**Лабораторна робота №1**

**Базові оператори. Програмування циклічних алгоритмів**

**Мета.** Засвоїти роботу базових операторів мови Java, навчитися реалізовувати циклічні оператори.

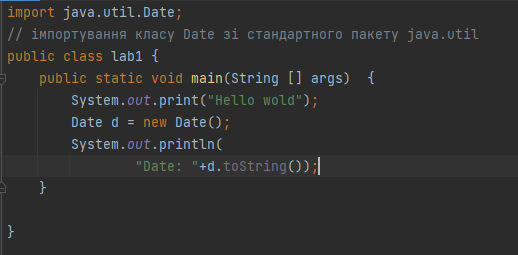
**Теоретичні відомості**

**Огляд структури Java-програми**

Всі Java-програми містять в собі 4 основні різновиди будівельних блоків: класи (classes), методи (methods), змінні (variables) і пакети (packages). Java-програма може містити в собі будь-яку кількість класів, але один з них завжди має особливий статус, і безпосередньо взаємодіє з оболонкою часу виконання. Цей клас називають первинним класом (primary class).

Коли програма запускається з командного рядка, системі потрібен тільки один спеціальний метод, що повинен бути присутнім у первинному класі, – метод main.

Стандартна програма має вигляд:



Наведена програма виводить на екран повідомлення "Hello, Java!" та поточну системну дату.

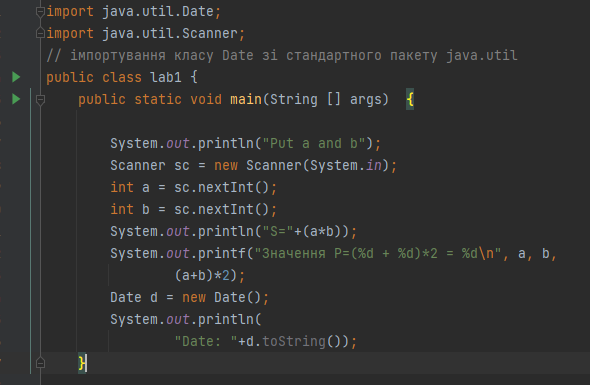
**Виведення даних у консолі Java-програм**

Для виведення інформації на консоль використовуються методи стандартного класу PrintStream: – print – println – printf – format (точна копія printf)

Кожна програма на мові Java містить стандартний об'єкт типу PrintStream – System.out. Таким чином, виведення інформації на екран буде записуватися як

System.out.print(…), System.out.println(…), або System.out.printf(…).

Методи print та println повинні завжди мати один параметр – вираз будь-якого типу, що може бути автоматично приведений до рядкового типу.



**Базові оператори**

Набір операторів мови Java містить:

• оператори опису змінних та інших об’єктів;

• оператори-вирази;

• оператори привласнення;

• умовний оператор if;

• три оператори циклу while, do-while, for;

• оператор варіанту switch;

• Оператори переходу break, continue та return;

• блок {};

• пустий оператор; - просто крапка з комою.

Тут наведений не весь набір операторів Java, він буде доповнюватися по мірі вивчення мови.

В мові Java немає оператора goto.

Будь-який оператор закінчується крапкою з комою.

Можна поставити крапку з комою в кінці будь-якого виразу, і він стане оператором. Але сенс це має тільки для операцій привласнення, інкременту та декременту, викликів методів. Крапка з комою в Java не розділяє оператори, а є частиною оператора.

Лінійне виконання алгоритму забезпечується послідновним записом операторів. Перехід від строки до строки в вихідному тексту не має жодного значення для компілятора, він здійснюється тілька для наочності та читабельності тексту.

**Блок**

Блок вкладає в себе декілька операторів з метою використати їх як один оператор в тих місцях, де за правилами мови можна записати тільки один оператор. Наприклад, {х = 5; у = 7;}. Можна записати і пустий блок, просто через декілька фігурних дужок {}.

Блоки операторів часто використовуються для обмеження області дії змінних та просто для покращення читабельності тексту програми.

**Основні арифметичні оператори**

В Java визначені **арифметичні бінарні операції**, котрі використовуються в математичних виразах таким же чином, як вони використовуються в алгебрі (Таблиця 1.). Припустимо, ціла змінна A дорівнює 10, а змінна B рівна 20.

Табл. 1. Арифметичні бінарні операції

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Опис** | **Приклад** |
| + | Складає значення по обидві сторони від оператора | A + B дасть 30 |
| - | Віднімає правий операнд від лівого операнда | A - B дасть -10 |
| \* | Перемножує значення по обидві сторони від оператора | A \* B дасть 200 |
| / | Оператор ділення ділить лівий операнд на правий операнд | A / B дасть 2 |
| % | Ділить лівий операнд на правий операнд та повертає залишок | A % B дасть 0 |

Складові арифметичні оператори з привласненням об’єднуючі арифметичні оператори з операціями присвоєння.

**int a = 5;**

**int sum += a; // еквівалентно sum = sum + a;**

**int sub -= a; // еквівалентно sub = sub - a;**

**int mult \*= a; // еквівалентно mult = mult \* a;**

**int div /= a; // еквівалентно div = div / a;**

**int mod %= a; // еквівалентно mod = mod % a;**

Складові оператори з привласненням існують для всіх арифметичних операторів з двома операндами.

**Унарні операції**

В Java визначені наступні унарні операції:

* Унарний мінус "-" – змінює знак числа чи виразу на протилежний;
* унарний плюс "+" – не виконє ніяких дій над числом чи виразом;
* побітове доповнення "~" (тільки для цілих) – інвертє всі біти поля числа (замінює 0 на 1 та 1 на 0);
* інкремент "+ +» – збільшує значення змінної на 1;
* декремент "- -" – зменшує значення змінної на 1.

Приклад 1. Унарних операцій "+" та "-":

byte i = 3, j, k;

j= -i; // j = -3

k = +i; // k = 3

Приклад 2. Операції побітового доповнення:

byte a = 39;

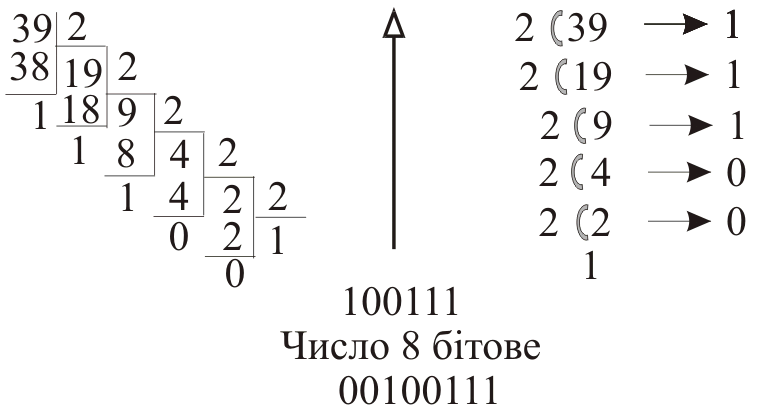
byte b;

b = ~a;

Числа a та b є числами типу **byte**, тобто представляються всередину комп’ютера як двійкові цілі числа зі знаком.

Ціла частина числа перетворюється із десяткової системи числення в іншу – послідовним діленням цілої частини на основу системи числення (для двійкової системи – на 2, для 8-ї – на 8, для 16-ї – на 16 і тд.) до отримання цілого залишку, меншого, ніж основа системи числення.

Наприклад, число 8-біт 39 представимо у двійковій системі числення.



Числа a та b є числами типу **byte**, тобто представляються всередину комп’ютера як двійкові цілі числа зі знаком довжиною 8 біт, тому двійкове представлення чисел a та b буде виглядати наступним чином:

a = 00100111

b = 11011000

Як видно з того представлення, всі нульові біти в числі a замінені на одиничні біти в числі b, а одиничні біти в a замінені на нульові біти. Десятковим представленням числа b буде –88.

Додаткове пояснення

Всі цілочисельні типи, за виключенням *char* – типи зі знаком, тобто можуть бути додатковими чи від’ємними. В Java застосовується двійкове доповнення, за яким від’ємні числа представляються в результаті інвертування всіх бітів значення (заміни 1 на 0 і навпаки) та подальшого додавання 1 до результату.

Наприклад, щоб представити число (-39) необхідно:

1. Представити число 39 у двійковій системі 00100111 в одному байті.
2. Виконати інвертацію всіх бітів **1**1011000, додати 1, що призведе до значення **1**1011001 та буде відповідати значенню -39.

Щоб декодувати від’ємне число, необхідно

Спочатку інвертувати всі біти 00100110, що приводить нас до числа 25 + 22 + 21 = 38.

1. Потім додати 1 до результату, отримаємо 00100111 або 39.

Двійкове доповнення використовується у більшості комп’ютерних мов. Опустимо теорію, потрібно тільки пам’ятати, що старший біт визнчає знак цілого числа.

& - побітовий оператор “І”

Він досить схожий по написанню на логічний “І” (&).

Для прикладу, візьмемо числа 277 та 432: 100010101 - число 277 в двійковій формі 110110000 - число 432 в двійковій формі. Далі оператор & порівнює перший біт верхнього числа з першим бітом нижнього. Оскільки це оператор «І», то результат дорівнюватиме 1. В усіх інших випадках результатом буде 0.

**100010101** & **110110000** = **100010000 - результат роботи** &

В цьому виразі спочатку порівнюємо перші біти двох чисел один з одним, потім другі біти, треті і тд. Тільки в двох випадках обидва біти в числах були рівні 1 (перший та п’ятий за рахунком біти). Результатом усіх інших порівнянь став 0. Том як резльтат вийшло число 100010000. У десятковій систему йому відповідає число 272.

Приклад 3. Перевірка результату в Java:

public class Main {

public static void main(String[ ] args) {

System.out.println(277&432);

}

}

***Вивід в консоль*: 272**

|**- побітове “АБО”.**

Принцип роботи той самий – порівнюються два числа по бітам. Тільки тепер, якщо **хоча б один** з бітів дорівнює 1, результат буде рівним 1. Розглянемо на тих самих числах – 277 та 432:

**100010101** | **110110000** = **110110101 - результат роботи** |.

Тут результат вже інший: нулями залишилися тільки ті біти, котрі в обидвох числах були нулями. Результат робочи число 110110101. В десятковій системі йому відповідає число 437. Перевіримо:

Пример 4.

public class Main {

public static void main(String[] args) {

System.out.println(277|432);

}

}

**Вивід в консоль: 437**. Всі розрахунки вірні!

^**— побітове виняткове “АБО”.**

Цей оператор схожий на звичайне «або». Різниця в одному: звичайне «або» повертає true, якщо хоча б один оперант є істинним. Але не обов’язково один – якщо обидва будуть true – то і результат true.

А виняткове “або” повертає true **тільки, якщо один з операндів є істинним. Якщо істинні обидва операнди,** звичайне “або” поверне true, а ось **виняткове або**  поверне false. Тому він і називається винятковим.

Знаючи принцип побітових операцій, можна легко виконати операцію 277^432.

**100010101** ^ **110110000** = **010100101 - результат роботи** ^

Ось і результат. Ті біти, котрі були в обох числах однакові, повернули 0 (не спрацювала формула «один з»). А ось ті, котрі утворювали пару 0 - 1 чи 1 - 0, перетворилися в одиницю. В результаті ми отримали число 010100101. В десятковій системі йому відповідає число 165. Поглянемо, чи правильно ми порахували:

Приклад 5.

public class Main {

public static void main(String[ ] args) {

System.out.println(277^432);

}

}

**Вивід у консолі: 165**

**Інкрементування та декрементування**

*Інкремент*, позначається **+ +** та збільшує значення змінної на одиницю. *Декремент* **позначається - -** та зменшує значення на одиницю. Інкремент та декремент часто використовують в циклі *for*.

Знаки операції інкременту та декременту можуть розміщуватися як до, так і після змінної. Ці варіанти називаються відповідно префіксним та постфіксним записами цих операцій. Знак операції в префіксному записі повертає значення свого операнда *після* вираховування виразу. При постфіксному записі знак операції *спочатку* повертає значення свого операнда і тільки після цього вираховує інкремент чи декремент.

Різниця префіксної та постфіксної форм проявляється тільки тоді, коли ці операції присутній у виразах. Якщо форма префіксна, то спочатку відбудеться збільшення (чи зменшення) змінної на одиницю, а потім з новим значенням будуть зроблені подальші обрахування. Якщо форма постфіксна, то розрахунок виразу буде відбуватися зі старим значенням змінної, а змінна збільшеться (чи зменшиться) на одиницю після обрахування виразу.

Приклад 6.

int x = 1, y, z;

y = ++x;

z=x++;

Змінній y буде присвоєне значення 2, оскільки спочатку значення x буде збільшено на 1, а потім результат буде присвоєний змінній y. Змінній z буде присвоєне значення 2, оскільки спочатку змінній z буде присвоєне значення, а потім значення x буде збільшено на 1. В обох випадках нове значення змінної x дорівнюватиме 2. (Після виконання цього фрагмента кода значення **х** дорівнюватиме **3**).

Слід відзначити, що в Java, на відміну від мови C, операції декремента та інкремента можуть застовуватися і до дійсних змінних (типу float та double).

Приклад 7.

int a = 3, b = 3;

// prefix = 1

int prefix = 15 / 3 - ++a;

// postfix = 2

int postfix = 15 / 3 - b++;

Приклад 8.

int a = 3, b = 3;

// на консоль виведе 4

System.out.println(++a);

//на консоль виведе 3

System.out.println(b++);

та ще один приклад

Приклад 9.

int x = 3, y = 3;

x = x++ + ++x;

y = --y - y--;

**Побітові операції**

Побітові операції розглядають вихідні числові значення як поля бітів та виконують над ними наступні дії:

* встановлення біта в *i*-ій позиції поля результату в 1, якщо обидва біти в *i*-их позиціях операндів дорівнюють 1, чи 0 в іншому випадку – побітове І ("&");
* встановлення біта в *i*-ій позиції поля результату в 1, якщо хоча б один біт в *i*-их позиціях операндів дорівнюють 1, чи в 0 в іншому випадку – побітове АБО ("|");
* встановлення біта в *i*-ій позиції поля результат в 1, якщо біти в *i*-их позиціях операндів не рівні один одному, чи в 0 в іншому випадку – побітове виключення АБО ("^");
* зсув вліво бітів поля першого операнду на кількість бітів, котра визначається другим операндом (біт знаку числа при цьому не змінюється) – **побітовий зсув вліво з урахуванням знаку** "<<";
* зсув вправо бітів поля першого операнда на кількість бітів, котра визначається другим операндом (біт знака числа при цьому не змінюється) – **побітовий зсув вправо з урахуванням знаку** ">>";
* зсув вправо бітів поля першого операнда на кількість бітів, котра визначається другим операндом (біт знака числа при цьому також посувається) – **побітовий зсув вправо без урахування знаку** ">>>".

Приклади побітових операцій:

Пример 10.

**Побітове І**

int x = 112;

// x: 00000000 00000000 00000000 01110000

int y = 94;

// y: 00000000 00000000 00000000 01011110

int z;

z = x & y;

//z=80:00000000 00000000 00000000 01010000

Приклад 11.

**Побітове АБО**

int x = 112;

// x: 00000000 00000000 00000000 01110000

int y = 94;

// y: 00000000 00000000 00000000 01011110

int z;

z =x | y;

// z = 126:

// z: 00000000 00000000 00000000 01111110

Приклад 12.

**Побітовое виключне АБО**

int x = 112;

// x: 00000000 00000000 00000000 01110000

int y = 94;

// y: 00000000 00000000 00000000 01011110

int z;

z =x ^ y;

// z = 46;

// z: 00000000 00000000 00000000 00101110

**Зсув вліво з урахуванням знаку**

Оператор зсуву вліво, <<, зміщує всі біти вліво на вказану кількість позицій. Число зправа доповнюється нулем:

значення << кількість

Параметр *кількість* вказує, на скільки потрібно зсунути вліво біти в параметрі *значення*. Операція використовується для швидкого множення на 2. При кожному зсуві вліво найстарший біт зсувається за межі допустимого значення та губиться, а зправа дописується нуль. Це означає, що при застосування оператора зсуву вліво до операнда типу *int* біти губляться, як тільки вони зсуваються за межі 31 позиції. Якщо операнд має тип *long*, біти губляться після зсуву за межі 63 позиції.

Якщо оператор застосовується до числа, множення на 2 котрого буде більше максимально значення (2147483647), то результатом буде від’ємне число. Це відбується тому, що крайній лівий біт, котрий відповідає за знак числа, ставиться в 1, що дорівнює від’ємним числам.

Автоматичне підвищення типу, що використовується в Java, може призвести до дивних результів при виконанні зсув в значеннях типу *byte*, *short*. При вираховуванні виразів типу значень **byte** та **short** підвищується до типу **int**. Це означає, що результатом виконання зсуву вліво значення типу **byte** или **short** буде значення **int**, та зсунуті вліво позиції не будуть відкинуті до тих пір, поки вони не будуть зсунуті за межі 31 позиції.

Приклад 13.

int x = 31, z;

// x: 00000000 00000000 00000000 00011111

z = x << 2;

// z = 124

// z:00000000 00000000 00000000 01111100

**Зсув вправо з урахуванням знаку (арифметичний)**

Оператор зсуву вправр, >> , зміщує всі біти значення вправо на вказану кількість позицій, знак при цьому копіюється:

значення>> кількість

Параметр кількість позицій вказує, на скільки потрібно зсунути вправо біти в значенні. Наступний код, для значення 32, виконує зсув бітів вправо на 2 позиції, в результаті чого значення змінної стає рівним 8.

Приклад 14.

int a = 32;

a = a >> 2 // тепер дорівнює 8

Крайні біти при зсуві просто губляться. Наприклад, значення 32 при зсуві вправо на дві позиції також дорівнює значенню 8, так як губляться два молодших біта.

00100000 32

>>2

-------------

00001000 8

При кожному зсуві вправо виконується поділ на два з відкиданням будь-якого залишку. Ця операція набагато ефективніша, ніж звичайний поділ на два, тому часто використовується в складних обрахуваннях. При цьому потрібно бути впевненим, що ніякі біти не будуть зсунуті за межі першої границі.

При виконанні зсуву вправо старші (розміщені в крайніх лівих позиціях) біти, вільні розміщуються в результаті зсуву, заповнюючи попереднім вміст старших бітів. Цей ефект називається додатковим знаковим розрядом та служить для зберігання знака від’ємних чисел при їх зсуві.

Приклад 15.

int x = -17, z;

z = x >> 2;

// z = -5:

Приклад 16.

int a=35;

a=a>> 2;

System.out.println(a);// a=8

00100011 35

>> 2

----------------

00001000 8

**Зсув вправо без урахування знаку**

Як було показано вище, при кожному виконанні оператор >> автоматично заповнює старший біт його попереднім вмістом. В результаті знак значення зберігається. Однак іноді це не є бажаним. Наприклад, при роботі зі значеннями пікселей та графічними зображеннями. Як правило, в цих випадках потребується зсув нуля в позицію старшого біта незалежно від його початкового значення. Таку дію називають зсувом вправо без урахування знака, коли завжди вставляється нуль в позицію старшого біта.

Приклад 17:

int x = -17, z;

// x: 11111111 11111111 11111111 11101111

z = x >>> 2;

// z = 1073741819

// z: 00111111 11111111 11111111 11111011

#### **Комбіновані операції**

В Java для бінарних арифметичних операцій можна використовувати комбіновані (складові) знаки операцій:

Ідентифікатор операція = вираз

Це еквівалентно наступній операції:

ідентифікатор = ідентифікатор операція вираз

В них використовується комбінація з двох операцій:

* присвоєння
* арифметична операція

#### 

#### **Операції порівняння**

В Java визначено наступні операції порівняння:

"= =" (дорівнює), "!=" (не дорівнює),

">" (більше), ">=" (більше або дорівнює),

"<" (менше) "<=" (менше чи дорівнює)

Ці операції мають два операнди та повертають булеве значення, котре відповідає результату порівняння (false чи true). Варто звернути увагу, що при порівнянні двох величин на рівність в Java, як і в C та в C++, використовуються символи "= =" (два, що йдуть без пробілу один за одним знаки рівності), на відміну від оператора присвоєння, в якому використовується символ "=".

**Приклад**

boolean isEqual, isNonEqual, isGreater,

isGreaterOrEqual, isLess, isLessOrEqual;

int x1 = 5, x2 = 5, x3 = 3, x4 = 7;

// isEqual = true

isEqual = x1 = = x2;

// isNonEqual = false

isNonEqual = x1 != x2;

// isGreater = true

isGreater = x1 > x3;

// isGreaterOrEqual = true

isGreaterOrEqual = x2 >= x3;

// isLess = true

isLess = x3 < x1;

// isLessOrEqual = false

isLessOrEqual = x1 <= x3;

#### **Булеві операції**

Булеві операції виконуються над булевими змінними та їх результатом також є значення типу boolean. В Java визначені наступні булеві операції (Таблиця 2.).

Табл. 2. Опис булевих операцій

|  |  |
| --- | --- |
| Оператор | Опис |
| & | Логічне AND (І) |
| && | Скорочене AND |
| | | Логічне OR (АБО) |
| || | Скорочене OR |
| ^ | Логічне XOR (виключне OR (АБО)) |
| ! | Логічне унарне NOT (НЕ) |
| &= | AND з привласненням(присвоєнням) |
| |= | OR з привласненням(присвоєнням) |
| Оператор | Описание |
| ^= | XOR з привласненням(присвоєнням) |
| == | Дорівнює |
| != | Не дорівнює |
| ?: | Тернарний (потрійний) умовний оператор |

* **заперечення** "!" – змінює false на true, чи навпаки;
* операція І "&" – результат дорівнює true, тільки, якщо обидва операнди дорівнюють true, інакше результат – false;
* операція АБО "|" – результат дорівнює true, тільки, якщо хоча б один з операндів дорівнює true, інакше результат – false.
* операція виключне АБО "^" – результат дорівнює true, тільки, якщо операнди не дорівнюють один одному, інакше результат – false.

Операції "&", "|" та "^" можна, також як і відповідні побітові операції, використовувати в складових операціях привласнення/присвоєння: "&=", "|=" та "^=".

Крім того, до булевих операнд також застосовуються операції "= =" (дорівнює) та "!=" (не дорівнює).

Як видно з визначення операції АБО та І (Таблиця 3), операція АБО призводить до результату true, коли перший операнд дорівнює true, незалежно від значення другого операнда, а операція І призводить до результату false, коли перший операнд дорівнює false, незалежно від значення другого операнду.

Табл.3. Результати виконання логічних операторів

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | A | B | A & B | A ^ B | !A |
| false | false | false | false | false | true |
| true | false | true | false | true | false |
| false | true | true | false | true | true |
| true | true | true | true | false | false |

**Скорочені логічні оператори**

Крім стандартних операторів **AND (&)** та **OR ( | )** існують скорочені оператори **&&** та **||**.

Якщо поглянути на таблицю 4., то видно, що результат виконання оператора **OR** дорівнює **true**, коли значення операнду A дорівнює **true**, незалежно від значення операнду В. Аналогічно, результат виконання оператора **AND** дорівнює **false**, коли значення операнду A дорівнює **false**, незалежно від значення операнду B. Виходить, що нам не потрібно обраховувати значення другого операнду, якщо результат можна визначити вже за першим операндом. Це стає зручним в тих випадках, коли значення правого операнду залежить від значення лівого.

Табл. 4. Особливості логічних операторів

**AND (&)** та **OR ( | )**

|  |  |  |  |
| --- | --- | --- | --- |
| **А** | **В** | **А | B** | **A & B** |
| false | false | false | false |
| true | false | true | false |
| false | true | true | false |
| true | true | true | true |

Приклад

boolean isInRange, isValid, isNotValid,

isEqual, isNotEqual;

int x = 8;

isInRange = x > 0 && x < 5;

// isInRange = false

isValid = x > 0 || x > 5; // isValid = true

isNotValid = !isValid;

// isNotValid = false

isEqual = isInRange == isValid; // isEqual = false

// isNotEqual = true

isNotEqual = isInRange != isValid

Приклад

public class SoklogOpe {

public static void main(String[] args) {

int dem = 5;

if(dem != 0 && 10/dem > 0)

System.out.print("Можна виконати операцію" + "ділення");

else

System.out.print("Не можна виконати операцію" + "ділення");

}

}

#### **Тернарний (потрійний) умовний оператор**

Умовна операція записується в формі

вираз-1? вираз -2: вираз -3.

При цьому спочатку обчислюється вираз вираз -1, котре повинне повернути булеве значення, а потім буде виконаний вираз-2 якщо значення true, або вираз -3, якщо вираз -1 повертає false).

Приклад 22.

x=n>1? 0: 1;

Змінній x буде присвоєне значення 0, якщо n > 1 (вираз n > 1 повертає true) або 1, якщо n ≤ 1 (вираз n > 1 повертає false).

**Пріоритети операції**

Операції в виразах виконуються зліва направо, однак, у відповідності зі своїм пріорітетом. Так операція множення у виразі

y = x + z\*5;

буде виконана раніше ніж операція додавання, оскільки пріорітет операції множення вище, ніж операції додавання.

Пріоритети операцій (в порядку зменшення пріорітету) в Java наведені в таблиці 5.

Табл. 5. Пріорітети операцій

|  |  |
| --- | --- |
| Пріоритет | Оператори |
| 1 | Префіксі оператори ++ та - - |
| 2 | Постфіксні оператори ++, - -, ~ та ! |
| 3 | + / % |
| 4 | + - |
| 5 | >> >>> << |
| 6 | > >= < <= |
| 7 | = = != |
| 8 | & |
| 9 | ^ |
| 10 | | |
| 11 | && |
| 12 | | | |
| 13 | ?: |
| 14 | = += -= /= %= &= ^= |= |

Круглі дужки підвищують старшинство операцій, котрі знаходяться всередині них. Так, якщо в наведений вище вираз вставити дужки:

y = (x + z) \* 5;

то спочатку буде виконана операція додавання, а потім операція множення.

Іноді дужки використовують просто для того, щоб зробити вираз більш читабельним, наприклад:

(x > 1) && (x <= 5).

#### **Умовний оператор вибору (if – else)**

Оператор **if** дозволяє програмі в залежності від мов програмування виконати оператор чи групу операторів, ґрунтуючись на значенні булевої змінної або виразу. Оператор **if** є основним оператором вибору в Java та дозволяє вибірково змінювати хід виконання програми – і це одна з основних відмінностей між простим обраховуванням.

Умовний оператор Java має такий формат:

if (*булевий - вираз*)

*оператори-1*;

else

*оператори-2*;

Якщо значення булевого-виразу дорівнює true, то виконуються оператори-1, інакше виконуються оператори-2. Оператор else може бути пропущеним, у випадку, якщо булевий-вираз дорівнює false, виконується оператор, котрий є наступним після оператора if. Якщо оператори-1 чи оператори-2 містять декілька операторів, вони повинні бути укладені в фігурні дужки (блоки).

***Вкладені оператори if***

Достатньо популярною є конструкція із вкладених умовних операторів. Ідея достатньо проста: в одному умовному операторі викликається інший умовний оператор, в якому може викликатися третій умовний оператор, і т. д.

Синтаксис команди:

If ()

else if (**умова\_1**) {**оператори\_1**}

else if (**умова \_2**) { **оператори \_2**}

else if (**умова \_3**) { **оператори \_3**}

. . .

else if (**умова \_N**) { **оператори\_N**}

else { **оператори\_N+1**}

**Приклад 1.**

if(i = = 10)

{

if(j < 20) a =b;

if(k > 100) c =d;

**// else відносить до if (k > 100)**

else a = c;

}

**// else відносить до if (i == 10)**

else a = d;

**Приклад**

int month = 3; // березень

String season; // пора року

if(month == 1 || month == 2 || month == 12)

season = "Зима";

else if (month == 3 || month == 4 || month == 5)

season = "Весна";

else if (month == 6 || month == 7 || month == 8)

season = "Літо";

else if (month == 9 || month == 10 || month == 11)

season = "Осінь";

else season = "Ви з якої планети?";

System.out.println("Листя розпускаються " + season);

Приклад

if (x = = 0)

y = 0;

else

{

y = 1;

z = x + 1;

}

**Оператор розгалуження switch**

Оператор варіанту switch організовує розгалуження за декількома напрямками. Кожна гілка позначається константою чи константним виразом будь-якого цілого типу (крім long) та обирається, якщо значення визначеного виразу співпадає з цією константою. Оператор switchмає наступний формат**:**

switch (вираз)

{

case значення-1: оператори-1;

case значення-2: оператори-2;

…

default: оператори-n;

}

Вираз, котрий ставиться в круглих дужках після ключового слова switch, може належати до одного з типів char, byte, short або int, а також однією зі значень змінної лічильного типу. Перевірка значення цього виразу на рівність заданому константному значенню здійснюється за допомогою операторів case, котрі входять до оператора switch (всі оператори case повинні містити різноманітні значення). Як тільки один із операторів case опізнає значення виразу, керування отримують оператори, наступні після символу ":" після цього оператора case.

Після цього виконуються оператори, котрі містяться в усіх наступних операторах case і в операторі default. Для того, щоб обійти виконання наступних операторів case та оператора default, слід задати останнім оператором для даного case оператор break. Якщо значення *виразу* не співпадає з жодним із *значень*, виконуються оператори, наступні після оператора default (цей оператор може бути пропущеним). Оператори, наступні після двокрапки, можна не заключати у фігурні дужки.

Таким чином, константи у варіантах case виступають у ролі міток, точок входу в оператор варіанту, а далі виконуються всі оператори, що залишилися, в порядку їх запису.

Після виконання одного варіанту оператор **switch** продовжує виконувати всі варіанти, що залишилися.

Може знадобитися виконати один і той самий оператор в різних гілках case. В цьому випадку ставлять декілька міток case один за одним. Ось простий приклад.

Приклад

switch(dayOfWeek)

{

case 1: case 2: case 3: case 4:case 5:

System.out.println("Week-day");

break;

case 6: case 7:

System.out.println("Week-end");

break;

default: System.out.printlnt("Unknown day");

}

Потрібно не забувати завершувати варіанти оператором break.

Приклад

int month = 3;

String monthString;

switch (month) {

case 1: monthString = "Січень";

break;

case 2: monthString = "Лютий";

break;

case 3: monthString = "Березень";

break;

case 4: monthString = "Квітень";

break;

case 5: monthString = "Травень";

break;

case 6: monthString = "Червень";

break;

case 7: monthString = "Липень";

break;

case 8: monthString = "Серпень";

break;

case 9: monthString = "Вересень";

break;

default: monthString = "Не знаємо"+ "такого";

break;

}

System.out.println(monthString);

Приклад

Наступний приклад випадковим чином генерує англійські букви. Програма визначає голосні вони чи не приголосні:

import java.util.\*;

Random random = new Random();

for (int i = 0; i < 100; i++){

int c = random.nextInt(26) + 'a';

System.out.println((char)c + ", " + c + ": ");

switch (c) {

case 'a':

case 'e':

case 'i':

case 'o':

case 'u':

System.out.println("Голосна");

break;

case 'y':

case 'w':

System.out.println("Умовна голосна");

break;

default: System.out.println("Приголосна");

break;

} }

**Реалізація циклічних операторів**

В Java визначені наступні оператори циклу:

* оператор for;
* розширений оператор for;
* оператор while;
* оператор do…while.

**. Реалізація циклічного оператору for**

Оператор циклу for має наступний формат:

for (ініціалізація-циклу; контрольний-вираз; кроковий-вираз)

тіло-циклу

Ініціалізація-циклу виконується один раз перед першим проходом тіла циклу. Секція ініціалізації дозволяє ініціалізувати змінну циклу, котра буде мати область видимості, обмежену цим циклом. Секція ініціювання виконується один раз під час входу в цикл.

Після виконання ініціалізації в контрольному-виразі перед початком кожного кроку виконання тіла циклу (в тому числі і перед його першим виконанням) перевіряється логічний вираз. Якщо умова повертає **true**, то виконується тіло циклу, в протилежному випадку виконання циклу зупиняється. Кроковий-вираз обраховується в кінці кожного виконання тіла циклу.

Як правило, в ньому відбувається збільшення або зменшення значення деякої змінної, від якої залежить значення контрольного виразу. Тіло циклу може містити один або декілька операторів, котрі укладені у фігурні дужки. Процес виконується до тих пір, поки умова не поверне значення **false**.

Приклад

Найпростіший приклад:

for (int x = 0; x < 9; x = x + 1)

System.out.println("\n Значення x: " + x);

Суворо кажучи, в такому вигляді (x = x + 1) сучасні програмісти не пишуть, так як є скорочена форма запису **(**x++**).** Попередній приклад можна переписати по-іншому:

for (int x = 0; x < 9; x++)

Цей запис є класичним і правильним, якщо нам потрібно порахувати від 0 до 8. Може виникнути спокуса написати, наприклад, так:

for (int x = 0; x <= 8; x++)

Результат буде таким самим, але такий код є небажаним.

**Старайтесь писати традиційно**. Особливо це проявляється у роботі з масивами.

Всі секції заголовку циклу **for** є необов’язковими.

Цикл без усіх трьох секцій стає нескінченим, так як в якості умови розуміється логічний літерал true.

for (; ; ) {

Інструкція

}

Можна вкласти дві змінні в оператор **for**, щоб не виконувати опрацювання другої змінної вручну:

Приклад

int a, b;

for (a = 1, b = 4; a < b; a++, b--) {

System.out.println("a = " + a + "\n");

System.out.println("b = " + b + "\n");

}

Як видно з коду, кома є роздільником для двох змінних. Тепер обидва розділених комою оператори в ітераційній частині циклу виконується при кожному виконанні циклу. **Цей код вважається більш ефективним,** хоч на практиці зустрічається рідка.

Вкладені цикли:

* Один цикл може виконуватися всередині іншого.
* Всередині вкладеного циклу може бути вкладений ще один цикл, утворюючи наступний рівень вкладеності.
* Велика кількість рівнів вкладеності (3 та більше) значно ускладнює програму.

Приклад

Роздрукувати таблицю множення:

public class ТабУмн {

public static void main(String[] args) {

int i, j, a = 0;

for (i = 1; i < 10; i++)

{

for (j = 1; j < 10; j++) {

a = i \* j;

System.out.print(a + " | ");

}

System.out.println();

}

}}

Початок

Int i=1; i<10; i++

Int j=1; j<10; j++

***println(i+ " \* "+ j + " = " + i\*j)***

***println("\_\_\_\_\_\_\_\_\_\_")***

**Да**

**Да**

Кінець

ні

ні

так

так

Результати виконання програми:

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |

3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 |

4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |

5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |

6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |

7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |

8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |

9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

Приклад 10.

Підсумовування елементів масиву:

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

int sum = 0;

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

sum+=a[i];

В результаті виконання циклу змінної sum буде присвоєне значення 15.

**Реалізація циклічного оператора з передумовою (while)**

Найбільш часто використовуваним циклом є оператор **while.**

Синтаксис оператора

**while (умова)**

Тіло циклу

**{**

**інструкція**

**}**

Тут *умовою може* бути будь-який булевий вираз. Тіло циклу буде виконуватися до тих пір, поки умовний вираз є істинним. Коли *умова* повертає значення **false**, управління передається рядку коду, котрий йде після циклу. Якщо в циклі використовується тільки один оператор, то фігурні дужки можна опустити (але краще так не робити).

Логічний вираз вираховується перед початком циклу, а потім кожен раз перед виконанням чергового повтору оператора.

У випадку, якщо за першої перевірки умова отримає значення **false**, оператор не буде виконуватися жодного разу.

Якщо необхідно отримати цикл, котрий буде виконуватися нескінченно, в якості умови варто написати логічний літерал **true**.

While (true){

інструкція}

У цьому випадку необхідно передбачити вихід з циклу за допомогою команди **break**.

Приклад з використання циклу **while**, який виконує зворотний відлік від 10 до 0.

Приклад

int counter = 10;

while (counter > 0) {

System.out.println("Осталось " + counter + " сек.\n");

counter--;

}

Приклад 12.

Якщо потрібно збільшувати від одиниці до 10, то код буде наступним.

while(counter < 11){

System.out.println(counter);

counter++;

}

Приклад 13.

Тіло циклу **while** може бути пустим. Наприклад:

int i, j;

i = 10;

j = 30;

// обраховуємо середнє значення двох змінних

while (++i < --j)

; // у циклі немає тіла

System.Out.Println ("Среднє значення" + " дорівнює" + i);

Професійні програмісти часто використовують цикли без тіла, в яких сам по собі керуючий вираз може виконати всі необхідні дії.

**Реалізація циклічних операторів із пост умовою (do - while)**

У випадку, якщо тіло циклу має виконуватися хоч один раз потрібно виконати оператор **do … while**.

Синтаксис оператора:

do

{

інструкція

}

while(умова)

Цей оператор виконує тіло циклу хоч один раз, тому що умова перевіряється в кінці циклу. Цикл виконується до тих пір, поки в результаті перевірки вираз не поверне значення **false**.

Приклад 14.

int counter = 10;

do {

System.Out.Println("Осталось " + counter + " сек.\n");

counter--;

}

while (counter > 0);

Приклад 15.

Якщо тепер змінити умову **counter < 0**, то цикл відпрацює один раз і виведе один рядок:

int counter = 10;

do {

System.Out.Println ("Осталось " + counter + " сек.\n");

}

while (--counter > 0);

В цьому прикладі об’єднали декремент лічильника та порівняння з нулем в одному виразі. Програма працює таким чином. На початку вона виконує операцію зменшення на одиницю та повертаючи нове значення лічильника. Потім порівнюється значення з нулем. Якщо воно більше нуля, виконання циклу продовжується. В іншому випадку цикл переривається.

Приклад 16.

Ще один приклад циклу while:

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

int sum = 0, i = 0;

while (i < a.length)

{

sum+=a[i];

i ++;

}

Приклад 17.

Порівняння виконання операторів while та do…while:

|  |  |
| --- | --- |
| int a[]={1,2,3,4,5};  int sum=0, i=0;  while (i < 0)  {  sum+=a[i];  i++;  } | int a[] = {1,2,3,4,5};  int sum=0, i=0;  do  {  sum+=a[i];  i ++;  }  while(i < 0); |

У першому випадку цикл не буде виконаний жодного разу та значення змінних sum та i дорівнюватимуть 0, в іншому випадку цикл буде виконано один раз та значення змінних sum та i дорівнюватимуть 1.

**ОПЕРАТОРИ ПЕРЕХОДУ break, continue, return**

В Java відсутній оператор

goto мітка

мови C, котрий дозволяє виконати перехід до оператора зі вказаною *міткою,* оскільки цей оператор може істотно ускладнити логічну структуру програми. В цих же випадках, коли необхідно пропустити виконання деяких операторів програми, в Java використовуються оператор переривання break та оператор продовження continue. До операторів переходу зазвичай відносять і оператор return.

**Оператор** **break**

Оператор **break** завершує послідовність операторів в операторі [switch](http://developer.alexanderklimov.ru/android/java/switch.php), дозволяє вийти з циклу та може бути використаним в якості безумовного переходу (goto).

Розглянемо приклад виходу з циклу. Використовуючи оператор **break**, можна викликати негайне завершення циклу, пропускаючи умовний вираз та інший код у тілі циклу. Коли додаток зустрічає оператор **break** всередині циклу, воно зупиняє виконання циклу та передає керування тому оператору, котрий йде наступним після циклу.

Приклад 18.

do { вираз чи блок;

if (умова передчасного виходу)

break;

}

while (умова продовження циклу);

При використанні вкладених циклів оператор **break** здійснює вихід тільки з самого внутрішнього циклу, не впливаючи на зовнішній цикл.

Приклад 19.

Створимо приклад з використанням вкладених циклів:

for (int i = 1; i < 4; i++) {

System.out.println("Прохід " + i + ": ");

for (int j = 0; j < 100; j++) {

// вихід із циклу, якщо j дорівнює 5

if (j == 5) break;

System.out.println(j + " ");

}

System.out.println("\n");

}

System.out.println("Цикли завершили" + "роботу");

Оператор переривання:

break мітка;

передає керування за межі циклу чи оператора вибору рядку коду, котрий відмічений вказаною *міткою.* *Мітка* є звичайним ідентифікатором Java, за котрою йде двокрапка.

Приклад 20.

Приклад використання оператора break з *міткою* (визначення індексів першого нульового елементу в двовимірному масиві):

int a[ ][ ] = {{1,2,3},{5,6,7},{8,9,0},{11,12,13}};

int i, j;

…

iLoop:

for (i = 0; i < 4; i++)

{

for (j = 0; j < 3; j++)

{

if (a[i][j] = = 0)

break iLoop;

}

}

У цьому випадку необхідно перервати виконання не тільки того циклу, в якому знаходиться оператор break (циклу по j), але і зовнішнього по відношенню до нього циклу (циклу по i), інакше виконання зовнішнього циклу буде продовжено. Використання мітки iLoop в операторі break дозволяє перервати виконання зовнішнього циклу.

**Оператор продовження continue** схожий на оператор **break**. Він має дві форми. *Перша форма* складається тільки зі слова **сontinue** та здійснює негайний перехід до наступної ітерації циклу. Якщо мітка опущена, оператор continue передає керування на самий кінець тіла циклу (після останнього його оператора). Це буває корисно для того, щоб пропустити при виконанні тіла циклу деякі з операторів.

Приклад.

В черговому фрагменті коду оператор **continue** дозволяє обійти ділення на нуль:

for (int i=0; i<N; i++)

{

if(i==j) continue;

s+=1.0/(i-j);

}

Приклад 22.

У наступному прикладі виводимо по два числа в кожному рядку:

for (int i = 0; i < 10; i++) {

System.out.println(i + " ");

if (i % 2 == 0)

continue;

System.out.println("\n");

}

У цьому коді оператор % слугує для перевірки парності значення змінної *i*. Якщо значення парне, виконання циклу продовжиться далі, а якщо непарне, то відбудеться перехід на новий рядок.

*Друга форма* містить мітку:

**continue *мітка***

мітка записується, як всі ідентифікатори, з букв Java, цифр та знаку підкреслення, але не потребує ніякого опису. Мітка ставиться перед оператором чи відкриваючою фігурною дужкою та відділяється двокрапкою. Так виходить помічений оператор чи помічений блок.

Друга форма використовується тільки у випадку декількох вкладених циклів для негайного переходу до чергової ітерації одного з зовнішніх циклів, а саме, поміченого циклу.

Приклад 23.

Створимо приклад виводу трикутної таблиці множення чисел від 0 до 9.

outer: for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

if (j > i) {

System.out.println("\n");

continue outer;

}

System.out.println(" " + (i \* j));

}

}

System.out.println("\n");

В цьому прикладі оператор **continue** перериває цикл підрахунку значень змінної **j** та продовжує його з наступної ітерації циклу підрахунку змінної **i.**

Приклад 24.

Приклад використання оператора continue **(**підсумовування елементів масиву з парними значеннями):

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

int sum = 0;

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

{

if(a[i]%2 != 0)

continue;

sum += a[i];

}

Виконання оператора sum += a[i]; в циклі буде пропущене для тих значень елементів масиву, для яких залишок при діленні на 2 не дорівнює 0 (непарних чисел).

**Оператор return** використовують для виконання явного виходу з методу. Оператор можна використовувати в будь-якому місці методу для повернення управління тому об’єкту, котрий викликав даний метод. Таким чином, **return** зупиняє виконання методу, в якому він знаходиться.

**Завдання для виконання**

1. **Перейти за посиланням:**[**https://itvdn.com/ua/skills/practicums/java-starter**](https://itvdn.com/ua/skills/practicums/java-starter)
2. **Авторизуватися або зареєструватися на сайті.**
3. **Виконати завдання перших 2 блоки:**  
   **3.1 Змінні.**  
   **3.2 Умовні конструкції.**
4. **Зробити скріншоти після виконання останнього завдання кожного блоку (2 скріншоти), щоб було видно Ваш обліковий запис.**
5. **Прикріпити та надіслати скріншоти до завдання у Teams.**