Міністерство освіти і науки України

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ МОВОЮ JAVA

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за спеціальністю 176 Мікро- та наносистемна техніка

Укладачі: Д. Д. Татарчук, Ю. В. Діденко, Г. С. Свєчніков

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2024

УДК 004.43(075.8)
О–29

Укладачі: *Татарчук Дмитро Дмитрович*, д-р техн. наук, доц.
Діденко Юрій Вікторович, канд. техн. наук, доц.
Свєчніков Георгій Сергійович, канд. фіз.-мат. наук, старш. наук. співроб.

Рецензент *Казміренко В. А.*, кандидат технічних наук, доцент, доцент кафедри електронної інженерії, КПІ ім. Ігоря Сікорського

Відповідальний редактор *Обухова Т. Ю.*, канд. техн. наук, доц.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 2 від 08.11.2024 р.)
за поданням вченої ради факультету електроніки
(протокол № 10/2024-2 від 21.10.2024 р.)

О–29 **Об'єктно-орієнтоване програмування мовою Java** [Електронний ресурс] : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмою «Мікро- та наноелектроніка» спец. 176 Мікро- та наносистемна техніка / КПІ ім. Ігоря Сікорського ; уклад.: Д. Д. Татарчук, Ю. В. Діденко, Г. С. Свєчніков. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2024. – 153 с.

У посібнику викладено основи програмування мовою Java, розглянуто вбудовані типи даних, оператори та базові модулі мови. Особливу увагу приділено засобам об’єктно-орієнтованого програмування мови Java. Окремо розглянуто мережеві засоби мови програмування Java.

Посібник призначений для студентів-бакалаврів, які навчаються за освітньо-професійною програмою «Мікро- та наноелектроніка», а також всім хто бажає вивчити основи мови програмування Java.

УДК 004.43(075.8)

Реєстр. № НП 24/25-126. Обсяг 6,9 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Зміст

Перелік умовних скорочень і позначень

IP – інтернет протокол (internet protocol)

JDK – набір для розробки мовою Java (Java development kit)

TCP – протокол управління передачею (transmission control protocol)

UDP – протокол дейтаграм користувача (user datagram protocol)

ООП – об’єктно орієнтоване програмування

Вступ

Java – одна з найпоширеніших мов програмування. Вона має розвинені засоби для програмування мережевих задач, для роботи з базами даних для розробки сучасного користувацького інтерфейсу тощо. Реалізація програм на мові Java не залежить від операційної системи та типу комп'ютера. Тому Java програми легко переносяться з однієї платформи на іншу, що зумовлює широке використання мови програмування Java для розробки різноманітного програмного забезпечення, в тому числі програм для Інтернету. З вищевказаного стає зрозумілим, що знання мови програмування Java є дуже важливим для тих, хто хоче створювати сучасне програмне забезпечення.

Даний посібник призначений для студентів кафедри мікроелектроніки факультету електроніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського». Студенти даної кафедри починають своє знайомство з мовами програмування з вивчення мов С та С++. Тому викладення матеріалу в даному посібнику буде побудоване на подібності мов Java та С++.

За своїм синтаксисом Java дуже схожа на мову програмування С++, хоча і має свої особливості. Основною відмінністю Java від С++ є те, що всі програми Java об'єктно орієнтовані. Написати на Java програму без використання об'єктів неможливо. Навіть екземпляри простих змінних мови Java є об'єктами певних класів. Тому у людей, які вивчали мову програмування С++, як правило, не виникає значних труднощів при вивченні мови програмування Java. Більшість простих типів даних мови Java майже не відрізняються від подібних типів мови С++, а оператори Java можна розглядати як удосконалені оператори мови С++. Дещо важче проходить вивчення типів даних відсутніх у С++, але наполеглива праця дозволяє вирішити цю проблему. Тож почнемо розгляд мови програмування Java за тією ж схемою за якою розглядали мови С та С++.

Основи програмування мовою Java

Алфавіт мови програмування Java

Множина допустимих символів

Множина допустимих символів мови Java містить великі (A-*Z*) та малі (a-z) літери латинського алфавіту й арабські цифри (0-9), символ підкреслювання та знак долару ($) [1-3].

Пробільні символи

Символи «пробіл», «табуляція», «перехід на нову строку», «повернення каретки», «нова сторінка», «вертикальна табуляція», «горизонтальна табуляція» називають пробільними, оскільки вони виконують ті ж функції, що й пробіли між словами в тексті. Розглядається як пробільний і символ кінця файлу. Коментарі компілятор мови Java також розглядає як пробільні символи. Компілятор мови Java ігнорує ці символи, якщо їх використовують не в складі символьних змінних і констант.

Розділові символи

В мові програмування Java використовують той же самий набір розділових символів, що і в мовах C та C++ (таблиця 1). Як і в C та C++ ці символи мають спеціальний зміст.

Спеціальні символи

Спеціальні символи призначені для представлення пробільних символів, невідображуваних символів і символів, які мають спеціальне призначення. Спеціальні символи мови Java також співпадають з спеціальними символами мов C та C++ (таблиця 2) [1-3].

Таблиця 1. Розділові символи

Символ	**Назва**	**Символ**	**Назва**
,	Кома	.	Крапка
;	Крапка з комою	:	Двокрапка
?	Знак питання	'	Одинарні лапки (апостроф)
!	Знак оклику	|	Вертикальна лінія
/	Слеш, знак ділення	\	Обернений слеш
~	Тильда	_	Підкреслювання
(	Ліва кругла скобка	)	Права кругла скобка
{	Ліва фігурна скобка	}	Права фігурна скобка
[	Ліва квадратна скобка	]	Права квадратна скобка
<	Знак «менше»	>	Знак «більше»
%	Процент	&	Амперсанд
#	Ґратка	^	Стрілка вгору
-	Знак мінус	=	Знак рівності
+	Знак плюс	*	Знак множення

Таблиця 2. Спеціальні символи

Символ	**Назва**	**Символ**	**Назва**
\n	Перехід на новий рядок	\r	Повернення каретки
\b	Знищення символу	\t	Горизонтальна табуляція
\f	Перехід на нову сторінку	\\	Обернений слеш
\''	Подвійні лапки	\'	Одинарні лапки
\ddd*	Вісімкова константа	\uxxx**	Шістнадцяткова константа

*d – вісімкова цифра
**x – шістнадцяткова цифра

Правила формування ідентифікаторів в Java

При формуванні ідентифікаторів в мові програмування Java необхідно дотримуватись наступних правил [3,4]:

- ідентифікатор має починатись з літери і може включати букви, цифри, символ підкреслювання та символ долару. В принципі, синтаксис мови дозволяє починати ідентифікатор з символу долара або символу підкреслювання, але це заборонено домовленістю по оформленню коду в Java (Java Code Conventions). Крім того символ долару, згідно з цією домовленістю, не використовують ніколи. Згідно з вищевказаною домовленістю, ім'я змінної має починатися з маленької літери латинського алфавіту, імена класів мають починатися з великої літери. Використання пробільних символів у ідентифікаторах не допускається;
- як ідентифікатор не можна використовувати ключове або зарезервоване слово Java;
- компілятор мови Java чутливий до регістру. Імена MyVar та myvar – це різні імена;
- при формуванні ідентифікаторів необхідно використовувати замість абревіатур повні слова англійської мови (не трансліт). Це дозволяє зробити код програми більш зрозумілим і зручним для супроводження;
- якщо ім'я змінної складається з одного слова, то його потрібно записувати маленькими літерами. Якщо ж воно складається з кількох слів, то кожне слово крім першого треба починати з великої літери (mySuperVariable);
- в іменах констант та змінних, які протягом роботи програми зберігають постійне значення, необхідно слова відокремлювати символом підкреслювання і набирати всі слова цілком великими літерами (NUMBER_OF_HOURS_IN_A_DAY).

Ключові слова

Ключові слова – це слова, які в мові програмування Java мають спеціальне значення. Наведемо приклади ключових слів [1-4]:

abstract	assert	boolean	break	byte	case
catch	char	class	const	continue	default
do	double	else	enum	extends	final
finally	float	for	goto	if	implements
import	instanceof	int	interface	long	native
new	package	private	protected	public	return
short	static	strictfp	super	switch	synchronized
this	throw	throws	transient	true	try
void	volatile	while			

Тут наведено далеко не всі ключові слова мови програмування Java. Повний список можна знайти у документації на конкретну версію мови програмування Java.

Коментарі в Java

Коментарі використовують для документування програми. Компілятором вони сприймаються як окремий пробільний символ та ігноруються при компіляції.

Мова Java підтримує три типи коментарів: однорядковий, багаторядковий та документаційний [1-5].

Однорядковий коментар починається символами // і закінчується в кінці рядка. Він займає завжди лише один рядок. Багаторядковий коментар міститься між символами /*….*/ і може займати декілька рядків. Документаційний коментар починається символами /** і закінчується

символами */. Він багаторядковий. Кожен рядок коментаря починається з символу *. Наведений нижче приклад демонструє використання коментарів:

```
/* даний приклад демонструє використання
коментарів це - багаторядковий коментар */
// це - однорядковий коментар

package prob;
/**
* а це документаційний коментар
* він може бути багаторядковим
*/

//оголошуємо класс HelloWorld
public class prob
{
    public static void main(String[] args) {
    // виводимо повідомлення Hello World!
    System.out.println("Hello, World!");
    }
}
```

Документаційні коментарі дуже функціональні і дозволяють створення окремої документації у вигляді сторінок HTML за допомогою інструменту Javadoc Але розгляд даних питань виходить за рамки нашого посібника.

Операції в мові Java

Операції – це дії, що виконуються над даними. Операції позначають спеціальними символами чи групами символів, що визначають дії з обробки

даних. Дані, над якими виконуються операції, називають операндами. Компілятор інтерпретує кожну з таких комбінацій як самостійну лексему. Наприклад, у виразі `x+y` змінні *x*, *y* – це операнди, а «+» – це символ, що означає операцію додавання.

Якщо операцію застосовують одночасно до двох операндів, то її називають бінарною (наприклад, додавання чи віднімання), а якщо тільки до одного операнда, то – унарною (наприклад, символ «−» у позначенні від'ємного числа). Кожна операція може застосовуватись лише до операндів певних типів. Операції мови програмування такі самі як у більшості С-подібних мов програмування, але мають деякі відмінності (таблиці 3,4,5,6). Операції повинні використовуватись в програмі саме так, як вони представлені у таблиці, без пробільних символів в тих операціях, які зображуються кількома символами [1-5].

Таблиця 3. Арифметичні операції

Операція	**Назва**	**Операція**	**Назва**
`+`	Додавання	`++`	Інкремент
`-`	Віднімання, унарний мінус	`--`	Декремент
`*`	Множення	`/`	Ділення
`%`	Залишок від ділення	`=`	Присвоювання
`-=`	Віднімання з присвоюванням	`+=`	Додавання з присвоюванням
`*=`	Множення з присвоюванням	`%=`	Присвоювання залишку від цілочисленного ділення
`/=`	Ділення з присвоюванням		

Таблиця 4. Логічні операції

!	Логічне НІ	|	Логічне АБО
&	Логічне І	^	Виключне АБО
&&	Скорочене логічне І	||	Скорочене логічне АБО
|=	АБО з присвоюванням	^=	Виключне АБО з присвоюванням
&=	І з присвоюванням	~	Побітове НІ
?:	Умовна операція (вираз 1 ? вираз 2 : вираз 3)		

Якщо логічні операції |, &, ^ застосовують до цілих чисел, то вони працюють як побітові логічні операції.

Таблиця 5. Операції відношення

>	Більше	<	Менше
<=	Менше чи дорівнює	>=	Більше чи дорівнює
!=	Не дорівнює	==	Перевірка на рівність

Таблиця 6. Операції зсуву

<<	Зсув вліво	>>	Зсув вправо
<<=	Зсув вліво з присвоюванням	>>=	Зсув вправо з присвоюванням
>>>	Беззнаковий зсув вправо	>>>=	Беззнаковий зсув вправо з присвоюванням

Крім представлених в таблицях 3–6 операцій в мові програмування Java існує ще ряд спеціальних операцій які будуть розглянуті пізніше.

Операнди й операції формують вирази. Кожен вираз в Java має своє значення (результат виконання операцій над операндами). У найпростішому випадку вираз може складатися лише з одного операнда. Операції у виразі виконуються у порядку їхнього пріоритету [1-5]. Пріоритет операцій наведено у таблиці 7. Пріоритет зменшується в таблиці зверху-вниз. Операції, що знаходяться в одному рядку таблиці, мають однаковий пріоритет. Для зміни порядку виконання операцій використовують дужки (операції, що розміщені в дужках, виконуються в першу чергу).

Таблиця 7. Пріоритет операцій

Операції	**Категорія операцій**
`() [] .`	Скобки і крапка
`++ -- ~` ! (приведення типів)	Унарні операції
`* / %`	Мультиплікативні операції (множення ділення і т.і.)
`+ -`	Адитивні операції (додавання, віднімання і т.і.)
`<< >> >>>`	Операції зсуву
`> < >= <= instanceof`	Операції відношення
`== !=`	
`&`	Логічні операції
`^`	
`|`	
`&&`	
`||`	
= (операції з присвоюванням)	Операції присвоювання

Більшість наведених операцій повністю аналогічна відповідним операціям мови C++ і не потребує особливого пояснення. Основні відмінності у логічних операціях. Так в мові C++ існують дві окремі групи логічних

операцій: побітові логічні операції та звичайні логічні операції. У мові Java такого поділу немає. Дія логічних операцій мови Java залежить від типу даних.

Коли логічні операції застосовують до цілих чисел, вони грають роль побітових операторів. Якщо їх застосовують до операндів типу boolean, то вони діють як звичайні логічні операції [1-5]. Даний факт ілюструється у програмі наведеній нижче:

```
public class boolean_prob
{
    public boolean_prob()
    {

    }

    public static void main(String[] args)
    {
        int a=3, b=5;
        boolean b1=true, b2=false, b3=true, b4=false;
        //використання логічних операцій до цілих чисел
        //0011 & 0101 = 0001
        System.out.println(a+" & "+b+"="+(a & b));
        //0011 | 0101 = 0111
        System.out.println(a+" | "+b+"="+(a | b));
        //0011 ^ 0101 = 0110
        System.out.println(a+" ^ "+b+"="+(a ^ b));
        System.out.println();
        //використання логічних операцій
        //до змінних типу boolean
        //&
        System.out.println(b1+" & "+ b2+"="+(b1 & b2));
```

```
        System.out.println(b1+" & "+ b3+"="+(b1 & b3));
        System.out.println(b2+" & "+ b3+"="+(b2 & b3));
        System.out.println(b2+" & "+ b4+"="+(b2 & b4));
        System.out.println();
        //|
        System.out.println(b1+" | "+ b2+"="+(b1 | b2));
        System.out.println(b1+" | "+ b3+"="+(b1 | b3));
        System.out.println(b2+" | "+ b3+"="+(b2 | b3));
        System.out.println(b2+" | "+ b4+"="+(b2 | b4));
        System.out.println();
        //^
        System.out.println(b1+" ^ "+ b2+"="+(b1 ^ b2));
        System.out.println(b1+" ^ "+ b3+"="+(b1 ^ b3));
        System.out.println(b2+" ^ "+ b3+"="+(b2 ^ b3));
        System.out.println(b2+" ^ "+ b4+"="+(b2 ^ b4));
        System.out.println();
    }

}
```

В результаті виконання даної програми на моніторі комп’ютера буде відображено наступні повідомлення:

```
3 & 5=1
3 | 5=7
3 ^ 5=6

true & false=false
true & true=true
false & true=false
false & false=false
```

```
true | false=true
true | true=true
false | true=true
false | false=false

true ^ false=true
true ^ true=false
false ^ true=true
false ^ false=false
```

Крім того в Java з’явилось кілька логічних операцій, яких не було в C++ – це так звані скорочені логічні операції (`&&` та `||`). Суть їх дії полягає в тому, що результат багатьох логічних операцій однозначно визначається значенням першого операнда і не залежить від другого (дивись результати виконання попередньої програми). Так, наприклад, якщо перший операнд операції логічного І `false`, то результат завжди буде `false` незалежно від другого операнда. Аналогічно, якщо перший операнд логічної операції АБО `true`, то результат завжди буде `true`. Це дає змогу не витрачати час на перевірку значення другого операнда і прискорити виконання програми.

Ще однією новою по відношенню до мови C++ є операція без-знакового зсуву вправо [3-5]. Вона діє як і звичайний зсув вправо з тією відмінністю, що звичайний зсув не діє на знаковий біт, а беззнаковий діє на знаковий біт. Тому відмінність у результатах проявляється лише для від’ємних чисел, що проілюстровано у наведеній нижче програмі:

```
public class unsigned_shift
{
        public static void main(String[] args)
        {
            int a=4,b=-4;
```

```
            System.out.println(a+">>1="+(a>>1));
            System.out.println(a+">>2="+(a>>2));
            System.out.println(a+">>>1="+(a>>>1));
            System.out.println(a+">>>2="+(a>>>2));
            System.out.println(b+">>1="+(b>>1));
            System.out.println(b+">>2="+(b>>2));
            System.out.println(b+">>>1="+(b>>>1));
            System.out.println(b+">>>2="+(b>>>2));
        }
}
```

В результаті виконання програми на дисплеї буде відображено наступне:

```
4>>1=2
4>>2=1
4>>>1=2
4>>>2=1
-4>>1=-2
-4>>2=-1
-4>>>1=2147483646
-4>>>2=1073741823
```

З наведених результатів видно, що для додатних чисел і беззнаковий і звичайний зсув еквівалентний цілочисловому діленню на ступені числа 2.

Оператори мови Java

Мова Java має майже такий самий набір операторів, що й мова C++(за винятком оператора `goto`, якого немає у мові Java, Java має зарезервоване слово `goto` , але воно не несе ніяких функцій). Проте деякі оператори Java мають свої особливості і додаткові можливості [1-5]:

- `;` – порожній оператор;
- `{}` – складений оператор;
- `if ...else...` – умовний оператор;
- `switch` – оператор вибору;
- `while` – цикл з передумовою ;
- `do...while` – цикл з післяумовою ;
- `for` – оператор покрокового циклу;
- `break` – оператор переривання циклу чи блоку switch;
- `continue` – оператор переходу до наступної ітерації циклу;
- `return` – оператор виходу з поточного методу.

Розглянемо оператори більш докладно.

Порожній оператор

Порожній оператор – це оператор, який складається лише з символу крапка з комою – « `;` ». Цей оператор не виконує ніяких дій. Його використовують там, де за синтаксисом повинен бути оператор, але при цьому потрібно, щоб не виконувалось ніяких дій.

Складений оператор

Складений оператор в мові Java – це кілька операторів, розміщених між символами фігурних дужок « { » та « } ». Складений оператор використовують в тих випадках, коли за синтаксисом С у даному місці може знаходитись лише один оператор, а за алгоритмом програми необхідно виконати кілька операторів. Складений оператор трактується компілятором Java як один оператор.

Умовний оператор

Умовний оператор використовують тоді, коли, в залежності від якоїсь умови, необхідно виконати один з двох операторів. Умовний оператор має наступний синтаксис:

```
if(умова)
    оператор1;
 else
    оператор2;
```

Якщо умова виконується (є істинною), то виконується оператор з розділу `if`. Якщо ж умова не виконується, то виконується оператор з розділу `else`. На відміну від C++ умова повинна повертати результат тільки типу `boolean` (числовий результат не допускається). Допустимою є також скорочена форма умовного оператора:

```
if(умова) оператор;
```

Таку форму використовують тоді, коли якась дія повинна бути виконана лише при істинності умови, а в протилежному випадку ніякі дії не повинні виконуватись. Для прикладу розглянемо програму, яка читає з клавіатури число і відображає повідомлення чи є число позитивним чи негативним:

```
import Java.util.Scanner;
public class if_prob
{
    public static void main(String[] args)
    {
        int choice=0;
        //створюємо вхідний потік, пов'язаний
        //з клавіатурою
        Scanner sc = new Scanner(System.in);
        System.out.print("Input integer number:");
        //скорочена форма оператора if
```

```
        //тут ми перевіряємо наявність у
        //вхідному потоці цілого числа
        //якщо число є зчитуємо його
        if(sc.hasNextInt()) {choice=sc.nextInt();}
        System.out.println();
        //повна форма оператора if
        //перевіряємо число на від'ємність та
        //виводимо відповідне повідомлення
        if(choice<0)
            {System.out.print("choice is negative");}
        else
            {System.out.print("choice is positive");}
        //закриваємо вхідний потік
        sc.close();
}
```

Оператор вибору

Оператор вибору використовують тоді, коли треба зробити вибір більш ніж з двох альтернативних варіантів. Оператор вибору має синтаксис:

```
switch(змінна)
{
    case альтернатива_1:
    оператори;
    break;
    case альтернатива_2:
    оператори;
    break;
…
```

```
        case альтернатива_n:
        оператори;
        break;
    default:
        оператори;
    }
```

Алгоритм виконання оператора вибору наступний: значення змінної, яка подана в дужках після ключового слова switch, порівнюється з альтернативними значеннями, записаними після ключових слів `case`. Якщо знайдено значення, яке співпадає, то виконуються оператори із відповідного розділу `case`. Якщо таких значень не виявлено, то виконуються оператори з розділу `default`. Розділ `default` може бути відсутній. Тоді, при відсутності значень, які співпадають, керування передається на оператор, який знаходиться безпосередньо після оператора вибору. До версії JDK 7 керуюча змінна могла бути типу `byte`, `short`, `int`, `char` або перераховного. Починаючи з версії JDK 7 допустимим є також тип `String`.
Приклад:

```
import Java.util.Scanner;
public class case_prob
{
    public static void main(String[] args)
    {
        int choice=0;
        //створюємо вхідний потік
        Scanner sc = new Scanner(System.in);
        System.out.print("Input number from 1 to 2:");
        if(sc.hasNextInt()) {choice=sc.nextInt();}
        System.out.println();
```

```
        //оператор вибору перевіряє введене значення
        //та відображає відповідне повідомлення
        switch(choice)
        {
            case 1:
                System.out.println("choice=1");
                break;
            case 2:
                System.out.println("choice=2");
                break;
            default:
                System.out.println("incorrect input");
                break;
        }
    }
}
```

Цикл з передумовою

Оператор циклу з передумовою в Java має синтаксис:

```
while(умова)
    оператор;
```

Алгоритм виконання оператора циклу з передумовою наступний:

1. Перевіряється умова.

2. Якщо умова не виконується, то цикл завершується. Якщо ж умова виконується (результат операції не дорівнює 0), то виконується оператор і відбувається перехід до п.1.

Якщо умова не виконується з самого початку, то цикл не виконується жодного разу. Для прикладу розглянемо програму, яка розраховує факторіал числа, введеного з клавіатури:

```
import Java.util.Scanner;
public class wile_prob
{
    public static void main(String[] args)
    {
        int num=0, factorial=1;
        //створюємо вхідний потік, пов'язаний
        //з клавіатурою
        Scanner sc = new Scanner(System.in);
        System.out.print("Input integer number:");
        //перевіряємо наявність у вхідному потоці
        //цілого числа, якщо число є зчитуємо його
        if(sc.hasNextInt()) {num=sc.nextInt();}
        System.out.println();

        //розраховуємо факторіал
        //якщо num менше нуля чи нуль
        //цикл не виконується
        while(num>0)
        {
            factorial*=num;
            num--;
        }
        //відображаємо отримане значення факторіалу
        //або повідомлення про некоректне число
        //у разі, коли воно від’ємне
        if(num<0)
        {
            System.out.print("incorrect number:"+num);
        }
        else
```

```
        {
            System.out.print("factorial="+factorial);
        }
    }
}
```

Цикл з післяумовою

Оператор циклу з післяумовою має синтаксис:

```
do
   оператор;
while (умова);
```

Алгоритм виконання оператора циклу з післяумовою наступний:

1. Виконується оператор.
2. Перевіряється умова. Якщо умова не виконується, то цикл завершується. Якщо ж умова виконується, то відбувається перехід до п.1.

На відміну від циклу з передумовою даний цикл завжди виконується хоча б один раз. В наведеній нижче програмі продемонстровано використання циклу з післяумовою для обчислення факторіалу числа введеного з клавіатури.

```
import Java.util.Scanner;
public class do_while_prob
{
    public static void main(String[] args)
    {
        int num=0, factorial=1;
        //створюємо вхідний потік, пов'язаний
        //з клавіатурою
        Scanner sc = new Scanner(System.in);
        System.out.print("Input integer number:");
```

```
        //перевіряємо наявність у вхідному потоці
        //цілого числа, якщо число є зчитуємо його
        if(sc.hasNextInt()) {num=sc.nextInt();}
        System.out.println();
        //перевірка значення введеного числа
        if(num<=0)
        {
            if(num<0)
            {
                //повідомлення про некоректне число
                System.out.println("error num="+num);
            }
            else
            {
            //відображаємо значення факторіалу нуля
            System.out.println("factorial="+factorial);
            }
        }
        else
        {
            //розраховуємо факторіал
            do
            {
                factorial*=num;
                num--;
            }while(num>0);
            System.out.println("factorial="+factorial);
        }
        System.out.println("End of program");
    }
}
```

Оператор покрокового циклу

Оператор покрокового циклу має синтаксис:

```
for(початковий вираз; умовний вираз; вираз приросту)
   оператор;
```

Оператор покрокового циклу виконується наступним чином:

1. Розраховується початковий вираз.

2. Перевіряється умовний вираз. Якщо результат умовного виразу дорівнює 0 (умова не виконується), то закінчуємо виконання циклу, якщо ж результат умовного виразу не дорівнює 0 (умова виконується), то виконуємо оператор.

3. Виконується вираз приросту і здійснюється перехід до п. 2.

Якщо умовний вираз з самого початку дорівнює нулю, то цикл не виконується жодного разу.

Як правило початковий вираз – це початкове значення змінної, яка використовується в циклі, умовний вираз – кінцеве значення змінної, а вираз приросту – це вираз, за яким обчислюється наступне значення змінної. Для прикладу розглянемо програму, що розраховує факторіал числа, введеного з клавіатури:

```
import Java.util.Scanner;

public class for_prob
{
    public static void main(String[] args)
    {
        int num=0, factorial=1,i;
        Scanner sc = new Scanner(System.in);
        System.out.print("Input integer number:");
        if(sc.hasNextInt()) {num=sc.nextInt();}
```

```
        System.out.println();
        if(num<0)
        {
            System.out.println("error num="+num);
        }
        else
        {
            for(i=num;i>0;i--){factorial*=i;}
            System.out.println("factorial="+factorial);
        }
        System.out.println("End of program");
    }
}
```

Іноді виникає необхідність виконати певні дії для кожного елементу деякої послідовності, наприклад масиву. Для цього випадку існує альтернативна форма даного оператора:

```
for(тип змінна: оброблювана послідовність)
   оператор;
```

Розглянемо приклад:

```
public class for_each_prob
{
    public static void main(String[] args)
    {
        int nums[]={3,2,3,4}, sum=0;
        for(int x:nums)
        {
            sum+=x;
```

```
        }
        System.out.println("sum="+sum);
    }
}
```

Оператор переривання цикла

Оператор переривання цикла використовується для миттєвого припинення процесу виконання циклу або оператора вибору. В мові Java оператор переривання існує в двох формах. Перша форма не відрізняється від аналогічного оператора мов C та C++ і має синтаксис:

```
break;
```

Дія цього оператора також цілком аналогічна відповідному оператору C та C++. Він просто перериває дію цикла або оператора вибору, у якому знаходиться:

```
public class break1_prob
{
    public static void main(String[] args)
    {
        int i,j,k;
        for(i=1;i<3;i++)
        {
            System.out.println("i="+i);
            for(j=1;j<3;j++)
            {
                System.out.println("j="+j);
                for(k=1;k<3;k++)
                {
                    System.out.println("k="+k);
```

```
                    System.out.println("break");
                    break;
                }
            }
        }
    }
}
```

Результат виконання програми наведено нижче.

```
i=1
j=1
k=1
break
j=2
k=1
break
i=2
j=1
k=1
break
j=2
k=1
break
```

З відображених повідомлень видно, що оператор `break` достроково завершує внутрішній цикл і ніяк не впливає на два зовнішніх цикли.

А що робити якщо треба перервати не лише самий внутрішній цикл, але і другий або всі три цикли одразу? Для цього існує друга форма оператора розриву:

```
Break мітка;
```

Мітка вказує на блок коду, у який потрібно передати виконання програми. Для пояснення сказаного розглянемо три приклади, які відрізняються лише положенням мітки.

```
public class break1_prob
{

    public static void main(String[] args)
    {
        int i,j,k;
        for(i=1;i<3;i++)
        {
            System.out.println("i="+i);
            for(j=1;j<3;j++)
            point:{
                System.out.println("j="+j);
                for(k=1;k<3;k++)
                {
                    System.out.println("k="+k);
                    System.out.println("break");
                    break point;
                }
            }
        }
    }
}

i=1
j=1
k=1
break
```

```
j=2
k=1
break
i=2
j=1
k=1
break
j=2
k=1
break
```

Результат дії цієї програми нічим не відрізняється від попередньої. Переривається лише самий внутрішній цикл. Виконання програми передається у блок коду, що охоплює самий внутрішній цикл, тобто у цикл з індексом `j`, а точніше на оператор, що слідує одразу за самим внутрішнім циклом.

```
public class break1_prob
{
    public static void main(String[] args)
    {
        int i,j,k;
        for(i=1;i<3;i++)
        point:{
            System.out.println("i="+i);
            for(j=1;j<3;j++)
            {
                System.out.println("j="+j);
                for(k=1;k<3;k++)
                {
                    System.out.println("k="+k);
```

```
                    System.out.println("break");
                    break point;
                }
            }
        }
    }
}

i=1
j=1
k=1
break
i=2
j=1
k=1
break
```

В даному випадку переривається і середній цикл з індексом j. Управління передається у зовнішній по відношенню до нього блок на оператор, що знаходиться безпосередньо після середнього циклу.

```
public class break1_prob
{
    public static void main(String[] args)
    {
        int i,j,k;
        point:for(i=1;i<3;i++)
        {
            System.out.println("i="+i);
            for(j=1;j<3;j++)
            {
                System.out.println("j="+j);
                for(k=1;k<3;k++)
                {
                    System.out.println("k="+k);
```

```
                        System.out.println("break");
                        break point;
                    }
                }
            }
        }
}

i=1
j=1
k=1
break
```

А в цьому випадку перериваються всі три цикли. Управління передається оператору зовнішнього по відношенню до першого циклу (з індексом `i`), який знаходиться безпосередньо після циклів.

Оператор переходу до наступної ітерації цикла

Оператор продовження цикла може використовуватись лише в циклах. Даний оператор також існує в двох формах. Перша форма має синтаксис:

```
continue;
```

У цій формі оператор передає керування на початок наступної ітерації цикла, ігноруючи всі оператори циклу, які знаходяться після нього.

Друга форма оператора має синтаксис:

```
continue мітка;
```

Перша форма нічим не відрізняється від аналогічного оператора мов C та C++. Що ж до другої форми то характер відмінностей такий же як і в другій формі оператора переривання цикла. Для того,щоб розібратися з цими відмінностями читачеві рекомендується самостійно виконати чотири попередні програми замінивши в них оператор `break` на оператор `continue`.

Оператор виходу з поточного методу

Для завершення виконання поточного методу використовують оператор `return`:

```
return результат;
```

де результат – це змінна, константа або вираз. Тип результату повинен узгоджуватись з типом результату, який задекларовано при визначенні методу.

Таких операторів може бути у методі кілька. Як тільки управління передається на оператор return, виконання методу завершується, незалежно від того досягнуто кінець функції чи ні. Управління передається оператору, який знаходиться безпосередньо за оператором виклику даного методу.

Оператор `return` може бути відсутній. В такому випадку метод виконується до кінця, після чого управління передається оператору, який знаходиться безпосередньо за оператором виклику методу. При цьому значення результата виконання функції невизначено і його не можна використовувати. Дія оператора `return` у мові програмування Java нічим не відрізняється від дії аналогічного оператора мов С та С++.

Типи даних в Java

Прості типи даних

У мові програмування Java вісім простих типів даних. Серед них чотири цілих типи (таблиця 8), два типи даних з плаваючою крапкою (таблиця 9), символьний тип (`char`) та логічний тип (`boolean`) [1-5].

Таблиця 8. Цілі типи даних

Тип	**Розмір, біт**	**Опис**	**Клас-оболонка**
`byte`	8	Цілі числа у діапазоні від -128 до 127	`Byte`
`short`	16	Цілі числа у діапазоні від -32768 до 32767	`Short`
`int`	32	Цілі числа у діапазоні від - 2147483648 до 2147483647	`Integer`
`long`	64	Цілі числа у діапазоні від -9223372036854775808 до 9223372036854775807	`Long`

Таблиця 9. Типи даних з плаваючою крапкою

Тип	**Розмір, біт**	**Діапазон значень модуля**	**точність (число десятко-вих цифр)**	**Клас-оболонка**
`float`	32	$3.4*10^{-38}..3.4*10^{38}$	7	`Float`
`double`	64	$1.7*10^{-308}..1.7*10^{308}$	15	`Double`

Тип `char` – це символьний тип для подання символьних значень (літер). Він представляє собою 16-бітний код Unicode. Діапазон значень коду лежить в межах 0..65536, що дозволяє закодувати всі відомі символи національних мов світу. Кодова таблиця побудована таким чином, що перші 128 позицій займають символи з таблиці кодів ASCII, що робить їх сумісними. Клас-оболонка має назву Character.

Тип `boolean` – це логічний (булевський) тип. Він займає в пам'яті один байт і може приймати лише два значення: true та false. Клас-оболонка має назву Boolean.

Для виконання математичних операцій з числовими даними в Java існує бібліотека математичних функцій, яка реалізована у вигляді класу `Math`.

Підключити бібліотеку можна за допомогою ключового слова `import`. Для підключення бібліотеки математичних функцій `Math` достатньо добавити наступну команду:

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

Бібліотека математичних функцій містить ряд корисних функцій:

- `abs(double value)` – повертає абсолютне значення аргументу `value`;
- `acos(double value)` – повертає арккосинус `value`. Параметр `value` повинен мати значення від -1 до 1;
- `asin(double value)` – повертає арксинус `value`. Параметр `value` повинен мати значення від -1 до 1;
- `atan(double value)` – повертає арктангенс `value`;
- `cbrt(double value)` – повертає кубічний корінь числа `value`;
- `ceil(double value)` – повертає найменше ціле число з плаваючою крапкою, яке не менше за `value`;
- `cos(double d)` – повертає косинус кута `d`;
- `cosh(double d)` – повертає гіперболічний косинус кута `d`;
- `exp(double d)` – повертає основу натурального логарифму, зведену до ступеня `d`;

- `floor(double d)` – повертає найбільше ціле число, яке не більше `d`;
- `floorDiv(int a, int b)` – повертає цілий результат ділення `a` на `b`;
- `log(double a)` – повертає натуральний логарифм числа `a`;
- `log1p(double d)` – повертає натуральний логарифм числа `(d + 1)`;
- `log10(double d)`: повертає десятковий логарифм числа d;
- `max(double a, double b)` – повертає максимальне число з чисел `a` та `b`;
- `min(double a, double b)` – повертає мінімальне число з чисел `a` та `b`;
- `pow(double a, double b)` – повертає число `a`, зведене до ступеня `b`;
- `random()` – повертає випадкове число від 0.0 до 1.0;
- `rint(double value)` – повертає число `double`, яке представляє найближче до `value` ціле число;
- `round(double d)` – повертає число `d`, округлене до найближчого цілого числа;
- `scalb(double value, int factor)` – повертає добуток числа `value` на 2 піднесене до ступеня `factor`;
- `signum(double value)` – повертає число 1, якщо число `value` позитивне, -1, якщо значення `value` негативне. Якщо значення дорівнює 0, то повертає 0;
- `sin(double value)` – повертає синус кута `value`;
- `sinh(double value)` – повертає гіперболічний синус кута `value`;
- `sqrt(double value)` – повертає квадратний корінь числа `value`;
- `tan(double value)` – повертає тангенс кута `value`;

- `tanh(double value)` – повертає гіперболічний тангенс кута `value`;
- `toDegrees(double value)` – переводить радіани в градуси;
- `toRadians(double value)` – переводить градуси в радіани.

Розглянемо приклад:

```
package math_prob;
import Java.util.Scanner;
import static Java.lang.Math.*;

public class math_prob {

public static void main(String[] args) {
// TODO Auto-generated method stub
    Scanner in = new Scanner(System.in);
    System.out.print("Введіть радіус кола (ціле
       число): ");
    int radius = in.nextInt();
    double area = PI * pow(radius, 2);
    System.out.printf("Площа кола з радіусом %d
         дорівнює %f \n", radius, area);
}
}
```

Посилальні типи

На відміну від примітивних типів, які дозволяють зберігати значення прямо всередині змінних, посилальні типи зберігають посилання на об'єкти. Тобто. десь у пам'яті існує якийсь об'єкт, а в змінній-посиланні зберігається адреса цього об'єкта в пам'яті (посилання). Тобто це “візитка” з адресою, маючи яку можна знайти даний об'єкт у спільній пам'яті та виконувати з ним

деякі маніпуляції. Фактично значення прямо всередині змінних зберігають лише примітивні типи, а решта типів зберігають лише посилання об'єкт. Посилання на будь-який об'єкт у Java є посилальною змінною.

Наприклад, змінні `int` та `double` — примітивні. Їх значення зберігаються безпосередньо всередині змінних цих типів. В той же час змінна `String` зберігає адресу (посилання) об'єкта типу `String` у пам'яті.

При привласненні значення змінній примітивного типу це значення копіюється у цю змінну (дублюється). При присвоюванні ж посилальної змінної копіюється лише адресу об'єкта, сам об'єкт у змінну не копіюється.

Такий підхід дозволяє уникнути копіювання великих обсягів пам'яті. Якщо кудись потрібно передати дуже великий об'єкт, то передають посилання на цей об'єкт і все. Посилання займає набагато менше місця.

Розмір всіх змінних-посилань (незалежно від типу) однаковий і становить 4 байти для 32-розрядної машини Java та 8 байт 64-розрядної Java-машини.

Посилальним змінним можна присвоювати лише значення посилальних типів. Їм не можна присвоювати довільні значення.

Наприклад:

```
//так можна
String example1 = "Приклад 1";
String s = example1;
//а так не можна
s++;
//і так також не можна
s=0x1500;
```

Якщо посилальна змінна створена, але їй ще не присвоєно значення, то вона містить значення `null`.

Об'єкти в Java

Мова програмування Java з самого початку була розроблена як об'єктно орієнтована мова програмування, тому як і в будь-якій об'єктно орієнтованій мові програмування основними принципами, на яких все базується є інкапсуляція, успадкування і поліморфізм [6]. Нагадаємо значення цих понять.

Інкапсуляція – це поєднання в одній абстрактній інформаційній структурі даних (полів) та алгоритмів їх обробки (методів).

Успадкування – це засіб побудови нових (***породжених***) абстрактних інформаційних структур на основі уже існуючих (***базових***). При цьому породжені структури утворюються шляхом додавання нових полів та методів або шляхом модифікації існуючих і можуть успадковувати деякі або всі властивості базових структур. Це дозволяє повторно використовувати існуючий програмний код і, за рахунок цього, прискорити розробку програмного забезпечення. Породжені структури називають ***потомками,*** а базові – ***предками***.

Поліморфізм – властивість інформаційної структури, яка полягає у можливості зміни її поведінки у залежності від того з якими іншими структурами вона взаємодіє.

Основним засобом реалізації цих принципів є клас. Клас в Java – це основна структурна одиниця програми. Сама ж програма в цій мові являє собою набір взаємодіючих класів. Для зручності пов'язані класи групують у пакет (`package`). Всі бібліотечні класи Java розміщені у пакетах, що дозволяє більш повно реалізувати принцип інкапсуляції.

Для створення класів використовують ключове слово class. Опис класу має такий синтаксис [1-6]:

```
class ім'я_класу
{
    //опис змінних
    тип змінна;
```

```
    тип змінна;

    //опис методів
    тип метод(аргументи);
    тип метод(аргументи);

}
```

Методи і змінні називають членами класу. Члени-змінні класу називають змінними екземпляра або полями екземпляра, члени-методи – методами екземпляру.

Однак треба пам’ятати, що сам по собі клас – це лише логічна абстракція (опис структури даних). Для того, щоб в програмі використати клас необхідно створити конкретний екземпляр класу – об’єкт. Створити об’єкт можна за допомогою наступного синтаксису:

```
ім’я_класу ім’я_об’єкту;
```

Наведемо приклад:

```
//описуємо клас vector і поміщуємо його у пакет vector
package vector_prob;
import static Java.lang.Math.*;

public class vector_prob {
    //змінні класу
    double x;
    double y;
    double z;
```

```
    //методи класу
    //параметризований метод
    void set_xyz(double xx, double yy, double zz)
    {
        x=xx;
        y=yy;
        z=zz;
    }
    //непараметризовані методи
    double get_x(){return x;};
    double get_y(){return y;};
    double get_z(){return z;};
    double get_length(){return sqrt(x*x+y*y+z*z);};
}
//описуємо клас use_vector_prob
//для зручності розміщуємо його у тому ж пакеті vector
package vector_prob;

public class use_vector_prob {

    public static void main(String[] args) {
    //створюємо об'єкт класу vector_prob
    vector_prob a=new vector_prob();
    a.set_xyz(0, 3, 4);
    System.out.println("a has length "+a.get_length());
    }
}
```

В даному прикладі розроблено два класи – клас радіус вектор (vector_prob), який описує структуру радіус вектора і клас

use_vector_prob, який створює об'єкт класу vector_prob і використовує його.

В наведеному прикладі відсутній конструктор, тому при роботі програми використовується конструктор передбачений системою (як і в мові C++), а значення координат вектора доводиться встановлювати вручну за допомогою функції set_xyz(double xx, double yy, double zz). Такий підхід зазвичай не використовується, оскільки це може призводити до помилок, які важко виявити. Звичайною практикою є розробка класів з власними, визначеними програмістом конструкторами. Конструктор – це функція-метод класу яка має таке ж ім'я, як у класу. Цей метод призначений для створення та ініціалізації об'єктів класу, виділення пам'яті під об'єкти й присвоювання початкових значень полям об'єкту.

Зазвичай розроблюється кілька конструкторів: конструктор без параметрів (конструктор за замовчуванням), який ініціалізує змінні об'єкту значеннями за замовчуванням, та параметризований конструктор, який дозволяє під час створення об'єкту ініціалізувати його змінні значеннями відмінними від значень за замовчуванням [6].

Слід зазначити, що, на відміну від C++, відсутні деструктори, тобто програміст не має змоги знищувати об'єкти, які більше не потрібні[6]. Це робить так званий «збирач сміття», який є частиною мови Java. Це зроблено для того, щоб запобігти виникненню помилок розподілу пам'яті, які можуть призвести до серйозних наслідків. Якщо необхідно, щоб перед знищенням об'єкту було виконано якісь дії, то для цього створюють метод finalize(), в якому прописують всі необхідні дії. Цей метод викликається системою безпосередньо перед знищенням об'єкту. Метод finalize() завжди визначається як захищений метод (protected). Це потрібно для того, щоб метод неможливо було визвати поза межами класу (дію ключового слова protected буде розглянуто пізніше). При використанні методу finalize() необхідно пам'ятати, що «збирач сміття» може бути

запущений системою в час, коли об’єкт знаходиться поза зоною його дії. В цьому випадку при знищенні об’єкту метод `finalize()` не буде виконано. Тому даний метод абсолютно не придатний для коректного вивільнення ресурсів програми, а тим більше для нормального її завершення [6].

Для прикладу додамо до попередньої програми два конструктори – конструктор за замовчуванням та параметризований конструктор.

```
package vector_1;
import static Java.lang.Math.*;
public class vector_1 {
    double x;
    double y;
    double z;
    void set_xyz(double xx, double yy, double zz)
    {
        x=xx;
        y=yy;
        z=zz;
    }
    //конструктор за замовчуванням
    vector_1()
    {
        x=0;
        y=0;
        z=0;
    }
    //параметризований конструктор
    vector_1(double xx, double yy, double zz)
    {
        x=xx;
        y=yy;
        z=zz;
```

```
    }

    double get_x(){return x;};
    double get_y(){return y;};
    double get_z(){return z;};
    double get_length(){return  sqrt(x*x+y*y+z*z);};

}

package vector_1;
public class use_vector_1 {

    public static void main(String[] args) {

        vector_1 a=new vector_1();
        vector_1 b=new vector_1(0.0,3.0,4.0);
        System.out.println("vectot  a  has  length  "
                                +a.get_length());
        System.out.println("vectot  b  has  length  "
                                +b.get_length());
    }
}
```

В даному прикладі додано два конструктори. Конструктор за замовчуванням створює вектор з нульовими координатами, а параметризований конструктор створює вектор із заданими координатами.

Результат виконання програми матиме вигляд:

```
vectot a has length 0.0
vectot b has length 5.0
```

Модифікатори доступу

При описі класу використовуються модифікатори доступу, які визначають режим доступу до полів та методів класу. У мові програмування Java використовують наступні модифікатори доступу [1-10]:

- `public` – всі поля та методи описані як public доступні об’єктам інших класів;

- `private` – всі поля та методи описані як `private` доступні тільки методам свого власного класа;
- `protected` – всі поля та методи описані protected доступні методам свого власного класа та всіх його нащадків.

Всі поля і методи, доступ до яких не описаний жодним з модифікаторів доступні всім класам пакету у якому даний клас розміщений.

Бажано, за можливості, використовувати доступ до полів класу лише через його власні методи.

Якщо передбачається використовувати клас у інших пакетах, то конструктор класу потрібно об'явити як `public`.

Приклад:

```
package access_example_package1;
public class circle_class {
//поля x, y, z – доступні лише методам класу
//circle_class
    private float x;
    private float y;
    private float r;
    public circle_class(float xx, float yy, float rr)
    {
        x=xx;
        y=yy;
        r=rr;
    }
    public float circle_square() {
        float s= (float) 3.14*r*r;
        return s;
        }
```

```
    float circle_length(){
        float l=(float) 3.14*2*r;
        return l;
    }
    public float get_circle_length(){
        float g_l=circle_length();
        return g_l;
    }
}

package access_example_package2;
import access_example_package1.*;
public class example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        circle_class cc = new circle_class(1,1,1);
        float S = cc.circle_square();
        System.out.println("S = " + S + " sm*sm");
        /*Так не можна бо зазамовчуванням
         * cc.circle_length() доступна лише
         * у access_example_package1
         * float L =  cc.circle_length()
         */
        //потрібно так:
        float L =  cc.get_circle_length();
        System.out.println("L = " + L + " sm");
    }
}
```

Результат виконання:

```
S = 3.14 sm*sm
L = 6.28 sm
```

Реалізація успадкування у мові програмування Java

Успадкування - це один з базових принципів ООП. Оскільки Java підтримує концепцію ООП, то цей механізм також реалізований у Java. При використанні успадкування новий клас, який успадковує властивості базового (батьківського) класу, має всі ті властивості, які має батьківський клас. Для реалізації успадкування у мові програмування Java використовують операнд `extends`, після якого вказується ім'я базового класу. Тим самим відкривається доступ до всіх полів та методів базового класу [1-10].

Використання наслідування дозволяє створити загальний "Java class", який визначає характеристики, загальні для набору зв'язаних елементів, а потім на його основі створити додаткові класи, які успадкують загальні характеристики, до яких буде додано додаткові унікальні для них характеристики.

Наслідуваний клас Java називають суперкласом `super`. Наслідуючий клас називають підкласом. Тобто, підклас – це спеціалізована версія суперкласу, яка успадковує всі властивості суперкласу і додає свої власні унікальні елементи.

Приклад:

```
//опис батьківського класу
package inheritance_prob;

public class human{
    private int age;
    private String name;
    private String surname;
```

```
    public human (int a, String n, String s){
        age = a;
        name=n;
        surname = s;
    }

    public void get_data(){
        System.out.println("name - " + name);
        System.out.println("surname - " + surname);
        System.out.println("age - " + age);
    }

}

//опис породженого класу
package inheritance_prob;

public class professional extends human
{
    private String profession;
    public professional(int aa, String nn, String
         ss, String p){
        super ( aa, nn, ss);
        profession=p;
    }

    public void get_data(){
        super.get_data();
        System.out.println("profession   -   "   +
              profession);
```

```
        }
    }

    //головна процедура
    package inheritance_prob;

    public class inheritance_prob {

        public static void main(String[] args) {
            // TODO Auto-generated method stub
            System.out.println("Super class:");
            human hh = new human(20,
                "Serhii","Petrenko");
            hh.get_data();
            System.out.println("subclass:");
            professional pp = new professional(25,
                "Dmytro", "Suhak", "student");
            pp.get_data();
        }
    }
```

Результат виконання:

```
Super class:
name - Serhii
surname - Petrenko
age - 20
subclass:
name - Dmytro
surname - Suhak
```

```
age - 25
profession – student
```

Часто виникає потреба заборонити успадкування якогось класу. Для реалізації такої можливості використовується модифікатор `final` [6-9]. Клас, позначений як `final`, не піддається успадкування і всі його методи опосередковано набувають властивість `final`. Можна також використовувати модифікатор `final` не до всього класу, а до окремих методів. Позначення методу класу модифікатором `final` означає, що цей метод не може бути змінений (перевизначений) у жодному похідному класі.

Приклад:

```
package final_example_2;

public class SuperClass{
    public final void printReport(){
        System.out.println("Report");
    }
}

package final_example_2;

class SubClass extends SuperClass{
    public void printReport(){
//Помилка компіляції. Не можна перевизначити
//метод, описаний як final
        System.out.println("MyReport");
    }
}
```

```
package final_example_2;

public class final_example_2_main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SubClass scl = new SubClass();
        scl.printReport();
    }
}
```

Виникає питання, а навіщо такі можливості можуть знадобитись?

Застосування модифікатора `final` в оголошеннях класів і методів здатне підвищити рівень безпеки коду. Якщо клас або метод класу має модифікатор `final`, ніхто не може змінити його функціональність і порушити його роботу. Наприклад, доцільно застосувати модифікатор `final` в оголошенні методу, який передбачає перевірку пароля, щоб гарантувати, що зловмиснику не вдасться змінити вихідну реалізацію цього методу, "підсунувши" програмі його перевизначену версію [6].

Однак треба розуміти, що вживання модифікатора `final` в оголошенні методу чи класу накладає серйозні обмеження на можливість подальшого використання та розвитку коду. Тому використання модифікатора `final` дуже виваженого підходу з урахуванням всіх переваг і обмежень.

Ще один важливий момент полягає в тому, щоб не плутати перевизначення методу з перенавантаженням методу. При перевизначенні методу кількість і типи аргументів такі самі, як і в методі, який перевизначається. При цьому версія батьківського методу в успадкованому класі замінюється новою версією. При перевантаженні кількість аргументів та/або їх тип відрізняється від кількості аргументів та/або їх типів методу базового класу. У даному випадку нова версія методу не заміняє

батьківський метод, а існує одночасно з ним. Який метод викликати компілятор «вирішує» виходячи з типів та кількості аргументів. Перевантаження класів та методів є одним з базових способів реалізації поліморфізму.

Реалізація поліморфізму у мові програмування Java

Поліморфізм поряд з успадкуванням та інкапсуляцією є одним з базових понять об'єктно-орієнтованого програмування. Слово поліморфізм грецького походження і означає "що має багато форм". Класичним прикладом поліморфізму є реалізація арифметичних операцій для різних числових типів даних. Кожен тип даних має свою власну структуру, свій розмір і т.д. Однак для користувача все це сховано. Так всі арифметичні операції з точки зору програміста виконуються за допомогою типових операторів (+, - , *, /) однаковим способом для різних типів даних, а сам внутрішній алгоритм операції обирається мовою програмування в залежності від конкретного типу даних. Іншим прикладом поліморфізму можна вважати наявність кількох конструкторів у класі, які мають однакову назву, але різні алгоритми створення об’єкту, які залежать від кількості та типів аргументів. Ще одним прикладом може бути вивід написів на екран. Один метод (наприклад `System.out.println(Object x)`) виводить на екран значення різних типів, перетворюючи їх у текстовий формат. Такі методи називають перенавантаженими [6-9].

Приклад:

```
//опис класу multi_vector
package polimorf_example;
import Java.util.Scanner;

public class multi_vector {
    private int n;
    private float v[];
```

```
private Scanner sc;
public multi_vector(){
    n=3;
    v = new float [3];
    sc = new Scanner(System.in);
}
public multi_vector(int nn){
    n=nn;
    v = new float [n];
    sc = new Scanner(System.in);
}
private Scanner Scan() {
    // TODO Auto-generated method stub
    Scanner ssc= new Scanner(System.in);
    return ssc;
}

public void set_multi_vector(){
    for (int i=0;i<n;i++){
        System.out.println("Input v["+i+"]:");
        sc = Scan();
        if(!sc.hasNextFloat())
            while(! sc.hasNextFloat()){
                System.out.println("Incorrect
                    input");
                sc = Scan();
            }
            v[i]= sc.nextFloat();

    }
```

```
    }

    public void show_multivector(){
        for (int i=0;i<n;i++){
            System.out.println("v["+i+"] = " + v[i]);
            }
    }
}

//головна процедура
package polimorf_example;

public class polimorf_example {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        multi_vector vv1 = new multi_vector();
        vv1.set_multi_vector();
        vv1.show_multivector();
        multi_vector vv2 = new multi_vector(2);
        vv2.set_multi_vector();
        vv2.show_multivector();
    }
}
```

Результат виконання:

```
Input v[0]:
9
Input v[1]:
3
```

```
Input v[2]:
w
Incorrect input
1.0
Incorrect input
1,0
v[0] = 9.0
v[1] = 3.0
v[2] = 1.0
Input v[0]:
1
Input v[1]:
11
v[0] = 1.0
v[1] = 11.0
```

Як було сказано вище, слід відрізняти перевизначення (заміщення) від перенавантаження. Для демонстрації відмінності між перевизначенням і перенавантаженням наведемо приклад. У попередньому підрозділі було показано, що неможливо перевизначити метод з модифікатором `final`. Однак це не завадить перенавантажити такий метод [6-9].

Приклад:

```
package final_example;

public class obj1 {
    protected int a;
    protected int b;
    public final void set_data(int aa, int bb)
    {
```

```
        a=aa;
        b=bb;
    }
}

package final_example;
public class obj2 extends obj1{
    private int c;
    //перенавантажимо метод set_data,
    //додавши ще один аргумент
    public void set_data(int aa, int bb, int cc)
    {
        super.set_data(aa, bb);
        c=cc;
    }
    public void print_data (){
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" c = " + c);
    }
}

package final_example;
public class final_example_main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        obj2 o2 = new obj2();
        o2.set_data(1, 2, 3);
        o2.print_data();
    }
```

```
}
```

Результат виконання:

```
a = 1
b = 2
c = 3
```

Змінювані та незмінювані об'єкти

Всі об'єкти в мові програмування Java можна розділити на змінювані (`mutable`) та незмінювані (`immutable`).

Змінювані (`mutable`) об'єкти – це такі об'єкти, значення яких може бути змінене після ініціалізації. Коли ми вносимо зміни в існуючі змінювані об'єкти, то не створюється новий об'єкт, а просто змінюється значення уже існуючого об'єкта. Такі об'єкти мають спеціальні методи для зміни їх значення (`set()`-методи).

Незмінювані (`immutable`) об'єкти – це такі об'єкти, значення яких не може бути змінене після ініціалізації. Такі об'єкти не мають `set()`-методів. Проте вони, як правило мають ряд методів, які дозволяють створити новий об'єкт даного класу із зміненим значенням. До незмінюваних об'єктів відносяться, наприклад, об'єкти класу `String`. Клас `String` буде розглянутий пізніше [7,8].

Приклад

```
//змінюваний клас1 поля даного класу a та b
//описані, як public, отже доступні для зміни
//навіть за відсутності set()-методу
public class mut1 {
    public int a;
    public int b;
}
```

```
//змінюваний клас2 поля даного класу a та b
//описані, як private, однак він має public метод,
// set_a_b, який дозволяє змінити їх значення
public class mut2 {
    private int a;
    private int b;

    public set_a_b(int aa, int bb)
    {
        a=aa;
        b=bb;
    }
}

//незмінюваний клас1 поля даного класу a та b
//описані як private final,
//вони отримують значення при створенні і
//в подальшому не можуть бути змінені
public class immut1 {
    private final int a;
    private final int b;
    public immut1(int aa, int bb)
    {
        a=aa;
        b=bb;
    }
}
```

Абстрактні класи та методи

Абстрагування – це виділення самих істотних характеристик об'єкта, які відрізняють його від всіх інших об'єктів і чітко визначають його з точки зору подальшого розгляду та аналізу.

Абстрактні класи та методи – основні інструменти для реалізації абстракцій. Абстрактні класи це – опис структури для створення нових класів. Абстрактні класи застосовують для визначення загального інтерфейсу чи поведінки, яка може спільно використовуватися одночасно кількома класами нащадками. Абстрактні методи не маю тіла, але вони можуть містити тип, параметри і модифікатори, що повертається. Конкретна реалізація методу відбувається на етапі створення класу нащадка [6-9].

Екземпляр (об’єкт) абстрактного класу не можна створити безпосередньо. Абстрактний клас служить основою для розробки інших класів. При оголошенні такого класу використовують ключове слово `abstract`.

При роботі з абстрактними класами треба враховувати наступні правила і обмеження:

- абстрактний метод не може бути оголошений як `final` або `private`;
- абстрактний метод обов'язково має оголошуватися всередині абстрактного класу;
- клас нащадок абстрактного класу, який розширює абстрактний клас, обов'язково має надавати реалізацію всіх його успадкованих абстрактних методів;
- абстрактні класи можуть мати конструктори, але їх не можна викликати неявним чином безпосередньо із класів нащадків. Конструктор конкретного класу-нащадка може лише явно викликати конструктор відповідного абстрактного класу (використовуючи метод `super()`) або конкретний конструктор суперкласу.

Приклад:

```
package abstract_example;
//абстрактний клас з абстрактним методом
public abstract class point_class {
    protected float x_coord, y_coord;
    public point_class(float x, float y) {
        x_coord = x;
        y_coord = y;
      }
    public abstract void get_coord}

package abstract_example;
//породжений клас
public class line_segment extends point_class{
    private float x2_coord, y2_coord;
    public line_segment(float x1, float y1, float x2,
                        float y2){
        super(x1,y1);
        x2_coord=x2;
        y2_coord=y2;
    }
//метод перекриває абстрактний метод
    public void get_coord(){
        System.out.println("x1 = " + x_coord);
        System.out.println("y1 = " + y_coord);
        System.out.println("x2 = " + x2_coord);
        System.out.println("y2 = " + y2_coord);
    }
}
```

```
package abstract_example;

public class use_abstract_example {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        line_segment l1 = new line_segment(1,2,11,12);
        l1.get_coord();
    }
}
```

Результат виконання:

```
x1 = 1.0
y1 = 2.0
x2 = 11.0
y2 = 12.0
```

Масиви в Java

В мові Java, як і в C та C++ часто використовують масиви. Масив – це набір однотипних елементів. Кожен елемент має свій номер (індекс). Всі елементи масиву впорядковані за своїм індексом. Масиви можуть бути одновимірними та багатовимірними.

На відміну від мов C та C++ в Java використовують лише динамічні масиви. Статичні масиви в мові Java відсутні. Вони створюються за допомогою оператора new. Однак в Java відсутній оператор delete, тому програміст не може самостійно знищувати масиви і будь-які інші динамічні об’єкти. Цю функцію бере на себе система, а саме так званий збірник сміття. Збірник сміття за спеціальним алгоритмом знаходить об’єкти, які вже не використовуються програмою і звільняє пам'ять, яку вони займають [1-10].

Наведемо приклади команд для створення масивів.

```
//створення одновимірного масиву
//з 10-ти елементів типу int
int nums[]=new int[10];
//або так
int[] nums=new int[10];
//ще один спосіб створення одновимірного масиву
//з 10-ти елементів типу int
int num1[]={12,1,6,15,7,0,9,2,21,17};

//створення двовимірного масиву
//розміром 3x4 (12-ть елементів типу int)
int table[][]=new int[3][4];
//або так
int[][] table=new int[3][4];

//а можна і так
int table1[][]={
    {10,11,3,14},
    {15,26,7,5},
    {0,23,6,13}
};
```

До конкретного елементу масиву можна звернутись за його номером (індексом). Нумерація починається з нуля:

```
/* змінна a отримала значення елементу масиву symbol з
індексом 3 */
a=symbol[3];
/* змінна ch отримала значення елементу двовимірного
масиву symbol_matrix з координатами (2,1) */
ch=symbol_matrix[2][1];
```

При зверненні до елементів масиву необхідно пам’ятати, що нумерація елементів по кожному з вимірів починається з нуля.

Крім того необхідно пам’ятати, що в Java масиви реалізовані як об’єкти, тому кожен масив має змінну `length`. В цій змінній міститься розмір (кількість елементів) масиву [6-9]. Так для того щоб визначити кількість елементів вищеописаного масиву `nums`, достатньо прочитати значення змінної `nums.length`. У нашому випадку ця змінна дорівнює 10. Для двовимірних масивів це буде кількість рядків масиву, оскільки складні масиви розглядаються в Java як масиви, які складаються з набору одновимірних масивів. Тому в даному випадку елементом масиву є рядок. Такий підхід дозволяє, наприклад, реалізувати двовимірні масиви з різною кількістю елементів у рядках. Розглянемо приклад:

```
package massives_prob;
public class massives_prob
{
    public static void main(String[] args)
    {
        //різні форми визначення масивів
        int nums[]=new int[10];
        int x=0;
        int table[][]= {
                {1,2,13},
                {5,6},
                {3,7,11}
        };
        //заповнимо масив nums
        for(int i:nums){nums[x]=x*x;x++;}
        //відобразимо кількість елементів у nums
        System.out.println("nums.length="+nums.length);
```

```
        //відобразимо значення 6-го елементу nums
        System.out.println("nums[5]="+nums[5]);
        //відобразимо кількість рядків у table
        System.out.print("table.length="+table.length);
        System.out.println();
        //відобразимо 2-ий елемент 2-го рядка table
        System.out.println("table[1][1]="+table[1][1]);
        //відобразимо довжину 2-го рядка table
        System.out.println("table[1].length="
                                +table[1].length);
        //відобразимо довжину 3-го рядка table
        System.out.println("table[2].length="
                                +table[2].length);
    }
}
```

Результат виконання програми:

```
nums.length=10
nums[5]=25
table.length=3
table[1][1]=6
table[1].length=2
table[2].length=3
```

Рядкові дані в Java

Рядок є послідовністю символів. Для роботи з рядками у мові програмування Java визначено клас `String`, який надає ряд методів для маніпуляції рядками. Фізично об'єкт `String` є посиланням на область пам'яті, у якій розміщені символи.

Для створення нового рядка можна використовувати один із конструкторів класу String, або безпосередньо присвоїти рядок у подвійних лапках [1-4]:

```
package string_prob;

public class string_prob {

public static void main(String[] args) {
    // TODO Auto-generated method stub
    String str1 = "Java";
    String str2 = new String(); // порожній рядок
    String str3 = new String(new char[]
        {'h','e','l','l','o'});
    //3 -початковий індекс, 4 -кількість символів
    String str4 = new String(new char[]{'w', 'e', 'l',
        'c', 'o', 'm', 'e'}, 3, 4);
    System.out.println(str1); // Java
    System.out.println(str2); //
    System.out.println(str3); // hello
    System.out.println(str4); // come

    }
}
```

Працюючи з рядками, важливо розуміти, що об'єкт String є незмінним (immutable). Тобто за будь-яких операцій над рядком, які змінюють цей рядок, фактично буде створюватися новий рядок.

Оскільки рядок розглядається як набір символів, можна застосовувати метод length() для знаходження довжини рядка або довжини набору символів:

```
    String str1 = "Java";
    System.out.println(str1.length()); // 4
```

За допомогою методу `toCharArray()` можна перетворити рядок на масив символів:

```
String str1 = new String(new char[] {'h', 'e', 'l',
'l', 'o'});
char[] helloArray = str1.toCharArray();
```

Рядок може бути порожнім. Для цього йому треба присвоїти порожні лапки або видалити зі стоки всі символи:

```
String s = "";   // порожній рядок
if(s.length() == 0) System.out.println("String is
empty");
```

В цьому випадку довжина рядка, що повертається методом length(), дорівнює 0.

Клас `String` має спеціальний метод, який дозволяє перевірити рядок на порожнечу - `isEmpty()`. Якщо рядок порожній, він повертає true:

```
String s = "";   // порожній рядок
if(s.isEmpty())    System.out.println("String    is
empty")
```

Рядкова змінна може мати значення `null`:

```
String s = null;   // рядок має значення null
if(s == null) System.out.println("String is null");
```

Значення `null` не еквівалентно порожньому рядку. Наприклад, у наступному випадку виникне помилка виконання:

```
// рядкова змінна не посилається на об’єкт
String s = null; if(s.length()==0)
System.out.println("String is empty");// Помилка
```

Оскільки змінна `s` не вказує на конкретний об'єкт типу `String`, то некоректно звертатися до методів об'єкту `String`. Для уникнення подібних помилок, можна попередньо перевіряти рядок на значення `null`:

```
// рядок не вказує на конкретний об’єкт
String s = nullif(s==null || s.length()==0);
System.out.println("String is empty");
```

Клас `String` має ряд методів:

- `concat()`: об’єднує рядки;
- `valueOf()`: перетворює об’єкт у рядковий тип;
- `join()`: об’єднує рядки з врахуванням розділюючого символу;
- `compareTo()`: порівнює два рядки;
- `charAt()`: повертає символ із рядка по його індексу;
- `getChars()`: повертає послідовність символів;
- `equals()`: порівнює рядки з урахуванням регістра;
- `equalsIgnoreCase()`:порівнює рядки без урахування регістра;
- `regionMatches()`: порівнює задані послідовності символів у заданих рядках;
- `indexOf()`: знаходить індекс першого входження послідовності символів у рядок;
- `lastIndexOf()`:знаходить індекс останнього входження послідовності символів у рядок;
- `startsWith()`: визначає чи починається рядок заданою послідовністю символів;

- `endsWith()`:визначає чи закінчується рядок заданою послідовністю символів;
- `replace()`: заміняє у рядку одну послідовністю символів іншою;
- `trim()`: відаляє у рядку початкові і кінцеві пробіли;
- `substring()`: повертає послідовність символів рядка, починаючи з заданого початкового індексу до кінця рядка або до заданого кінцевого індексу;
- `toLowerCase()`: переводить усі символи рядка у нижній регістр;
- `toUpperCase()`:переводить усі символи рядка у верхній регістр;

Класи-оболонки

В мові Java крім примітивних типів є вісім спеціальних класів-оболонок: `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, `Double`, `Boolean`. Класи-оболонки Java є об'єктною формою відповідних примітивних типів Java: `byte`, `short`, `char`, `int`, `long`, `float`, `double`, `boolean`. Класи-оболонки мають свої переваги і недоліки. Основною перевагою класів-оболонок є те, що вони надають широкий спектр методів, які значно розширюють можливості маніпуляції примітивними типами і дозволяють інтегрувати примітивні типи в ієрархію об'єктних типів Java. Так, наприклад, використовуючи ці методи, можна реалізувати функції, приведення одного типу даних до іншого, перетворення об'єктів у рядки і т.д. Основним недоліком класів-оболнок є те, що вони займають значно більше пам’яті ніж примітивні типи даних [8,9].

Розглянемо кожен з класів оболонок детальніше.

В основі всіх класів-оболонок для числових типів лежить абстрактний клас `Number` (є базовим для них).

Клас `Number` має один конструктор – `public Number()`.

Крім того він має низку методів:

- `byte byteValue()` – перетворення значення на тип `byte`;
- `abstract double doubleValue()` – перетворення значення на тип `double`;
- `abstract float floatValue()` – перетворення значення на тип `float`;
- `abstract int intValue()` – перетворення значення на тип `int`;
- `abstract long longValue()` – перетворення значення на тип `long`;
- `short shortValue()` – перетворення значення в тип `short`.

Всі методи класу `Number` доступні у класах-оболонках числових типів. Крім того кожен з них має свої специфічні методи [6-9].

Клас `Integer`

Клас `Integer` – це оболонка простого типу `int`. Об'єкт містить поле типу `int`. Крім того, цей клас забезпечує кілька методів для перетворення `int` в `String` і назад, а також інші константи та методи корисні при роботі з `int` [6-9].

Конструктори класу `Integer`:

- `Integer(int value)` – створення об’єкта `Integer` на основі аргумента `int`;
- `Integer(String s)` – створення об’єкта `Integer` на основі рядкового аргумента.

Поля класу `Integer`:

- `static int MAX_VALUE` - максимальная величина типа `int`;
- `static int MIN_VALUE` - минимальная величина типа `int`;

- `static Class TYPE` - объект класса, представляющий простой тип `int`.

Основні методи класу `Integer`:

- `byte byteValue()` – перетворення значення на тип `byte`;
- `int compareTo(Integer, Integer)` – порівняння двох цілих чисел;
- `int compareTo(Object o)` – порівняння значення з іншим об'єктом;
- `Integer decode(String nm)` – перетворення рядка у `Integer`;
- `double doubleValue()` – перетворення значення у тип `double`;
- `boolean equals(Object obj)` – порівняння з іншим об’єктом;
- `float floatValue()` – перетворення значення у тип `float`;
- `int hashCode()` – отримання hashcode для об’єкта;
- `int intValue()` – перетворення значення у тип `int`;
- `long longValue()` – перетворення значення у тип `long`;
- `int parseInt(String s)` – перетворення текстового значення у тип `int`;
- `int parseInt(String s, int radix)` – перетворення текстового значення із знаком в системі числення, визначеній другим аргументом, у тип `int`;
- `short shortValue()` – перетворення значення у тип `short`;
- `String toBinaryString(int i)` – перетворення цілого значення `i` у текстовий вигляд з базою 2 (двійковий);
- `String toHexString(int i)` – перетворення цілого значення i у текстовий вигляд з базою 16(шістнадцятковий);
- `String toOctalString(int i)` – перетворення цілого значення i у текстовий вигляд з базою 8 (вісімковий);
- `String toString()` – перетворення значення у тип `String`;

- `String toString(int i)` – перетворення значення i у тип `String`;
- `String toString(int i, int radix)` – перетворення цілого значення i у рядок в заданій системі числення `radix`;
- `Integer valueOf(String s)` – створення об'єкту `Integer`, ініціалізованого величиною, визначеною у строковій змінній `s`;
- `Integer valueOf(String s, int radix)` створення об'єкту `Integer`, ініціалізованого величиною, визначеною у строковій змінній s, записаній у системі числення `radix`.

Приклад:

```
public class Integer_example_1 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //створення об'єкту Integer
        Integer i = new Integer(1);
        //виведення числового значення об'єкту Integer
        System.out.println ("i = " + i );
        //перетворення числового значення змінної i
        //типу Integer у текстове відображення
        //та відображення даного тексту на екран
        String s = i.toString();
        System.out.println ("s = " + s );
        //перетворення текстового представлення
        //числового значення у тип Integer
        //та відображення отриманої змінної на екрані
        String s1 = "13";
        Integer i1 = Integer.parseInt(s1);
        System.out.println ("i1 = " + i1 );
```

```
    }
}
```

Результат виконання:

```
i = 1
s = 1
i1 = 13
Integer.MAX_VALUE = 2147483647
Integer.MIN_VALUE = -2147483648
```

Клас `Byte`

Клас `Byte` є стандартною оболонкою для байтових величин [8].

Конструктори класу `Byte`:

- `Byte (byte value)` – створення об'єкта `Byte` із значенням `value`;
- `Byte (String s)` – створення об'єкта `Byte` на основі текстового значення `s`.

Поля класу `Byte`:

- `static int MAX_VALUE` – максимальна величина типу `byte`;
- `static int MIN_VALUE` – мінімальна величина типу `byte`;
- `static Class TYPE` – об'єкт класу, що представляє тип `byte`.

Основні методи класу `Byte`:

- `byte byteValue()` – отримання значення типу `byte`;
- `int compareTo(Byte byte)` – порівняння з об’єктом `Byte`;
- `int compareTo(Object o)` – порівняння з іншим об’єктом;
- `static Byte decode(String nm)` – перетворення рядка у тип `Byte`;
- `double doubleValue()` – перетворення значення у тип `double`;
- `boolean equals(Object obj)` – перевірка на рівність з іншим об’єктом;

- `float floatValue()` – перетворення значення у тип `float`;
- `int hashCode()` – отримання hash-коду об’єкта;
- `int intValue()` – перетворення значення у тип `int`;
- `long longValue()` – перетворення значення у тип `long`;
- `static byte parseByte(String s)` – перетворення текстового значення у тип `byte`;
- `static byte parseByte(String s, int radix)` – перетворення текстового значення в системе числення `radix` у тип `byte`;
- `short shortValue()` – перетворення значення у тип `short`;
- `String toString()` – перетворення значення у тип `String`;
- `static String toString(byte b)` – перетворення значення типу `byte` у тип в `String`;
- `static Byte valueOf(String s)` – перетворення текстового значення у тип `Byte`;
- `static Byte valueOf(String s, int radix)` – перетворення текстового значення в системі числення `radix` у тип `Byte`;

Приклад:

```
public class Byte_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        byte i = 3;//створили змінну типу byte
//створили об’єкт типу Byte на її основі
        Byte b1 = new Byte(i);
        System.out.println ("b1 = " + b1 );
        Byte b2 = new Byte("1");//Byte з тексту
        System.out.println ("b2 = " + b2 );
        //перетворили Byte  у double
        double d=b1.doubleValue();
        System.out.println ("d = " + d );
        int j=b2.intValue();//перетворили Byte у int
        System.out.println ("j = " + j );
```

```
        j=b1.compareTo(b2);//порівняли 2-а Byte
        if(j<0)
            System.out.println ("b1 < b2 ");
        else if(j>0)System.out.println ("b1 > b2 ");
        else System.out.println ("b1 = b2 ");
    }
}
```

Результат виконання:

```
b1 = 3
b2 = 1
d = 3.0
j = 1
b1 > b2
```

Клас `Float` [8]

Конструктори класу `Float`:

- `Float (double value)` – створює об'єкт `Float`, на основі аргументу `value`, при цьому `double value` перетворюється на тип `float`;
- `Float (float value)` – створює об'єкт `Float`, на основі аргументу `value`;
- `Float (String s)` – створює об'єкт `Float` із заданого рядка .

Поля класу `Float`:

- `static int MAX_EXPONENT` – максимальний показник ступеня, який може мати змінна типу `float`;
- `static float MAX_VALUE` – максимальне значення модуля для типу `float`;

- `static int MIN_EXPONENT` – мінімальний показник ступеня, який може мати змінна типу `float`;.
- `static float MIN_VALUE` – мінімальне значення модуля для типу `float`;
- `static float NaN` – константа, яка містить значення Not-a-Number (`NaN`) для типу `float`;
- `static float NEGATIVE_INFINITY` – константа, яка позначає від'ємну нескінченність для типу `float`;
- `static float POSITIVE_INFINITY` – константа, яка позначає додатню нескінченність для типу `float`;
- `static int SIZE` – число бітів, які використовуються для подання значення типу `float`;
- `static Class <Float> TYPE` – екземпляр класу, який представляє тип примітиву `float`.

Основні методи класу `Float`:

- `byte byteValue()` – отримання значення типу `byte`;
- `int(float f1, float f2)` – порівнює два значення типу `float`;
- `int compareTo(Float anotherFloat)` – порівнює числові еквіваленти двох значень типу `Float`;
- `double doubleValue()` – перетворює значення типу `Float` у тип `double`;
- `float floatValue()` – повертає числове значення даного об'єкту `Float` у форматі типу `float`;
- `int hashCode()` – повертає хеш-код для даного об'єкту типу `Float`;
- `float intBitsToFloat (int біти)` – перетворює бітове представлення заданого числа з плаваючою крапкою у число типу `float`;
- `int intValue()` – перетворює значення даного числа типу `Float` у тип `int`;
- `isInfinite()` – перевіряє значення даного числа типу `Float` на нескінченість. Повертає відповідно `true` або `false`;

- `isNaN()` – перевіряє чи дорівнює значення даного об'єкта типу `Float` значенню `NaN`. Повертає `true` (якщо не дорівнює) або `false`;
- `isNaN(float v)` – перевіряє чи дорівнює числове значення `float` даного об'єкта типу `Float` значенню `NaN`. Повертає відповідно `true` або `false`;
- `long longValue()` – перетворює значення даного числа типу `Float` у тип `long`;
- `short shortValue()` – перетворює значення даного числа типу `Float` у тип `short`;
- `toHexString(float f)` – перетворює числове значення `float` даного об'єкта типу `Float` у рядкову змінну, яка відповідає шістнадцятковому представленні;
- `toString()` – перетворює об'єкт типу `Float` у рядкову змінну;
- `toString(float f)` – перетворює числове значення `float` у рядковий тип;
- `FloatOf(float f)` – перетворює значення типу `float` у об'єкт типу `Float`;
- `FloatOf(String s)` – перетворює строкове відображення числа з плаваючою крапкою у об'єкт типу `Float`.

Приклад:

```
public class Float_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        float f = 9;//створили змінну f типу float
        System.out.println("f="+f);//вивели f на екран
        //створили об'єкт F1 типу Float на основі f
        Float F1 = new Float(f);//вивели його на екран
        System.out.println("F1 = "+F1);
        //створили об'єкт F2 типу Float із тексту
        Float F2 = new Float("2.3");
        System.out.println("F2 = "+F2);
        //вивели розмір у бітах об'єкту типу Float
```

```
        System.out.println("Float.SIZE="
                                +Float.SIZE+" bit");
        //Порівняли 2-а об’єкти типу Float
        int j = F1.compareTo(F2);
        if(j<0)
            System.out.println ("F1 < F2 ");
        else if(j>0)System.out.println ("F1 > F2 ");
        else System.out.println ("F1 = F2 ");
        //перетворили Float на int
        int i = F2.intValue();
        System.out.println("i = "+i);
    }
}
```

Результат виконання:

```
f = 9.0
F1 = 9.0
F2 = 2.3
Float.SIZE=32 bit
F1 > F2
i = 2
```

Клас ***Short***

Конструктори класу `Short`:

- `Short (short)` – створює об'єкт `Short`, який представляє вказану величину типу `short`;
- `Short (String)` – створює об'єкт типу `Short`, на основі текстового представлення величини типу `short`.

Поля класу Short:

- static short MAX_VALUE – максимальне значення, яке може мати величина типу short;
- static short MIN_VALUE – мінімальне значення, яке може мати величина типу short;
- static int SIZE – це число бітів, яке займає у пам’яті величина типу short;
- static Class <Short> TYPE – екземпляр класу, який представляє тип примітиву short.

Основні методи класу Short:

- byte byteValue() – перетворює значення даного об’єкта типу Short на величину типу byte;
- int compareTo(Short) – порівнює два об'єкти Short;
- double doubleValue() – повертає значення даного Short у вигляді типу double;
- float floatValue() – повертає значення цього Short як число з плаваючою комою;
- int hashCode() – повертає хеш-код даного об’єкта Short;
- int intValue() – повертає значення даного об’єкта Short у вигляді типу int;
- long longValue() – повертає значення даного об’єкта Short у вигляді типу long;
- static short parseShort(String s) – перетворює рядковий аргумент на десяткове число зі знаком;
- static short parseShort(String s, int radix) – перетворює рядковий аргумент у тип Short, вважаючи, рядковий аргумент текстовим записом числа у системі числення, вказаній у аргументі radix;.

- `short shortValue()` – повертає значення даного об’єкта `Short` у вигляді типу `short`;
- `String toString()` – повертає об'єкт `String`, який відповідає текстовому представленню значення даного об’єкта `Short`;
- `static String toString(short s)` – повертає новий об'єкт `String`, для заданого числа типу `short`;
- `static Short valueOf(short s)` – повертає екземпляр `Short`, який представляє вказане значення `short`;
- `static Short valueOf(String s)` – повертає екземпляр `Short` на основі текстового представлення значення `short`, записаного в десятковій системі числення;
- `static Short valueOf(String s, int radix)` – повертає екземпляр `Short` на основі текстового представлення значення `short`, записаного в системі числення вказаній у аргументі `radix`.

Приклад:

```
public class Short_example {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        short sh = 34; //створюємо змінну типу short
        //створюємо на її основі об’єкт типу Short
        Short sh1 = new Short(sh);
        System.out.println("sh1 = "+sh1);
        // об’єкт типу Short на основі рядкової змінної
        Short sh2 = new Short("45");
        System.out.println("sh2 = "+sh2);
        //виводимо граничні значення типу short
```

```
        System.out.println("MIN_VALUE = "
                           +Short.MIN_VALUE );
        System.out.println("MAX_VALUE = "
                           +Short.MAX_VALUE );
        //виводимо розмір типу short у бітах
        System.out.println("SIZE="+Short.SIZE+" bit");
        //перетворюємо Short на double
        double f = sh2.doubleValue();
        System.out.println("f = "+f);
        //створюємо текстове представлення числа 15
        //у двійковому форматі
        String x = "1111";
        //перетворюємо його в тип short
        short sh3=Short.parseShort(x, 2);
        System.out.println("sh3 = "+sh3);
    }
}
```

Результат виконання:

```
sh1 = 34
sh2 = 45
MIN_VALUE  = -32768
MAX_VALUE  = 32767
SIZE=16 bit
f = 45.0
sh3 = 15
```

Клас Long

Конструктори класу `Long`:

- `Long(long l)` – створює об'єкт типу `Long` на основі значення типу `long`;
- `Long(String s)` – створює об'єкт типу `Long` на основі значення типу `String`.

Поля класу `Long`:

- `static long MAX_VALUE` – максимальне значення, яке може мати змінна типу `long`;
- `static long MIN_VALUE` –мінімальне значення, яке може мати змінна типу `long`;
- `static int SIZE` – об'єм пам'яті, яку займає змінна типу `long` у бітах.

Основні методи класу `Long`:

- `int compareTo(Long b)` – порівнює значення даного об'єкту типу `Long` з вказаним у аргументі;
- `static long parseLong(String s)` – перетворює текстове представлення числа типу `long` на числове значення типу `long`;
- `static Long valueOf(long b)` – створює об'єкт типу `Long` на основі змінної типу `long`, вказаної в аргументі;
- `static Long valueOf(String s)` – створює об'єкт типу `Long` на основі текстового представлення числа типу `long`;
- `short shortValue()` – перетворює об'єкт типу `Long` у значення типу `short`;
- `byte byteValue()` – перетворює об'єкт типу `Long` у значення типу `byte`;

- `int intValue()` – перетворює об’єкт типу `Long` у значення типу `int`;
- `long longValue()` – перетворює об’єкт типу `Long` у значення типу `long`;
- `float floatValue()` – перетворює об’єкт типу `Long` у значення типу `float`;
- `double doubleValue()` – перетворює об’єкт типу `Long` у значення типу `double`.

Приклад:

```
public class Long_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        long l = 567;
        Long L1 = new Long(l);
        System.out.println("L1 = "+L1);
        Long L2 = new Long("345");
        System.out.println("L2 = "+L2);
        System.out.println("MAX_VALUE                    =
"+Long.MAX_VALUE);
        System.out.println("MIN_VALUE                    =
"+Long.MIN_VALUE);
        byte b=L1.byteValue();
        System.out.println("b = "+b);
    }
}
```

Результат виконання:

```
L1 = 567
L2 = 345
```

```
MAX_VALUE = 9223372036854775807
MIN_VALUE = -9223372036854775808
b = 55
```

Клас `Double` [8]

Конструктори класу `Double`:

- `Double (double)` – створює об'єкт типу `Double` на основі аргументу типу `double`;
- `Double (String s)` – створює об'єкт типу `Double` на основі аргументу типу `String`.

Поля класу `Double`:

- `static int MAX_EXPONENT` – максимальний показник ступеня, який може мати змінна даного типу;
- `static double MAX_VALUE` – найбільше позитивне значення типу `double`;
- `static int MIN_EXPONENT` – мінімальний показник ступеня, який може мати змінна даного типу;
- `static double MIN_VALUE` – найменше позитивне ненульове значення типу `double`;
- `static double NaN` – значення `Not-a-Number` (`NaN`) для типу `double`;
- `static double NEGATIVE_INFINITY` – значення, яке відповідає від'ємній нескінченності для типу `double`;
- `static double POSITIVE_INFINITY` – значення, яке відповідає додатній нескінченності для типу `double`;
- `static int SIZE` – це число бітів, які використовуються для представлення значень типу `double`.

Основні методи класу Double:

- byte byteValue() – конвертує значення типу Double у значення типу byte;
- int CompareTo(Double D) – порівнює значення поточного об’єкту типу Double із значенням об’єкту типу Double вказаним у аргументі;
- double doubleValue() – конвертує значення поточного об’єкту типу Double у значення типу double;
- float floatValue() – конвертує значення поточного об’єкту типу Double у значення типу float;
- int hashCode() – повертає хеш-код для даного об'єкта Double;
- int intValue() – конвертує значення поточного об’єкту типу Double у значення типу int;
- boolean isInfinite() перевіряє чи є значення даного об’єкта типу Double нескінченним для даного типу;
- static boolean isInfinite(double v) - перевіряє чи є значення аргументу типу double нескінченним для даного типу;
- long longValue() – конвертує значення даного об’єкту типу Double у тип long;
- static double parseDouble(String s) – перетворює текстове представлення числа з плаваючою крапкою у тип double;
- short shortValue() – перетворює текстове представлення числа з плаваючою крапкою у тип short;
- static String toHexString (double d) – перетворює значення типу double шістнадцяткове текстове представлення у шістнадцятковій системі числення;
- String toString() – конвертує значення поточного об’єкту типу Double у об’єкт типу String;

- `static String toString(double d)` – конвертує значення поточного об'єкту типу `Double` у тип `String`;
- `static Double valueOf(double d)` – конвертує значення типу `double` у об'єкту типу `Double`;
- `static Double valueOf (String s)` – конвертує об'єкт типу `String` у об'єкт типу `Double`.

Приклад:

```
public class Double_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Double D1 = new Double(36);
        System.out.println("D1 = "+D1);
        double d = D1.doubleValue();
        System.out.println("d = " +d);
        float f = D1.floatValue();
        System.out.println("f = " +f);
        Double D2 = new Double("44.1");
        System.out.println("D2 = "+D2);
        System.out.println("MAX_VALUE  =
                "+Double.MAX_VALUE);
        System.out.println("MIN_VALUE  =
                "+Double.MIN_VALUE);
        System.out.println("NEGATIVE_INFINITY  =
                "+Double.NEGATIVE_INFINITY);
        System.out.println("POSITIVE_INFINITY  =
                "+Double.POSITIVE_INFINITY);
        System.out.println("SIZE  = "+Double.SIZE+
                  "bits");
```

```
        System.out.println("D1 hash-code: "+
             D1.hashCode());
        System.out.println("D2 hash-code: "+
             D2.hashCode());
        byte b = D1.byteValue();
        System.out.println("b = "+b);
    }
}
```

Результат виконання:

```
D1 = 36.0
d = 36.0
f = 36.0
D2 = 44.1
MAX_VALUE  = 1.7976931348623157E308
MIN_VALUE  = 4.9E-324
NEGATIVE_INFINITY  = -Infinity
POSITIVE_INFINITY  = Infinity
SIZE  = 64 bits
D1 hash-code: 1078067200
D2 hash-code: -1937063935
b = 36
```

Клас `Character`

Для роботи з символьними даними у мові Java є спеціальний тип `char`. На відміну від мови C/C++, де `char` – це цілочисельний тип з розміром 8 біт, Java для `char` застосовується кодування Unicode і для зберігання Unicode-символів використовується 16 біт. Діапазон допустимих значень – від 0 до 65536 [6,8,9].

Класом-оболонкою для типу `char` є клас `Character`.

Конструктор класу `Character`.

Клас `Character` має лише один конструктор `Character('a')` , який у якості параметра використовує значення типу `char`.

Поля класу `Character`:

- `BYTES` – розмір типу `char` в байтах;
- `MIN_VALUE` – мінімальне значення для типу `Character`;
- `MAX_VALUE` – максимальне значення для типу `Character`;
- `MIN_RADIX` – мінімальне значення системи числення, яке дозволено для представлення типу `Character`;
- `MAX_RADIX` – максимальне значення системи числення, яке дозволено для представлення типу `Character`;
- `SIZE` – розмір типу `char` в бітах.

Основні методи класу `Character`:

- `charValue()` – повертає символ, якому відповідає даний об’єкт типу `Character`;
- `boolean equals(Character Ch)` – порівнює поточний об'єкт типу `Character` із іншим об'єктом типу `Character`;
- `int hashCode()` – повертає хеш-код об'єкта;
- `int compareTo(Character Ch)` – порівнює поточний символ із іншим символом;
- `int compare(char ch1, char ch2)` – порівняти два значення типу `char`;
- `String toString()` – перетворити об’єкт типу `Character` на об’єкт типу `String`;
- `int digit(char ch, int radix)` – повернути числове значення символу `ch` у заданій системі числення `radix`;

- `char fordigit(int num, int radix)` – визначає символьне представлення числа `num` у заданій системі числення `radix`;
- `boolean isAlphabetic(char ch)` – перевіряє, чи є символ `ch` алфавітним;
- `boolean isDefined(char ch)` – перевіряє, чи визначено символ у кодуванні Unicode;
- `boolean isDigit(char ch)` – визначає, чи є символ `ch` цифрою;
- `boolean isLowerCase(char ch)` – визначає, чи є заданий символ `ch` малою літерою;
- `boolean isUpperCase(char ch)` – визначає, чи є символ `ch` символом верхнього регістра;
- `boolean isWhiteSpace(char ch)` – визначає, чи є певний символ пробілом згідно з класифікацією Java.

Приклад:

```
public class Character_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Character Ch1 = new Character('a');
        System.out.println("Ch1 = " + Ch1);
        Character Ch2 = new Character('b');
        System.out.println("Ch2 = " + Ch2);
        int i = Ch1.compareTo(Ch2);
        if (i<0) System.out.println("Ch1 <Ch2 ");
        else if (i>0)System.out.println("Ch1 >Ch2 ");
        else System.out.println("Ch1=Ch2 ");
        boolean b=Ch1.equals(Ch2);
        System.out.println("b = " + b);
        char ch1 = Ch1.charValue();
```

```
        System.out.println("ch1 = " + ch1);
        i = Character.MIN_VALUE;
        System.out.println("MIN_VALUE = " + i);
        i = Character.MAX_VALUE;
        System.out.println("MAX_VALUE = " + i);
       System.out.println("BYTES = " + Character.BYTES
            + " bytes");
       System.out.println("SIZE = " + Character.SIZE +
            " bits");
        i=Character.MIN_RADIX;
        System.out.println("MIN_RADIX = " + i);
        i=Character.MAX_RADIX;
        System.out.println("MAX_RADIX = " + i);
        i = Ch1.hashCode();
        System.out.println("hashCode = " + i);
        i=Character.digit('F', 16);
        System.out.println("in hexadecimal system i=" + i);
        ch1=Character.forDigit(12,16);
        System.out.println("ch1 = " + ch1);
    }
}
```

Результат виконання:

```
Ch1 = a
Ch2 = b
Ch1 <Ch2
b = false
ch1 = a
MIN_VALUE = 0
MAX_VALUE = 65535
BYTES = 2 bytes
```

```
SIZE = 16 bits
MIN_RADIX = 2
MAX_RADIX = 36
hashCode = 97
in hexadecimal system i = 15
ch1 = c
```

Клас Boolean

Клас `Boolean` є оболонкою типу `boolean`. Об'єкт `Boolean` містить єдине поле логічного типу. Крім того, цей клас включає методи перетворення `boolean` в `String` і назад, а також константи та методи корисні при роботі з логічним типом [8,9].

Конструктори класу `Boolean`:

- `Boolean(boolean value)` – створення об'єкта типу `Boolean` на основі типу `boolean`;
- `Boolean(String s)` – створення об'єкта типу `Boolean` на основі текстового значення `s["true" | "False"]`.

Поля класу `Boolean`:

- `static Boolean FALSE` – логічний об'єкт, який відповідає значенню «хибно»;
- `static Boolean TRUE` – логічний об'єкт, який відповідає значенню «істинно».

Основні методи класу `Boolean`:

- `boolean booleanValue()` – повертає значення типу `boolean` для поточного об’єкту типу `Boolean`;
- `boolean equals(Object obj)` – порівнює поточний об’єкт типу `Boolean` з об’єкт типу `Boolean`, вказаним у аргументі;

- `static boolean getBoolean(String name)` перетворює текстове значення об’єкту типу `String` на значення типу `boolean`;
- `int hashCode()` повертає значення hash-коду для поточного об'єкта типу `boolean`;
- `String toString()` – перетворює значення поточного об’єкта типу `Boolean` на текстове значення типу `String`;
- `static Boolean valueOf(String s)` – перетворює текстове значення об’єкта типу `String` на об’єкт типу `Boolean`.

Приклад:

```
public class Boolean_example {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Boolean B1 = new Boolean("true");
        System.out.println("B1 = " + B1);
        boolean b1 = false;
        Boolean B2 = new Boolean(b1);
        System.out.println("B2 = " + B2);
        boolean b2 = B2.booleanValue();
        System.out.println("b2 = " + b2);
        boolean b3 = B2.equals(B1);
        System.out.println("b3 = " + b3);
        boolean b4 = Boolean.getBoolean("1r2t");
        System.out.println("b4 = " + b4);
        int i = B1.hashCode();
        System.out.println("B1.hashCode = " + i);
    }
}
```

Результат виконання:

```
B1 = true
B2 = false
b2 = false
b3 = false
b4 = false
B1.hashCode = 1231
```

Інтерфейси і пакети

Інтерфейс – це посилальний тип в Java. Він схожий із абстрактним класом. Це сукупність абстрактних методів. За допомогою інтерфейсу можна вказати, що саме повинен виконувати клас, який його реалізує, але не спосіб реалізації. Спосіб реалізації обирає сам клас. Інтерфейси – це один із механізмів реалізації принципу поліморфізму [6,8,9].

Інтерфейс має ряд ознак, які роблять його подібним до класу:

- інтерфейс може містити будь-яку кількість методів;
- інтерфейси, як і класи, можуть бути оголошені з рівнем доступу `public` або `default`;
- інтерфейс розміщують у файлі з розширенням `.Java`, і ім'я інтерфейсу збігається з ім'ям файлу;
- байт-код інтерфейсу знаходиться у файлі з розширенням `.class`;
- інтерфейси включають у пакети, а їхній відповідний файл байт-коду повинен бути в структурі каталогів, яка збігається з ім'ям пакета.

Проте є і відмінності інтерфейсів від класів:

- неможливо створити екземпляр інтерфейсу;
- інтерфейси не мають конструкторів;
- усі методи інтерфейсів абстрактні;
- інтерфейс не розширюється класом, він реалізується класом;

- інтерфейс може бути побудований на основі кількох інтерфейсів (компесація відсутності множинного успадкування).

Змінні інтерфейсів є `public static final` за замовчуванням і ці модифікатори необов'язкові при оголошенні.

Щоб вказати, що клас реалізує інтерфейс, при оголошенні класу вказують ключове слово `implements` та ім'я інтерфейсу. Клас, який реалізує інтерфейс повинен містити повний набір методів, визначених у цьому інтерфейсі, але може мати і свої власні методи (відсутні у інтерфейсі).

Приклад:

```
//перший інтерфейс (файл show_interface)
package interf_example;
public interface show_interface {
        public void show();
}

//другий інтерфейс (файл get_interface)
package interf_example;
public interface get_interface {
    double get (double a);
}

//клас, який реалізує обидва інтерфейси
package interf_example;
public class comp_num implements show_interface,
get_interface{
    public double re, im;
    //конструктор за замовчуванням
    public comp_num(){
        re=0.0;
```

```
        im=0.0;
    }

//параметризований конструктор
    public comp_num(double rr, double ii){
        re=rr;
        im=ii;
    }
    //реалізація методу show (show_interface)
    public void show(){
        System.out.println("re = " + re);
        System.out.println("im = " + im);

    }
    //реалізація методу get (get_interface)
    public double get (double a){
        return a;
    }
}

//головний клас
package interf_example;
public class main_class {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        comp_num cn = new comp_num(1.0, 2.0);
        cn.show();
        double d,b;
        d=cn.get(cn.re);
        System.out.println("d = " + d);
```

```
        b=cn.get(cn.im);
        System.out.println("b = " + b);
    }
}
```

Структура Java програми

Будь-яка програма мовою Java складається з одного або декількох класів. Початок класу позначають службовим словом `class`, за яким слідує ім’я класу. Тіло класу поміщають у фігурних дужках. Всі дії у програмі виконуються за допомогою методів класів. Один із методів повинен називатися `main`. Саме з цього методу починається виконання програми [1-10].

Методи можуть повертати результат (тип результату вказується при визначенні методу перед його назвою, а може і не повертати (тип результату `void`). Якщо метод повертає результат, то це завжди лише один результат.

Метод може мати вхідні аргументи, або не мати їх. Якщо метод має аргументи, то їх перераховують через кому у дужках після імені методу. Тип кожного аргументу вказується перед його ім’ям. Перед типом аргументу можуть бути записані модифікатори (`public`, `static` тощо). Модифікатори не є обов’язковими, але винятком є метод `main`, для якого вони необхідні.

Типи відносин між класами [6,8,9]

Більшість класів програми пов'язані між собою і в процесі виконання програми взаємодіють один з одним. У процесі взаємодії міжкласами Java може існувати два типи відносин `IS-A` та `HAS-A`.

Відносини типу `IS-A` – це успадкування, яке було розглянуте вище. HAS-A відносини засновані на використанні. Тип `HAS-A` включає асоціацію, агрегацію та композицію.

Агрегація та композиція є окремими випадками асоціації. Агрегація – це відношення, коли один об'єкт є частиною іншого. А композиція – це ще тісніший зв'язок, коли об'єкт не тільки є частиною іншого об'єкта, а й взагалі не може належати іншому об'єкту. Розглянемо їх детальніше.

Якщо об'єкти одного класу містять посилання на один або більше об'єктів іншого класу, але при цьому відносини між об'єктами не носять характеру "володіння" або контейнеризації, то такі відносини називають асоціацією (`association`).

Приклад:

```
public class A_class {}

public class B_class{
   private A_class a;
}
```

Якщо об'єкт класу `A_class` створюється ззовні класу `B_class` і передається у конструктор для встановлення зв'язку, то такий тип відносин називається агрегацією. Якщо об'єкт класу `B_class` буде видалено, об'єкт класу `A_class` може й надалі використовуватись, за умови, що залишились діючі посилання на нього.

Приклад:

```
public class B_class {
    private A_class a;
    public B_class (A_class a) {
        this. A_class = a;
    }
}
```

Якщо об'єкт класу `A_class` створюється у конструкторі об’єкту класу `B_class`, то такий зв’язок називається композицією. Об'єкт класу `A_class` не може існувати без об'єкта класу `B_class`, що його створив:

Приклад:

```
public class B_class {
    private A_class a;
    public B_class() {
        this.a = new A_class();
    }
}
```

Використання описаних вище типів відносин між класами дозволяє полегшити вирішення проблем, які виникають під час перенавантаження методів.

Обробка виняткових ситуацій

Виняткова ситуація у Java – це деяка проблема, яка може виникнути виникає в ході виконання програми.

Наприклад:

- ArithmeticException – ділення на нуль;
- IndexOutOfBoundException - індекс поза межами масиву;
- IllegalArgumentException - використання невірного аргументу під час виклику методу;
- NullPointerException - використання порожнього посилання;
- NumberFormatException - помилка перетворення рядка на число;
- InterruptedException: потік перерваний іншим потоком;
- ClassNotFoundException: неможливо знайти клас.

Повний перелік виняткових ситуацій можна за необхідності знайти у документації на конкретну версію мови програмування Java.

У разі виникнення в Java виняткової ситуації (exception) за замовчуванням, програма/додаток завершуються в аварійному режимі, що у багатьох випадках є небажаним. Тому виникає необхідність якось перехоплювати такі події і обробляти їх для уникнення серйозних наслідків, які можуть мати місце в результаті аварійного завершення програми.

Для цієї мети у мові програмування Java передбачено використання спеціальної конструкції [6,8,9]:

```
try {
    // код, у якому може виникнути
    // винякова ситуація
} catch (виняток1 e1) {
    //код для обробки винятку1
} catch (Виняток2 e2) {
    // код для обробки винятку2
} catch (Виняток3 e3) {
    //код для обробки винятку3
    ...
}finally {
    //оператори, які обов'язково
    //повинні бути виконані при
    //завершенні обобки виняткової ситуації
    //(за необхідності)
}
```

У блоці try розміщують оператори, при виконанні яких можливе виникнення виняткових ситуацій. Ублоках catch розміщуються оператори, які призначені для обробки конкретної виняткової ситуації, яка вказується в

круглих дужках після ключового слова `catch`. Таких блоків може бути кілька. Їх кількість залежить від кількості можливих виняткових ситуацій, які планується обробляти. Блок `finally` – необов'язковий. За необхідності там розміщують дії, які повинні бути виконані при закінченні обробки будь-якої з передбачених виняткових ситуацій.

Слід пам'ятати, що:

- вираз `catch` не може існувати без оператора `try`;
- за наявності блоку `try/catch`, вираз `finally` не є обов'язковим;
- існування будь-якого коду в проміжку між блоками `try`, `catch`, `finally` є неможливим.

Приклад:

```
package try_example1;

public class try_example1 {

    public static void main(String[] args) {
    int array[] = new int[2];
    //блок перехоплення виняткової ситуації
    //виходу за межі масиву
    try {
    System.out.println("Доступ до 3-го елементу:" + array[3]);
    }catch(ArrayIndexOutOfBoundsException e){
    System.out.println("Виняткова ситуація:" + e);
    }finally {
        int i=0;
        for(i=0;i<array.length;i++)
        {
            array[i] = i;
        }
```

```
            System.out.println("Оператор finally виконаний.");
        }
        System.out.println("Ситуацію залагоджено:");
        for(int i:array)
            {
             System.out.println("array[" + i +"] = " + array[i]);
            }
        }
}
```

Результат виконання:

```
Виняткова
ситуація:Java.lang.ArrayIndexOutOfBoundsException: 3
Оператор finally виконаний.
Ситуацію залагоджено:
array [0] = 0
array [1] = 1
```

За необхідності виняткову ситуацію програміст може згенерувати сам програмно за допомогою ключового слова `throw`. При цьому необхідно відрізняти ключове слово `throw` від ключового слова `throws`:

- `throw` – використовується для генерації виняткової ситуації;
- `throws` – використовується в сигнатурі методів для попередження, що метод може генерувати виняткову ситуацію.

Крім того програміст має змогу створювати свої власні виняткові ситуації шляхом наслідування від класу `Exception`.

Приклад:

```
package try_example2;
public class getFactorial {
```

```
    //визначення програмістом нової виняткової
    //ситуації, яка виникатиме при заданні
    //від'ємного числа для розрахунку факторіала
    public      int   calcFactorial(int   num)   throws
Exception{

        if(num<1)  throw  new  Exception("The  number  is
less than 1");
        int result=1;
        for(int i=1; i<=num;i++){
        result*=i;
        }
        return result;
    }
}
package try_example2;

public class m_class {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        getFactorial ff = new getFactorial();
        //блок обробки виняткової ситуації
        //визначеної програмістом вище
        try{
            int result = ff.calcFactorial(-6);
            System.out.println(result);
        }
        catch(Exception ex){
            System.out.println(ex.getMessage());
```

```
        }
        }
    }
```

Результат виконання:

```
The number is less than 1
```

Система введення-виведення даних

Виведення інформації на консоль та зчитування інформації з консолі

Найбільш простим способом взаємодії з користувачем є використання консолі. На консоль можна виводити інформацію або, навпаки, зчитувати інформацію з консолі. Для виведення інформації на консоль в Java використовують клас *System* [1-6]. В попередніх прикладах ми уже активно використовували цей клас не вдаючись до подробиць. Наразі настав час трохи розширити наші уявлення про цей клас.

Для створення потоку виведення у класі `System` визначено об'єкт `out`. У цьому об'єкті визначено метод `println`, який дозволяє вивести на консоль деяке значення з наступним переведенням курсору консолі на наступний рядок. Якщо нема потреби переведення курсору на наступний рядок, то можна використати метод `print`, який робить те саме, але не переводячи курсор на наступний рядок.

Приклад:

```
package con_out;

public class con_out {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //демонстрація методу print
        System.out.print("Hi, I am Java print! ");
```

```
        System.out.print("I don't go to the next line! ");
        //демонстрація методу println
        System.out.println("");
        System.out.println("And I am Java println!");
        System.out.println("I am go to the next line! ");
    }
}
```

Результат виконання:

```
Hi, I am Java print! I don't go to the next line!
And I am Java println!
I am go to the next line!
```

Також у цьому класі є метод для форматованого виведення інформації на консоль – `printf`.

Приклад:

```
package con_out_printf;

public class con_out_printf {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int ii=10;
        float ff=13;
        char cc='$';
//приклад форматування виводу на екран з
//допомогою методу printf
        System.out.printf("ii=%d ff=%f cc=%c", ii, ff, cc\n);
        System.out.printf("ff(engineering format)= %e", ff);
    }
}
```

Результат виконання:

```
ii=10 ff=13,000000 cc=$
ff(engineering format)= 1,300000e+01
```

Як видно з прикладу метод `printf` дозволяє сформувати рядок для виведення згідно потреб, вказавши позицію для виведення змінної, а також формат її відображення. Порядок підстановки змінних відповідає порядку їх переліку, формат виведення змінних визначається за допомогою специфікаторів формату, а положення у рядку визначається положенням відповідних специфікаторів формату.

Можна використовувати такі специфікатори формату [1,3,4,6,8-10]:

- %d – для виведення цілих чисел;
- %x – для виведення шістнадцяткових чисел;
- %f – для виведення чисел із плаваючою крапко;
- %e – для виведення чисел в експоненційному (інженерному) фоматі;
- %c – для виведення одиночного символу;
- %s – для виведення строкових значень.

Крім методи `print`, `println` та `printf` допускають використання спеціальних символів (`\n`,`\t` і т.д.).

Для отримання даних з консолі у класі `System` визначено об'єкт `in`. Однак безпосередньо через об'єкт `System.in` не дуже зручно працювати, тому зазвичай використовують клас `Scanner` з пакета `Java.util`, який, у свою чергу, використовує `System.in`. Клас Scanner має ряд методів, які дозволяють отримати введені користувачем значення:

`next()` – зчитує введений рядок до першого пропуску;

`nextLine()` – зчитує весь введений рядок;

`nextInt()` – зчитує введене число типу `int`;

`nextDouble()` – зчитує введене число типу `double`;

nextBoolean() – зчитує значення типу boolean;

nextByte() – зчитує введене число типу byte;

nextFloat() – зчитує введене число типу float;

nextShort() – зчитує введене число типу short.

Приклад:

```
package con_scan;
import Java.util.Scanner;
public class con_scan {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Scanner in = new Scanner(System.in);
        System.out.print("Input your first name: ");
        //приклад зчитування рядка з клавіатури
        String name = in.nextLine();
        System.out.print("Input your age (full years): ");
        //приклад зчитування числа типу int з
        //клавіатури
        int age = in.nextInt();
        System.out.print("Input your weight, kg: ");
        //приклад зчитування числа типу float з
        //клавіатури
        float height = in.nextFloat();
        //приклад форматованого виведення інформації
        //на консоль з допомогою метода printf
        System.out.printf("\nYour name: %s \nyour age: %d
                \nyour height: %.2f \n", name, age, height);
        in.close();
    }
}
```

Результат виконання:

```
Input your first name: John
Input your age (full years): 22
Input your weight, kg: 66,7

Your name: John
your age: 22
your height: 66,70
```

Потокове введення-виведення інформації. Робота з файлами

Ключовим поняттям в процесі обміну інформацією є поняття потоку. Потік – це джерело інформації або одержувач інформації; його можна пов’язати будь-яким зовнішнім пристроєм.

Стосовно роботи з введенням-виведенням інформації потік (`stream`) розуміють як абстракцію, яка використовується для читання або запису інформації (файлів, сокетів, тексту консолі тощо). Об'єкт, з якого зчитується інформація, називається потоком введення, а об'єкт, у який записують інформацію, – потоком виведення.

Мова програмування Java підтримує два види потоків [8,9]: текстовий і бінарний. Текстовий потік – це послідовність рядків; кожен рядок має нуль або більше символів і закінчується символом «`\n`».

Бінарний потік – це послідовність байтів, які є деякими проміжними даними і характеризуються тим, що якщо їх записати, а потім за тими ж правилами, то зчитана інформація співпаде з початковою.

Основою всіх класів мови програмування Java, які керують потоками є два абстрактні класи: `InputStream` (на основі якого реалізовані потоки введення) і `OutputStream` (на основі якого реалізовані потоки виведення).

Але працювати з байтами не завжди зручно, тому для роботи з потоками символів були додані абстрактні класи `Reader` (для читання потоків символів) і `Writer` (для запису потоків символів).

Усі інші потокові класи є їх нащадками. Основні потокові класи: `InputStream`, `OutputStream`, `Reader`, `Writer`, `FileOutputStream`, `FileInputStream`, `FileReader`, `FileWriter`, `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`, `ByteArrayInputStream`, `ByteArrayOutputStream`, `CharArrayReader`, `CharArrayWriter`, `FilterInputStream`, `FilterOutputStream`, `FilterReader`, `FilterWriter`, `DataInputStream`, `DataOutputStream`, `ObjectInputStream`, `ObjectOutputStream`.

Розглянемо деякі з них.

Клас `InputStream`, основні методи:

- `int available()` – повертає кількість байтів, доступних для читання у потоці;
- `void close()` – закриває потік;
- `int read()` – повертає цілочисленне представлення наступного байта в потоці або (-1) у разі відсутності у потоці непрочитаних байтів;
- `int read(byte[] buffer)` – зчитує байти з потоку масив `buffer`, повертає кількість зчитаних байтів або (-1) якщо не було зчитано жодного байту;
- `int read(byte[] buffer, int offset, int length)` – зчитує деяку кількість байтів, яка дорівнює `length`, з потоку в масив `buffer`, розміщуючи зчитані байти в масиві, починаючи зі зміщення `offset`, тобто з елемента `buffer[offset]`. Метод повертає кількість успішно прочитаних байтів;

- `long skip(long number)` – пропускає в потоці при читанні деяку кількість байт, яка дорівнює `number`.

Клас `OutputStream` ,основні методи:

- `void close()` – закриває потік;
- `void flush()` – очищає буфер виводу, записуючи весь його вміст;
- `void write(int b)` – записує у вихідний потік один байт, представлений параметром `b`;
- `void write(byte[] buffer)` – записує у вихідний потік масив байтів `buffer`;
- `void write(byte[] buffer, int offset, int length)` – записує у вихідний потік деяке число байтів, що дорівнює `length`, з масиву `buffer`, починаючи зі зміщення `offset`, тобто з елемента `buffer[offset]`.

Абстрактний клас `Reader`, основні методи:

- `absract void close()` – закриває потік введення;
- `int read()` – повертає цілочисленне представлення наступного символу у потоці. або (-1) у разі відсутності у потоці непрочитаних байтів;
- `int read(char[] buffer)` – зчитує з потоку у масив `buffer` символи, кількість яких дорівнює довжині масиву `buffer` та повертає кількість успішно зчитаних символів або (-1), якщо досягнуто кінець потоку за меншої кількості зчитаних символів;
- `int read(CharBuffer buffer)` – зчитує у об'єкт `CharBuffer` символи з потоку та повертає кількість успішно зчитаних символів або (-1), якщо досягнуто кінець потоку за меншої кількості зчитаних символів;

- `absract int read(char[] buffer, int offset, int count)` – зчитує з потоку у масив `buffer` символи кількістю `count`, починаючи зі зміщення `offset`;
- `long skip(long count)` – пропускає задану в count кількість символів та повертає кількість успішно пропущених символів.

Абстрактний клас `Writer`, основні методи:

- `Writer append(char c)` – додає до кінця вихідного потоку символ c та повертає об'єкт тип `Writer`;
- `Writer append(CharSequence chars)` – додає до кінця вихідного потоку набір символів chars та повертає об'єкт типу `Writer`;
- `abstract void close()` – закриває потік;
- `abstract void flush()` – примусово записує дані з буфера у потік та очищує буфери потоку;
- `void write(int c)` – записує в потік один символ, код якого передано у вигляді аргументу;
- `void write(char[] buffer)` – записує у потік масив символів;
- `absract void write(char[] buffer, int off, int len)` – записує у потік з масиву `buffer` (починаючи з індексу `off`) `len` символів;
- `void write(String str)` – записує рядок у потік;
- `void write(String str, int off, int len)` – записує з рядка (починаючи з індексу `off`) у потік `len` символів.

Функціонал, описаний класами `Reader` та `Writer`, успадковується безпосередньо класами символьних потоків, зокрема класами `FileReader` та `FileWriter` відповідно, призначеними для роботи з текстовими файлами.

Класи FileInputStream і FileOutputStream є нащадками класів InputStream та OutputStream відповідно та успадковують всі їх методи [6,8,9].

Приклад:

```
package file_stream_example;

import Java.io.*;

public class file_stream_example {
    public static void main(String[] args) {
    //задаємо шлях до файлів
    String f_path="C:\\workspace\\file_stream_example\\
                  bin\\file_stream_example\\";
    //відкриваємо вхідний і вихідний потоки
    //для роботи з файлом, та
    //проводимо операції запису та зчитування
    //інформації, враховуючи можливість
    //виникнення виняткової ситуації, спричиненої
    //помилками введення виведення
    try(FileInputStream inf=new
              FileInputStream(f_path+"inp_file.txt");
              FileOutputStream outf=new
              FileOutputStream(f_path+"out_file.txt"))
        {
            byte[] buffer = new byte[256];
            int count;
            // зчитуємо текст з файла у буфер
            while((count=inf.read(buffer))!=-1){
                // записуємо текст з буфера у файл
                outf.write(buffer, 0, count);
```

```
            }
            System.out.println("Operation is
                              successful!");
            inf.close();
            outf.close();
        }
        catch(IOException ex){
             System.out.println(ex.getMessage());
        }
    }
}
```

Результат виконання:

```
Operation is successful!
```

Класи `FileInputStream` і `FileOutputStream` призначені для запису бінарних файлів, але можуть бути використані і для роботи з текстовими файлами, проте для роботи з текстовими файлами більше підходять інші класи.

Інші потокові класи для введення-виведення інформації подібні. Мають багато спільного з розглянутими вище, оскільки всі вони є нащадками класів InputStream та OutputStream. Методи роботи з ними також подібні, тому читачеві пропонується розглянути їх самостійно.

Багатопотокове програмування

Як і більшість сучасних мов програмування Java підтримує багатопотоковість. За допомогою багатопотоковості можна виділити у додатку кілька потоків, які виконуватимуть різні завдання одночасно. Більшість сучасних програм використовують в своїй роботі кілька потоків.

Використання багатопотоковості має свої переваги і недоліки. До переваг можна віднести збільшення швидкості обчислень, спільний доступ всіх потоків даної програми до глобальних змінних (що може полегшити обмін даними між потоками), можливість за необхідності створювати локальні змінні доступні лише конкретному потоку. До недоліків можна віднести більш інтенсивне використання пам’яті, оскільки кожен потік створює власний стек, необхідність синхронізації потоків, що ускладнює програму, а також той факт, що у випадку, коли платформа, на якій виконується програма, не підтримує багатопоточність, то програма буде виконуватись значно повільніше, ніж однопотокова [6,8,9].

Створення потоків

У мові програмування Java функціональність потоку реалізується у класі `Thread`. Щоб створити новий потік, необхідно створити об'єкт класу `Thread`.

Клас Thread має набір методів для роботи з потоками. Розглянемо основні з них:

- `getName()` – повертає ім'я потоку;
- `setName(String name)` – встановлює ім'я потоку;
- `getPriority()` – повертає пріоритет потоку;
- `setPriority(int proirity)` – встановлює пріоритет потоку (від 1 до 10), головний потік за замовчуванням має пріоритет 5;
- `isAlive()` – повертає true, якщо потік активний;
- `isInterrupted()` – повертає true, якщо потік був перерваний;
- `join()` – очікує завершення потоку;
- `run()` – визначає точку входження у потік;
- `sleep(int time)` – призупиняє потік на задану кількість мілісекунд;
- `start()` – запускає потік, викликаючи його метод run().

При запуску програми починає працювати її головний потік (за замовчуванням він має ім’я `main`). Всі інші потоки програми породжуються від нього або від породжених ним потоків.

Для створення нового потоку можна створити новий клас або наслідуючи його від класу `Thread` або реалізуючи в класі інтерфейс `Runnable`.

Приклад наслідування від класу `Thread`

```
package thread_example_1;
//створюємо власний потоковий клас
//шляхом успадкування і розширення
//бібліотечного класу Thread
public class my_thread extends Thread{
    //перенавантажуємо класу конструктор
    my_thread(String name){
        super(name);
    }
    //перенавантажуємо метод run
    public void run(){
         //виводимо повідомлення про старт
         //роботи потоку
        System.out.printf("I am %s and I started. \n",
              Thread.currentThread().getName());
         //використання методу sleep вимагає
         //врахування можливості виникнення
         //виняткової ситуації InterruptedException
        try{
            Thread.sleep(300);
        }
        catch(InterruptedException e){
```

```
            System.out.println("I am %s and I my work was
                        interrupted.");
        }
        //виводимо повідомлення про закінчення
        //роботи потоку
        System.out.printf("I am %s and I finished my work.
                \n", Thread.currentThread().getName());
    }
}
//головний клас
package thread_example_1;

public class thread_demo_1 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //виводимо повідомлення про старт
        //роботи головного потоку
        System.out.printf("%s thread started.\n",
                Thread.currentThread().getName());
        new my_thread("my_thread").start();
        //виводимо повідомлення про закінчення
        //роботи головного потоку
        System.out.printf("%s thread finished.\n",
                Thread.currentThread().getName());
    }

}
```

Результат виконання:

```
main thread started.
main thread finished.
I am my_thread and I started.
I am my_thread and I finished my work.
```

Приклад реалізації інтерфейсу `Runnable`

```
package thread_example_2;
//створення потокового класу з використанням
//інтерфейсу Runnable
public class my_thread_2 implements Runnable {

    Thread thrd;
    //створюємо параметризований конструктор
    //класу my_thread_2
    my_thread_2(String name){
    //створюємо власне сам потік
    thrd=new Thread(this, name);
    //запускаємо потік на виконання
    thrd.start();
     }
    //створюємо метод run оскільки
    //інтерфейс Runnable вимагає
    //обов'язкового створення даного методу
    //у класі
    public void run(){
        //виводимо повідомлення про старт
        //роботи потоку
       System.out.printf("%s started.\n",
                                thrd.getName());
```

```
            //використання методу sleep вимагає
            //врахування можливості виникнення
            //виняткової ситуації InterruptedException

            try{
                Thread.sleep(750);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been
                                        interrupted");
            }
            //виводимо повідомлення про закінчення
            //роботи потоку
            System.out.printf("%s finished.\n",
                                thrd.getName());
        }
}

package thread_example_2;

public class thread_demo_2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //виводимо повідомлення про старт
        //роботи головного потоку
        System.out.println(Thread.currentThread().getName()+"
                            thread started.");
        //створюємо потік класу my_thread_2
        my_thread_2 myThread2  = new
                    my_thread_2("myThread_2");
```

```
        //виводимо повідомлення про закінчення
        //роботи головного потоку
        System.out.println(Thread.currentThread().getNa
                    me()+" thread finished.");
    }
}
```

Результат виконання:

```
main thread started.
main thread finished.
myThread_2 started.
myThread_2 finished.
```

Кожен потік Java має пріоритет. Це допомагає операційній системі визначати порядок, у якому плануються потоки. Пріоритети потоків Java лежать в межах від 1 до 10. За замовчуванням потік має пріоритет 5.

Потоки з більш високим пріоритетом мають більше шансів на отримання процесорного часу. Однак, пріоритети потоків не гарантують певного порядку виконання потоків, вони лише перерозподіляють вірогідність.

Приклад:

```
package thread_example_3;

public class my_thread_3 extends Thread{

    Thread thrd;
    int sleep_time;
    //створюємо конструктор з параметрами:
    //ім'я потоку, пріоритет, тривалість сну
    my_thread_3(String name, int prior,int slt){
```

```
        //створюємо новий потік з заданим ім'ям
        thrd=new Thread(this, name);
        //задаємо час для методу sleep
        sleep_time=slt;
        //встановлюємо пріоритетпотоку
        thrd.setPriority(prior);
        }
        //визначаємо метод run
        public void run(){
            //виводимо повідомлення про старт потоку
            System.out.printf("%s started.\n", thrd.getName());
            //виконуємо метод sleep з обов'язковим
            //урахуванням можливості виникнення
            //виняткової ситуації InterruptedException
            try{
                Thread.sleep(sleep_time);
            }
            catch(InterruptedException e){
                System.out.println("Thread has been interrupted");
            }
            //виводимо повідомлення про закінчення
            // роботи потоку
            System.out.printf("%s finished.\n", thrd.getName());
        }
    }
```

```
//головний клас
package thread_example_3;

public class demo_tread_example_3 {

    public static void main(String[] args) {
        //виводимо повідомлення про старт
        //роботи головного потоку
        System.out.println(Thread.currentThread().getName()+"
                        thread started.");
        //створюємо 1-ий потік: ім'я - myThread_1, пріоритет 2
        //тривалість сну 3 с
        my_thread_3 myThread1 =
                    new my_thread_3("myThread_1",2,3000);
        //запускаємо цей потік
        myThread1.start();
        //створюємо 2-ий потік: ім'я - myThread_2, пріоритет 2
        //тривалість сну 0,1 с
        my_thread_3 myThread2 =
                  new my_thread_3("myThread_2",7,100);
        //запускаємо цей потік
        myThread2.run();
        //виводимо повідомлення про закінчення
        //роботи головного потоку
        System.out.println(Thread.currentThread().getName()     +
              " thread finished.");
    }
}
```

Результат виконання:

```
main thread started.
myThread_2 started.
myThread_1 started.
myThread_2 finished.
main thread finished.
myThread_1 finished.
```

Як видно з результатів виконання другий потік почав виконуватись раніше за перший хоч був запущений пізніше, але у нього був пріоритет 7, а у першого потоку пріоритет був 2. Крім того зверніть увагу, що виконання головної програми закінчилась до закінчення виконання першого потоку. Якщо ж треба, щоб основна програма гарантовано дочекалась завершення якогось конкретного потоку, то для цього потоку необхідно виконати метод `join()`. Для ілюстрації виконаємо попередній приклад, але додамо до нього запуск методу `join()` у методі `run()` для обох потоків.

Приклад:

```
package thread_example_4;

public class my_thread_4 extends Thread{

    Thread thrd;
    int sleep_time;
    my_thread_4(String name, int prior,int slt){
    thrd=new Thread(this, name);
    sleep_time=slt;
    thrd.setPriority(prior);
    }
```

```
    public void run(){
        System.out.printf("%s started.\n", thrd.getName());
        try{
        //під час запуску потоку декларуємо запуск
        //методу join для даного потоку
        thrd.join();
            Thread.sleep(sleep_time);
        }
        catch(InterruptedException e){
            System.out.println("Thread has been interrupted");
        }
        System.out.printf("%s finished.\n", thrd.getName());
    }
}

package thread_example_4;

public class demo_thread_example_4 {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName()+
                    " thread started.");
        my_thread_4 myThread1 =
                  new my_thread_4("myThread_1",2,3000);
        myThread1.run();
        my_thread_4 myThread2 =
                  new my_thread_4("myThread_2",7,100);
        myThread2.run();
        System.out.println(Thread.currentThread().getName()     +
                 " thread finished.");
```

```
    }
}
```

Результат виконання:

```
main thread started.
myThread_1 started.
myThread_1 finished.
myThread_2 started.
myThread_2 finished.
main thread finished.
```

Зверніть увагу на порядок виконання. В даному випадку пріоритет зовсім не вплинув на порядок виконання програми. Виконання другого потоку почалося лише після завершення першого потоку, а основна програма завершилась лише після, того як завершились обидва потоки.

Синхронізація потоків

Кожен об'єкт Java має асоційований з ним монітор. Монітор – це об'єкт, який використовується як взаємовиключне блокування. Коли потік ініціює блокування, то кажуть, що він входить до монітора. Тільки один потік виконання може в один і той же час володіти монітором. Всі інші потоки виконання, які намагаються увійти до заблокованого монітора, будуть призупинені доти, доки перший потік не вийде з монітора. Кажуть, що вони чекають на монітор [6,8,9].

Потік, який володіє монітором, може, якщо забажає, повторно увійти до нього. Якщо потік засинає, він утримує монітор. Один потік може захопити одразу кілька моніторів.

Якщо у програмі виконується одночасно кілька потоків, які використовують спільні ресурси, то виникає вірогідність одночасного звернення кількох потоків до якогось із ресурсів, що може привести до

взаємного блокування та інших небажаних наслідків. Тому виникає проблема синхронізації потоків.

Існує два способи створення синхронізованого коду:

- створення синхронізованих методів;
- створення синхронізованих блоків.

Для здійснення синхронізації використовують ключове слово `synchronized`. Якщо створюється синхронізований метод, то це ключове слово вказують у його описі. Коли потік виконує синхронізований метод, то для інших потоків цей метод стає тимчасово недоступним допоки даний поток не завершить роботу з ним.

Можна також синхронізувати об'єкт у блоці команд, виділивши потрібний блок команд фігурними дужками. Перед дужками необхідно вказати ключове слово `synchronized`.

Приклад:

```
package synchro_demo;
//клас, через який буде здійснюватись доступ
//до спільного ресурсу
public class Average_Array {
    private float average;
    //створення синхронізованого методу
    //який використовує спільний ресурс
    //масив, на який вказує посилальна змінна arr[]
    synchronized float average_Array(float arr[]){
        average=0;
        for(int i=0;i<arr.length;i++)
        {
            average+=arr[i];
            System.out.println("Running"+Thread.currentThread().
                    getName()+". "+"Sum = "+average);
```

```
                try{
                    Thread.sleep(20);
                }catch(InterruptedException ex){
                    System.out.println("Tread interrupted");
                }
            }
            //обчислення середнього значення масиву
            average=average/arr.length;
            return average;
        }
}

package synchro_demo;
//потоковий клас, який буде визивати синхронізований
//метод average_Array(float arr[]) класу Average_Array
public class child_thread implements Runnable{
    Thread thrd;
    //створення об'єкта класу, який буде здійснювати
    //доступ до спільного ресурса
    static Average_Array ava= new Average_Array();
    //посилальна змінна, яка міститиме адресу
    //спільного ресурсу
    float a[];
    //змінна для збереження результату роботи
    //метода average_Array(float arr[])
    //класу Average_Array
    float res;
    //конструктор створюваного потокового класу
    public child_thread(String name,float arr[])
    {
```

```
        thrd = new Thread(this, name);
        //присвоєння значення посилальній змінній,
        //яка міститиме адресу спільного ресурсу
        a=arr;
        //запуск потоку прямо із конструктора
        //потоковий метод start() викликає метод
        // run для даного потоку
        thrd.start();
    }

    //метод run створюваного потокового класу
    public void run()
    {
        float average;
        System.out.println(thrd.getName()+"starting!");
        //виклик методу average_Array(a)
        //класу Average_Array для здійснення доступу
        //для спільного ресурсу
        average=ava.average_Array(a);
        System.out.println("Average value for" + thrd.getName()+
            " - "+average);
        System.out.println(thrd.getName()+" stopping");
    }
}

package synchro_demo;
//головний клас
public class demo_main {

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

```
        System.out.printf("%s    thread    started.\n",
              Thread.currentThread().getName());
        //створення спільного ресурсу
        float a[]={2, 4, 6, 8, 10};
        //створення і запуск потоків, конкуруючих
        //за спільний ресурс
        child_thread ch1 = new child_thread("Tread 1",a);
        child_thread ch2 = new child_thread("Tread 2",a);
        System.out.printf("%s    thread    finished.\n",
                Thread.currentThread().getName());
    }
}
```

Результат виконання:

```
main thread started.
main thread finished.
Tread 1starting!
Tread 2starting!
RunningTread 1. Sum = 2.0
RunningTread 1. Sum = 6.0
RunningTread 1. Sum = 12.0
RunningTread 1. Sum = 20.0
RunningTread 1. Sum = 30.0
Average value forTread 1 - 6.0
Tread 1 stopping
RunningTread 2. Sum = 2.0
RunningTread 2. Sum = 6.0
RunningTread 2. Sum = 12.0
RunningTread 2. Sum = 20.0
RunningTread 2. Sum = 30.0
Average value forTread 2 - 6.0
Tread 2 stopping
```

Як видно з прикладу, один з потоків захоплює ресурс (масив даних) і не дозволяє іншому потоку використовувати цей ресурс до свого завершення.

При синхронізації потоків необхідно враховувати кілька важливих моментів [6,8,9]:

- потоки, які викликають нестатичні синхронізовані методи одного й того ж класу, блокуватимуть один одного лише тоді, коли вони викликані для одного об'єкта;
- потоки, які викликають статичні синхронізовані методи одного класу, завжди блокуватимуть один одного. Вони блокуються монітором Class об'єкта. Статичні синхронізовані та нестатичні синхронізовані методи не блокуватимуть один одного ніколи;
- для синхронізованих блоків потрібно дивитися, який об'єкт використовується для синхронізації;
- блоки синхронізовані по одному об'єкту блокуватимуть один одного.

Для розширення можливості синхронізації потоків у Java також реалізовано механізм взаємодії потоків за допомогою методів `wait()`, `notify()` та `notifyAll()`. Ці методи реалізовані як `final` у класі `Object`, тому вони доступні всім класам.

Всі три методи можуть бути викликані тільки з `synchronized` контексту.

Метод `wait()` змушує викликаючий потік віддати монітор і призупинити виконання до тих пір, поки якийсь інший потік не увійде до того ж монітору і не викличе метод `notify()`.

Метод `notify()` відновлює роботу потоку, який викликав `wait()` на тому самому об'єкті.

Метод `notifyAll()` відновлює роботу всіх потоків, які викликали `wait()` на тому самому об'єкті. Одному із потоків дається доступ, як правило на основі пріоритетності.

Розглянемо приклад. Нехай ми маємо два потоки і спільній ресурс – змінну. Один з потоків зменшує значення змінної, а інший збільшує. До того ж нам треба гарантувати, що кожна операція збільшення або зменшення змінної гарантовано не буде перериватись іншою операцією і ми на виході матимемо адекватне значення змінної. Крім того вимагатимемо, щоб змінна не ставала від'ємною, тобто, щоб не можна було відняти від неї більше за її значення. Така ситуація виникає, наприклад, при роботі зі складом. Не можна взяти зі складу товару більше, ніж там є. У спрощеному вигляді це можна реалізувати наступним кодом:

```
package synchro_demo2;

public class counter {
    //змінна – ресурс для спільного доступу
    private int mycounter=0;
    //змінна індикатор, яка вказує,
    //чи було додано щось до змінної mycounter
    //використовується для синхронізації
    //операцій додавання/віднімання
    //операції будуть виконуватись по черзі
    boolean added = false;
    //синхронізований метод додавання
    public synchronized int add_to_counter(int dc)
    {
        //перевірка стану індикатора і затримка
        //допоки індикатор не набуде потрібного
        //стану
        while (added) {
            try {
                wait();
            } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
    //виконання додавання
    this.mycounter+=dc;
    System.out.println("Added " + dc +"!");
    System.out.println("Now mycounter has value " +
                    this.mycounter +"!");
    added = true;
    //вивільнення спільного ресурсу
    notify();
    return this.mycounter;
}
//синхронізований метод віднiманн
public synchronized int subtract_from_counter(int dc)
{
    //перевірка стану індикатора і затримка
    //допоки індикатор не набуде потрібного
    //стану
    while (!added) {
        try {
            wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    //віднімання (відняти можна лише тоді, коли
    //спільний ресурс має достатню величину)
    if(dc<= this.mycounter){
    this.mycounter -= dc;
```

```
            System.out.println("Subtructed " + dc +"!");
             System.out.println("Now mycounter has value "
                       + this.mycounter +"!");
            }else {
              System.out.println("Operation is not possible
                    counter = " + this.mycounter +" which
                    is less than dc = "+dc+"!");
            }
            added = false;
             //вивільнення спільного ресурсу
            notify();
            return this.mycounter;
        }
}

package synchro_demo2;
//клас, який збільшує спільний ресурс
public class supplier implements Runnable{
    private counter suppcounter;
    //конструктор класу
    public supplier(counter my_supp_counter) {
    super();
        this.suppcounter = my_supp_counter;
    }
    //перенавантажений метод run
    @Override
    public void run() {
        for (int i = 1; i <=3; i++) {
            //виклик методу додавання
             this.suppcounter.add_to_counter(i);
```

```
        }
    }
}

package synchro_demo2;
//клас, який зменшує спільний ресурс
public class user implements Runnable{

    private counter usercounter;
//конструктор класу
    public user(counter myusercounter) {
        this.usercounter = myusercounter;
    }
    //перенавантажений метод run
    @Override
    public void run() {
        for (int i = 2; i <=3; i++) {
             //виклик методу віднімання
            this.usercounter.subtract_from_counter(i);
        }

    }
}

package synchro_demo2;
//головний клас
public class sync_demo2_main {

    public static void main(String[] args) {
         System.out.println("Main tread is beginning!");
```

```
        //створення спільного ресурсу
        counter mcounter=new counter();
        //створення конкуруючих потоків
        user muser = new user(mcounter);
        supplier msupplier = new supplier(mcounter);
        Thread u1 = new Thread(muser);
        Thread s1 = new Thread(msupplier);
        //запуск конкуруючих потоків
        u1.start();
        s1.start();
        System.out.println("Main tread is finishing!");
    }
}
```

Результат виконання:

```
Main tread is beginning!
Main tread is finishing!
Added 1!
Now mycounter has value 1!
Operation is not possible counter = 1 which is less
than dc = 2!
Added 2!
Now mycounter has value 3!
Subtructed 3!
Now mycounter has value 0!
Added 3!
Now mycounter has value 3!
```

Додаткові засоби мови програмування Java

Мережеві засоби Java

Однією з найсильніших сторін мови програмування Java є наявність розвинених засобів для розробки мережевих застосунків. Набір класів і інтерфейсів для мережевих застосунків знаходиться у пакеті Java.net [6,8,9].

Пакет Java.net забезпечує підтримку двох загальних мережевих протоколів:

- TCP – TCP – це протокол управління передачею, який забезпечує надійний зв'язок між двома застосунками. Java TCP зазвичай використовується через Інтернет-протокол, який називається TCP/IP.
- UDP - UDP - це протокол дейтаграм користувача, протокол без встановлення з'єднання, який дозволяє передавати пакети даних між застосунками.

Відмінності TCP від UDP:

- TCP надійніший та здійснює контроль над процесом обміну даними, що гарантує доставку пакетів даних у незмінному вигляді, у заданій послідовності та без втрат, UDP нічого не гарантує;
- TCP вимагає попередньо встановленого з'єднання, UDP з'єднання не вимагає;
- UDP забезпечує вищу швидкість передачі;
- UDP краще підходить для мережевих ігор та програм, які відтворюють потокове відео, звук тощо.

Робота по протоколу TCP

В основі роботи по протоколу TCP лежить поняття сокету. Сокети забезпечують механізм зв'язку між двома комп'ютерами, які використовують

TCP. Клієнтська програма створює сокет на своєму кінці зв'язку та намагається підключити цей сокет до сервера.

Зв’язок відбувається на основі заданих адрес сервера і клієнта. Адреса передбачає ідентифікатор машини у просторі мережі Internet та номер порта. Ідентифікатор машини може бути доменним ім'ям, наприклад, me.kpi.ua, або звичайним IP.

Порт - унікальний номер, з яким пов'язаний певний сокет (цей термін буде розглянутий далі).

Коли з'єднання встановлено, сервер створює об'єкт сокета на кінці зв'язку. Після цього клієнт і сервер можуть спілкуватися, записуючи дані у сокет та зчитуючи дані із сокету.

Клас Java.net.Socket є сокетом, а клас Java.net.ServerSocket надає механізм серверної програми для прослуховування клієнтів і встановлення з'єднань з ними.

При встановленні з'єднання TCP між двома комп'ютерами з використанням сокетів виконуються такі етапи [6,8-10]:

- сервер створює екземпляр об'єкта `ServerSocket`, який визначає, за яким номером порту має відбуватися зв'язок;
- сервер викликає метод `accept()` класу `ServerSocket`, який очікує, доки клієнт не підключиться до сервера за вказаним портом;
- клієнт створює екземпляр об'єкта сокета, вказуючи ім'я сервера та номер порту підключення (сервер уже повинен бути запущений і має очікувати підключення клієнта);
- конструктор класу `Socket` здійснює спробу підключити клієнта до вказаного сервера та номера порту, якщо зв'язок встановлений, клієнт має змогу підтримувати зв'язок з сервером через об’єкт класу `Socket`;
- на стороні сервера метод `accept()` повертає посилання на новий сокет на сервері, підключений до поточного клієнтського сокету;

Після встановлення з'єднання зв'язок може відбуватися з використанням потоків вхідних/вихідних даних. Кожен сокет має потік вихідних даних (`OutputStream`), і потік вхідних даних (`InputStream`). `OutputStream` клієнта підключений до `InputStream` сервера, а `InputStream` клієнта підключений до `OutputStream` сервера.

TCP є двостороннім протоколом зв'язку, тому дані можуть передаватися з обох потоків одночасно. Розглянемо класи, які надають методи для використання сокетів.

Клас `ServerSocket` [8,9]

Конструктори класу `ServerSocket`:

- `public ServerSocket(int port) throws IOException` - створює серверний сокет, пов'язаний із зазначеним портом, якщо порт вже пов'язаний з іншим застосунком, то генерується виняткова ситуація;
- `public ServerSocket(int port, int backlog) throws IOException` – аналогічний попередньому, але другий параметр `backlog` вказує, скільки клієнтів, можуть одночасно знаходитись в черзі очікування;
- `public ServerSocket(int port, int backlog, InetAddress address) throws IOException` – третій параметр `InetAddress` вказує локальну IP-адресу і використовується у серверах, які можуть мати декілька IP-адрес;
- `public ServerSocket() throws IOException` – створює серверний сокет не прив’язаний до сервера, для подальшої роботи з сервером потребує виконання процедури прив’язки сокета до сервера.

Основні методи класу `ServerSocket`:

- `public int getLocalPort()` – повертає номер порта, який прослуховується даним серверним сокетом;

- `public Socket accept() throws IOException` – прослуховує порт чекаючи на вхідного клієнта, блокується допоки клієнт не підключиться до сервера на вказаному порту або не закінчиться час очікування сокету (за умови, що значення часу очікування було встановлено за допомогою методу `setSoTimeout()`, або на невизначений термін, якщо час очікування не встановлено);
- `public void setSoTimeout(int timeout)` – встановлює час очікування клієнта методом `accept()` сокета сервера;
- `public void bind (SocketAddress, int backlog)` – прив'язує сокет до вказаного сервера та порту в об'єкті `SocketAddress`, якщо `ServerSocket` було створено за допомогою конструктора без аргументів.

Клас `Socket` [8,9]

Конструктори класу `Socket`:

- `public Socket(String host, int port) throws UnknownHostException, IOException` – підключається до вказаного сервера через вказаний порт, у разі невдачі генерує виняткову ситуацію;
- `public Socket(InetAddress host, int port) throws IOException` – метод аналогічний попередньому конструктору, але додатково хост позначається хост об'єктом типу `InetAddress`;
- `public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException` – здійснює підключення до вказаного хоста та порту, створюючи на локальному хості сокет з вказаними адресою та портом;
- `public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException` – метод ідентичний попередньому конструктору, але у

першому аргументі хост позначається об'єктом `InetAddress`, а не рядковою змінною з адресою;

- `public Socket()` – створює непідключений сокет, який для підключення до сервера потребує використання метода `connect()`.

Основні методи класу `Socket`:

- `public void connect(SocketAddress host, int timeout) throws IOException` – підключає сокет до зазначеного хоста, якщо сокет був створений за допомогою конструктора без аргументів;
- `public InetAddress getInetAddress()` – повертає адресу іншого комп'ютера, до якого підключено даний сокет;
- `public int getPort()` – повертає номер порта, до якого прив'язаний сокет на віддаленій машині;
- `public int getLocalPort()` – повертає номер порта, до якого прив'язаний сокет на локальній машині;
- `public SocketAddress getRemoteSocketAddress()` – повертає адресу віддаленого сокету;
- `public InputStream getInputStream() throws IOException` – повертає потік вхідних даних сокету (потік вхідних даних підключено до потоку вихідних даних віддаленого сокету);
- `public OutputStream getOutputStream() throws IOException` – повертає потік вихідних даних сокету (потік вихідних даних підключено до потоку вхідних даних віддаленого сокету);
- `public void close() throws IOException` – закриває сокет і робить даний об'єкт сокета не здатним знову підключатися до сервера.

Клас `InetAddress` [8,9]

Основні методи класу `InetAddress`:

- `static InetAddress getByAddress(byte[] addr)` – повертає об'єкт `InetAddress` з урахуванням необробленої IP-адреси;
- `static InetAddress getByAddress(String host, byte[] addr)` – створює `InetAddress` на основі імені хоста та IP-адреси;
- `static InetAddress getByName(String host)` – визначає IP-адресу хоста на основі заданого імені хоста;
- `String getHostAddress()` – повертає рядок IP-адреси у текстовій формі;
- `String getHostName()` – повертає ім'я хоста для даного об’єкту типу `InetAddress`;
- `static InetAddress InetAddress getLocalHost()` – повертає локальний хост;
- `String toString()` – конвертує IP-адресу в адресний рядок.

Розглянемо приклади програм сервера та клієнта.

Приклад сервера:

```
import Java.io.*;
import Java.net.*;

class SampleServer extends Thread
{
    Socket s;
    int num;

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

```
    try
    {
        //змінна, яка містить кількість підключень
        int i = 0;
        // задаємо параметри для сокета
        //localhost, порт 3128
        ServerSocket server =
              new ServerSocket(3128, 0,
              InetAddress.getByName("localhost"));
        System.out.println("server is started");

        // слухаємо порт
        while(true)
        {
            // чекаємо на нове підключення,
            // після з'єднання запускаємо обробку
            // клієнта у новий  потік та збільшуємо
            //змінну i на одиницю
            new SampleServer(i, server.accept());
            i++;
        }
    }
    catch(Exception e)
    //обробка виняткової ситуації
     {System.out.println("main_init error: "+e);}
}
//конструктор класу SampleServer
public SampleServer(int num, Socket s)
{
    // копіюємо дані
```

```
        this.num = num;
        this.s = s;
        // запускаємо новий потік (див. ф-ю run())
        setDaemon(true);
        setPriority(NORM_PRIORITY);
        start();
    }

    public void run()
    {
        try
        {
            //отримуємо із сокета потік вхідних даних
            InputStream is = s.getInputStream();
            //відсилаємо вихідний потік даних
            //від сервера клієнту
            OutputStream os = s.getOutputStream();
            // буфер даних  64 кілобайта
            byte buf[] = new byte[64*1024];
            // читаємо дані від клієнта до 64кб,
            //результат — кількість прийнятых даних
            int r = is.read(buf);
            // формуємо рядок з отриманою
            //від клієнта інформацією
            String data = new String(buf, 0, r);
            // додаємо дані про сокет:
           System.out.println("I got from client: "+data);
            data = "i="+num+": "+"\n"+data;
            // відправляємо дані:
            os.write(data.getBytes());
```

```
            System.out.println("I sent to client: "+data);
            // закриваємо з'єднання
            s.close();
        }
        catch(Exception e)
        {
         // обробка виняткової ситуації
         System.out.println("run_init error: "+e);}
    }
}
```

Після запуску буде виведено на консоль повідомлення:

```
server is started
```

Подальші повідомлення залежать від інформації надісланої програмою клієнтом.

Приклад клієнта:

```
import Java.io.*;
import Java.net.*;

class SampleClient extends Thread
{
    public static void main()throws IOException {
        main(null);
    }

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

```
            //створюємо сокет і з'єднуємось із
            //сервером за адресою localhost:3128
            // отримуємо відповідь від сервера
            Socket s = new Socket("localhost", 3128);
            // виводимо у потік перший аргумент
            // заданий при визові:  сокета та порт
            String arg = "client: " +
                  s.getInetAddress().getHostAddress() +
                  ":"+s.getLocalPort();
            s.getOutputStream().write(arg.getBytes());
            System.out.println("I sent to server: "+arg);
            // читаємо відповідь
            byte buf[] = new byte[64*1024];
            int r = s.getInputStream().read(buf);
            String data = new String(buf, 0, r);
            System.out.println("I got from server: "+data);
            // виводимо відповідь на консоль
            System.out.println(data);
            s.close();
        }
        catch(Exception e)
        {
         //обробляємо виняткову ситуацію
         System.out.println("init error: "+e);}
    }
}
```

Результат виконання залежить від кількості попередніх з’єднань і може бути приблизно таким:

Сервер	Клієнт
server is started	I sent to server:
I got from client:	client: 127.0.0.1:55129
client: 127.0.0.1:55129	I got from server:
I sent to client:	i=0:client: 127.0.0.1:55129
i=0:client: 127.0.0.1:55129	i=0:client: 127.0.0.1:55129

Робота по протоколу UDP **[6,8,9]**

У мові програмування Java для роботи з UDP використовуються об'єкти класів DatagramSocket та DatagramPacket, які входять до складу пакету Java.net.

Клас `DatagramSocket`

Конструктори класу `DatagramSocket`:

- `DatagramSocket()` – створює об’єкт класу `DatagramSocket`, який прєднується до будь-якого вільного порту на локальній машині;
- `DatagramSocket(int port)` – створює об’єкт класу `DatagramSocket`, який прєднується до зазначеного порту на локальній машині;
- `DatagramSocket(int port, InetAddress addr)` – створює об’єкт класу `DatagramSocket`, який прєднується до зазначеного порту за однією з адрес локальної машини (`addr`).

Основні методи класу `DatagramSocket`:

- `void bind(SocketAddress addr)` – зв'язує даний об’єкт типу `DatagramSocket` з заданими адресою та портом;
- `void close()` – завершує роботу даного об’єкту типу `DatagramSocket`;
- `void connect(InetAddress address, int port)` – з'єднує даний об’єкт типу `DatagramSocket` із віддаленою адресою;

- `void disconnect()` – від'єднує сокет;
- `InetAddress getInetAddress()` – повертає адресу, з якою зв'язаний даний об’єкт типу `DatagramSocket`;
- `InetAddress getLocalAddress()` – повертає локальну адресу, з якою зв'язаний даний об’єкт типу `DatagramSocket`;
- `int getLocalPort()` – повертає номер порту на локальному комп’ютері, з яким зв'язано даний об’єкт типу `DatagramSocket`;
- `SocketAddress getLocalSocketAddress()` – повертає адресу кінцевої точки, з якою зв'язується даний об’єкт типу `DatagramSocket`;
- `int getPort()` – повертає номер порту, з яким з'єднується даний об’єкт типу `DatagramSocket`;
- `boolean isBound()` – повертає стан даного об’єкту типу `DatagramSocket`;
- `boolean isClosed()` – перевіряє, чи закритий даний об’єкт типу `DatagramSocket`;
- `boolean isConnected()` перевіряє – стан з'єднання даного об’єкту типу `DatagramSocket`;
- `void receive(DatagramPacket p)` – зчитує пакет дейтаграм;
- void send(DatagramPacket p) – відправляє пакет.

Клас `DatagramPacket`

Конструктори класу `DatagramPacket`:

- `DatagramPacket(byte[] buf, int length, InetAddress address, int port)` – створює пакет датаграми для надсилання пакетів довжини `length` на вказаний номер порту на вказаному вузлі;
- `DatagramPacket(byte[] buf, int offset, int length)` – створює `DatagramPacket` для прийому пакетів довжини `length`, вказуючи зміщення у буфері;

- `DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)` – створює пакет датаграми для відправлення пакетів довжини `length` зі зміщенням `offset` до вказаного номера порту на вказаному вузлі;
- `DatagramPacket(byte[] buf, int offset, int length, SocketAddress address)` – створює пакет датаграми для відправлення пакетів довжини `length` зі зміщенням `offset` до вказаного номера порту на вказаному вузлі, відрізняється від попереднього способом задання адреси;
- `DatagramPacket(byte[] buf, int length, SocketAddress address)` – створює пакет датаграми для надсилання пакетів довжини `length` на вказаний номер порту на вказаному вузлі, відрізняється від попереднього відсутністю параметра `offset`.

Основні методи класу `DatagramPacket`:

- `InetAddress getAddress()` – повертає IP-адресу машини, якою відправляється ця дейтаграма або з якої була отримана дейтаграма;
- `byte[] getData()` – повертає буфер даних;
- `int getLength()` – повертає довжину даних, які будуть надіслані або довжину отриманих даних;
- `int getPort()` – повертає номер порту на віддаленому вузлі, якому відправляється ця дейтаграма або з якого була отримана дейтаграма;
- `SocketAddress getSocketAddress()` – отримує `SocketAddress` (зазвичай IP-адресу + номер порта) віддаленого вузла, якому цей пакет відправляється або з якого він прибуває;
- `void setAddress(InetAddress iaddr)` – встановлює IP-адресу машини, з якої відправляється дана дейтаграма;
- `void setData(byte[] buf)` – встановлює буфер даних для даного пакета;

- `void setData(byte[] buf, int offset, int length)` – аналогічний попередньому, але додатково задає зміщення в буфері;
- `void setLength(int length)` – встановлює довжину даного пакета;
- `void setPort(int iport)` – встановлює номер порта на віддаленому вузлі, якому надсилається дана дейтаграма;
- `void setSocketAddress(SocketAddress address)` – встановлює `SocketAddress` (зазвичай IP-адресу + номер порту) віддаленого вузла, якому відправляється дана дейтаграма.

Розглянемо приклад.

Отримувач дейтаграм:

```
import Java.io.IOException;
import Java.net.DatagramPacket;
import Java.net.DatagramSocket;

class Recipient {

   public static void main(String[] args) {
       try {
           DatagramSocket ds = new DatagramSocket(1050);

           while (true) {
               DatagramPacket pack =
                     new DatagramPacket(new byte[5], 5);
               ds.receive(pack);
               System.out.println(new String(pack.getData()));
           }
       } catch (IOException e) {
           e.printStackTrace();
```

```
        }
    }
}
```

Відправник дейтаграм:

```
import Java.io.IOException;
import Java.net.DatagramPacket;
import Java.net.DatagramSocket;
import Java.net.InetAddress;
import Java.util.*;

class Sender {
    private String host;
    private int port;

    Sender(String host, int port) {
        this.host = host;
        this.port = port;
    }

    private void sendMessage(String mes) {
        try {
            byte[] data = mes.getBytes();
            InetAddress address =
                  InetAddress.getByName(host);
            DatagramPacket pack = new DatagramPacket(data,
                        data.length, address, port);
            DatagramSocket ds = new DatagramSocket();
            ds.send(pack);
            ds.close();
```

```
        } catch (IOException e) {
            System.err.println(e);
        }
    }

    public static void main(String[] args) {
    Sender sender = new Sender("localhost", 1050);

    String message = "Hello";
    Timer timer = new Timer();
    timer.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
            sender.sendMessage(message);
            System.out.println("Sending hello!");
        }
    }, 1000, 1000);
}

}
```

Результат роботи:

Відправник	**Отримувач**
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello
Sending hello!	Hello

Список використаних та рекомендованих джерел

1. Галкін О.В., Катеринич Л.О., Шкільняк О.С. Програмування на Java 8: Навчальний посібник для студентів факультету комп'ютерних наук та кібернетики. – К.: ЛОГОС, 2017. – 186 с.
2. Копитко М.Ф., Іванків К.С. Основи програмування мовою Java: Тексти лекцій. − Львів: Видавничий центр ЛНУ ім. Івана Франка, 2002.− 83 с.
3. Васильєв А. Н. Програмування мовою Java. — К.: Bohdan Books, 2022. — 2022. — 699 с.: іл. ISBN 9661058792, 9789661058797
4. Горбань А.В. Програмування в Java: Навчальний посібник [електронний ресурс]. –2008. – 310 с.
http://programming.in.ua/programming/basisprogramming/144-programming-Java-book.html
5. Java-програмування: комп'ютерний практикум [Електронний ресурс] : навч. посіб. для студ. спеціальності 122 «Комп'ютерні науки», освітньо-професійної програми «Комп'ютерний моніторинг та геометричне моделювання процесів і систем» / КПІ ім. Ігоря Сікорського ; уклад.: Ю. А. Тарнавський. – Електронні текстові дані (1 файл: 686 Кбайт). – Київ : КПІ ім. Ігоря Сікорського, 2021. – 95 с.
6. Брнакевич І.Є., Вагін П.П. Програмування мовою Java: використання фундаментальних класів: Тексти лекцій. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2002. – 75 с.
http://blues.franko.lviv.ua/ami/books/ami/Java_fundamental.pdf
7. Ратушняк Т. В. Програмування мовою JAVA: практикум: навчальний посібник. Державна фіскальна служба України, Університет державної фіскальної служби України. – Ірпінь, 2017. – 212 с.
8. Schildt H. Java: A Beginner's Guide: 8th Edition, McGraw-Hill Education, 2018, 684 p
9. Deitel P., Deitel H. Java How to Program, Early Objects: 11th Edition, Pearson, 2017, 1296 p.

10. Java. Теорія і практика: навчальний посібник для студентів природничих спеціальностей університетів / Кадомський К.К., Ніколюк П.К. – Вінниця: Донну, 2019. – 197 с.

Інтернет ресурси

1. http://www.iwanoff.inf.ua/Java_ua_2/index.html

2. Програмування на Java // http://Javaland.com.ua

3. Java Tutorial // https://www.w3schools.com/Java/ (англ.)

Предметний покажчик