ЛАБОРАТОРНА РОБОТА 1

Тема: Основні принципи програмування на мові Java

Сьогодні мова програмування Java знаходить своє застосування для рішення різноманітних задач. Це обумовлено як і її об'єктно-орієнтованою фундаментальністю — навіть найпростіша програма на Java повинна реалізовувати класи та методи в них, так і потужністю створених API — класи та інтерфейси згруповано у тематичні пакети, а система їх документування представляється у вигляді достатньо зручного дерева класів з необхідними гіперпосиланнями та є доступною для кожної версії Java на веб-серверах її розробників.

Безумовно, мова програмування Java має багато спільного з іншими мовами програмування — синтаксис, типи змінних та методів, загальне бачення використання концепцій ООП, оголошення деяких структур та інше. І це все тому, що її створення відбувалось на вже закладеному фундаменті такими мовами програмування як Smalltalk, Pascal, C та C++. Але популярність Java не була б такою високою, якщо б вона лише копіювала те, що вже було в інших мовах програмування. За задумом розробників мова програмування Java створювалась як дійсно кросплатформна, тобто таку, що підтримує парадигму — "пишеш один раз — використовуєш де завгодно". Після компіляції програми на Java утворюється байт-код — певний програмний напівфабрикат, для виконання якого потрібне спеціальне середовище - JVM (Java Virtual Machine - віртуальна машина Java). Саме наявність JVM для будь-якої операційної системи робить її кросплатформною. Тому розробники Java створюють не тільки її компілятори та різноманітні інструменти, що допомагають розробці та тестуванню коду, а обов'язково і середовище виконання.

Приступаючи до вивчення мови програмування Java слід розуміти, що вона є основою не тільки для створення додатків для так званих десктопних систем — звичайних комп'ютерів із системними блоками та моніторами та ноутбуків, а також дозволяє вирішувати різноманітні задачі із створення додатків як для корпоративних інформаційних систем, так і Інтернет. Тобто мова програмування Java надає розробнику значну кількість технологій та пов'язаних з ними класів для створення додатків, що потребують ресурси комп'ютерних мереж.

Розробники використовують різні версії платформи Java для створення програм Java. Платформа Java для розробки додатків, які у вигляді окремих програмам працюють на настільних комп'ютерах зветься Java Standard Edition (Java SE) та має open source варіант — OpenJDK. Платформа Java для розробки корпоративних програм та сервлетів, які працюють на окремих серверних комп'ютерних системах, зветься Java Enterprise Edition (Java EE) та має значну кількість різноманітних специфікацій, що забезпечуються відповідними бібліотеками. Компіляція та виконання компонент Java EE забезпечується Java SE, тому Java EE є спеціальною надбудовою останньої. Варіантом open source для Java EE є Jakarta EE.

Для того, щоб створити програму на мові програмування Java достатньо будь-якого текстового редактору, що відкриває та дозволяє вносити зміни у тестові файли зі звичайним ASCII кодуванням. Такий файл з кодом програми може бути перетворено у байт код для запуску у JVM за допомогою компілятора Java, наприклад, із пакету JDK (Java Development Kit). Зазвичай цей робиться з командного рядка. Якщо компіляція пройшла без помилок, то буде утворено файл з байт кодом. Але цей найпростіший спосіб створення додатків на Java стає суттєво трудомістким при розширені проектів, контролю зв'язку компонент та виявленні різноманітних помилок. Тому найчастіше для створення додатків на Java використовують IDE (Intergrated Development Environment), серед яких слід виділити Eclipse, Apache NetBeans та IntelliJ IDEA

Створення тексту програми на мові програмування є первинним етапом у її розробці. Синтаксис Java - правила для поєднання символів у мовні одиниці, у певній частині зроблено подібними до мов Smalltalk, Pascal, C та C++, які до моменту створення самої Java вже широко використовувались для створення програмного забезпечення. Найбільше подібностей саме між Java та C/C++:

- мають однакові стилі однорядкових та багаторядкових коментарів;

– багато зарезервованих слів Java ідентичні до таких у C/C++ (наприклад, for, if, switch і while) і окремо у С++ (наприклад, catch, class, public, try);
– такі примітивні типи даних Java як символи, числа з плаваючою комою одинарної та подвійної точності, а також цілі числа різної довжини використовують однакові зарезервовані слова: char, double, float, int, long, short;
– однакове позначення основних операторів: арифметичних (+, -, *, / та %) та умовних (?:);
– однакове використання символів дужок: для методів та функцій “()”, для розмежування блоків операторів “{}” та для позначення масивів “[]”.

Безумовно мова програмування Java має відмінності від C/C++ навіть у синтаксисі для підтримки певних корисних речей. Наприклад, для підтримки системи автоматичного документування Javadoc використовується додатковий стиль коментарів із так званими дескрипторами.

Платформа Java складається з віртуальної машини та середовища виконання. Віртуальна машина Java (JVM) є програмним процесором, що надає певний набір інструкцій, а середовище виконання складається з бібліотек для запуску програм і їх взаємодії з базовою операційною системою. Середовище виконання включає стандартну бібліотеку класів, які призначено для виконання типових задач, таких, наприклад, як звичні математичні операції.

Компілятор Java, за суттю, є спеціальною програмою, що перекладає код на мові програмування Java в об’єктний код, який складається з інструкцій, що виконуються JVM, та пов’язаних даних. Байт-код програми та дані компілятор зберігає у файлах із розширенням “.class”. Програма Java виконується за допомогою інструмента, який завантажує і запускає JVM і передає їй основний файл класу програми. JVM використовує свій компонент завантажувача класів для завантаження файлу класу до пам'яті. Після завантаження файлу класу верифікатор байт-коду JVM визначає дійсність байт-коду файлу класу та його безпечність для системи. Верифікатор завершує роботу JVM, якщо виявляє проблему з байт-кодом. Коли байт-код допущено до виконання, компонент інтерпретатора JVM, зазвичай, інтерпретує його по одній інструкції. Інтерпретація складається із ідентифікації інструкцій байт-коду та виконання еквівалентних інструкцій операційної системи. Певні послідовності інструкцій байт-коду, що виконуються багаторазово, можуть бути скомпільовані у рідний машинний код під час виконання. Це робиться за допомогою компілятору Just-In-Time (JIT), який є компонентом середовища виконання і призначений для покращення продуктивності програм Java під час їх виконання.

Програми Java складаються із класів, які містять нейтральні до апаратної платформи байт-коди і можуть бути інтерпретовані JVM на багатьох різних архітектурах комп'ютерів. Під час виконання JVM завантажує файли класів, визначає семантику кожного окремого байт-коду та виконує відповідні обчислення. Додаткове використання процесора та пам’яті під час інтерпретації означає, що програма Java працює повільніше у порівняні з програмою, яку можна було б реалізувати на рідному для архітектури комп’ютера машинному коді. Саме компілятор JIT допомагає підвищити продуктивність програм Java шляхом компіляції байт-кодів у рідний машинний код під час виконання.

Компіляція JIT вимагає часу процесора та використання пам’яті. При першому запуску JVM викликаються тисячі методів, компіляція яких може суттєво вплинути на час запуску. На практиці методи не компілюються під час першого виклику. Для кожного методу JVM підтримує кількість викликів, яка починається з попередньо визначеного порогового значення компіляції та зменшується щоразу, коли метод викликається. Коли кількість викликів досягає нуля, запускається компіляція JIT. Тому часто використовувані методи компілюються невдовзі після запуску JVM, а менш використовувані методи компілюються набагато пізніше або не компілюються JIT зовсім. Поріг компіляції JIT допомагає JVM швидко запуститися і при цьому мати покращену продуктивність. Порогове значення необхідно для отримання оптимального балансу між часом запуску та довгостроковою продуктивністю.

Також під час виконання інтерпретатор може отримати запит на виконання байт-коду класу з іншого файлу. Коли це відбувається, інтерпретатор просить завантажувач класів завантажити файл класу, а верифікатора байт-коду перевірити байт-код перед його виконанням. Також під час

виконання інструкції байт-коду від JVM можуть вимагати відкриття певного файлу, відображення образу на екрані або виконання іншого завдання, яке взаємодіє з рідною базовою платформою. Для забезпечення цих потреб JVM передає запит через свій міст Java Native Interface (JNI) на рідну базову платформу (рис.1).

Ще однією властивістю JVM є надання безпечного середовища, в якому виконується код, що забезпечується перевіркою байт-коду перед його виконанням. Це запобігає пошкодженню базової платформи зловмисним кодом і, можливо, крадіжці конфіденційної інформації.

Версію Java SE можна завантажити з Oracle за адресою https://www.oracle.com/java/technologies/java-se-glance.html. Щоб отримати відповідний варіант із відкритим кодом, слід скористатись посиланням https://openjdk.java.net/install/.

Каталог інсталяції містить різні файли залежно від вибраної версії, серед яких необхідно звернути увагу на такі підкаталоги як:

- bin: містить різноманітні інструменти JDK, наприклад, компілятор javac, середовище виконання програм java, засіб створення, оновлення та екстракції Java ARchive — jar, генератор документації Java - javadoc, інспектор серійних версій serialver;
- lib: містить файли бібліотеки Java та рідної платформи, які використовуються інструментами JDK.

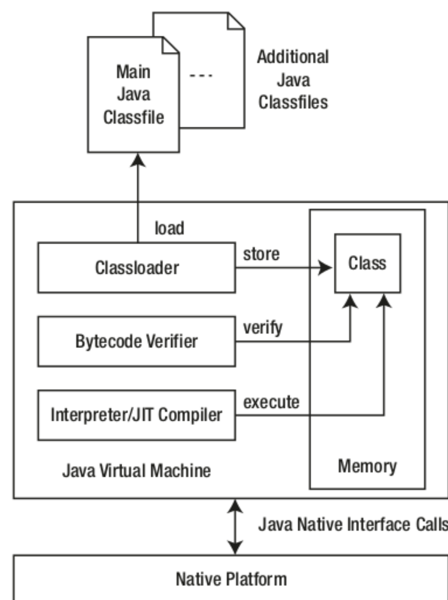

Рис. 1. JVM забезпечує усі необхідні компоненти для завантаження, верифікації, та виконання файлу класа.

Створення простої програми

Найпростішою послідовністю створення програми на Java є написання коду у текстовому редакторі, збереження коду у файлі з розширенням “.java”, компіляція коду у цьому файлі з командного рядка за допомогою javac та запуск отриманого байт-коду у середовищі виконання java.

Для написання програми можна використати будь-який текстовий редактор, що дозволяє зберегти текст у ASCII кодуванні. Тобто, якщо за замовчуванням редактор зберігає текст у файлі з розширенням txt, то скоріше за все, це саме такий редактор. Наприклад, в операційній системі Linux такими редакторами є gedit, mcedit, vi та інші. Приклад програми на Java представлено на рис.2.

```
gorbenko@gorbenko-System-Prod
class HelloWorld{

public static void main(String[] args){
        System.out.println("Hello, World!");
}
}
~
~
:w HelloWorld.java
```

Рис.2. Проста програма на Java в редакторі vi.

Загальна структура програми складається із одного класу HelloWorld, межі якого позначено фігурними дужками {} і який всередині має один метод main(). Фігурні дужки методу позначають блок операторів цього методу. Слово main для імені методу є зарезервованим і вказує на те, що саме цей метод буде першим виконуватись при запуску програми з командного рядку. Перед іменем методу знаходяться два модифікатори доступу — public та static, які вказують на те, що метод є доступним іззовні та належить виключно класу, а також є ключове слово void, що вказує на те, що метод не має певного типу, тобто ніякого значення будь-якого типу в результаті свого виконання повертати не буде.

Після написання самого коду програми на Java його необхідно зберегти у файлі, що обов'язково повинен мати таке саме ім'я, як і його клас HelloWorld та розширення java — HelloWorld.java. Така вимога в Java є обов'язковою, тому що середовище виконання java буде шукати метод main у класі, ім'я якого задається у командному рядку.

Для компіляції програми на Java у теці, в яку зберегли файл з кодом, слід набрати команду:

```
javac HelloWorld.java
```

Програма занадто проста, тому її компіляція пройде миттєво і у теці поряд з файлом з кодом програми з'явиться новий файл з байт-кодом (рис.3).

```
/..
 HelloWorld.class
 HelloWorld.java
```

Рис.3. Програма на Java та її байт-код.

Програма з байт-кодом автоматично отримує розширення .class і таке саме ім'я, що і файл з кодом. Для виконання програми на Java необхідно у середовищі виконання java виконати її байт-код, що можна зробити за допомогою наступної команди:

```
/HelloWorld$ java HelloWorld
Hello, World!
```

Останній рядок — це результат роботи програми. На відмінну від процедури компіляції середовищу виконання не потрібно вказувати розширення — воно автоматично сприймає тільки файли з розширенням .class. Ім'я файлу та регістр літер в ньому мають значення, так як вказують на клас, який повинен мати метод з іменем main. Якщо змінити, наприклад, ім'я файлу з коректним байт-кодом наступним чином:

```
$ cp HelloWorld.class helloworld.class
```

і після спробувати запустити байт-код у середовищі виконання java з його новим ім’я, то виникне помилка:

```
gorbenko@gorbenko-System-Product-Name:~/work/HelloWorld$ java helloworld
Error: Could not find or load main class helloworld
Caused by: java.lang.NoClassDefFoundError: HelloWorld (wrong name: helloworld)
```

Рис.4. Помилка виконання програми на Java, що має різні імена для файлу та класу із методом main.

Якщо змінити ім’я головного методу main() на будь-яке інше, то компіляція такого файлу помилки не виявить, тому що тільки клас, байт-код якого буде виконуватись першим у середовищі виконання java, повинен мати такий метод. Помилку буде згенеровано середовищем виконання java при спробі виконати такий байт-код.

Для швидкого створення та діагностування програми на Java під час її розробки краще використовувати одне із інтегрованих середовищ розробки, наприклад, Apache NetBeans IDE. Створення нової програми у такому середовищі починається зі створення нового проекту File → New Project і далі обирається категорія та тип проекту (рис.5).

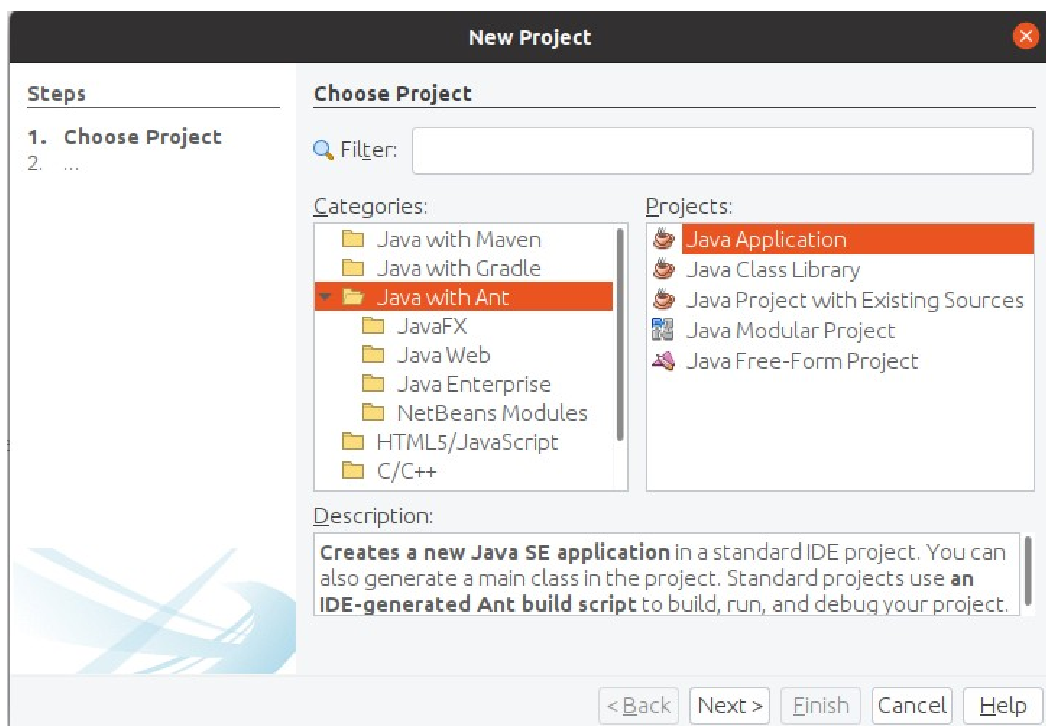

Рис.5. При створені нового проекту обирається його тип

Після натискання кнопки Next відкривається наступне вікно, в якому задається назва проекту, контролюються теки розташування проекту, вказується автоматичне створення головного класу з методом main() та, за необхідністю, під'єднуються бібліотеки із вказаної теки (рис.6). Слід звернути увагу на те, як записується головний клас. Наприклад, у даному випадку його ім’я автоматично генерується у вигляді helloworld.HelloWorld, де перша частина helloworld є іменем пакету, а друга - HelloWorld — іменем головного класу в цьому пакеті. Це відрізняється від прикладу, який було створено за допомогою простого текстового редактору і змінить не тільки ієрархію дерева класів проекту, але і розташування файлів на диску. Усі файли класів, що знаходяться у одному пакеті, будуть на диску знаходитись у одній теці з ім’ям цього пакету, якщо воно складається з одного слова. Якщо пакет має у своїй назві певну домену підпорядкованість, наприклад, ua.edu.znu.examples, то таке ім’я утворює таку саму вкладеність тек на диску: ua → edu → znu → examples.

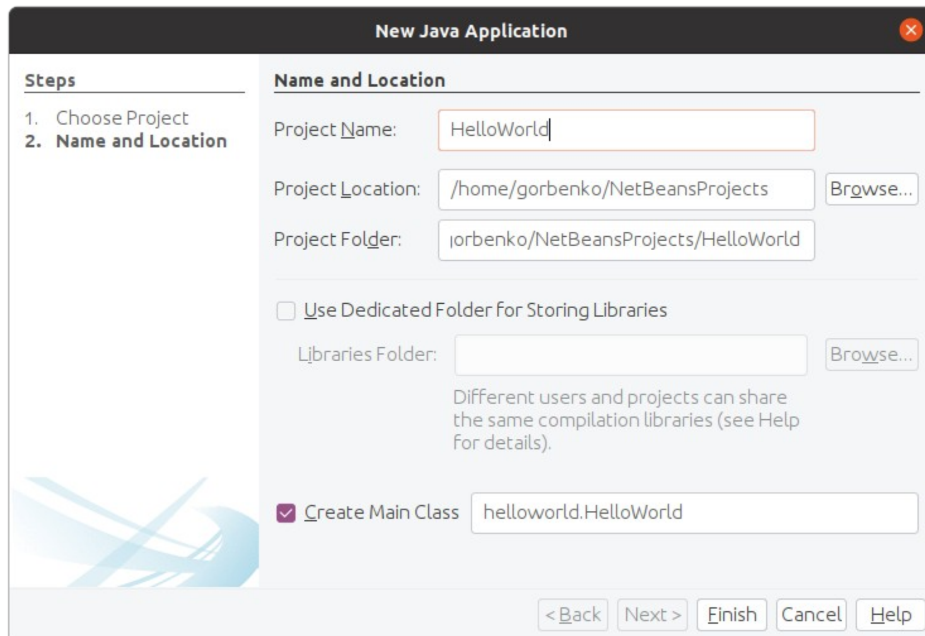

Рис.6. Вікно первинного налаштування проекту

Після натискання кнопки Finish створюється обраний тип проекту з певним шаблоном класу та його головного методу main().

```
package helloworld;

/**
 *
 * @author gorbenko
 */
public class HelloWorld {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

Рис.7. Автоматично згенерований шаблон головного класу

Шаблон додатково має не тільки посилання на пакет - package helloworld; але і автоматично згенеровані анотації документування @author та @param. При необхідності значення цих анотацій можна редагувати безпосередньо у файлі. Код методу main() можна додати замість “// TODO code application logic here” і продовжити настільки, на скільки це необхідно. Замінимо його на:

System.out.println("Hello, World!");

Для створення програми в Apache NetBeans IDE необхідно натиснути на знак “молоток”, а для її запуску — на знак “зеленого трикутника”:

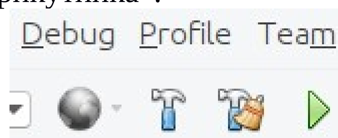

Рис.8. Кнопки побудови проекту та запуску програми.

Результат роботи програми з’явиться у вікні “Output”:

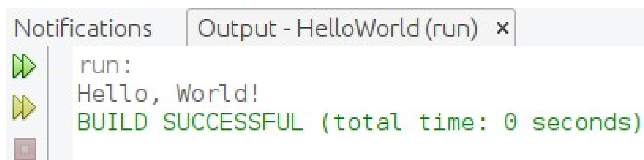

Рис.9. Результат роботи програми.

IDE контролює відповідність назви проекту, файлу головного класу та його імені. Будь-які їх зміни призведуть до відповідних підкреслень червоною хвилястою лінією та появою знаків попередження на полях. Зміна імені головного методу main() такої реакції не матиме, але спроба запустити таку програму викличе діалогове вікно з попередженням про відсутність класу з методом main(). Тому застосування IDE прискорює виявлення помилок та розробку програмного забезпечення на мові програмування Java.

Приклади програм на Java

Будь-яка мова програмування краще вивчається на прикладах рішення звичайних задач. Розглянемо декілька таких, що з одного боку демонструють властивості Java, а з іншого дають можливість побачити схожість та відмінності з іншими мовами програмування.

Перший такий приклад буде стосуватись передаванню та використанню аргументів командного рядка. Більшість додатків потребує отримання певних початкових даних для своєї роботи. Їх передавання до програми може здійснюватись через аргументи командного рядку. Розглянемо приклад, в якому у командному рядку через пробіл задаються імена, кожне з яких виводиться зі словом Hello.

```
public class HelloName {
	public static void main(String[] args) {
		for(int i=0; i < args.length; i++){
			System.out.println("Hello, " + args[i]+"!");
			}
	}
}
```

Реалізація мови програмування Java дозволяє передавати аргументи командного рядку методу main в якості аргументів args. Аргументи args є масивом типу String, кількість яких залежить від кількості слів після імені програми, при її запуску. Наприклад, якщо програму запустити наступним чином:

java HelloName Василь Петро Іванка

то її результатом буде

```
Hello, Василь!
Hello, Петро!
Hello, Іванка!
```

Усередині програми кількість елементів у масиві аргументів args визначається за допомогою його властивості: `args.length.`

Для звернення до елементів масиву використовується args[i], де значення індексу елементу змінюється відповідно до умов виконання циклу:

```
for(int i=0; i < args.length; i++)
```

тобто починаючи із значення `i=0` доки не досягне значення кількості елементів у масиві args із кроком у 1, що задається як інкремент `i++.` Знак менше < в умові циклу вказує на те, що при досягнені змінної циклу i значення `args.length` так зване тіло циклу вже виконуватись не

буде. Чому це важливо? Тому, що це не дозволить звернутись до масиву args за індексом, якого в ньому немає. У мові програмування Java, як і у багатьох інших мовах програмування, індексація елементів у масиві починається із значення 0. Тому, якщо в масиві один елемент, то його індекс буде 0, якщо два елементи — то 0 та 1, і т.д. У розглянутому прикладі довжина масиву `args.length` , тоді значення індексів його елементів починається з 0 і закінчується значенням `args.length-1.` А це означає, що в цьому масиві немає елементу з індексом `args.length` , а звернення до елементу за таким індексом викличе помилку.

Наступним розглянемо приклад додавання двох чисел, значення яких передаються у програму як аргументи командного рядку.

```
public class SumNumbers {

	public static void main(String[] args) {

		if(args.length>0) System.out.println("перший доданок ="+args[0]);
		if(args.length>1){
			System.out.println("другий доданок ="+args[1]);
			int sum =Integer.parseInt(args[0])+Integer.parseInt(args[1]);
			System.out.println("сума чисел = "+sum);
		}
	}
}
```

Числа передаються як аргументи командного рядку, тому вони будуть елементами масиву args. Але цей масив має тип String, тобто рядковий, а не числовий. Тому числа, що задаються як аргументи командного рядку будуть вважатись у програмі певною символьною послідовністю, а не числовим значенням і результатом додавання таких аргументів буде не їх сума, а конкатенація — послідовно записані цифри спачатку одного числа, а потім другого. Для того, щоб задані аргументи отримали числове значення, яке відповідає їх написаню, у наведеній програмі використано метод parseInt() класу Integer. У програмі це виглядає як: `Integer.parseInt(args[0])` та `Integer.parseInt(args[1])`. Завдяки цьому перший та другий аргументи перетворюються у числові значення. Після такого перетворення операція додавання буде арифметичною і її результатом буде сума чисел.

Після компіляції програми і її запуску, як показано нижче:

```
java  SumNumbers  12  23
```

отримаємо наступний результат

```
перший доданок = 12
другий доданок = 23
сума чисел = 35
```

Також у наведеній програмі слід звернути увагу на роботу умовного оператору if. За його допомогою написано дві умови, що контролюють правильність роботи програми, якщо аргументи не буде задано, або буде задано тільки один аргумент.

У більшості сучасних мов програмування існують оператори циклу. У мові програмування Java таких операторів два: for та while. Їх використання подібне до мов програмування С та С++. Зазвичай оператори циклу використовуються для виконання однотипових дій над елементами масивів або структур даних, що індексуються. Але цим застосування циклів не обмежується, тому що обидва оператори використовують умови для визначення продовження або завершення виконання тіла циклу.

Розглянемо задачу обчислення квадратного кореня числа за ітераційною формулою Герона:

$$X_{n+1}=\frac{1}{2}\left(X_n+\frac{a}{X_n}\right)$$

де *a* — число, корінь якого обчислюється; X_n , X_{n+1} — значення кореню після n та n+1 ітерації. Наступна програма реалізує ітерації за цією формулою:

```
public class Sqrt {
public static void main(String[] args) {
    if(args.length>0) sqrt(Double.parseDouble(args[0]));
    else System.out.println("команда задається у вигляді: java Sqrt число");
    }
      static void sqrt(double a){
         if(a>0){
            double b=a;
            int i=0;
            while ((Math.abs(b*b-a)>1.0e-12)&&(i<200)){
                  b=(b+a/b)/2;
                  i++;
            }
            System.out.println("корінь "+a+" = "+b+"  after "+i+" cycles");
         }
      }
}
```

На відміну від попередніх програм у цій програмі два методи: перший - main, який дозволяє запустити саму програму на виконання та метод sqrt, який за суттю виконує роль функції. Методи, що виконують роль функцій, можуть бути оголошені у тому самому класі, де вони потрібні і їх розташування у класі не регламентується, тобто вони можуть бути як до, так і після місця використання. Аргументом методу *sqrt* може бути або число, або змінна типу double. У методі *sqrt* ітераційний процес організовано за допомогою оператора циклу while, що має передумову `(Math.abs(b*b-a)>1.0e-12)&&(i<200)`. Цей логічний вираз має дві частини, які поєднано за допомогою скороченої буліанівської операції AND та позначено як &&. Скорочений варіант операції впливає на повноту виконання всього виразу. У даному випадку, якщо частина ліворуч отримає значення False, то частина, що праворуч, перевірятись вже не буде. Змінна *b* має тип *double*, при ініціалізації отримує значення числа, для якого обчислюється корінь, і призначена для зберігання проміжного значення кореню X_n, яке обчислюється за формулою b=(b+a/b)/2 у тілі циклу. Змінна b змінюється у кожному циклу і її нове значення залежить від попереднього значення. Умова `Math.abs(b*b-a)>1.0e-12` перевіряє досягнення точності обчислень - абсолютне значення різниці між квадратом b та числом a не може перевищувати 10^{-12} , інакше ітерації будуть продовжуватись доки їх число не досягне 200. Цілочисельна змінна *i* є лічильником циклу, тому ініціалізується значенням 0 перед оператором циклу while та збільшується на одиницю наприкінці кожної ітерації. Саме вона і перевіряється на кількість виконаних ітерацій.

Метод *sqrt* викликається у головному методі main і в якості аргументу отримує значення від першого елементу масиву args. Для переведення значення елементу масиву із типу String у тип double використовується метод Double.parseDouble(). Обчислення квадратного кореню можливо тільки для чисел більше 0, що перевіряється у методі sqrt.

Слід відзначити, що оператори циклів for та while є взаємозамінними і їх використання стосується зручності розробника. Але для більшої прозорості коду зазвичай оператор for використовують, якщо необхідно реалізувати цикл із заданим числом ітерацій, а оператор while використовують для випадків, коли кількість циклів важко передбачити, тому що це залежить від умови в операторі.

Розглянемо програмну реалізацію методу Лейбніца з обчислення числа π.

У 1671 році Джеймс Грегорі встановив, що в області визначення тангенсу:

$$\Theta = tg\Theta - \frac{1}{3}tg^3\Theta + \frac{1}{5}tg^5\Theta - \frac{1}{7}tg^7\Theta + \frac{1}{9}tg^9\Theta \pm \dots$$

Цей результат дозволив Лейбніцу отримати достатньо простий вираз для обчислення числа π, якщо прийняти що $\Theta = \frac{\pi}{4}$

$$\frac{\pi}{4}=tg\frac{\pi}{4}-\frac{1}{3}tg^3\frac{\pi}{4}+\frac{1}{5}tg^5\frac{\pi}{4}-\frac{1}{7}tg^7\frac{\pi}{4}+\frac{1}{9}tg^9\frac{\pi}{4}\pm\ldots$$

$$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}\pm\ldots$$

і після множення на 4:

$$\pi=4-\frac{4}{3}+\frac{4}{5}-\frac{4}{7}+\frac{4}{9}\pm\ldots$$

У цьому виразі знаменник доданків - це послідовні непарні числа, значення яких може бути обчислено за формулою ($2\times n$-1), де $n \in N$.

Наведений ряд є принципово нескінченим, тому отримати суму нескінченого числа доданків неможливо, але можна обчислити число π з певною точністю, тобто для обмеженої кількості доданків. Наприклад, обчислення можна виконувати до тих пір, поки сума усіх членів, що залишились не буде перевищувати 10^{-10}. Наведений ряд є знакозмінним і тим, що сходиться, тому для формулювання як застосувати критерій точності обчислень запишемо його у вигляді

$$\pi=\left(4-\frac{4}{3}\right)+\left(\frac{4}{5}-\frac{4}{7}\right)+\left(\frac{4}{9}-\frac{4}{11}\right)+\ldots$$

Тепер маємо у ряді доданки, що представляють собою різницю між двома дробами, чисельник яких дорівнює 4, а знаменник — непарне число. Кожний наступний такий доданок є меншим за значенням до свого попередника. Тому критерій точності обчислень можна застосувати саме до величини доданка — якщо його значення стає меншим за критерій точності обчислень, то їх можна зупиняти. Нижче наведено програму, що реалізує цей метод:

```
public class CalcPi {
public static void main(String[] args) {
    if(args.length>0)
     {System.out.println("задана точність: "+Double.parseDouble(args[0]));
      leibnic(Double.parseDouble(args[0]));}
    else
      System.out.println("use a comand line like: java CalcPi 1e-6");
}
static void leibnic(double a){
    double pi=0,d1,d2;
    int i=0;
    do{
        i++;
        d1=4.0/(2*i-1);
        pi+=d1;
        i++;
        d2=4.0/(2*i-1);
        pi-=d2;
    }while((d1-d2)>a);
    System.out.println("Число Пі (метод Лейбніца): "+pi);
    System.out.println("точність: "+(d1-d2));
    System.out.println("зроблено циклів: "+i);
    }
}
```

Після компіляції програми і її запуску, як показано нижче:

java CalcPi 2.0e-18

отримаємо наступний результат

}

```
задано точність: 2.0E-18
Число Пі (метод Лейбніца): 3.1415926525880504
точність: 1.9999998848322568E-18
зроблено циклів: 999999950
```

Для реалізації методу Лейбніца був застосований оператор циклу do/while, особливістю якого є перевірка умови після виконання блоку операторів циклу

```
do{
    //блок операторів
}while(умова)
```

Необхідність такого оператору циклу обумовлено реалізацією методу Лейбніца — потрібно обчислити значення принаймні двох членів ряду, щоб визначити виконання критерію точності.

У багатьох задачах для зберігання даних та виконання одного набору операцій над даними використовують масиви. Масив є локалізованою групою змінних одного типу, на які посилаються через загальне ім'я — ім'я масиву. Можна створювати масиви будь-якого типу, що підтримується мовою програмування Java або створюється її засобами, і вони можуть мати один або кілька вимірів. Доступ до певного елемента в масиві здійснюється за його індексом, що є його звичайним атрибутом. Масиви надають зручний засіб групування пов'язаної інформації.

Одновимірний масив, по суті, є списком змінних одного типу, зв'язок між яким підтримується на рівні індексів елементів. Для створення масиву, спочатку створюється змінна масиву потрібного типу. Тип змінної масиву визначає тип даних кожного елемента, який містить масив.

Розглянемо принципи використання масиву на прикладі генерування значень його елементів, пошуку його найменшого та сортування масиву. Нижче наведено програму на Java, що реалізує ці методи

```
public class Arrays {

public static void main(String[] args) {
      double x=0;
      final int n = 20;
      double[] array = new double[n];
      System.out.print("Згенеровано масив: ");
      for (int i=0;i<array.length;i++){
            array[i]=Math.round(50*Math.random());
            System.out.print(array[i]+" ");
      }
      System.out.print("\nМінімальний елемент: "+minX(array));
      sort(array);
}

public static double minX(double[] array){
      double x=array[0];
      for(int i=1;i<array.length;i++){
         if(x>array[i]){
            x=array[i];
         }
      }
      return x;
}

public static void sort(double[] array){
      double temp;
      System.out.print("\nМасив після сортування: ");
        for(int i=0;i<array.length-1;i++){
            for(int j=i+1;j<array.length;j++){
```

```
                if(array[j]<array[i]){
                    temp=array[j];
                    array[j]=array[i];
                    array[i]=temp;
                }
            }
        System.out.print(array[i]+", ");
        }
    }
}
```

Методи пошуку мінімального елементу в масиві та сортування масиву за алгоритмом прямого упорядкування в класі *Arrays* реалізовано окремими методами: minX() та sort(). Оголошення, створення та генерація значень елементів масиву реалізовано у методі main. Оголошення та створення масиву може бути поєднано, як і у наведеній програмі

```
double[] array = new double[n];
```

або роз'єднано — спочатку робиться оголошення масиву десь на початку програми, а потім — створення масиву, перед тим, як його будуть використовувати

```
double[] array;
....
array = new double[n];
```

При оголошені масиву створюється певний запис, в якому є і ім'я масиву, і параметр, що отримає посилання на майбутній масив. Структура масиву утворюється тільки при його створені, наприклад, за допомогою оператора new, одночасно з ім'ям масиву пов'язується адреса на цю структуру. У наведеному прикладі кількість елементів у масиві задається безпосередньо при його створені у квадратних дужках `double[n]`. Достатньо часто для ініціалізації елементів масиву застосовують цикл, де змінну циклу можна використати як індекс елементу. У випадках, коли необхідно перевірити роботу алгоритмів на основі масивів числових типів, елементи масиву можна ініціалізувати за допомогою випадкових чисел. Наприклад, як у наведеному вище

```
array[i]=Math.round(50*Math.random()),
```

де метод Math.random() має тип double та повертає випадкове число у діапазоні від 0.0 до 1.0; множник 50 потрібен для розширення діапазону від 0.0 до 50.0; метод Math.round() у даному випадку використовується для створення більшої наочності — значення елементів округляються до найближчого цілого і виведення значень елементів стає компактним.

В якості аргументів у методах minX() та sort() задано масив array, але цим методам передаються не масиви, а посилання на них. Тому, будь-які операції над елементами масиву всередині методу в дійсності будуть виконуватись над за тими самими адресами, де знаходяться елементи масиву зовні. Метод minX() має тип double, тому завершується оператором return, що повертає значення найменшого елементу. Метод minX() працює як функція — за отриманими аргументами повертає певне значення. Метод sort() має тип void, тому нічого не повертає, але всередині переставляє елементи масиву за певним алгоритмом. Тому що елементи масиву всередині цього методу і ззовні мають однакові адреси, то сортування елементів всередині методів приводить до їх сортування зовні. Завершення роботи методу sort() - це завершення процедури сортування заданого масиву без будь-якого додаткового передавання даних.

Крім одновимірних масивів у Java підтримуються масиві із більшою розмірністю. У більшості практичних задач масиви із розмірністю вище другої використовуються не часто. Двовимірний масив у Java технічно реалізується як одновимірний масив з елементами, які самі є

одновимірними масивами. Оголошення двовимірного масиву є подібним до відповідної процедури одновимірного масиву, але використовується вже дві пари квадратних дужок, наприклад:

```
тип[][] ім’я = new тип[розмір_1][розмір_2];
```

де розмір_1 вказує на кількість рядків у двовимірному масиві, а розмір_2 — на кількість елементів у рядку.

Подібно до одновимірного масиву створення двовимірного масиву може бути розділено на дві частини: його оголошення:

```
тип[][] ім’я;
```

та безпосереднє створення

```
ім’я = new тип[розмір_1][розмір_2];
```

Для ініціалізації елементів такого масиву використовуються вкладені цикли за розмірністю масиву, тобто для двовимірного масиву це буде два цикли: верхній цикл буде відповідати за зміну одного індексу, наприклад, індексу рядка, а цикл всередині його — за зміну індексу елемента у рядку.

У випадку, коли значення елементів масиву є відомими та не будуть змінюватись, а також їх кількість є припустимою для безпосереднього набору у коді програми, то такий масив може бути ініціалізованим за допомогою списків значень. Наприклад

```
double data[][]={{0.1,0.2,0.3},{0.4,0.5,0.6}};
```

Ця команда створює та ініціалізує двовимірний масив `data` із розмірами 2 на 3. Масив `data` є масивом із двох елементів, кожний із яких є масивом із трьох елементів. Елемент `data[0][0]` отримає значення 0.1, а елемент `data[0][2]` — значення 0.3, елемент `data[1][0]` — значення 0.4 , а елемент `data[1][2]` — значення 0.6.

У зв’язку з особливостями реалізації багатовимірних масивів, вони можуть мати різни значення в межах одного виміру. Наприклад, двовимірний масив може бути створений з різною довжиною рядків:

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

Зазвичай, це корисно при створені трикутних масивів, але реалізація в Java цим не обмежується і можна створювати масиви із будь-якою структурою і навіть із пустими рядками всередині масиву:

```
double data[][]={{0.1,0.2,0.3},{0.4,0.6},{0.4,0.6,0.7,0.8},{},{1.0,1.1}};
for(double k1[] : data){
   for(double k2 : k1){
      System.out.print(" "+k2);
   }
   System.out.println("");
}
```

У наведеному прикладі демонструється можливості, щодо створення масиву з різною довжиною рядків та показано принципи поелементного розбирання такого масиву. Дійсно, для цього краще підходить цикл типу for-each, при використані якого не потрібно вказувати довжину рядків або кількість стовпців, тощо. Але слід звернути увагу на те, що, наприклад, двовимірний

масив спочатку розбирається на рядки — у наведеному прикладі k1[], а потім у кожному окремому рядку іде вибірка його елементів. Результатом роботи цього коду буде

```
0.1 0.2 0.3
0.4 0.6
0.4 0.6 0.7 0.8

1.0 1.1
```

Подібне відбувається і при визначені розмірів масиву. Для наведеного прикладу data.length має значення 5 — вказує на кількість рядків разом із тим, у якому не задано ні одного елементу. Для отримання довжини рядка — кількості елементів у рядку, необхідно отримати значення такого параметру, але для рядків. Наприклад, data[0].length має значення 3 — кількість елементів у першому рядку, рядку з індексом 0.

У теорії суцільного середовища використовується тензор Леві-Чевіта, який може бути представлений тривимірним масивом. За визначенням, тензор Леві-Чевіта

$$\varepsilon_{ijk}=\begin{cases}+1 & P(i,j,k)=+1\\ -1 & P(i,j,k)=-1\\ 0 & i=j\vee j=k\vee k=i\end{cases}$$

де $P(i,j,k)=+1$ вказує на парне переставлення індексів, а $P(i,j,k)=-1$ - на непарне. Іншими словами, якщо у елемента тензора хоча б два індекси однакові, то його значення 0, елементи з індексами (1,2,3),(2,3,1) та (3,1,2) мають значення 1, а елементи з індексами (3,2,1), (1,3,2) та (2,1,3) мають значення -1.

Для ініціалізації такого масиву можна використати як оператори циклу, так і спосіб безпосередньої ініціалізації. Перший спосіб реалізується наступним чином:

```
public class Tensor {

      public static void main(String[] args){
            int i,j,k;
            byte[][][] epsilon=new byte[3][3][3];
            for(i=0;i<3;i++)
                  for(j=0;j<3;j++)
                        for(k=0;k<3;k++)
                              epsilon[i][j][k]=0;
            epsilon[0][1][2]=epsilon[1][2][0]=epsilon[2][0][1]=1;
            epsilon[1][0][2]=epsilon[0][2][1]=epsilon[2][1][0]=-1;
                for(byte[][] x1: epsilon){
                    for(byte[] x2: x1){
                        for(byte x3: x2){
                            System.out.print(x3+" ");
                        }
                        System.out.println();
                    }
                    System.out.println();
                }
            }
}
```

Також цей тензор може бути реалізовано за допомогою списків значень

```
public class Tensor {

      public static void main(String[] args){
```

```
        byte[][][] epsilon={{{0,0,0},{0,0,1},{0,-1,0}},
                            {{0,0,-1},{0,0,0},{1,0,0}},
                            {{0,1,0},{-1,0,0},{0,0,0}}
        };
        for(byte[][] x1: epsilon){
            for(byte[] x2: x1){
                for(byte x3: x2){
                    System.out.print(x3+" ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Останній спосіб виконує менше операцій — оголошення масиву та пряма ініціалізація його елементів, а також код є більш прозорим. Зрозуміло, що другий спосіб краще застосовувати для невеликих масивів і коли значення елементів у них відомі.

Крім числових масивів у Java широко використовуються символьні масиви. Зазвичай їх оголошення та ініціалізація мало чим відрізняється від числових. У наступному прикладі наведено простий спосіб створення символьного масиву:

```
class CharArray{
      public static void main(String[] args){
            char[] words=new char[]
            {'С','и','м','в','о','л','ь','н','и','й',' ','м','а','с','и','в'};
            System.out.println(words);
      }
}
```

Це стандартний спосіб створення символьного масиву: одночасно з його оголошенням виконується ініціалізація списком символів елементів. Завдяки автоматичному перетворенню символьний масив можна вивести у текстовому форматі за допомогою методу `System.out.println(words)`.

Мова програмування Java має достатньо жорсткі обмеження до використання типів даних: будь-які операції зі змінними можуть бути виконаними, якщо змінні одного типу, або приведені до одного типу. Символьний масив можна використати для розділення змінної або константи типу String на окремі символи:

```
String s1 = "Цей рядок буде розділено на окремі символи";
char[] words = s1.toCharArray();
System.out.println(words);
```

Кожний елемент масиву words ініціалізується окремим символом із рядка s1. Як уже було зауважено вище, з іменем масиву зв’язується адреса пам’яті, де розташовано елементи масиву. У мові програмування Java ця область має певний розмір, який визначається або заданою кількістю елементів, або значенням, що задається при його ініціалізації. Звернення до будь-якого елементу цього масиву за індексом, що перевищує або дорівнює розміру масиву, призведе до помилки. У Java не можна звертатись до елементів за встановленими межами масиву. За такими діями слідкує середовище виконання. Але це не означає, що ім’я масиву назавжди зв’язано з областю первинної ініціалізації. З іменем оголошеного масиву можна зв’язувати інші області, тобто інші масиви, навіть якщо кількість елементів у них різна. Наприклад, у наведеному нижче коді масив word ініціалізується для п’яти елементів, а далі значення його елементів та його розмір змінюються у відповідності до змінної s1. Якщо при ініціалізації масив words мав п’ять елементів із значеннями

'С','л','о','в','о', то далі його розмір збільшується, а значення елементів змінюються у відповідності до змінної s1.

```
char[] words=new char[]{'С','л','о','в','о'};
System.out.println(words);
String s1 = "Цей рядок буде розділено на окремі символи";
words = s1.toCharArray();
System.out.println(words);
```

Недоліком цього є неможливість повернути попередні параметри та значення елементів масиву, якщо вони не зберігались в іншому масиві. Фактично утворюється ще одне посилання для того ж самого масиву.

Наступний приклад демонструє, що ім’я масиву зв’язується з посиланням на локалізацію в пам’яті елементів:

```
char[] words3;
char[] words4;
String s1 = "Цей рядок буде розділено на окремі символи";
String s2 = "Цей рядок буде розділено на окремі символи";
System.out.println(s1.equals(s2));
words3 = s1.toCharArray();
words4 = s2.toCharArray();
System.out.println(words3.length);
System.out.println(words4.length);
System.out.println(words3.equals(words4));
```

Результатом його роботи буде:

```
true
42
42
false
```

Тобто дві змінні типу String s1 та s2 мають однакові значення і їх порівняння за допомогою методу equals() повертає значення true. Але якщо їх використати для ініціалізації елементів двох масивів words1 та words2, то кількість елементів у масивах - 42 та їх значення будуть однаковими, але порівняння масивів за допомогою методу equals() поверне значення false. Це відбувається тому, що з іменами масивів пов’язані не їх елементи, а посилання на їх локалізацію у пам’яті. Порівняння відбувається не над масивами, а над посиланнями, посилання різні — реально існують дві структури — тому результатом порівняння є false.

Для порівняння масивів за їх вмістом слід використовувати метод equals класу Arrays з пакету java.util. Для наведеного вище прикладу наступний рядок дозволить виконати порівняння масивів за їх вмістом і поверне значення true:

```
System.out.println(Arrays.equals(words3, words4));
```

За допомогою оператору привласнення “=” можна передавати посилання для масивів будь-яких типів. Середовище виконання контролює відповідність типів масивів ліворуч та праворуч від оператору “=”, але дозволяє змінювати розмір масивів. Це розширює можливості використання масивів і забезпечує достатньо зручний спосіб передавання посилання на елементи масивів всередину методів.

Завдання до лабораторної роботи

1. Встановіть необхідне програмне забезпечення на комп’ютер (JDK, IDE). Вивчіть приклади, які представлено у теоретичній частині.
2. Напишіть програму обчислення довжини гіпотенузи із значень катетів, які задаються як аргументи командного рядку.
3. Напишіть програму рішення рівняння $A^3+B^3+C^3=ABC$ методом перебирання (A,B,C є цифрами числа ABC).
4. Напишіть програму, в якій за допомогою циклу обчислюється 2^n. Яку максимальну ступінь можна обчислити? Чим це обумовлено?
5. Напишіть програму для обчислення факторіалу довільного числа n. Факторіал якого максимального числа можна обчислити? Чим це обумовлено?
6. Напишіть програму, що перевіряє істинність теореми Ферма: для $a,b,c\in\mathbb{N}$ не існує таких значень, що задовольнять рівнянню $a^n+b^n=c^n$. У програмі передбачте, що значення a,b,c буде змінюватись від 1 до 100 та $2<n<10$. Перевірте роботу програми для n = 2. Скільки рішень знайдено у цьому випадку?
7. Напишіть програму, що для заданого числового масиву дозволяє міняти місцями елементи з максимальним та мінімальним значенням.
8. У звіті наведіть код цих прикладів та скрин-шоти результатів роботи цих програм.