remove();  // видаляє останній обраний елемент
       // (на який посилається ітератор)
```

Після першого виклику методу `next()` ітератор указуватиме на початковий елемент контейнеру. Колекції Java повертають об'єкт-ітератор за допомогою методу `iterator()`.

Приклад

```
List<String> s = new ArrayList<String>();
s.add("First");
s.add("Second");
for (Iterator<String> i = s.iterator(); i.hasNext(); ) {
    System.out.println(i.next());
}
```

Як видно з наведеного прикладу, ітератор теж є узагальненим типом.

Альтернативна форма циклу **`for`** (`for each`) дозволяє обійти список без явного створення ітератору.

Приклад

```
List<Integer> a = new ArrayList<Integer>();
a.add(1);
a.add(2);
a.add(3);
a.add(4);
for (Integer i : a) {
    System.out.print(i + " ");
}
```

Як і для масивів, альтернативна форма циклу `for` не дозволяє змінювати значення елементів, або видаляти їх.

Спеціальний вид ітератору списку, `ListIterator`, надає додаткові можливості ітерації, зокрема, проходження по списку у зворотному порядку. В наведеному нижче прикладі для перевірки, чи є слово паліндромом

використовується список символів і `ListIterator`, який забезпечує проходження в зворотному порядку.

Приклад

```
import java.util.*;

public class ListIteratorTest {
    public static void main(String[] args) {
        String palStr = "racecar";
        List<Character> palindrome =
                new LinkedList<Character>();
        for (char ch : palStr.toCharArray()) {
            palindrome.add(ch);
        }
        System.out.println("Input string is: " + palStr);
        ListIterator<Character> iterator =
                palindrome.listIterator();
        ListIterator<Character> revIterator =
                palindrome.listIterator(palindrome.size());
        boolean result = true;
        while (revIterator.hasPrevious()
               && iterator.hasNext()) {
            if (iterator.next() != revIterator.previous()) {
                result = false;
                break;
            }
        }
        if (result) {
            System.out.print("Input string is a palindrome");
        }
        else {
            System.out.print(
                "Input string is not a palindrome");
        }
    }
}
```

4.2.5. Додаткові можливості роботи з колекціями

Починаючи з Java 8, стандартні інтерфейси пакету `java.util` доповнені методами, орієнтованими на використання лямбда-виразів і посилань на методи. Для забезпечення сумісності з попередніми версіями Java нові методи інтерфейсів представлені з усталеною реалізацією. Зокрема, інтерфейс `Iterable` визначає метод `forEach()`, який дозволяє виконати в циклі деякі дії, що не змінюють елементів колекції. Дію можна задати лямбда-виразом або посиланням на метод.

Приклад

```
public class ForEachDemo {
    static int sum = 0;

    public static void main(String[] args) {
        Iterable<Integer> numbers =
            new ArrayList(Arrays.asList(2, 3, 4));
        numbers.forEach(n -> sum += n);
        System.out.println(sum);
    }
}
```

У наведеному вище прикладі здійснюється сумування елементів колекції. Змінна-сума описана як статичне поле класу, оскільки лямбда-вирази не можуть змінювати локальні змінні.

Інтерфейс `Collection` визначає метод `removeIf()`, який дозволяє видалити з колекції дані, відповідні деякому правилу-фільтру. У наведеному нижче прикладі з колекції цілих чисел видаляються непарні елементи. Метод `forEach()` використовується для виведення елементів колекції в стовпчик.

Приклад

```
Collection<Integer> c = new ArrayList(
                        Arrays.asList(2, 4, 11, 8, 12, 3));
c.removeIf(k -> k % 2 != 0);
// Решта елементів виводиться в стовпчик:
c.forEach(System.out::println);
```

Інтерфейс `List` надає методи `replaceAll()` і `sort()`. Останній можна використовувати замість аналогічного статичного методу класу `Collections`, проте визначення ознаки сортування є обов’язковим.

Приклад

```
List<Integer> list = new ArrayList(
                        Arrays.asList(2, 4, 11, 8, 12, 3));
list.replaceAll(k -> k * k); // замінюємо числа квадратами
System.out.println(list);    // [4, 16, 121, 64, 144, 9]
list.sort(Integer::compare);
System.out.println(list);    // [4, 9, 16, 64, 121, 144]
list.sort((i1, i2) -> Integer.compare(i2, i1));
System.out.println(list);    // [144, 121, 64, 16, 9, 4]
```

4.3. Робота з чергами та стеками

Черга в широкому сенсі є структурою даних, яку заповнюють поелементно, та отримують з неї об’єкти за певним правилом. У вузькому сенсі цим правилом є "першим прийшов – першим вийшов" (FIFO, First In – First Out). У черзі, організованій за принципом FIFO, додавання елемента можливо лише в кінець черги, отримання – тільки з початку черги.

У бібліотеці контейнерів черга представлена інтерфейсом `Queue`. Методи, оголошені в цьому інтерфейсі, наведені в таблиці 4.2.

Таблиця 4.2

Тип операції	Генерує виняток	Повертає спеціальне значення
Додавання	`add(e)`	`offer(e)`
Видалення з отриманням елемента	`remove()`	`poll()`
Отримання елемента без видалення	`element()`	`peek()`

Метод `offer()` повертає **`false`**, якщо не вдалося додати елемент, наприклад, якщо реалізована черга з обмеженою кількістю елементів. У цьому випадку метод `add()` генерує виняток. Аналогічно `remove()` і `element()` генерують виняток, якщо черга порожня, а `poll()` і `peek()` в цьому випадку повертають **`null`**.

Для реалізації черги найзручніше використовувати клас `LinkedList`, який реалізує інтерфейс `Queue`.

Приклад

```
import java.util.LinkedList;
import java.util.Queue;

public class SimpleQueueTest {

    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("First");
```

```
        queue.add("Second");
        queue.add("Third");
        queue.add("Fourth");
        String s;
        while ((s = queue.poll()) != null) {
            System.out.print(s + " ");
                // First Second Third Fourth
        }
    }
}
```

Клас `PriorityQueue` впорядковує елементи відповідно до компаратора (об'єкта класу, що реалізує інтерфейс `Comparator`), заданого в конструкторі як параметр. Якщо об'єкт створити за допомогою конструктора без параметрів, елементи будуть упорядковані в природному порядку (для чисел – за зростанням, для рядків – за абеткою).

Приклад

```
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueTest {

    public static void main(String[] args) {
        Queue<String> queue = new PriorityQueue<>();
        queue.add("First");
        queue.add("Second");
        queue.add("Third");
        queue.add("Fourth");
        String s;
        while ((s = queue.poll()) != null) {
            System.out.print(s + " ");
                // First Fourth Second Third
        }
    }

}
```

Інтерфейс `Deque` (дек, double-ended-queue) надає можливість додавати й видаляти елементи з обох кінців. Методи, оголошені в цьому інтерфейсі, наведені в табл. 4.3.

Таблиця 4.3

Тип операції	Робота з першим елементом	Робота з останнім елементом
Додавання	`addFirst(e)` `offerFirst(e)`	`addLast(e)` `offerLast(e)`
Видалення з отриманням елемента	`removeFirst()` `pollFirst()`	`removeLast()` `pollLast()`
Отримання елемента без видалення	`getFirst()` `peekFirst()`	`getLast()` `peekLast()`

Кожна з пар представляє відповідно функцію, яка генерує виняток, і функцію, яка повертає спеціальне значення. Є також методи, що дозволяють видалити перше або останнє входження заданого елемента (`removeFirstOccurence()` і `removeLastOccurence()` відповідно).

Для реалізації інтерфейсу можна використовувати спеціальний клас `ArrayDeque`, або зв'язний список (`LinkedList`).

Стек – це структура даних, організована за принципом "останній прийшов – перший вийшов" (LIFO, last in - first out). Можливі три операції зі стеком: додавання елементу (`push`), видалення елементу (`pop`) і читання головного елемента (peek).

У JRE 1.1 стек представлений класом `Stack`.

Приклад

```
import java.util.Stack;

public class StackTest {

    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("First");
        stack.push("Second");
        stack.push("Third");
        stack.push("Fourth");
        String s;
        while (!stack.isEmpty()) {
            s = stack.pop();
            System.out.print(s + " ");
               // Fourth Third Second First
        }
```

```
    }
}
```

Цей клас в даний час не рекомендований до використання. Замість нього можна використовувати інтерфейс `Deque`, який оголошує аналогічні методи.

Приклад

```
import java.util.ArrayDeque;
import java.util.Deque;

public class AnotherStackTest {

    public static void main(String[] args) {
        Deque<String> stack = new ArrayDeque<>();
        stack.push("First");
        stack.push("Second");
        stack.push("Third");
        stack.push("Fourth");
        String s;
        while (!stack.isEmpty()) {
            s = stack.pop();
            System.out.print(s + " ");
                // Fourth Third Second First
        }
    }
}
```

Стеки часто використовуються в різних алгоритмах. Зокрема, за допомогою стеку в деяких задачах можна позбутися рекурсії.

4.4. Статичні методи класу Collections. Алгоритми

4.4.1. Створення спеціальних контейнерів з використанням класу Collections

Як клас `Arrays` для масивів, для колекцій існує допоміжний клас `Collections`. Цей клас надає низку функцій для роботи з колекціями, зокрема зі списками. Велика група функцій призначена для створення колекцій різних типів. Наведений нижче приклад демонструє створення колекцій за допомогою статичних методів класу `Collections`: відповідно, порожнього списку (`emptyList()`), "одинака" (`singletonList()`) і списку тільки для читання зі

звичайного (`unmodifiableList()`) або колекції тільки для читання (`unmodifiableCollection()`).

Приклад

```
List<Integer> emptyList = Collections.emptyList();
System.out.println(emptyList); // []
List<Integer> singletonList = Collections.singletonList(10);
System.out.println(singletonList); // [10]
List<Integer> list = new ArrayList<>
    (Arrays.<Integer>asList(1, 2, 3));
List<Integer> unmodifiableList =
    Collections.unmodifiableList(list);
Collection<Integer> collection =
    Collections.unmodifiableCollection(list);
```

Всі наведені вище функції створюють набори даних тільки для читання.

Аналогічно працюють методи, що створюють відповідні множини - `emptySet()`, `singleton()`, `unmodifiableSet()`.

4.4.2. Алгоритми

Під алгоритмом в бібліотеці колекцій слід розуміти деяку функцію, що реалізує роботу з колекцією (отримання певного результату або перетворення елементів колекції). В Java Collections API ця функція як правило, статична і узагальнена.

Як і клас `Arrays`, клас `Collections` містить реалізацію статичних функцій `sort()` та `fill()` з аналогічними правилами застосування. Окрім того, є велика кількість статичних функцій, які дозволяють обробляти списки без застосування циклів. Це, наприклад, такі функції, як: `max()` (пошук максимального елемента), `min()` (пошук мінімального елемента), `indexOfSubList()` (пошук індексу першого повного входження підсписку у список), `frequency()` (визначення кількості разів входження певного елемента у список), `reverse()` (заміна порядку елементів на протилежний), `rotate()` (циклічний зсув списку на задану кількість елементів), `shuffle()` ("тасування" елементів), `nCopies()` (створення нового списку з визначеною кількістю однакових елементів). Використання цих функції можна проілюструвати на наведеному нижче прикладі.

Приклад

```
List<Integer> a = Arrays.asList(0, 1, 2, 3, 3, -4);
System.out.println(Collections.max(a)); // 3
System.out.println(Collections.min(a)); // -4
System.out.println(Collections.frequency(a, 2)); // 1 раз
System.out.println(Collections.frequency(a, 3)); // 2 рази
Collections.reverse(a);
        // змінюємо порядок елементів на протилежний
System.out.println(a);    // [-4, 3, 3, 2, 1, 0]
Collections.rotate(a, 3);
        // зсуває список циклічно на 3 елементи
System.out.println(a);    // [2, 1, 0, -4, 3, 3]
List<Integer> sublist = Collections.nCopies(2, 3);
        // новий список містить 2 трійки
System.out.println(Collections.indexOfSubList(a,sublist));//4
Collections.shuffle(a);   // "тасуємо" елементи
System.out.println(a);    // елементи в довільному порядку
Collections.sort(a);
System.out.println(a);    // [-4, 0, 1, 2, 3, 3]
List<Integer> b = new ArrayList<Integer>(a);
Collections.fill(b, 8);
System.out.println(b);    // [8, 8, 8, 8, 8, 8]
Collections.copy(b, a);
System.out.println(b);    // [-4, 0, 1, 2, 3, 3]
System.out.println(Collections.binarySearch(b, 2)); // 3
Collections.swap(b, 0, 5);
System.out.println(b);    // [3, 0, 1, 2, 3, -4]
Collections.replaceAll(b, 3, 10);
System.out.println(b);    // [10, 0, 1, 2, 10, -4]
```

Клас `Collections` також надає методи спеціально для роботи зі списками, отримання першого і останнього індексу початку входження підсписку в список (`indexOfSubList()`, `lastIndexOfSubList()`) тощо.

4.5. Робота з множинами та асоціативними масивами

4.5.1. Множини

Множина – це колекція, що не містить однакових елементів. Три основних реалізації інтерфейсу `Set` – `HashSet`, `LinkedHashSet` і `TreeSet`. Як і списки, множини є узагальненими типами. Класи `HashSet` і `LinkedHashSet` використовують хеш-коди для ідентифікації елемента. Клас `TreeSet`

використовує двійкове дерево для збереження елементів і гарантує їх певний порядок.

Метод `add()` додає елемент до множини і повертає **`true`** якщо елемент раніше був відсутній. В іншому випадку елемент не додається, а метод `add()` повертає **`false`**. Усі елементи множини видаляються за допомогою методу `clear()`.

Приклад

```
Set<String> s = new HashSet<String>();
System.out.println(s.add("First"));  // true
System.out.println(s.add("Second")); // true
System.out.println(s.add("First"));  // false
System.out.println(s);               // [First, Second]
s.clear();
System.out.println(s);               // []
```

Метод `remove()` видаляє зазначений елемент множини, якщо такий є. Метод `contains()` повертає **`true`**, якщо множина містить зазначений елемент.

У наведеному нижче прикладі до множини цілих чисел додається десять випадкових значень у діапазоні від -9 до 9.

Приклад

```
import java.util.*;

public class SetOfIntegers {
    public static void main(String[] args) {
        Set<Integer> set = new TreeSet<Integer>();
        Random random = new Random();
        for (int i = 0; i < 10; i++) {
            Integer k = random.nextInt() % 10;
            set.add(k);
        }
        System.out.println(set);
    }
}
```

Результуюча множина як правило, містить менш, ніж 10 чисел, оскільки окремі значення можуть повторюватися. Оскільки ми використовуємо `TreeSet`, числа зберігаються та виводяться в упорядкованому (за зростанням) вигляді.

Для того, щоб додати саме десять різних чисел, програму можна модифікувати, наприклад із застосуванням циклу **`while`** замість **`for`**.

Приклад

```
while (set.size() < 10) {
    . . .
}
```

Можна створити масив, який містить копії елементів множини. В такий спосіб можна звертатися до елементів за індексом. Можна вивести елементи множини в зворотному порядку, як наведено у прикладі.

Приклад

```
Set<Integer> set = new HashSet<>(Arrays.asList(1, 2, 4));
Object[] arr = set.toArray();
for (int i = set.size() - 1; i >= 0; i--) {
    System.out.println(arr[i]);
}
```

Оскільки множина може містити тільки різні елементи, її можна використати для підрахунку різних слів, літер, цифр тощо – створюється множина та викликається метод `size()`. Застосовуючи `TreeSet`, можна виводити слова та літери в алфавітному порядку. У наведеному нижче прикладі вводиться речення та виводяться всі різні літери речення (не враховуючи роздільників) в алфавітному порядку.

Приклад

```
import java.util.*;

public class Sentence {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Функція nextLine() читає рядок до кінця:
        String sentence = scanner.nextLine();
        // Створюємо множину роздільників:
        Set<Character> delimiters = new HashSet<Character>(
            Arrays.asList(' ', '.', ',', ':', ';', '?',
                '!', '-', '(', ')', '\"'));
        // Створюємо множину літер:
```

```
        Set<Character> letters = new TreeSet<Character>();
        // Додаємо всі літери крім роздільників:
        for (int i = 0; i < sentence.length(); i++) {
            if (!delimiters.contains(sentence.charAt(i))) {
                letters.add(sentence.charAt(i));
            }
        }
        System.out.println(letters);
    }
}
```

Порядок сортування елементів `TreeSet` можна задати, реалізувавши інтерфейс `Comparable`, або передавши в конструктор `TreeSet` посилання на об’єкт класу, який реалізує інтерфейс `Comparator`. Можна відсортувати дерево у зворотному порядку, як наведено у прикладі.

Приклад

```
import java.util.*;

public class CompTest {

    public static void main(String args[]) {
        TreeSet<String> ts = new TreeSet<String>(
            new Comparator<String>()
        {
            @Override
            public int compare(String s1, String s2) {
                return s2.compareTo(s1);
            }
        });
        ts.add("C");
        ts.add("E");
        ts.add("D");
        ts.add("B");
        ts.add("A");
        ts.add("F");
        for (String element : ts)
            System.out.print(element + " ");
        }
    }
}
```

4.5.2. Робота з асоціативними масивами

Асоціативні масиви можуть зберігати пари посилань на об’єкти. Асоціативні масиви теж є узагальненими типами. Асоціативні масиви у Java представлені узагальненим інтерфейсом `Map`, який реалізовано, зокрема, класом

HashMap. Інтерфейс SortedMap, похідний від Map, вимагає впорядкованого за ключем зберігання пар. Інтерфейс NavigableMap, що з'явився в Java SE 6, розширює SortedMap і додає нові можливості пошуку за ключем. Цей інтерфейс реалізовано класом TreeMap.

Кожне значення (об'єкт), яке зберігається в асоціативному масиві, зв'язується з конкретним значенням іншого об'єкта (ключа). Метод put(key, value) додає значення (value) і асоціює з ним ключ (key). Якщо асоціативний масив раніше містив пари з вказаним ключем, нове значення заміщає старе. Метод put() повертає попереднє значення, пов'язане з ключем, або **null**, якщо ключ був відсутній. Метод get(Object key) повертає об'єкт за заданим ключем. Для перевірки знаходження ключа та значення застосовуються методи containsKey() і containsValue().

На логічному рівні можна представити асоціативний масив через три допоміжні колекції:

– keySet - множина значень ключа;

– values - список значень;

– entrySet - множина пар ключ-значення.

Через відповідні функції keySet(), values() та entrySet() можна здійснювати певні дії, в першу чергу послідовне проходження елементів.

У наведеному нижче прикладі обчислюється кількість входжень різних слів у речення. Слова і відповідні кількості зберігаються в асоціативному масиві. Використання класу TreeMap гарантує алфавітний порядок слів.

Приклад

```
import java.util.*;

public class WordsCounter {
    public static void main(String[] args) {
        Map<String, Integer> m =
                new TreeMap<String, Integer>();
        String s = "the first men on the moon";
        StringTokenizer st = new StringTokenizer(s);
        while (st.hasMoreTokens()) {
            String word = st.nextToken();
            Integer count = m.get(word);
```

```
            m.put(word, (count == null) ? 1 : count + 1);
        }
        for (String word : m.keySet()) {
            System.out.println(word + " " + m.get(word));
        }
    }

}
```

Використання `keySet()` передбачає окремий пошук кожного значення за ключем. Більш рекомендованим є обхід через множину пар.

Приклад

```
for (Map.Entry<?, ?> entry : m.entrySet())
    System.out.println(entry.getKey() + " " +
        entry.getValue());
```

Тут метод `entrySet()` дозволяє одержати представлення асоціативного масиву у вигляді колекції `Set`.

Порядок сортування елементів `TreeMap` також можна змінити, вказавши як параметр конструктора `TreeMap` об'єкт класу, який реалізує інтерфейс `Comparator`, або задавши ключ як об'єкт класу, який реалізує інтерфейс `Comparable`.

Приклад

```
import java.util.*;

public class TreeMapKey implements Comparable<TreeMapKey> {
  private String name;

  public String getName() {
    return name;
  }

  public TreeMapKey(String name) {
    super();
    this.name = name;
  }

  @Override
  public int compareTo(TreeMapKey o) {
    return name.substring(o.getName().indexOf(" "))
      .trim().compareToIgnoreCase(o.getName()
      .substring(o.getName().indexOf(" ")).trim());
  }
```

```
  public static void main(String args[]) {
    TreeMap<TreeMapKey, Integer> tm =
      new TreeMap<TreeMapKey, Integer>();
    tm.put(new TreeMapKey("Петро Іванів"),
      new Integer(1982));
    tm.put(new TreeMapKey("Іван Петрів"), new Integer(1979));
    tm.put(new TreeMapKey("Василь Сидір"),
      new Integer(1988));
    tm.put(new TreeMapKey("Сидір Василько"),
      new Integer(1980));
    for (Map.Entry<TreeMapKey, Integer> me : tm.entrySet()) {
      System.out.print(me.getKey().getName() + ": ");
      System.out.println(me.getValue());
    }
    System.out.println();
  }
}
```

Клас `Hashtable` – одна з реалізацій інтерфейсу `Map`. `Hashtable` крім розміру має ємність (розмір буфера, виділеного під елементи масиву). Він характеризується показником завантаженості – часткою буфера, після заповнення якої ємність автоматично збільшується. Конструктор `Hashtable()` без параметрів створює порожній об’єкт з ємністю в 101 елемент і показником завантаженості 0.75.

Клас `Properties`, похідний від `Hashtable`, замість пар довільних об’єктів зберігає пари рядків. Якщо в конкретній задачі і ключі і значення елементів асоціативного масиву мають тип `String`, зручніше скористатися класом `Properties`. У класі `Properties` визначені методи `getProperty(String key)` і `setProperty(String key, String value)`.

Вправи для самостійної роботи

1. Реалізувати статичну узагальнену функцію заміни порядку елементів на протилежний. Здійснити тестування функції на двох масивах різних типів..

2. Прочитати з клавіатури значення елементів списку цілих чисел. Знайти добуток елементів..

3. Увести кількість елементів майбутньої множини цілих чисел та діапазон чисел. Сформувати цю множину з випадкових значень. Вивести елементи множини, відсортовані за збільшенням.

4. Увести речення та обчислити кількість різних слів у реченні.

Контрольні запитання

1. Що таке параметризований тип?

2. Для чого використовують узагальнені функції?

3. Чому в контейнерному класі не можна зберігати цілі і дійсні числа безпосередньо, а можна тільки посилання?

4. Що таке колекція?

5. У чому перевага використання ітераторів у порівнянні з індексами елементів контейнера?

6. Яку структуру даних використовують для реалізації `LinkedList`?

7 Для чого використовують стеки?

8. Чим множина відрізняється від списку?

9. Наведіть приклади використання асоціативних масивів.

5. РОБОТА З ВИНЯТКАМИ ТА ФАЙЛАМИ

5.1. Обробка винятків

5.1.1. Основні концепції

Використання механізму обробки винятків є дуже важливою складовою частиною практики програмування на Java. Майже кожна програма на Java містить певні частини цього механізму. Об'єкти-винятки дозволяють програмісту відокремити точки виникнення помилок під час виконання від коду, який ці помилки повинен обробити. Це дозволяє створювати більш надійно працюючі універсальні класи і бібліотеки.

Виняток – це подія, що виникає під час виконання програми і порушує нормальне виконання інструкцій коду. Механізм генерації та обробки винятків дозволяє передати інформацію про помилку з місця виникнення у місце, де ця помилка може бути оброблена. Винятки в Java поділяють на синхронні (помилка часу виконання, ситуація, згенерована за допомогою **`throw`**) і асинхронні (системні, збої віртуальної машини Java). Місце виникнення другої групи винятків виявити досить складно.

Механізм винятків присутній в усіх сучасних мовах об'єктно-орієнтованого програмування. У порівнянні з іншими мовами, Java реалізує більш строгий механізм роботи з винятками.

5.1.2. Синтаксис генерації винятків

Для генерації винятку використовується оператор **`throw`**. Після ключового слова **`throw`** міститься об'єкт класу `java.lang.Throwable`, або класів, похідних від нього. Для програмних винятків найчастіше використовується клас `java.lang.Exception` (похідний від `Throwable`). Використання `Exception` замість `Throwable` дозволяє відокремити власний виняток від системних помилок. Найкраща практика керування винятками – створювати класи, похідні від `Exception`. Такі похідні класи зазвичай відбивають специфіку конкретної програми.

Приклад

```
class SpecificException extends Exception {
}
```

Є також базовий клас для генерації системних помилок – клас `Error`. Класи `Exception` і `Error` мають загальний базовий клас – `Throwable`.

Виняток генерується шляхом використання ключового слова **`throw`**, за яким розташовують об'єкт-виняток. У більшості випадків об'єкт-виняток створюється в точці генерації винятку за допомогою оператора **`new`**. Типове твердження **`throw`** може виглядати, як наведено у прикладі.

Приклад

```
void f() . . .
    . . .
    if (/* помилка */) {
        throw new SpecificException();
    }
```

У заголовку функції необхідно перелічити усі типи винятків, які генерує ця функція. Це слід зробити за допомогою ключового слова **`throws`**.

Приклад

```
void f() throws SpecificException, AnotherException {
    . . .
    if (/* помилка */) {
        throw new SpecificException();
    }
    if (/* інша помилка */) {
        throw new AnotherException();
    }
    . . .
}
```

У наведеному нижче прикладі функція **`reciprocal`**`()` генерує виняток у випадку ділення на нуль.

Приклад

```
class DivisionByZero extends Exception {
}
```

```
class Test {

    double reciprocal(double x) throws DivisionByZero {
        if (x == 0) {
            throw new DivisionByZero();
        }
        return 1 / x;
    }
}
```

На відміну від C++, Java не допускає створення винятків примітивних типів. Дозволені тільки об’єкти, похідні від `Throwable` або `Exception`.

Під час успадкування для перевизначених функцій список винятків повинен зберігатися.

5.1.3. Синтаксис обробки винятків

Виняток, який був згенерований у певній частині коду, повинен бути перехоплений в іншій частині. Наприклад, якщо ми хочемо звернутися до функції, яка потенційно може згенерувати виняток, виклик цієї функції поміщають у блок **`try`** `{ }`.

Приклад

```
double x, y;
. . .
try {
    y = reciprocal(x);
}
```

Після блоку **`try`** повинен міститись один чи декілька оброблювачів (блоків **`catch`**). Кожен такий оброблювач відповідає визначеному типу винятку.

Приклад

```
catch (DivisionByZero d) {
    // обробка винятку
}
catch (Exception ex)  {
    // обробка винятку
}
```

Класи винятків утворюють ієрархію. Під час порівняння типів винятків оброблювач базового типу сприймає також винятки всіх створених від нього

типів. Звідси випливає, що оброблювачі похідних типів варто розміщати до оброблювачів базових типів. Припустимо, є така ієрархія класів винятків.

Приклад

```
class BaseException extends Exception {
}

class FileException extends BaseException {
}

class FileNotFoundException extends FileException {
}

class WrongFormatException extends FileException {
}

class MathException extends BaseException {
}

class DivisionByZero extends MathException {
}

class WrongArgument extends MathException {
}
```

Припустимо, є деяка функція, яка може згенерувати всі типи винятків.

Приклад

```
public class Exceptions {
    public static void badFunc() throws BaseException {
        // можуть виникнути різні винятки
    }
}
```

Залежно від логіки програми різні типи винятків можна обробляти більш детально.

Приклад

```
try {
    Exceptions.badFunc();
}
catch (FileNotFoundException ex) {
    // файл не знайдено
}
catch (WrongFormatException ex) {
    // хибний формат
}
```

```
catch (FileException ex) {
    // інші помилки, пов'язані з файлами
}
catch (MathException ex) {
    // усі математичні помилки обробляємо разом
}
catch (BaseException ex) {
    // підбираємо всі інші винятки функції badFunc()
}
catch (Exception ex) {
    // про всяк випадок
}
```

Після останнього блоку **catch** можна розмістити блок **finally**. Цей код завжди виконується незалежно від того, виник чи не виник виняток, навіть якщо в якомусь з блоків був здійснений вихід з функції.

Приклад

```
try {
    openFile();
    // інші дії
}
catch (FileError f) {
    // обробка винятку
}
catch (Exception ex) {
    // обробка винятку
}
finally {
    closeFile();
}
```

У версії Java 7 до синтаксису винятків додані нові конструкції, які роблять роботу з винятками більш зручною. Наприклад, можна створити обробник подій різних типів з використанням побітової операції "АБО".

Приклад

```
public void newMultiCatch() {
    try {
        methodThatThrowsThreeExceptions();
    }
    catch (ExceptionOne | ExceptionTwo | ExceptionThree e) {
        // обробка всіх винятків
    }
}
```

Інші додаткові можливості пов’язані з так званим блоком управління ресурсами ("try-with-resources"). Для об’єкта класу, який реалізує інтерфейс `java.lang.AutoCloseable` можна розмістити створення об’єкта безпосередньо після **`try`**. Для такого об’єкта автоматично буде викликано метод `close()` після завершення блоку **`try`** `{}` (аналогічно виконанню коду в **`finally`**).

Приклад

```
try (ClassThatImplementsAutoCloseable sc
          = new ClassThatImplementsAutoCloseable()) {
    // дії, які можуть призвести до винятку
}
catch (Exception f) {
    // обробка винятку
}   // автоматичний виклик sc.close()
```

На відміну від C++, не можна використовувати **`catch`** `(...)` для перехоплення будь-якого винятку. Замість цього можна використовувати перехоплення винятків базових класів:

```
catch (Exception ex) {
    // обробка винятку
}
```

або

```
catch (Throwable ex) {
    // обробка винятку
}
```

Типова реалізація оброблювача винятку – виклик методу `printStackTrace()`.

Приклад

```
catch (Throwable ex)  {
    ex.printStackTrace();
}
```

Цей метод здійснює виведення інформації про трасування стеку в стандартний потік повідомлень про помилки `System.err`. Нижче наведений приклад роботи функції `printStackTrace()`.

Приклад

```
java.lang.NullPointerException
        at SomeClass.g(SomeClass.java:9)
        at SomeClass.f(SomeClass.java:6)
        at SomeClass.main(SomeClass.java:3)
```

Якщо в межах блоку **`catch`** () { } не можна повністю обробити виняток, його можна передати далі.

Приклад

```
catch (SomeException ex) {
    // локальна обробка винятку
    throw ex;
}
```

Іноді для адекватної обробки інформації про виняток необхідно володіти певною додатковою інформацією. Наприклад, ми створюємо функцію, всередині тіла якої необхідно знайти квадратний корінь. Якщо аргумент від'ємний, необхідно генерувати виняток. Для налагодження програми корисно знати, яке саме від'ємне значення було отримане. Можна створити клас-виняток, об'єкт якого зберігатиме це значення. У конструкторі воно встановлюється, а в точці обробки винятку його можна отримати за допомогою геттера. Цей підхід можна продемонструвати на такому прикладі. Створюємо клас-виняток.

Приклад

```
public class WrongArgumentException extends Exception {
    private double arg;

    public WrongArgumentException(double arg) {
        this.arg = arg;
    }

    public double getArg() {
        return arg;
    }
}
```

Виняток може бути згенерований у якійсь функції, якщо неможливо використати аргумент.

Приклад

```
public class SomeLib {

    public static void doSomeUseful(double x)
            throws WrongArgumentException {
        // перевірка x
        if (x < 0)
            throw new WrongArgumentException(x);
        double y = Math.sqrt(x);
        // подальша робота
    }
}
```

Тепер перехоплений об’єкт-виняток може бути застосований для отримання більш детальної інформації.

Приклад

```
public class ExceptionTest {
    public static void main(String[] args) {
        double x =
            new java.util.Scanner(System.in).nextDouble();
        try {
            // . . .
            SomeLib.doSomeUseful(x);
            // . . .
        }
        catch (WrongArgumentException e) {
            System.err.println(
                e.getClass().getName() + e.getArg());
        }
    }
}
```

Як видно з наведеного прикладу, за допомогою викликів методу `getClass().getName()` можна отримати ім’я класу. Це можна зробити для будь-якого об’єкта (не тільки винятку).

Виклик функції, що може згенерувати виняток, поза блоком **`try`** приводить до помилки компіляції. Перевірка повинна обов’язково виконуватися.

Приклад

```
double f(double x) {
    double y;
```

```
    try {
        y = reciprocal(x);
    }
    catch (DivisionByZero ex) {
        ex.printStackTrace();
        y = 0;
    }
    return y;
}
```

Неперехоплений виняток може бути передано зовнішньому оброблювачу з використанням ключового слова **throws**.

Приклад

```
double g(double x) throws DivisionByZero {
    double y;
    y = reciprocal(x);
    return y;
}
```

Це правило обов'язкове для усіх винятків Java крім об'єктів класу `RuntimeException` або його нащадків. Про генерацію таких винятків не треба вказувати в заголовку функції. Програміст може обробляти чи ігнорувати такі винятки на свій розсуд. Функції, які генерують такі винятки, не декларують їх у своєму заголовку. Типовий клас винятків такого виду – `NullPointerException`.

Середовище IntelliJ IDEA дозволяє автоматизувати процес створення блоків перехоплення та обробки винятків. Якщо в тексті функції помітити блок та застосувати функцію `Code | Surround With... | try / catch`, помічений блок буде розташовано у блоці перехоплення винятків (**`try`** `{ }`), а далі будуть додані **`catch`**-блоки, які міститимуть стандартну обробку всіх можливих винятків.

5.2. Потоки введення та виведення. Потоки символів

5.2.1. Загальні концепції

Як більшість сучасних мов і платформ, Java узагальнює поняття потоків (streams), розповсюджуючи спільні підходи на файлові, консольні, мережеві та інші процесси введення-виведення.

Класи, які здійснюють файлове введення та виведення, а також інші дії з потоками, розташовані у пакеті `java.io`. Класи цього пакету пропонують низку методів для створення таких потоків, читання, запису, тощо. Існує дві підмножини класів – відповідно для роботи з текстовими та бінарними (двійковими) файлами.

Уся робота з потоками, окрім стандартних потоків `System.in` і `System.out`, повинна передбачати перехоплення виняткiв, пов'язаних з введенням-виведенням. Це `IOException` та його нащадки – `FileNotFoundException`, `ObjectStreamException` та інші.

Дуже важливо закрити всі файли, взаємодія з якими мала місце. Під час закриття файлів здійснюється запис у файл даних, що залишилися в буфері, звільнення буфера та інших ресурсів, пов'язаних з файлом. Закрити файл можна за допомогою методу `close()`. Наприклад, для потоку `in`:

```
in.close();
```

Якщо програма, яка потребує файлового введення, завантажується у середовищі IntelliJ IDEA, необхідні для читання файли слід розмістити у теці проєкту (не у теці пакету). Саме у теці проєкту можна знайти результуючі файли, які з'являються після завершення виконання програми, що включає файлове виведення.

У програмі можна одночасно відкрити декілька потоків введення і декілька потоків виведення.

5.2.2. Робота з потоками символів

Потоки, призначені для роботи з текстовою інформацією, мають назву потоків символів. Імена класів таких потоків закінчуються відповідно словами

"...Reader" і "...Writer". Безпосередню роботу з текстовими файлами здійснюють об'єкти класів FileReader та FileWriter.

Важливий елемент роботи з файловими потоками – це буферизація. Буферизація передбачає створення в оперативній пам'яті спеціальної області (буферу), у яку дані завантажуються з файлу для подальшого поелементного читання або дані поелементно записуються з подальшим переписуванням на диск. Об'єкти класу BufferedReader здійснюють таке буферизоване читання.

Для буферизованого виведення застосовують об'єкти класу BufferedWriter. Безпосереднє форматоване виведення здійснюється методами print() та println() об'єкту класу PrintWriter.

У наведеному нижче прикладі з файлу з ім'ям data.txt здійснюється читання одного цілого і одного дійсного значення, їхня сума записується у файл results.txt.

Приклад

```
import java.io.*;
import java.util.StringTokenizer;

public class FileTest {

    void readWrite() {
        try {
            FileReader fr = new FileReader("data.txt");
            BufferedReader br = new BufferedReader(fr);
            String s = br.readLine();
            int x;
            double y;
            try {
                StringTokenizer st = new StringTokenizer(s);
                x = Integer.parseInt(st.nextToken());
                y = Double.parseDouble(st.nextToken());
            }
            finally {
                br.close();
            }
            double z = x + y;
            FileWriter fw = new FileWriter("results.txt");
            PrintWriter pw = new PrintWriter(fw);
            pw.println(z);
            pw.close();
        }
        catch (IOException ex) {
```

```
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new FileTest().readWrite();
    }

}
```

Для відкриття файлу створюється об’єкт класу `FileReader`, у конструкторі якого вказується рядок – ім’я файлу. Посилання на створений об’єкт передається у конструктор класу `BufferedReader`. Читання з файлу здійснюється за допомогою методу `readLine()`, який повертає посилання на рядок символів, або **`null`**, якщо досягнуто кінець файлу.

Змінна s типу `String` посилається на рядок, який містить два числа. Для виділення з цього рядку окремих лексем використовують об’єкт класу `StringTokenizer`, у конструктор якого передається рядок. Посилання на окремі частини рядку поступово отримують за допомогою методу `nextToken()`. Ці посилання можуть бути використані безпосередньо, або використовуються для перетворення даних у числові значення (статичні методи `parseDouble()` та `parseInt()` класів `Double` та `Integer` відповідно).

Для читання з файлу можна використовувати вже знайомий клас `Scanner`. Фактичним параметром конструктора може бути файловий потік. Попередній приклад можна реалізувати за допомогою класу `Scanner`. Можна також скоротити код шляхом виключення непотрібних змінних. Крім того, доцільно скористатися конструкцією **`try`** `() { }` Java 7 для автоматичного закриття потоку.

Приклад

```
import java.io.*;
import java.util.Scanner;

public class FileTest {

    void readWrite() {
        try (Scanner scanner = new Scanner(
                new FileReader("data.txt"))) {
```

```
            try (PrintWriter pw = new PrintWriter(
                    "results.txt")) {
                pw.println(
                scanner.nextInt() + scanner.nextDouble());
            }
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new FileTest().readWrite();
    }
}
```

Перевагою такого підходу є можливість довільного розташування вихідних даних (не обов'язково в одному рядку). Як видно з наведеного прикладу, кілька блоків **try** { } можуть використовувати один блок **catch** { }. Альтернативою є розміщення декількох тверджень всередині дужок.

Приклад

```
try (Scanner scanner = new Scanner(
        new FileReader("data.txt"));
        PrintWriter pw = new PrintWriter("results.txt")) {
    pw.println(scanner.nextInt() + scanner.nextDouble());
}
catch (IOException ex) {
    ex.printStackTrace();
}
```

Під час роботи з класом `Scanner` можна визначити додаткові параметри, наприклад, встановити символ-роздільник (або послідовність символів). Перед читанням даних можна додати такий рядок, як наведено у прикладі.

Приклад

```
scanner.useDelimiter(",");
```

Тепер об'єкт-сканер буде сприймати коми як роздільники (замість пропусків).

5.3. Робота з бінарними потоками (потоками байтів)

Для роботи з нетекстовими (бінарними) файлами використовують потоки, імена яких замість "Reader" або "Writer" містять "Stream", наприклад InputStream, FileInputStream, OutputStream, FileOutputStream тощо. Такі потоки мають назву потоків байтів. У наведеному нижче прикладі здійснюється копіювання двійкового файлу FileCopy.class у теку проєкту з новим ім'ям.

Приклад

```
import java.io.*;

public class FileCopy {

    public static void copy(String inFile, String outFile) {
        byte[] buffer = new byte[1024]; // Буфер байтів
        try (InputStream input = new FileInputStream(inFile);
             OutputStream output =
                 new FileOutputStream(outFile)) {
            int bytesRead;
            while ((bytesRead = input.read(buffer)) >= 0) {
                output.write(buffer, 0, bytesRead);
            }
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    public static void main(String[] args) {
        copy("FileCopy.class", "FileCopy.copy");
    }
}
```

Як видно з наведеного прикладу, Java дозволяє використовувати звичайну риску (/) замість зворотної. Це – більш універсальний підхід, прийнятний для різних операційних систем. Крім того, зворотну риску необхідно було б записати двічі (\\).

Для роботи з бінарними файлами існують додаткові можливості – використання потоків даних і потоків об'єктів. Так звані потоки даних (data streams) підтримують бінарне введення / виведення значень примітивних типів даних (**boolean**, **char**, **byte**, **short**, **int**, **long**, **float** і **double**), а також значень

типу `String`. Усі потоки даних реалізують інтерфейси `DataInput` або `DataOutput`. Для більшості задач достатньо стандартних реалізацій цих інтерфейсів – `DataInputStream` і `DataOutputStream`. Дані у файлі зберігаються в такому вигляді, в якому вони представлені в оперативній пам'яті. Для запису рядків використовують метод `writeUTF()`. У наведеному нижче прикладі здійснюється запис даних.

Приклад

```
import java.io.*;

public class DataStreamDemo {

    public static void main(String[] args) {
        double x = 4.5;
        String s = "all";
        int[] a = { 1, 2, 3 };
        try (DataOutputStream out = new DataOutputStream(
                new FileOutputStream("data.dat"))) {
            out.writeDouble(x);
            out.writeUTF(s);
            for (int k : a)
                out.writeInt(k);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Тепер дані можна прочитати в іншій програмі.

Приклад

```
import java.io.*;
import java.util.*;

public class DataReadDemo {

    public static void main(String[] args) {
        try (DataInputStream in = new DataInputStream(
                new FileInputStream("data.dat"))) {
            double x = in.readDouble();
            String s = in.readUTF();
            List<Integer> list = new ArrayList<>();
            try {
```

```
                while (true) {
                    int k = in.readInt();
                    list.add(k);
                }
            }
            catch (Exception e) {
            }
            System.out.println(x);
            System.out.println(s);
            System.out.println(list);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Примітка. У наведеній вище програмі вихід з циклу здійснюється через збудження винятку. Такий підхід не є рекомендованим, оскільки генерація винятків знижує ефективність роботи програми. У нашому випадку доцільно було б окремо зберігати у файлі довжину масиву перед його елементами, а потім використовувати цю довжину для організації циклу **for** під час читання.

Для читання і запису даних може бути також використаний клас `java.io.RandomAccessFile`. Об'єкт цього класу дозволяє вільно пересуватися всередині файлу в прямому і зворотному напрямку. Основною перевагою класу `RandomAccessFile` є можливість читати і записувати дані в довільне місце файлу.

Для того щоб створити об'єкт класу `RandomAccessFile`, необхідно викликати його конструктор з двома параметрами: ім'я файлу для введення / виведення і режимом доступу до файлу. Для визначення режиму можна використовувати спеціальні рядки, такі як "`r`" (для читання), "`rw`" (для читання й запису) тощо. Відкриття файлу даних може бути таким, як наведено у прикладі.

Приклад

```
// для читання:
RandomAccessFile f1 = new RandomAccessFile("file1.dat", "r");
// для читання й запису:
RandomAccessFile f2 = new RandomAccessFile("file2.dat","rw");
```

Після того як файл відкритий, можна використовувати методи на кшталт `readDouble()`, `readInt()`, `readUTF()` тощо для читання або `writeDouble()`, `writeInt()`, `writeUTF()` тощо для виведення.

В основі керування файлом лежить вказівник на поточну позицію, де відбувається читання або запис даних. На момент створення об'єкта класу `RandomAccessFile` вказівник встановлюється на початок файлу і має значення 0. Виклики методів `read...()` і `write...()` зсувають позицію поточного вказівника на кількість прочитаних або записаних байтів. Для довільного зсуву вказівника на деяку кількість байтів можна використовувати метод `skipBytes()`, або ж встановити вказівник у певне місце файлу викликом методу seek(). Для того щоб дізнатися поточну позицію, в якій знаходиться вказівник, потрібно викликати метод `getFilePointer()`. Наприклад, в одній програмі ми записуємо дані в новий файл.

Приклад

```
RandomAccessFile out = new RandomAccessFile("new.dat", "rw");
int a = 1, b = 2;
out.writeInt(a);
out.writeInt(b);
out.close();
```

В іншій програмі ми читаємо друге ціле число.

Приклад

```
RandomAccessFile fi = new RandomAccessFile("new.dat", "rw");
fi.skipBytes(4); // переміщаємо указівник до другого числа
int c = fi.readInt();
System.out.println(c);
fileIn.close();
```

Дізнатися довжину файлу в байтах можна за допомогою функції `length()`.

Вправи для самостійної роботи

1. Прочитати з текстового файлу дійсні значення (до кінця файлу), знайти їх суму та вивести в інший текстовий файл.

2. Прочитати з текстового файлу цілі значення (до кінця файлу), знайти добуток парних елементів та вивести в інший текстовий файл..

3. Прочитати з бінарного файлу цілі значення (до кінця файлу), замінити від'ємні значення модулями, додатні нулями та вивести отримані значення в інший бінарний файл.

Контрольні запитання

1. Чи можна використовувати основний результат функції, якщо відбулася генерація винятку?

2. Чи можна розмістити виклик функції, що генерує виняток, поза блоком try?

3. У чому призначення функції `printStackTrace()`?

4. Які додаткові можливості синтаксису перехоплення винятків з'явилися у версії Java 7?

5. Чим відрізняються потоки байтів від потоків символів за областю застосування?

6. Які класи забезпечують роботу з текстовими файлами і бінарними файлами?

7. У чому сенс явного закриття файлів?

8. Яким чином можна забезпечити автоматичне закриття потоків?

9. Для чого використовують файли даних `DataOutputStream` і `DataInputStream`? Які у них переваги і недоліки?

СПИСОК ЛІТЕРАТУРИ

1. Bloch J. Effective Java: 3rd Edition, Addison Wesley, 2017, 412 p.

2. Schildt H. Java: A Beginner’s Guide: 8th Edition, McGraw-Hill Education, 2018, 684 p.

3. Schildt H. Java: The Complete Reference: 11th Edition, McGraw-Hill Education, 2018, 1208 p.

4. Horstmann C. S. Core Java Volume I – Fundamentals: 11th Edition, Prentice Hall 2018, 889 p.

5. Horstmann C. S. Core Java SE 9 for the Impatient: 2nd Edition Addison-Wesley Professional, 2017, 576 p.

6. Eckel B. Thinking in Java 4th Edition: Pearson, 2006, 1150 p.

7. Deitel P., Deitel H. Java How to Program, Early Objects: 11th Edition, Pearson, 2017, 1296 p.

8. Deitel P., Deitel H. Java How To Program, Late Objects: 11th Edition, Pearson, 2017, 1248 p.

9. Ратушняк Т. В. Програмування мовою JAVA. Практикум : навчальний посібник. Державна фіскальна служба України, Університет державної фіскальної служби України. – Ірпінь, 2017. – 212 с.

10. Копитко М.Ф., Іванків К.С. Основи програмування мовою Java : тексти лекцій. – Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002. – 83 с.

11. Брнакевич І.Є., Вагін П.П. Програмування мовою Java: використання фундаментальних класів : тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 75 с.

ЗМІСТ

Навчальне видання

ІВАНОВ Лев Вадимович
ПАШНЄВ Андрій Анатолійович

МОВА І ПЛАТФОРМА JAVA В ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЯХ

Навчальний посібник
для студентів з галузі знань «Інформаційні технології»
усіх форм навчання

Роботу до видання рекомендував проф. Гамаюн І. П.

В авторскій редакції

План 2022 р., поз. 28

Гарнітура Times New Roman. Др. арк. 5,5.
Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Електронна версія