Тема 2. Основи синтаксису Java

Лекція 3

Особливості синтаксису Java

- Синтаксис Java багато в чому схожий з синтаксисом C# та C++. Як ми побачимо далі, набір ключових слів і операторів цих двох мов мають багато спільного.
- Малі та великі літери – розрізняються (наприклад someName і SomeName – різні імена). Існує угода називати класи з великої літери (SomeClass), змінні та методи – з маленької (someVar, someMethod()), а константи – усіма великими літерами (SOME_CONST).
- Кожен оператор завершується крапкою з комою.
- Оператори можуть перебувати тільки всередині блоків. Блоками є тіло методу, тіло складного оператора (умовного оператора, циклу, секції try, синхронізованої секції), блок ініціалізації класу (про який піде мова в наступних лекціях). Також будь-яка послідовність операторів може бути згрупована в окремий блок. Усюди, де за правилами мови може перебувати одиничний оператор, замість нього може використовуватися блок. У всіх випадках блок виділяється фігурними дужками, наприклад:

```
if (condition) {        // Блок – тіло складного оператора
     operators;
     {                  // Блок – угрупування операторов
          Operators;
     }
}
```

- Ознакою оголошення або виклику методу є круглі дужки після імені:

```
void someMethod() {          // Оголошення методу
     operators;
}
// ...
void someMethod() {          // Оголошення методу
         someMethod();       // Виклик методу
}
```

- Java – повністю *об'єктно-орієнтована* мова. Тобто будь-яка програма будується з класів і об'єктів цих класів, процедурне програмування не підтримується.
- Мова Java *строго типізована*. Це значить, що тип змінної завжди вказується при її оголошенні, а перетворення типів обмежене жорсткими правилами. Наприклад, в Java неможливо перетворити логічне значення в ціле і назад.

  Для порівняння, в мові C++ тип змінної також потрібно вказувати при її оголошенні, але за допомогою покажчиків можна безпосередньо звернутися до області пам'яті змінної і працювати зі збереженим там значенням як з масивом байтів незалежно від типу змінної. В Java, навпаки, JVM повністю приховує деталі внутрішньої організації типу даних або класу і дозволяє виконувати тільки ті операції, які передбачені цим типом даних або класом. Механізм покажчиків в Java відсутній, тобто неможливо безпосередньо звернутися до довільної комірки пам'яті. Замість цього використовуються ***посилання***, які, подібно до покажчиків, вказують на деяку комірку пам'яті, але, в той же час, строго типізовані, як і змінні. Як ми побачимо далі, поняття «посилання» і «змінна» в Java дуже близькі і часто розглядаються як синоніми.
- Java підтримує автоматичне *керування пам'яттю*. Це означає, що програмісту не потрібно піклуватися про звільнення виділеної пам'яті. Всі об'єкти автоматично створюються в динамічній області пам'яті. JVM містить спеціальний компонент – ***збирач сміття,*** – який знаходить невикористовувані об'єкти і видаляє їх з пам'яті.

- Для створення локалізованих версій додатків дуже важлива підтримка універсальних багатобайтових кодів символів (*Unicode*). У таблиці Unicode кожен символ кожної мови представлений унікальним кодом, тому, на відміну від інших кодувань, не виникає проблем з коректним відображенням символів. У той час, як у багатьох мовах для перекодування тексту в Unicode необхідно використовувати спеціальні функції, в Java підтримка Unicode вбудована. Це означає, що прямо в коді програми можуть використовуватися будь-які Unicode-символи на будь-якій мові, і ці символи завжди коректно відображаються.
- Java має багатий набір стандартних класів. Так, є ціла вбудована бібліотека класів-колекцій, які реалізують різноманітні структури даних: вектор, список, стек, хеш-таблицю. При цьому багато структур даних мають кілька різних реалізацій з різними властивостями. Наприклад, структура даних «список» реалізована як на основі динамічного масиву (клас ArrayList), так і на основі зв'язного списку (клас LinkedList).
- Java має потужні засоби (бібліотеки, фреймворки, допоміжні технології), що полегшують створення web-додатків (в тому числі з використанням протоколу RMI), web-сервісів і розподілених компонентів. Найбільш простим прикладом є *сервлети* – класи, що дозволяють обробляти і виконувати HTTP-запити.
- До мови вбудовані засоби створення багатопотокових додатків – синхронізація потоків і управління ними. Сюди відносяться секції синхронізації, підтримка моніторів об'єктів, з якими познайомимося далі в цьому курсі.

Синтаксис мови Java активно розвивається. З кожною новою версією з'являються нові можливості для прискорення процесу створення програмних продуктів. Наприклад, починаючи з версії JDK 1.5, до складу мови включені ***анотації*** – спеціальні короткі вставки, зазвичай декларативного характеру, які часто здатні замінити великі фрагменти звичайного коду. У наступних версіях в ядро мови додані можливості параметричної типізації (***generics***), *лямбда вирази* і безліч інших засобів, що дозволяють легко писати більш ефективний і читабельний код.

Оголошення

Всі змінні, методи і класи, використовувані в програмі, повинні бути заздалегідь ***оголошені***. Розглянемо синтаксис оголошень. У дужках [] показані необов'язкові елементи оголошення, а жирним шрифтом виділені обов'язкові ключові слова.

1. Змінні:
```
[Спосіб_доступу] [модифікатори] Тип ім'яЗмінної
                                    [= початковеЗначення];
```
2. Константи:
```
[Спосіб_доступу] static  final  Тип ім'яЗмінної
                                    [= ПочатковеЗначення];
```
3. Методи (функції)
```
[Спосіб_доступу] [модифікатори]
ТипЗначенняЩоПовертається   ім'яМетода (
                        ТипПараметра ім'яПараметра, ...) {
   Оператори;
   return ЗначенняЩоПовертається;
}
```
4. Клас:
```
[Спосіб_доступу] [модифікатори] class им'яКласу {
      Тіло класу (оголошення змінних і методів)
}
```

Тут *Спосіб_доступу* – це один з модифікаторів `private`, `protected`, `public`. Тонкощі використання способів доступу ми розглянемо в наступних лекціях. На початковому етапі спосіб доступу можна просто не вказувати.

До *модифікаторів* змінних відносяться ключові слова `final`, `static`, `volatile`, `transient`. Зміст цих модифікаторів будемо розглядати далі.

Як приклад наведемо оголошення невеликого класу, що містить оголошення двох методів sum() і main(). У методі main() також оголошуються локальні змінні, необхідні для його роботи: i, j, res. В цілому, синтаксис оголошень в java схожий з іншими близькими мовами – C# і C++.

```
public class Summ {              // Оголошення класу додатку

  int sum(int a, int b) {        // Метод, повертаючий ціле число
    return a+b;
  }

  public static void main(String[] args) {        // Головний метод
    int i = 1, j = 2;                             // Дві цілі змінні
    double res = sum(i,j);                        // Дійсна змінна
    System.out.println(res);
  }
}
```

Ключові слова (key words)

Нижче наведено список ключових слів Java. Всі ці слова можна використовувати як імена. Жирним виділені ті з них, які є новими в порівнянні з мовою C++. Курсивом виділено ключові слова, зміст яких істотно змінений у порівнянні з C++. Нарешті, сірим виділені зарезервовані слова, які на даний момент не використовуються.

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

Літерали (константи)

Деякі імена зарезервовані за ***літералами***. Це не ключові слава, але вони також не можуть бути використані в якості ідентифікаторів (імен) у вашій програмі:

```
true;
false;
null.
```

Правила запису **арифметичних констант** в цілому аналогічні мові C++. Числа можна записувати в десятковій системі:

- 10 – звичайна ціла константа (типу int);
- 1.23 – звичайна дійсна константа (типу double);
- 1e5 – 100000 в експоненційної форе;
- 1.23e-3 – 0.00123 в експоненційної формі;

в вісімковій:

- 010 – це число 8;
- 0123 – це 83 (1 * 64 + 2 * 8 + 3);

або в шістнадцятковій:

- 0x10 – це 16
- 0x123 – це 291 (1 * 256 + 2 * 16 +3).

Починаючи з Java SE 7 константи цілочислових типів byte, short, int, long можна записувати в двійковій системі, наприклад:

```
byte aByte = (byte)0b00100001;
short aShort = (short)0b1010000101000101;
int anInt2 = 0b101;
```

Такий запис дуже зручний при перевірці бітових «прапорців». Наприклад, щоб перевірити, 3-й і 8-й біти змінної flag на рівність нулю, достатньо написати:

```
if ((flag & 0b100001000) == 0b0) { ... }
```

Для вказівки типу константи застосовуються суфікси:

l (або L) – long,
f (або F) – float,
d (або D) – double.

Наприклад, 1L – одиниця, але типу long, а не int.

Строкові літерали записуються в подвійних лапках, наприклад "`це рядок`".

Символьні літерали записуються в апострофах, наприклад `'F'`, `'ш'`.

В строкових і символьних літералах є правила для запису спеціальних символів. По-перше, є набір визначених спеціальних символів:

– '\ n' – кінець рядка (переведення рядка)
– '\ r' – повернення каретки
– '\ t' – табуляція

і ряд інших.

По-друге, можна явно записати восьмеричний код символу, наприклад: '\ 001' – це символ з кодом 1.

Керуючі конструкції

1. Умовний оператор (if)

Синтаксис:

```
if ( <умова> )
       <оператор1>
[else
       <оператор2>]
```

Тут `<умова>` — це логічний вираз типу boolean, тобто вираз, що повертає true або false. Вирази інших типів, навіть цілочислових, не можуть бути перетворені в boolean, тому перевірка на нуль завжди вимагає явного порівняння:

```
if (a == 0) {...}
```

Як видно з синтаксису, частина else є необов'язковою. Після if і після else стоїть по одному оператору. Якщо потрібно помістити туди кілька операторів, то потрібно поставити **блок** { }. В Java прийнято блок ставити завжди, навіть якщо після if або else стоїть один оператор.

Приклади:

```
if ( a > b ) {
    x = a;
```

```
} else {
    x = b;
}

if ( flag ) {
    flag = false;
    init();
}
```

В останньому прикладі flag – логічна змінна або поле, init() – метод, що викликається, якщо флаг дорівнює true (кажуть, "якщо flag встановлений").

2. Багатоваріантний вибір (switch)

Служить для організації вибору по деякому значенню однієї з кількох гілок виконання.
Синтаксис:

```
switch ( <вираз> ) {
       case <константа1>:
            <оператори1>
       case <константа2>:
            <оператори2>
       ...
       [default:
            <оператори_D>]
}
```

Зауважимо, що <вираз> має видавати ціле число або символьне значення; константи повинні бути того ж типу, що і значення цього виразу.

Елементи `case <константа>:` є мітками переходу. Якщо значення виразу збігається з константою, то буде здійснено перехід на цю мітку. Якщо значення виразу не збігається ні з однією з констант, то все залежить від наявності фрагмента **default**. Якщо він є, то перехід відбувається на мітку `default`, якщо його немає, то весь оператор switch пропускається.

В операторі `switch` фрагменти `case` не є блоками. Якщо після останнього оператора даного case-фрагмента стоїть наступний case, то виконання буде продовжено, починаючи з першого оператора цього case-фрагмента. Тому в операторі switch зазвичай застосовується оператор **break**, який ставиться в кінці кожного case-фрагмента.

3. Оператори циклу

3.1. Цикл з передумовою (while)

Синтаксис:

```
while ( <умова> )
        <оператор>
```

Як і у випадку оператора if, в Java прийнято <оператор> заключати в фігурні дужки.
Приклад:

```
int x = 123, i=0;
while ( x>0 ) {
    x /= 2;
    i++;
}
```

3.2. Цикл с постумовою (do while)

Синтаксис:

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

Відрізняється від попереднього тільки тим, що умова кожен раз перевіряється не перед, а після виконання тіла циклу. Тому в **do while()** тіло циклу завжди виконується як мінімум один раз, в той час як **while()** може не виконатися жодного разу, якщо умова є помилковою спочатку.

Приклад:

```
int x = 123, i=0;
do {
    x /= 2;
    i++;
} while ( x>0 );
```

Даний приклад еквівалентний попередньому.

3.3. Цикл for()

Синтаксис:

```
for ( <ініціалізація>; <умова>; <зміна> )
     <оператор>
```

Вираз <ініціалізація> виконується один раз перед першим витком циклу. Перед кожним витком перевіряється <умова>, і після кожного витка циклу виконується вираз <зміна>. Як і в if() вираз «умова» може бути тільки логічним (типу boolean).

Для зручності складання циклів з лічильником в даній конструкції можливо кілька варіантів:

- <ініціалізація> може бути не виразом, а описом змінної з ініціалізацією, наприклад `int i = 0`. Така змінна є *локальною* для циклу і не доступна після його завершення.
- <ініціалізація> може бути списком виразів через кому, наприклад `i = 0, r = 1`.
- <зміна> також може бути списком виразів, наприклад `i++, r *= 2`. Ці два випадки використання оператора «*кома*» є винятком. Ніде, крім заголовка циклу for цей оператор не використовується.
- Всі складові (<ініціалізація>, <умова> і <зміна>) є необов'язковими. Якщо відсутня умова, то вона вважається завжди істинною, і вихід з циклу повинен бути організований якимись засобами всередині самого циклу.

Приклад:

```
int x = 123;
for (int i = 0; x>0; i++) {
x /=2;
}
```

3.4. Цикл for each

Синтаксис:

```
for (<оголошення змінної> : <масив або колекція> )
     <оператор>
```

Це дуже зручна модифікація циклу for, яка дозволяє пройти по всіх елементах масиву або колекції без використання лічильника. Рекомендується всюди, де це можливо, використовувати саме таку форму циклу. Усередині цієї форми for обов'язково оголошується нова локальна змінна, тип якої повинен бути сумісний з типом елементів масиву (колекції). Наприклад:

```
int sum =0, a[] = {5,6,7,8};
```

```
for(int i : a)  sum+=i;
```

Тут перед циклом оголошується масив `a[]` з чотирьох цілих чисел. Всередині циклу оголошується змінна `i` – теж ціла. На кожному витку виконання циклу в змінну `i` заноситься (копіюється) значення чергового елемента з масиву, і для неї виконується тіло циклу. Цикл гарантовано обробить всі елементи масиву і після цього завершиться. Зверніть увагу, що в даному випадку через змінну `i` неможливо змінити вміст масиву. Зовсім інша ситуація буде, якщо в масиві зберігаються не числа, а об'єкти – до цього питання повернемося в наступній лекції.

4. Оператори переходу: break, continue, return.

Використання цих операторів повністю аналогічно мові C++.

break – завершує виконання циклу або оператора switch();

continue – перехід на наступний виток циклу;

return – завершення поточного методу з можливістю повернути значення в метод, що його викликав.

Як приклад спільного використання операторів розгалуження і циклів розглянемо програму, яка генерує випадковим чином 100 символів латинського алфавіту і класифікує їх як "голосні", "приголосні" і "іноді голосні". В останню категорію віднесені символи 'y' і 'w'.

```
public class SymbolTest {

    public static void main(String[] args) {
        for ( int i = 0; i < 100; i++ ) {
            char c = (char)(Math.random()*26 + 'a');
            System.out.print(c + ": ");
            switch ( c ) {
            case 'a': case 'e': case 'i':
            case 'o': case 'u':
                System.out.println(" - голосна");
                break;
            case 'y': case 'w':
                System.out.println(" – інколи голосна");
                break;
            default:
                System.out.println(" - приголосна");
            }
        }
    }
}
```

В даному прикладі є кілька нових для нас елементів.

- Використовується метод random() класу Math. Подивимося документацію по класу Math і розберемося, що він робить.
- В операторі
  ```
  char c = (char) (Math.random() * 26 + 'a');
  ```
  проводиться складання арифметичного значення з символом. При такому складанні в Java символ перетворюється в число, яке дорівнює коду цього символу.
- В операторі
  ```
  System.out.print(c + ":");
  ```
  символ с приводиться до строкового типу для з'єднання з рядком.

Слід також звернути увагу на фрагменти case. Формально тут 7 таких фрагментів, але 5 з них не містять ніяких операторів. Оскільки в них немає операторів break, фактично для кількох символів виконуються однакові дії.

Введення-виведення

У цій лекції розглянемо засоби *консольного введення-виведення*. Самі по собі консольні додатки використовуються нечасто, але вивчені класи і методи дозволять в майбутньому аналогічно виконувати введення-виведення з текстовими файлами і іншими текстовими потоками, наприклад, з мережевими з'єднаннями.

Форматоване виведення за допомогою об'єкта `System.out`

Крім методів print() і println(), з якими ми вже знайомі, об'єкт System.out надає методи format() і printf() для форматованого виведення. Можливості цих методів однакові і збігаються з можливостями функції printf() мови C.

Формат виведення вказується за допомогою першого параметру – рядка, що містить *специфікатори формату*. Для цілих чисел використовуються специфікатори %d, %x; для дійсних – %f, %e, %g (незалежно від типу – float або double); для рядків – %s. Після символу % можна вказувати ширину поля виведення і кількість знаків після коми для речових значень. Наприклад:

```
double d = Math.PI;
System.out.printf("%f; %10.2f;\n %-10.0ff\n", d, d, d);
```

виведе

```
3,141593;       3,14;
 3        f
```

Якщо тип переданого значення не відповідає типу специфікатора, то генерується *виключення*. Якщо таке виключення не оброблено явно, то це приводить до аварійного завершення програми.

Для форматованого виведення в рядок можна використовувати статичний метод `String.format()`, який має аналогічні параметри. Він створює об'єкт класу String, і в нього записує відформатований текст, наприклад:

```
System.out.println( String.format("%s = %f", "Pi", Math.PI) );
```

Клас `java.util.Scanner`

В Java існує кілька способів реалізації форматованого введення. Найбільш простий з них – використовувати клас `java.util.Scanner`. Цей клас знаходиться в пакеті java.util, тому його необхідно підключити на початку програми:

```
import java.util.Scanner;  // Підключаємо клас на початку програми
```

При створенні об'єкт класа Scanner зв'язується з потоком введення, файлом чи рядком:

```
Scanner in = new Scanner(System.in);          // Стандартний потік введення
Scanner in1 = new Scanner( "123 + 125" );           // Рядок
Scanner in2 = new Scanner( new File("a.txt") );   // Файл в директорії
                                                    // проекту
```

У разі читання з файлу додатково необхідно обробляти можливі *виключення*. У всіх випадках вхідні дані розглядаються як послідовність *слів*, між якими стоять *роздільники*. За замовчанням роздільниками є пробіл, табуляція і новий рядок. Для читання даних за словами в класі Scanner є методи виду **next*Type*(),** де *Type* – один з примітивних типів:

```
String s = in.next();   // зчитати наступне слово у вигляді рядка
int a = in.nextInt();   // зчитуємо ціле число
byte b = in.nextByte(); // зчитуємо байтове число
...
double d = in.nextDouble();  //зчитуємо дійсне число
```

Читання до кінця рядка виконує метод **nextLine()**.

Приклад 1. Зчитуємо вирази виду "число операція число"

```
Scanner in = new Scanner(System.in);
int i1 = in.nextInt();          // зчитуємо ціле число a
char c = in.next().charAt(0);   // зчитуємо  знак операції
int i2 = in.nextInt();          // зчитуємо  ціле число b
String c = in.nextLine();     // пропускаємо все до кінця рядка
```

Методи **next*Type*()** мають такі особливості:

- вони блокують поточний потік виконання програми, поки в потоці введення не з'явиться наступне слово (або кінець рядка);
- всі вони, крім next() і nextLine() генерують виключення, якщо формат слова не відповідає очікуваному типу.

Тому для кожного з методів next*Type*() є метод **hasNext*Type*(),** який перевіряє, чи є в потоці слово потрібного формату, але нічого не вводить і не змінює поточну позицію в потоці. Також є методи **hasNext()** і **hasNextLine()**, які перевіряють наявність в потоці відповідно слова або кінця рядка, без урахування формату.

Метод **useDelimiter(String pattern)** дозволяє задати роздільник слів за допомогою ***регулярного виразу (Regex)***. У Java регулярні вирази використовуються дуже часто і допомагають розпізнавати рядки і виділяти значущі частини рядків. *Регулярний вираз – це рядок спеціального виду, шаблон, який описує формат тексту.* Схожі функції виконує рядок формату в методі printf(), але регулярні вирази – це набагато більш гнучкий і потужний засіб. Типовими задачами, які розв'язуються за допомогою регулярних виразів є:

- перевірка правильності введення, наприклад, перевірка правильності написання Email або номера телефону;
- пошук фрагментів тексту певного виду, наприклад, пошук всіх тегів-посилань в html-документі;
- контекстна заміна, наприклад в тексті замінити всі коми, що стоять перед великою літерою, на крапки.

Найпростішим регулярним виразом є звичайний рядок без всяких спеціальних символів, наприклад пробіл " ". За допомогою такого виразу можна знайти тільки точні співпадіння в тексті. Якщо ж потрібно знайти, наприклад, всі пробільні символи (пробіли, переклади рядків, табуляції), то потрібно використовувати спеціальний символ "\s".

Приклад 2. Зчитування послідовності чисел, розділених пробілами, комами, або крапкою з комою.

```
Scanner in = new Scanner("123, 123 345 ;   33");
in.useDelimiter("[\\s,;]+");    // Набір символів-роздільників
                                // задається регулярним виразом
while( in.hasNextDouble() ) System.out.println( in.nextDouble() );
if (in.hasNextLine()) System.out.println( in.nextLine() );
          // Останнє слово, после якого може не быть роздільника
```

Примітка. Тут регулярний вираз `[\\s,;]` означає будь-який із символів «пробіл», «кома», «крапка з комою». Перед "\s" ставиться повторний слеш, щоб компілятор не намагався замінити ці два символи на один подібно '\n'. Символ '+' після дужок означає, що роздільник може складатися з одного або декількох символів. Якщо '+' не поставити, то два пробіли, що йдуть один за другим, розпізнаються як два окремих роздільники, між якими знаходиться порожнє слово, і це порожнє слово буде з'являтися на виході методу next().

Більш докладно регулярні вирази розглянемо далі.

Приклад 3*. Виведення вмісту файлу на екран.

```
import java.io.File;        // Підключаємо клас для роботи з файлами
// ...
try {     // Блок, в якому можливі виключення
     // Зв'язуємо Scanner з файлом
     // Тут можливі виключення, наприклад, якщо такого файлу немає
     Scanner in = new Scanner( new File("1.txt") );
     while (in.hasNextLine())                    // Поки є рядки в файлі
          System.out.println( in.nextLine() );   // Виводимо їх
} catch (Exception e) {          // Обробник виключення
     e.printStackTrace();        // Вивести інформацію про виключення
}
```

Методи

boolean hasNext(String pattern);

String findInLine(String pattern);

призначені для пошуку в потоці введення відповідно слова або під-рядка, що відповідає заданому формату. Формат задається у вигляді регулярного виразу. Для доступу до результатів порівняння використовується метод **Scanner.match()**.

Приклад 4. Нехай у файлі urls.txt записані посилання URL і коментарі до них, наприклад:

```
My references:
Reference 1:   ftp://file_server.com:21/top_secret/life_plans.pdf
https://regexone.com/lesson/introduction#section     - reference 2
file://localhost:4040/zip_file
https://s3cur3-server.com:9999/
market://search/angry%20birds
```

Необхідно обробити цей файл і вивести всі домени, на які ведуть посилання, із зазначенням портів, до яких вони звертаються. Наступний короткий фрагмент коду виконує це завдання, використовуючи клас Scanner і регулярні вирази.

```
String pattern = "(\\w+)://([\\w\\-\\.]+)(:(\\d+))?";   // Рядок-шаблон
Scanner in = new Scanner(new File("urls.txt"));         // Відкрити файл
while (in.hasNextLine()) {                          // Пока файл не закінчився
    if (in.findInLine(pattern) != null) {           // Перевірити, чи є в рядку URL
        String domain = in.match().group(2);        // Отримати частини URL
        String port = in.match().group(4);
        if (in.match().group(4) == null) {          // Перевірити наявність порта
            port = "default";
        }
        System.out.format("Domain: %s, port: %s\n", domain, port);
    }
    in.nextLine();                                  // Пропустити весь залишок рядка
}
```

Для наведеного файлу результатом роботи програми буде наступний текст:

```
Domain: file_server.com, port: 21
Domain: regexone.com, port: default
Domain: localhost, port: 4040
Domain: s3cur3-server.com, port: 9999
Domain: search, port: default
```

В цьому прикладі використано відразу кілька можливостей регулярних виразів, детальніше про які можна дізнатися з документації Regex.