2. ОСНОВИ МОВИ ПРОГРАМУВАННЯ JAVA

2.1. Структура програми та її елементи: змінні, константи, типи даних, коментарі, введення, виведення, перетворення типів.

Основним будівельним блоком програми на мові Java є **оператор (statement)**. Оператор виконує деяку дію, наприклад, виклик методу, оголошення змінних, присвоєння змінній значення. Після кожного оператора в Java ставиться крапка з комою (;). Також поширеною конструкцією є блок коду. Блок коду містить набір операторів, які містяться між відкриваючою та закриваючою фігурними дужками:

```
{
   System.out.println("Привіт!");
   System.out.println("Ласкаво просимо до Java!");
}
```

Java є об'єктно-орієнтованою мовою, тому всю програму можна представити як набір взаємодіючих між собою класів та об'єктів. Візьмемо як шаблон певний вигляд програми:

```
public class Program {
   public static void main(String args[]) {
       System.out.println("Привіт Java!");
   }
}
```

Вхідною точкою програми на мові Java є **метод *main***, який визначений в класі *Program*. Саме з нього починається виконання програми. Його присутність в програмі є обов'язковою, при цьому заголовок повинен бути тільки таким, як вище. При запуску програми віртуальна машина Java шукає в головному класі програми метод *main*, і після його виявлення запускає його.

Програма може містити **коментарі**. Коментарі дозволяють пояснити суть програмного коду, та зробити його більш читабельним. При компіляції коментарі ігноруються і не мають жодного впливу на роботу програми та її розмір.

У Java є два типи коментарів: однорядковий та багаторядковий. Однорядковий коментар розміщується на одному рядку після подвійного слеша //, а багаторядковий коментар розміщується між символами /* текст коментаря */.

Для зберігання даних у програмі призначені **змінні**. Останні представляють іменовану область пам'яті, яка зберігає значення конкретного типу. Кожна змінна має тип, ім'я та значення. Тип визначає, яку інформація може зберігати змінна та діапазон допустимих значень.

Ім'я змінної повинно задовольняти наступним вимогам:

- може містити будь-які алфавітно-цифрові символи, а також символи підкреслення, при цьому перший символ не повинен бути цифрою;
- немає містити знаків пунктуацій та пробілів;
- не може бути ключовим словом мови Java.

Крім цього, при оголошенні та наступному використанні змінних потрібно враховувати регістрозалежність мови, тобто змінні *s* та *S*, вважатимуться різними. Можна визначити значення змінної при її оголошенні. Цей процес називається **ініціалізацією**:

```
int a = 10; // оголошення і ініціалізація змінної
```

Крім цього, для ініціалізації змінних можна використовувати ключове слово *var*:

```
var a = 10;
```

Ключове слово *var* пишеться замість типу даних, а сам тип змінної виводиться з того значення, що в даний момент їй присвоєно.

Окрім змінних в Java для зберігання даних можна використовувати **константи**. Останні дозволяють присвоїти значення лише один раз. Прийнято, називати константи іменами у верхньому регістрі. Константа оголошується так само, як і змінна, але з ключовим словом *final*:

```
final int L = 5;
```

Особливістю Java є те, що дана мова є строго типізованою. Кожна змінна чи константа представляє конкретний тип і даний тип є строго визначений. Тип даних визначає діапазон значень, який може зберігатися в змінній чи константі.

Система вбудованих базових типів даних представлена наступними типами:

- *boolean* (зберігає значення true або false);
- *char* (зберігає одиночний символ в кодуванні UTF-16 і займає 2 байта, тому діапазон збережених значень від 0 до 65 535);
- *byte* (зберігає ціле число від -128 до 127 і займає 1 байт);
- *short* (зберігає ціле число від -32768 до 32767 і займає 2 байта);
- *int* (зберігає ціле число від -2147483648 до 2147483647 і займає 4 байта);
- *long* (зберігає ціле число від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 і займає 8 байт);
- *float* (зберігає число з плаваючою крапкою від -3.4 * 1038 до 3.4 * 1038 і займає 4 байта);
- *double* (зберігає число з плаваючою крапкою від ± 4.9 * 10-324 до ± 1.8 * 10308 і займає 8 байт).

Як роздільник цілої та дробової частини в дрібних літералах змінних типу *double* та *float* використовується крапка. Змінна оголошена певним типом може приймати тільки ті значення, які відповідають її типу. Якщо змінна представляє цілочисельний тип, то вона не може зберігати дробові числа.

Всі цілочисельні літерали, наприклад, числа 10, 4, -5, сприймаються як значення типу *int*, проте ми можемо присвоювати цілочисельні літерали іншим цілочисельним типам: *byte, long, short*. В цьому випадку Java автоматично здійснює відповідні перетворення:

```
byte x = 1; short y = 2; long z = 2121;
```

При присвоєнні змінній типу *long* дуже великого числа, яке виходить за межі допустимих значень виникне помилка на етапі компіляції. Для присвоєння таких великих чисел потрібно додати до числа суфікс l або L, який вказуватиме, що число представляє тип *long*:

```
long n = 2147483649L;
```

Як правило, значення для цілочисельних змінних задаються у десятковій системі числення, однак можна застосовувати і інші системи числення, наприклад:

```
int n13 = 0b1101; // двійкова система, число 13
```

Для задання шістнадцяткового числа після символів 0x вказується число в шістнадцятковому форматі. Таким же чином вісімкове значення вказується після символу *0*, а двійкове число – після символів *0b*.

При присвоєнні змінній типу *float* дрібного літерала з плаваючою крапкою, наприклад, 3.1, 4.5, Java автоматично розглядає цей запис як значення типу *double*. І щоб вказати, що дане значення має розглядатися як *float*, потрібно використати суфікс *f*:

```
float x = 30.6f;
```

В якості значення змінна символьного типу отримує одиночний символ в одинарних лапках:

```
char c = 'e';
```

Крім того, змінній символьного типу також можна присвоїти цілочисельне значення від 0 до 65535. У цьому випадку змінна знову ж буде зберігати символ, а цілочисельне значення буде вказувати на номер символу в таблиці символів Unicode (UTF-16), наприклад:

```
char c = 102; // символ 'f'
```

Символьні змінні не варто плутати із рядковими, тобто символ 'a' не ідентичний рядку "a". Рядкові змінні представляють об'єкт *String*, який на відміну від *char* або *int* не є примітивним типом в Java:

```
String h = "Hello ...";
```

Найбільш простий спосіб взаємодії з користувачем представляє **консоль**. На останню можна виводити деяку інформацію або, навпаки, зчитувати з неї деякі дані. Для взаємодії з консоллю в Java застосовується **клас *System***, а його функціональність забезпечує **консольне введення та виведення**.

Для створення потоку виведення в класі *System* визначений об'єкт out. В цьому об'єкті визначено **метод *println***, який дозволяє вивести на консоль деяке значення з подальшим переведенням курсору консолі на наступний рядок. Наприклад:

```
public class Program {
   public static void main (String [] args) {
      System.out.println("Привіт світ!");
      System.out.println("Пока світ ...");
   }
}
```

При необхідності можна і не переводити курсор на наступний рядок. В цьому випадку можна використовувати **метод *System.out.print()***, який відрізняється від *println* лише тим, що не переводить курсор на наступний рядок. Але за допомогою методу *System.out.print* також можна здійснити перевід каретки на наступний рядок. Для цього треба використати символ \n:

```
System.out.print("Привіт світ \n");
```

Часто виникає необхідність підставляти в рядок виводу дані. Наприклад, є дві змінні, і їх значення потрібно вивести на екран. У цьому випадку, можна написати так:

```
System.out.println("a =" + a + "; b =" + b);
```

Крім цього, в Java є функція для форматованого виведення: *System.out.printf()*. З її допомогою можна переписати попередній рядок коду наступним чином:

```
System.out.printf("a =%d; b =% d \ n", a, b);
```

В даному випадку символи *%d* позначають специфікатор, замість якого підставляється один з аргументів. Специфікаторів і відповідних їм аргументів може бути багато. В даному випадку, тільки два аргументи, тому замість першого *%d* підставляється значення змінної x, а замість другого – значення змінної y. Сама буква *d* означає, що даний специфікатор буде використовуватися для виведення цілочисельних значень.

Крім специфікатора *%d* можна використовувати ще ряд специфікаторів для інших типів даних:

- *%x* для виведення шістнадцяткових чисел;
- *%c* для виведення одиночного символу;
- *%e* для виведення чисел у експоненційній формі, наприклад, *1.3e+01*;
- *%f* для виведення чисел з плаваючою точкою;

- *%s* для виведення рядкових значень.

Наприклад:

```
public class Program {
   public static void main(String [] args) {
      String n = "Vasia";
      int  a= 23;
      float h = 1.8f;
      System.out.printf("Ім'я:%s Вік:%d Висота:%.2f \n", n, a, h);
   }
}
```

При виведенні чисел з плаваючою крапкою можна вказати кількість знаків після крапки, для цього потрібно використати специфікатор *%.2f*, де *.2* вказує, що після коми буде два знаки.

Для введення в консолі в класі *System* визначено об'єкт *in*. Однак безпосередньо через об'єкт *System.in* не дуже зручно працювати, тому, як правило, використовують **клас *Scanner***, який, в свою чергу використовує *System.in*. Наприклад, наступна програма виконує введення числа:

```
import java.util.Scanner;
public class Program {
   public static void main (String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("Введіть ціле число:");
      int a = in.nextInt();
      System.out.printf("Ваш номер:%d \n", a);
      in.close();
   }
}
```

Оскільки клас *Scanner* знаходиться в пакеті *java.util*, то потрібно здійснити його імпорт, що і зроблено у програмі. Для створення самого об'єкта *Scanner* в його конструктор передається об'єкт *System.in*. Після цього можна отримати введені значення. Наприклад, в даному випадку спочатку виводимо запрошення для вводу, а потім записуємо введене число в змінну *num*. Щоб отримати введене число, використовується метод *in.nextInt()*, який повертає введене з клавіатури цілочисельне значення. Клас *Scanner* має ще ряд методів, які дозволяють отримати введені користувачем значення:

- *next()* зчитує введений рядок до першого пробілу;
- *nextBoolean()* зчитує значення boolean;
- *nextLine()* зчитує весь введений рядок;
- *nextInt()* зчитує введене число int;
- *nextByte()* зчитує введене число byte;
- *nextShort()* зчитує введене число short;
- *nextFloat()* зчитує введене число float;
- *nextDouble()* зчитує введене число double.

Кожен базовий тип даних займає певну кількість байт пам'яті. Це накладає обмеження на операції, в які залучені різні типи даних. Розглянемо наступний приклад:

```
int x = 4; byte y = x;
```

У коді вище, виникне помилка. Хоча і тип *byte*, і тип *int* представляють цілі числа. Більше того, значення змінної *x*, яке присвоюється змінній типу *byte* потрапляє в діапазон значень для типу *byte* (від -128 до 127). Але виникне помилка, оскільки в даному випадку є спроба присвоїти деякі дані, які займають 4 байти, змінній, яка займає всього один байт. Проте в програмі може знадобитися щоб подібне перетворення було виконано. У цьому випадку необхідно використовувати **операцію перетворення типів**:

```
int x = 4;
byte y = (byte) x; // перетворення типів: від типу int до типу byte
```

Операція перетворення типів передбачає вказівку в дужках того типу, до якого треба перетворити значення. Наприклад, у випадку операції *(byte) x*, виконується перетворення даних типу *int* в тип *byte*, в результаті буде значення типу *byte*.

Коли в одній операції залучені дані різних типів, не завжди необхідно використовувати операцію перетворення типів. Деякі види перетворень виконуються неявно, тобто автоматично (рис. 2.1).

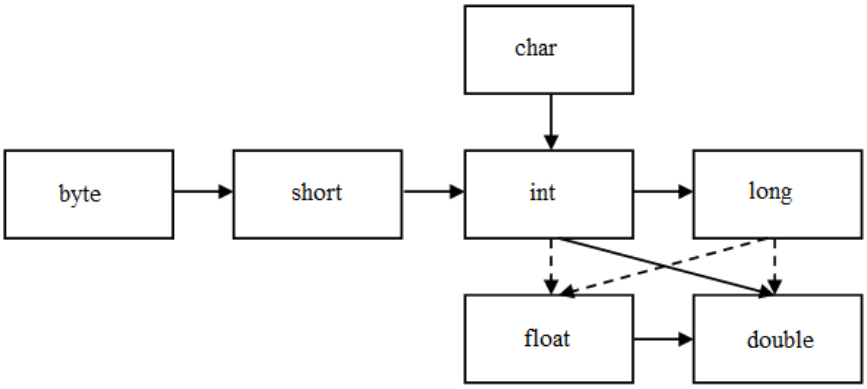

Рис. 2.1. Схема автоматичного перетворення типів даних

Стрілками показані (рис. 2.1) перетворення типів даних, які можуть виконуватися автоматично, наприклад:

byte x = 7;

int y = x; // перетворення від byte до int

Пунктирними стрілками показані автоматичні перетворення з втратою точності, наприклад:

int x = 2147483647;

float y = x; // від типу int до типу float

System.out.print(y); // 2.14748365E9

У всіх інших перетвореннях примітивних типів явно застосовується операція перетворення типів. При застосуванні явних перетворень можна також зіштовхнутися з втратою даних. Наприклад, у наступному випадку буде втрата даних:

int x = 258;

byte y = (byte)x;

System.out.println(y); // 2

Результатом, тобто виведеним на екран числом, буде число 2. У даному випадку число 258 у межах дії для типу *byte* (від -128 до 127), тому станеться усічення значення.

Можливі ситуації, коли доводиться застосовувати різні операції, наприклад, сума та добуток, над значеннями різних типів. Тут, також діють деякі правила:

- якщо один з операндів операції відноситься до типу *double*, то і другий операнд перетвориться до типу *double*;
- якщо попередня умова не виконується, а один з операндів операції відноситься до типу *float*, то і другий операнд перетвориться до типу *float*;
- якщо попередні умови не виконуються, один з операндів операції відноситься до типу *long*, то і другий операнд перетвориться до типу *long*;
- інакше всі операнди операції перетворюються до типу *int*.

2.2. Операції мови Java.

Більшість операцій в Java аналогічні тим, які застосовуються в інших С-подібних мовах. Операції розрізняють за кількістю операндів: унарні виконуються над одним операндом, бінарні – над двома, а також тернарні – над трьома. Операндом є змінна або значення (наприклад, число), яка бере участь в операції. Розглянемо всі види операцій.

Арифметичні операції виконуються з даними, що містять числа. У Java є бінарні та унарні арифметичні операції. До бінарних операцій відносять наступні:

\+ (операція додавання двох чисел);

\- (операція віднімання двох чисел);

* (операція множення двох чисел);

/ (операція ділення двох чисел);

% (отримання цілочисельного залишку від ділення двох чисел).

При діленні варто враховувати, що якщо в операції беруть участь два цілих числа, то результат ділення буде округлятися до цілого числа, навіть якщо результат присвоюється змінній *float* чи *double*:

double k = 10/4; // результат 2

Крім цього, є дві унарні арифметичні операції, які проводяться над одним числом: ++ (інкремент) – збільшення числа на одиницю та --(декремент) – зменшення числа на одиницю. Кожна з цих операцій є префіксною та постфіксною. Префіксна операція передбачає спочатку збільшення/зменшення числа на одиницю, після чого обчислення виразу, а постфіксна, навпаки, спочатку обчислення виразу, після чого збільшення/зменшення числа на одиницю. Наприклад, при *y=5* змінні *z=y++* та *x=++y* матимуть різні значення, відповідно, *z=5* та *x=6*.

Одні операції мають більший пріоритет, ніж інші, і тому виконуються спочатку. **Операції в**

порядку зменшення пріоритету:

- ++ (інкремент), -- (декремент);
- * (множення), / (ділення), % (залишок від ділення);
- + (додавання), - (віднімання).

Дужки дозволяють перевизначити порядок обчислень, наприклад:

int x = 8; int y = 7;
*int z = (x + 5) * ++y;*
System.out.println (z); // результат: 104

Незважаючи на те, що операція додавання має менший пріоритет, але спочатку буде виконуватися саме вона, а не множення, тому що операція суми вкладена в дужки.

Крім пріоритету, операції відрізняються за **властивістю асоціативності**. Коли операції мають один і той же пріоритет, порядок обчислення визначається асоціативністю операторів. Залежно від асоціативності є два типи операторів:

- лівоасоціативні (виконуються зліва направо);
- правоасоціативні (виконуються справа наліво).

Деякі операції, наприклад, операції множення і ділення, мають один і той же пріоритет. Яким тоді буде результат обчислення виразу:

*int x = 10/5*2;*

Як трактувати цей вираз як (10/5)*2 чи як 10/(5*2)? Адже в залежності від трактування отримаються різні результати.

Всі арифметичні оператори (крім префіксного інкремента і декремента) є лівоасоціативні, тобто виконуються зліва направо. Тому вираз 10/5*2 необхідно трактувати як (10/5)*2, тобто результатом буде число 4.

Порозрядні операції виконуються над окремими розрядами або бітами чисел. У даних операціях в якості операндів можуть виступати тільки цілі числа. Кожне число має певне двійкове подання. Наприклад, число 4 в двійковій системі 100, а число 5 - 101 і так далі.

Тип *byte* займає 1 байт або 8 біт, відповідно представлений 8 розрядами. Тому значення 7 в двійковому коді дорівнюватиме 00000111. Тип *short* займає в пам'яті 2 байти або 16 біт, тому число даного типу буде представлено 16 розрядами. І в цьому випадку 7 в двійковій системі буде мати вигляд 0000 0000 0000 0111.

У випадку логічних порозрядних операцій числа розглядаються у двійковому поданні, наприклад, 2 в двійковій системі дорівнює 10 і має два розряди, число 7 – 111 і має три розряди.

Логічне порозрядне множення (&) виконується порозрядно, і якщо у обох операндів значення розрядів дорівнює 1, то операція повертає 1, інакше повертається число 0.

Логічне порозрядне додавання (|) проводиться теж по двійковим розрядам, але в цьому випадку повертається одиниця, якщо хоча б у одного числа в даному розряді є одиниця (операція "логічне АБО").

Логічне порозрядне виключне АБО (^) повертає 1 у випадку коли розряди чисел різні і 0 в протилежному випадку.

Логічне порозрядне заперечення (~) інвертує всі розряди числа, тобто, якщо значення розряду дорівнює 1, то воно стає рівним нулю, і навпаки.

Крім цього, над розрядами чисел виконуються операції зсуву. Зсув може бути виконаний вправо і вліво:

- *a*<<*b* – зсуває число *a* вліво на *b* розрядів;
- *a*>>*b* – зсуває число *a* вправо на *b* розрядів. Наприклад, 16>>1 зсуває число 16 (яке в двійковій системі 10000) на один розряд вправо, тобто в результаті виходить 1000 або число 8 в десятковому поданні;
- *a*>>>*b* – на відміну від попередніх типів зсувів дана операція являє беззнаковий зсув. Наприклад, вираз -8 >>> 2 дорівнюватиме 1073741822.

Таким чином, якщо вихідне число, яке треба зсунути в ту або іншу сторону, ділиться на два, то фактично виходить множення або ділення на два. Тому подібну операцію можна використовувати замість безпосереднього множення або ділення на два, оскільки операція зсуву на апаратному рівні менш «вартісна» операція на відміну від операції ділення або множення.

Умовні вирази представляють деяку умову, яка повертає значення типу *boolean*, тобто значення *true* (якщо умова істинна), або значення *false* (якщо умова хибна). До умовних виразів відносяться операції порівняння та логічні операції. В операціях порівняння порівнюються два операнди, і повертається значення типу *boolean – true*, якщо вираз вірний, і *false*, якщо вираз невірний. Такими операціями є:

- == – перевірка рівності, яка порівнює два операнди на рівність і повертає *true* (якщо операнди рівні) і *false* (якщо операнди не рівні);
- != – перевірка нерівності, яка порівнює два операнди і повертає *true*, якщо операнди НЕ рівні, і *false*, якщо операнди рівні;
- < – «менше ніж», яка повертає *true*, якщо перший операнд менше другого, інакше повертає *false*;
- > – «більше ніж», яка повертає *true*, якщо перший операнд більше другого, інакше повертає *false*;
- >= – «більше або дорівнює», яка повертає *true*, якщо перший операнд більше другого або дорівнює

другому, інакше повертає *false*;

- <= – «менше або дорівнює», яка повертає *true*, якщо перший операнд менше другого або дорівнює другому, інакше повертає *false*.

До логічних операцій відносять наступні:

- |, ||, при яких *c* дорівнює *true*, якщо або *a*, або *b*, або обидва дорівнюють true у виразі c = a | b, інакше – *false*;
- &, &&, при яких *c* дорівнює *true*, якщо і *a*, і *b* рівні *true* у виразі *c = a&b*, інакше – *false*;
- !, при якій *c* дорівнює *true*, якщо *b* дорівнює *false* у виразі *c = !b*, інакше – *false*;
- ^, при якій *c* дорівнює *true*, якщо або *a*, або *b* (але не одночасно) дорівнюють *true* у виразі *c=a^b*, інакше – *false*.

Дві пари операцій |, || та &, && виконують схожі дії, проте вони не рівнозначні. Вираз *c = a | b* буде обчислювати спочатку обидва значення – *a* і *b* і на їх основі обчислювати результат. У виразі ж *c = a || b* спочатку буде обчислюватися значення *a*, і якщо воно дорівнює *true*, то обчислення значення *b* вже сенсу немає, тому що в будь-якому випадку вже *c* буде рівне *true*. Значення *b* буде обчислюватися тільки в тому випадку, якщо а дорівнює *false*. Те ж саме стосується пари операцій &, &&. Таким чином, операції || і && більш зручні в обчисленнях, дозволяючи скоротити час на обчислення значення виразу і тим самим підвищуючи продуктивність. А операції | і & більше підходять для виконання порозрядних операцій над числами. Приклади:

- *boolean a = (3>5) || (4<5); // 3>5 – false, 4<5 – true, тому повертається true;*
- *boolean b = (3>5) && (4<5); // 3> 5 – false, тому повертається false (4<5 – true, але не обчислюється);*
- *boolean c = (3>5) ^ (4<5); // 3>5 – true, тому повертається true (4<5 – false).*

Операціями присвоєння на мові Java є такі операції: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=. Всі вони, за винятком першої операції (= –звичайного присвоєння), є поєднанням операції присвоєння та деякої іншої операції. Наприклад, результатом операції += у виразі *c += b* буде запис у змінну с суми с та b.

При роботі з операціями важливо розуміти їх **пріоритет**, який можна описати наступною послідовністю **в порядку його зменшення**:

- expr++, expr--;
- ++expr, --expr, +expr -expr, ~, !;
- *, /, %;
- +, -;
- <<, >>, >>>;
- <, >, <=, >=, instanceof;
- ==, !=;
- &;
- ^;
- |;
- &&;
- ||;
- ? : (тернарний оператор);
- =, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>= (оператори присвоювання).

Не охарактеризованими операціями, як видно з останньої послідовності, залишились **унарний плюс та унарний мінус**. Остання дозволяє змінити знак на протилежний. Що ж стосується унарного плюса, то ця операція дозволяє виконати явне висхідне перетворення типів. Наприклад, для рядків коду:

```
char k='a';
System.out.println(k);
System.out.println(+k);
```

результатом є вивід на екран символу ‘a’ та 97, оскільки 97 – це ASCII-код символу ‘a’.

2.3. Умовні конструкції, цикли, масиви.

Одним з фундаментальних елементів багатьох мов програмування є умовні конструкції. Дані конструкції дозволяють спрямувати роботу програми по одному із напрямів в залежності від певних умов. У мові Java використовуються умовні конструкції: *if ... else* та *switch ... case*.

Конструкція *if … else* перевіряє істинність деякої умови і залежно від результатів перевірки виконує певний код:

```
int n1 = 6; int n2 = 4;
if (n1>n2)
   System.out.println("Перше число є більшим другого");
```

Після ключового слова *if* наводиться умова; якщо ця умова виконується, то спрацьовує код, що розміщений в блоці *if*, який розташовуватиметься в фігурних дужках у випадку декількох операторів. В якості умови виступає операція порівняння двох чисел.

Оскільки, в даному випадку перше число більше другого, то вираз *n1>n2* істинний і повертає значення true. Отже, управління переходить в блок коду після круглих дужок і виконується оператор *System.out.println("Перше число є більшим другого")*. Якби перше число виявилося меншим за друге або рівне йому, то оператори в блоці if не виконувалися б. До умовної конструкції в попередньому коді можна додати блок *else*, тим самим доповнивши програму операторами у випадку хибності умови.

Але при порівнянні чисел можна нарахувати три стани: перше число більше другого, перше число менше другого і числа рівні. За допомогою виразу *else if* можна обробляти декілька умов:

```
int n1 = 6; int n2 = 8;
if (n1 > n2)
   System.out.println ("Перше число більше другого");
else if (n1 < n2) {
   System.out.println( "Перше число менше другого");
else
   System.out.println ( "Числа рівні");
```

Крім цього, можна поєднати одночасно декілька умов, використовуючи логічні оператори:

```
if ((n1 > n2 && n1 > 7) || n2 > 0)
```

В цьому випадку, блок if буде виконуватися, якщо *n1>n2* рівне *true* та одночасно *n1>7* рівне *true*, або ж у випадку коли *n2>0*.

Конструкція *switch … case* схожа до конструкції if … else, оскільки дозволяє обробити одночасно декілька умов:

```
int n = 8;
switch(n) {
   case 1:
      System.out.println("число дорівнює 1");
      break;
   case 8:
      System.out.println("число дорівнює 8");
      n++;
      break;
   case 9:
      System.out.println("число дорівнює 9");
      break;
   default:
      System.out.println("число не дорівнює 1, 8, 9");
}
```

Після ключового слова *switch* в дужках йде вираз для порівнняня. Значення цього виразу послідовно порівнюється зі значеннями, розміщеними після операторів *case*, і якщо збіг знайдено, то буде виконуватись відповідний блок *case*.

В кінці блоку *case* ставиться оператор *break*, щоб уникнути виконання інших блоків. Наприклад, якщо прибрати оператор *break*, то в наступному випадку:

```
case 8:
   System.out.println("число дорівнює 8");
   n++;
case 9:
   System.out.println("число дорівнює 9");
   break;
```

виконається блок *case 8*, (оскільки змінна num дорівнює 8). Але оскільки в цьому блоці оператор *break* відсутній, то виконається і блок *case 9*.

Для обробки ситуації, коли збіги не буде знайдено можна додати блок *default*, як в прикладі вище. Хоча блок *default* необов'язковий. Також можна визначити одну дію одночасно для декількох блоків *case* поспіль:

```
int num = 3; int output = 0;
switch(num) {
   case 2:
   case 3:
   case 4:
      output = 6;
      break;
   default:
      output = 24;
}
```

До умовних конструкцій відноситься **тернарна операція**, яка має наступний синтаксис: [перший операнд – умова] ? [другий операнд]: [третій операнд]. Таким чином, в цій операції беруть участь одночасно

три операнди. Залежно від умови тернарна операція повертає другий або третій операнд: якщо умова рівна *true*, то повертається другий операнд; якщо умова рівна *false*, то третій. наприклад:

```
int x = 3; int y = 2;
int z = x<y ? (x+y) : (x-y);
```

Результатом тернарної операції є змінна *z*. Спочатку перевіряється умова *x<y*, якщо вона *true*, то *z* дорівнюватиме другому операнду *(x+y)*, інакше *z* дорівнюватиме третьому операнду *(x-y)*.

Ще одним типом керуючих конструкцій є цикли. **Цикли** дозволяють в залежності від певних умов виконувати певну дію декілька разів. У мові Java є такі види циклів:

- *for*;
- *while*;
- *do ... while*.

Цикл *for* має наступне формальне визначення:

```
for([ініціалізація лічильника]; [умова]; [зміна лічильника])
{
   // дії
}
```

Розглянемо стандартний цикл for:

```
for(int i = 1; i < 9; i++) {
   System.out.printf ( "Квадрат числа %d дорівнює %d \n", i, i * i);
}
```

Перша частина – оголошення циклу (*int i=1*) створює і ініціалізує лічильник *i*. Лічильник необов'язково повинен бути типу *int*. Це може бути і будь-який інший числовий тип, наприклад, *float*. Перед виконанням циклу значення лічильника буде дорівнювати 1. У даному випадку це те ж саме, що і оголошення змінної. Друга частина – умова, при якій буде виконуватися цикл. У даному випадку цикл буде виконуватися, поки не досягне 9. І третя частина – приріст лічильника, в нашому випадку це приріст на одиницю. Не обов'язково збільшувати, можна і зменшувати: *i--*. В результаті блок циклу спрацює 8 разів, поки значення і не стане рівним 9. І кожен раз це значення буде збільшуватися на 1. Необов'язково вказувати всі умови при оголошенні циклу. Наприклад, можна написати так:

```
int i=1;
for (;;)
   System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
```

Немає ініціалізованої змінної-лічильника та немає умови, тому цикл буде працювати вічно – нескінченний цикл. Або можна опустити ряд блоків:

```
int i = 1;
for(; i <9;) {
   System.out.printf("Квадрат числа %d дорівнює %d \n", i, i * i);
   i++;
}
```

Цей приклад еквівалентний першому прикладу: у нас також є лічильник, тільки створений він поза циклом, є умова виконання циклу, і є приріст лічильника вже в самому блоці *for*. Цикл *for* може визначати одночасно кілька змінних і управляти ними:

```
int n = 10;
for (int i = 0, j = n - 1; i < j; i++, j--) {
   System.out.println (i * j);
}
```

Цикл *do … while* спочатку виконує код циклу, а потім перевіряє умову в інструкції *while*. І поки ця умова істинна, цикл повторюється. Наприклад:

```
int j = 7;
do {
   System.out.println(j);
   j--;
}
while(j> 0);
```

В даному випадку код циклу спрацює 7 разів, поки *j* не дорівнюватиме нулю. Цикл *do ... while* гарантує хоча б одноразове виконання дій, навіть якщо умова циклу хибна.

Цикл *while* одразу перевіряє істинність деякої умови, і якщо умова істинна, то код циклу виконується:

```
int j = 6;
while(j> 0) {
   System.out.println(j);
   j--;
}
```

Оператор *break* дозволяє вийти з циклу в будь-який його момент, навіть якщо цикл не закінчив

свою роботу:

```
for (int i = 0; i < 10; i++) {
   if (i == 5)
      break;
   System.out.println(i);
}
```

Коли лічильник стане рівним 5, спрацює оператор *break*, і цикл завершиться. Тепер зробимо так, щоб якщо число дорівнює 5, цикл не завершувався, а просто переходив до наступної ітерації. Для цього використовуємо оператор continue:

```
for (int i = 0; i < 10; i++) {
   if (i == 5)
      continue;
   System.out.println(i);
}
```

У цьому випадку, коли виконання циклу дійде до числа 5, програма просто пропустить це число і перейде до наступного.

Масив являє набір однотипних значень. Оголошення масиву схоже на оголошення звичайної змінної, яка зберігає одне значення, причому є два способи оголошення масиву:

- тип_даних назва_масиву[];
- тип_даних[] назва_масиву;

Наприклад, визначимо масив чисел:

```
int nums[];
int[] nums2;
```

Після оголошення масиву можна його ініціалізувати:

```
int nums[];
nums = new int [4]; // масив з 4 чисел
```

Створення масиву виконується за допомогою наступної конструкції: *new тип_даних [кількість_елементів]*, де *new* – ключове слово, що виділяє пам'ять для зазначеної в дужках кількості елементів. Наприклад, у виразі *nums = new int [4]* створюється масив з чотирьох елементів *int*, і кожен елемент буде мати значення за замовчуванням – число 0.

Також можна одразу при оголошенні масиву форматувати його:

```
int[] nums = new int [5]; // масив з 5 чисел
```

При такій ініціалізації всі елементи масиву мають значення за замовчуванням. Для числових типів (в тому числі для типу *char*) це число 0, для типу *boolean* це значення *false*, а для інших об'єктів це значення *null*. Наприклад, для типу int значенням за замовчуванням є число 0, тому масив *nums* буде складатися з 5 нулів. Крім цього, можна задати конкретні значення для елементів масиву при його створенні двома рівноцінними способами:

```
int[] nums = new int [] {1, 2, 3, 5};
int[] nums2 = {1, 2, 3, 5};
```

Варто зазначити, що в цьому випадку в квадратних дужках не вказується розмір масиву, тому що він обчислюється за кількістю елементів в фігурних дужках. Після створення масиву можна звернутися до будь-якого його елементу за індексом, який передається в квадратних дужках після назви змінної масиву:

```
int[] nums = new int [4]; // ініціалізуємо масив
nums[0] = 1; nums [1] = 2; nums [2] = 4; nums [3] = 100;
// отримуємо значення третього елемента масиву
System.out.println(nums [2]); // 4
```

Індексація елементів масиву починається з 0, тому в даному випадку, щоб звернутися до четвертого елементу в масиві потрібно використовувати вираз *nums[3]*.

Оскільки масив визначений тільки для 4 елементів, то не можна звернутися, наприклад, до шостого елементу: *nums[5] = 5;*. При спробі це зробити буде відповідна помилка.

Найважливіша властивість, якою володіють масиви, є властивість *length*. Вона повертає довжину масиву, тобто кількість його елементів:

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

Можлива ситуація, коли невідомо індекс останнього елемента, і щоб отримати останній елемент масиву, ми можемо використовувати цю властивість.

Крім одновимірних масивів також бувають і багатовимірні. Найбільш відомий **багатовимірний масив** – це двовимірний:

```
int[][] nums = {{0, 1, 2}, {3, 4, 5}};
```

Оскільки масив *nums* двовимірний, він являє собою просту матрицю. Його також можна було створити наступним чином: *int [] [] nums2 = new int [2] [3];*. Кількість квадратних дужок вказує на розмірність масиву. А числа в дужках – на кількість рядків і стовпців. І також, використовуючи індекси, можна використовувати елементи масиву в програмі:

```
// встановимо елемент першого стовпця другого рядка
nums[1][0] = 44;
System.out.println(nums [1] [0]);
```

Оголошення тривимірного масиву могло б виглядати так:

```
int[][][] nums3 = new int [2] [3] [4];
```

Багатовимірні масиви можуть бути також представлені як "зубчасті масиви". В одному з прикладів вище, двовимірний масив мав два рядки та три стовпці, тому отримувалась прямокутна матриця. Але можна кожному елементу в двомірному масиві присвоїти окремий масив з різною кількістю елементів:

```
int[][] nums = new int [3] [];
nums[0] = new int [2];
nums[1] = new int [3];
nums[2] = new int [5];
```

Існує також спеціальна версія циклу for, яка призначена для перебору елементів в наборах елементів, наприклад, в масивах і колекціях, аналогічна дії **циклу *for each***, який є в інших мовах програмування. Формальне його оголошення:

```
for (тип_даних назва_змінної: контейнер) {
   // дії
}
```

Наприклад:

```
int[] array = new int [] {1, 2, 3, 4, 5};
for (int i: array) {
   System.out.println(i);
}
```

В якості контейнера в даному випадку виступає масив даних типу *int*. Потім оголошується змінна з типом *int*. Те ж саме можна було б зробити і за допомогою звичайної версії *for*:

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

У той же час, звичайна версія циклу *for* більш гнучка у порівнянні зі спеціальною. Зокрема, у звичайній версії можна змінювати елементи.

За допомогою, циклів можна здійснювати перебір елементів багатовимірних масивів:

```
int[] [] nums = new int [] []{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
for (int i = 0; i < nums.length; i++) {
   for (int j = 0; j < nums [i] .length; j++) {
      System.out.printf ( "%d", nums[i][j]);
   }
   System.out.println();
}
```

Спочатку створюється цикл для перебору по рядках, а потім всередині першого циклу створюється внутрішній цикл для перебору по стовпцях конкретного рядка. Подібним чином можна перебрати і тривимірні масиви і набори з великою кількістю розмірностей.

2.4. Методи, параметри та перевантаження методів, повернення з методів, рекурсивні функції.

Методи являють собою набір операторів, які виконують певні дії. **Загальне визначення методів** виглядає наступним чином:

```
[Модифікатори] тип_знач_рез назва_методу ([параметри]) {
   // тіло методу
}
```

Модифікатори і параметри необов'язкові. По замовчуванню головний клас будь-якої програми на Java містить метод *main*, який є точкою входу в програму. Створимо декілька методів у програмі:

```
public class Program {
   public static void main(String args[]) {
      method_hello();
      method_welcome();
      method_welcome();
   }
   static void method_hello() { System.out.println("Привіт");  }
   static void method_welcome() { System.out.println("Ласкаво просимо
   в Java"); }
}
```

Тут визначені два додаткові методи: *method_hello* і *method_welcome*, кожен з яких виводить деяке повідомлення на консоль. Методи визначаються всередині класу – в даному випадку всередині класу

Program, в якому визначено метод main. Крім цього, у методі *main* викликається один раз метод *method_hello* і два рази метод *method_welcome*. В цьому і полягає одна з переваг методів, тобто в тому, що можна винести деякі загальні дії в окремий метод і потім викликати багаторазово їх в різних місцях програми. Оскільки обидва методи не мають ніяких параметрів, то після їх назви при виклику ставляться порожні дужки. Також слід зазначити, що для виклику в певному методі інших методів цього ж класу, вони повинні мати модифікатор static.

За допомогою параметрів можна передати в методи різні дані, які будуть використовуватися для обчислень. При виклиці таких методів в програмі необхідно передати в якості параметрів значення, які відповідають типу параметра:

```
public class Program {
   public static void main(String args[]) {
      int x= 5;
      int y= 4;
      method_sum(x,y); // 9
      method_sum(6,x); // 11
      method_ sum(8,2); // 10
   }
   static void method_sum(int a, int b) {
      int c=a+b;
      System.out.println(c);
   }
}
```

Оскільки метод *method_sum* приймає два значення типу *int*, то на місце параметрів потрібно передати два значення типу *int*. Це можуть бути і числові літерали, і змінні типів даних, які представляють тип *int* або можуть бути автоматично перетворені в тип *int*. Значення, які передаються в якості параметрів, ще називаються аргументами. Значення передаються параметрам по позиції, тобто перший аргумент першому параметру, другий аргумент – другому параметру і так далі.

Метод може приймати параметри змінної довжини одного типу. Наприклад, потрібно передати в метод невідому завчасно кількість чисел та обчислити їх суму. Параметри змінної довжини дозволяють вирішити цю задачу:

```
public class Program {
   public static void main(String args[]) {
      sum_numbers(3, 4, 5); // 12
      sum_numbers(5, 4, 2, 1, 5); // 17
      sum_numbers(); // 0
   }
   static void sum_numbers(int ... numbers) {
      int res = 0;
      for(int n: numbers)
         res += n;
      System.out.println(res);
   }
}
```

Три крапки перед назвою параметра *int ... numbers* вказує на те, що він буде необов'язковим і представлятиме масив. Можна передати в метод *sum_numbers* одне число, декілька чисел, а можна взагалі не передавати ніяких параметрів. Причому, якщо потрібно передати декілька параметрів, то необов'язковий параметр повинен вказуватися в кінці:

```
public static void main(String[] args) {
   another_sum("Ласкаво просимо!", 31, 7);
   another_sum("Привіт світ!");
}
static void another_sum(String mes, int ... numbers) {
   System.out.println(mes);
   int res = 0;
   for (int x: numbers)
      res+= x;
   System.out.println (res);
}
```

Методи можуть повертати деяке значення. Для цього застосовується оператор *return: return тип_значення_результату*. Після оператора *return* вказується значення, що повертається, тобто те, яке є результатом методу. Це може бути літеральне значення, значення змінної або якогось складного виразу. Розглянемо приклад:

```
public class Program {
```

```
    public static void main(String args[]) {
        int a = sum(1, 1, 1);
        int b = sum(5, 4, 3);
        System.out.println(a); // 3
        System.out.println(b); // 12
    }
    static int sum (int x, int y, int z) {  return x + y + z; }
}
```

У методі в якості типу значення, що повертається замість *void* можна використовувати будь-який інший тип. В даному випадку метод *sum* повертає значення типу *int*, тому цей тип вказується перед назвою методу. Причому, якщо в якості типу результату визначено будь-який інший, відмінний від *void*, то метод обов'язково повинен використовувати оператор *return* для повернення значення. При цьому значення, що повертається завжди повинно мати той же тип, що значиться у визначенні функції. Метод може використовувати кілька викликів оператора *return* для повернення різних значень в залежності від деяких умов.

Оператор *return* застосовується не тільки для повернення значення з методу, але і для виходу з методу. У подібній якості оператор *return* застосовується в методах, які нічого не повертають, тобто мають тип *void.*

У програмі можна використовувати методи з одним і тим же ім'ям, але з різними типами і/або кількістю параметрів. Такий механізм називається **перевантаженням методів**. Наприклад:

```
public class Program {
    public static void main (String[] args) {
        System.out.println(sum (3, 4)); // 7
        System.out.println(sum (5.5, 5.2)); // 10.7
        System.out.println(sum (2, 5, 3)); // 10
    }
    static int sum(int a, int b, int c) { return a + b + c;  }
    static double sum(double a, double b) { return a + b;  }
    static int sum(int a, int b) { return a + b; }
}
```

Тут визначено три варіанти або три перевантаження методу *sum()*. При його виклику система вибере саме ту версію методу, яка підходить за типом та кількостю переданих параметрів. Варто зазначити, що на перевантаження методів впливає тільки кількість і тип параметрів. Однак відмінність в типі значення, що повертається для перевантаження не має ніякого значення. При цьому програма, в якій наявні методи відрізняються тільки типом результату є некоректною і не буде скомпільована.

Розглянемо **рекурсивні функції**. Головна відмінність рекурсивних функцій від звичайних методів полягає в тому, що рекурсивна функція може викликати саму себе. Наприклад, розглянемо функцію, яка обчислює факторіал числа:

```
static int factorial(int a) {
    if (a == 1) { return 1;  }
    return  a * factorial(a-1);
}
```

Спочатку перевіряється умова рівності одиниці; якщо змінна *x* не дорівнює 1, то вона множиться на результат цієї ж функції, в яку як параметр передається число *a-1*, тобто відбувається рекурсивний спуск. І так далі, поки не дійдемо до того моменту, коли значення параметру буде дорівнювати одиниці.

Рекурсивна функція обов'язково повинна мати певний базовий варіант, який використовує оператор *return* і який розміщається, зазвичай, на початку функції. У випадку з факторіалом це *if (a==1) return 1*. І всі рекурсивні виклики повинні звертатися до підфункцій, які в кінцевому рахунку сходяться до базового варіанту. Так, при передачі в функцію додатного числа при подальших рекурсивних викликах підфункцій в них буде передаватися кожен раз число, менше на одиницю. І врешті-решт ми дійдемо до ситуації, коли число буде дорівнювати 1, і буде використаний базовий варіант.

2.5. Обробка виняткових ситуацій.

В процесі виконання програми можуть виникати помилки, причому необов'язково з вини розробника. Деякі з них важко передбачити, а іноді і зовсім неможливо. Так, наприклад, може несподівано обірватися з'єднання з мережею при передачі файлу. Подібні ситуації називаються винятковими.

У мові Java передбачені спеціальні засоби для обробки таких ситуацій. Одним з таких засобів є **конструкція *try ... catch ... finally***. При виникненні виключення в блоці *try* управління переходить в блок *catch*, який повинен обробити виняткову ситуацію. Якщо такого блоку не знайдено, то користувачу відображається повідомлення про необроблене виключення, а подальше виконання програми зупиняється. І щоб такої зупинки не відбулося потрібно використовувати блок *try..catch*. Наприклад:

```
int[] nums = new int [3];
```

```
nums[4] = 45;
System.out.println(nums[4]);
```

Оскільки масив *nums* може містити тільки 3 елементи, то при виконанні інструкції *nums[4]=45* консоль відобразить виключну ситуацію, і виконання програми буде аварійно завершено. Тепер спробуємо обробити цей виняток:

```
try {
   int[] nums = new int [3];
   nums[4] = 45;
   System.out.println(nums[4]);
}
catch (Exception ex) {
   ex.printStackTrace ();
}
System.out.println("Програмний додаток завершений");
```

При використанні блоку *try ... catch* спочатку виконуються всі оператори в блоці *try*. Якщо в блоці *try* виникає виключення, то звичайний порядок виконання зупиняється і переходить до інструкції *catch*. Тому коли виконання програми дійде до рядка *nums[4] = 45* програма перейде до блоку *catch*.

Вираз *catch* має наступний синтаксис:

catch(тип_винятковоїситуації ім'я_змінної)

У прикладі вище, оголошується змінна *ex*, яка має тип *Exception*. Якщо виникло виключення, яке не є винятком типу, зазначеного в операторі *catch*, то воно не обробляється, а програма просто зависає або генерує повідомлення про помилку.

Тип *Exception* є базовим класом для всіх винятків, тому вираз *catch(Exception ex)* буде обробляти всі виключення. Обробка виключення, в даному випадку, зводиться до виведення на консоль стеку трасування помилки за допомогою методу *printStackTrace()*, визначеного в класі *Exception*.

Після завершення виконання блоку *catch* програма продовжує свою роботу, виконуючи інші оператори після блоку catch.

Конструкція *try..catch* також може мати блок *finally*. Однак цей блок необов'язковий, і його можна при обробці виключень опускати. Блок *finally* виконується в будь-якому випадку, чи виникло виключення в блоці *try* чи ні:

```
try {
   int[] nums = new int [3];
   nums[4] = 45;
   System.out.println(nums[4]);
}
catch(Exception ex) {
   ex.printStackTrace();
}
finally {
   System.out.println("Необов'язковий блок finally");
}
System.out.println("Програмний додаток завершений");
```

У Java є багато різних типів виключень, і можна розмежувати їх обробку, включивши додаткові блоки *catch*:

```
int[] nums = new int [3];
try {
   nums[6] = 45;
   nums[6] = Integer.parseInt("kkk");
}
catch (ArrayIndexOutOfBoundsException ex) {
   System.out.println("Вихід за межі масиву");
}
catch (NumberFormatException ex) {
   System.out.println("Помилка перетворення з рядка в число");
}
```

Якщо виникає виключення певного типу, то програма переходить до відповідного блоку *catch*.

Можливі ситуації коли потрібно згенерувати виняткові ситуації в програмі, у таких випадках використовують **оператор *throw***. За допомогою цього оператора можна створити виключення "вручну", тобто викликати його в процесі виконання. Наприклад, в програмі відбувається введення числа, при цьому потрібно, щоб у випадку, коли число більше 30, виникало виключення:

```
package firstapp;
import java.util.Scanner;
public class FirstClass {
```

```
    public static void main (String[] args) {
        try {
            Scanner in = new Scanner(System.in);
            int n = in.nextInt();
            if (n >= 30) {
                throw new Exception("Число n має бути менше 30");
            }
        }
        catch (Exception ex) {
            System.out.println (ex.getMessage());
        }
        System.out.println("Програмний додаток завершений");
    }
}
```

Тут для створення об'єкта виключення використовується конструктор класу *Exception*, в який передається повідомлення про виключення. І якщо число *x* виявиться більше 29, то буде згенеровано виключення і управління перейде до блоку *catch*. Блоці *catch* виведе повідомлення про виключення за допомогою методу *getMessage()*.