1 ВСТУП ДО JAVA

1.1 Коротка історія створення мови

Історія Java розпочалася у 1991 р., коли група інженерів фірми Sun, котру очолював Patrick Naughton, та James Gosling працювали над розробкою мови програмування для простих пристроїв на кшталт дистанційних перемикачів для кабельного телебачення.

Оскільки такі пристрої не володіли значними обчислювальними потужностями та об'ємами пам'яті, мова повинна була генерувати якомога коротший код і не залежати від якоїсь конкретної архітектури. Проект отримав кодову назву “Green”.

Згаданим вимогам відповідав підхід, запропонований Niklaus Wirth (автором мови Pascal), ідея якого полягала в тому, щоб мова програмування генерувала проміжний код для гіпотетичної обчислювальної машини, не залежної від конкретної архітектури. Такі гіпотетичні машини називають віртуальними. Звідси і назва: Java Virtual Machine або JVM.

Інженери Sun працювали з мовою C++ на платформі Unix, тому у підсумкові нова мова, яку назвали Oak, отримала схожий синтаксис, а також стала об'єктно-орієнтованою. Пізніше виявилося, що мова з такою назвою вже існує, тому Oak перейменували у Java.

У 1993 р. з'явилося інтерактивне телебачення і кишенькові комп'ютери. Цікавість до проекту впала.

Тим часом, WWW набувала все більшого поширення. Постала потреба у сучасному браузері. Patrick Naughton та Jonathan Payne розробили HotJava браузер, який було представлено на SunWorld ’95 23 травня 1995 р. Браузер продемонстрував можливості мови, після чого почався бум популярності, який триває досі.

1.4 Найпростіша програма на Java

Перевіримо встановлення Java, написавши просту програму без використання середовища розрбки. Для цього слід створити файл у довільному текстовому редакторі, із розширенням java. Наприклад, test.java (рис. 1.1).

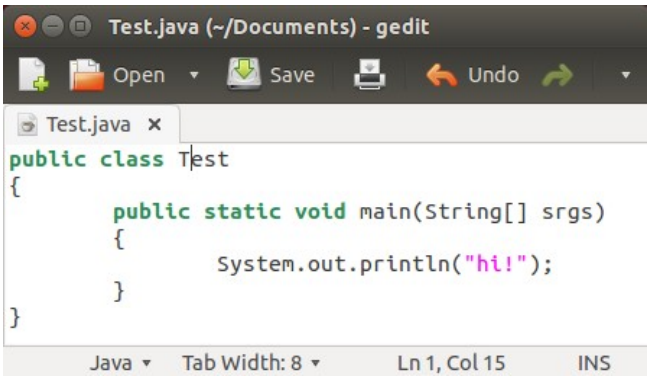


Рисунок 1.1 — Найпростіша програма на мові Java

Зверніть увагу на те, що назва класу співпадає з назвою файлу. Це – важливо. Файл повинен називатися так, як називається public-клас, який він містить.

Тепер потрібно скомпілювати цей файл за допомогою рядка у консолі:

javac Test.java

У разі успішної компіляції буде створено файл Test.class. Для виконання цього файлу

слід застосувати наступну інструкцію:

java Test

При цьому у консоль буде виведено “hi!”.

Зверніть увагу на те, що для виконання файлу test не потрібно вказувати його розширення .class.

1.5 Використання середовища розробки

Напишемо програму з тим самим кодом, скориставшись середовищем розробки IntelliJ IDEA.

5.1 Після завантаження середовища сліб вибрати пункт меню Create new project.

5.2 Виберіть тип проекту Java, а також вкажіть, де знаходяться файли JDK (Project SDK, рис. 1.2). Для того, щоб вказати шлях до JDK, натисніть кнопку New. Після цього слід двічі натиснути кнопку Next і вказати назву проекту. Нехай це буде My First Project (рис. 1.3).

5.4 Наступним етапом є створення класу. Для цього розгорніть список My First Project (рис. 1.4), виберіть пункт src, натисніть праву кнопку миші і виберіть New → Java Class. У вікні, що відкриється, введіть назву класу, наприклад, Test. Прийнято назви класів записувати з великої літери.

5.5 У вікні редактора, що відкриється, напишіть код, аналогічний до попереднього.

5.6 З пункту меню Run виберіть “Run... (Shift+Alt+F10)”. З вікна (рис. 1.5) виберіть конфігурацію “Test”. Якщо такої опції немає, слід вибрати Edit Configurations. При цьому відкриється вікно Run/Debug Configuration. Натисніть кнопку “+”, виберіть зі списку Add New Configuration пункт Application. Справа, у вкладці Configuration, задайте Name: Test (назва не пов'язана з назвою класу і може бути іншою), Main Class: Test, Working Directory: шлях до проекту.

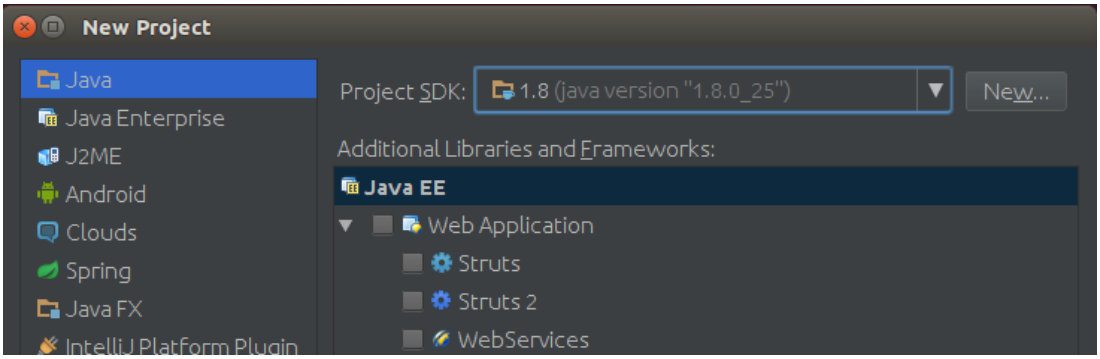


Рисунок 1.2 — Вибір типу проекту та встановлення необхідного SDK

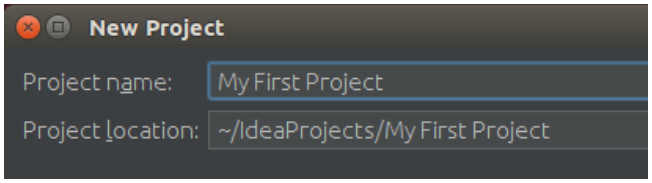


Рисунок 1.3 — Завершення процесу створення проекту

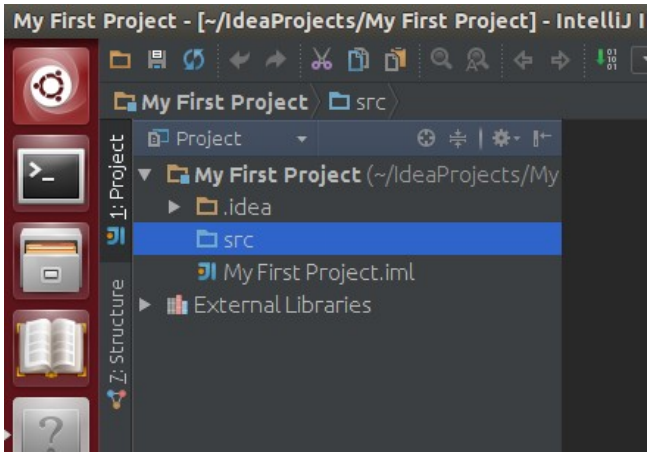


Рисунок 1.4 — Створення класу

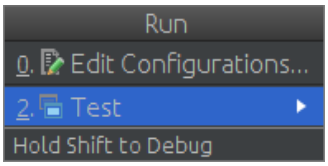


Рисунок 1.5 — Виконання програми

Після цього можна повторно вибрати пункт меню Run... , вибрати щойно задану конфігурацію і скомпілювати програму.

Після першої компіляції стане активним пункт меню Run → Run 'Test' (Shift F10).

1.6 Детальніше про код прикладу

```
public class Test {
        public static void main(String[] args){
        System.out.println("hi");
        }
}
```

Мова Java є чутливою до регістру (case sensetive). На відміну від C++, на мові Java будь-який програмний код повинен бути “загорнутий” у клас. Тому функція main належить класові Test.

Модифікатор доступу класу Test: public. Дане зроблено з метою забезпечення доступу до даного класу і виконання методу main, модифікатор доступу котрого також – public (метод main пивинен бути оголошениий як public у відповідності до Java Language Specification).

Метод main має тип void і не повертає коду завершення операційній системі на відміну від C/C++. У випадкові успішного завершення повертається код 0. Для повернення іншого коду слід скористатися методом System.exit.

Метод main завжди є static, тобто таким, який оголошений усередині класу, проте не оперує над об'єктами.

Стандартна термінологія Java визначає усі функції, котрі містить клас, як методи, а не функції-члени (C++).

За допомогою фігурних дужок позначено тіло методу. У даному випадкові тіло містить один оператор. Оператори відділяють один від одного за допомогою крапки з комою (;).

println – метод об'єкту System.out, використовують для виведення рядка у консоль.

Параметри функції main – масив args елементів, що мають тип String.

1.7 Коментарі на мові Java

Для задання коментаря використовують С-подібні конструкції:

```
// однорядковий коментар;
/*
        багаторядковий
        коментар
*/
```

Існує третій тип, який використовують для автоматичного генерування документації. Він виглядає наступним чином:

```
/**
        коментар
*/
```

Увага: багаторядкові коментарі не можуть бути вкладеними. У зв'язку з цим не можна деактивувати коментування частини коду шляхом коментування елементів /* та */ за допомогою цієї ж конструкції.

1.8 Типи даних

Java – жорстко типізована мова. Існують два види даних у Java: базові типи (примітиви) та тип референсні типи/об'єкти. Базивих типів налічують 8: з них 4 викоритовують для цілих чисел, 2 – для чисел із плаваючою крапкою, 1 – логічний тип і 1 – тип для позначення індивідуальних літер.

Базові типи даних Java (http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html):

Тип	Ширина, біт*	Опис
byte	8	Діапазон: від -128 до 127 включно
short	16	Діапазон: від -32,768 до 32,767 включно
int	32	Діапазон: від -2^{31} до $2^{31}-1$ (-2,147,483,648 до 2,147,483,647)
long	64	Діапазон: від -2^{63} до $2^{63}-1$ (–9,223,372,036,854,775,808 до 9 ,223,372,036,854,775,807)
float	32	IEEE 754** (+/-3.4028235+E38)
double	64	IEEE 754** (+/-1.7976931348623157+E308)
boolean	1	Значення: true/false
char	16	Діапазон: від '\u0000' (0) до '\uffff' (65,535 включно)

Увага: ширина типу визначає спосіб поведінки типу, а не зберігання у пам'яті. Так, змінна типу *boolean* не займає у пам'яті 1 біт при тому, що значущим для неї є 1 біт.

Розглядемо приклад:

```
public class First {
  public static void main(String[] args){
     long   x1 = 1;  int    x2 = 2;
     short  x3 = 3;  byte   x4 = 4;

     // float x5 = 1.1; // Error:(10, 19) java:
     //incompatible types: possible lossy conversion from double to float

     float  x6 = 1.1f;
     double x7 = 1.2;
     boolean x8 = true;

     System.out.println("hello!");
     System.out.println(x1);  System.out.println(x2);
     System.out.println(x3);  System.out.println(x4);
     System.out.println(x6);  System.out.println(x7);
     System.out.println(x8);
     if(true) System.out.println("hi");
  }
}
```

Зверніть увагу на те, що змінну x5 ініціалізувати таким чином, як у закоментованій частині коду, не вдасться. Це пов'язано із тим, що дефолтний тип числа з плаваючою крапкою не *float*, а – *double*. Неявного перетворення типів при цьому не відбувається. Натомість змінну x6 можна ініціалізувати значенням 1.1f. Модифікатор *f* вказує на тип числа – *float*.

Важливо: тип char у Java використовують для позначення індивідуальних символів у Unicode. Довжина символів при цьому – плаваюча.

Важливо: на відміну від C/C++ у Java немає перетворення для булевих значень у цілі числа (0 та 1).

1.9 Змінні

На мові Java кожна змінна повинна мати тип. Оголошують змінну, спершу вказуючи тип, а після нього – ідентифікатор змінної. Наприклад:

int x;

Правила побудови ідентифікаторів – наступні. Ідентифікатор повинен починатися з літери і утворювати послідовність літер та цифр.

Поняття “літера” та “цифра” у Java ширші, ніж у інших мовах програмування. Літери визначеня як 'A'–'Z','a'–'z', '_', '$', або довільний Unicode-символ, що позначає літеру у певній мові. Таким чином, ідентифікатори можуть містить і грецькі літери, і т. п. Аналогічно – щодо цифр.

Використовувати символ $ для утворення ідентифікаторів не рекомендовано.

Як ідентифікатори не можна використовувати зарезервовані слова Java.

1.10 Ініціалізація змінних

Після оголошення змінної її слід ініціалізувати. Змінні ініціалізують за допомогою

символу “=”:

```
int x;
x = 5;
```

Оголошення та ініціалізація можуть бути і поєднаними:

```
int x = 5;
```

Оголошувати змінні можна у довільному місці програми. Правилом хорошого тону є оголошення змінної якомога ближче до місця, де її буде використано уперше.

1.11 Область видимості

Локальними є змінні та константи, оголошені усередині блоку. Глобальними щодо блоку є змінні та константи, оголошені перед даним блоком.

На відміну від мов C/C++, Java не дозволяє співпадання ідентифікаторів локальних та глобальних змінних та констант.

1.12 Константи

Константи на мові Java оголошують за домопогою ключового слова final:

```
final int x = 5;
```

Змінити значення x після ініціалізації не можна.

1.13 Оператори

(http://docs.oracle.com/javase/tutorial/java/nutsandbolts/opsummary.html)

1.13.1 Арифметичні оператори +, -, * та / позначають додавання, віднімання, множення та ділення. Оператор / як результат повертає ціле число у випадкові, якщо обидва його аргументи – цілі числа, і число з плаваючою крапкою, якщо серед аргументів є число з плаваючою крапкою.

У випадкові ділення на нуль цілого числа результатом є вийняткова ситуація, а у випадку ділення числа з плаваючою крапкою – або нескінченність, або NaN.

1.13.2 Унарні оператори

Оператор + позначає додатнє число, - – від'ємне.

Як і C та C++, Java володіє операторами інкремента та декремента. Існує також префіксна та постфіксна форми цих операторів. Наприклад, результатом виконання коду

```
int x = 1;
System.out.println(x++);
System.out.println(x);
```

будуть цифри 1 та 2. Якщо замінити оператор x++ на ++x, результат зміниться на “2 2”. Це пов'язано з тим, що префіксна форма операторів інкременту та декременту одразу виконує дію, а постфіксна повертає попереднє значення змінної, а після цього виконує інкремент чи декремент.

Аналогічним чином працюють оператори -- у префіксній та постфіксній формах.

Зверніть увагу на те, що оператори ++ та -- можуть бути застосовані тільки до змінних, тому запис 3++ не має сенсу.

Як і C та C++, Java володіє скороченим оператором присвоювання:

```
int x = 1;
x += 2;
```

оператор += є скороченою формою x = x + 2. Аналогічно можна записати для -, * , / та % (залишок від ділення).

Оператор ! заперечує значення булевої змінної. Так, результатом виконання

```
boolean x = true;
System.out.println(!x);
```

є false.

1.13.3 Оператори рівності та відношень аналогічні до тих, що наявні у C/C++:

==	дорівнює
!=	не дорівнює
>	більше, ніж
>=	більше або дорівнює
<	менше, ніж
<=	менше або дорівнює

1.14 Форматування виведення

http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#print-boolean-
http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#println--

Простий спосіб вивести щось у консоль – скористатися методами *print()* або *println()* об'єкта System.out. Метод *print()* відрізняється від *println()* тим, що останній завершує виведення символом переходу на новий рядок.

```
public class Test {
   public static void main(String[] args){
      System.out.print("Is ");
      System.out.print("anybody ");
      System.out.print("out ");
      System.out.print("there?\n");
      System.out.println("Is anybody out there?");
   }
}
```

Результат:

Is anybody out there?
Is anybody out there?

Пакет *java.io* включає клас *PrintStream,* який володіє двома методами форматування, що можуть замінити *print* and *println*. Ці методи, *format* and *printf* – еквівалентні. Вже знайомий об'єкт *System.out* є об'єктом класу *PrintStream*, таким чином можна застосувати методи класу *PrintStream* до об'єкту *System.out*. Тобто, можна використовувати *format* або *printf* усюди, де у програмному коді застосовано методи *print* or *println*. Наприклад,

System.out.format(.....);

http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#format-java.lang.String-java.lang.Object...-
http://docs.oracle.com/javase/8/docs/api/java/io/PrintStream.html#printf-java.lang.String-java.lang.Object...-

public PrintStream format(String format, Object... args);
public PrintStream printf(String format, Object... args);

де *format* – рядок зі специфікацією форматування, що повинна бути застосована до *args,* (списку змінних) для виведення з використанням форматування.

Format має тип *String* і може складається з фіксованого тексту та специфікаторів виведення.

Формат специфікатора для символьних та числових типів наступний:

%[argument_index$][flags][width][.precision]conversion

Опційні параметри наведено у прямокутних дужках.

Agument_index$ використовують для позначення позиції аргумента у спискові параметрів *args*. У наступному прикладові цифрами 1, 2, 3 та 4 позначено рядки “Is ”, “anybosy ”, “out ”, “there?” відповідно.

```
public class Test {
   public static void main(String[] args){
     System.out.printf("%4$s %3$s %2$s %1$s", "there?", "out ", "anybody ", "Is ");
   }
}
```

Результат: Is anybody out there?

Опційний параметр *flags* – набір символів, які модифікують формат виведення. Набір доступних флагів залежить від *conversion*.

Flag	General	Character	Integral	Floating point	Date/ Time	Опис
'-'	так	так	так	так	так	Вирівнювання за лівою межею
'#'	так[1]	-	так[3]	так	-	Альтернативна, залежна від *conversion*, форма результату
'+'	-	-	так[4]	так	-	Результат завжди включає знак

' '	-	-	так[4]	так	-	Результат включає префіксальний пропуск для додатніх значень
'0'	-	-	так	так	-	Результат доповнюється нулями зліва
','	-	-	так[2]	так[5]	-	Результат включає групові сепаратори (grouping separators), що залежать від локалі
'('	-	-	так[4]	так[5]	-	Від'ємні значення воводяться у дужках ()

Примітки:

1. Залежно від Formattable.
2. Тільки для 'd'-перетворень.
3. Тільки для 'o', 'x', та 'X'-перетворень.
4. Для 'd', 'o', 'x', та 'X' -перетворень, застосованих до BigInteger , або 'd' , застосованих до byte, Byte, short, Short, int та Integer, long, і Long.

5 Тільки для 'e', 'E', 'f', 'g', and 'G' перетворень.

Інші символи, не визначені як флаги, не дозволені і зарезервовані на майбутнє. extensions.

Приклад:

```
   public static void main(String[] args){
      // вирівнювання за правою межею, ширина: 4 символи
      System.out.printf("%4d\n",3);
      // вирівнювання за лівою межею, ширина: 4 символи
      System.out.printf("%-4d\n",3);
      // відображається знак, ширина: 4 символи
      System.out.printf("%+4d\n",3);
      // додається пропуск перед додатнім числом
      System.out.printf("x:% d\n",3);
      // увімкнуто заповнюючі нулі, ширина: 4 символи
      System.out.printf("%04d\n",3);
      // увімкнуто відображення групових сепараторів
      System.out.printf("%,d\n",3000);
      // від'ємне число взято у дужки
      System.out.printf("%(d\n",-3);
   }
}
```

Результат:

```
   3
3
  +3
x: 3
0003
3,000
(3)
```

Опційний параметр *width* – додатнє десяткове число, що позначає мінімальну кількість

символів для виведення.

Опційний параметр *precision* – додатнє десяткове число, зазвичай використовують для обмеження кількості символів. Конкретна поведінка залежить від типу перетворення.

```
public class Test {
   public static void main(String[] args){
      System.out.printf("%.2f\n",1f);
      System.out.printf("%.2f\n",1.123);
      // символ "." - враховується
      System.out.printf("%05.2f\n",1.123);
      // символ "." - враховується
      System.out.printf("%06.2f",1.123);
   }
}
```

Результат:

1.00
1.12
01.12
001.12

Перетворення (*conversions*) поділяють на наступні категорії:

1. General (загальні) – можуть бути застововані до аргументу довільного типу.
2. Character (символьні) – можуть бути застосовані до базових типів, які містять символи Unicode: char, Character, byte, Byte, short та Short. Ці перетворення також можуть бути застосовані до типів int та Integer, якщо Character.isValidCodePoint(int) повертає true.
3. Numeric (числові)
    1. Integral (цілочислені) – можуть бути застосовані до цілочислених типів Java: byte, Byte, short, Short, int та Integer, long, Long, та BigInteger (але не char чи Character)
    2. Floating Point (з плаваючою крапкою) – можуть бути застосовані до типів Java з плаваючою крапкою: float, Float, double, Double, та BigDecimal .
4. Date/Time (дати/часу) – можуть бути застосовані до типів Java, здатних кодувати дату та час: long, Long, Calendar, Date and TemporalAccessor .
5. Percent (відсоток) – формує літерал '%' ('\u0025') .
6. Line Separator (лінійний розділювач)- формує платформозалежний лінійний розділювач.

У таблиці наведено перетворення, що підтримуються. Перетворення, позначені символами верхнього регістру ('B', 'H', 'S', 'C', 'X', 'E', 'G', 'A', and 'T') аналогічні до перетворень, позначених відповідними літерами нижнього регістру, за вийнятком представлення результату перетворення у верхоньому регістрів у відповідності до правил, що визначає локаль Locale. Результати таких перетворень аналогічні до тих, які можна отримати шляхом застосування String.toUpperCase() : *out.toUpperCase()*.

Таблиця 1.– Формати перетворень

Перетво-рення	Категорія аргументу	Опис
'b', 'B'	general	Якщо аргумент *arg* має значення *null*, результат – "*false*". Якщо *arg* має типу boolean або Boolean, тоді результат є рядком, повернутим String.valueOf(arg). У інших випадках результат

Перетво-рення	Категорія аргументу	Опис
		має значення "true".
'h', 'H'	general	Якщо аргумент *arg* має значення *null*, результат – "null". В інших випадках результат отримують за допомогою виклику *Integer.toHexString(arg.hashCode())*.
's', 'S'	general	Якщо аргумент *arg* має значення *null*, результат – "null". Якщо *arg* містить Formattable, викликається arg.formatTo . Інакше, результат отримують шляхлм виклику arg.toString().
'c', 'C'	character	Результат: символ Unicode
'd'	integral	Результат: десяткове ціле число
'o'	integral	Результат: вісімкове число
'x', 'X'	integral	Результат: шістнадцяткове число
'e', 'E'	floating point	Результат: десяткове число у науковому форматі
'f'	floating point	Результат: десяткове число
'g', 'G'	floating point	Результат отримують у вигляді числа у науковвій нотації або десяткового числа у залежності від точності і значення після округлення
'a', 'A'	floating point	Результатом є шістнадцяткове число з плаваючою крапкою з мантисою та експонентою. Це перетворення не підтримується для типу BigDecimal у зв'язку з тим, що останій знаходиться у категорії аргументів із плаваючою крапкою
't', 'T'	date/time	Префікс для символів перетворення дати та часу (див. Date/Time Conversions).
'%'	percent	Результат є літералом '%' ('\u0025')
'n'	line separator	Результатом є платформо-залежний лінійний розділювач

Приклад:

```
public class Test {
   public static void main(String[] args){
      System.out.printf("%d\n",5);
      System.out.printf("%o\n",8);
      System.out.printf("%h\n",15);
      System.out.printf("%H\n",15);
      System.out.printf("%e\n",1.1);
      System.out.printf("%E\n",1.1);
      // округлення
      System.out.printf("%g\n",11.7999);
      System.out.printf("%g\n",11.79999);
      System.out.printf("%g\n",111.799);
      System.out.printf("%g\n",1111.799);
      System.out.printf("%g\n",111111.799);
      System.out.printf("%g\n",1111111.7999);
```

```
        System.out.printf("%.2g\n",11.7999);
        System.out.printf("%.3g\n",11.7999);
        System.out.printf("%.3f\n",11.7999);
        System.out.printf("%10.4f\n",11.7999);
        System.out.printf("%-10.4f\n",11.7999);
        System.out.printf("%%\n");
        System.out.printf("%n");
        System.out.printf("----");
    }
}
```

Результат:

```
5
10
f
F
1.100000e+00
1.100000E+00
11.7999
11.8000
111.799
1111.80
111112
1.11111e+06
12
11.8
11.800
  11.7999
11.7999
%

----
```

1.15 Організація введення даних

Введення даних з клавіатури відбувається дещо складніше, ніж виведення через об'єкт System.out. Для цього будують об'єкт *scanner,* приєднаний до System.in:

Scanner in = new Scanner(System.in);

Після цього застосовують один із групи методів класу для читання. Наприклад:

```
public class Test {
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
        System.out.println("integer:");
        int i = in.nextInt();
        System.out.println("float:");
        float f = in.nextFloat();
```

```
        System.out.println("double:");
        double d = in.nextDouble();
        System.out.println("string:");
        String s = in.next();
        System.out.println("----------------");
        System.out.printf("%d %f %f", i, f, d, s);
    }
}
```

Результат:

integer:
12
float:
2.23
double
2.34
string:
hello
----
12 2.230000 2.340000 hello

2 ОПЕРАТОРИ УМОВИ ТА РОЗГАЛУЖЕННЯ. ПОРОЗРЯДНІ ОПЕРАТОРИ. ЦИКЛИ

2.1 Оператор *if*

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html

if (Condition)
Statement1 (s);
else
Statement2 (s);

Наприклад:

```
int a = 1;
int b = 2;
if(a<b)
   System.out.println("a<b");
else
   System.out.println("a не менше за b");
```

Параметр оператора *if* повинен мати тип *boolean*. Частина *else* не є обов'язковою. Оператори *if* можуть бути вкладеними, а вирази 1 та 2 – складеними (*{}*).

2.2 Оператор багатоваріантного розгалуження *switch*

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

```
int a = 1;
switch (a) {
   case 1:
      System.out.println("1");
      break;
   case 2:
      System.out.println("2");
      break;
   default:
      System.out.println("3");
}
```

Параметром оператора *switch* може бути ціле число або *String* (починаючи з Java 7)

```
public class Test {
   public static void main(String[] args){
      String a = "hello";  String b = "world";
      switch(a){
         case "hello": System.out.println("text: hello");  break;
         case "wordl": System.out.println("text: world"); break;
         default: System.out.println("text: another word");
      }
   }
}
```

Для успішної компіляції даного прикладу слід вибрати з меню (Itellij IDEA) File пункт Project Structure... і встановити значення Project language level: 7 – Diamonds, ARM, multy catch, etc або вибрати вищий рівень залежно від потреби.

2.3 Тернарний оператор

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html

Тернарний оператор можна розглядати як еквівалент *if-then-else*:

Result = Condition ? Value1 : Value2

Якщо умова *Condition* є дійсною (*true*), то *Result = Value1*, інакше R*esult = Value2*.

```
public class Test {
   public static void main(String[] args){
      int a = 1;
      String text = (a == 1) ? "Hello!" : "Bye!";
      System.out.println(text);
   }
}
```

Тернарний оператор підвищує читебельність коду, є зручним для використання замість *if-then-else* у випадкові, коли вирази компактні.

2.4 Логічні оператори

&& *умовне AND (І, кон'юнкція)*
|| *умовне OR (АБО, диз'юнкція)*
! *умовне NOT*
Дані оператори працюють тільки з булевими виразами.

```
public class Test {
   public static void main(String[] args){
      int a = 1;
      int b = 2;
      if((a<b) && ((b>3) || a<2 )) System.out.println("ok");
   }
}
```

2.5 Порозрядні оператори

Таблиця 2.1 – Порозрядні оператори

Оператор	Призначення
&	Порозрядне AND (*кон'юнкція*)
^	Порозрядне виключаюче XOR (виключна диз'юнкція)
|	Порозрядне OR (диз'юнкція)
~	Порозрядне NOT (інверсія)

```
public class Test {
   public static void main(String[] args){
      int a =              0x1115;
      int mask =           0x000F;
      System.out.println(a&mask);
   }
}
```

Результат: 5.

a = 0001 0001 0001 0101 (2) = 1115 (16)
mask = 0000 0000 0000 1111 (2) = 000F (16)

Змінна mask “маскує” нульовими бітами відповідні біти змінної a. Таким чином, можна дізнатися значення окремого біта чи групи біт.

```
public class Test {
   public static void main(String[] args){
      int a =     0b00001111;
      int b =     0b11110000;
      System.out.println(a | mask);
   }
}
```

Результат: 255. Чотири старших одиничкі отримано зі змінної mask, чотири молодших - зі змінної a.

```
public class Test {
   public static void main(String[] args){
      int a =     0b11111111;
      int b =     0b11110000;
      System.out.println(a ^ mask);
   }
}
```

Результат: 15. Операцію логічної диз'юнкції називають додаванням за модулем 2. При цьому результат набуває значення 1 у тому випадкові, якщо тільки один із операндів дорівнює 1.

```
public class Test {
   public static void main(String[] args){
      int a =     0b00001111;
      System.out.println(~a);
      System.out.println(Integer.toBinaryString(~a));
   }
}
```

Результатом будуть рядки: -16 та 11111111111111111111111111110000. Чому так? На перший погляд, інверсія числа 0b00001111 повинна дати 0b11110000. Слід пам'ятати, що тип даних int оперує 32-розрядними числами. Отже, насправді число 0b00001111 представлене як 0b00000000000000000000000000001111, а інверсія: 11111111111111111111111111110000.

Тоді яким чином число, що повинно представляти 4294967280 (10) чи 0xFFFFFFF0 (16), інтерпретується як -16? Справа у тому, що цілі числа на мові Java представляють у так званому додатковому (доповнюючому) коді (табл. 2.2).

Таблиця 2.2 – Доповнюючий код

Двійковий код	3-й біт	2-й біт	1-й біт	0-й біт	Десяткове число
0000	0	+0	+0	+0	0
0001	0	+0	+0	+2^0	1
0010	0	+0	+2^1	+0	2
0011	0	+0	+2^1	+2^0	3
0100	0	+2^2	+0	+0	4
0101	0	+2^2	+0	+2^0	5
0110	0	+2^2	+2^1	+0	6
0111	0	+2^2	+2^1	+2^0	7
1000	-2^3	+0	+0	+0	-8
1001	-2^3	+0	+0	+2^0	-7
1010	-2^3	+0	+2^1	+0	-6
1011	-2^3	+0	+2^1	+2^0	-5
1100	-2^3	+2^2	+0	+0	-4
1101	-2^3	+2^2	+0	+2^0	-3
1110	-2^3	+2^2	+2^1	+0	-2
1111	-2^3	+2^2	+2^1	+2^0	-1

Таким чином, : 11111111111111111111111111110000 (2 доп.) = -2^31+(2^31-1-15) = -16.

У табл. 2.3 наведено оператори зсуву.

Таблиця 2.3 – Оператори зсуву

<<	зсув на 1 розряд уліво, молодший біт заповнюється нулем
>>	зсув на 1 розряд управо, старший (знаковий) біт зберігає своє значення
>>>	зсув на 1 розряд управо, старший біт заповнюється нулем

```
public class Test {
   public static void main(String[] args){
      int a = 1;
      for(int i = 0; i < 32; i++){
         String b = Integer.toBinaryString(a);
         int l = b.length();
         String c = "";
         if(l<32) c = String.format("%0"+(32-l)+"d",0);
         System.out.printf("%s%s %d\n",c,b, a);
         a = a << 1;
      }
   }
}
```

Результат роботи програми:

```
00000000000000000000000000000001        1
00000000000000000000000000000010        2
00000000000000000000000000000100        4
00000000000000000000000000001000        8
00000000000000000000000000010000        16
00000000000000000000000000100000        32
00000000000000000000000001000000        64
00000000000000000000000010000000        128
00000000000000000000000100000000        256
00000000000000000000001000000000        512
00000000000000000000010000000000        1024
00000000000000000000100000000000        2048
00000000000000000001000000000000        4096
00000000000000000010000000000000        8192
00000000000000000100000000000000        16384
00000000000000001000000000000000        32768
00000000000000010000000000000000        65536
00000000000000100000000000000000        131072
00000000000001000000000000000000        262144
00000000000010000000000000000000        524288
00000000000100000000000000000000        1048576
00000000001000000000000000000000        2097152
00000000010000000000000000000000        4194304
00000001000000000000000000000000        8388608
00000010000000000000000000000000        16777216
00000100000000000000000000000000        33554432
00001000000000000000000000000000        67108864
00010000000000000000000000000000        134217728
00100000000000000000000000000000        268435456
01000000000000000000000000000000        536870912
10000000000000000000000000000000        -2147483648
```

Зверніть увагу на останнє значення (2^32 : -2147483648).
Розглянемо побітовий зсув управо:

```
public class Test {
   public static void main(String[] args){
      int a = 0x80000000;
      for(int i = 0; i < 32; i++){
         String b = Integer.toBinaryString(a);
         int l = b.length();
         String c = "";
         if(l<32) c = String.format("%0"+(32-l)+"d",0);
         System.out.printf("%s%s\t%d\n",c,b, a);
         a = a >> 1;
      }
   }
}
```

```
10000000000000000000000000000000        -2147483648
11000000000000000000000000000000        -1073741824
11100000000000000000000000000000        -536870912
11110000000000000000000000000000        -268435456
11111000000000000000000000000000        -134217728
11111100000000000000000000000000        -67108864
11111110000000000000000000000000        -33554432
11111111000000000000000000000000        -16777216
11111111100000000000000000000000        -8388608
11111111110000000000000000000000        -4194304
11111111111000000000000000000000        -2097152
11111111111100000000000000000000        -1048576
11111111111110000000000000000000        -524288
11111111111111000000000000000000        -262144
11111111111111100000000000000000        -131072
11111111111111110000000000000000        -65536
11111111111111111000000000000000        -32768
11111111111111111100000000000000        -16384
11111111111111111110000000000000-8192
11111111111111111111000000000000-4096
11111111111111111111100000000000-2048
11111111111111111111110000000000-1024
11111111111111111111111000000000 -512
11111111111111111111111100000000 -256
11111111111111111111111110000000 -128
11111111111111111111111111000000 -64
11111111111111111111111111100000 -32
11111111111111111111111111110000 -16
11111111111111111111111111111000 -8
11111111111111111111111111111100 -4
11111111111111111111111111111110 -2
11111111111111111111111111111111 -1
```

Легко помітити, що у випадкові зсуву уліво молодший розряд заповнюється нулем, а у випадкові зсуву управо старший розряд від'ємного числа заповнюється одиницею, зберігаючи знак числа (для додатнього числа – 0). Розглянемо роботу оператора >>>, котрий зсуває біти управо, при цьому не зберігаючи значення знакового розряду:

```
public class Test {
   public static void main(String[] args){
      int a = 0x80000000;
      for(int i = 0; i < 32; i++){
         String b = Integer.toBinaryString(a);
         int l = b.length();
         String c = "";
         if(l<32) c = String.format("%0"+(32-l)+"d",0);
         System.out.printf("%s%s\t%d\n",c,b, a);
         a = a >>> 1;
      }
   }
}
```

Результат роботи програми:

```
10000000000000000000000000000000    -2147483648
01000000000000000000000000000000    1073741824
00100000000000000000000000000000    536870912
00010000000000000000000000000000    268435456
00001000000000000000000000000000    134217728
00000100000000000000000000000000    67108864
00000010000000000000000000000000    33554432
00000001000000000000000000000000    16777216
00000000100000000000000000000000    8388608
00000000010000000000000000000000    4194304
00000000001000000000000000000000    2097152
00000000000100000000000000000000    1048576
00000000000010000000000000000000    524288
00000000000001000000000000000000    262144
00000000000000100000000000000000    131072
00000000000000010000000000000000    65536
00000000000000001000000000000000    32768
00000000000000000100000000000000    16384
00000000000000000010000000000000    8192
00000000000000000001000000000000    4096
00000000000000000000100000000000    2048
00000000000000000000010000000000    1024
00000000000000000000001000000000    512
00000000000000000000000100000000    256
00000000000000000000000010000000    128
00000000000000000000000001000000    64
00000000000000000000000000100000    32
00000000000000000000000000010000    16
00000000000000000000000000001000    8
00000000000000000000000000000100    4
00000000000000000000000000000010    2
00000000000000000000000000000001    1
```

Оператор <<< не існує.

2.6 Оператор циклу while з передумовою

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html

Синтаксис оператора:

```
while (condition) statement;
while (condition) {
        statement1;
        statement ...;
}
```

Розглянемо приклад:

```
public class Test {
   public static void main(String[] args){
      int i = 1;
      while(i < 10){
         int j = 1;
         while(j < 10){
            System.out.printf("%3d",i*j);
            j++;
         }
         System.out.println();
         i++;
      }
   }
}
```

Результат виконання (таблиця Піфагора):

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81
```

2.7 Оператор while з післяумовою

```
public class Test {
   public static void main(String[] args){
      int i = 1;
      do {
         int j = 1;
         do {
            System.out.printf("%3d",i*j);
            j++;
         } while(j < 10);
         System.out.println();
         i++;
      } while (i < 10);
   }
}
```

Синтаксис:

```
do
  statement ;
while (condition);
```

```
do {
  statement1;
  statement2;
  ...
} while (condition);
```

Результат виконання програми – аналогічний до попереднього випадку.

2.8 Оператор *for*

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html

Синтаксис:

```
for (initialization; termination; increment) {
    statement(s);
}
```

Оператор *for* надає можливість записати у компактній формі ітераційний процес над діапазоном значень.

Вираз *initialization* ініціалізує цикл, він виконується один раз. Коли вираз *termination* повертає значення *false*, ітерації циклу припиняються. Вираз *increment* виконується під час кожної ітерації. Зазвичай у цьому виразі інкрементують або декрементують значення параметра цикла.

Кожен із параметрів оператора *for* є опційним. Можливий запис оператора циклу *for* узагалі без параметрів. У цьому випадкові утворюється так званий “вічний” цикл.

```
for (;;) {
    statement(s);
}
```

Обчислимо за допомогою оператора *for* таблицю Пафігора (завдання – аналогічне до двох попередніх).

```
public class Test {
   public static void main(String[] args){
      for(int i = 1; i < 10; i++) {
         for (int j = 1; j < 10; j++)
            System.out.printf("%3d", i * j);
         System.out.println();
      }
   }
}
```

Область видимості змінної, оголошеної усередині циклу з оператором *for*, визначається тілом циклу. Поза межами тіла циклу змінна не досяжна.

2.9 Оператор *break*

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

Синтаксис:

break;

Розробники Java залишили у спискові зарезервованих слів *goto,* проте оператора з таким ідентифікатором не реалізовано. Використання оператора *goto* є джерелом потенційних помилок, хоча обґрунтований "стрибок" із циклу може виявитися вигідним. Для цього створено оператор *break* із міткою.

Розглянемо спершу звичайний оператор *break*.

Нехай нам потрібно дізнатися, як зростає банківський депозит упродовж 100 років при стартовому капіталі 100 грн і 10% річних. Податок на пасивний дохід у задачі не враховано.

```
public class Test {
   public static void main(String[] args){
      double x = 100;
      double rate = 0.1;
      for(int i = 1; i < 101; i++){
         x = x + x * 0.1;
         System.out.printf("year: %3d total: %.2f\n", i, x);
      }
   }
}
```

Результат:

```
year:   1 total: 110.00
year:   2 total: 121.00
year:   3 total: 133.10
…
year:  98 total: 1138893.58
year:  99 total: 1252782.94
year: 100 total: 1378061.23
```

Скажімо, чекати 100 років – довго. Тому нам цікаво, за скільки років вдасться акумулювати суму 1000 грн. Модифікуємо програму.

```
public class Test {
   public static void main(String[] args){
      double x = 100;
      double rate = 0.1;
      for(int i = 1; i < 101; i++){
         x = x + x * 0.1;
         System.out.printf("year: %3d total: %.2f\n", i, x);
         if(x>=1000.0) break;
      }
   }
}
```

Результат: ...
year: 25 total: 1083.47

У даному прикладові використано оператор *break*, який здійснює безумовне переривання ітерацій циклу, у тілі якого він розташований, після чого програма завершує роботу.

2.10 Оператор *break із міткою (labeled break)*

За деяких обставин може виникнути потреба реалізувати безумовний вихід із вкладених циклів. Один із варіантів вирішення проблеми: використання додаткових булевих змінних і постійний їх моніторинг усередині циклів різних рівнів вкладеності. Проте, це не найкращий спосіб, він вимагає додаткових операцій порівняння і займає зайвий час.

До послуг розробника на Java – оператор *break* із міткою (*labeled break*).

Синтаксис:

label:

```
loop1 {
        loop2 {
                loop..{
                        break label;
                }
        }
}
```

Тут *loop* – абстрактний оператор циклу (*while*, *do – while, for*).

```
public class Test {
   public static void main(String[] args){
     stop:
     for(int i = 1; i< 3; i++)
       for(int j = 1; j < 3; j++) {
          System.out.printf("i=%d j=%d\n", i, j);
          if ((i == 2) && (j == 2)) break stop;
       }
      System.out.println("stop");
   }
}
```

Результат:

```
i=1 j=1
i=1 j=2
i=2 j=1
i=2 j=2
stop
```

Оператор *break* з міткою перериває виконання оператора, позначеного міткою. Він не передає управління потоком мітці. Після виходу з циклу наступним негайно виконується

оператор, місце якого – за оператором, позначеним міткою (тут: System.out.println("stop");).

Також оператор *break* використовують для безумовного виходу з тіла оператора *switch*.

2.11 Оператор *continue*

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

Синтаксис:

continue;

Оператор *continue* припиняє виконання поточної ітерації циклу, у тілі якого він розташований, і передає керування операторові циклу, викликаючи цим або виконання наступної ітерації, або завершення циклу (залежно від виконання умови).

Наприклад, потрібно реалізувати цикл зі значенням параметру від 1 до 10, виключаючи 6:

```
public class Test {
   public static void main(String[] args){
      for(int i = 1; i < 11; i++){
         if(i == 6) continue;
         System.out.println(i);
      }
   }
}
```

Коли значення змінної циклу дорівнюватиме 6, буде виконано оператор *continue*, у результаті чого цикл перейде на нову ітерацію.

2.12 Оператор *continue з міткою*

Існує форма оператора *continue* з міткою, яка дозволяє перейти до наступної ітерації зовнішнього циклу, позначеного цією міткою.

```
public class Test {
   public static void main(String[] args){
      cont:
      for(int i = 1; i< 4; i++){
         for(int j = 1; j < 4; j++) {
            if (j == 2) continue cont;
            System.out.printf("i=%d j=%d\n", i, j);
         }
      }
   }
}
```

Легко помітити, що після набуття змінною циклу *j* значення 2 припиняється виконання поточної ітерації внутрішнього циклу і управління передається зовнішньому операторові *for*.

2.13 Оператор *return*

Оператор *return* – оператор розгалуження, котрий здійснює вихід із поточного методу і повертає потік управління до точки програми, з якої було викликано даний метод. Оператор *return* має дві форми: одна повертає значення, інша – ні.

Для того, щоб повернути значення, достатньо записати його (або вираз, що обчислює це значення) після ключового слова *return*:

```
return i;
return a+b;
...
```

Тип даних, що повертає оператор, повинен співпадати із оголошеним типом методу, якого стосується даний оператор *return*.

Якщо метод оголошено як *void*, використовують форму оператора *return*, котра не повертає жодного значення:

```
return;
```

3 МАСИВИ

3.1 Одновимірні масиви

Масивом називають структуру даних, призначену для зберігання набору однотипних значень. Доступ до окремих елементів масиву отримують за допомогою індексів.

Оголошують масив шляхом вказання типу масиву, за яким йдуть дужки “[]” та ім'я змінної масиву:

Тип [] змінна;

Наприклад,

```
int[] a;
```

Таким чином тільки оголошують масиви, у даному прикладові масив – не ініціалізовано. Для ініціалізації масиву використовують оператор new:

new тип [розмір];

Наприклад:

```
public class Test {
   public static void main(String[] args){
      int[] a = new int[5];
      a[0] = 1;
      a[1] = 2;
      a[2] = 3;
      a[3] = 4;
      a[4] = 5;
      for (int i = 0; i < 5; i++)
         System.out.printf("a[%d]=%d\n", i, a[i]);
   }
}
```

Оперативно вивести вміст масиву зручно за допомогою методу toString() класу Arrays. Попередньо слід імпортувати import java.util.Arrays:

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] a = new int[5];
      a[0] = 1;
      a[1] = 2;
      a[2] = 3;
      a[3] = 4;
      a[4] = 5;
      System.out.println(Arrays.toString(a));
   }
}
```

Результат: [1, 2, 3, 4, 5].

Масив можна оголосити також за допомогою конструкції

```
int a[];
```

Важливо: при створенні числового масиву його елементи ініціалізуються нулем, елементи масиву типу boolean – значенням false, елементи масиву об'єктів – значенням null, яке вказує на те, що окремі елементи ще не ініціалізовано.

Приклад:

```
public class Test {
   public static void main(String[] args){
      int[]       IntArray    = new int[5];
      float[]     FloatArray  = new float[5];
      double[]    DoubleArray = new double[5];
      boolean[]   BooleanArray= new boolean[5];
      String[]    StringArray = new String[5];
      for(int i = 0; i < 5; i++)
         System.out.printf(  "IntArray[%d]=%d  " +
                     "FloatArray[%d]=%.2f  " +
                     "DoubleArray[%d]=%.2f  " +
                     "BooleanArray[%d]=%b  " +
                     "StringArray[%d]=%s\n",
                i, IntArray[i],
                i, FloatArray[i],
                i, DoubleArray[i],
                i, BooleanArray[i],
                i, StringArray[i]);
   }
}
```

Властивість length містить значення довжини масиву:

```
public class Test {
   public static void main(String[] args){
      int[]       IntArray    = new int[5];
      System.out.println(IntArray.length);
   }
}
```

Змінити розмір масиву після його створення не можна. Можна змінювати індивідуальні значення окремих елементів. У випадкові, коли є потреба змінювати розмір масиву у ході виконання програми, слід скористатися іншою структурою даних – array list.

3.2 Оператор foreach

Java володіє потужним оператором циклу, який дозволяє отримати доступ до елементів масиву без потреби оперувати їхніми індексами.

for (змінна: колекція) оператор;

```
public class Test {
   public static void main(String[] args){
      int[] IntArray = {1, 2, 3, 4, 5};
      for(int variable : IntArray)
         System.out.println(variable);
   }
}
```

Поняття колекції тут позначає масив, об'єкт або клас, який імплементує інтерфейс iterable.

У випадкові ініціалізації масиву при його оголошенні оператор new не використовують. Елементи масиву задають через кому у дужках {}.

3.3 Анонімні масиви

Анонімні масиви не має ідентифікаторів, асоційованих з ними. Такі масиви оголошують наступним чином:

new тип[] {ініціалізатори};

За допомогою анонімних масивів можна здійснити переініціалізацію існуючих масивів:

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_1 = {1, 2, 3, 4, 5};
      System.out.print(Arrays.toString(arr_1));
      System.out.println("; length = " + arr_1.length);
      int[] arr2 = {6, 7, 8, 9, 10, 11, 12};
      arr_1 = arr2;
      System.out.print(Arrays.toString(arr_1));
      System.out.println("; length = " + arr_1.length);
   }
}
```

Також можна реалізувати доступ до елементів масиву у циклі, попередньо не ініціалізуючи оголошений масив:

```
public class Test {
   public static void main(String[] args){
      for (String var : new String[]{"h", "e", "l", "l", "o"})
         System.out.println(var);
   }
}
```

3.4 Копіювання масивів

Дозволено копіювання вмісту одної змінної масиву в іншу. При цьому слід пам'ятати, що у результаті обидві змінні вказуватимуть на один і той же масив.

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_copy_1 = new int[] {1, 2, 3, 4};
      int[] arr_copy_2;
      arr_copy_2 = arr_copy_1;
      System.out.println(Arrays.toString(arr_copy_2));
      arr_copy_1[0] = 10;
      System.out.println(Arrays.toString(arr_copy_2));
   }
}
```

Для того, щоб скопіювати вміст одного масиву у інший і не створити зв'язку між масивами, слід скористатися методом copyOf класу Arrays:

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#copyOf-boolean:A-int-

public static int[] copyOf(int[] original,int newLength);

Копіює масив original, обрізаючи або додаючи нулі так, шоб новий масив набув розміру newLength. У результаті виконання методу елементи масивів original та кінцевого матимуть однакові значення для довільного індексу. Для довільного індексу, дозволеного копією, але не дозволеного оригіналом, елементи копії дорівнюватимуть нулю. Такі індекси існують тоді і тільки тоді, коли довжина newLength більша за реальну довжину масиву original.

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_copy_1 = new int[] {1, 2, 3, 4};
      int[] copied_array = Arrays.copyOf(arr_copy_1, arr_copy_1.length);
      System.out.println(Arrays.toString(copied_array));
      arr_copy_1[0] = 20;
      // зміна значення arr_copy_1[0] не впливає на copied_array
      System.out.print(Arrays.toString(copied_array));
   }
}
```

```
[1, 2, 3, 4]
[1, 2, 3, 4]
```

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_copy_1 = new int[] {1, 2, 3, 4};
      int[] copied_array_1 = Arrays.copyOf(arr_copy_1, 10);
      int[] copied_array_2 = Arrays.copyOf(arr_copy_1, 3);
      System.out.println(Arrays.toString(copied_array_1)+"\n"+Arrays.toString(copied_array_2));
   }
}
```

```
[1, 2, 3, 4, 0, 0, 0, 0, 0, 0]
[1, 2, 3]
```

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_copy = new int[] {1, 2, 3, 4};
      int[] arr_dest = new int[] {5, 6, 7, 8, 9};
      System.out.println("copy: " + Arrays.toString(arr_copy));
      System.out.println("dest: " + Arrays.toString(arr_dest));
      arr_dest = Arrays.copyOf(arr_copy, 3);
      System.out.println("dest: " + Arrays.toString(arr_dest));
      arr_dest = Arrays.copyOf(arr_copy, 10);
      System.out.println("dest: " + Arrays.toString(arr_dest));
   }
}
```

```
copy: [1, 2, 3, 4]
dest: [5, 6, 7, 8, 9]
dest: [1, 2, 3]
dest: [1, 2, 3, 4, 0, 0, 0, 0, 0, 0]
```

За допомогою такого копіювання можна збільшити або зменшити розмір масиву, копіюючи його самого у себе і відповідним чином змінюючи параметр newLength:

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr = new int[] {1, 2, 3, 4};
      System.out.println("old:\t\t" + Arrays.toString(arr));
      arr = Arrays.copyOf(arr, arr.length + 1);
      System.out.println("increased:\t" + Arrays.toString(arr));
      arr = Arrays.copyOf(arr, arr.length - 2);
      System.out.println("decreased:\t" + Arrays.toString(arr));
   }
}
```

```
old:            [1, 2, 3, 4]
increased:      [1, 2, 3, 4, 0]
decreased:      [1, 2, 3]
```

На мові Java рядок *int[] x = new int[5]* є еквівалентом рядка *int* x = new int [5]* на C++. Проте, адресна арифметика – відсутня. Таким чином, не можна отримати доступ до наступного елемента масиву за рахунок інкременту поточного значення *x*.

3.5 Сортування масивів

Для сортування числових масивів можна скористатися одним із методів сортуванння, реалізованих у класі Arrays, зокрема

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#parallelSort-byte:A-

sort(a);
sort(a, start, end); які реалізують алгоритм Quicksort.

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_1 = new int[] {4, 2, 1, 3, 9, 8, 5, 7, 0, 6};
      int[] arr_2 = Arrays.copyOf(arr_1, arr_1.length);
      System.out.println("unsorted:\t" + Arrays.toString(arr_1));
      Arrays.sort(arr_1);
      Arrays.sort(arr_2, 1, 5);
      System.out.println("sorted 1:\t" + Arrays.toString(arr_1));
      System.out.println("sorted 2:\t" + Arrays.toString(arr_2));
   }
}
```

```
unsorted:       [4, 2, 1, 3, 9, 8, 5, 7, 0, 6]
sorted 1:       [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sorted 2:       [4, 1, 2, 3, 9, 8, 5, 7, 0, 6]
```

3.6 Бінарний пошук

Бінарний пошук у масивах реалізовано групою методів класу Arrays:

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#binarySearch-byte:A-byte-

public static int binarySearch(mun_x[] a, mun_x key); ma
public static int binarySearch(mun_x[] a, int fromIndex, int toIndex, mun_x key);

де *a* – масив типу x; *key* – значення, яке шукають, *fromIndex : toIndex – діапазон пошуку серед індексів масиву.*

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr = new int[] {4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11};
      System.out.println("length:\t\t" + arr.length);
      System.out.println("unsorted:\t" + Arrays.toString(arr));
      int index;
      index = Arrays.binarySearch(arr, 5);
      System.out.println(index);
      Arrays.sort(arr);
      System.out.println("sorted 1:\t" + Arrays.toString(arr));
      index = Arrays.binarySearch(arr, 5);
      System.out.println(index);
      index = Arrays.binarySearch(arr, 3, 6, 5);
      System.out.println(index);
      index = Arrays.binarySearch(arr, 12);
      System.out.println(index);
      index = Arrays.binarySearch(arr, 10);
      System.out.println(index);
   }
}
```

```
length:          11
unsorted:        [4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11]
-5
sorted 1:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
5
5
-12
-11
```

Методи повертають значення індекса, що відповідає шуканому елементові, у випадкові, якщо масив його містить. Якщо шуканий елемент відсутній у масиві, методи повертають значення (-(*insertion point*) – 1), де *insertion point* – індекс елемента, де міг би знаходитися шуканий, іншими словами, індекс першого більшого за *key* елемента масиву, або *a.length*, якщо усі елементи масиву менші за *key.* Таким чином методи повертають додатнє значення тільки у випадкові, коли шукане значення *key* знайдено.

Попередньо масив повинен бути впорядковано за допомогою одного з методів *sort()* класу *Arrays*. Інакше результат буде не визначеним.

Якщо масив містить декілька значень *key*, то немає гарантії, котрий із них буде знайдено.

3.7 Порівняння масивів

Два масиви вважають однаковими, якщо вони містять однакову кількість елементів, при цьому елементи масивів з однаковими індексами мають однакові значення.

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#equals-boolean:A-boolean:A-

public static boolean equals(mun_x[] a,mun_x[] a2);

Методи *equals* повертають *true* у випадкові, якщо масиви – однакові, і *false* – якщо різні.

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args){
      int[] arr_1 = new int[] {4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11};
      int[] arr_2 = new int[] {3, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11};
      int[] arr_3 = new int[] {4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11};
      System.out.println("arr1:\t" + Arrays.toString(arr_1));
      System.out.println("arr2:\t" + Arrays.toString(arr_2));
      System.out.println("arr3:\t" + Arrays.toString(arr_3));
      System.out.println("arr1 is equal to arr2: " + Arrays.equals(arr_1, arr_2));
      System.out.println("arr1 is equal to arr3: " + Arrays.equals(arr_1, arr_3));
   }
}
```

```
arr1:    [4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11]
arr2:    [3, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11]
arr3:    [4, 2, 1, 3, 9, 8, 5, 7, 0, 6, 11]
arr1 equals to arr2: false
arr1 equals to arr3: true
```

3.7 Заповнення масивів однаковими значеннями

https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#fill-boolean:A-boolean-

Заповнення масивів однаковими значеннями реалізують за допомогою методів fill:
public static void fill(тип_x[] a, тип_x val);

де *a* – масив, *val* – значення, яким заповнюють *a*.

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      int[] iArr = new int[10];
      Arrays.fill(iArr, 5);
      double[] dArr = new double[10];
      Arrays.fill(dArr, 5.5);
      boolean[] bArr = new boolean[10];
      Arrays.fill(bArr, true);
      System.out.println(Arrays.toString(iArr));
      System.out.println(Arrays.toString(dArr));
      System.out.println(Arrays.toString(bArr));
   }
}
```

```
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5, 5.5]
[true, true, true, true, true, true, true, true, true, true]
```

3.8 Багатовимірні масиви

Багатовимірні масиви оголошують, як і одновимірні, скориставшись відповідною кількістю пар дужок “[]”. Наприклад, двовимірний масив оголошують таким чином:

тип[][] ідентифікатор; наприклад:

```
int[][] a;
```

Як і у випадкові одновимірних масивів, масив слід ініціалізувати за допомогою виклику оператора *new*:

```
int[][] a = new int [x][y]; або
int[][] a;
a = new int[x][y];
```

де *x* та *y* – розмір масива. Якщо розміри масива відомі та відомі значення, якими він повинен бути ініціалізований, можна скористатися скороченим записом без використання оператора *new*:

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

```
            };
```

Оперативно вивести значення усіх елементів багатовимірного масиву можна за допомогою методу *deepToString()* класу *Arrays:*

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      int[][] a = {
                  {1, 2},
                  {3, 4}
               };
      System.out.println(Arrays.deepToString(a));
   }
}
```

```
[[1, 2], [3, 4]]
```

Оскільки на Java немає багатовимірних масивів узвагалі, а, насправді, існують фейкові масиви, котрі розглядаються як масиви масивів, існує можливість переставляти елементи масивів (котрі самі є масивами) місцями:

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      double[][] multiarr = new double[2][2];
      multiarr[0][0] = 0.0;
      multiarr[0][1] = 0.1;
      multiarr[1][0] = 1.0;
      multiarr[1][1] = 1.1;
      System.out.println(Arrays.deepToString(multiarr));
      double[] temp = multiarr[1];
      multiarr[1] = multiarr[0];
      multiarr[0] = temp;
      System.out.println(Arrays.deepToString(multiarr));
   }
}
```

```
[[0.0, 0.1], [1.0, 1.1]]
[[1.0, 1.1], [0.0, 0.1]]
```

3.8 Масиви із неоднорідною верхньою межею (Ragged arrays)

```
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      int raggedArray[][] = new int[10][];
      for (int i = 0; i < 10; i++)  { raggedArray[i] = new int[i + 1];   Arrays.fill(raggedArray[i], i); }
      for (int i = 0; i < 10; i++)
         System.out.println(Arrays.toString(raggedArray[i]));
```

```
    }
}
```

```
[0]
[1, 1]
[2, 2, 2]
[3, 3, 3, 3]
[4, 4, 4, 4, 4]
[5, 5, 5, 5, 5, 5]
[6, 6, 6, 6, 6, 6, 6]
[7, 7, 7, 7, 7, 7, 7, 7]
[8, 8, 8, 8, 8, 8, 8, 8, 8]
[9, 9, 9, 9, 9, 9, 9, 9, 9, 9]
```

3.9 Передача параметрів у функцію

Примітиви передають у функцію за значенням, а об'єкти та масиви – за посиланням. Для запобігання можливій зміні значень елементів масиву його слід передавати за допомогою функції CopyOf

```
import java.util.Arrays;
public class Test {
    public static void f(int[] a){
        for(int i = 0; i < a.length; i++)
            a[i] = 1;
        return;
    }
    public static void f1(int x){
        x = 1;
        return;
    }
    public static void main(String[] args) {
        int[] array = {1, 2, 3};
        f(array);
        System.out.println(Arrays.toString(array));
        array = new int[]{1, 2, 3};
        f(Arrays.copyOf(array, array.length));
        System.out.println(Arrays.toString(array));
        int iVar = 5;
        f1(iVar);
        System.out.println(iVar);
    }
}
```

```
[1, 1, 1]
[1, 2, 3]
5
```

4. Контроль за доступом до членів класу

Модифікатори рівня доступу визначають, чи інші класи можуть використовувати певні поля та методи класу. Існують 2 рівні котролю доступу:

- верхнього рівня: *public* або package-private (не вказують явно);
- на рівні членів: public, protected, private, package-private (не вказують явно).

Клас може бути оголошено з модифікатором *public*. У цьому випадкові він доступний звідусіль. Якщо клас не має модифікатора узагалі (рівень доступу за замовчуванням, відомий як package-private), його видно тільки у межах пакету, у якому оголошено клас.

На рівні членів класу використання модифікаторів *public* та неявного package-private аналогічне, як і на верхньому рівні. Для членів класу існують два додаткових модифікатори: *private* та *protected*. Модифікатор *private* визначає, що доступ до членів класу може бути здійснено тільки класом, у якому їх оголошено. Модифікатор protected визначає, що доступ до членів класу може бути отримано у межах пакету, в якому даний клас оголошено, а також у підкласові (включно із підкласами в інших пакетах).

	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

При написанні програм слід застосовувати якомога більші обмеження доступу (*private*, якщо нема потреби знижувати рівень доступу). Таокж слід уникати оголошення констант як *public*.

4. Вкладені класи (nested class)

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Мова Java дозволяє оголосити один клас усередин іншого. Класи, розташовані усередині інших класів, називають вкладеними (*nested*). У загальному випадкові вкладені класи виглядають наступним чином:

```
class OuterClass {
   ...
   class NestedClass {
      ...
   }
}
```

Вкладені (*nested*) класи поділяють на статичні (*static nested classes*) та нестатичні (*non-static nested classes*). Нестатичні класи називають також внутрішніми (*inner classes*).

```
class OuterClass {
   ...
   static class StaticNestedClass {
      ...
   }
   class InnerClass {
      ...
   }
}
```

Статичні вкладені класи, як і поля та методи класів, асоціюють із зовнішніми класами, у яких вони оголошені. Як і статичні методи, статичні класи не можуть отримувати прямий доступ до полів та методів екземплярів своїх зовнішніх класів: вони можуть використовувати їх тільки через посилання на об'єкти.

Внутрішні класи, як і поля та методи екземплярів класів, асоціюють із екземплярами зовнішніх класів. Внутрішні класи мають прямий доступ до полів та методів об'єктів своїх зовнішніх класів. Оскільки внутрішній клас асоційований із екземпляром класу, він не може бути статичним.

```
class OuterClass {
   ...
   class InnerClass {
      ...
   }
}
```

Екземпляр InnerClass може існувати тільки за умови існування екземпляру OuterClass.

Як член зовнішнього класу, внутрішній може бути оголошений як *public*, *private* чи *protected* або *package private*. Зовнішній клас може бути оголошений як *public* або *package private.*

Для створення екземпляру внутрішнього класу спочатку слід створити екземпляр його зовнішнього класу.

Екземпляр статичного вкладеного класу створюють у його зовнішньому класі наступним чином:

```
public class Outer {
   static class StaticNested{
      public void PrintHello(){
         System.out.println("hello");
      }
   }
   public static void main(String[] args) {
      StaticNested nestedObject = new StaticNested();
      nestedObject.PrintHello();
   }
}
```

Екземпляр статичного вкладеного класу створюють поза його зовнішнім класом наступним чином:

```
class Enclosing{
   static class StaticNested{
      public void PrintHello() {
         System.out.println("hello");
      }
   }
}
public class Outer {
   public static void main(String[] args) {
      Enclosing.StaticNested StaticNested = new Enclosing.StaticNested();
      StaticNested.PrintHello();
   }
}
```

Приклад реалізації внутрішнього класу:

```
class OuterClass {
   private int x;
   public OuterClass(int x){
      this.x = x;
      InnerClass z = new InnerClass();
      System.out.println(z.GetX());
   }
   class InnerClass {
      public int GetX(){
         return x;
      }
   }
}
public class Test{
   public static void main(String[] args) {
      OuterClass OClass = new OuterClass(5);
   }
}
```

Вкладені класи використовують у наступних випадках:

- коли це — спосіб логічного групування класів, які використовують тільки разом. Якщо клас використовується іншим і тільки одним класом, то логічно вбудувати його у нього;

- збільшення рівня інкапсуляції. Нехай існують два класи верхнього рівня A та B. Нехай клас B вимагає доступу до поля класу A, яке оголошено як private. У такому випадкові доцільним є оголосити клас B вкладеним у A. Оскільки B буде членом A, то отримає доступ до членів A. Крім цього, B буде схованим від решти класів;

- спрощення підтримки коду та збільшення його читабельності. Є сенс малі класи вбудовувати у класи верхнього рівня, таким чином вони будуть розташовані ближче до місця їхнього використання.

Існують два спеціальні види внутрішніх класів: локальні класи (*local classses*) та анонімні класи (*anonymous classes*).

4. Затінення (shadowing)

Якщо оголошення типу (такого, як змінна-член внутрішнього класу або ім'я параметра

методу) в особливому блоці (такому, як внутрішній клас або оголошення методу) оспівпадає за іменем з оголошенням у зовнішньому блокові, таке оголошення затінює (shadows) оголошення зовнішнього блоку. У такому випадкові слід використовувати ключове слово *this* для доступу до затіненого іменем параметра методу поля внутрішнього класу або this у поєднанні із іменем зовнішнього типу (*OuterType.this.ім'я поля)* для доступу до поля зовнішнього блоку.

```
class OuterClass{
   private int x = 1; // non-static field
   private class NestedClass{
      private int x = 2;
      public NestedClass(int x){
         System.out.println("hello from nested class constructor\n" +
               "x = " + x +
               "\nthis.x = " + this.x +
               "\nOuterClass.this.x = " + OuterClass.this.x);
      }
   }
   public OuterClass(int x){
      NestedClass nestedClassInstance = new NestedClass(x);
   }
}
public class Test {
   public static void main(String[] args) {
      OuterClass outerClassInstance = new OuterClass(3);
   }
}
```

```
hello from nested class constructor
x = 3
this.x = 2
OuterClass.this.x = 1
```

```
class OuterClass{
   private static int x = 1; // static field
   private static class NestedClass{
      private int x = 2;
      public NestedClass(int x){
         System.out.println("hello from nested class constructor\n" +
               "x = " + x +  "\nthis.x = " + this.x + "\nthis.x = " + OuterClass.x);
      }
   }
   public OuterClass(int x){
      NestedClass nestedClassInstance = new NestedClass(x);
   }
}
public class Test {
   public static void main(String[] args) {
      OuterClass outerClassInstance = new OuterClass(3);
   }
}
```

У випадкові вкладеного класу затінення може відбуватися статичним полем зовнішнього блоку чи параметром методу. Для доступу до поля класу, затіненого іменем параметру, як і у випадкові методу внутрішнього класу, використовують ключове слово *this*.

На відміну від випадку затінення поля внутрішнього класу полем зовнішнього класу, доступ до поля зовнішнього класу із вкладеного здійснюють за допомогою конструкції *OuterType.ім'я поля*.

Конструкція *OuterType.this.ім'я поля* для вкладеного класу не має сенсу, оскільки *this* — посилання на екземпляр зовнішнього класу, а його існування не є обов'язковим для вкладеного класу.

Нестатичне поле зовнішнього класу не затіняється у вкладеному класі, оскільки до такого поля не може бути доступу зі статичного блоку ні за допомогою *OuterType.this.ім'я поля, ні OuterType.ім'я поля*.

Можливості доступу до зовнішніх полів зі статичних вкладених та внутрішніх класів наведено у таблиці:

	Static outer field	Non-static outer field
Inner class	+ (*OuterType.ім'я поля, OuterType.this.ім'я поля*)	+ (*OuterType.this.ім'я поля*)
Static nested class	+ (*OuterType.ім'я поля*)	-

4. Локальні класи (local classes)

https://docs.oracle.com/javase/tutorial/java/javaOO/localclasses.html

Локальний клас – підвид внутрішнього класу. Локальними класами називають класи, оголошені усередині будь-якого блоку, позначеного фігурними дужками {}, який може складатися з нуля чи більше операторів. Зазвичай локальні класи оголошують усередині тіла методу.

Оголошують локальні класи усередині довільного блоку, наприклад: тіла методу, оператора *for*, оператора розгалуження *if* тощо.

```
public class Test {
   static String regularExpression = "[^a-zA-Z]"; // as class fiels
   public static void ComparePalindroms(String first, String second) {
      // final String regularExpression = "[^a-zA-Z]"; // as local variable
      // Java 8 SE
      // String regularExpression = "[^a-zA-Z]";
      class Palindrom {
         String formattedString = null;
         Palindrom(String symbols) {
            String currentSymbols = symbols.replaceAll(
                  regularExpression, "");
            if(currentSymbols.length() != 0) {
               formattedString = currentSymbols;
               formattedString = formattedString.toLowerCase();
            }
            int length = formattedString.length();
            int halflength = length / 2;
            for(int i = 0; i < halflength; i++)
```

```
                if(formattedString.charAt(i) != formattedString.charAt(length-i-1)){
                    formattedString = null;
                    break;
                }
            }
            String GetSymbols(){
                return formattedString;
            }
            void PrintParameters(){
                // Java 8 SE
                System.out.println("the first parameter: " + first +
                ", \nthe second parameter: " + second);
            }
        }
        Palindrom x1 = new Palindrom(first);
        Palindrom x2 = new Palindrom(second);
        // Java 8 SE
        x1.PrintParameters();
        if(x1.formattedString == null)
            System.out.println("the first parameter is not a palindrom");
        if(x2.formattedString == null)
            System.out.println("the second parameter is not a palindrom");
        if(x1.formattedString != null && x2.formattedString != null)
            if(x1.GetSymbols().equals(x2.GetSymbols()))
                System.out.println("palindroms are equal");
            else
                System.out.println("palindroms are not equal");
    }
    public static void main(String[] args) {
        ComparePalindroms("race car", "Madam, I'm Adam");
        System.out.println("-------------");
        ComparePalindroms("madam, it's adam", "Madam, I'm Adam");
        System.out.println("-------------");
        ComparePalindroms("madam, i'm adam", "Madam, I'm Adam");
    }
}
```

```
the first parameter: race car,
the second parameter: Madam, I'm Adam
palindroms are not equal
-------------
the first parameter: madam, it's adam,
the second parameter: Madam, I'm Adam
the first parameter is not a palindrom
-------------
the first parameter: madam, i'm adam,
the second parameter: Madam, I'm Adam
palindroms are equal
```

Локальні класи мають доступ до членів своїх зовнішніх класів. Також локальні класи можуть отримувати доступ до локальних змінних, які оголошені як *final*. У Java 8 SE доступ можна також отримувати до локальних змінних або параметрів зовнішнього блоку,

оголошених як *final* або *effectively final*. Змінна або параметр, чиї значення ніколи не змінюються після ініціалізації, вважають *effectively final*.

Починючи з Java 8 SE локальний клас методу може отримувати доступ до параметрів методу.

Оголошення типу (такого, як змінна) у локальному класі затінює оголошення з аналогічним іменем у зовнішньому блоці.

Локальні класи не можуть містити статичних членів. Локальні класи статичних методів можуть отримувати доступ тільки до статичних полів зовнішніх класів. Усередині локального класу не можна оголосити інтерфейс. Локальні класи можуть мати статичні члени, якщо вони є константами.

4. Анонімні класи (Anonymous classes)

https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html

Анонімні класи дозволяють оголошувати та створювати екземпляри класу одночасно. Анонімні класи схожі з локальними за вийнятком того, що не мають імені класу. Локальні класи є оголошеннями, а анонімні – виразами. Тому можуть бути оголошені у виразах.

Синтаксис виразу анонімного класу схожий на виклик конструктора за вийнятком того, що у даному випадкові це – оголошення класу, що міститься у блокові коду.

Вираз анонімного класу складається із наступних елементів:

- оператора new;
- імені інтерфейса, який імплементується або класу, який успадковується;
- дужок (), що містять параметри конструктора (як у випадкові звичайного створення екземпляру класу). У випадкові імплементації інтерфейсу конструктор відсутній, тому дужки – порожні;
- тіла, яке є тілом оголошення класу. У тілі дозволені оголошення методів, оператори – заборонені.

Оскільки оголошення анонімного класу є виразом, воно повинно бути частиною оператора (що створює екземпляр класу).

Доступ анонімних класів до локальних змінних зовнішніх блоків:

-анонімний клас має доступ до полів зовнішнього класу;

- анонімний клас не може отримувати доступ до зовнішніх локальних змінних, які не оголошені як final;
- як і у вкладених класах, оголошення типу (такого, як змінна) затінює зовнішні оголошення.

Анонімні класи мають ті ж обмеження щодо своїх членів, що і локальні:

- не можна оголосити статичного члена або інтерфейса у анонімному класі;
- анонімні класи можуть мати статичні члени, якщо вони оголошені як final.

У анонімних класах можна оголошувати:

- поля;
- додаткові методи;
-ініціалізатори екземплярів;
- локальні класи.

Проте, усередині анонімного класу не можна оголосити конструктора.

Анонімні класи використовують у наступних випадках:

- перевизначення (overriding);
- імплементація інтерфейса, потрібна одноразово;

Перевизначення класу:

```
class A {
   A(String str){
      str = str.toUpperCase();
      Print(str);
   }
   void Print(String str) {
      System.out.println("-" + str + "-");
   }
}
class B {
   B(String str){
      A a = new A(str){
         @Override
         void Print(String str) {
            System.out.println("+" + str + "+");
         }
      };
   }
}
public class Test{
   public static void main(String[] args) {
      A a = new A("hi");
      B b = new B("hi");
   }
}
```

-HI-
+HI+

Імплементація інтерфейсу:

```
interface A {
   public void Print(String str);
}
class B {
   B(String str) {
      A a = new A() {
         public void Print(String str) {
            System.out.println("+" + str + "+");
         }
      };
      a.Print(str);
   }
}
public class Test{
   public static void main(String[] args) {
      B b = new B("hi");
   }
}
```

4. Інтерфейс

http://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Визначення інтерфейса складається з модифікаторів, ключового слова interface, імені інтерфейса, розділеного комами необов'язкового списку параметрів і тіла самого інтерфейса.

У процесі розробки програмного забезпечення часом виникає потреба гарантувати обов'язкову реалізацію деяких методів одним чи кількома класами. Крім методів обов'язковими для таких класів можуть бути константи та вкладені типи. Сукупності констант, методів та вкладених типів називають інтерфейсами.

Інтерфейс може містити абстрактні методи (abstract methods), методи по замовчуванню (default methods) та статичні методи (static methods).

Оголошують інтерфейс за допомогою ключового слова interface:

```
interface Figure {
   double square();
}
```

Для імплементації інтерфейсу при оголошенні класу використовують ключове слово *implements*.

У наступному прикладі класи Rectangle, Circle та Triangle імплементують інтерфейс Figure, який містить метод *double square()*. Це означає, що кожен із цих класів повинен містити реалізацію класу *double square()*.

```
interface Figure {
   double Square();
}
class Rectangle implements Figure{
   private double length;
   private double width;
   public Rectangle(double length, double width){
      this.length = length;
      this.width = width;
   }
   public double Square(){
      return length * width;
   }
}
class Circle implements Figure{
   private double radius;
   public Circle(double radius){
      this.radius = radius;
   }
   public double Square(){
      return Math.PI * radius * radius;
   }
}
class Triangle implements Figure{
   private double a, b, c;
   public Triangle(double a, double b, double c){
      this.a = a;
```

```
        this.b = b;
        this.c = c;
    }
    public double Square(){
        double p = (a + b + c) / 2;
        return Math.sqrt(p * (p - a) * (p - b) * (p – c));
    }
}
public class Test {
    public static void main(String[] args) {
        Figure[] arr = new Figure[3];
        arr[0] = new Rectangle(10.0, 10.0);
        arr[1] = new Circle(10.0);
        arr[2] = new Triangle(1.0,1.0,1.0);
        for(Figure fig : arr)
            System.out.printf("%.3f\n", fig.Square());
    }
}
```

Специфікатор доступу *public* означає, що інтерфейс доступний будь-якому класові у будь-якому пакеті. Якщо інтерфейс оголошено без цього специфікатора, він доступний тільки у поточному пакеті.

Інтерфейси не містять описи методів.

За допомогою інтерфейсного посилання можна отримувати доступ до об'єктів класів, що реалізують інтерфейс. Класи реалізують інтерфейси, також інтерфейси можуть успадковувати інші інтерфейси.

На відміну від класів, які можуть успадковувати тільки один клас, інтерфейси можуть розширювати довільну кількість інтерфейсів, які вказують у розділеному комами спискові параметрів при оголошенні інтерфейса.

Підтримка інтерфейсу класом – це свого роду угода, яка формалізує поведінку класу.

Інтерфейс можна використати у якості типу.

4. Абстрактні методи та класи

http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html

Абстрактний клас – це клас, який оголошено за допомогою ключового слова *abstract* і містить або ні – абстрактні методи.

Не можна створити екземпляра абстрактного класу. Його можна тільки успадкувати.

Абстрактним методом називають метод, який оголошено за допомогою *abstract* без реалізації:

abstract void Print(int x);

У випадкові успадкування породжений клас зазвичай реалізує усі абстрактні методи абстрактного класу. Інакше – він теж повинен бути оголошений як абстрактний.

```
abstract class AbstractClass{
    private int y;
    public AbstractClass(int y){
```

```
        this.y = y;
    }
    public int AddX(int x){
        return y + x;
    }
    abstract void Print(int x);
}
class InhClass1 extends AbstractClass{
    public InhClass1(int y){
        super(y);
    }
    public void Print(int x){
        System.out.println(x);
    }
}
abstract class InhClass2 extends AbstractClass{
    private int x;
    public InhClass2(int x, int y){
        super(y);
        this.x = x;
    }
    public int GetX(){
        return x;
    }
}
class InhClass3 extends InhClass2{
    public InhClass3(int x, int y){
        super(x, y);
    }
    public void Print(int x){
        System.out.println(this.GetX()+x);
    }
}
public class Test {
    public static void main(String[] args) {
        InhClass1 x = new InhClass1(7);
        x.Print(x.AddX(5));
        InhClass3 y = new InhClass3(1, 2);
        y.Print(3);
    }
}
```

```
12
4
```

Методи інтерфейсу, які не оголошені як *default* або *static*, беззастережно є абстрактними. Тому немає потреби (хоча і можна) оголошувати їх як *abstract*.

Розглянемо приклад:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
   public static void main(String[] args) {
      List arr = new ArrayList();
      String o = "Hello";
      arr.add(o);
      String i  = arr.get(0); // помилка
      System.out.println(i);
   }
}
```

Результат:

Error:(8, 22) java: incompatible types: java.lang.Object cannot be converted to java.lang.String.

Метод add(E e) дозволяє використати як параметр елемент довільного типу (для якого батьківським є тип Object). Того ж типу (Object) буде і результат, котрий повертає метод get(). Тобто, додавання до ArrayList елемента типу String не означає, що буде повернуто елемент типу String. Натомість отримаємо посилання на елемент батьківського супертипу Object.

Приведемо тип елемента, котрий повертає метод get(), до потрібного нам String, скориставшись cast:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
   public static void main(String[] args) {
      List arr = new ArrayList();
      String o = "Hello";
      arr.add(o);
      String i  = (String)arr.get(0); // перетворення типу
      System.out.println(i);
   }
}
```

Результат:

Hello

Неважко помітити, що такий спосіб додавання елементів до ArrayList та отримання їх несе потенційну небезпеку виникнення помилки виконання, оскільки ніде не контролюється відповідність типів на цих двох етапах.

Змінимо тип параметра, котрий додаємо до ArrayList, на int. Зверніть увагу на те, що компілятор не зустрічає жодних помилок і код успішно компілюється. Проте, помилка виникає під час виконання і пов’язана вона із перетворенням String(Integer):

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List arr = new ArrayList();
        int o = 1;
        arr.add(o);
        String i  = (String)arr.get(0); // помилка
        System.out.println(i);
    }
}
```

Результат:

Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

Знаходження причин помилок виконання займає набагато більше часу, ніж виявлення та усунення їх на етапі компіляції.

Generics дозволяє типам (класам та інтерфейсам) бути параметрами при оголошенні класів, інтерфейсів та методів. Подібно до формальних параметрів, що використовують при оголошенні методів, параметри типу дозволяють повторно використовувати код для різних вхідних значень. Різниця у тому, що вхідні величини для локальних параметрів — це значення, а для параметрів типу — типи.

Код, що використовує generics:

- надає строгу перевірку типів при компіляції;
- дозволяє уникнути cast;
- надає можливість реалізувати узагальнені (generic) алгоритми, що працюють з колекціями різних типів, можуть бути кастомізовані, безпечніші щодо використання типів та легші для читання.

Перепишемо наш приклад з використанням generics:

```
import java.util.ArrayList;
import java.util.List;
public class Test {
    public static void main(String[] args) {
        List<String> arr = new ArrayList<>();
        String o = "Hello";
        arr.add(o);
        String i = arr.get(0); // перетворення типу
        System.out.println(i);
    }
}
```

Результат:

Hello

Узагальнені (generic) типи

Джерело

Узагальненим типом називають узагальнений клас або інтерфейс, параметрами котрого є типи. Наведений нижче приклад демонструє суть концепції.

Розглянемо клас Box, що оперує об’єктами довільного типу.

```
class Box {
   private Object object;
   public void set(Object object) {
      this.object = object;
   }
   public Object get() {
      return object;
   }
}
public class Test {
   public static void main(String[] args) {
      Box myBox = new Box();
      myBox.set("Hello");
      System.out.println(myBox.get());
      myBox.set(5);
      System.out.println(myBox.get());
      myBox.set(5.0);
      System.out.println(myBox.get());
   }
}
```

Результат:

```
Hello
5
5.0
```

Оскільки методи set() та get() оперують типом Object, параметри можуть бути довільними, проте не (int, float, double тощо) примітивними. У випадкові примітивних типів використовують класи-оболонки (детальніше: тут). Аутопакування та розпакування відбувається автоматично.

Під час компіляції неможливо встановити, як клас буде використано під час роботи програми. Одна частина програми може помістити в об’єкт Integer у той час, як інша — намагатиметься отримати з нього String. У результаті трапиться помилка виконання.

Напишемо узагальнену (generic) версію цього класу.

Generic клас оголошують наступним чином:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

Секція, що знаходиться після імені класу і обмежена знаками < та >, визначає параметр типу (також їх ще називають змінними-типами). Змінні T1, T2, ..., Tn — це змінні-типи.

Таким чином, generic-версія класу Box матиме вигляд:

```
class Box<T> {
   private T object;
   public void set(T object) {
      this.object = object;
   }
   public T get() {
      return object;
   }
}
public class Test {
   public static void main(String[] args) {
      Box<String> myBox1 = new Box<>();
      Box<Integer> myBox2 = new Box<>();
      Box<Double> myBox3 = new Box<>();
      myBox1.set("Hello");
      myBox2.set(5);
      myBox3.set(5.0);
      System.out.println(myBox1.get());
      System.out.println(myBox2.get());
      System.out.println(myBox3.get());
   }
}
```

Результат:

```
Hello
5
5.0
```

Спроба ініціалізувати поле об’єкта (myBox1, myBox2 чи myBox3) значенням невластивого йому типу викличе помилку компіляції.

Змінна-тип може бути довільним непримітивним типом: класом, інтерфейсом, масивом чи іншою змінною-типом.

Таким самий підхід застосовують до узагальнених інтерфейсів.

Угода про імена параметрів типу

За угодою імена параметрів типу — це одинарні літери верхнього регістру. Завдяки угоді імена типів відрізняються від імен змінних.

Найбільш вживані імена параметрів-типів наступні:

- E - Element (використовується у Java Collections Framework)
- K – Key (Ключ);
- N – Number (Число);

- T - Type (Тип);
- V - Value (Значення);
- S,U,V etc. - 2й, 3й, 4й типи.

Виклик та створення екземпляру узагальненого типу

Посилання на узагальнений клас (у нашому випадкові — Box) у тілі програми реалізують, попередньо здійснивши виклик узагальненого типу із заміною узагальненого параметра T на конкретне значення типу:

```
Box<String> myBox1;
Box<Integer> myBox2;
Box<Double> myBox3;
```

Виклик узагальненого типу можна розглядати як виклик звичайного методу. При цьому замість передачі аргументу у метод передають аргумент-тип (у даному прикладі: String, Integer та Double).

Зверніть увагу на пояснення різниці між аргументом-типом та параметром типу:

символ T у Foo<T> позначає параметр типу, а String у Foo<String> f — аргумент типу.

У результаті виклику узагальненого типу отримують параметризований тип.

Для створення екземпляру даного класу використовують оператор new, помістивши <тип> між іменем класу та круглими дужками:

```
Box<String> myBox1 = new Box<String>();
```

Diamond

Починаючи з Java 7 і вище можна замінити набір аргументів-типів, необхідних для виклику конструктора узагальненого класу порожнім набором аргументів-типів (<>), оскільки компілятор може визначити їх з контексту. Пару кутових дужок <> називають діамантовим оператором:

```
Box<String> myBox1 = new Box<>();
Box<Integer> myBox2 = new Box<>();
Box<Double> myBox3 = new Box<>();
```

Детальніше про Diamond.

Множинні параметри типу

Створимо ArrayList з пар елементів, кожний із яких може мати довільний тип. Пару елементів подамо як окремий generic-клас.

```
import java.util.ArrayList;
import java.util.List;
class Box<T1, T2> {
   private T1 value1;
   private T2 value2;
   Box(T1 value1, T2 value2) {
      this.value1 = value1;
      this.value2 = value2;
   }
   public T1 getValue1() {
      return value1;
   }
   public T2 getValue2() {
      return value2;
   }
}
public class Test {
   public static void main(String[] args) {
      Box<Integer, String> var1 = new Box<>(1, "Apple");
      int vBox1Val1 = var1.getValue1();
      String vBox1Val2 = var1.getValue2();
      System.out.println(vBox1Val1 + " " + vBox1Val2);
      Box<Double, Double> var2 = new Box<>(1.5, 5.5);
      Double vBox2Val1 = var2.getValue1();
      Double vBox2Val2 = var2.getValue2();
      System.out.println(vBox2Val1 + " " + vBox2Val2);
   }
}
```

Результат:

```
1 Apple
1.5 5.5
```

Описане вище стосується і інтерфейсів.

Generic-методи

Узагальненими (generic)-методами називають методи із власними параметрами типу. На відміну від узагальнених типів область видимості параметрів типу у даному випадкові визначається generic-методами, у яких вони оголошені.

Синтаксис generic-методу включає параметр типу усередині кутових дужок, котрий знаходиться перед типом, що повертає метод.

Розглянемо приклад:

```
class Util {
   public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey()) &&
            p1.getValue().equals(p2.getValue());
   }
}
class Pair<K, V> {
   private K key;
   private V value;
   public Pair(K key, V value) {
      this.key = key;
      this.value = value;
   }
   public void setKey(K key) { this.key = key; }
   public void setValue(V value) { this.value = value; }
   public K getKey()   { return key; }
   public V getValue() { return value; }
}
public class Test {
   public static void main(String[] args) {
      Pair<Integer, String> p1 = new Pair<>(1, "apple");
      Pair<Integer, String> p2 = new Pair<>(2, "pear");
      boolean same = Util.<Integer, String>compare(p1, p2);
      System.out.println(same);
      Pair<Integer, String> p3 = new Pair<>(1, "apple");
      Pair<Integer, String> p4 = new Pair<>(2, "pear");
      same = Util.compare(p3, p4);
      System.out.println(same);
   }
}
```

Результат:

```
false
false
```

Зверніть увагу на виклик методу boolean same = Util.<Integer, String>compare(p1, p2);. Компілятор може визначити типи аргументів, підставлених у відповідні параметри типу, тому немає потреби вказувати їх явно: same = Util.compare(p3, p4);.

Обмеження параметрів типу

Часом виникає потреба обмежити типи, котрі можуть бути використані як аргументи типу одним чи кількома значеннями. Наприклад, метод, що оперує числами, може вимагати як параметри тільки екземпляри Number чи його дочірніх класів.

Для оголошення обмеженого параметру типу після імені параметру типу подають ключове слово extends, після якого вказують верхню межу типу. Наприклад, у даному випадкові — Number .

Зауважте, що у даному контексті під extends розуміють як "extends" (як у випадкові

класів), так і "implements" (як для інтерфейсів). Тобто, у будь-якому разі використовують ключове слово extends.

Розглянемо приклад:

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // помилка: параметр – String!
    }
}
```

Такий код не скомпілюється, оскільки виклик integerBox.inspect("some text"); містить невірний параметр: подано String замість Number.

Змінивши "some text" на, до прикладу, 1.5, отримаєм:

```
T: java.lang.Integer
U: java.lang.Double
```

Поняття XML

Детальніше (основна офіційна інформація): https://www.w3.org/XML/
Навчальна сторінка w3c: http://www.w3schools.com/xml/xml_whatis.asp

XML (Extensible Markup Language) — це стандарт побудови мов розмітки ієрархічно структурованих даних. Є підмножиною мови SGML.

XML документи мають деревоподібну структуру, що складається із кореневого та дочірніх елементів.

Розглянемо приклад (http://www.w3schools.com/xml/xml_tree.asp):

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Даному прикладові відповідає структура, зображена на рис. 1.

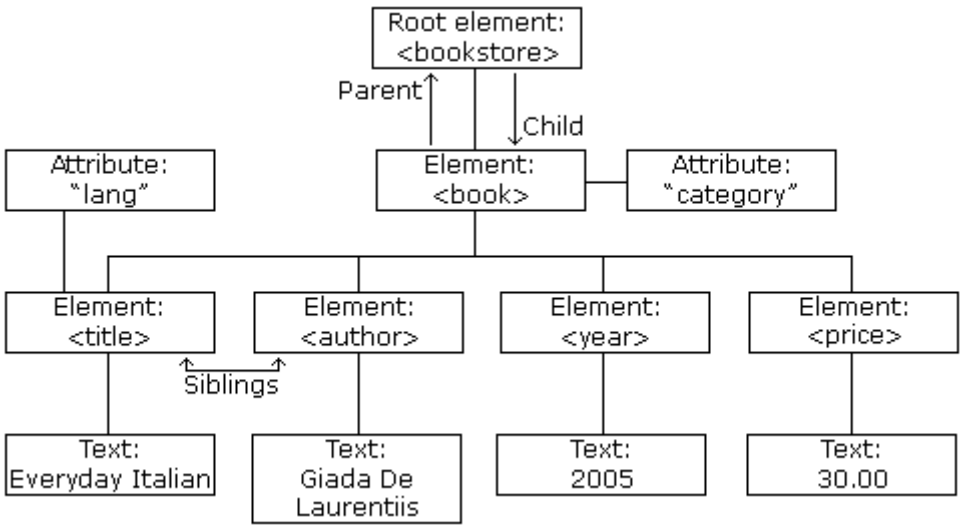


Рисунок 1 -Деревоподібна структура XML документа

Рядок <?xml version="1.0" encoding="UTF-8"?> називають прологом. Пролог є опційним. Якщо пролог існує, він повинен знаходитися на початкові XML-документа. Кодування XML-документів за замовчуванням — UTF-8, при цьому файл повинен бути збережений з кодуванням UTF-8. У разі, якщо кодування відрізняється від UTF-8, його слід явно вказати у прологові, скориставшись властивістю encoding. UTF-8 також є дефолтним кодуванням для HTML5, CSS, JavaScript, PHP та SQL.

Структура XML-документа є простою:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML-документ є деревом елементів.

Складові XML-документа

З точки зору DTD, усі XML документи складаються з наступних блоків:

- елементи (elements);
- атрибути (attributes);
- сутності (entities);
- PCDATA;
- CDATA.

Документ обов'язково повинен мати кореневий елемент. У наведеному вище документі кореневим є елемент <bookstore>.

Дочірні елементи: <book>. У свою чергу, елемент <book> має власні дочірні елементи: <title>, <author>, <year>, <price>.

Атрибути: category, lang.

Елементи

Елементи — основні складові XML документів. Елементи можуть містити текст, інші елементи або бути порожніми.

Елементом є усе, що починається з відкриваючого (включно) тегу і закінчується заключним (включно) тегом, наприклад:

```
<price>29.99</price>
```

Елемент може містити:
- текст;
- інші елементи;
- атрибути;

Кожен XML елемент повинен мати заключний тег. Пролог не має заключного тегу, оскільки не є елементом XML.

XML елементи повинні бути коректно вкладені:

```
<a><b>hello</b></a> — правильно;
<a><b>hello</a></b> — неправильно.
```

Елемент, що не має вмісту, називають пустим:

```
<element></element> або <element />
```

Порожні елементи можуть мати атрибути.

Особливості формування імена елементів:

- імена елементів — регістрозалежні;
- ім'я елемента повинно починатися з літери або символу підкреслювання;
- ім'я елемента не може починатися із будь якої комбінації літер x, m та l з довільними регістрами (XML, xMl, XMl тощо);
- ім'я елемента може містити літери, цифри, дефіси, символи підкреслювання та крапки;
- ім'я елемента не може містити пробілів.

Елемент може мати довільне ім'я (крім xml).

Імена елементів

Давайте елементам імена-описи: <person>, <firstname>, <lastname>.

Створюйте короткі та прості імена, наприклад: <book_title>, а не <the_title_of_the_book>.

Уникайте "-" (деякі програми знак мінус можуть інтерпретувати як віднімання одного значення від іншого).

Уникайте "." (деякі програми можуть інтерпретувати "first.name" як "name" - властивість об'єкта "first").

Уникайте ":". Двокрапка зарезервована для просторів імен.

Стилі імен елементів

Не існує визначених стилів імен XML елементів. Існує ряд стилів, які використовують на практиці (http://www.w3schools.com/xml/xml_elements.asp):

Стиль	Приклад	Опис
Нижній регістр	<firstname>	Усі символи — у нижньому регістрі
Верхній регістр	<FIRSTNAME>	Усі символи — у верхньому регістрі
Підкреслювання	<first_name>	Між словами — символ підкреслювання
Pascal регістр	<FirstName>	Перша літера кожного слова — у верхньому регістрі

Стиль	Приклад	Опис
Верблюжий регістр	<firstName>	Перша літера кожного, крім першого, слова — у верхньому регістрі

XML елементи — розширювані

XML елементи можуть бути розширені за потреби містити додаткову інформацію. Наприклад, нехай існує елемент:

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

Нехай виникла потреба додати до елемента <note> деяку інформацію, після чого елемент набув вигляду:

```
<note>
  <date>2008-01-10</date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Перевага XML у тому, що мова — розширювана. Парсер, орієнтований на обробку старого формату, без шкоди для програми працюватиме із новим форматом (звичай, що у цьому випадкові нові елементи будуть проігноровані).

Атрибути

Детальніше:
https://www.w3.org/TR/REC-xml/#attdecls,
http://www.w3schools.com/xml/xml_attributes.asp

XML елементи можуть мати атрибути у вигляді пар ім'я/значення. Призначення атрибутів — містити дані, що стосуються елемента.

Елементи можуть мати атрибути Значення повинні бути взяті у лапки. Можна використовувати як одинарні, так і подвійні лапки.

```
<book category="web">
```

Якщо атрибут містить подвійні лапки, можна скористатися одинарними, наприклад:

```
<gangster name='George "Shotgun" Ziegler'>
Також можна використати сутність:
<gangster name="George &amp;quot;Shotgun&amp;quot; Ziegler">
```

XML елементи vs атрибути

Розглянемо приклади:

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

У першому випадкові gender є атрибутом. У другому — елементом. Обидва приклади надають однакову інформацію. Не існує правил щодо того, коли використовувати атрибути, а коли — елементи.

Наступні три документи містять одну і ту ж інформацію. У першому date — атрибут, у другому — елемент, у третьому — елемент подано у розширеному вигляді.

Атрибут:

```
<note date="2008-01-10">
  <to>Tove</to>
  <from>Jani</from>
</note>
```

Елемент:

```
<note>
  <date>2008-01-10</date>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

Розширений елемент:

```
<note>
  <date>
    <year>2008</year>
    <month>01</month>
    <day>10</day>
  </date>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

Третій приклад є найбільш зручним з точки зору обробки.

Чи уникати атрибутів XML?

Візьміть до уваги деякі особливості використання атрибутів:
- елементи можуть володіти дочірніми елементами, атрибути — ні;
- атрибути не можуть описувати деревоподібні структури, елементи — можуть;
- елементи легко розширюються для використання у майбутньому, атрибути — ні.

Не створюйте елементів, подібних до наступного:

```
<note day="10" month="01" year="2008"
to="Tove" from="Jani" heading="Reminder"
body="Don't forget me this weekend!">
</note>
```

XML атрибути метаданих

Іноді елементи володіють ID посиланнями. Такі ID використовують для ідентифікації

(як у HTML). Наприклад:

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```

Атрибути id використовують для ідентифікації елементів note. Id не є частиною елемента note. Тобто, метадані (дані про дані) слушно представляти як атрибути замість зберігання їх у елементах.

Сутності. Екранування символів

Сутності (Entities) використовують для створення скорочень для спеціальних символів.

Сутність складається з трьох частин: символа “&”, імені сутності та символа “;”.

Включення керуючих символів у тіло xml може призвести до помилок у роботі парсера. Розв'язком є екранування таких символів. Парсер інтерпретуватиме їх як дані і не плутатиме з розміткою:

Символ	Екранування	Пояснення
<	&lt;	позначає початок тегу; екранування потрібне завжди
&	&amp;	позначає початок посилання на сутність; екранування потрібне завжди
>	&gt;	екранувати, чи ні – залежить від контексту; варто екранувати завжди
'	&apos;	обов'язково екранувати у атрибутах, визначених за допомогою одинарних лапок; варто екранувати завжди
"	&quot;	обов'язково екранувати у атрибутах, визначених за допомогою подвійних лапок; варто екранувати завжди

Варто завжди екранувати вказані символи, хоча це не завжди необхідно.

Коментарі у XML

Синтаксис коментарів у XML подібний до HTML:

```
<!-- This is a comment -->
```

Два послідовних дефіси усередині коментаря — заборонені.

```
<!-- This is a -- comment -->
```

New Line у XML

Windows-додатки зберігають новий рядок як CR + LF (carriage return +line feed);
Unix та Mac OSX: LF;
Old Mac: CR;
XML зберігає New Line як LF.

Пробіли

Послідновні пробіли у XML, на відміну від HTML, зберігаються. HTML відсікає пробільні символи і залишає тільки один.

Well Formed XML

XML документ, який відповідає синтаксичним правилам, називають добре сформованим "Well Formed" XML документом.

Що таке простір імен?

Згідно із https://www.w3.org/TR/REC-xml-names/ простір імен — це набір елементів та атрибутів, який ідентифікують за допомогою IRI (Uniform Resource Identifier, http://www.rfc-editor.org/rfc/rfc3986.txt). Такий набір часом називають словником.

Синтаксис XML дозволяє змішувати вміст різних мов розмітки. Нехай існує елемент <table>, що містить дані таблиці. Припустимо, що існує інший елемент з таким же ідентифікатором — <table>, який містить інформацію про стіл (з атрибутами - довжиною, шириною).

Оскільки елементи володіють різними списками атрибутів та мають однакові імена, парсер стикнеться із конфліктом імен.

Уникнути конфлікту можна за допомогою просторів імен. Простір імен можна уявляти як сферу, усередині якої певний елемент має певну значення (семантику).

Позначають простір імен за допомогою спеціального ідентифікатора, який є нічим іншим, як символьний рядок. Зазвичай як ідентифікатор використовують HTTP URL, хоча жодного стосунку до мережі він не має і не інтерпретується як адреса ресурсу. Ідея проста:

HTTP URL — унікальні, тому можуть бути використані як ідентифікатори.

Ім'я елемента чи атрибута складається з двох частин: простору імен та локального імені. Повне ім'я має вигляд: {namespace URI}local name.

Оголошення простору імен

Розглянемо, як використати простір імен у XML. Простір імен оголошують за допомогою атрибута xmlns.

Оголосити простір імен можна двома способами: прив'язуючи простір імен до префікса або оголосити його таким, що має значення за замовчуванням (дефолтний).

Перший спосіб виглядає наступним чином xmlns:prefix = "namespace URI", наприклад:

```
<prefix:foo xmlns:prefix = "http://prefix.com/">
    <prefix:bar />
</prefix:foo>
```

Простір імен за замовчуванням оголошують наступним чином:

```
<foo xmlns = "http://prefix.com/">
    <bar />
</foo>
```

У першому випадкові простір імен "http://prefix.com/" пов'язано із префіксом prefix. При цьому обидва імені семантично рівноправні. Вони не означають нічого. Можна, скажімо, замінити префікс “prefix” на “bar”. Це не викличе жодних семантичних змін. Дочірній елемент <bar> належить просторові імен "http://prefix.com/", про що свідчить відповідний префікс.

У другому випадкові простір імен оголошено за замовчуванням, область видимості його визначається елементом, у якому оголошено простір імен, та його вмістом. Усі елементи, що знаходяться в області видимості і не мають власного префікса, належать просторові імен за замовчуванням.

У XML атрибут, що не має префікса, не належить жодному з просторів імен. Простір імен за замовчуванням не поширюється на атрибути.

Випадок, коли простір імен оголошено за замовчуванням, видається зручнішим. У такому разі для чого існують обидва випадки оголошення?

Простори імен зазвичай прив'язують до префіксів тоді, коли їх — більше одного. Наприклад:

```
<foo xmlns = "http://prefix.com/">
    <a:bar xmlns:a = "http://prefix.com/a" />
</foo>
```

У даному прикладі елемент foo належить просторові імен "http://prefix.com/", а його дочірній елемент bar — просторові імен "http://prefix.com/a".

Простір імен можна оголосити усередині довільного елемента.

Оголошення можуть перевизначати одне одного:

```
<foo xmlns = "http://prefix.com/">
    <bar />
    <a:bar xmlns:a = "http://prefix.com/a" xmlns = "http://anotherbar.com/" />
        <bar />
    </a:bar>
</foo>
```

Тут: перший елемент <bar> належить просторові імен "http://prefix.com/", другий — "http://prefix.com/a", а третій — "http://anotherbar.com/".

СТВОРЕННЯ XML -ДОКУМЕНТА

Нехай необхідно створити наступний XML-документ:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<order id="1">
   <!--Важливе замовлення для Івасика-Телесика-->
   <person>Івасик-Телесик</person>
   <item id="1" status="delivered">
      <title>Гуси-лебеді</title>
      <quantity>5</quantity>
   </item>
</order>
```

Код, який його формує:

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Attr;
import org.w3c.dom.Comment;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import java.io.File;
public class dom {
   public static void main(String argv[]) {
      try {
         DocumentBuilderFactory dbFactory =
               DocumentBuilderFactory.newInstance();
         DocumentBuilder dBuilder =
               dbFactory.newDocumentBuilder();
         Document doc = dBuilder.newDocument();
         // root element
         Element rootElement = doc.createElement("order");
         doc.appendChild(rootElement);
         // setting attribute to element
         Attr attr = doc.createAttribute("id");
         attr.setValue("1");
         rootElement.setAttributeNode(attr);
         Comment comment = doc.createComment("Важливе замовлення для Івасика-
Телесика");
         rootElement.appendChild(comment);
         Element elementPerson = doc.createElement("person");
         rootElement.appendChild(elementPerson);
```

```
            elementPerson.appendChild(
                    doc.createTextNode("Івасик-Телесик"));
            Element elementItem = doc.createElement("item");
            rootElement.appendChild(elementItem);
            attr = doc.createAttribute("id");
            attr.setValue("1");
            elementItem.setAttributeNode(attr);
            attr = doc.createAttribute("status");
            attr.setValue("delivered");
            elementItem.setAttributeNode(attr);
            Element elementTitle = doc.createElement("title");
            elementItem.appendChild(elementTitle);
            elementTitle.appendChild(
                    doc.createTextNode("Гуси-лебеді"));
            Element elementQuantity = doc.createElement("quantity");
            elementItem.appendChild(elementQuantity);
            elementQuantity.appendChild(
                    doc.createTextNode("5"));
            TransformerFactory transformerFactory =
                    TransformerFactory.newInstance();
            Transformer transformer =
                    transformerFactory.newTransformer();
            transformer.setOutputProperty(OutputKeys.INDENT, "yes");
            transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");
            DOMSource source = new DOMSource(doc);
            StreamResult result =
                    new StreamResult(new File("output.xml"));
            transformer.transform(source, result);
            StreamResult consoleResult =
                    new StreamResult(System.out);
            transformer.transform(source, consoleResult);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

ПАРСИНГ XML-ДОКУМЕНТА. DOM-ПАРСЕР

Нехай існує деякий XML-документ (input.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<order id="1">
    <!--Важливе замовлення для Івасика-Телесика -->
    <person>Івасик-Телесик</person>
    <item>
        <title>Гуси-лебеді</title>
        <quantity>5</quantity>
    </item>
    <item>
        <title>Яблука</title>
        <quantity>10</quantity>
    </item>
</order>
```

Скористаємося DOM-парсером для отримання доступу до елементів та атрибутів:

```
import java.io.File;
import
javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder; import
org.w3c.dom.*; public class dom {
   public static void main(String[] args) {
try {
        File inputFile = new File("input.xml");
        DocumentBuilderFactory dbFactory
            = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();
System.out.println("Кореневий елемент: "
            + doc.getDocumentElement().getNodeName());
        NodeList nList = doc.getDocumentElement().getChildNodes();
        System.out.println("Кількість дочірніх вузлів: " +
nList.getLength());          System.out.println("----------------------------");
for (int i = 0, l = nList.getLength(); i < l; i++) {
            Node nNode = nList.item(i);
            System.out.print("Child node #" + i + " - " + nNode.getNodeName() + ": ");
System.out.println(nNode.getTextContent());
        }
     } catch (Exception e)
{           e.printStackTrace();
     }
   }
}
```

Результат:

```
Кореневий елемент: order
Кількість дочірніх вузлів: 9
```

```
--------------------------Child
node #0 - #text:

Child node #1 - #comment: Важливе замовлення для Івасика-Телесика
Child node #2 - #text:

Child node #3 - person: Івасик-Телесик
Child node #4 - #text:

Child node #5 - item:
            Гуси-лебеді
            5

Child node #6 - #text:

Child node #7 - item:
            Яблука
            10
```

Зверніть увагу на те, що як відступи у xml-документові використано табуляції. Замість табуляції можна використати для відступів 4 (загальноприйняте значення, за потреби можна встановити іншу кількість) послідовних пробіли. Для зміни способу організації відступів скористайтесь меню Edit → Convert Indents →To Spaces/To Tabs. Налаштувати відступи можна з меню File → Settings → Editor → Code Style → XML (https://www.jetbrains.com/idea/help/code-style-xml.html#d1227290e130).

Приберемо із input.xml форматування, записавши його одним рядком:

```
<?xml version="1.0" encoding="UTF-8"?><order id="1"><!--Важливе замовлення для Івасика-Телесика --><person>Івасик-Телесик</person><item><title>Гуси-лебеді</title><quantity>5</quantity></item><item><title>Яблука</title><quantity>10</quantity></item></order>
```

Результат відрізнятиметься від отриманого щойно:

```
Кореневий елемент: order
Кількість дочірніх вузлів: 4
---------------------------
Child node #0 - #comment: Важливе замовлення для Івасика-Телесика
Child node #1 - person: Івасик-Телесик
Child node #2 - item: Гуси-лебеді5
Child node #3 - item: Яблука10
```

Легко помітити, що результати відрізняються наявністю вузлів #text. У даному випадкові ці вузли містять символ переходу на новий рядок. Змініть документ таким чином, щоб він містив наступне: <order id="1">**hello**<!-- . Результат включатиме вузол #text:

```
Кореневий елемент: order
Кількість дочірніх вузлів: 5
---------------------------
Child node #0 - #text: hello
Child node #1 - #comment: Важливе замовлення для Івасика-Телесика
Child node #2 - person: Івасик-Телесик
Child node #3 - item: Гуси-лебеді5
Child node #4 - item: Яблука10
```

#text — це результат виконання getNodeName() над порожнім вузлом. Позбутися таких вузлів можна різними способами. Модифікуємо код прикладу:

```
for (int i = 0, l = nList.getLength(); i < l; i++) {
   Node nNode = nList.item(i);
if(nNode instanceof Text) {
      String value = nNode.getNodeValue().trim();
if (value.equals("")) continue;
   }
   System.out.print("Child node #" + i + " - " + nNode.getNodeName() + ":
");     System.out.println(nNode.getTextContent()); }
```

Отримаємо:

```
Кореневий елемент: order
Кількість дочірніх вузлів: 9
---------------------------
Child node #1 - #comment: Важливе замовлення для Івасика-
Телесика Child node #3 - person: Івасик-Телесик Child node #5 - item:
      Гуси-лебеді
      5
Child node #7 - item:
      Яблука
      10
```

Якщо потрібно розглянути тільки дочірні елементи, можна скористатися наступним підходом:

```
for (int i = 0, l = nList.getLength(); i < l; i++)
{     Node nNode = nList.item(i);
   if(nNode.getNodeType() == nNode.ELEMENT_NODE) {
      System.out.print("Child node #" + i + " - " + nNode.getNodeName() + ": ");
System.out.println(nNode.getTextContent());
   }
}
```

У цьому випадкові взято до уваги тільки дочірні вузли — елементи. Вузли іншого типу проігноровані.

Значення nodeName, nodeValue та attributes залежать від типу вузла. Детальніше: https://docs.oracle.com/javase/ 8 /docs/api/org/w3c/dom/Node.html.

Interface	nodeName	nodeValue	**атрибути**
Attr	Те саме, що і Attr.name	Те саме, що і Attr.value	null
CDATASection	"#cdata-section"	Те саме, що і CharacterData.data, вміст секції CDATA	null
Comment	"#comment"	Те саме, що і CharacterData.data, вміст коментаря	null
Document	"#document"	null	null
DocumentFragment	"#document-fragment"	null	null
DocumentType	Те саме, що і DocumentType.name	null	null
Element	Те саме, що і Element.tagName	null	Named NodeMap
Entity	Ім'я сутності	null	null
EntityReference	Ім'я сутності, на яку посилаються	null	null
Notation	Ім'я нотації	null	null

Interface	nodeName	nodeValue	атрибути
Processing Instruction	Те саме, що і ProcessingInstruction.target	Те саме, що і ProcessingInstruction.data	null
Text	"#text"	Те саме, що і CharacterData.data, вміст текстового вузла	null

Метод getTextContent() повертає текстовий вміст даного вузла та його нащадків. Повернутий рядок не містить розмітки. Результат залежить від типу вузла.

Тип вузла	Коментар
ELEMENT_NODE, ATTRIBUTE_NODE, ENTITY_NODE, ENTITY_REFERENCE_NODE, DOCUMENT_FRAGMENT_NODE	поєднання значення атрибута textContent кожного дочірнього взула за винятком вузлів COMMENT_NODE та PROCESSING_INSTRUCTION_NODE. Якщо вузол не має нащадків, повертається пустий рядок.
TEXT_NODE, CDATA_SECTION_NODE, COMMENT_NODE, PROCESSING_INSTRUCTION_NODE	nodeValue
DOCUMENT_NODE, DOCUMENT_TYPE_NODE, NOTATION_NODE	null

У наступному прикладі отримують доступ до значень атрибутів елемента. Скористаємося методом getAttributes(), для цього дещо модифікуємо XML-документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<order id="1">
   <!--Важливе замовлення для Івасика-Телесика -->
   <person>Івасик-Телесик</person>
   <item id="1" status="delivered">
      <title>Гуси-лебеді</title>
      <quantity>5</quantity>
   </item>
   <item id="2" status="delivered">
      <title>Яблука</title>
      <quantity>10</quantity>
   </item>
</order>
```

Вище до елементів <item> додано атрибути id та status. Зверніть увагу, що відступи організовано за допомогою пробілів (1 відступ — 4 пробіли). На роботу парсера це не впливає.

```
import java.io.File;
import
javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder; import
org.w3c.dom.*; public class dom {
   public static void main(String[] args) {
try {
        File inputFile = new File("input.xml");
        DocumentBuilderFactory dbFactory
```

```
                = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();
System.out.println("Кореневий елемент: "
                + doc.getDocumentElement().getNodeName());
        NodeList nList = doc.getDocumentElement().getChildNodes();
        System.out.println("Кількість дочірніх вузлів: " +
nList.getLength());          System.out.println("----------------------------");
for (int i = 0, l = nList.getLength(); i < l; i++) {              Node nNode =
nList.item(i);
            if(nNode.getNodeType() == nNode.ELEMENT_NODE) {
               System.out.println("Child node #" + i + " - " + nNode.getNodeName());
               NamedNodeMap chNodes = nNode.getAttributes();
for (int j = 0, len = chNodes.getLength(); j < len; j++) {
                  Node chNode = chNodes.item(j);
                  System.out.println(chNode.getNodeName() + ": " +
chNode.getTextContent());
               }
            }
        }
      } catch (Exception e)
{           e.printStackTrace();
      }
   }
}
```

Результат:

```
Кореневий елемент: order
Кількість дочірніх вузлів: 9
----------------------------
Child node #3 - person
Child node #5 - item
id: 1
status: delivered
Child node #7 - item
id: 2
status: delivered
```

Запити у XML-документах

Нехай дано XML-документ:

```
<?xml version="1.0" encoding="UTF-8"?>
<order id="1">
   <!--Важливе замовлення для Івасика-Телесика -->
   <person>Івасик-Телесик</person>
   <item id="1" status="delivered">
      <title>Гуси-лебеді</title>
      <quantity>5</quantity>
   </item>
```

```
  <item id="2" status="delivered">
    <title>Яблука</title>
    <quantity>10</quantity>
  </item>
</order>
```

Завдання полягає у тому, щоб знайти у ньому елементи <item>, їхні атрибути та відповідні дочірні елементи і їхні текстові значення.

```
import java.io.File;
import
javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder; import
org.w3c.dom.*; public class dom {
   public static void main(String[] args) {
try {
        File inputFile = new File("input.xml");
        DocumentBuilderFactory dbFactory
            = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(inputFile);
doc.getDocumentElement().normalize();
System.out.println("Кореневий елемент: "
            + doc.getDocumentElement().getNodeName());
        NodeList nList = doc.getElementsByTagName("item");
        // якщо потрібно вибрати усі елементи, скористайтесь
        // параметром методу getElementsByTagName("*");
        System.out.println("Кількість елементів <item>: " +
nList.getLength());          System.out.println("----------------------------");
for (int i = 0, l = nList.getLength(); i < l; i++) {
            Node nNode = nList.item(i);
            Element element = (Element) nNode;
            String id = element.getAttribute("id");
            String status = element.getAttribute("status");
            System.out.println("id: " + id);
            System.out.println("status: " + status);
NodeList childTitleElements = element.getElementsByTagName("title");
            String childTitle = childTitleElements.item(0).getTextContent();
            System.out.println("title: " + childTitle);
            NodeList childQuantityElements = element.getElementsByTagName("quantity");
            String childQuantity = childQuantityElements.item(0).getTextContent();
            System.out.println("quantity: " + childQuantity);
System.out.println("----------");
        }
      } catch (Exception e)
{          e.printStackTrace();
      }
   }
}
```

Результат:

```
Кореневий елемент: order
Кількість елементів <item>: 2
---------------------------
id: 1 status: delivered
title: Гуси-лебеді
quantity: 5 ---------id:
2 status: delivered
title: Яблука quantity:
10
----------
```

JSON

Детальніше:
http://www.json.org/, http://json.org/example.html, http://www.w3schools.com/json/default.asp

JSON — формат обміну даними, повний текст можна завантажити за посиланням: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

JSON читабельний та простий для написання людиною, легкий для створення та обробки машиною. В основі — підмножина мови програмування JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON — це текстовий формат, що не залежить від мов програмування, проте використовує погодження, відомі програмістам мов родини C, включаючи C++, C#, Java, JavaScript, Perl, Python та багатьох інших.

JSON побудовано на двох структурах:

- колекції пар ім'я/значення. У різних мовах виглядає як об'єкт, запис, структура, словник, хеш-таблиця, список з ключами чи асоціативний масив;
- впорядкований список значень. У більшості мов виступає як масив, вектор, список чи послідовність.

Це — універсальні структури даних. Практично усі сучасні мови програмування підтримують їх у тій чи іншій формі. Відповідно, слушною є побудова формату обміну даними між різними мовами таким, в основі якого закладено саме ці структури.

У JSON вони набувають однієї із наступних форм.

Об'єкт (object), який є невпорядкованим набором пар ім'я/значення. Об'єкт починають з { (left brace) та закінчують з } (right brace). Після кожного імені повинна бути двокрапка : (colon), а пари ім'я/значення розмежовують за допомогою коми , (comma).

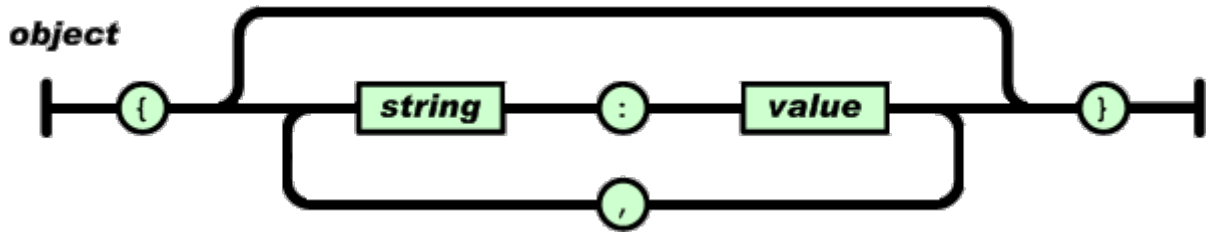


Рисунок 1 — Object у JSON

Масив (array) — упорядкована колекція значень. Масив починають з [(left bracket) і закінчують з] (right bracket). Значення розмежовують за допомогою коми , (comma).

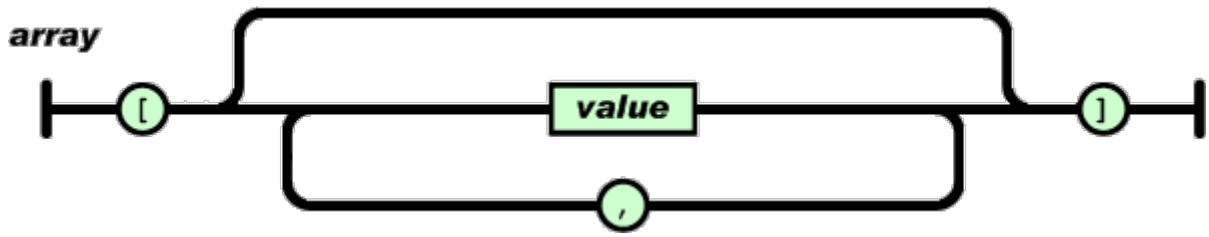


Рисунок 2 — Array у JSON

Значення (*value)* може бути

- рядком (string*)* у подвійних лапках;
- числом (*number);*

- true чи false;
- об'єктом (*object);*
- масивом (*array)*.
- null.

Ці структури можуть бути вкладеними.

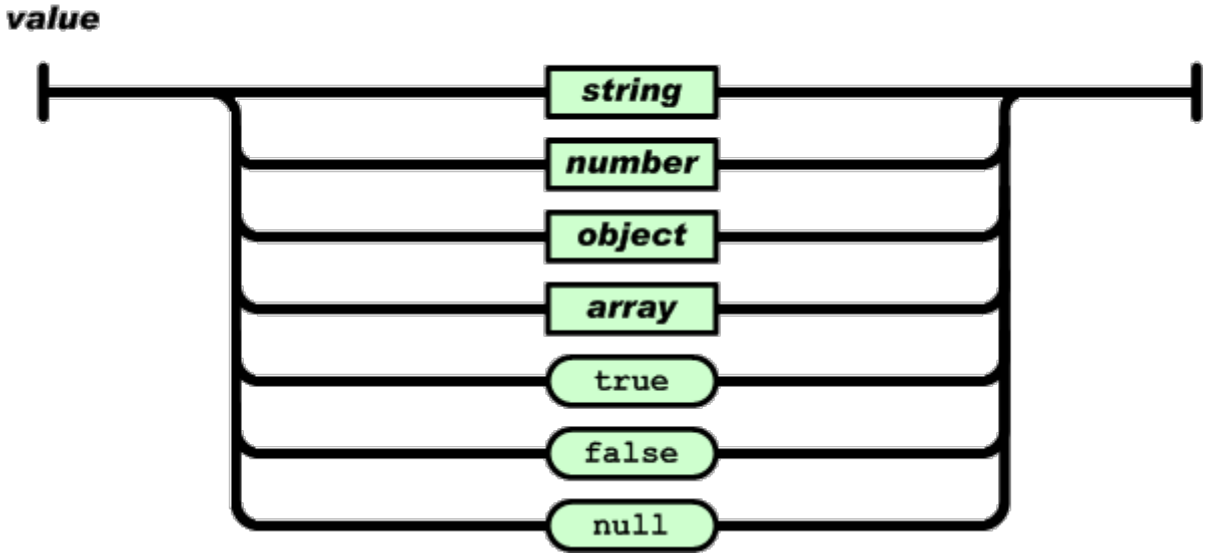


Рисунок 3 — Value у JSON

Рядок (string*)* — послідовність з нуля та більше Unicode-символів у подвійних лапках. Для екранування використовують символи \ (backslash escapes). Символ (character) представляють як односимвольний рядок. String дуже схожий на C чи Java String.

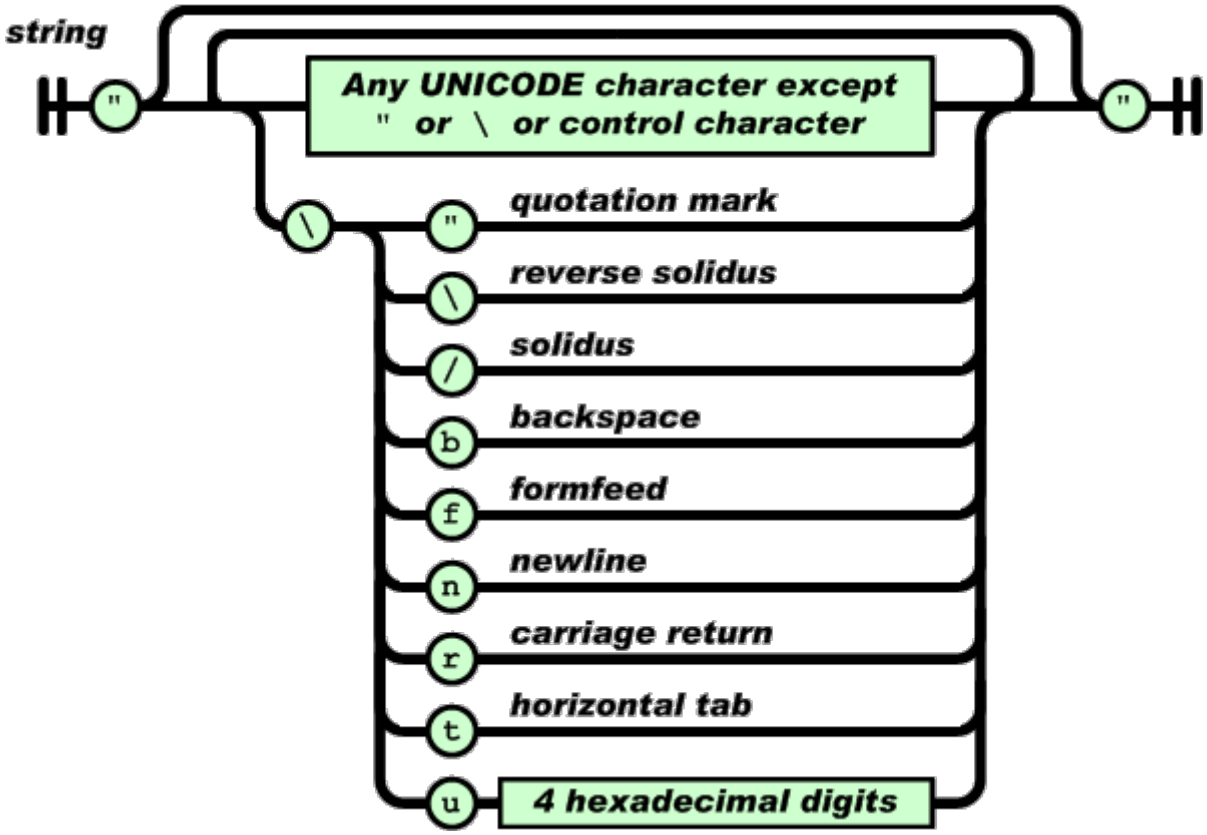


Рисунок 4 — String у JSON

Число (number*)* дуже подібне до C чи Java-чисел. Числа у вісімковій чи шістнадцятковій системах числення у JSON не використовують.

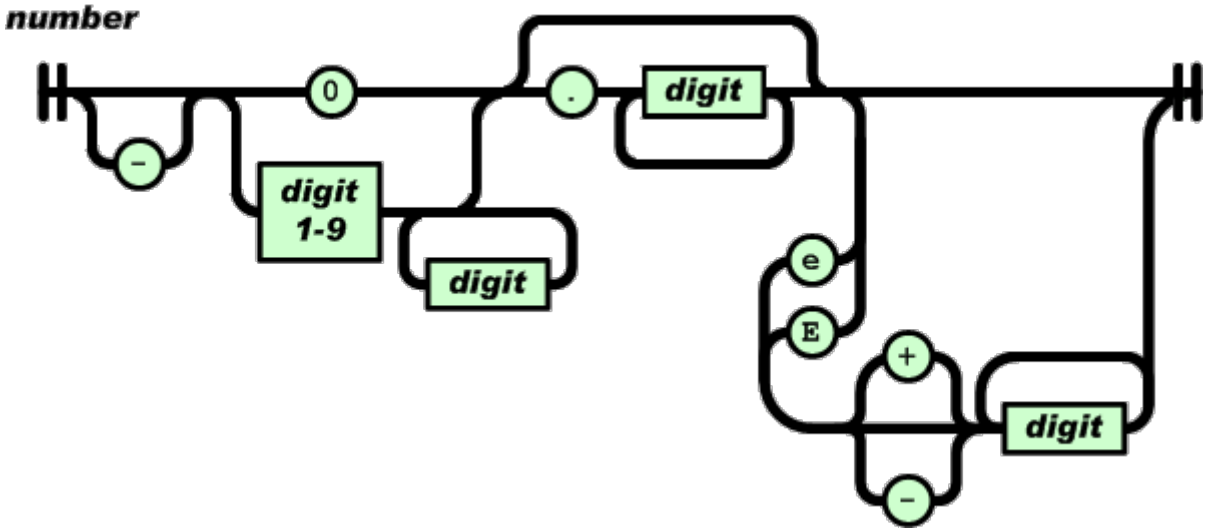


Рисунок 5 — Number у JSON

СЕРІАЛІЗАЦІЯ ТА ДЕСЕРІАЛІЗАЦІЯ JSON

Розглянемо приклад. Нехай у проекті існує клас (файл Foo.java):

```
public class Foo {
   private int width;
   private int height;
   private int depth;
   public Foo(int width, int height, int depth){
      this.width = width;
      this.height = height;
      this.depth = depth;
   }
}
```

Нехай існує екземпляр класу Foo:

```
Foo fooInstance = new Foo(1, 2, 3);
```

Завдання: представити вміст полів екземпляра класу Foo у форматі JSON. Забезпечити зворотне перетворення: створити екземпляр класу Foo, маючи рядок у форматі JSON, котрий задає значення полів екземпляра класу Foo.

Для роботи необхідно завантажити і встановити бібліотеку серіалізації Java — GSON: проект на GitHub. Gson 2.6.1 Download на Maven Central. Версія проекту на момент написання цього файлу: 2.6.1.

GSON:

- надає простих toJson() та fromJson() методів для перетворення Java-об'єктів у формат JSON і навпаки;
- дозволяє перетворювати у- та з формату JSON вже існуючі незмінювані об'єкти;
- підтримує Java Generics;
- дозволяє налаштовувати представлення об'єктів;
- підтримує об'єкти довільної складності (з глибокою ієрархією та широким використанням generic-типів).

Створіть проект. Виберіть у IntelliJ IDEA пункт меню File -> Project Structure:

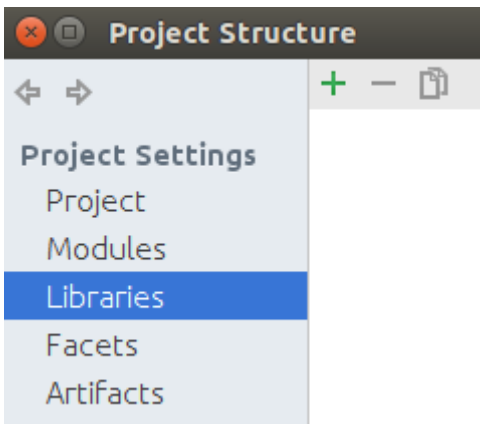


Рисунок 1 — Project Structure

Виберіть з меню Project Settings пункт Libraries. Натисніть “+”. З випадаючого списку New Project Library виберіть Java і вкажіть шлях до попередньо збереженого файлу gson-2.6.1.jar (чи новішого).

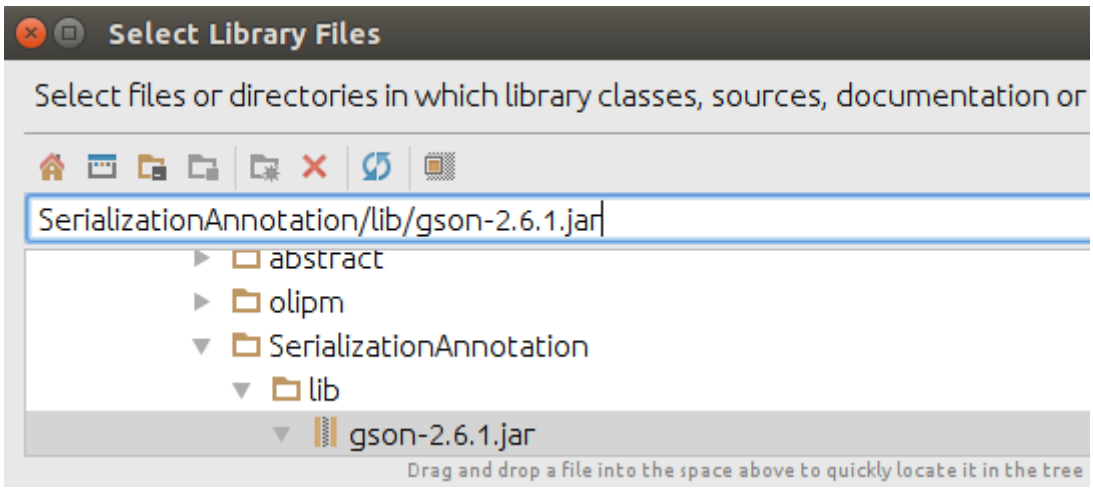


Рисунок 2 — Додавання бібліотеки до проекту уручну

Бібліотеку можна додати без попереднього завантаження, за допомогою Maven. Дл цього виберіть з меню Project Settings пункт Libraries. Натисніть “+”. З випадаючого списку New Project Library виберіть пункт From Maven.

Введіть у поле пошуку рядок com.google.code.gson:gson:2.6.1 (за потреби — вкажіть потрібну версію) і натисніть кнопку пошуку. Відзначивши Download to, можна вказати шлях для завантаження бібліотеки. Нехай, наприклад, це буде папка lib проекту.

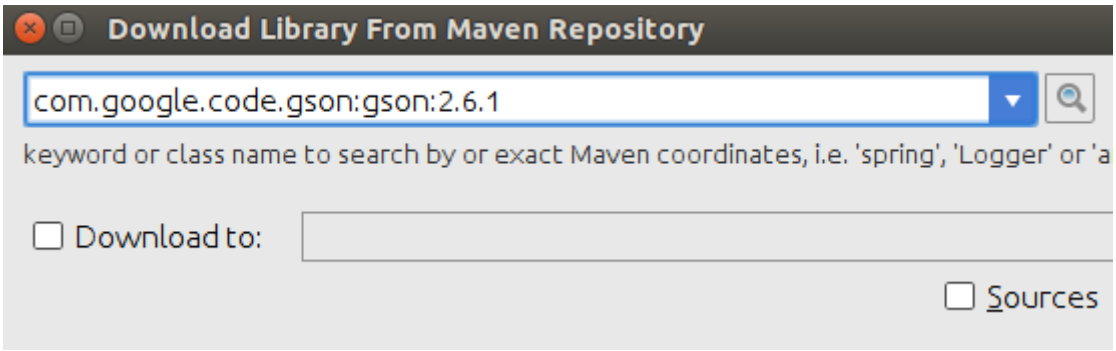


Рисунок 3 — Завантаження бібліотеки з Maven репозиторію

Створіть у проекті клас із довільним іменем, нехай — Bar:

```
import com.google.gson.Gson;
public class Bar {
   public static String serializeToJson(Foo fooInstance) {
      Gson gson = new Gson();
      String j = gson.toJson(fooInstance);
      return j;
   }
   public static Foo deserializeFromJson(String jsonString) {
```

```
        Gson gson = new Gson();
        Foo myBox = gson.fromJson(jsonString, Foo.class);
        return myBox;
    }
    public static void main(String[] args) {
        Foo fooInstance = new Foo(1, 2, 3);
        System.out.println(serializeToJson(fooInstance));
        fooInstance = deserializeFromJson("{\"width\":4,\"height\":5,\"depth\":6}");
        System.out.println(serializeToJson(fooInstance));
    }
}
```

Результат:

```
{"width":1,"height":2,"depth":3}
{"width":4,"height":5,"depth":6}
```

Клас Bar містить два статичних методи: public static String serializeToJson(Foo fooInstance) та public static Foo deserializeFromJson(String jsonString).

Перший як параметр приймає посилання на екземпляр класу Foo і повертає рядок у форматі JSON, другий — навпаки: працює із рядком-параметром у форматі JSON і повертає посилання на екземпляр класу Foo.

Зверніть увагу на те, що поля у JSON форматі мають ті ж назви, що і поля класу. За потреби їх можна звільнити на довільні за допомогою анотацій. Змінимо клас Foo:

```
import com.google.gson.annotations.SerializedName;
public class Foo {
    @SerializedName("w")
    private int width;
    @SerializedName("h")
    private int height;
    @SerializedName("d")
    private int depth;
    public Foo(int width, int height, int depth){
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
}
```

Відповідно, у класі Bar змінимо параметр методу deserializeFromJson

```
з ("{\"width\":4,\"height\":5,\"depth\":6}"); на ("{\"w\":4,\"h\":5,\"d\":6}");
```

Результат:

```
{"w":1,"h":2,"d":3}
```

```
{"w":4,"h":5,"d":6}
```

Розглянемо випадок вибіркової серіалізації полів класу (детальніше: https://google-gson.googlecode.com/svn/trunk/gson/docs/javadocs/com/google/gson/annotations/Expose.html).

Скористаємося типом анотації Expose. Дана анотація вказує на те, що член класу підлягає серіалізації або десеріалізації:

> @Expose(serialize = true) або @Expose() використовують, якщо член класу повинен бути серіалізований або десеріалізований.
>
> @Expose(serialize = false) — член класу не підлягає серіалізації або десеріалізації.

Аналогічно записують для десеріалізації (замість serialize — deserialize; обох можна поєднувати за допомогою коми). Значення за замовчуванням — true.

Така анотація не дасть ефекту, доки екземпляр Gson не буде побудовано з GsonBuilder та виконано метод GsonBuilder.excludeFieldsWithoutExposeAnnotation().

Таким чином, класи Foo та Bar виглядатимуть наступним чином.

```
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;
public class Foo {
   @SerializedName("w")
   @Expose
   private int width;
   @SerializedName("h")
   @Expose(serialize = true, deserialize = true)
   private int height;
   @SerializedName("d")
   @Expose(serialize = false, deserialize = false)
   private int depth;
   public Foo(int width, int height, int depth){
      this.width = width;
      this.height = height;
      this.depth = depth;
   }
}
```

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
public class Bar {
   public static String serializeToJson(Foo fooInstance) {
      Gson gson = new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();
      String j = gson.toJson(fooInstance);
      return j;
   }
```

```
    public static Foo deserializeFromJson(String jsonString) {
        Gson gson = new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create();
        Foo myBox = gson.fromJson(jsonString, Foo.class);
        return myBox;
    }
    public static void main(String[] args) {
        Foo fooInstance = new Foo(1, 2, 3);
        System.out.println(serializeToJson(fooInstance));
        fooInstance = deserializeFromJson("{\"w\":4,\"h\":5,\"d\":6}");
        System.out.println(serializeToJson(fooInstance));
    }
}
```

Результат:

```
{"w":1,"h":2}
{"w":4,"h":5}
```

Такий підхід корисний, якщо потрібно явно вказати опції серіалізації для кожного члена класу окремо. Якщо потрібно тільки виключити певне поле з серіалізації/десеріалізації, достатньо позначити його як transient, наприклад:

```
private transient int depth;
```

Модифікуємо класи Foo та Bar.

```
import com.google.gson.annotations.Expose;
import com.google.gson.annotations.SerializedName;
public class Foo {
    @SerializedName("w")
    @Expose
    private int width;
    @SerializedName("h")
    @Expose(serialize = true, deserialize = true)
    private int height;
    private transient int depth;
    public Foo(int width, int height, int depth){
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
}
```

```
import com.google.gson.Gson;
public class Bar {
   public static String serializeToJson(Foo fooInstance) {
      Gson gson = new Gson();
      String j = gson.toJson(fooInstance);
      return j;
   }
   public static Foo deserializeFromJson(String jsonString) {
      Gson gson = new Gson();
      Foo myBox = gson.fromJson(jsonString, Foo.class);
      return myBox;
   }
   public static void main(String[] args) {
      Foo fooInstance = new Foo(1, 2, 3);
      System.out.println(serializeToJson(fooInstance));
      fooInstance = deserializeFromJson("{\"w\":4,\"h\":5,\"d\":6}");
      System.out.println(serializeToJson(fooInstance));
   }
}
```

Результат:

```
{"w":1,"h":2}
{"w":4,"h":5}
```

СОРТУВАННЯ ОБ’ЄКТІВ JAVA

У Java існують два інтерфейси, призначені для порівняння об’єктів: Comparable та Comparator.
Comparable об’єкт може порівнювати себе із іншими об’єктами. Для порівняння своїх екземплярів клас повинен імплементувати інтерфейс Comparable.
Компаратор-об’єкт (Comparator) може порівнювати два різних об’єкти. Клас порівнює не власні екземпляри, а екземпляри інших класів. Клас компаратора повинен імплементувати інтерфейс Comparator.
Кожен із інтерфейсів має метод, котрий повинен бути імплементований.
java.lang.Comparable: int compareTo(Object o1) — метод порівнює поточний екземпляр класу із об’єктом o1 і повертає один із можливих результатів:

- додатній – даний об’єкт більший за o1;
- 0 – даний об’єкт та o1 – однакові;
- від’ємний – даний об’єкт менший за o1.

java.util.Comparator: int compare(Object o1, Objecto2) – метод порівнює об’єкти o1 та o2 і повертає один із можливих результатів:

- додатній – об’єкт o1 більший за o2;
- 0 – об’єкти o1 та o2 – однакові;
- від’ємний – об’єкт o1 менший за o2.

Методи java.util.Collections.sort(List) та java.util.Arrays.sort(Object[]) можуть бути використані для природного сортування об’єктів.
Методи java.util.Collections.sort(List, Comparator) and java.util.Arrays.sort(Object[], Comparator) можуть бути використані для порівняння об’єктів за наявності відповідного об’єкта Comparator.

Нехай, скажімо, існує клас, котрий має два поля: int id та int x. Скажімо, природнім сортуванням вважатимемо сортування на основі порівнянь за полем id. Для такого сортування можна скористатися інтерфейсом Comparable. Уявімо, що виникла потреба сортування за полем x. Можна змінити реалізацію методу compareTo(). При цьому втратиться попередня реалізація порівняння за полем id. У такому випадкові слід створити Comparator.

Сортування Object[] o

```
import java.util.Arrays;
public class Main {
   public static void main(String[] args) {
      int[] arr = {32, 2};
      Arrays.sort(arr);
      arr.toString();
      for (int var : arr) {
         System.out.println(var);
      }
   }
}
```

Результат:

```
2
32
```

У даному прикладі масив arr містить елементи примітивного типу (int). Для int існує відповідний клас-обгортка Integer, котрий імплементує інтерфейс Comparable. Батьківським класом для Int є клас Object. Змінимо тип елементів масиву на String та ініціалізуємо його елементи значеннями відповідного типу:

```
import java.util.Arrays;
public class Main {
   public static void main(String[] args) {
      String[] arr = {"world", "hello"};
      Arrays.sort(arr);
      arr.toString();
      for (String var : arr) {
         System.out.println(var);
      }
   }
}
```

Результат:

```
hello
world
```

Arrays.sort(Object o) сортує елементи String за рахунок того, що клас String імплементує інтерфейс Comparable. Батьківським класом для String є клас Object.

Сортування об’єктів за допомогою Comparable

Створимо клас, який імплементує інтерфейс Comparable. При цьому клас повинен реалізувати метод int compareTo(T o):

```
import java.util.Arrays;
class Data implements Comparable<Data> {
   private int x;
   public Data(int x) {
      this.x = x;
   }
   @Override
   public int compareTo(Data o) {
      int result = 0;
      if (x < o.x)
         result = -1;
      else if (x > o.x)
         result = 1;
      return result;
   }
   public int getX() {
      return x;
   }
```

```
}
public class Main {
    public static void main(String[] args) {
        Data var1 = new Data(123);
        Data var2 = new Data(2);
        Data var3 = new Data(321);
        Data[] arr = {var1, var2, var3};
        Arrays.sort(arr);
        arr.toString();
        for (Data var : arr) {
            System.out.println(var.getX());
        }
    }
}
```

Результат:

2
123
321

Напрям сортування можна змінити, помінявши місцями -1 та 1 у методі compareTo(Data o). Оскільки тип поля x класу Data – int, то порівняння можна реалізувати наступним чином:

```
@Override
public int compareTo(Data o) {
    return Integer.compare(x, o.getX());
}
```

У такому випадкові елементи будуть посортовані за зростанням.

Сортування List

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> lst = new ArrayList<>();
        lst.add(11);
        lst.add(5);
        lst.add(77);
        System.out.println(lst.toString());
        Collections.sort(lst);
        System.out.println(lst.toString());
```

```
    }
}
```

Результат:

[11, 5, 77]
[5, 11, 77]

При додаванні елементів до ArrayList у даному випадкові використовується autoboxing, оскільки замість типу Integer параметри методу add() мають тип примітивний тип int. Елементи ArrayList у даному випадкові сортуються за допомогою методу sort() класу Collections за рахунок того, що клас Integer імплементує інтерфейс Comparable.

Сортування об’єктів за допомогою Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Comparator;
class Data {
    private int x;
    public Data(int x) {
        this.x = x;
    }
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
}
class CustomComparator implements Comparator<Data> {
    @Override
    public int compare(Data o1, Data o2) {
        return ((Integer) o1.getX()).compareTo(o2.getX());
    }
}
public class Main {
    public static void main(String[] args) {
        Data d1 = new Data(4);
        Data d2 = new Data(1);
        Data d3 = new Data(0);
        List<Data> arr = new ArrayList<>();
        arr.add(d1);
        arr.add(d2);
        arr.add(d3);
        Collections.sort(arr, new CustomComparator());
        for (Data element : arr) {
```

```
            System.out.println(element.getX());
        }
    }
}
```

Результат:

0
1
4

Регулярні вирази

https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html
https://docs.oracle.com/javase/tutorial/essential/regex/

Тестування зручно проводити на сайті https://regex101.com.

Регулярний вираз – це шаблон, котрий описує деякий текст. Наприклад, шаблон “regexp” відповідає аналогічному наборові символів у тексті:

```
"/regexp/", "regexp"
```

Розділяючі символи

Використовуючи PCRE-вирази, їх оточують на початкові та укінці символами-роздільниками. Ними можуть бути довільні символи окрім числобуквених, пробільних та символу зворотної скісної риски. Часто використоувють пряму скісну риску (/), гратку (#) чи тильду (~). Наприклад: "/regexp/".

Літеральні символи

Найпростіший регулярний вираз складається із символів (1 чи більше), котрі позначають літери. Наприклад: “кіт”. У цьому випадкові регулярний вираз позначає послідовність літер, котра складається із символів “к”, “і” та “т”. Регулярні вирази — чутливі до регістру.

Метасимволи

Оскільки регулярні вирази можуть позначати не тільки послідовності літер, потрібно зарезервувати можливість використання деяких з них для зі спеціальною метою. Існує ряд символів, котрі мають окреме призначення (їх називають метасимволами):

Метасимволи зовні квадратних дужок

Символ	Назва	Зміст
\	зворотна скісна риска	загальний екрануючий символ із кількома застосуваннями
^	символ вставки	початок тексту чи рядка у багаторядковому тексті
$	знак долара	кінець тексту чи рядка у багаторядковому тексті
.	крапка	довільний символ за вийнятком кінця рядка
|	вертикальна лінія	початок альтернативної гілки
?	знак питання	розширює значення, квантифікатор 0 чи 1, також робить жадібного квантифікатора ледачим
*	зірочка	квантифікатор 0 або більше
+	плюс	Квантифікатор 1 або більше

Символ	Назва	Зміст
(	відкриваюча кругла дужка	початок підшаблона
)	закриваюча кругла дужка	кінець підшаблона
[	відкриваюча квадратна дужка	початок визначення символьного класу
]	закриваюча квадратна дужка	кінець визначення символьного класу
{	відкриваюча фігурна дужка	початок min/max квантифікатора
}	закриваюча фігурна дужка	кінець min/max квантифікатора

Метасимволи між квадратними дужками (символьний клас)

\	зворотна скісна риска	загальний екрануючий символ
^	символ вставки	заперечення класу у випадкові, якщо символ – на першому місці
-	мінус	позначає діапазон символів

- Зворотна скісна риска

Якщо потрібно скористатися довільним із цих символів у якості літерального, спеціальні символи слід екранувати за допомогою зворотної скісної риски, наприклад “\?”:

```
"/is anybody home\?/", "is anybody home?"
```

Проте, якщо прибрати з регулярного виразу символ екранування знаку питання, то вираз набуде цілком іншого значення. Знак питання після символу позначає вантифікатор 0 чи 1, тобто символ, після якого він стоїть, може бути наявним у тексті, а може і — ні:

```
"/is anybody home?/", "is anybody hom?"
```

Зауважте, що із символами Unicode ситуація стає іншою. Розглянемо особливість.

```
// символ "а" у регулярному виразові - кириличний
"/./", "а"
```

Результат не той, що очікувався. Оскільки літера “а” представлена двома кодами, а крапка у регулярному виразі відповідає довільному символові крім переходу на новий рядок, було поставлено у відповідність крапки цим двом кодам. Якщо додати ще одну крапку до регулярного виразу, тоді кириличну UTF-8 літеру “а” буде знайдено:

```
"/../", "а"
```

Результат:

а

Тепер – пошук успішний, проте універсальність втрачено: якщо у тексті трапляться дві літери латинкою підряд — вони також відповідатимуть регулярному виразові.
Розглянемо аналогічний приклад зі знаком запитання і запишемо текст кирилицею (перевірте, чи кодування — файла — UTF-8. PhpStorm за замовчуванням використовує UTF-8):

```
"/чи є хтось вдома?/", "чи є хтось вдом?"
```

Для даного регулярного виразу не знайдено відповідності у тексті, оскільки UTF-8 використовує дві кодові точки для позначення символу. Код символу “а” - 0x0430. Без модифікатора “/u” у кінці після розділяючого символу “/” регулярний вираз працює із 8-бітними кодами. Таким чином, ? позначає повтор останньої кодової точки символа “а”, а не цілого символу. Виправити ситуацію без вмикання підтримки UTF-8 можна, взявши “а” у круглі дужки, таким чином сформувавши підшаблон:

```
"/чи є хтось вдом(а)?/", "чи є хтось вдом?"
```

Результатом пошуку є вираз, що містить регулярний вираз, літера а у результатові пошуку — відсутня, що дозволено квантифікатором ?. Окремо знайдений підшаблон “(а)” — порожній.

Уникнути непотрібного підшаблона можна за допомогою модифікатора “u”:

```
"/чи є хтось вдома?/u", "чи є хтось вдом?"
```

Результат:

чи є хтось вдом

Аналогічно — коректно працюватиме представлення літери за допомогою символа “.”:

```
"/./u", "а"
```

Результат:

а

- Символи “^” та “$”

Символи “^” та “$” позначають початок та кінець тексту чи рядка (у випадкові багаторядкового тексту), у якому здійснюється пошук. Можуть бути задані окремо чи у поєднанні.

```
"/^текст$/u", "текст"
```

Результат:

текст

Якщо змінити текст так, як у наступному прикладі – результат пошуку буде порожнім.

```
"/^текст$/u", "текст текст"
```

Такий результат пояснюється тим, що у шаблонові задано початок та кінець рядка і реальний текст не відповідає такому шаблонові. Якщо прибрати з шаблона, наприклад, символ кінця рядка, то буде знайдено одне співпадіння. Те, що відповідає слову “текст” на початкові рядка.

- Символ “.”

Крапку використовують для позначення довільного символу окрім кінця рядка.

```
"/^задамо довільний символ: .$/u", "задамо довільний символ: А"
```

Результат:

задамо довільний символ: А

Приклад для випадку багаторядкового тексту:

```
"/^задамо довільний символ: .$/um","задамо довільний символ: А\nзадамо довільний символ: Б
```

Результат:

задамо довільний символ: А
задамо довільний символ: Б

Зверніть увагу на модифікатор /m (multiline).

- Вертикальна лінія |

Вертикальну лінію використовують для розділення альтернативних шаблонів. Як альтернативу також може бути використано порожній рядок. Перевірка співпадінь проводиться зліва-направо, при цьому повертається перше співпадіння:

```
"/білий кіт|пес дивився на Місяць/u","білий кіт дивився на Місяць"
```

Результат:

```
білий кіт
```

Зверніть увагу: повернути не повний шаблон, а частину, що завершується співпадінням однієї із альтернатив. Якщо потрібно знайти повну відповідність тексту шаблонові, альтернативи слід задати у круглих дужках (як підшаблон).

```
"/білий (кіт|пес) дивився на Місяць/u","білий кіт дивився на Місяць";
```

Результат:

```
білий кіт дивився на Місяць
кіт
```

- Знак питання ?

Вище було згадано використання “?” в якості квантифікатора, котрий позначає 0 або 1 входжень. Квантифікатор можна застосовувати не тільки до окремого символу, а і до підшаблону та символьного класу (буде розглянуто пізніше).

```
"/білий (кіт |пес )?дивився на Місяць/iu","Білий дивився на Місяць";
```

Результат:

```
Білий дивився на Місяць
```

Зверніть увагу на використання модифікатора “/i” (Case insensitive match).
Розглянемо приклад знаходження коментарів C/C++. Символ зірочка “*” у шаблоні позначає кількість повторень від 0 і більше.

```
"/\/\*.*\*\//","/* перший коментар */ код /* другий коментар */";
```

Результат:

```
/* перший коментар */ код /* другий коментар */
```

Результат відрізняється від очікуваного. Замість двох коментарів, які можна було б опрацювати окремо, отримуємо рядок, котрий включає все, що знаходилося у вихідному тексті. Це пояснюється жадібністю квантифікатора “*”, застосованому до символа крапка “.”, що позначає довільний символ окрім переходу на наступний рядок (за замовчуванням). Якщо після квантифікатора стоїть символ “?”, квантифікатор стає “ледачим” і позначає мінімально

можливу кількість співпадань.

```
"/\/\*.*?\*\/","/* перший коментар */ код /* другий коментар */";
```

Результат:

```
/* перший коментар */
/* другий коментар */
```

Обмеження жадібності квантифікатора * за допомогою ? дозволяє отримати очікуваний результат.

- Зірочка *

Квантифікатор “*” позначає входження символу, підшаблона чи символьного класу 0 чи більше разів.

```
"/12345*6789/", "12345556789"
```

Результат:

```
12345556789
```

Змінимо текст, в якому здійснюється пошук (видалимо символи 555):

```
12346789
```

Як видно, повернуто рядок, котрий не містить п’ятірок. Комбінація “5*” означає, що 5-ки можуть трапитися у тексті довільну кількість разів підряд, а можуть бути відсутніми узагалі.

- Плюс +

Квантифікатор “+” позначає входження символу, підшаблона чи символьного класу 1 чи більше разів.

```
"/5+/", "12345556789"
```

Результат:

```
555
```

Змінимо текст, в якому здійснюється пошук (видалимо символи 555):

Результат — очікуваний, оскільки у тексті відсутні п’ятірки, а квантифікатор + передбачає наявність хоча б одного входження.

- Круглі дужки ( ) - початок та кінець підшаблона

```
"/(білий|чорний) (пес|кіт)/u", "білий пес, чорний кіт",
```

Результат:

```
білий пес
чорний кіт

білий
чорний

пес
кіт
```

Вище було згадано, що у випадкові, коли є альтернативи, задані за допомогою вертикальної лінії, пошук зупиняється при першому ж співпадінні. Для продовження пошуку альтернативи слід задати у круглих дужках, створивши підшаблони.
Якщо немає потреби включати результати пошуку підшаблонів до видачі, можна застосувати комбінацію “?:” після відкриваючої круглої дужки підшаблона:

```
"/(?:білий|чорний) (?:пес|кіт)/u", "білий пес, чорний кіт"
```

Результат:

```
білий пес
чорний кіт
```

- Квадратні дужки [ ] - початок та кінець символьного класу

За допомогою квадратних дужок позначають символьний клас. Якщо закриваюча квадратна дужка є частиною символьного класу, вона повинна бути першим символом класу (після “^”, якщо наявне заперечення) або екрануватися за допомогою зворотної скісної риски.
Символьному класову відповідає один символ у тексті. Такий символ повинен бути у перелікові символів класу (окрім випадку, коли клас містить заперечення “^”). Якщо символ заперечення є частиною класу — він не може бути на його початкові або повинен екрануватися.
Символ “-” може бути використаний для задання діапазону, наприклад [a-z], [A-Z], [A-z], [0-9].

```
"/[a-z]+/", "abc159zADF"
```

Результат:

```
abc
0
z
6
```

POSIX-нотація символьних класів: імена класів подають у вигляді *[:* ім’я класу *:]* усередині квадратних дужок символьного класу. PCRE також підтримує цю нотацію. Імена класів, що підтримуються, наступні:

Ім’я класу	Зміст
alnum	літери та цифри
alpha	літери
ascii	символів з кодами 0 - 127
blank	пробіл або табуляція
cntrl	Керуючі символи
digit	десяткові цифри (те саме, що і \d)
graph	друковані символи (крім пробіла)
lower	літери нижнього регістру
print	друковані символи з пробілом
unct	друковані символи без літер та цифр
space	білий пробіл (не те саме, що \s)
upper	літери верхнього регістру
word	"word" символи (те саме \w)
xdigit	шістнадцяткові цифри

Приклад:

```
"/[01[:alpha:]%]+/", "abc10%]"
```

Результат:

```
abc10%
```

Усі не буквоцифрові символи за винятком \, -, ^ (на початкові) та закриваючої дужка] не є спеціальними символьними класами. Проте можують бути без шкоди екрановані. Термінатор шаблону – завжди спеціальний і повинен бути екранований у випадкові використання усередині шаблона.

- Фігурні дужки { } - квантифікатори

Повтори задають за допомогою квантифікаторів, що можуть застостосуватися до наступних елементів:

- символ (може бути екранований);
- метасимвол крапка . ;
- символьний клас;
- зворотнє посилання;
- підшаблон у круглих дужках (за винятком коли це – припущення).

Загальний квантифікатор повторень задає мінімальну та максимальну кількість дозволених співпадінь за допомогою двох чисел, розділених комою, у фігурних дужках. Наприклад: шаблонові a{1,3} відповідають “a”, “aa” та “aaa”. Друге число повинно бути меншим за 65536, а перше — не більшим за друге. Закриваюча фігурна дужка у квантифікаторі не є спеціальним символом і її не екранують. Якщо друге число не задано — верхня межа відсутня. Якщо не задано другого числа та кому — квантифікатор вказує на точне число потрібних співпадінь. Відкриваючу фігурну дужку у позиції, де квантифікатор не дозволений, або такий, що не відповідає синтаксисові квантифікатора, - розглядають як

символьні літерали.
Вище були розглянуті три односимвольні квантифікатори:

* - позначає 0 чи більше співпадінь;
+ - позначає 1 чи більше співпадінь;
? - позначає 0 чи 1 співпадіння.

- Ледачі, жадібні та наджадібні квантифікатори

За замовчуванням квантифікатори - жадібні, тобто відповідають максимальній кількості (до максимальної кількості дозволених разів) повторень, не спричиняючи збою решти шаблону. Часом це може викликати ускладнення.Вище розглянуто приклад знаходження коментарів C/C++. Шаблон "/\/*.**\//" не дозволяє вирішити питання, оскільки “.*” позначають будь-що з моменту старту і до кінця рядка (за замовчуванням . не позначає переходу на наступний рядок). Таким чином, регулярний вираз не може зупинитися після того, як зустрічає кінець блока коментаря “*/” і продовжує позначати код після нього (до кінця рядка).

За допомогою знаку питання після квантифікатора можна перетворити квантифікатор з жадібного на ледачого (нежадібного), котрий позначатиме мінімальну кількість входжень. Таким чином, модифікований шаблон "/\/*.*?*\//" вирішує завдання.

Квантифікатори, за якими стоїть знак плюс “+” - наджадібні (possessive). Вони позначають максимальну кількість символів і не повертаються для позначення решти шаблона. Розглянемо приклад. Нехай тест, який потрібно знайти за допомогою регулярного виразу, виглядає як

```
" довільні літери, цифри, символи  та \\" екрановані подвіні лапки \\" "
```

Напишемо регулярний вираз. Він може виглядати так:

```
/"(\\"|[^"])*"/u
```

Починається вираз лапками, після яких йде підшаблон, що містить альтернативи: або екрановані лапки, або довільний символ, котрий не є лапками. У кінці — лапки. Модифікатор /x дозволяє ввести у регулярний вираз пробільні символи (для зручності читання), котрі будуть потім проігноровані при знаходженні відповідностей.

```
'"довільні символи,  \\" екрановані подвійні лапки \\" "';'/ " ( \\\\" | [^"] )* " /ux';
```

Результат:

```
"довільні символи,  \" екрановані подвійні лапки \" "
```

На перший погляд – усе гаразд: вираз обмежений подвійними лапками, усередині – довільні символи, подвійні лапки – екрановані. Модифікуємо текст: екрануємо останні подійні лапки:

```
'"довільні символи,  \\" екрановані подвійні лапки \\" \\"';'/ " ( \\\\" | [^"] )* " /ux';
```

Регулярному виразові знову знаходиться відповідність у вигляді цілого рядка. Проте, якщо аналізувати його зліва-направо, стає очевидним, що текст не відповідає виразові: він закінчується екранованими подвійними лапками, а не звичайними подвійними лапками. Можемо скласти вироджений випадок цього рядка:

```
'"\\"';
```

У такому випадкові результатом буде:

```
"\"
 \
```

Такий результат отримуємо тому, що при аналізові варіантів \" | [^"] після проходження однієї гілки відбувається повернення назад і аналізується наступна гілка. Тобто:

1) знаходиться відповідність подвійних лапок у шаблоні аналогічному символові тексту — Ok;

2) знаходяться екрановані подвійні лапки у тексті, що відповідають лівій частині підшаблона. На цьому рядок завершується, аналізувати далі — нічого;

3) відбувається відкат до точки початку п. 2. При цьому вибирається права альтернатива у підшаблоні — довільний символ окрім подвійних лапок. Зворотна скісна риска відповідає йому. Рядок завершується подійними лапками, що відповідає шаблонові.

Таким чином знайдено описаний вище результат. Для того, щоб відмінити відкати, слід після квантифікатора поставити знак плюс “+”. При цьому квантифікатор стане наджадібним (possessive).

```
'"довільні символи,  \\" екрановані подвійні лапки \\" \\"';'/ " ( \\\\" | [^"] )*+ " /ux';
```

Входження цілого шаблону — відсутнє, підшаблону — також. Отже, регулярний вираз для даного прикладу може бути таким:

```
'"довільні символи,  \\" екрановані подвійні лапки \\" "';'/ " ( \\\\" | [^"] )*+ " /ux';
```

Результат:

```
"довільні символи,  \" екрановані подвійні лапки \" "
```

Детальніше:

https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html
https://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html

У Java API об'єкт, з якого можна читати послідовність байт, називають вхідним потоком (input stream). Об'єкт, у який можна записувати послідовність байт, називають вихідним потоком (output stream).

Абстрактні класи InputStream та OutputStream формують основу ієрархії класів введення-виведення.

Найчастіше джерелами та місцями призначення байт є файли, хоча можуть бути і мережеві з'єднання чи блоки пам'яті.

Класи FileInputStream, FileOutputSream

Конструктори класу FileOutputStream.

FileOutputStream(File file) Створює вихідний файловий потік для запису у файл, представлений об'єктом File
FileOutputStream(File file, boolean append) Створює вихідний файловий потік для запису у файл, представлений об'єктом File. Якщо параметр append має значення true, байти буде дописано у кінець файла
FileOutputStream(FileDescriptor fdObj) Створює вихідний файловий потік для запису до заданого параметром fdObj файлового дескриптора, котрий представляє існуюче з'єднання з існуючим файлом у файловій системі
FileOutputStream(String name) Створює вихідний файловий потік для запису у файл з вказаним іменем name.
FileOutputStream(String name, boolean append) Створює вихідний файловий потік для запису у файл з вказаним іменем name. Якщо параметр append має значення true, байти буде дописано у кінець файла

void	**write**(byte[] b) Записує b.length байт зі вказаного масиву b до вихідного файлового потоку
void	**write**(byte[] b, int off, int len) Записує len байт зі вказаного масиву b, починаючи з відступу off до вихідного файлового потоку
void	**write**(int b) Записує вказаний байт b до вихідного файлового потоку

Зверніть увагу на те, що параметр методу write(int b) оголошено як int, хоча метод описано як такий, що записує байт до потоку. Даний метод класу FileOutputSream успадковано з класу OutputStream. Згідно із документацією тільки молодші 8 біт параметра буде використано, старші 24 біти буде проігноровано.

Розглянемо кожен із варіантів методу write. Запишемо до файла масив байт.

```
import java.io.*;
```

```
public class Test {
   public static void main(String[] args) {
      try {
         byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
         OutputStream os = new FileOutputStream("test.txt");
         os.write(text);
         os.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

У результаті буде створено файл test.txt, розмір файла: 5 байт. Вміст (для перегляду використано редактор bless) наведено на рис. 1.

Рисунок 1 – Вміст файла test.txt (зліва – шістнадцяткові коди, справа – символьне представлення)

Оголосимо посилання на об’єкт потоку таким чином, що байти записувалися у кінець файла.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
         OutputStream os = new FileOutputStream("test.txt", true);
         os.write(text);
         os.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

У результаті щоразу при запускові програми файл test.txt поповнюватимуть байти масиву text, записані у кінець файла.

Розглядемо методом write, котрий оперує зміщенням блоку байт усередині масиву та їхньою кількістю.

```
import java.io.*;
```

```
public class Test {
   public static void main(String[] args) {
      try {
         byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
         OutputStream os = new FileOutputStream("test.txt");
         os.write(text, 1, text.length - 1);
         os.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

У даному прикладі задано одиничне зміщення у масиві байт, призначених для запису у потік, тому, порівняно із попереднім прикладом, у результатові відсутнє значення 0x31 (початковий елемент масиву). Оскільки кількість байт, призначених для запису у потік, зменшилася на 1, значення другого параметра методу: text.length – 1 (останній елемент масиву). Таким чином, у файл будуть записані елементи масиву з індексами від 1 до 4. Елемент з індексом 0 буде пропущений.

Результат:

Рисунок 2 – Вміст файла test.txt. Випадок запису у файл частини масиву байт

Реалізуємо побайтовий запис елементів масиву до файла.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
         OutputStream os = new FileOutputStream("test.txt");
         for(byte b : text)
            os.write(b);
         os.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

Результат буде аналогічним до випадку з методом write(byte[] b):

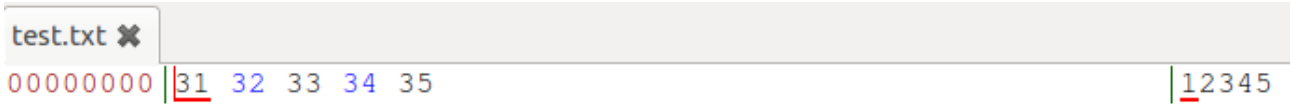


Рисунок 3 – Вміст файла test.txt. Побайтовий запис до файла

Скористаємося щойно записаним файлом, і прочитаємо його за допомогою методів класу FileInputStream (таб. 2).

Конструктори класу FileInputStream

FileInputStream(File file) Створює FileInputStream шляхом відкривання з'єднання з існуючим файлом, ім'я котрого задають за допомогою об'єкта file
FileInputStream(FileDescriptor fdObj) Створює FileInputStream шляхом використання файлового дескриптора fdObj, котрий представляє існуюче з'єднання з існуючим файлом файлової системи
FileInputStream(String name) Створює FileInputStream шляхом відкривання з'єднання з існуючим файлом, ім'я котрого задають за допомогою шляху name у файловій системі

int	read() Читає байт даних з вхідного файлового потоку
int	read(byte[] b) Читає до b.length байт даних з вхідного файлового потоку у масив байт b
int	read(byte[] b, int off, int len) Читає до len байт даних з вхідного файлового потоку у масив байт b, записуючи їх зі зміщенням у масиві off

Для того, щоб дізнатися об'єм даних, доступних для читання, використаємо метод available класу FileInputStream. Метод не блокує виконання потоку. У деяких випадках може виглядати блокуючим у зв'язку із повільним виконанням (наприклад, при читанні у повільній мережі). Таким чином, читання з потоку не буде блокуючим, оскільки метод read спробує прочитати доступну для нього кількість байт і не виходитиме поза її межі.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         InputStream is = new FileInputStream("test.txt");
         int size = is.available();
         for(int i = 0; i < size; i++) {
            byte text = (byte)is.read();
            System.out.printf("%h ", text);
         }
         is.close();
      } catch (IOException e) {
         System.out.println(e);
      }
```

```
    }
}
```

Результат:

31 32 33 34 35

Метод read() блокує виконання потоку, якщо читання все ще не є можливим (наприклад, при читанні з сокета). Повертає -1, коли досягнуто кінець файла.

У наступному прикладі записаний файл прочитано до масиву байт цілим (так, щоб початковий елемент масиву було заповнено початковим байтом файла).

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         InputStream is = new FileInputStream("test.txt");
         int size = is.available();
         if (size > 0) {
            byte[] text = new byte[size];
            is.read(text);
            for(int i = 0; i < size; i++)
               System.out.printf("%h ", text[i]);
         }
         is.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

Результат:

31 32 33 34 35

Аналогічно до запису частини масиву байт у файл, прочитаємо з файла декілька байт до масиву, вказавши зміщення у ньому та кількість байт.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         InputStream is = new FileInputStream("test.txt");
         int size = is.available();
         if (size > 3) {
            System.out.println("length: " + size);
```

```
            byte[] text = new byte[5];
            is.read(text, 1, 2);
            for(int i = 0; i < 5; i++)
               System.out.printf("%h ", text[i]);
         }
         is.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

Результат:

length: 5
0 31 32 0 0

У даному прикладі перевіряють кількість байт, доступних для читання, і, якщо кількість більша за 3, створюють масив з 5 елементів типу byte та записують у нього 2 байти з файлового потоку зі зміщенням 1. Тому початковий елемент масиву має значення 0 (ініціалізація за замовчуванням), наступних два – відповідні байти з файлу і останні два – 0 (ініціалізація за замовчуванням).

Усі методи read – блокуючі.

За потреби можна пропустити певну кількість байт у потокові. Метод skip(long n) пропускає n байт і повертає кількість байт, яка насправді була пропущена.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         InputStream is = new FileInputStream("test.txt");
         int size = is.available();
         if (size > 1) {
            System.out.println(is.skip(1) + " byte/s skipped");
            System.out.printf("%h\n", is.read());
            System.out.println(is.skip(-1) + " byte/s skipped");
            System.out.printf("%h", is.read());
         }
         is.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
}
```

Результат:

1 byte/s skipped

32
-1 byte/s skipped
32

Метод може припинити пропускати байти через деяку невелику кількість байт (навіть, 0). Якщо n – від'ємне число, метод намагатиметься здійснювати пропуск у зворотньому напрямі. Якщо файл не підтримує ріверс, генерується IOException. Знак повернутого методом числа залежить від напряму: додатній відповідає прямому, від'ємний – зворотньому.

Метод може "пропускати" більше байт, ніж залишилося у файлі. У цьому випадкові не виникає виняткової ситуації і кількість пропущених байт може включати деяку кількість байт поза EOF файла. При спробі прочитати з потоку далі його останнього символу буде отримано -1, що відповідає EOF.

Виняткові ситуації:
IOException - якщо n від'ємне і вказує на передпочаток файла, якщо потік не підтримує переміщення чи виникають помилки I/O.

Класи BufferedOutputStream та BufferedInputStream

Клас BufferedOutputStream імплементує буферизований вихідний потік. За допомогою такого потоку програма може записувати байти до основного вихідного потоку без потреби виклику системи, що лежить в основі, для кожного байту окремо.

Поля

protected byte[]	buf внутрішній буфер для збереження даних
protected int	count кількість дійсних байт у буфері

Конструктори

BufferedOutputStream(OutputStream out) Створює новий буферизований вихідний потік для запису даних до вказаного вихідного потоку, що лежить в основі такої передачі
BufferedOutputStream(OutputStream out, int size) Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size

Деякі методи класу (детальніше – див. посилання)

void	flush() Очищає буферизований вихідний потік
void	write(byte[] b, int off, int len) Записує len бакт з маиву b, починаючи з off до буферизованого вихідного потоку
void	write(int b) Записує вказаний байт до буферизованого вихідного потоку

Методи, успадковані з java.io.FilterOutputStream

void	write(byte[] b) Записує до буферизованого вихідного потоку b.length байт
void	close() Закриває буферизований вихідний потік і звільняє системні ресурси, асоційовані з цим потоком. Метод close() класу BufferedOutputStream викликає метод flush() цього класу, після цього – викликає метод close() вихідного потоку, що лежить в основі

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      byte[] text = {0x31, 0x32, 0x33, 0x34, 0x35};
      try{
         BufferedOutputStream bufferedOutputStream =
               new BufferedOutputStream(
                     new FileOutputStream("test.txt")
               );
         bufferedOutputStream.write(text);
         bufferedOutputStream.close();
      } catch ( IOException e){
      }
   }
}
```

Результат:

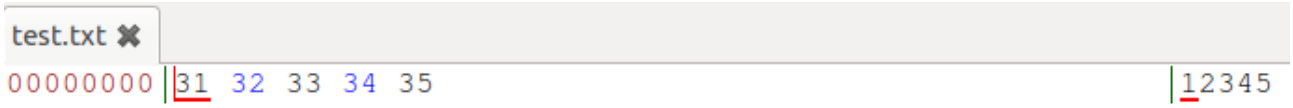


Рисунок 4 – Вміст файла test.txt. Запис за допомогою буферизованого потоку виведення

```
import java.io.*;
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      byte[] text = new byte[5000000];
      Arrays.fill(text, (byte)0x31);
      try{
         FileOutputStream fileOutputStream =
               new FileOutputStream("test.txt");
         final long startTime = System.nanoTime();
         for(byte b : text)
            fileOutputStream.write(b);
         fileOutputStream.close();
         final long duration = System.nanoTime() - startTime;
         System.out.printf("%.2f s", duration / 1000000000.0);
```

```
        } catch ( IOException e){
        }
    }
}
```

Результат:

7.27 s

```
import java.io.*;
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      byte[] text = new byte[5000000];
      Arrays.fill(text, (byte)0x31);
      try{
         BufferedOutputStream bufferedOutputStream =
               new BufferedOutputStream(
                     new FileOutputStream("test.txt")
               );
         final long startTime = System.nanoTime();
         for(byte b : text)
            bufferedOutputStream.write(b);
         bufferedOutputStream.close();
         final long duration = System.nanoTime() - startTime;
         System.out.printf("%.2f s", duration / 1000000000.0);
      } catch ( IOException e){
      }
   }
}
```

Результат: 0.11 s

Конкретні результати при відтворенні прикладів залежатимуть від конігурації системи. Легко помітити, що записування у файл 5 000 000 байт з використанням буферизованого потоку виведення відбувається набагато швидше за побайтовий запис без використання буфера.

Буферизоване виведення зручне у випадкові, якшо у файл часто записують невеликі об’єми даних.

Розмір буфера за замовчуванням: 8192 байт (оголошений у файлі BufferedOutputStream.java)

```
public BufferedOutputStream(OutputStream out) {
   this(out, 8192);
}
```

Важливо пам’ятати про закривання буферизованого потоку за допомогою методу close(). При цьому вивільняється буфер (еквівалент виклику методу flush()) та закривається основний файловий потік виведення.

Якщо не закрити потік і у буфері залишаться дані, вони не будуть записані до файла, оскільки не буде викликано метод flush(). У наведеному нижче прикладі змодельовано описану ситуацію.

```
import java.io.*;
public class Test extends BufferedOutputStream {
   public static void main(String[] args) {
      try {
         Test bufferedOutputStream =
               new Test(
                     new FileOutputStream("test.txt"){
                        @Override
                        public void close() throws IOException {
                           System.out.println("FileOutputStream is closed");
                           super.close();
                        }
                     }
               );
         bufferedOutputStream.write(0x31);
         bufferedOutputStream.close();
      } catch (IOException e) {
         System.out.println(e);
      }
   }
   public Test(OutputStream os) {
      super(os);
   }
   @Override
   public synchronized void write(int b) throws IOException {
      System.out.println("buffer size: " + super.buf.length);
      super.write(b);
      System.out.println("bytes count: " + super.count);
   }
   @Override
   public synchronized void flush() throws IOException {
      System.out.println("BufferedOutputStream is flushed");
      super.flush();
   }
}
```

Результат:

```
buffer size: 8192
bytes count: 1
BufferedOutputStream is flushed
FileOutputStream is closed
```

Рисунок 4 – Вміст файла test.txt. Результат запису у файл із наступним закриттям буферизованого потоку виведення

Для отримання доступу до полів класу BufferedOutputStream, котрі оголошені як protected, створимо клас Test, нащадок BufferedInputStream. Конструктор класу викликає конструктор батьківського класу, передаючи йому свій параметр – посилання на об’єкт класу OutputStream (у даному випадкові – FileOutputStream).

Власна реалізація методу write(int b) дає можливість дізнатися розмір буфера потоку та кількість елементів, що заповнюють цей буфер. Як видно з результату, розмір буфера 8192 байт, після запису 1 байту до буфера поле count батьківського класу дорівнює 1.

Зверніть увагу на те, що метод flush() не викликають явно, проте результат виконання програми демонструє, що виклик відбувається. Згідно із документацією, метод close() класу BufferedOutputStream вивільняє буфер за допомогою методу flush(). Для демонстрації цього створено власну реалізацію методу, котра повідомляє про його виклик.

Також метод close() повинен закрити потік, котрий лежит в основі передачі байт до файла, у даному випадкові – FileOutputSream. При ініціалізації посилання на об’єкт класу BufferedInputStream як параметр у конструктор передано анонімний клас на основі FileOutputSream з власною реалізацією методу close(), котра повідомляє про його виклик.

Якщо закоментувати рядок bufferedOutputStream.close() – запис до файлу не відбудеться, оскільки не буде викликано метод flush(). Даний метод можна також викликати і уручну.

Клас BufferedInputStream

Поля

protected byte[]	buf Внутрішній буфер, призначений для зберігання даних
protected int	count Індекс, більший на 1 за індекс останнього значущого елемента буфера
protected int	marklimit Максимальна кількість елементів, котру можна прочитати у прямому напрямі після виклику методу mark() перед викликом методу reset(), котрий може завершитися невдачею і спровокувати появу виняткової ситуації
protected int	markpos Значення, котре мало поле pos на момент виклику методу mark()
protected int	pos Поточа позиція у буфері

Конструктори

BufferedInputStream(InputStream in) Створює BufferedInputStream та зберігає його аргумент, вхідний потік in, для подальшого використання
BufferedInputStream(InputStream in, int size)

Створює BufferedInputStream з буфером заданого розміру та зберігає його аргумент, вхідний потік in, для подальшого використання

Методи

int	available() Повертає оціночну кількість байт, котра може бути пропущена чи прочитана з вхідного потоку без блокування наступного виклику методу цього вхідного потоку
void	close() Закриває цей вхідний потік та звільняє ресурси системи, пов'язані з потоком
void	mark(int readlimit) Запам'ятовує позицію у потоці, до якої пізніше можна повернутися за допомогою методу reset(). Детальніше про метод – у батьківському класі InputStream
boolean	markSupported() Перевіряє, чи даний вхідний потік підтримує меоди mark() та reset()
int	read() Читає байт з вхідного потоку. Детальніше – у батьківському класі InputStream.
int	read(byte[] b, int off, int len) Читає байти із байт-орієнтованого вхідного потоку у вказаний масив b, записуючи у нього елементи зі зміщенням off від початкую; len – максимальна кількість елементів, котрі потрібно прочитати
void	reset() Повертає позицію у буфері до такої, що була запам'ятована за допогою методу mark(). Детальніше – у батьківському класі InputStream
long	skip(long n) Пропускає вазану кількість байт. Детальніше – у батьківському класі InputStream.

Клас BufferedInputStream реалізує буферизацію вхідного потоку даних і забезпечує підтримку міток, шо дозволяють за потреби повертутися до позиції у файлі, позначеної міткою.

При створенні BufferedInputStream створюється внутрішній буфер. У процесі читання чи пропуску байт зі вхідного потоку буфер оновлюється за потреби. При проставлянні мітки запам'ятовується позиція мітки у файлі. При потребі до цієї позиції можна повертунутися (reset). При цьому усі байти, що були прочитані, починаючи з позиції мітки, будуть прочитані повторно. Потік може вважати мітку нечинною, якщо кількість байт, прочитаних після неї, перевищує кількість, встановлену при створенні мітки (mark(int position)).

Подібно до описаного, порівняємо швидкість читання з файла за допомогою буферизованого вхідного потоку і побайтове читання без буферизації. Для цього скористаємося одним із попередніх прикладів, за допомогою якого згенеруємо файл 5 000 000 байт даних.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      byte[] text = new byte[5000000];
```

```
        try {
            BufferedInputStream bufferedInputStream =
                    new BufferedInputStream(
                    new FileInputStream("test.txt"));
            long size = bufferedInputStream.available();
            final long startTime = System.nanoTime();
            if (size == 5000000) {
                int j = 0;
                int b;
                while ((b = bufferedInputStream.read()) != -1)
                    text[j++] = (byte) b;
                System.out.println("cycle counter: " + j);
            }
            final long duration = System.nanoTime() - startTime;
            bufferedInputStream.close();
            System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
            System.out.printf("%.2f s", duration / 1000000000.0);
        }catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Результат:

```
cycle counter: 5000000
0x31 0x31 ...
0.12 s
```

Тепер зробимо аналогічне, тільки без буферизації.

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        try {
            FileInputStream fileInputStream =
                    new FileInputStream("test.txt");
            long size = fileInputStream.available();
            final long startTime = System.nanoTime();
            if (size == 5000000) {
                int j = 0;
                int b;
                while ((b = fileInputStream.read()) != -1)
                    text[j++] = (byte) b;
                System.out.println("cycle counter: " + j);
            }
            final long duration = System.nanoTime() - startTime;
```

```
            fileInputStream.close();
            System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
            System.out.printf("%.2f s", duration / 1000000000.0);
        }catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

Результат:

```
cycle counter: 5000000
0x31 0x31 ...
2.35 s
```

Для оцінки різниці у швидкості доступу прочитаємо файл блоками, а не побайтно:

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        byte[] text = new byte[5000000];
        try {
            BufferedInputStream bufferedInputStream =
                    new BufferedInputStream(
                            new FileInputStream("test.txt")
                    );
            final long startTime = System.nanoTime();
            int bytesRead = bufferedInputStream.read(text);
            final long duration = System.nanoTime() - startTime;
            System.out.println("bytes read: " + bytesRead);
            bufferedInputStream.close();
            System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
            System.out.printf("%.3f s", duration / 1000000000.0);
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

```
0x31 0x31 ...
0.003 s
```

Легко помітити, що швидкість побайтового читання та буферизованого відрізняється на декілька порядків.

Якщо програма постійно читає невеликі об’єми даних, BufferedInputStream значно покращує продуктивність. Кожен запит на читання над небуферизованим потоком

виливається у системний запит до операційної системи на читання певної кількості байт. Таким чином може виникати значна перевитрата ресурсів операційної системи. Буферизований потік зменшує кількість таких запитів шляхом читання одного відносно великого (наприклад, 8192 байт) блоку даних у внутрішній буфер, після чого програма отримує дані з цього буфера. Таким чином, значно скорочується кількість системних запитів.

Проте, якщо розмір блоку, котрий має бути прочитаний, більше за розмір буфера, використання буферизованого потоку зменшуватиме продуктивність.

Особливістю класу BufferedInputStream є можливість зберегти за допомогою методу mark(int readlimit) поточну позицію у буфері pos і вказати кількість байт readlimit, котрі можуть бути прочитані у прямому напрямі, після чого за допомогою методу reset() повернутися до збереженої позиції і повторно прочитати байти з буфера. Потік не гарантує таку можливість, якщо з потоку прочитано більше за readlimit байт, починаючи з позиції маркера pos. Для кращого розуміння розглянемо частково реалізацію класу BufferedInputStream (java version "1.8.0_77"):

```
private void fill() throws IOException {
    byte[] buffer = getBufIfOpen();
    if (markpos < 0)
        pos = 0;            /* no mark: throw away the buffer */
    else if (pos >= buffer.length)  /* no room left in buffer */
        if (markpos > 0) {  /* can throw away early part of the buffer */
            int sz = pos - markpos;
            System.arraycopy(buffer, markpos, buffer, 0, sz);
            pos = sz;
            markpos = 0;
        } else if (buffer.length >= marklimit) {
            markpos = -1;   /* buffer got too big, invalidate mark */
            pos = 0;        /* drop buffer contents */
        } else if (buffer.length >= MAX_BUFFER_SIZE) {
            throw new OutOfMemoryError("Required array size too large");
        } else {            /* grow buffer */
            int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
                    pos * 2 : MAX_BUFFER_SIZE;
            if (nsz > marklimit)
                nsz = marklimit;
            byte nbuf[] = new byte[nsz];
            System.arraycopy(buffer, 0, nbuf, 0, pos);
            if (!bufUpdater.compareAndSet(this, buffer, nbuf)) {
                // Can't replace buf if there was an async close.
                // Note: This would need to be changed if fill()
                // is ever made accessible to multiple threads.
                // But for now, the only way CAS can fail is via close.
                // assert buf == null;
                throw new IOException("Stream closed");
            }
            buffer = nbuf;
        }
    count = pos;
    int n = getInIfOpen().read(buffer, pos, buffer.length - pos);
    if (n > 0)
```

```
        count = n + pos;
    }
    public synchronized int read() throws IOException {
        if (pos >= count) {
            fill();
            if (pos >= count)
                return -1;
        }
        return getBufIfOpen()[pos++] & 0xff;
    }
    public synchronized void mark(int readlimit) {
        marklimit = readlimit;
        markpos = pos;
    }
    public synchronized void reset() throws IOException {
        getBufIfOpen(); // Cause exception if closed
        if (markpos < 0)
            throw new IOException("Resetting to invalid mark");
        pos = markpos;
    }
```

Метод reset() генерує виняткову ситуацію, якщо markpos<0 (насправді, -1 – це значення, котре присвоюється змінній у випадкові, якщо неможливо зберегти позицію маркера або він поки що не встановлений взагалі).

Метод mark(int readlimit) ініціалізує поле класу marklimit числом, величина котрого відповідає кількості байт, у межах якої гарантовано можливий відкат до позиції маркера (позиція маркера є поточною позицією у файлі на момент встановлення маркера). За межами marklimit таке повернення не гарантується. Нижче буде розглянуто, чому саме.

Метод read() або повертає значення елемента буфера buf з індексом pos у випадкові, якщо ще не досягнуто індекса останнього прочитаного до буфера елемента (після цього значення індекса інкрементують), або намагається перезаповнити буфер (метод fill()) новими елементами з основного вхідного потоку, що лежить в основі такого введення. Перезаповнення буфера відбувається у випадкові, коли pos >= count (згідно з наведеним кодом). Якщо після спроби викликати метод fill() ситуація pos >= count не міняється, метод read() повертає -1, що означає досягнення кінця файла.

Розглянемо роботу методу fill(). Для цього проаналізуємо деякі характерні варіанти заповнення буфера.

pos	-	0								buf.length
markpos	-1									

Рисунок – Стан полів до першого завантаження буфера або повне оновлення буфера із неможливістю відкату до маркера

pos	-	0								buf.length
markpos	-1									

Рисунок – Стан полів відповідає прочитаному останньому елементові буфера, маркер – не встановлено

У першому варіанті не прочитано жодного елемента, маркер не встановлено, тому читання наступного байта з потоку не викликає труднощів.

Другий варіант відрізняється від першого тим, що буфер заповнено повністю (pos = buf.length). Позиція у буфері виходить за його межі. Маркер – не встановлено. За потреби прочитати наступний байт слід обнулити поле позиції у буфері і, фактично, перейти до першого варіанту.

Якщо маркер не встановлено, відбувається чергування першого та другого описаних станів із усіма проміжними варіантами, котрі не наведено. На кількість заповнених елементів у буфері вказує поле count (більше на 1 за індекс останнього заповненого елемента).

Алгоритм використання буфера ускладнюється, якщо встановлено маркер.

При цьому існують два важливих варіанти. Один із них: маркер знаходиться між початком та кінцем буфера. Необхідно прочитати деяку кількість байт байт. При цьому можна без шкоди для даних, захищених маркером, перезаписати елементи буфера, що знаходяться зліва від маркера, починаючи із нульового (зсунувши буфер уліво). Поля: pos = pos - markpos; markpos = 0. Напрям зсуву показано на рисункові стрілками: ←.

pos	-	0								buf.length
markpos	-1	|←			←					

Рисунок – Стан полів відповідає прочитаному останньому елементові буфера, позиція маркера – між початком та кінцем буфера

Інший варіант характерний тим, що позиція маркера markpos дорівнює 0. Варіант розгалужується у залежності від значення marklimit. Величина marklimit має або скінченне у межах даного буфера значення, або виходить за його межі.

pos	-	0								buf.length
markpos		0								
marklimit						→|				max

Рисунок – Стан полів відповідає прочитаному останньому елементові буфера, маркер знаходиться на початкові буфера, marklimit вказує на елемент усередині буфера

З рисунка видно, що не існує жодної можливості зсунути елементи буфера уліво. У будь-якому випадкові елементи, пам’ятати значення котрих зобов’язує маркер, буде втрачено.

Якщо ж marklimit вказує на елемент поза межами буфера (тобто, розмір буфера менший за кількість елементів, пам’ятати котрі зобов’язує маркер), розмір буфера намагаються збільшити удвічі (змінна nsz у класі). У випадкові, якщо значення змінної nsz більше за marklimit, розмір буфера обмежують величиною marklimit. Описане повторюють щоразу при викликові методу fill().

index	-	0								buf.length
pos	-									buf.length
markpos		0								
marklimit										→

Рисунок – Прочитано останній елемент буфера, маркер знаходиться на початкові буфера, marklimit вказує на елемент поза межами буфера

Наведений нижче код ілюструє алгоритм збільшення розміру буфера. На початкові перевіряють можливість збільшення розміру буфера удвічі.

```
int nsz = (pos <= MAX_BUFFER_SIZE - pos) ?
      pos * 2 : MAX_BUFFER_SIZE;
if (nsz > marklimit)
   nsz = marklimit;
```

Зверніть увагу на те, що розмір буфера вказано непрямо, замість buf.length використано pos. При цьому pos із формулювання задачі більший за індекс останнього елемента і за визначенням не може бути більшим ніж на 1, оскільки у такому випадкові мав би бути прочитаним ще один байт, а його фізично нікуди прочитати на даному етапі.

Розглянемо приклад. Підготуємо файл, у якому перший байт відрізняється від усіх інших. Такий підхід зручний, якщо потрібно продемонструвати повторне читання.

```
import java.io.*;
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      byte[] text = new byte[5000000];
      Arrays.fill(text, (byte)0x31);
      text[0] = 0x35;
      try{
         BufferedOutputStream bufferedOutputStream =
               new BufferedOutputStream(
                     new FileOutputStream("test.txt")
               );
         bufferedOutputStream.write(text);
         bufferedOutputStream.close();
      } catch ( IOException e){
         e.printStackTrace();
      }
   }
}
```

Читаємо файл за допомогою буферизованого вхідного потоку.

```
import java.io.*;
public class Test extends BufferedInputStream {
   public static void main(String[] args) {
      byte[] text = new byte[8192];
      try {
         Test bufferedInputStream =
               new Test(
                     new FileInputStream("test.txt"));
         System.out.println("mark supported = " +
```

```
                bufferedInputStream.markSupported());
            bufferedInputStream.printPos();
            bufferedInputStream.mark(100);
            bufferedInputStream.printPos();
            bufferedInputStream.read(text, 0, 8192);
            bufferedInputStream.printPos();
            // bufferedInputStream.read();
            // bufferedInputStream.printPos();
            bufferedInputStream.reset();
            bufferedInputStream.printPos();
            bufferedInputStream.close();
            System.out.printf("0x%x 0x%x ...\n", text[0], text[1]);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public Test(InputStream i) {
        super(i);
    }
    int getPos() {
        return super.pos;
    }
    int getMarkPos(){
        return super.markpos;
    }
    int getCount() {return super.count;}
    int getMarkLimit() {return super.marklimit; }
    int getBufferLength() {return super.buf.length; }
    void printPos(){
        System.out.println("pos = " + getPos());
        System.out.println("markpos = " + getMarkPos());
        System.out.println("count = " + getCount());
        System.out.println("marklimit = " + getMarkLimit());
        System.out.println("buffer.length = " + getBufferLength());
        System.out.println("---------------");
    }
}
```

Результат:

```
mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
---------------
pos = 0
markpos = 0
count = 0
```

```
marklimit = 100
buffer.length = 8192
---------------
pos = 8192
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
---------------
0x35 0x31 ...
pos = 0
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
---------------
0x35 0x31 ...
```

Видно, що на початкові усі поля-індекси класу дорівнюють 0, маркер – не встановлено. На другому кроці встановлюють маркер і поле marklimit набуває значення 100.

Наступний крок: читання 8192 байт з вхідного потоку. При цьому pos набуває значення 8192, count – 8192 (на 1 більше за індекст останнього значущого елемента буфера), marklimit– зберігає своє значення 100 незмінним.

Далі виконують метод reset(). Метод успішно завершується поверненням до маркера, встановленим на нульовій позиції (pos: 0, markpos: 0). Можна повторно читати елементи потоку (насправді – буфера).

Розкоментуйте закоментовані рядки прикладу. Ситуація цілком зміниться: при повністю заповненому буфері і нульовій позиції маркера та значенні marklimit, що не перекриває буфер повністю, прочитати ще один байт, не пошкодивши буфер немає можливості.

```
mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
---------------
pos = 0
markpos = 0
count = 0
marklimit = 100
buffer.length = 8192
---------------
pos = 8192
markpos = 0
count = 8192
marklimit = 100
buffer.length = 8192
---------------
0x35 0x31 ...
pos = 1
```

```
markpos = -1
count = 8192
marklimit = 100
buffer.length = 8192
---------------
java.io.IOException: Resetting to invalid mark
```

Якщо прибрати з тексту програми виклик reset(), то байти з потоку успішно читатимуться далі. Зрозуміло, що повернутися до маркера буде неможливо, хоча marklimit і встановлено.

Змінимо параметр методу mark(100) на mark(10000). Таким чином, слід запам’ятати кількість байт, явно більшу за поточний розмір буфера.

```
mark supported = true
pos = 0
markpos = -1
count = 0
marklimit = 0
buffer.length = 8192
---------------
pos = 0
markpos = 0
count = 0
marklimit = 10000
buffer.length = 8192
---------------
pos = 8192
markpos = 0
count = 8192
marklimit = 10000
buffer.length = 8192
---------------
0x35 0x31 ...
pos = 8193
markpos = 0
count = 10000
marklimit = 10000
buffer.length = 10000
---------------
pos = 0
markpos = 0
count = 10000
marklimit = 10000
buffer.length = 10000
---------------
0x35 0x31 ...
```

З результату видно, що розмір буфера збільшено таким чином, що він відповідає параметрові marklimit. Також показано, що у випадкові успішного виконання методу reset() відбувається перехід на позицію встановленого раніше маркера (під час читання отримують ті самі байти).

Класи DataOutputStream та DataInputStream

Розглянуті вище класи працюють з байтами. DataOutputStream (DataInputStream) надає можливість записувати (читати) дані базових типів Java до вихідного (вхідного) потоку у бінарному вигляді. Прочитати їх можна за допомогою DataInputStream.

DataOutputStream

Поля

protected int	written Кількість байт, записаних до data output stream на даний момент

Конструктор

DataOutputStream(OutputStream out)
Створює новий потік data output stream для запису даних у вихідний потік out.

Методи

void	flush() Вивільняє data output stream
int	size() Повертає поточне значення лічильника written – кількість байт, записаних до data output stream на даний час
void	write(byte[] b, int off, int len) Записує len байт з масиву байт зі зміщенням off у вихідний потік, котрий лежить в основі
void	write(int b) Записує байт (молодші 8 біт аргумента) параметра у вихідний потік, котрий лежить в основі
void	writeBoolean(boolean v) Записує boolean у вихідний потік, котрий лежить в основі, як 1-байт
void	writeByte(int v) Записує байт (молодші 8 біт аргумента) параметра у вихідний потік, котрий лежить в основі
void	writeBytes(String s) Записує рядок як послідовність байт у потік, що лежить в основі. Кожен символ рядка записують, відкидаючи його старші 8 біт
void	writeChar(int v) Записує символ у потік, що лежить в основі, як 2 – байтове значення, першим – старший байт
void	writeChars(String s) Записує рядок у потік, що лежить в основі, як послідовність символів
void	writeDouble(double v) Перетворює аргументи типу double у long з використанням методу doubleToLongBits класу Double, після чого записує long до потоку, що лежить в основі, як 8 байт. Першим записують старший байт

void	writeFloat(float v) Перетворює аргументи типу float у int з використанням методу doubleToIntBits класу Float, після чого записує int до потоку, що лежить в основі, як 4 байти. Першим записують старший байт
void	writeInt(int v) Записує int до потоку, що лежить в основі, як 4 байти. Першим записують старший байт
void	writeLong(long v) Записує long до потоку, що лежить в основі, як 8 байт. Першим записують старший байт
void	writeShort(int v) Записує short до потоку, що лежить в основі, як 2 байти. Першим записують старший байт
void	writeUTF(String str) Записує рядок до потоку, що лежить в основі, за допомогою модифікованого modified UTF-8 кодування

Зверніть увагу на методи write(int b) та writeByte(int v). Функціонально вони однакові. Обидва записують у потік молодші 8 біт аргумента. Проте, writeByte – final (не може бути перевизначений дочірнім класом) та unsynchronized, а write – не є final та є synchronized.

DataInputStream

Конструктор

DataInputStream(InputStream in)
Створює DataInputStream, котрий використовує заданий InputStream.

Методи

int	read(byte[] b) Читає деяку кількість байт з вхідного потоку і зберігає їх у масиві b
int	read(byte[] b, int off, int len) Читає до len байт з вхідного потоку і зберігає їх у масиві b зі зміщенням off
boolean	readBoolean() Читає байт з вхідного потоку. Якщо значення байта – одиниця – повертає true, інакше – false. Парний метод – writeBoolean()
byte	readByte() Читає байт з вхідного потоку, розглядаючи його як величину у межах від -128 до +127. Парний метод - writeByte()
char	readChar() Читає char (2 байти) з вхідного потоку. Парний метод - writeChar()
double	readDouble() Читає 8 байт з вхідного потоку і повертає double. Парний метод - writeDouble()
float	readFloat() Читає 4 байти з вхідного потоку і повертає float. Парний метод – writeFloat()
void	readFully(byte[] b) Читає деяку кількість байт з вхідного потоку та зберігає їх у масиві b. Кількість

	байт дорівнює розмірові масиву b
void	readFully(byte[] b, int off, int len) Читає з вхідного потоку строго len елементів та зберігає їх у масиві b зі зміщенням off
int	readInt() Читає з вхідного потоку 4 байти і повертає еквівалентний int. Парний метод - writeInt()
String	readLine() Застарілий Слід використовувати клас BufferedReader
Long	readLong() Читає з вхідного потоку 8 байт і повертає еквівалентний Long. Парний метод – writeLong()
short	Читає з вхідного потоку 2 байт і повертає еквівалентний short. Парний метод – writeShort()
int	readUnsignedByte() Метод читає 1 байт, розтягує його за допомогою нулів у старших розрядах до типу int і повертає результат у діапазоні 0 – 255
int	readUnsignedShort() Читає 2 байти з вхідного потоку і повертає результат у діапазоні 0 – 65535
String	readUTF() Читає рядок у модифікованому UTF-8 форматі
static String	readUTF(DataInput in) Статичний метод. Читає рядок у модифікованому UTF-8 форматі з потоку in та повертає рядок символів як String
int	skipBytes(int n) Пропускає n байт у вхідному потокові

Методи, успадковані від java.io.FilterInputStream: available, close, mark, markSupported, read, reset, skip.

Методи, успадковані від java.lang.Object: clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait.

Метод close() класу DataOutputStream викликає метод flush() цього класу, після чого викликає метод close() потоку, що є параметром конструктора. Метод close() класу DataInputStream викликає метод close() потоку, що є параметром конструктора.

Розглянемо кілька прикладів.

```
import java.io.*;
import java.util.Arrays;
public class Test {
   public static void main(String[] args) {
      try {
         byte[] arr = {1,2,3};
         DataOutputStream dataOutputStream =
               new DataOutputStream(new FileOutputStream("test.txt"));
         dataOutputStream.write(arr);
```

```
            dataOutputStream.write(arr, 1, arr.length - 1);
            dataOutputStream.writeFloat(1.4f);
            dataOutputStream.writeDouble(5.6);
            dataOutputStream.writeBoolean(true);
            dataOutputStream.write(-1);
            dataOutputStream.writeByte(-127);
            dataOutputStream.writeShort(32767);
            dataOutputStream.writeInt(32768);
            dataOutputStream.writeLong(2147483648L);
            dataOutputStream.close();

            DataInputStream dataInputStream =
                    new DataInputStream(new FileInputStream("test.txt"));
            Arrays.fill(arr, (byte)0);
            dataInputStream.read(arr);
            System.out.println(Arrays.toString(arr));
            Arrays.fill(arr, (byte)0);
            dataInputStream.read(arr, 1, 2);
            System.out.println(Arrays.toString(arr));
            System.out.println(dataInputStream.readFloat());
            System.out.println(dataInputStream.readDouble());
            System.out.println(dataInputStream.readBoolean());
            System.out.println(dataInputStream.read());
            System.out.println(dataInputStream.readByte());
            System.out.println(dataInputStream.readShort());
            System.out.println(dataInputStream.readInt());
            System.out.println(dataInputStream.readLong());
            dataInputStream.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Результат:

```
[1, 2, 3]
[0, 2, 3]
1.4
5.6
true
255
-127
32767
32768
2147483648
```

Зверніть увагу на невідповідний, на перший погляд, результат виконання read(). За допомогою write(int) записано число -1. А прочитано воно як 255. Цілі типи характерні тим, що старший двійковий розряд числа, піднятий до ступеня з показником, який відповідає

номерові розряду, у випадкові від’ємних значень обчислюють зі знаком мінус. Наприклад:

Номер роздряду	3	2	1	0
Біт	1	1	1	1
Значення	-(2^3)	2^2	2^1	2^0

Отже, двійкове 1111 число у Java для 4-розрядного цілого типу позначає десяткове -1. Від кількості розрядів значення не змінюється у випадкові, якщо усі розряди – одиниці. Фактично, -1 int = 11111111111111111111111111111111. Молодші 8 біт показано на чорному тлі. При записові у потік метод write() ігнорує старші 24 біти і записує 8 молодших. Метод read() читає байт і повертає результат як int у діапазоні від 0 до 255. Якщо немає доступних для читання байт – метод повертає -1.

Таким чином, двійкові 11111111 метод розглядає як десяткове 255.

Натомість, метод readByte() розглядає байт як число у діапазоні від -128 до +127. Метод readByte() придатний для роботи у парі з методом writeByte().

Метод writeChar(int v) записує char до потоку у вигляді 2 байт, першим – старший байт.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         int i = 0x31; // ASCII код цифри "1", відповідає молодшому байтові utf-16
         DataOutputStream dataOutputStream =
               new DataOutputStream(new FileOutputStream("test.txt"));
         dataOutputStream.writeChar(i);
         dataOutputStream.close();
         DataInputStream dataInputStream =
               new DataInputStream(new FileInputStream("test.txt"));
         System.out.println(dataInputStream.readChar());
         dataInputStream.close();
      } catch (IOException e){
         e.printStackTrace();
      }
   }
}
```

Результат:

1

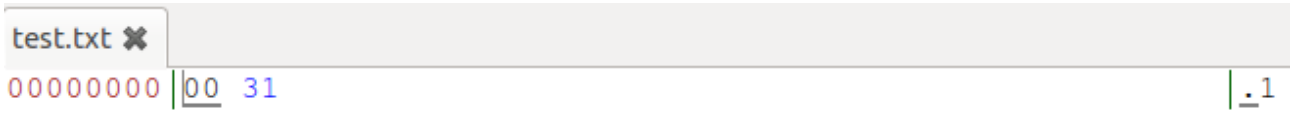


Рисунок – збереження char. Оскільки символ “1” входить до таблиці ASCII, у utf-16 старший байт – нульовий.

```
import java.io.*;
public class Test {
```

```
    public static void main(String[] args) {
        try {
            DataOutputStream dataOutputStream =
                    new DataOutputStream(new FileOutputStream("test.txt"));
            String s = "привіт hello";
            int l = s.length();
            dataOutputStream.writeInt(l);
            dataOutputStream.writeChars(s);
            dataOutputStream.close();
            DataInputStream dataInputStream =
                    new DataInputStream(new FileInputStream("test.txt"));
            int n = dataInputStream.readInt();
            for(int i = 0; i < n; i++)
               System.out.print(dataInputStream.readChar());
            dataInputStream.close();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

Результат: привіт hello

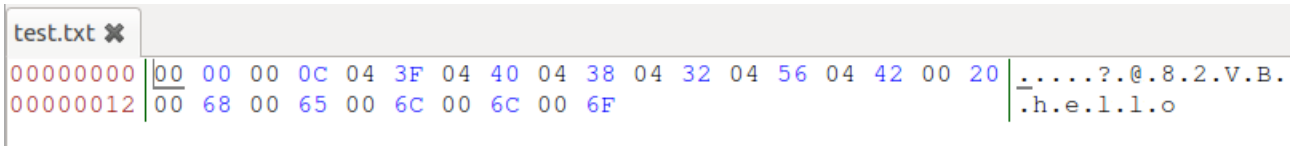


Рисунок – збереження String за допомогою writeChars(String s)

Зверніть увагу на те, що до файлу додатково додається інформація про кількість записаних char. Методу readChars() не існує, оскільки не відомо, скільки байт було записано попередньо.

Дане зроблено з метою демонстрації. Аналогічного результату можна досягнути, скориставшись методом available і поділивши результат на 2 (підставою є те, що кожен символ займає 2 байти).

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            DataOutputStream dataOutputStream =
                    new DataOutputStream(new FileOutputStream("test.txt"));
            String s = "привіт hello";
            dataOutputStream.writeBytes(s);
            dataOutputStream.close();
            DataInputStream dataInputStream =
```

```
            new DataInputStream(new FileInputStream("test.txt"));
        int n = dataInputStream.available();
        for(int i = 0; i < n; i++)
            System.out.printf("%h ", dataInputStream.read());
        dataInputStream.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
}
```

Результат:

3f 40 38 32 56 42 20 68 65 6c 6c 6f

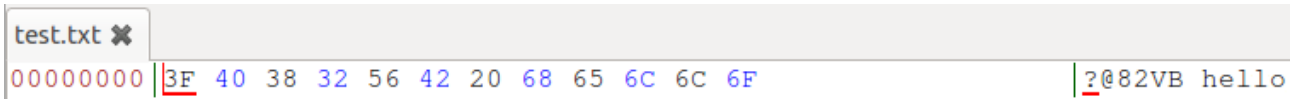


Рисунок – Збереження даних за допомогою writeBytes()

Оскільки попередньо було записано по 1 байтові до потоку, кількість

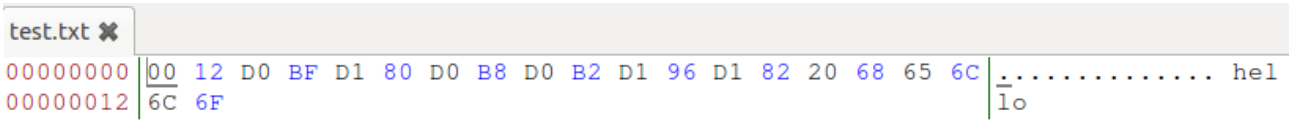
У попередньому прикладові символи кирилиці кодовані 2 байтами (так, як і латинські літери, проте для латинських літер з ASCII кодами до 128 старший байт дорівнює 0). Тобто, відновити кирилицю – можливо.

Даний приклад характерний тим, що старший байт при збереженні – відкинуто.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
        DataOutputStream dataOutputStream =
            new DataOutputStream(new FileOutputStream("test.txt"));
        String s = "привіт hello";
        dataOutputStream.writeUTF(s);
        dataOutputStream.close();
        DataInputStream dataInputStream =
            new DataInputStream(new FileInputStream("test.txt"));
        System.out.println(dataInputStream.readUTF());
        dataInputStream.close();
      } catch (IOException e){
        e.printStackTrace();
      }
    }
}
```

Результат:

привіт hello



Перші два байти – розмір даних, що записані у форматі modified UTF-8. $12^{(16)}=18^{(2)}$. Розмір файла – 20 байт, з них 2 – заголовок текстового блоку, 18 – текст.

ASCII control characters (character code 0-31)

Перших 32 символи таблиці ASCII – недруковані керуючі символи, їх використовують для контролю периферії. Такої, як принтери.

Dec	Hex	Symbol	Description	Dec	Hex	Symbol	Description
0	00	NUL	Null char	16	10	DLE	Data Line Escape
1	01	SOH	Start of Heading	17	11	DC1	Device Control 1 (oft. XON)
2	02	STX	Start of Text	18	12	DC2	Device Control 2
3	03	ETX	End of Text	19	13	DC3	Device Control 3 (oft. XOFF)
4	04	EOT	End of Transmission	20	14	DC4	Device Control 4
5	05	ENQ	Enquiry	21	15	NAK	Negative Acknowledgement
6	06	ACK	Acknowledgment	22	16	SYN	Synchronous Idle
7	07	BEL	Bell	23	17	ETB	End of Transmit Block
8	08	BS	Back Space	24	18	CAN	Cancel
9	09	HT	Horizontal Tab	25	19	EM	End of Medium
10	0A	LF	Line Feed	26	1A	SUB	Substitute
11	0B	VT	Vertical Tab	27	1B	ESC	Escape
12	0C	FF	Form Feed	28	1C	FS	File Separator
13	0D	CR	Carriage Return	29	1D	GS	Group Separator
14	0E	SO	Shift Out / X-On	30	1E	RS	Record Separator
15	0F	SI	Shift In / X-Off	31	1F	US	Unit Separator

ASCII printable characters (character code 32-127)

Коди 32-127 є спільними для усіх варіантів ASCII-таблиць, їх називають друкованими.

Dec	Hex	Символ	Опис	Dec	Hex	Символ	Опис
32	20		Space	80	50	P	

Dec	Hex	Символ	Опис	Dec	Hex	Символ	Опис
33	21	!	Exclamation mark	81	51	Q	
34	22	"	Double quotes (or speech marks)	82	52	R	
35	23	#	Number	83	53	S	
36	24	$	Dollar	84	54	T	
37	25	%	Procenttecken	85	55	U	
38	26	&	Ampersand	86	56	V	
39	27	'	Single quote	87	57	W	
40	28	(	Open parenthesis (or open bracket)	88	58	X	
41	29	)	Close parenthesis (or close bracket)	89	59	Y	
42	2A	*	Asterisk	90	5A	Z	
43	2B	+	Plus	91	5B	[	Opening bracket
44	2C	,	Comma	92	5C	\	Backslash
45	2D	-	Hyphen	93	5D	]	Closing bracket
46	2E	.	Period, dot or full stop	94	5E	^	Caret - circumflex
47	2F	/	Slash or divide	95	5F	_	Underscore
48	30	0	Zero	96	60	`	Grave accent
49	31	1	One	97	61	a	
50	32	2	Two	98	62	b	
51	33	3	Three	99	63	c	
52	34	4	Four	100	64	d	
53	35	5	Five	101	65	e	
54	36	6	Six	102	66	f	
55	37	7	Seven	103	67	g	
56	38	8	Eight	104	68	h	
57	39	9	Nine	105	69	i	
58	3A	:	Colon	106	6A	j	
59	3B	;	Semicolon	107	6B	k	
60	3C	<	Less than (or open angled bracket)	108	6C	l	
61	3D	=	Equals	109	6D	m	
62	3E	>	Greater than (or close angled bracket)	110	6E	n	
63	3F	?	Question mark	111	6F	o	
64	40	@	At symbol	112	70	p	
65	41	A		113	71	q	

Dec	Hex	Символ	Опис	Dec	Hex	Символ	Опис
66	42	B		114	72	r	
67	43	C		115	73	s	
68	44	D		116	74	t	
69	45	E		117	75	u	
70	46	F		118	76	v	
71	47	G		119	77	w	
72	48	H		120	78	x	
73	49	I		121	79	y	
74	4A	J		122	7A	z	
75	4B	K		123	7B	{	Opening brace
76	4C	L		124	7C	|	Vertical bar
77	4D	M		125	7D	}	Closing brace
78	4E	N		126	7E	~	Equivalency sign - tilde
79	4F	O		127	7F		Delete

OutputStreamWriter

OutputStreamWriter перетворює потік символів у потік байт з використанням вказаного кодування (charset). Кодування може бути вказано за його іменем, у явному вигляді або буде прийнято кодування за замовчуванням даної платформи.

Кожний виклик методу write() включає виклик конвертера над переданими йому символами. Отримані байти акумулюються у буфері перед записом до вихідного потоку. Символи, передані методові write(), не акумулюються.

Для максимальної ефективності зручно обгорнути OutputStreamWriter за допомогою BufferedWriter для уникнення частих викликів конвертера. Наприклад:

```
Writer out  = new BufferedWriter(new OutputStreamWriter(System.out));
```

Поля

lock – об’єкт, що використовують для синхронізації операцій над потоком

Конструктори

OutputStreamWriter(OutputStream out) Створє OutputStreamWriter із кодуванням за замовчуванням
OutputStreamWriter(OutputStream out, Charset cs) Створює OutputStreamWriter, що використовує кодування-параметр
OutputStreamWriter(OutputStream out, CharsetEncoder enc) Створює OutputStreamWriter, що використовує вказаний кодер
OutputStreamWriter(OutputStream out, String charsetName) Створює OutputStreamWriter, що використовує кодування, задане іменем

Методи

void	close() Закриває потік, вивільняючи його перед цим
void	flush() Вивільняє потік від даних
String	getEncoding() Повертає ім’я кодування, котре використовує потік
void	write(char[] cbuf, int off, int len) Записує з масиву char[] cbuf у потік len символів зі зміщенням від початку масиву off
void	write(int c) Записує char у потік.
void	write(String str, int off, int len) Записує у потік підрядок рядка str

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         OutputStreamWriter osr;
         Writer w = new BufferedWriter(
               osr = new OutputStreamWriter(
                     new BufferedOutputStream(
                           new FileOutputStream("test.txt")),
                     "Cp1251"
               )
         );
         w.write("привіт hello");
         System.out.println("encoding: " + osr.getEncoding());
         w.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Зверніть увагу на те, що у даному випадкові OutputStreamWriter – не анонімний клас. Змінну його оголошено попередньо, інакше не можна було б отримати доступ до методу getEncoding().

Результат:

encoding: Cp1251

Інформацію про кодування можна знайти тут

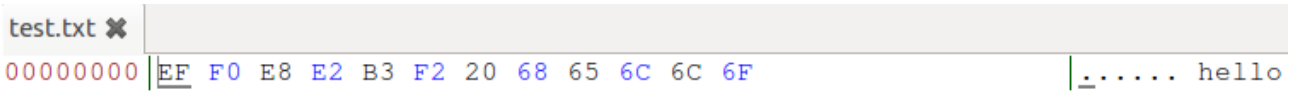


Рисунок – збереження даних за допомогою
OutputStreamWriter (кодування – “Windows-1251”)

Якщо змінити “Windows-1251” на “UTF-8”, отримаємо наступний результат:

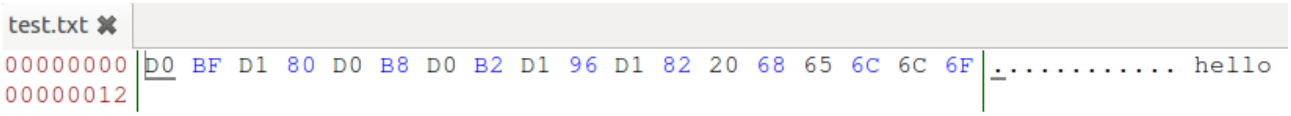


Рисунок – збереження даних за допомогою
OutputStreamWriter (кодування – “Windows-1251”)

Зверніть увагу на те, що латинка кодується 1 байтом на символ.

InputStreamReader

InputStreamReader поєднує потік байт з потоком символів. Потік читає байти, декодує їх у char, використовуючи задане кодування. Кодування може бути вказано за іменем, у явному вигляді або буде використано кодування за замочуванням.

Кожен виклик методу read() класу InputStreamReader вимагає отримання одного чи більше байт з байтового потоку, що лежить в основі читання. Для забезпечення ефективного декодування байт у символи може бути наперед прочитано більше байт, ніж потрібно на даний момент.

Максимальної ефективності можна досягнути шляхом огортання InputStreamReader за допомогою BufferedReader:

```
BufferedReader in  = new BufferedReader(new InputStreamReader(System.in));
```

Поле класу

lock – використовують для синхронізації операцій над потоком.

Конструктори

InputStreamReader(InputStream in) Creates an InputStreamReader that uses the default charset.
InputStreamReader(InputStream in, Charset cs) Creates an InputStreamReader that uses the given charset.
InputStreamReader(InputStream in, CharsetDecoder dec) Creates an InputStreamReader that uses the given charset decoder.
InputStreamReader(InputStream in, String charsetName) Creates an InputStreamReader that uses the named charset.

Методи

void	close() Closes the stream and releases any system resources associated with it.
String	getEncoding() Returns the name of the character encoding being used by this stream.
int	read() Reads a single character.
int	read(char[] cbuf, int offset, int length) Reads characters into a portion of an array.
boolean	ready() Tells whether this stream is ready to be read.

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         OutputStreamWriter isr;
         BufferedWriter w = new BufferedWriter(
               isr = new OutputStreamWriter(
                     new BufferedOutputStream(
                           new FileOutputStream("test.txt"){
                              @Override
                              public void close() throws IOException {
                                 super.close();
                                 System.out.println("closed");
                              }
                           }),
                     "Cp1251"
               )
         );
         StringBuilder sb = new StringBuilder();
         sb.append(String.format("%d\n", 1));
         sb.append(String.format("2 x 2 = %d", 2 * 2));
         w.write(sb.toString());
         System.out.println("encoding: " + isr.getEncoding());
         w.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Результат:

```
encoding: Cp1251
closed
```

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         InputStreamReader isr;
         BufferedReader r = new BufferedReader(
               isr = new InputStreamReader(
                     new BufferedInputStream(
                           new FileInputStream("test.txt"){
                              @Override
                              public void close() throws IOException {
                                 super.close();
                                 System.out.println("closed");
                              }
                           }),
                     "Cp1251"
               )
         );
         String line;
         while ((line = r.readLine()) != null)
            System.out.println(line);
         System.out.println("encoding: " + isr.getEncoding());
         r.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Результат:

```
1
2 x 2 = 4
encoding: Cp1251
closed
```

```
import java.io.*;
public class Test {
   public static void main(String[] args) {
      try {
         FileWriter fileWriter =
               new FileWriter("test.txt");
         final long startTime = System.nanoTime();
         for(int i = 0; i < 5000000; i++)
            fileWriter.write("7");
         final long duration = System.nanoTime() - startTime;
         System.out.printf("%.3f s", duration / 1000000000.0);
```

```
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Результат:

0.479 s

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            BufferedWriter bufferedWriter =
                    new BufferedWriter(
                    new FileWriter("test.txt"));
            final long startTime = System.nanoTime();
            for(int i = 0; i < 5000000; i++)
                bufferedWriter.write("д");
            final long duration = System.nanoTime() - startTime;
            System.out.printf("%.3f s", duration / 1000000000.0);
            bufferedWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Результат:

0.218 s

PrintWriter

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            PrintWriter printWriter =
                    new PrintWriter("test.txt");
            printWriter.printf("привіт %d", 12345);
            printWriter.close();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

Результат:

test.txt

```
00000000  D0 BF D1 80 D0 B8 D0 B2 D1 96 D1 82 20 31 32 33 34 35  ..........  12345
00000012
```

Рисунок

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            PrintWriter printWriter =
                    new PrintWriter("test.txt", "Cp1251");
            printWriter.printf("привіт %d", 12345);
            printWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Результат:

test.txt

```
00000000  EF F0 E8 E2 B3 F2 20 31 32 33 34 35              ......  12345
```

Рисунок

УПРАВЛІННЯ ПОТОКАМИ

Детальніше: http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

1. Створення та запуск потоку

Створіть клас MyCalculations:

```
public class MyCalculations implements Runnable {
   private int number;
   public MyCalculations(int number) {
      this.number = number;
   }
   @Override
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.printf("%s: %d * %d = %d\n", Thread.
               currentThread().getName(), number, i, i * number);
      }
   }
}
```

Додайте до проекту клас Main:

```
public class Main {
   public static void main(String[] args) {
      for (int i = 0; i <=5; i++) {
         MyCalculations calculations = new MyCalculations(i);
         Thread thread = new Thread(calculations);
         thread.start();
      }
   }
}
```

Створені потоки виконують свою роботу паралельно.

Кожна Java-програма має, принаймні, один потік. При запускові програми JVM виконує цей потік, котрий викликає метод main().

Кожен черговий виклик методу start() створює новий потік. Кількість потоків програми дорівнює кількості викликів методу start().

Виконання програми завершиться тоді, коли усі її потоки (non-daemon) завершать виконання. Якщо початковий потік (той, котрий виконує метод main()) завершив роботу, інші потоки продовжують виконувати роботу доки не завершать її.

Якщо один із потоків використає інструкцію System.exit(), усі потоки припинять виконання.

Додайте наприкінці методу run() наступний рядок

```
if(Thread.currentThread().getName().equals("Thread-2")) System.exit(0);
```

Результат буде схожим на наступний (щоразу він буде іншим):

```
Thread-0: 0 * 0 = 0
Thread-4: 4 * 0 = 0
Thread-5: 5 * 0 = 0
Thread-3: 3 * 0 = 0
Thread-3: 3 * 1 = 3
Thread-3: 3 * 2 = 6
Thread-3: 3 * 3 = 9
Thread-3: 3 * 4 = 12
Thread-2: 2 * 0 = 0
```

Аналогічного результату (**при цьому між варіантами існує принципова відмінність**) можна досягти, якщо оголосити клас, що спадкує Thread, реалізувати у ньому @override методу run(), а у головному потоці створити екземпляр цього класу. Метод start(), викликаний через даний екземпляр, запустить новий потік.

```
public class MyCalculations extends Thread {
   private int number;
   public MyCalculations(int number) {
      this.number = number;
   }
   @Override
   public void run() {
      for (int i = 0; i < 5; i++) {
         System.out.printf("%s: %d * %d = %d\n", Thread.
               currentThread().getName(), number, i, i * number);
      }
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      for (int i = 0; i < 5; i++) {
         MyCalculations calculations = new MyCalculations(i);
         calculations.start();
      }
   }
}
```

2. Переривання виконання потоку

Java-програма з більш ніж одним потоком завершує виконання, коли усі її потоки

завершать роботу (точніше, коли усі її потоки-не демони завершать роботу або коли один з потоків викличе метод System.exit()).

Часом потрібно завершити виконання потоку для завершення роботи програми або завершення завдання, що виконує потік.

Java надає механізм переривання для повідомлення потоку про те, що йому слід завершити роботу. Особливістю такого механізму є те, що потік повинен перевірити, чи його роботу бажають перервати, чи ні. Потік сам вирішує, чи реагувати йому на запит завершення роботи, чи не проігнорувати його і продовжити виконання.

Розглянемо приклад:

```
public class MyThread extends Thread {
   @Override
   public void run() {
      while (true) {
         if (isInterrupted()) {
            System.out.printf("Виконання потоку - перервано");
            return;
         }
      }
   }
}
```

Метод run() класу не виконує ніякої роботи окрім того, що перевіряє, чи отримано запит на припинення роботи потоку.

Надішлемо йому такий запит у класі Main:

```
public class Main {
   public static void main(String[] args) {
      Thread task = new MyThread();
      task.start();
      try {
         Thread.sleep(5000);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      task.interrupt();
   }
}
```

Результат: через 5 с у консоль потік виводить повідомлення

```
/usr/local/java/jdk1.8.0_66/bin/java
Виконання потоку - перервано
```

Додамо деякого функціоналу до потоку:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        int delay = 0;
        while (true) {
            try {
                Thread.sleep(500);
                delay += 500;
                System.out.println("delay: " + delay);
            } catch (InterruptedException e) {
                //e.printStackTrace();
                // на цей момент  interrupt status == false
                System.out.println("Sleep потоку було перервано іншим потоком");
                return;
            }
            if (isInterrupted()) {
                System.out.println("Виконання потоку - перервано");
                return;
            }
        }
    }
}
```

Результат:
delay: 500
delay: 1000
delay: 1500
delay: 2000
delay: 2500
delay: 3000
delay: 3500
delay: 4000
delay: 4500
Sleep потоку було перервано іншим потоком

Переривання Sleep відбувається негайно щойно надсилається іншим потоком. Потік надсилає переривання шляхом виклику interrupt над об'єктом Thread потоку, дія якого повинна бути перервана. Для того, щоб механізм переривання працював коректно, потік повинне підтримувати власне переривання. Потік може сам перервати власне виконання.

https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

Яким чином потік підтримує власне переривання? Це залежить від того, що він виконує у даний момент. Якщо потік часто викликає методи, здатні генерувати InterruptedException, він повертається з методу run після обробки виняткової ситуації.

Багато методів, що генерують InterruptedException, таких, як sleep, сконструйовані таким чином, що переривають власне виконання негайно, якщо надходить запит на переривання.

3. Interrupt Status Flag

Механізм переривань імплементовано шляхом використання внутрішнього флагу, котрий називають interrupt status. Виклик Thread.interrupt встановлює цей флаг. Коли потік перевіряє переривання, викликаючи статичний метод Thread.interrupted, флаг переривання скидається. Нестатичний метод isInterrupted, котрий використовують для перевірки статусу одного потоку іншим, не змінює значення статусного флага.

Довільний метод, генеруючи InterruptedException, скидає interrupt. Проте, довільний потік, що викликає переривання, може одразу встановити його знову.

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupted--

interrupted()

public static boolean interrupted()

Перевіряє, чи поточний потік було перервано. Скидає флаг *interrupted status*. Іншими словам, якщо метод буде викликано двічі, другий виклик поверне false, якщо виконання даного потоку не було перервано ще раз після першого виклику interrupted().

Повертає:

true, якщо даний потік було перервано, інакше — false.

isInterrupted()

public boolean isInterrupted()

Перевіряє, чи поточний потік було перервано. Не змінює значення флагу *interrupted status*.

Повертає:

true, якщо даний потік було перервано, інакше — false.

4. Контролювання переривання потоків

Розглянемо приклад з книги Java 7 Concurrency Cookbook (Javier Fernández González, Packt Publishing, 2012, C. 15 — 18).

```
import java.util.concurrent.TimeUnit;
public class Main {
   public static void main(String[] args) {
      FileSearch searcher = new FileSearch("/home/user/", "Retrofit.pdf");
      Thread thread = new Thread(searcher);
      thread.start();
      try {
         TimeUnit.SECONDS.sleep(30);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
      thread.interrupt();
   }
}
```

```
import java.io.File;
public class FileSearch implements Runnable{
   private String initPath;
   private String fileName;
   public FileSearch(String initPath, String fileName) {
      this.initPath = initPath;
      this.fileName = fileName;
   }
   @Override
   public void run() {
      File file = new File(initPath);
      if (file.isDirectory()) {
         try {
            directoryProcess(file);
         } catch (InterruptedException e) {
            System.out.printf("%s: The search has been " +
                  "interrupted",Thread.currentThread().getName());
         }
      }
   }
   private void directoryProcess(File file) throws
         InterruptedException {
      File list[] = file.listFiles();
      if (list != null) {
         for (int i = 0; i < list.length; i++) {
            if (list[i].isDirectory()) {
               directoryProcess(list[i]);
            } else {
               fileProcess(list[i]);
            }
         }
      }
      if (Thread.interrupted()) {
         throw new InterruptedException();
      }
   }
   private void fileProcess(File file) throws InterruptedException
   {
      if (file.getName().equals(fileName)) {
         System.out.printf("%s : %s\n",Thread.currentThread().
               getName() ,file.getAbsolutePath());
      }
      if (Thread.interrupted()) {
         throw new InterruptedException();
      }
   }
}
```

Результат:

```
Thread-0 : /home/user/Retrofit.pdf
```

Thread-0 : /home/user/android-sdk-linux/Retrofit.pdf
Thread-0: The search has been interrupted

Метод main класу Main запускає потік, оголошений у класі FileSearch. При цьому відбувається рекурсивний пошук файлу з потрібним іменем. Початкову адресу та ім'я файлу задають за допомогою конструктора FileSearch.

Методи directoryProcess та fileProcess перевіряють, чи не відбулося переривання роботи потоку за допомогою методу interrupted(). Якщо флаг переривання встановлено, метод interrupted() скидає його і програма генерує виняткову ситуацію InterruptedException, яка обробляється методом run().

5. Засинання та відновлення роботи потоків

https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep-long-

Для призупинення роботи потоку на заданий період часу використовують метод Thread.Sleep() або sleep() елемента TimeUnit.

sleep
public static void sleep(long millis) throws InterruptedException

Змушує поточний потік “заснути” (тимчасово припинити виконання) на задану кількість мілісекунд. Точність залежить від системних таймерів та планувальників задач.

Параметри:
millis — період часу у мілісекундах.
Виняткові ситуації:
IllegalArgumentException — у випадкові, якщо параметр має від'ємне значення;
InterruptedException — якщо довільний потік перервав виконання даного потоку. Флаг *interrupted status* даного потоку скидається у випадкові появи цієї виняткової ситуації.

sleep
public static void sleep(long millis, int nanos) throws InterruptedException

Змушує поточний потік “заснути” (тимчасово припинити виконання) на задану кількість мілісекунд + задану кількість наносекунд. Точність залежить від системних таймерів та планувальників задач.

Пераметри: The thread does not lose ownership of any monitors.
Parameters:
millis — час у мс;
nanos — 0-999999 додаткові нс часу засинання.
Виняткові ситуації:
IllegalArgumentException — у випадкові, якщо параметр має від'ємне значення або кількість нс не належить діапазонові 0-999999;
InterruptedException — якщо довільний потік перервав виконання даного потоку. Флаг *interrupted status* даного потоку скидається у випадкові появи цієї виняткової ситуації.

Перелічуваний тип TimeUnit
https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html

TimeUnit представляє інтервали часу із заданим рівнем гранулярності. TimeUnit не виконує

жодних дій з інформацією, а тільки допомагає організувати її. TimeUnit використовують для інформування методів про те, як інтерпретувати їхні часові параметри. Константи — наступні:

DAYS — одиниця часу, котра представляє двадцять чотири години;
HOURS — одиниця часу, котра представляє шістдесят хвилин;
MICROSECONDS — одиниця часу, котра представляє одну тисячну мілісекунди;
MILLISECONDS — одиниця часу, котра представляє одну тисячну секунди;
MINUTES — одиниця часу, котра представляє шістдесят хвилин;
NANOSECONDS — одиниця часу, котра представляє одну тисячну мікросекунди;
SECONDS — одиниця часу, котра представляє одну секунду.

Зверніть увагу на те, що немає гарантії того, що конкретна імплементація затримки буде здатною відчути перебіг часу на рівні гранулярності, заданому TimeUnit.

6. Очікування завершення роботи потоку (join)

Метод join() дозволяє потокові очікувати на завершення роботи іншого потоку. Якщо t — об'єкт Thread, потік котрого у даний момент виконується, у результаті

```
t.join();
```

даний потік повинен призупинити виконання доти, доки потік t завершить свою роботу. Перевантаження join дозволяє вказати період очікування. Проте, як і sleep, join залежить від синхронізації операційної системи, тому не слід очікувати, що join буде очікувати саме вказаний період. Як і sleep, join відповідає на переривання з InterruptedException.

Приклад з https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

```
public class SimpleThread {
   // Display a message, preceded by
   // the name of the current thread
   static void threadMessage(String message) {
      String threadName =
            Thread.currentThread().getName();
      System.out.format("%s: %s%n",
            threadName,
            message);
   }
   private static class MessageLoop
         implements Runnable {
      public void run() {
         String importantInfo[] = {
               "Рік починається у січні",
               "за січнем йде лютий",
               "за лютим - березень",
               "за березнем - квітень"
         };
         try {
            for (int i = 0;
```

```
                i < importantInfo.length;
                i++) {
                // Pause for 4 seconds
                Thread.sleep(4000);
                // Print a message
                threadMessage(importantInfo[i]);
            }
        } catch (InterruptedException e) {
            threadMessage("Мене - зупинили!");
        }
    }
}
public static void main(String args[])
        throws InterruptedException {
    // Delay, in milliseconds before
    // we interrupt MessageLoop
    // thread (default one hour).
    long patience = 1000 * 60 * 60;
    // If command line argument
    // present, gives patience
    // in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }
    threadMessage("Запускаємо потік MessageLoop");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    threadMessage("Очікуємо на завершення потоку MessageLoop");
    // loop until MessageLoop
    // thread exits
    while (t.isAlive()) {
        threadMessage("А снігу все нема...");
        // Wait maximum of 1 second
        // for MessageLoop thread
        // to finish.
        t.join(1000);
        if (((System.currentTimeMillis() - startTime) > patience)
                && t.isAlive()) {
            threadMessage("Змучилися чекати!");
            t.interrupt();
            // Shouldn't be long now
            // -- wait indefinitely
            t.join();
        }
```

```
        }
        threadMessage("Нарешті!");
    }
}
```

Результат виконання:

```
main: Запускаємо потік MessageLoop
main: Очікуємо на завершення потоку MessageLoop
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
Thread-0: Рік починається у січні
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
Thread-0: за січнем йде лютий
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
Thread-0: за лютим - березень
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
main: А снігу все нема...
Thread-0: за березнем - квітень
main: Нарешті!
```

7. Синхронізація

Потоки взаємодіють, надаючи доступ до полів та посилань на об'єкти, на які вказують поля. Така форма взаємодії є ефективною, проте уможливлює 2 типи помилок: інтерференцію потоків та помилки черговості.

Інструмент, який запобігає появі таких помилок, називається синхронізацією. Разом з тим, синхронізація може викликати конкуренцію потоків, котра виникає тоді, коли два або більше потоків намагаються отримати доступ до деякого ресурсу одночасно і у результаті програма виконується повільно або, навіть, взагалі припиняє виконання.

8. Інтерференція потоків

https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html

Розглянемо деякий клас

```
class Counter {
   private int c = 0;

   public void increment() {
      c++;
   }

   public void decrement() {
      c--;
   }

   public int value() {
      return c;
   }

}
```

Інтерференція виникає тоді, коли дві операції, що виконуються у різних потоках, діють над одними і тими ж даними. Наприклад, одночасно виконують методи increment() та decrement(). Така ситуація була б неможливою, якщо б ці методи були неподільними, проте вони складаються із декількох кроків.

9. Помилки консистентності пам'яті

Помилки консистентності (узгодженості) трапляються тоді, коли різні потоки не мають чіткого уявлення про те, чим повинні бути одні і ті ж дані.

Ключ для уникнення помилок консистентності — розуміння зв'язку “happens-before” (“трапилося до”).

Приклад: використання start() та join().

10. Синхронізовані методи

https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

Java надає 2 ідіоми синхронізації: синхронізовані методи та синхронізовані блоки. Для того, щоб метод був синхронізованим, слід додати ключове слово synchronized до його оголошення:

```
public class SynchronizedCounter {
   private int c = 0;

   public synchronized void increment() {
      c++;
   }

   public synchronized void decrement() {
```

```
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Якщо count — екземпляр SynchronizedCounter, то результатом оголошення методів синхронізованими є два ефекти:

- стає неможливою інтерференція потоків. Якщо один потік виконує синхронізований метод для об'єкта, інші потоки, що викликають синхронізовані методи для того ж об'єкта, повинні дочекатися завершення виконання методу потоком, котрий отримав доступ до даних швидше;

- якщо синхронізований метод існує, він автоматично встановлює зв'язок “happens-before” з будь-яким іншим послідовним викликом синхронізованого методу для цього самого об'єкту. Дане гарантує, що зміни стану об'єкта будуть видимими для інших потоків.

Конструктор класу не може бути синхронізованим. Синхронізація конструктора не має сенсу, оскільки під час створення екземпляра класу тільки один потік може мати доступ до конструктора.

Конструюючи об'єкт, доступний різним класам, слід бути уважним і не поширювати посилання на цей об'єкт раніше, ніж він стане доступним.

Final-поля, котрі не можуть бути змінені після синхронізації, доступні після створення екземпляру не синхронізованим методам.

11. Синхронізовані оператори

Синхронізація у Java побудована навколо внутрішньої сутності, відомої як “внутрішнє блокування” або блокування за допомогою “моніторів”.

Монітори запобігають доступові до об'єкта, до котрого отримує доступ синхронізований метод.

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

У даному випадкові оператор addName потребує синхронізації змін до lastName та nameCount.

Синхронізовані блоки зручні для покращення конкурування потоків у випадкові добре гранульованих даних:

```
public class MsLunch {
```

```
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

12. Атомарні дії

У програмуванні атомарному дія є тією, яка ефективно відбувається раптово. Атомарна дія не може зупинитися посередині. Вона або відбувається повністю, або не відбувається узагалі. Жодні побічні дії атомарної операції не будуть видимі, поки вона не завершиться.

Приклад, операції читання-запису змінних-посилань та більшості примітивних типів — атомарні. Читання та запис усіх змінних, оголошених як volatile — атомарні.

А оператор ++ не є атомарним.

13. Живучість

Здатність паралельної програми бути виконаною вчасно, називається живучістю.

14. Тупик (Deadlock)

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
```

```
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
              + " has bowed back to me!%n",
              this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

15. Створення та запуск потоків-демонів

Java володіє спеціальним типом потоків — демонів. Такі потоки мають дуже низький пріоритет і зазвичай виконуються у випадкові, якщо на даний час не виконуються інші потоки процесу. Коли демон залишається єдиним потоком програми, котрий виконується, віртуальна машина припиняє роботу такого потоку.

Розглянемо приклад з книги Java 7 Concurrency Cookbook (Javier Fernández González, Packt Publishing, 2012, С. 24 — 27).

```
import java.util.ArrayDeque;
import java.util.Date;
import java.util.Deque;
import java.util.concurrent.TimeUnit;
public class Chap1 {
    private static class Event {
        public Date getDate() {
            return date;
        }
        public void setDate(Date date) {
            this.date = date;
        }
        public String getEvent() {
            return event;
```

```
    }
    public void setEvent(String event) {
        this.event = event;
    }
    private Date date;
    private String event;
}
public static class WriterTask implements Runnable {
    private Deque<Event> deque;
    public WriterTask(Deque<Event> deque) {
        this.deque = deque;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            Event event = new Event();
            event.setDate(new Date());
            event.setEvent(String.format("The thread %s has generated an event",
                    Thread.currentThread().getId()));
            synchronized(this) {
                deque.addFirst(event);
                System.out.println("deque size: " + deque.size() + " thread: " +
                        Thread.currentThread().getId());
            }
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
public static class CleanerTask extends Thread {
    private Deque<Event> deque;
    public CleanerTask(Deque<Event> deque) {
        this.deque = deque;
        setDaemon(true);
    }
    @Override
    public void run() {
        while (true) {
            Date date = new Date();
            clean(date);
        }
    }
    private void clean(Date date) {
        long difference;
        boolean delete;
        if (deque.isEmpty()) {
            return;
```

```
            }
            delete = false;
            do {
                Event e = deque.getLast();
                difference = date.getTime() - e.getDate().getTime();
                if (difference > 10000) {
                    System.out.printf("Cleaner: %s\n", e.getEvent());
                    deque.removeLast();
                    delete = true;
                }
            } while (difference > 10000);
            if (delete) {
                System.out.printf("Cleaner: Size of the deque: %d\n", deque.size());
            }
        }
    }
    public static void main(String[] args) {
        Deque<Event> deque = new ArrayDeque<>();
        WriterTask writerTask = new WriterTask(deque);
        for (int i = 0; i < 3; i++) {
            Thread thread = new Thread(writerTask);
            thread.start();
        }
        CleanerTask cleanerTask = new CleanerTask(deque);
        cleanerTask.start();
    }
}
```

Результат виконання:

```
deque size: 1 thread: 10
deque size: 2 thread: 11
deque size: 3 thread: 12
deque size: 4 thread: 10
deque size: 5 thread: 11
deque size: 6 thread: 12
deque size: 7 thread: 10
deque size: 8 thread: 12
deque size: 9 thread: 11
deque size: 10 thread: 10
deque size: 11 thread: 12
deque size: 12 thread: 11
deque size: 13 thread: 10
deque size: 14 thread: 12
deque size: 15 thread: 11
deque size: 16 thread: 10
deque size: 17 thread: 12
deque size: 18 thread: 11
deque size: 19 thread: 12
deque size: 20 thread: 10
```

```
deque size: 21 thread: 11
deque size: 22 thread: 10
deque size: 23 thread: 12
deque size: 24 thread: 11
deque size: 25 thread: 12
deque size: 26 thread: 10
deque size: 27 thread: 11
deque size: 28 thread: 12
deque size: 29 thread: 10
deque size: 30 thread: 11
Cleaner: The thread 10 has generated an event
Cleaner: The thread 11 has generated an event
Cleaner: Size of the deque: 28
Cleaner: The thread 12 has generated an event
Cleaner: Size of the deque: 27
deque size: 28 thread: 10
deque size: 29 thread: 11
deque size: 30 thread: 12
Cleaner: The thread 10 has generated an event
Cleaner: The thread 11 has generated an event
Cleaner: Size of the deque: 28
Cleaner: The thread 12 has generated an event
Cleaner: Size of the deque: 27
deque size: 28 thread: 10
deque size: 29 thread: 11
deque size: 30 thread: 12
(результат наведено не повністю)
```

16. Обробка неконтрольованих виняткових ситуації у потоках

У Java існують два типи виняткових ситуації:

checked
unchecked

```
public class Chap1 {
   public static class ExceptionHandler implements Thread.UncaughtExceptionHandler {
      @Override
      public void uncaughtException(Thread t, Throwable e) {
         System.out.printf("An exception  has been captured.\n");
         System.out.printf("Thread: %s\n", t.getId());
         System.out.printf("Exception: %s: %s\n", e.getClass().getName(), e.getMessage());
         System.out.printf("Stack Trace: \n");
         e.printStackTrace(System.out);
         System.out.printf("Thread status: %s\n", t.getState());
      }
   }
   public static class Task implements Runnable {
```

```
        @Override
        public void run() {
            int n = Integer.parseInt("TT");
        }
    }
    public static void main(String[] args) {
        Thread thread = new Thread(new Task());
        thread.setUncaughtExceptionHandler(new ExceptionHandler());
        thread.start();
    }
}
```

Результат:

```
An exception  has been captured.
Thread: 10
Exception: java.lang.NumberFormatException: For input string: "TT"
Stack Trace:
java.lang.NumberFormatException: For input string: "TT"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.lang.Integer.parseInt(Integer.java:580)
 at java.lang.Integer.parseInt(Integer.java:615)
 at Chap1$Task.run(Chap1.java:20)
 at java.lang.Thread.run(Thread.java:745)
Thread status: RUNNABLE
```

Якщо закоментувати thread.setUncaughtExceptionHandler(new ExceptionHandler()), результат буде наступним:

```
Exception in thread "Thread-0" java.lang.NumberFormatException: For input string: "TT"
 at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
 at java.lang.Integer.parseInt(Integer.java:580)
 at java.lang.Integer.parseInt(Integer.java:615)
 at Chap1$Task.run(Chap1.java:20)
 at java.lang.Thread.run(Thread.java:745)
```

17. Використання локальних змінних у потоках

```
import java.util.Date;
import java.util.concurrent.TimeUnit;
public class Chap1 {
    public static class UnsafeTask implements Runnable {
        private Date startDate;
        @Override
        public void run() {
            startDate = new Date();
```

```
            System.out.printf("Starting Thread: %s : %s\n",
                  Thread.currentThread().getId(), startDate);
            try {
               TimeUnit.SECONDS.sleep((int) Math.rint(Math.random() * 10));
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.printf("Thread finished: %s : %s\n",
                  Thread.currentThread().getId(), startDate);
         }
      }
      public static class SafeTask implements Runnable {
         private static ThreadLocal<Date> startDate = new ThreadLocal<Date>(){
            protected Date initialValue() {
               return new Date();
            }
         };
         @Override
         public void run() {
            System.out.printf("Starting Thread: %s : %s\n",
                  Thread.currentThread().getId(), startDate.get());
            try {
               TimeUnit.SECONDS.sleep((int) Math.rint(Math.random() * 10));
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            System.out.printf("Thread finished: %s : %s\n",
                  Thread.currentThread().getId(), startDate.get());
         }
      }
      public static void main(String[] args) {
//       UnsafeTask unsafeTask = new UnsafeTask();
//       for(int i = 0; i < 3; i++) {
//          Thread thread = new Thread(unsafeTask);
//          thread.start();
//
//          try {
//             TimeUnit.SECONDS.sleep(2);
//          } catch (InterruptedException e) {
//             e.printStackTrace();
//          }
//       }
         SafeTask safeTask = new SafeTask();
         for(int i = 0; i < 3; i++) {
            Thread thread = new Thread(safeTask);
            thread.start();
            try {
               TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
               e.printStackTrace();
```

```
            }
        }
    }
}
```

Результат виконання із безпечним доступом до змінних:

```
Starting Thread: 10 : Fri Apr 15 07:17:52 EEST 2016
Starting Thread: 11 : Fri Apr 15 07:17:54 EEST 2016
Starting Thread: 12 : Fri Apr 15 07:17:56 EEST 2016
Thread finished: 12 : Fri Apr 15 07:17:56 EEST 2016
Thread finished: 10 : Fri Apr 15 07:17:52 EEST 2016
Thread finished: 11 : Fri Apr 15 07:17:54 EEST 2016
```

Як видно, кожен потік завершує своє виконання із тим значенням змінної, з яким він почав виконання.

Змінимо код з безпечного на небезпечний, закоментувавши і розкоментувавши відповідні частини.

Результат:

```
Starting Thread: 10 : Fri Apr 15 07:20:04 EEST 2016
Starting Thread: 11 : Fri Apr 15 07:20:06 EEST 2016
Thread finished: 10 : Fri Apr 15 07:20:06 EEST 2016
Starting Thread: 12 : Fri Apr 15 07:20:08 EEST 2016
Thread finished: 11 : Fri Apr 15 07:20:08 EEST 2016
Thread finished: 12 : Fri Apr 15 07:20:08 EEST 2016
```

Легко помітити, що потік завершує виконання із тим значенням дати, з яким почав виконання останній на даний момент потік.

18. Групування потоків

```
import java.util.Date;
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class Chap1 {
   public static class SearchTask implements Runnable {
      private Result result;
      public SearchTask(Result result) {
         this.result = result;
      }
      @Override
      public void run() {
         String name = Thread.currentThread().getName();
         System.out.printf("Thread %s: Start\n", name);
         try {
            doTask();
            result.setName(name);
```

```
            } catch (InterruptedException e) {
                System.out.printf("Thread %s: Interrupted.\n", name);
                return;
            }
            System.out.printf("Thread %s: End\n", name);
        }
        private void doTask() throws InterruptedException {
            Random random = new Random(new Date().getTime());
            int value = (int)(random.nextDouble() * 10);
            System.out.printf("Thread %s : %d\n", Thread.currentThread().getName(), value);
            TimeUnit.SECONDS.sleep(value);
        }
    }
    private static class Result {
        private String name;
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
    }
    public static void main(String[] args) {
        ThreadGroup threadGroup = new ThreadGroup("Searcher");
        Result result = new Result();
        SearchTask searchTask = new SearchTask(result);
        for (int i = 0; i < 5; i++) {
            Thread thread = new Thread(threadGroup, searchTask);
            thread.start();
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.printf("Number of threads: %d\n", threadGroup.activeCount());
        System.out.printf("Information about the ThreadGroup\n");
        threadGroup.list();
        Thread[] threads = new Thread[threadGroup.activeCount()];
        threadGroup.enumerate(threads);
        for (int i = 0; i < threadGroup.activeCount(); i++) {
            System.out.printf("Thread %s: %s\n", threads[i].getName(), threads[i].getState());
        }
        waitFinish(threadGroup);
        threadGroup.interrupt();
    }
    private static void waitFinish(ThreadGroup threadGroup) {
        while (threadGroup.activeCount() > 9) {
            try {
                TimeUnit.SECONDS.sleep(1);
```

```
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
         }
      }
   }
```

Результат виконання:

```
Thread Thread-0: Start
Thread Thread-0 : 1
Thread Thread-1: Start
Thread Thread-1 : 4
Thread Thread-0: End
Thread Thread-2: Start
Thread Thread-2 : 3
Thread Thread-3: Start
Thread Thread-3 : 5
Thread Thread-4: Start
Thread Thread-4 : 5
Thread Thread-1: End
Number of threads: 3
Information about the ThreadGroup
Thread Thread-2: End
java.lang.ThreadGroup[name=Searcher,maxpri=10]
   Thread[Thread-2,5,Searcher]
   Thread[Thread-3,5,Searcher]
   Thread[Thread-4,5,Searcher]
Thread Thread-3: TIMED_WAITING
Thread Thread-4: TIMED_WAITING
Thread Thread-4: Interrupted.
Thread Thread-3: Interrupted.
```

19. Обробка неконтрольованих виняткових ситуації у групах потоків

```
import java.util.Random;
public class Chap1 {
   public static class MyThreadGroup extends ThreadGroup {
      public MyThreadGroup(String name) {
         super(name);
      }
      @Override
      public void uncaughtException(Thread t, Throwable e) {
         System.out.printf("The thread %s has thrown an exception\n", t.getId());
         e.printStackTrace(System.out);
         System.out.printf("Terminating the rest of threads\n");
         interrupt();
```

```
        }
    }
    public static class Task implements Runnable {
        @Override
        public void run() {
            int result;
            Random random = new Random(Thread.currentThread().getId());
            while (true) {
                result = 1000 / ((int) (random.nextDouble() * 1000));
                System.out.printf("%s : %d\n", Thread.currentThread().getId(), result);
                if (Thread.currentThread().isInterrupted()) {
                    System.out.printf("%d : interrupted\n", Thread.currentThread().getId());
                    return;
                }
            }
        }
    }
    public static void main(String[] args) {
        MyThreadGroup threadGroup = new MyThreadGroup("MyThreadGroup");
        Task task = new Task();
        for (int i = 0; i < 2; i++) {
            Thread t = new Thread(threadGroup, task);
            t.start();
        }
    }
}
```

Приклад результату:

```
10 : 1
11 : 1
11 : 2
11 : 1
11 : 38
10 : 3
10 : 16
11 : 5
...
...
...
The thread 11 has thrown an exception
...
...
...
java.lang.ArithmeticException: / by zero
        at Chap1$Task.run(Chap1.java:21)
        at java.lang.Thread.run(Thread.java:745)
Terminating the rest of threads
10 : interrupted
```

20. Використання фабрики для створення потоків

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.TimeUnit;

public class Chap1 {

   public static class MyThreadFactory implements ThreadFactory {

      private int counter;
      private String name;
      private List<String> stats;

      public MyThreadFactory(String name) {
         this.counter = 0;
         this.name = name;
         stats = new ArrayList<>();
      }

      @Override
      public Thread newThread(Runnable r) {
         Thread t = new Thread(r, name + "-Thread_" + counter);
         counter++;
         stats.add(String.format("Created thread %d with name %s on %s",
               t.getId(), t.getName(), new Date()));
         return t;
      }

      public String getStats() {
         StringBuffer buffer = new StringBuffer();
         for (String stat : stats) {
            buffer.append(stat);
            buffer.append("\n");
         }
         return buffer.toString();
      }
   }

   public static class Task implements Runnable {

      @Override
      public void run() {
         try {
            TimeUnit.SECONDS.sleep(1);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
```

```
        }
    }

    public static void main(String[] args) {
        MyThreadFactory factory = new MyThreadFactory("MyThreadFactory");

        Task task = new Task();

        Thread thread;
        System.out.printf("Starting the threads\n");
        for (int i = 0; i < 10; i++) {
            thread = factory.newThread(task);
            thread.start();
        }

        System.out.printf("Factory stats:\n");
        System.out.printf("%s\n", factory.getStats());
    }
}
```

РОБОТА З БАЗАМИ ДАНИХ

У 1996 р. компанія Sun Microsystems випустила першу версію прикладного інтерфейсу API для організації доступу з програм Java до баз даних JDBC. Цей інтерфейс дозволяє з'єднуватися з базою даних, запитувати і оновлювати дані за допомогою мови структурованих запитів (Structured Query Language — SQL).

Виберіть пункт меню File → Project Structure. У відповідному вікні виберіть пункт Libraries (рис. 1).

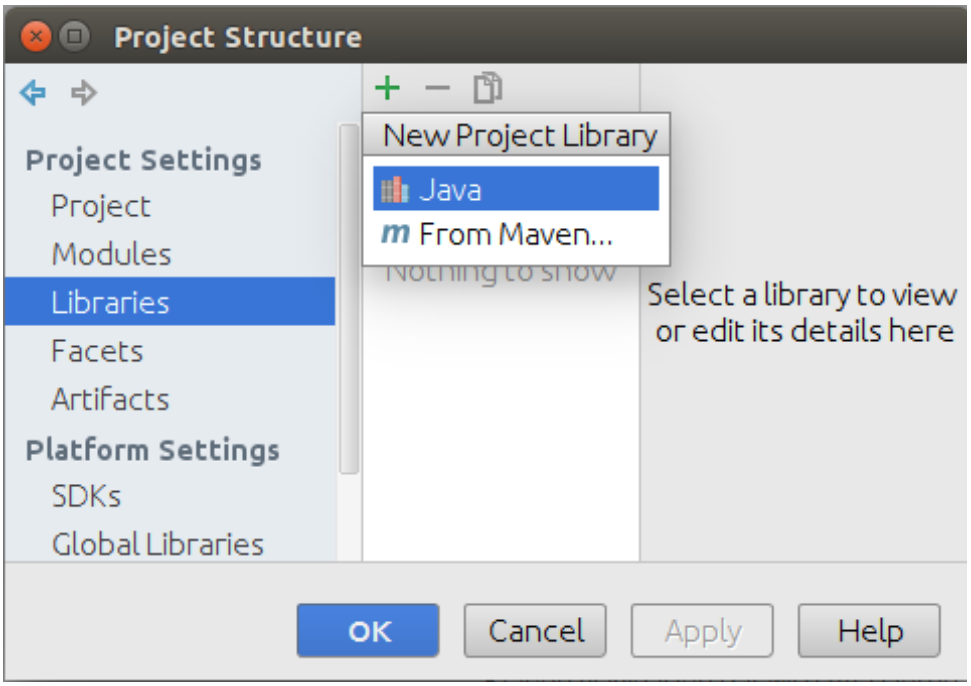


Рисунок 1 — Під'єднання бібліотек до проекту

Натисніть піктограму “+”. Зі списку New Project Library виберіть пункт Java. У вікні Select Library Files виберіть JDBC — драйвер. Для Linux шлях до драйвера повинен бути /usr/share/java (рис. 2).

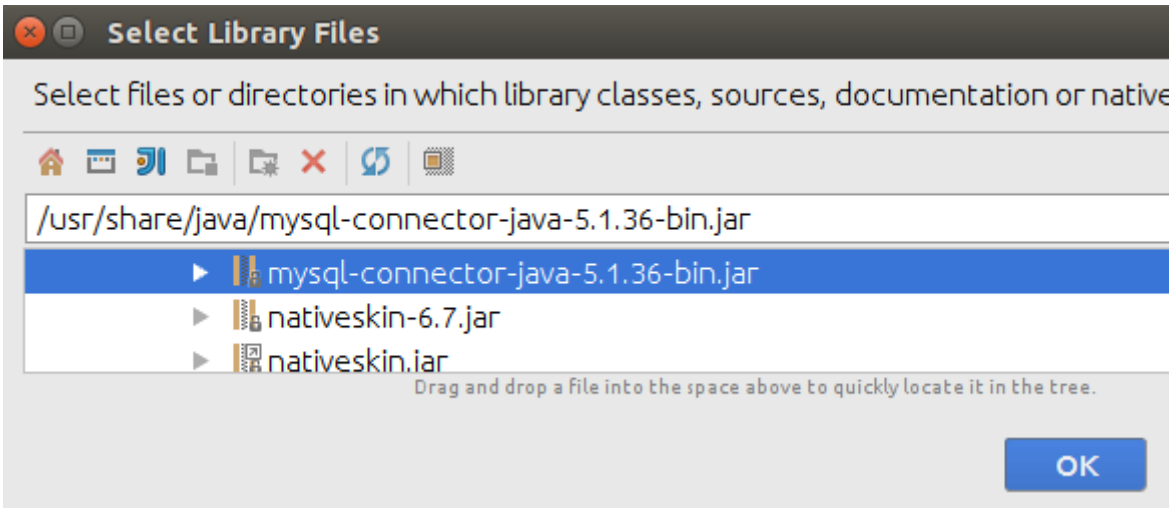


Рисунок 2 — Під'єднання драйвера JDBC для MySQL

http://dev.mysql.com/doc/workbench/en/wb-table-editor-columns-tab.html

3 Створення таблиці бази даних

Операції із таблицями бази даних можна здійснювати безпосередньо за допомогою консолі MySQL. На даному етапі скористаємося для наочності MySQL Workbench.

Завантажте MySQL Workbench. При першому запускові натисніть піктограму “+” (рис. 1). Ця дія необхідна для сторення та налаштування під'єднання до бази даних.

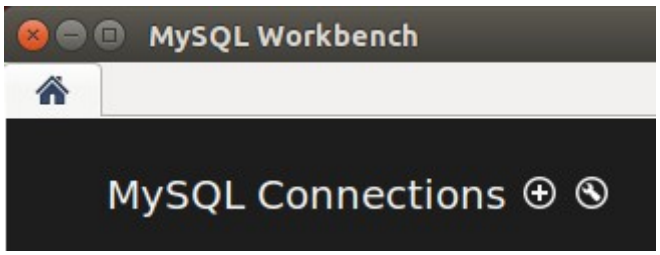


Рисунок 3 — Елемент стартового екрану SQL Workbench

Заповніть відповідні поля вікна Setup New Connection:

- Connection Name: задайте довільне ім'я, наприклад, MyConnection;
- Connection Method - Standard (TCP/IP);
- Hostname: 127.0.0.1, Port: 3306 (це стандартні значення);
- Username: ім'я користувача. Залежить від налаштувань MySQL;
- Password: пароль; можна не задавати його тут, тоді слід буде вводити пароль на вимогу;
- Default Schema: ім'я бази даних. Можна залишити поле порожнім і вибрати базу даних пізніше.

Натиснувши кнопку Test Connection можна перевірити правильність налаштування з'єднання

На цьому налаштування з'єднання із сервером MySQL завершено і можна перейти до створення та наповнення таблиці із даними.

Натисніть кнопку OK.

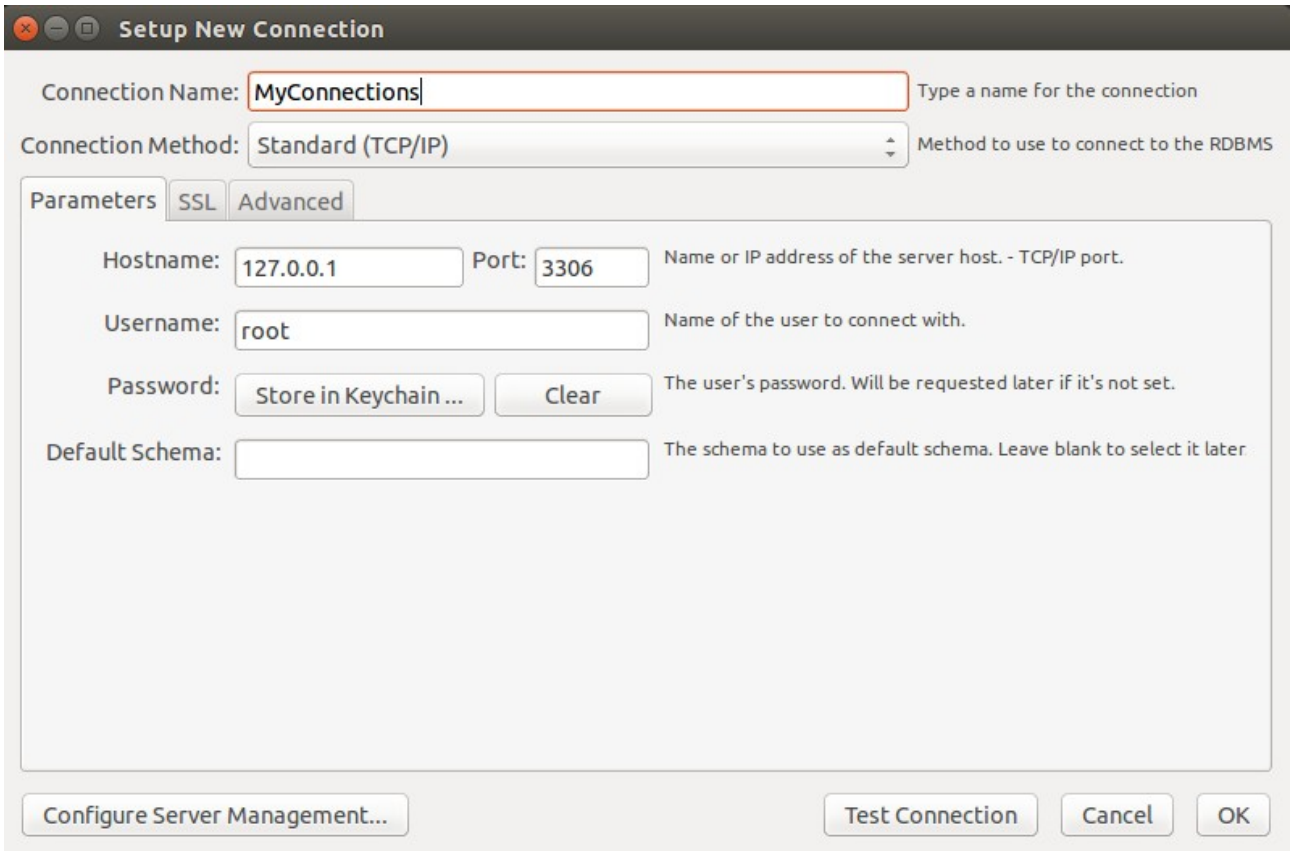


Рисунок 4 — Налаштування з'єднання із сервером MySQL

Рисунок 5 — Аутентифікація при перевірці з'єднання

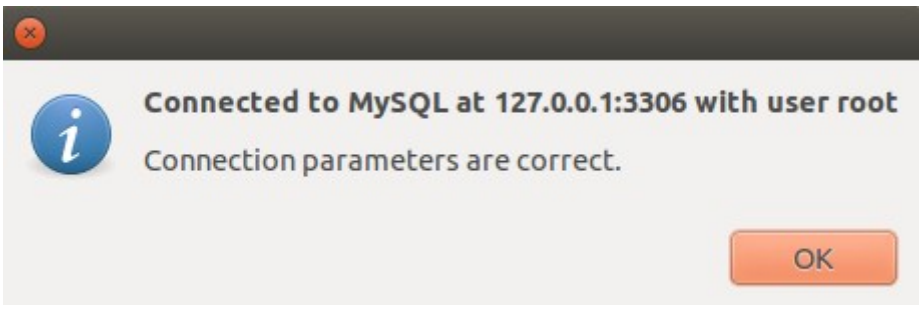


Рисунок 6 — Підтвердження правильності налаштувань з'єднання

Після створення з'єднання стартовий екран MySQL Workbench міститиме піктограму у вигляді прямокутника із відповідним іменем (рис. 11).

Клікніть по піктограмі. Після проходження користувачем аутентифікації відкриється вікно MySQL Workbench .

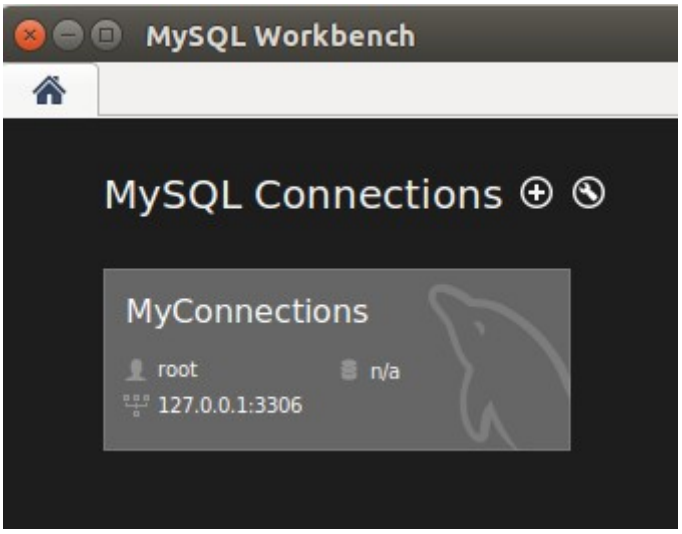


Рисунок 7 — Нове з'єднання

Рисунок 7 — Таблиці MySQL

Розкрийте тестову базу даних test, натиснувши на трикутник зліва від імені бази даних. Виберіть пункт Tables, натисніть праву кнопку миші і, вибравши Create Table, перейдіть до створення таблиці (рис. 8). Задайте ім'я таблиці та оголосіть її поля.

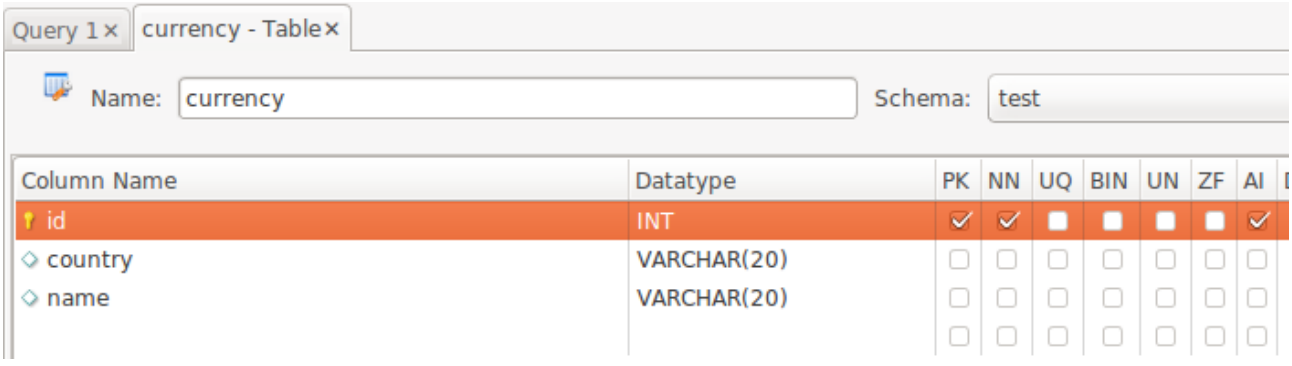


Рисунок 8 — Оголошення полів таблиці

Призначення флагів:

PK — Primary Key;
NN — Not Null;
UQ — Unique Key;
BIN — Binary
UN — Unsigned;
ZF — Zero-Filled;
AI — Auto Increment.

Для додавання чергового поля клікніть по рядкові під поточним. Після завершення редагування натисніть кнопку Apply. При цьому відкриється вікно підтвердження застосування SQL-скрипта до бази даних (рис. 9). Натисніть кнопку Apply. Успішність виконання скрипта підтверджує відповідне вікно (рис. 10).

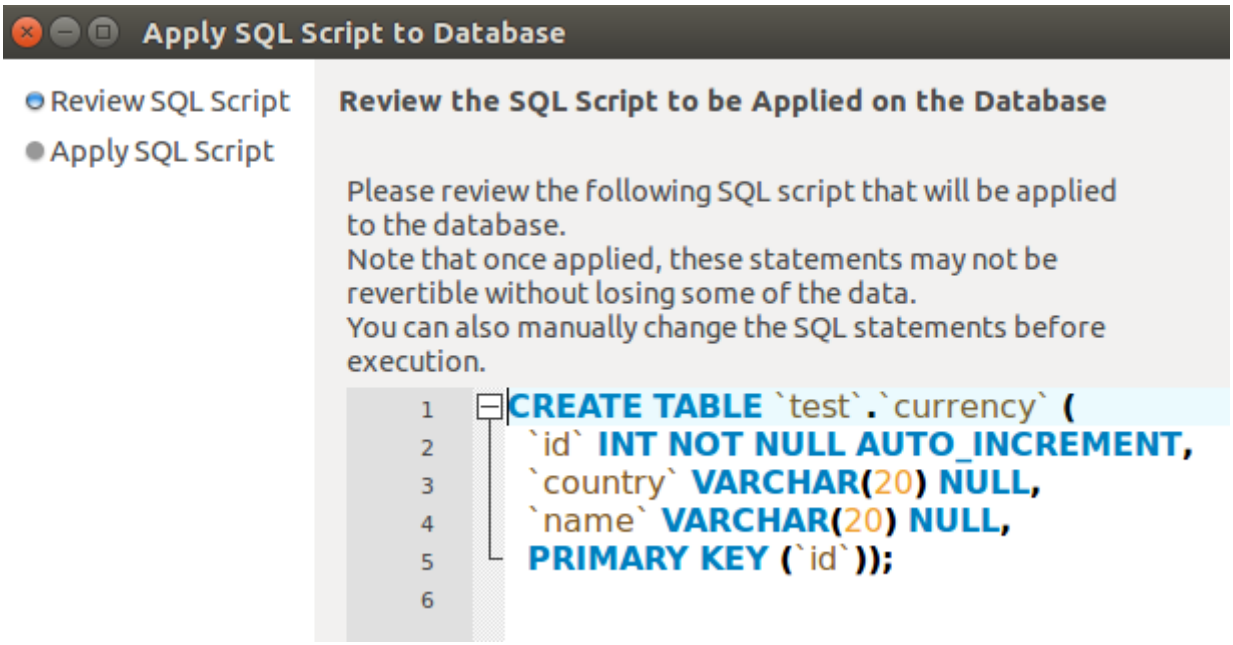


Рисунок 9 — Підтвердження застосування SQL-скрипта

CREATE TABLE `test`.`currency` (`id` INT NOT NULL AUTO_INCREMENT,
`country` VARCHAR(20) NULL, `name` VARCHAR(20) NULL, PRIMARY KEY (`id`));

Рисунок 10 — Підтвердження успішного застосування SQL-скрипта

4 Наповнення таблиці бази даних

Виконайте наступний запит до таблиці `test`.`currency` (рис. 11):

SELECT * FROM `test`.`currency`

Заповніть поля таблиці у відповідності до рис. 17. У кінці натисніть піктограму, що позначає блискавку.

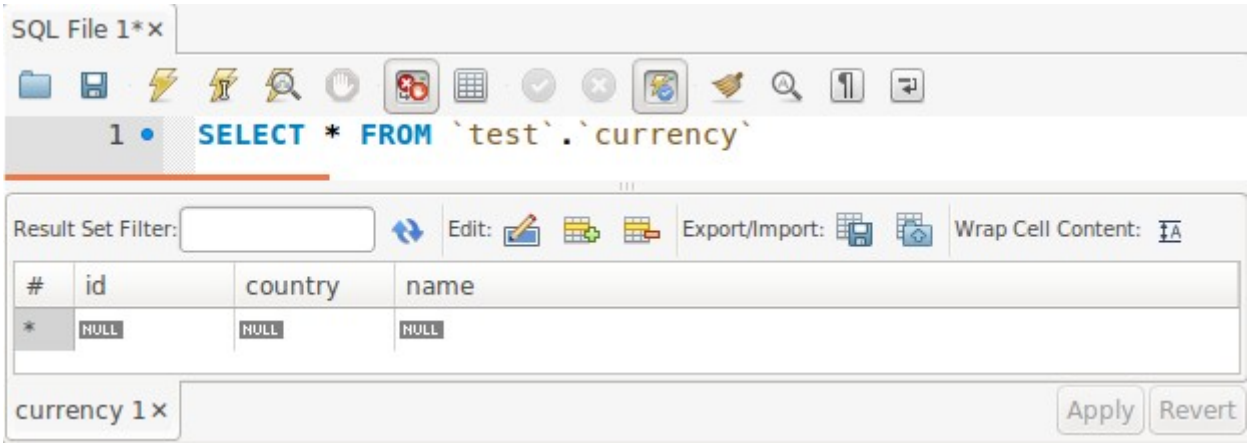


Рисунок 11 — Виконання запиту

Використовуйте подвійний клік лівої кнопки миші при наведеному курсорові на потрібне поле таблиці. Завершує редагування комірки кнопка клавіатури Enter.

Перехід між полями можна здійснювати також за допомогою кнопки Tab.

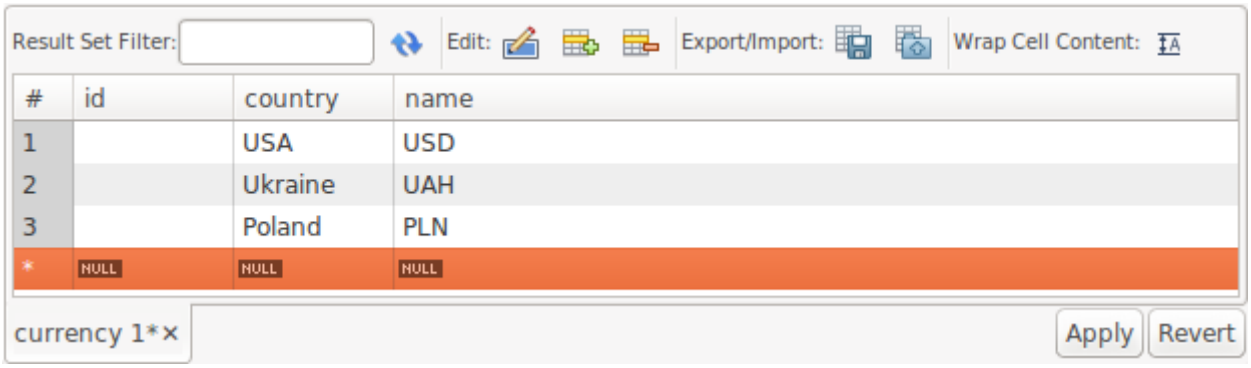


Рисунок 12 — Наповнення полів таблиці даними

Після завершення натисніть кнопку Apply. При цьому будуть виконані SQL-інструкції, про що свідчить підтверджуюче вікно (рис. 13):

```
INSERT INTO `test`.`currency` (`country`, `name`) VALUES ('USA', 'USD');
INSERT INTO `test`.`currency` (`country`, `name`) VALUES ('Ukraine', 'UAH');
INSERT INTO `test`.`currency` (`country`, `name`) VALUES ('Poland', 'PLN');
```

Прийміть зміни.

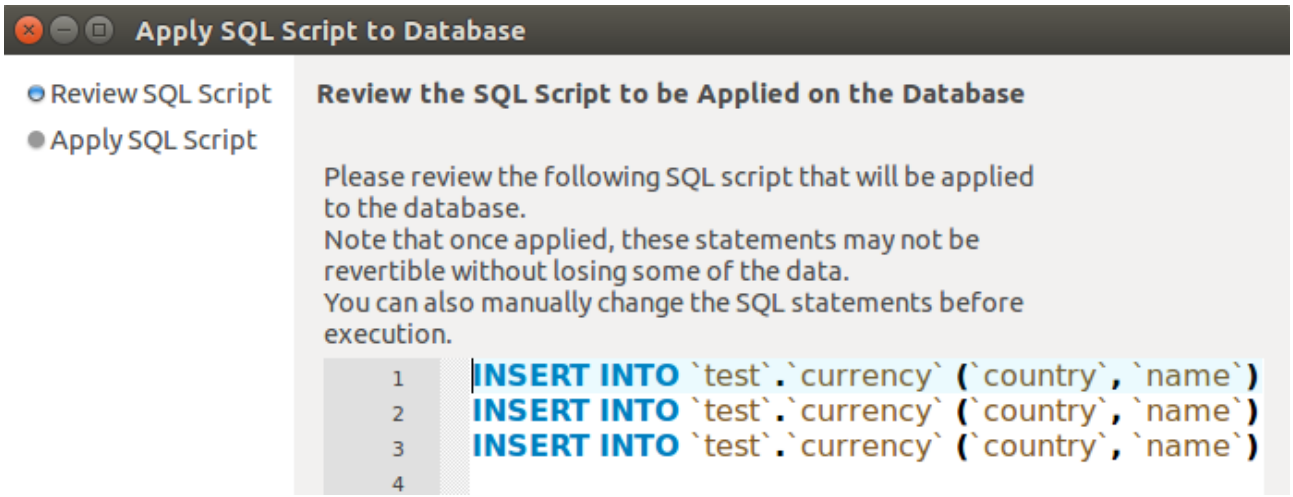


Рисунок 13 — Підтвердження застосування SQL-скрипта

5 Приклад

```
import java.sql.*;
class MysqlCon {
   public static void main(String args[]) {
         Connection connection = null;
         try {
            connection = DriverManager.
                  getConnection("jdbc:mysql://localhost:3306/test", "root", "password");
         } catch (SQLException e) {
            System.out.println("Connection Failed!");
            e.printStackTrace();
            return;
         }
         if (connection != null) {
            System.out.println("Connection Established\n");
         } else {
            System.out.println("Failed to make a connection");
         }
         try {
            Statement stmt = connection.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM currency");
            while (rs.next())
               System.out.println(
                     rs.getInt(1) + "  " +
                     rs.getString(2) + "  " +
                     rs.getString(3));
            connection.close();
         } catch (SQLException e) {
            System.out.println(e);
         }
   }
}
```

Розглянемо приклад детальніше. Клас *DriverManager* — основний сервіс для

управління набором JDBC драйверів. Зверніть увагу на те, що додаткам більше не потрібно завантажувати JDBC драйвери у явному вигляді за допомогою Class.forName(): (http://docs.oracle.com/javase/7/docs/api/java/sql/DriverManager.html).

Виклик методу getConnection() класу DriverManager ініціює спробу DriverManager знайти відповідний драйвер між драйверами, завантаженими при ініціалізації та завантаженими у явному вигляді за допомогою того самого завантажувача класів, що використовують аплет чи додаток:

public static Connection *getConnection(*
String *url, Properties info) throws SQLException*

Намагається встановити зв'язок у відповідності із заданим URL бази даних, вибравши відповідний драйвер з-поміж зареєстрованих JDBC драйверів.

Параметри:

url – URL бази даних у формі *jdbc:subprotocol:subname ;*

info – список довільних пар типу string дескриптор/значення у якості аргументів з'єднання; зазвичай повинні бути подані принаймі властивості "user" та "password".

Як результат повертається *Connection* до відповідного URL.

Винятки: *SQLException* – у випадкові виникнення помилки доступу до бази даних.

Interface Connection – з'єднання (сесія) зі вказаною базою даних. Виконання інструкцій SQL та отримання результату здійснюють за допомогою Connection. Детальніше тут: http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html .

Statement createStatement() throws SQLException

Створює об'єкт Statement для надсилання SQL-виразів до бази даних. SQL вирази без параметрів зазвичай виконують за допомогою об'єктів *Statement*. Якщо SQL-вираз виконують багато разів, доцільно скористатися об'єктом *PreparedStatement*.

Набір результатів *ResultSet*, створений за допомогою об'єкта Statement, має тип TYPE_FORWARD_ONLY з рівнем паралелізму CONCUR_READ_ONLY.

Повертає: новий об'єкт *Statement*.

Винятки: *SQLException* - у випадкові виникнення помилки доступу до бази даних або якщо метод застосовано над закритим з'єднаням.

ResultSet executeQuery(String sql) throws SQLException

http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html#executeQuery%28java.lang.String%29

Виконує даний SQL вираз та повертає об'єкт *ResultSet*. Метод не може бути використано над *PreparedStatement* та *CallableStatement*.

Параметри: *sql* - SQL-вираз.

Повертає: об'єкт *ResultSet,* що містить дані, які відповідають поточному запитові; не може бути *null.*

Винятки:

SQLException - у випадкові виникнення помилки доступу до бази даних, метод застосовано над закритим *Statement*, результатом даного SQL вираз є щось інше, ніж об'єкт *ResultSet*, метод викликано над *PreparedStatement* або *CallableStatement*.

SQLTimeoutException – якщо драйвер визначив, що таймаут, встановлений за допомогою методу *setQueryTimeout,* перевищено, і спробував скасувати поточний *Statement*.

6 Основні методи класу ResultSet

http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html

```
public boolean absolute(int row) throws SQLException;
```

переміщує курсор на задане число рядків від початку, якщо параметр row — додатній, і від кінця — якщо від'ємний.

```
public void afterLast() throws SQLException;
```

переміщує курсор у кінець результуючого набору, за останній рядок.

```
public void beforeFirst() throws SQLException;
```

переміщує курсор на початок результуючого набору, перед першим рядком.

```
public void deleteRow() throws SQLException;
```

видаляє поточний рядок із результуючого набору і бази даних.

```
public ResultSetMetaData getMetaData() throws SQLException;
```

надає об'єкт метаданих для даного ResultSet. Клас ResultSetMetaData містить інформацію про результуючі таблиці, таку як кількість стовпців, їх заголовки тощо.

```
public int getRow() throws SQLException;
```

повертає номер поточного рядка.

```
public Statement getStatement() throws SQLException;
```

повертає экземпляр Statement, котрий створив даний результуючий набір.

```
public boolean next() throws SQLException;
public boolean previous() throws SQLException;
```

дозволяють переміститися у результуючому наборі на один рядок вперед або назад. У результуючому наборі курсор встановлюється перед першим рядком, тому перше звернення до методу next() викликає позиціонування на перший рядок. Методи повертають true, якщо залишається рядок для подальшого переміщення. Якщо рядків для обробки більше немає, повертається false. Якщо відкритий потік InputStream для попереднього рядка, він закривається. Також очищається ланцюжок попереджень SQLWarning.

```
public void close() throws SQLException
```

здійснює негайне закриття ResultSet вручну. Зазвичай цього не потрібно, так як закриття Statement, пов'язаного з ResultSet, автоматично закриває ResultSet. Не всі розробники JDBC-драйверів дотримуються цих конвенцій, наприклад, драйвер Oracle самостійно не закриває ResultSet'и.

7 Основні методи класу Connection

http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html

```
public void close() throws SQLException;
```

дозволяє уручну звільнити ресурси такі, як мережеві з'єднання і блокування бази даних, пов'язані з даним об'єктом Connection (метод автоматично викликається при збірці сміття).

```
public Statement createStatement() throws SQLException;
public Statement createStatement(int type, int concur) throws SQLException;
```

створює об'єкт Statement, пов'язаний з сеансом Connection. Версія без аргументів створює об'єкт Statement, для якого екземпляри ResultSet мають тип тільки для читання і переміщення в прямому напрямку.

```
public boolean getAutoCommit() throws SQLException;
public void setAutoCommit(boolean ac) throws SQLException;
```

за замовчуванням об'єкти *Connection* знаходяться в режимі автозавершення. У цьому режимі кожна команда завершується одразу після виконання. Може виявитися кращим вручну завершити серію команд як єдину транзакцію. У цьому випадку метод *setAutoCommit()* використовують для відключення автозавершення. Потім, після виконання команд, викликають *commit()* або *rollback()*, залежно від успіху чи неуспіху транзакції. У режимі автозавершення команда завершується, коли вона виконана, або виконується наступна команда, в залежності від того, що відбудеться раніше. Команда, яка повертає *ResultSet*, виконана після екстракції останнього рядка або закриття об'єкта *ResultSet*. Якщо команда повертає множинні результуючі набори, завершення відбувається після екстракції останнього рядка останнього об'єкта ResultSet.

```
public void commit() throws SQLException;
```

метод робить постійними зміни, створені усіма командами, пов'язаними з даним з'єднанням і виконаними після останньої команди завершення або відкату транзакції. Використовувати його слід тільки при відключеному автозавершенняі. Він не завершує зміни, зроблені командами, які пов'язані з іншими об'єктами *Connection*.

```
public String getCatalog() throws SQLException;
public void setCatalog(String catalog) throws SQLException;
```

Якщо драйвер підтримує каталоги, то *setCatalog()* використовують для вибору підпростору бази даних із заданим ім'ям каталогу. Якщо драйвер каталоги не підтримує, запит ігнорується.

```
public DatabaseMetaData getMetaData() throws SQLException;
```

клас *DatabaseMetaData* надає методи, що описують таблиці бази даних, підтримку SQL, stored-процедури та інші відомості, що стосуються бази даних і даного Connection, які не стосуються безпосередньо виконання команд і отримання результуючих наборів даних. Метод створює екземпляр класу *DatabaseMetaData* для даного *Connection*.

```
public SQLWarning getWarnings() throws SQLException;
```

Повертає перше попередження зі списку, пов'язаного з даним об'єктом *Connection*.

7 Основні методи класу Statement

http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html

```
public void addBatch(String sql) throws SQLException;
```

додає задану команду SQL до поточного пакету команд.

```
public void cancel() throws SQLException;
```

у багатопоточному середовищі за допомогою цього методу можна зажадати припинення обробки, пов'язаної з даним Statement. У цьому сенсі метод аналогічний до методу *stop()* для об'єктів Thread.

```
public boolean execute(String sql) throws SQLException;
public ResultSet executeQuery(String sql) throws SQLException;
public int executeUpdate(String sql) throws SQLException;
```

виконує Statement, передаючи базі даних заданий SQL-рядок. Перший метод, *execute()* дозволяє виконати *Statement*, коли не відомо заздалегідь, є SQL-рядок запитом чи оновленням. Метод повертає *true*, якщо команда створила результуючий набір. Метод *executeQuery()* використовують для виконання запитів (на отримання даних). Він повертає для обробки результуючий набір. Метод *executeUpdate()* використовують для виконання оновлень. Він повертає кількість оновлених рядків.

```
public int[ ] executeBatch(String sql) throws SQLException;
```

надсилає базі даних пакет SQL-команд для виконання. Повертає масив чисел, що описують кількість рядків, опрацбованих кожною командою SQL.

```
public ResultSet getResultSet() throws SQLException;
```

повертає поточний *ResultSet*. Для кожного результату його слід викликати тільки один раз. Метод не потрібно викликати після звернення до *executeQuery()*, що повертає єдиний результат.

```
public void close() throws SQLException;
```

вручну закриває об'єкт Statement. Зазвичай цього не потрібно, оскільки *Statement* автоматично закривається при закритті пов'язаного з ним об'єкта *Connection*.