Любченко Н.Ю., Подорожняк А.О., Черних О.П.

ОСНОВИ МОВИ ПРОГРАМУВАННЯ SCALA

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

Н.Ю. Любченко, А.О. Подорожняк, О.П. Черних

ОСНОВИ МОВИ ПРОГРАМУВАННЯ SCALA

**Навчально-методичний посібник
для студентів комп’ютерних спеціальностей
вищих навчальних закладів**

Затверджено редакційно-
видавничою радою НТУ "ХПІ",
протокол № 3 від 12.10.2023

Харків
НТУ “ХПІ”
2023

УДК 004.43
Л93

Рецензенти:

А.А. Коваленко, докт. техн. наук, професор, Харківський Національний університет радіоелектроніки

В.В. Усик, канд. техн. наук, професор, Національний технічний університет "Харківський політехнічний інститут"

Любченко Н.Ю.
Л93 Основи мови програмування SCALA: навч.-метод. посібник / Любченко Н.Ю., А.О. Подорожняк Черних О.П. – Харків : НТУ "ХПІ", 2023. – 148 с.

Навчально-методичний посібник складається з восьми практичних робіт, які охоплюють першу частину курсу «Основи розподілених та паралельних обчислень», присвячені вивченню мови та компілятора Scala. Передбачається, що студент має певний досвід у програмуванні, має базові знання в області об'єктно-орієнтованого програмування і хоче отримати уявлення про те, що він може робити за допомогою Scala. Приклади представлені у вигляді програмного коду.

Для студентів комп'ютерних спеціальностей вищих навчальних закладів.

Іл. 62. Бібліогр. 5 назв.

УДК 004.43

ВСТУП

Навчально-методичний посібник «Основи мови програмування SCALA» – це частина навчального курсу «Паралельні та хмарні обчислювальні системи» призначена для знайомства з методами об’єктно-орієнтованого програмування. У посібнику коротко розглядаються найбільш важливі властивості мови SCALA, у якої використовується чиста об'єктно-орієнтована модель. Мова надає легковаговий синтаксис для визначення функцій. Scala представляє нову концепцію вирішення проблеми зовнішньої розширюваності – види (views). Види в Scala переводять в об'єктно-орієнтоване уявлення використовувані в Haskell класи типів. На відміну від класів типів, область видимості видів можна контролювати, причому в різних частинах програми можуть співіснувати паралельні види.

Навчально-методичний посібник містить 8 практичних робіт. Кожна з практичних робіт складається з необхідного для її виконання короткого теоретичного матеріалу та прикладів його використання в роботах. Наведені приклади є не фрагментами коду, а у більшості – готовими програмами.

Виконання кожної роботи передбачає попереднє вивчення відповідних лекційних тем. Практичні роботи виконуються відповідно до індивідуальних завдань і за ними оформляються звіти відповідно до вимог, що стосуються оформлення.

Практична робота 1

ОСНОВИ МОВИ SCALA

Мета: Оволодіння практичними навичками роботи з основними операторами мови Scala

Теми для попереднього опрацьовування

- Змінні
- Умовні оператори та цикли
- Функції

Теоретичні відомості

Scala – мультіпарадігмальна мова програмування, спроектована коротким і тіпобезпечним для простого і швидкого створення компонентного програмного забезпечення, поєднує можливості функціонального і об'єктно-орієнтованого програмування.

Scala – це статично типізована, об'єктно-орієнтована мова програмування, яка поєднує в собі імперативний та функціональний стилі програмування. Scala призначена для легкої інтеграції з програмами, які працюють на сучасних віртуальних машинах, в першу чергу на віртуальній машині Java (JVM).

Scala була розроблена в 2003 році групою Мартіна Одерскі в EPFL (*Федеральна політехнічна школа Лозанни або Swiss Federal Institute of Technology of Lausanne – державний вищий навчальний заклад у Швейцарії*). Раніше Мартін активно працював у галузі Java. Він був одним із розробників першої версії універсальних шаблонів Java та був оригінальним автором поточного компілятора *javac*. Робота над Scala була мотивована бажанням подолати обмеження, що накладаються зворотною сумісністю з Java.

Основні характеристики

- *Об'єктно-орієнтований підхід* – мова реалізує принципи ООП.
- *Функціональний* – мова розроблена з підтримкою функціонального підходу.

• *Інтегрована з Java* – може взаємодіяти з кодом написаним на Java; в Scala коді можна викликати будь-який об'єкт Java.

Інтерпретатори Scala

Scala REPL (*Read-Evaluate-Print-Loop – Читання-оцінка-друк-цикл*) – це інтерпретатор командного рядка, який використовується як "ігровий майданчик" для тестування коду Scala. REPL читає вирази, які йому передали, виконує їх, використовуючи компілятор Scala, друкує результат виконання і чекає наступного запиту. REPL дуже зручний засіб, що дозволяє на льоту перевірити правильність різних Scala-конструкцій і виразів

Для установки Scala REPL необхідно завантажити файл scala-2.13.5.msi (або іншу версію) з https://www.scala-lang.org/download/ і встановити Scala IDE. Щоб почати сеанс REPL, потрібно ввести команду `scala` в командному рядку операційної системи (рис. 1.1). Вихід з інтерактивного режиму CTRL+D або командою `:quit`.

```
C:\Windows\System32>scala
Welcome to Scala 2.12.15 (Java HotSpot(TM) 64-Bit Server VM, Java 17).
Type in expressions for evaluation. Or try :help.

scala> 1+1
res0: Int = 2

scala> val x =2
x: Int = 2

scala> :quit

C:\Windows\System32>
```

Рисунок 1.1 – Запуск інтерпретатор Scala

Перевірити версію Scala командою `scala -version` (рис. 1.2).

```
C:\Windows\System32>scala -version
Scala code runner version 2.12.15 -- Copyright 2002-2021, LAMP/EPFL and Lightbend, Inc.
```

Рисунок 1.2 – Перевірка версії Scala

Spark – програмна платформа розподіленої обробки даних, що входить в екосистему Apache Hadoop. Apache Spark (від англ. spark – іскра, спалах) – фреймворк з відкритим кодом для реалізації розподіленої обробки неструктурованих і слабоструктурованих даних, що входить в екосистему проектів Hadoop.

Apache Spark Shell – інтерактивна оболонка, яка надає середовище REPL для виконання команд Spark по одній і перегляду результатів. Цей процес корисний для розробки і відладки. Spark надає одну оболонку для кожної з підтримуваних мов: Scala, Python.

Для установки Spark необхідно завантажити файл spark-3.1.2-bin-hadoop3.2.tgz (або іншу версію) з https://spark.apache.org/downloads.html і встановити Spark Shell. Щоб почати сеанс REPL, потрібно ввести команду `spark-shell` (рис. 1.3). Після запуску опинилися в оболонці Scala, де можна запускати програми. Вихід із інтерактивного режиму CTRL+D або командою `:quit` .

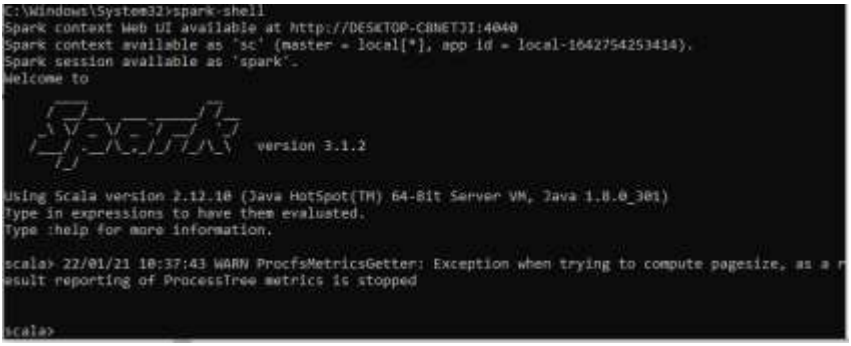


Рисунок 1.3 – Запуск Spark Shell

Запуск Scala у середовищі IntelliJ IDEA

IntelliJ IDEA – інтегроване середовище розробки програмного забезпечення для багатьох мов програмування, зокрема Java, Javascript, Python, Scala, розроблена компанією JetBrains.

Для установки IntelliJ IDEA необхідно завантажити з https://www.jetbrains.com/idea/download/#section=windows файл spark-

ideaIC-2022.1.2.exe (або іншу версію). Першим кроком перед створенням або відкриттям проекту Scala є установка плагіну Scala (рис. 1.4).

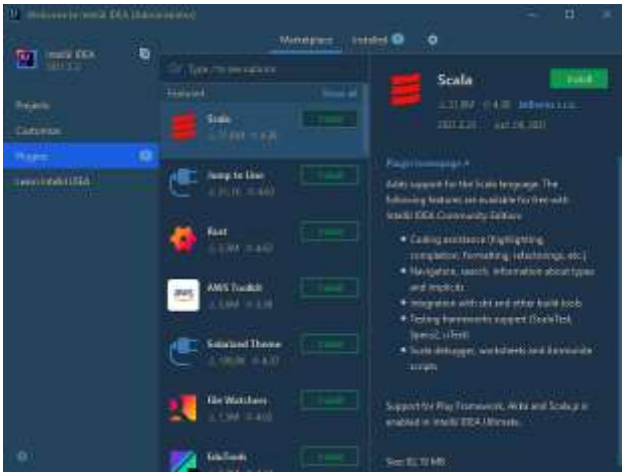


Рисунок 1.4 – Установка плагіну Scala

Інтерактивний режим можна викликати поєднанням Ctrl+Shift+D для виконання інструменту побудови REPL (рис. 1.5). Закрити інтерактивне вікно можна поєднанням Ctrl+Shift+F4.

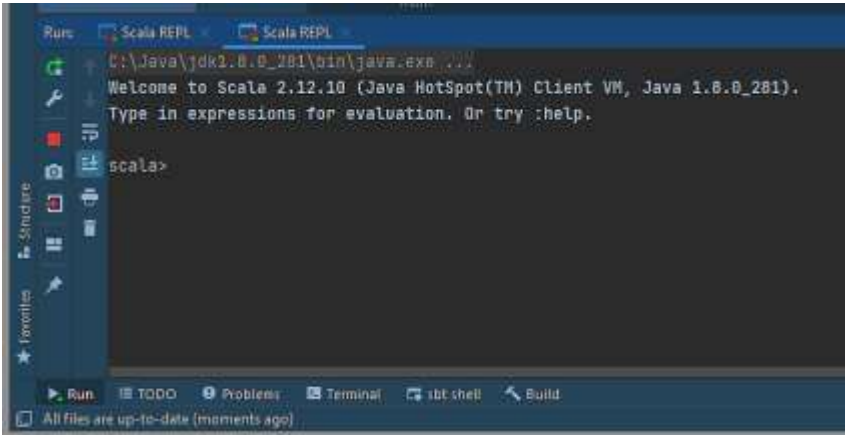


Рисунок 1.5 – Інструмент Scala REPL в IntelliJ IDEA

Основи мови SCALA

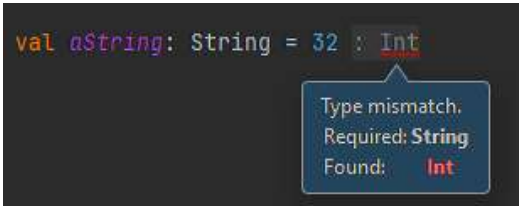
1. Типи, значення, змінні

Синтаксис простий: за ключовим словом `val` слідує ім'я змінної, потім ставиться двокрапка і вказується тип змінної. Після цього друкується знак рівності і конкретизується значення змінної:

```
val aString = "Hello"
```

Щоб вивести значення змінної на екран – використовується функція `println()`.

Необхідно стежити, щоб при оголошенні змінної – вказаний тип і тип значення, що присвоюється, збігалися. Якщо ні – компілятор про це повідомить:



Цілком можна обійтися і без вказівки типу змінній, написавши просто

```
val aString = "Hello"
```

замість

```
val aString : String= "Hello"
```

Це можливо завдяки тому, що компілятор може сам визначити тип змінної.

Типи змінних в Scala:

```
val aString = "Hello"
val aChar = 'C'
val aInt = 11
val aLong = 11L
val aFloat = 2.0f
val aDouble = 2.0
val aBoolean = true
```

Типи Long і Float

```
val aLong = 11
println(aLong.getClass)       // виводить int
```

Хотіли створити змінну типа `Long`, а на ділі – отримали `Int`. Щоб такої ситуації не виникало – для `Long` чисел на кінці дописують заголовну букву `L`

```
val aLong = 11L
println(aLong.getClass)        // виводить long
```

Те ж саме відноситься і до типу `Float` – лише в кінці дописують `f`, щоб компілятор не плутав з `Double`

```
val aFloat = 2.0f
println(aFloat.getClass)      // виводить float
```

Множинна ініціалізація змінних

```
val (x:Int, y:String) = (33, "Scala")
```

x: Int = 33

y: String = Scala

Ієрархія типів Scala

Scala – об'єктно-орієнтована мова, будь-яке значення – об'єкт, будь-яка дія – виклик методу. Типи:

- `Any`
- `AnyRef`
- `AnyVal`
- `Null`
- `Nothing`

Кожному класу відповідає один тип. Спрощена діаграма типів представлена на рис. 1.6). Нагорі ієрархії – тип **Any** – це супер тип для будь-якого іншого типу.

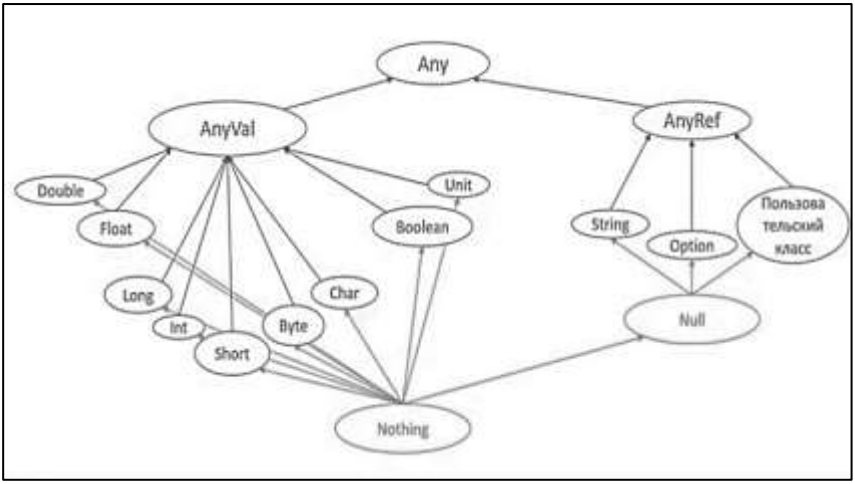


Рисунок 1.6 – Діаграма типів

***Приклад* 1**. Перевірити, чи є різні типи підтипом `Any` (метод `isInstanceOf` дозволяє перевірити тип змінної):

```
scala> 1.isInstanceOf [Any]     // тип Int
res1: Boolean = true
scala> "".isInstanceOf [Any]    // тип String
res2: Boolean = true
scala> class Myclass            // тип class
defined class Myclass
scala> val c = new Myclass
c: Myclass = Myclass@560123dc
scala> c.isInstanceOf [Any]
res3: Boolean = true
```

У невизначеній ситуації можна привласнити змінній тип `Any` і тоді в контексті ця змінна може прийняти будь-який тип, наприклад, `Int`.

```
scala> val x:Any = 12
x: Any = 12
```

```
scala> val s:Any = "Hello"
s: Any = Hello
```

Немає можливості тип `Any` відразу перетворити в якій-небудь інший тип:

```
scala> val z = x.toInt
<console>:12: error: value toInt is not a member of Any
    val z = x.toInt
            ^
```

Аде це можна зробити через `String` за допомогою методу **`toString`**:

```
scala> val y = x.toString
y: String = 12
```

Тепер можна отримати інші типи:

```
scala> val z = y.toInt
z: Int = 12
scala> val z = y.toDouble
z: Double = 12.0
```

Можна робити і так:

```
scala> val z = x.toString.toInt
z: Int = 12
```

Є також метод **`asInstanceOf`**, що застосовується для зміни типів. Він здатний трансформувати навіть тип `Any`:

```
scala> val x:Any = 55
x: Any = 55
scala> val y = x.asInstanceOf[Int]
y: Int = 55
```

Потрібний тип вказується в квадратних дужках (параметр типу). Схожий метод **`isInstanceOf`** дозволяє перевірити тип змінної:

```
scala> y.isInstanceOf[Double]
res1: Boolean = false
```

```
scala> y.isInstanceOf[Int]
```
res2: Boolean = true

Таким чином всі типи явно або неявно успадковуються від `Any`.

У типу `Any` є два прямих підтипу – `AnyVal` і `AnyRef`.

`AnyVal` призначений для представлення типів, які в JVM є об'єктами. *Числові* підтипи – `Int`, `Long`, `Double`, `Float`, `Short`, `Byte`, `Char`. *Нечислові* підтипи – `Boolean`, `Unit`. Всі ці типи (окрім `Unit`) відповідають примітивним типам з Java. Базовий тип `Unit`, відповідний `void` в інших мовах.

Між числовими типами діє автоматичне приведення (рис. 1.7):

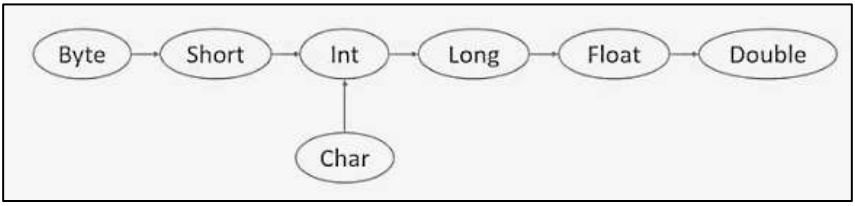


Рисунок 1.7 – Приведення типів

***Приклад* 2**.

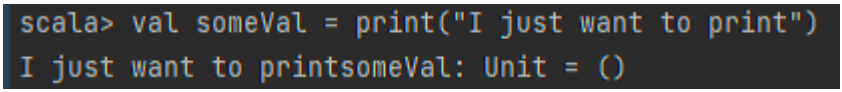
```
scala> val v1:Char = 'a'
v1: Char = a
scala> val v2: Double = v1
v2: Double = 97.0          // неявне приведення типів
scala> val v3: Long = v2  // помилка – неявне
                          // приведення з Double
                          // в Long неможливо
```

`Unit`

Все в Scala є виразом. Очікується, що вираз повертає значення. Але бувають випадки, коли жодного значення не повертається.

`Unit` особливо корисний у функціях, коли нічого, крім як вивести щось на екран, не потрібно (`print` – *виводить на друк текст, але нічого явно не повертає*):

```
val someVal = print("I just want to print")
```
I just want to printsomeVal: Unit = ()



Тобто код можна переписати так:

```
val someVal: Unit = print("I just want to print")
```

Якщо ініціювати змінну так:

`val v = ()`, то вона матиме тип `Unit`.

`AnyRef` є посиланням на тип (аналог java.lang.Object). Будь-який клас, створений користувачем буде явно успадковано від `AnyRef`.

***Приклад* 3**. Перевірити, чи є різні типи спадкоємцями `AnyRef`:

```
scala> class Myclass
scala> val c = new Myclass
scala> c.isInstanceOf [Any]
```
res1: Boolean = true
```
scala> c.isInstanceOf [AnyRef]
```
res2: Boolean = true
```
scala> "Hello".isInstanceOf [AnyRef]
```
res3: Boolean = true

// рядок має тип java.lang.String, який є підтипом java.lang.Object
```
scala> "Hello".getClass
```
res4: Class[_ <: String] = class java.lang.String

`Null` находиться внизу ієрархії посилальних типів, є спадкоємцем будь-якого типу посилань. Єдине значення типу `Null` – записуватись ключовим словом `null`. Це значення можна привласнити будь-якій змінній типу посилання.

```
scala> val v1:String = null
```
v1: String = null

```
scala> val v2:AnyRef = null
v2: AnyRef = null
scala> val v3:Myclass = null
v3: Myclass = null
scala> val v4:AnyVal = null          //помилка
```

Nothing знаходиться внизу ієрархії всіх типів, є підтипом будь-якого іншого типу, у тому числі і `Null`. В даного типу немає значень.

***Приклад* 4**. Визначити метод `error`, що генерує виключення. Метод приймає повідомлення, створює екземпляр виключення і викидає його. Тип результату – `Nothing` для сумісності з різними методами, які використовують дану функцію.

```
def error(mes:String):Nothing = throw new
IllegalStateException(mes)
def f1:Int = {
    val d = 2
    if(d==2) 2
    else error("5")
  }
def f2:String = {
    val s = "sss"
    if (s=="ccc") "ccc"
   else error("sss")
  }

scala> f1
res4: Int = 2
scala> f2
java.lang.IllegalStateException: sss
  at .error(<console>:11)
  at .f2(<console>:15)
  ... 28 elided
```

Тип `Nothing` сумісний з типами `Int` і `String`, тому конфлікту не відбувається. Можна зробити типом поверненого значення методу `error` тип `Int`, але тоді для методу `f2` виникає помилка, оскільки тип поверненого значення методу `f2` не відповідає поверненому типові методу `error`.

Таким чином `Nothing` є універсальним (оскільки є спадкоємцем будь-якого іншого типа).

***Приклад* 5**. Так само влаштований метод `???`.

Єдине призначення даного методу – генерувати виключення `NotImplementedError`. Тобто метод повертає `Nothing`.

Даний метод потрібний, коли оголошується функція без визначення (наприклад, функція доки ніде не викликається).

```
def f3 (a1:Int, a2:String):String = ???
```

Але код може бути скомпільований і запущений:

```
f3(10,"aaa")
```

Оголошення змінних

Змінні оголошуються ключовими словами `var`, `val` і `lazy val`:

`var` – змінні *можуть* змінювати своє значення після першої ініціалізації;

`val` – поля можна ініціалізувати *лише один раз* (аналог const);

`lazy val` – змінні також отримують значення *лише один раз* і *лише тоді, коли до цих значень здійснюється доступ*. Використання ледачих змінних має сенс, коли їх розрахунок займає значну кількість часу і результат розрахунку може і не знадобитися. У такій ситуації ледачі розрахунки можуть заощадити час.

Переважно використовувати `val` – змінні завжди, якщо немає прямої необхідності застосовувати варіант `var`. Це дає додаткову гарантію від випадкових помилок, пов'язаних з непередбаченими змінами в полях.

Якщо ви намагаєтеся дати змінній `aString` інше значення, компілятор видасть помилку і буде запропоновано змінити `val` на `var`:

Це пояснюється тим, що `val` (на відміну від `var`) створює змінну лише для читання (незмінну змінну).

Val – def

`val` використовується для оголошення змінних (незмінними – `immutable`), а `def` – для оголошення *функцій* (методів). В обох випадках синтаксис схожий: ключове слово (`val` чи `def`), ідентифікатор, знак рівності (=), вираз. І це не випадково, оскільки змінні також можуть бути інтерпретовані як вирази і оголошені словом `def`.

```
def x = 99
```
x: Int
```
val y = 99
```
y: Int = 99

В принципі, `x` і `y` можна далі використовувати на однакових правах:

```
x+y
```

res1: Int = 198

Однаково вказується і тип:

```
def x :String = "Hello"
```

x: String

Але між цими варіантами є і різниця – вона відбивається в результатах, що виводяться інтерпретатором (в разі `def` значення змінної `x` не виводиться на екран).

Якщо оголошення виглядає як:

```
def x = e
```

де `e` – вираз, о цей вираз не обчислюється відразу в момент оголошення, а обчислюється тільки там, де змінна буде використана (тому в нашому прикладі значення `x` не було виведено на екран; воно ще й не обчислювалося).

При оголошенні:

```
val y = e
```

вираз `e` обчислюється відразу, а потім вже підставляється на місце `y` його готове значення. Фактично оголошення із словом `def` майже те ж саме, що і `lazy val`.

Використання`;`

Знак "`;`" *обов'язковий* лише, якщо потрібно написати декілька виразів на одному рядку – таким чином відбувається розмежування написаних тверджень. Наприклад:

```
val aChar = 'c'; val aInt = 1
```

2. Робота з рядками

Функції роботи з рядками

```
 val aString: String = "Hello, world!"
println(aString.length)                 // 13
println(aString.charAt(1))              // e
println(aString.substring(0, 2))        // He
println(aString.split(" ").toList)
                            // List(Hello,, world!)
println(aString.startsWith("He"))       // true
println(aString.replace("!", "."))// Hello, world.
println(aString.toLowerCase)       // hello, world!
println(aString.toUpperCase)       // HELLO, WORLD!
println("abcd".reverse)            // dcba
println("abcd".take(2))            // ab
```

Невеликий приклад, як можна послідовно застосувати декілька методів і укласти код всього в одну строчку:

```
scala> println("example".toUpperCase.take(2)) //EX
```

Альтернативний код:

```
scala> val upperCaseVal = "example".toUpperCase
upperCaseVal: String = EXAMPLE
scala> val res = upperCaseVal.take(2)
res: String = EX
scala> println(res)
EX
```

Конвертація рядка в число і числа в рядок

Методи **`toInt`** і **`toString`**

```
val aNumber = "42".toInt
println(aNumber)                // 42
println(aNumber.getClass)       // int
```

Перетворення числа в рядок:

```
val aString = 42.toString
```

Додавання в початок і кінець

```
println('1' +: "42" :+ '3')     // 1423
println('a' +: "bc" :+ 'd')     // abcd
println("a" ++ "bc" :++ "d")    // abcd
```

Звернути увагу:

1) додавання в *початок* має на увазі наявність двокрапки після плюса. Відповідно, додавання в кінець – перед плюсом;

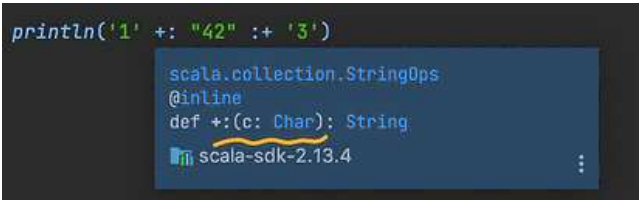
2) `+:` або `:+` має на увазі, що приєднувати будемо одиночні елементи (працюємо з типом `char`);

3) `++` або `:++` вказує, що приєднувати будемо декілька елементів (використовуємо елементи типу `String`), отже тут вже знадобляться подвійні лапки;

4) увага на типи, з якими працюємо: тут приєднання завжди відбувається до елементів типу `String` (у прикладі це "42" і "bc")

Насправді +, `+:` і `:+` є методами. Якщо навести на них покажчик мишки, то можна побачити:

```
def +:(c:Char):String
```



Це говорить про те, що `+:` приєднує лише значення типа `Char`.

Якщо переписати код у точковій нотації, вийде так:

```
val aStr = "42".+:('1')          // 142
println(aStr.:+('3'))            // 1423
```

Тому, якщо писати

```
println('1' :+ "42")             //Error
```

код буде видавати помилку, адже по суті тут була спроба зробити наступне:

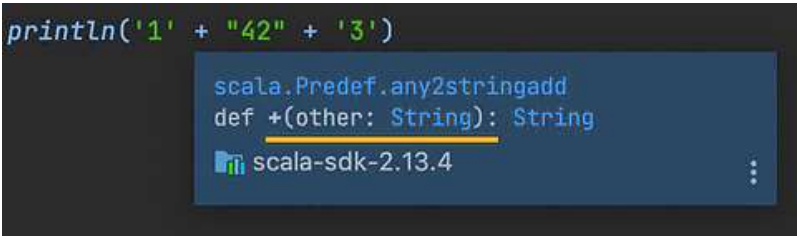
```
val aStr = '1'.:+("42")
println(aStr)
```

Крім того, що `:+` вимагає як аргумент параметр типу `Char`, тут спроба викликати метод `:+` для `'1'`, тобто для `Char`, а даний метод застосовний саме до рядків.

Схожа ситуація складається з оператором `+`

Але його відмінністю є те, що як аргумент йому підійдуть параметри типу `String`:

```
def +(other: String):String
```




```
val aStr = '1'.+("42")
println(aStr.+("3"))      // 1423
```

Як оператори працюють відносно колекцій:

```
println(1 +: List(2, 3))        // List(1, 2, 3)
println(List(1, 2) ++ List(3, 4)) // List(1, 2, 3, 4)
println(List(1, 2) +: List(3, 4))
                                 // List(List(1, 2), 3, 4)
```

Інтерполяція рядків

Інтерполяція застосовується, коли в рядок необхідно підставити якесь значення змінної.

1. **s-інтерполятор** – використовується для підставляння значення *змінної типу `String` в рядок*:

```
val name = "John"
println(s"Hello, $name") // Hello, John
```

Увага на використання `$` у коді! Це гарантує, що `name` буде *інтерпретоване як змінна*, значення якої потрібно підставити в рядок.

Якщо необхідно вставити вираз, то цей вираз вказуємо у фігурних дужках:

```
val name = "John"
val surname = "Smith"
println(s"Hello, ${name + surname}")
                              // Hello, JohnSmith
```

Є також функція `printf`, що приймає рядок опису формату в стилі мови C:

```
printf("Hello, %s! You are %d years old.\n", name, age)
```

Механізм інтерполяції рядків:

```
print(f"Hello, $name! In six months, you'll be ${age + 0.5}%7.2f years old.%n")
```

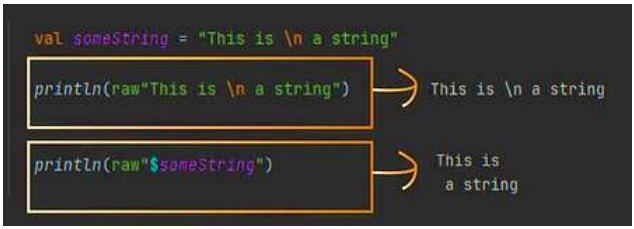
Форматований рядок передує буквою `f`. Вона може містити вирази, що починаються з символу `$`, і наступні за ними специфікатори формату в стилі мови C. Вираз `$name` у прикладі вище буде замінено значенням змінної `name`. Вираз `${age + 0.5}%7.2f` буде замінене значенням `age + 0.5`, що відформатоване як дійсне число в полі завдовжки 7 символів і з точністю до 2 знаків після коми. Вирази, що не є простими змінними, повинні полягати в `${...}`.

2. **raw-інтерполятор**

В Scala "raw-інтерполятор" не є стандартним терміном або концепцією. Це "raw string" або "raw string interpolator", які зазвичай використовуються для обробки рядків без спеціального оброблення службових символів, таких як `\n`, `\t`, `\\`, і т. д.

В Scala можна використовувати префікс `raw` разом зі стрічковим інтерполятором для створення такого рядка, у якому з префіксом `raw`, символи `\n` і `\t` не будуть екрануватися, і рядок буде містити саме такий вміст, як було введено. "Raw string" дуже корисний, коли потрібно включити рядок, який містить багато службових символів, без необхідності їхньої екранування.

```
scala> val someString = "This is \n a string"
```
someString: String =
This is
a string
```
scala> println(raw"This is \n a string")
```
This is \n a string
```
scala> println(raw"$someString")
```
This is
a string



3. Вирази і інструкції

Вирази (Expressions) і Інструкції (Instructions)

Створимо змінну `aVal` і привласнимо їй значення `1 + 2 * 3`

```
val aVal = 1 + 2 * 3
```

Змінна `aVal` буде мити тип `Int` і буде зберігати в собі число 7, що є результатом математичного виразу `1 + 2 * 3`

Таким чином, цей приклад демонструє дві основні властивості виразів:

1) вирази повертають якесь значення;

2) мають тип.

Саме властивість 1 дозволяє відразу прописувати вираз в `print` (без його попереднього привласнення якій-небудь змінній):

```
print(1 + 2 * 3)
```

И якщо вираз – це коли повертається якесь, як правило, обчислене значення, то інструкція – це коли виконується якась дія, але вираз не повертається → все в Scala являється виразом.

Типи виразів в Scala

Список того, що застосовується в Scala-виразах:

```
+ - / *
& | ^ << >> >>>
== != > >= <= < ! && ||
```

if-Вираз

Все в Scala являється виразом – це дуже поважно зрозуміти і запам'ятати. І `if` не є виключенням.

Розберемо приклад:

```
val aCondition = true
val ifExpressionValue = if (aCondition) "True Condition" else "False Condition"
println(ifExpressionValue)      // True Condition
```

Приклад показує, що в Scala, на відміну від інших мов програмування відразу пишеться значення, яке потрібно привласнити в першому і в другому випадку:

```
if (условие): привласнити значення (інструкція)
else: привласнити змінній інше значення (інструкція)
```

Можна взагалі прописати if-вираз в `print` і компілятор надрукує результат.

```
val aCondition = true
println(if(aCondition) "True Condition"
        else "False Condition")
                              // True Condition
```

Виконуються обидві властивості виразів: повертається значення `(True Condition) с типом (String).`

В Scala кожен вираз має тип. Наприклад, вираз

```
if (x > 0) 1 else -1
```

має тип `Int`, тому що *обидві гілки мають тип* `Int`:

```
scala> val x = 10
x: Int = 10
scala> if (x>0) 1 else -1
res1: Int = 1
```

Типом виразу, здатного повертати *значення різних типів*, такого як

```
if (x > 0) "positive" else -1
```

є супертип для обох гілок. У даному прикладі одна гілка має тип `java.lang.String`, а інша – тип `Int`. Їх загальний супертип називається `Any`:

```
scala> if (x>0) "positive" else -1
res2: Any = positive
```

Якщо гілка `else` відсутня,

```
if (x > 0) 1
```

може вийти, що інструкція `if` не матиме значення. Проте в Scala кожне вираження передбачає наявність якого-небудь значення. Ця проблема була вирішена введенням класу `Unit`, єдине значення якого записується як `()`.

Інструкція `if` без гілки `else` еквівалентна інструкції

```
if (x > 0) 1 else ()
```

```
scala> val e = 5 > 3
e: Boolean = true
scala> val x = if (e) 55
x: AnyVal = 55
scala> val e = 5 < 3
e: Boolean = false
scala> val x = if (e) 55
x: AnyVal = ()
```

Блок коду (Code block)

Блок коду – це все ті рядки коди, що пишуться у фігурних дужках.

```
val aCodeBlock = {
    val someVal = 1
    val y = 2
    if (someVal + y > 1) true else false
}
println(aCodeBlock)       // true
```

Блок коду також є виразом, результат (і тип) якого дорівнює результату (і типові) *останнього* описаного усередині блоку коду виразу.

Якщо, наприклад, дописати рядок `"String to return"` у кінець попереднього блоку коду, то `aCodeBlock` змінить тип на `String`, що відповідає типові останнього виразу `"String to return"`, описаного в блоці:

```
val aCodeBlock = {
  val someVal = 1
  val y = 2
  if (someVal + y > 1) true else false
  "String to return"
}
println(aCodeBlock)     // "String to return"
```

В цілому, це спосіб може обійтися без `return`.

Все, що сталося усередині блоку коду, залишається усередині блоку коду. Тобто якщо усередині блоку визначити змінну, яку потім потрібно використовувати поза цим блоком, – буде помилка.

Цикли *while* і *for*

Оператор while

```
var i = 0
while (i <= 5)  {
 print(i + " ")           //0 1 2 3 4 5
 i+= 1
}
```

Оператор for

У заголовку циклу `for` допускається вказувати декілька генераторів у формі змінна <- выраз, розділяючи їх крапками з комою. Наприклад:

```
scala> for (i <- 1 to 3; j <- 1 to 3)print(f"${10 * i + j}%3d")
```

11 12 13 21 22 23 31 32 33

Оператор `for` підтримує *обмежувач* (`guard`). Для цього треба після параметрів циклу вставити умовний оператор `if`:

```
scala> def isOdd(in: Int) = in % 2 == 1
```

isOdd: (in: Int)Boolean

```
scala> for {i <- 1 to 5 if isOdd(i)} print(i)
```

135

```
scala> for {i <- 1 to 5; j <- 1 to 5 if ((i * j) % 2 == 0)} printf(f"${i * j}%2d")
```

2 4 2 4 6 8 10 6 12 4 8 12 16 20 10 20

Коли тіло циклу `for` починається з інструкції `yield`, цикл конструюватиме колекцію, додаючи в неї по одному елементу в кожній ітерації. Такого роду цикли називають for-генераторами (*for-comprehension*):

```
scala> for (i <- 1 to 10) yield i % 3
```

res1: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 2, 0, 1, 2, 0, 1, 2, 0, 1)

```
scala> for (i <- 0 to 1; c <- "Hello") yield (c + i).toChar
```

res2: scala.collection.immutable.IndexedSeq[Char] = Vector(H, e, l, l, o, I, f, m, m, p)

Змінну циклу можна змінювати із заданим кроком:

```
for (i <- 0 to (7,2))print(i + " ")
```

0 2 4 6

За допомогою методу `reverse` можна виконувати обхід в зворотному порядку:

```
for (i <- (0 to (7,2)).reverse)  print(i + " ")
```

6 4 2 0

Цикл `for` можна ефективно використовувати для трансформації колекції або її частини в нову колекцію. Визначимо такий список:

```
scala> val a = (1 to 18 by 3).toList
```

a: List[Int] = List(1, 4, 7, 10, 13, 16)

визначає ранг – послідовність цілих чисел від 1 до 18 через 3, а метод `toList` перетворює ранг в список:

```
List(1, 4, 7, 10, 13, 16).
```

За допомогою ключового слова `yield` створили нову колекцію:

```
scala> for {i <- a if isOdd(i)} yield i
```

res13: List[Int] = List(1, 7, 13)

Зіставлення із зразком (*Pattern Matching*)

У функціональних мовах зіставлення із зразком (Pattern Matching) має основоположне значення. У Scala цей механізм якраз і побудований на `case`.

```
var sign = 1
val ch: Char = '+'
ch match {
  case '+' => sign = 1
  case '-' => sign = -1
  case _ => sign = 0
}
println(sign)                          //1
```

```
def fib(x: Int): Int = x match {
    case 0 => 0
    case 1 => 1
    case _ => fib(x - 1) + fib(x - 2)
  }
  println(fib(6))                    //8
```

Зіставляти із зразком можна будь-які типи даних (*буде пізніше детальніше*). Наприклад, аргумент типу `String`:

```
def f(name: String) = name match {
    case "Elwood" | "Madeline" => Some("Cat")
    case "Archer" => Some("Dog")
    case "Pumpkin" | "Firetruck" => Some("Fish")
    case _ => None
  }
  println(f("Elwood").get)           //Cat
```

Для ілюстрації результатом порівняння в даному прикладі вибрані елементи колекції `Option` і для здобуття результату типу `String` був використаний метод `get`. При варіанті, коли виходить `None`, метод `get` дасть виключення. Значить, краще використовувати метод `getOrElse`:

```
print(f("Elwoodo").getOrElse("No"))
```

Зіставляти можна паралельно будь-які типи даних в одному операторові. У прикладі тип аргументу функції оголошений, як `Any` (щоб можна було вводити дані різних типів). Результат завжди матиме тип `String`.

```
def f(x: Any): String = x match {
  case 1 => "One"
  case "David" | "Archer" | Some("Dog") => "Walk"
  case _ => "No Clue"
}
print(f("David"))              // Walk
```

Тестувати можна і типи даних. У кожному `case` ввели нові змінні: `s,i, a`; всім їм за умовчанням передаватиметься значення аргументу `x`. Використаний в прикладі тип `AnyRef` майже те ж саме, що і тип `Any`. Метод `getClass` визначає клас об'єкту, а метод `getName` виділяє лише ім'я класу:

```
def f(x: Any) = x match {
  case x: String => "String, length "+ x.length
  case x: Int if x > 0 => "Natural Int"
  case x: Int => "Another Int"
  case x: AnyRef => x.getClass.getName
  case _ => "null"
}
println(f("Hello"))       // String, length 5
println(f(-8))            // Another Int
println(f(8))             // Natural Int
println(f(0.87))          // java.lang.Double
print(f(null))            // null
```

4. Функції

Функція створюється таким чином.

1.Починається все з ключового слова `def`.

2.За яким слідує ім'я функції.

3.У круглих дужках (при необхідності) вказуються параметри і їх типи.

4.Після дужок ставиться двокрапка і конкретизується тип значення, що повертається.

Примітка: цілком можливо обійтися і без вказівки типу значення, що повертається і покластися в цій справі на компілятор.

5. Нарешті – ставиться знак рівності та прописується тіло функції.

Тіло функції цілком можна писати *у фігурних дужках (*в цьому випадку справу матимемо з блоком коду code block):

```
def aPerson(name: String, surname: String): String = {
        s"$name $surname"
}
```

Виклик функції:

```
println(aPerson("John", "Smith"))
```

Можливо прописати `print` в тілі самої функції, просто для цього необхідно змінити тип значення, що повертається на `Unit`:

```
  def aPerson(name: String, surname: String): Unit = println(s"$name $surname")
```

Виклик функції:

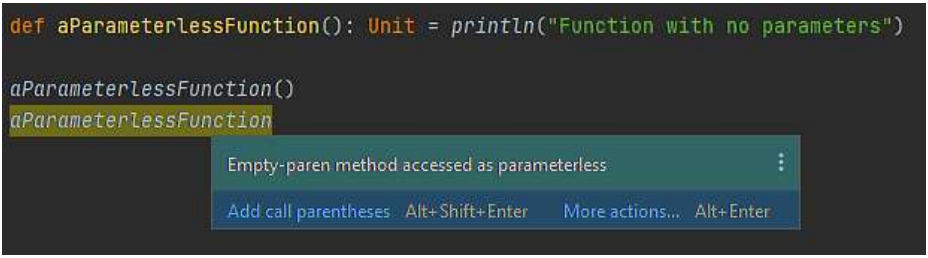
```
aPerson("John" , "Smith")
```

Функція без параметрів

Якщо створюється функція без параметрів, то її виклик можливий двома способами:

1) з круглими дужками на кінці;

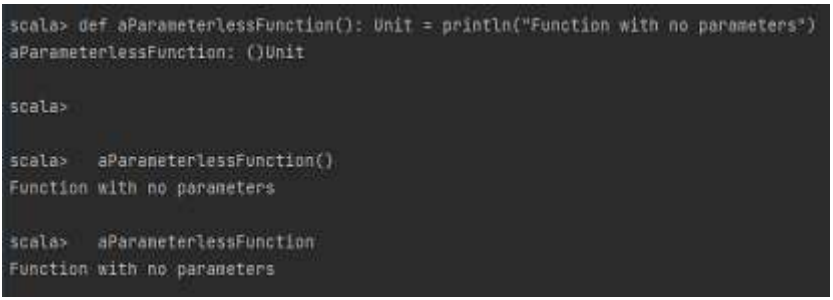
2) без круглих дужок на кінці. В цьому випадку компілятор просто повідомить, що метод, що викликається, – parameterless тобто без параметрів.




```
def aParameterlessFunction(): Unit =
    println("Function with no parameters")
```

Виклик функції:

```
aParameterlessFunction()
aParameterlessFunction
```



В Scala 3 функції без параметрів викликати без дужок вже не можна.

Параметри за умовчанням

У Scala можна встановити *дефолтні* значення для параметрів, що дозволить зайвий раз не вказувати параметри при виклику функції. Тільки треба стежити за порядком аргументів:

```
def aFunctionWithDefaultParameter(x: Int,
 y: String = "Default Parameter"): String = {
    s"x = $x and y = $y"
  }
println(aFunctionWithDefaultParameter(1))
             // x = 1 and y = Default Parameter
```

Виклик за ім'ям (call-by-name) vs Виклик за значенням (call-by-value)

Щоб побачити різницю між цими двома викликами (крім того, що поряд з параметром, що викликається по імені, друкується знак =>) можна проаналізувати код: **call-by-name**

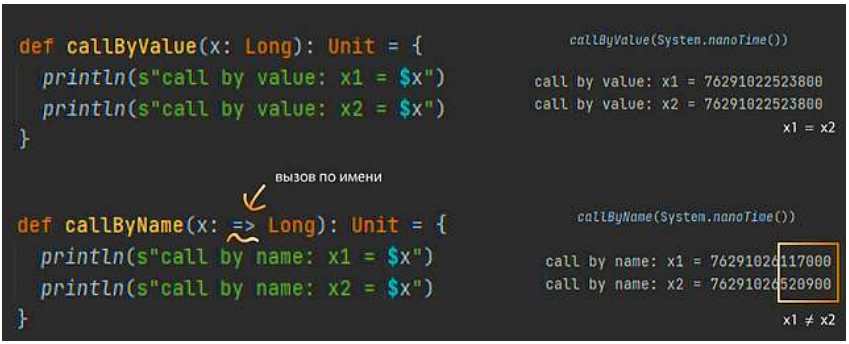
```
def callByValue(x: Long): Unit = {
    println(s"call by value: x1 = $x")
    println(s"call by value: x2 = $x")
  }
def callByName(x: => Long): Unit = {
    println(s"call by name: x1 = $x")
    println(s"call by name: x2 = $x")
  }
```

```
callByValue(System.nanoTime())
callByName(System.nanoTime())
```

Примітка: `System.nanoTime()` *повертає час виконання в наносекундах*.

Просто запам'ятати, що `callByName` йдуть *із стрілкою* і обчислюються (передаються у функцію) кожного разу при використанні усередині функції.

А тепер безпосередньо результат виконання:



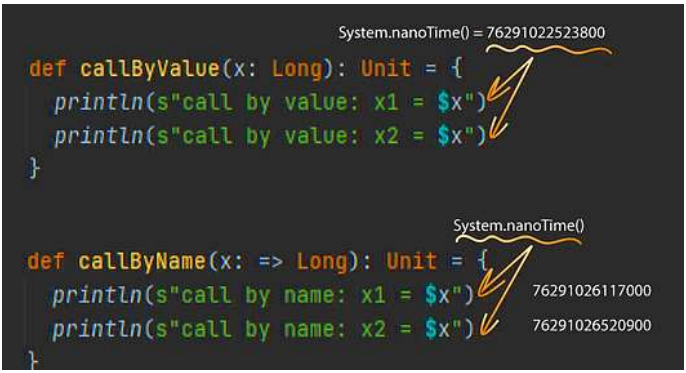
Якщо для `callByValue` на екран виводяться два однакові значення, то для `callByName` значення будуть різними. Це пояснюється різницею в підходах до обчислення значень параметрів. Виклик за значенням означає обчислення значення переданого виразу перед викликом функції.

Перевага: значення обчислюється лише один раз.

Виклик за ім'ям передбачає обчислення значення виразу в момент його виклику у функції.

Перевага: значення не обчислюється, якщо не використовується в тілі функції.

Для `callByValue` обчислили значення `System.nanoTime()` і підставили це значення у функцію. Проте для `callByName` попередніх обчислень не робилося, і значення обчислювалося вже безпосередньо у функції.



***Приклад* 6**. Яким буде результат коду?

```
def someFunc(): Int =  * someFunc() + 1
def callSomeFunc(x: Int, y: => Int) = println(x)
callSomeFunc(someFunc(), 1)
   // StackOverflowError  (нескінченна рекурсія)
```

Якщо змінити код:

```
def someFunc(): Int = 2 * someFunc() + 1
def callSomeFunc(x: Int, y: => Int) = println(x)
callSomeFunc(1, someFunc())           // 1
```

В даному випадку другий аргумент функції `callSomeFunc`, а саме `someFunc` не буде обчислений, оскільки використовується стратегія *call by name*. Отже функція `callSomeFunc` росто викличе `println` с аргументом 1.

***Приклад* 7**. Що буде надруковане при виклику функції `simpleTest` з різними способами передачі параметрів?

```
def calculateX(x: Int): Int = {
  println("считаем x")
  x
}
```

```
def calculateY(y: Int): Int = {
  println("считаем y")
  y
}
def simpleTest(x: Int, y: Int) = x * x
simpleTest(calculateX(5), calculateY(2))
                           // рахуємо x
                           // рахуємо y
def simpleTest(x: => Int, y: Int) = x * x
simpleTest(calculateX(5), calculateY(2))
                          // рахуємо y
                          // рахуємо x
                          // рахуємо x
def simpleTest(x: Int, y: => Int) = x * x
simpleTest(calculateX(5), calculateY(2))
                          // рахуємо x
def simpleTest(x: => Int, y: => Int) = x * x
simpleTest(calculateX(5), calculateY(2))
                          // рахуємо x
                          // рахуємо x
```

Вкладені функції

Існує можливість визначити функцію або навіть декілька функцій усередині однієї функції і там же її(або їх) викликати.

```
 def aBossFunction(): String = {
   def aHelperFunction(): String =
     "I'm here to help"

   aHelperFunction()
  }
println(aBossFunction)    // I'm here to help
```

Є можливість при оголошенні методу вказувати, що аргументи або результат мають **невизначений тип `Any`**.

```
scala> def f[Any](p: Any): List[Any] = p :: Nil
f: [Any](p: Any)List[Any]
```

```
scala> f(10)
```
res1: List[Int] = List(10)
```
scala> f("Bob")
```
res2: List[String] = List(Bob)
```
scala> f(10,"Bob")
```
res3: List[(Int, String)] = List((10,Bob))

Метод `f` приймає один аргумент типу `Any` і повертає список з елементами типу `Any`. У правій частині використаний оператор :: `::` (фактично це теж метод), який додає елемент `p` до списку в його початок.. Замість списку стоїть спеціальне значення `Nil`, що представляє в даному контексті порожній список (замість `Nil` можна явно вказати `List()`). Для того, щоб метод `f` був здатний визначати (виводити) конкретного типу результату з контексту замість невизначеного `Any`, після імені методу треба поставити `[Any]` (називається параметр типу). Без цього результат так і матиме тип `Any`.

Якщо не вказати параметр типу. Тепер всі елементи списку мають невизначений тип `Any`:

```
scala> def f(p: Any): List[Any] = p :: Nil
```
f: (p: Any)List[Any]
```
scala> f("Bob")
```
res4: List[Any] = List(Bob)

Змінна кількість аргументів

```
def sum(args: Int*) = {
var result = 0
for (arg <- args) result += arg
result
}
```

Цій функції можна передати скільки завгодно багато аргументів:

```
val s = sum(1, 4, 9, 16, 25)
println(s)                                  //55
```

Для діапазону чисел необхідно повідомити компілятор, що параметр повинен інтерпретуватися як послідовність аргументів. Для цього використовується**: _***.

```
val s = sum(1 to 5: _*)
println(s)                                  //15
```

Процедури

У Scala є спеціальна форма запису для оголошення функцій, що *не повертають значень*. Якщо тіло функції поміщене у фігурні дужки без попереднього символу =, тоді повернене значення матиме тип `Unit`. Такі функції називаються **процедурами**. Наприклад:

```
def box(s : String) {           // тут немає =
val border = "-" * (s.length + 2)
print(f"$border%n|$s|%n$border%n")
}
```

Або можна явно вказувати тип значення, що повертається `Unit`:

```
def box(s : String): Unit = {
...
}
```

Рекурсія

Розглянемо приклад обчислення факторіалу:

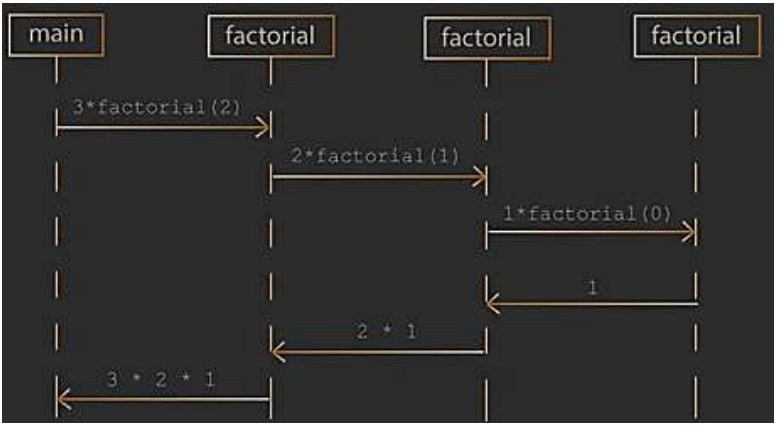
```
def factorial(n: Int): Int = {
    if (n <= 0) 1
    else n * factorial(n - 1)
}
println(factorial(3))           // 6
```

Проміжні обчислення зберігаються в стеку, а це означає, що кожен виклик рекурсивної функції зв'язаний з використанням стека.

Можна змінити код:

```
  def factorial(n: Int): Int = {
    if (n <= 0) 1
    else {
```

```
    println(s"Маємо число $n, для якого рахуємо
факторіал ${n-1}")
          var res = n * factorial(n-1)
          println(s" Обчислили факторіал $n")

          res
        }
      }
    factorial(3)
    // Маємо число 3, для якого рахуємо факторіал 2
    // Маємо число 2, для якого рахуємо факторіал 1
    // Маємо число 1, для якого рахуємо факторіал 0
    // Обчислили факторіал 1
    // Обчислили факторіал 2
    // Обчислили факторіал 3
```



Зберігаються всі виклики до стека – а це має свої обмеження по пам'яті. Як наслідок – `StackOverflowError` при завданні досить великого числа, для якого треба порахувати факторіал.

Тому потрібно використовувати *хвостову рекурсію*.

Хвостова рекурсія (Tail Recursion)

Переписаний код із застосуванням хвостової рекурсії виглядає таким чином:

```
def factorialWithTailRecursion(n: Int): Int = {
 def loop(x: Int, accumulator: Int = 1): Int = {
      if (x <= 1) accumulator
      else loop(x-1, x*accumulator)
 }
   loop(n, 1)
}
```

Відмітною особливістю функції з хвостовою рекурсією є те, що *останньою операцією йде виклик цієї самої функції.* І тепер проміжні обчислення накопичуються в акумуляторі.

Таке вживання допоміжної функції, написаної в стилі хвостової рекурсії, дозволяє обійтися без використання додаткового стека для зберігання проміжних результатів.

Як писати функції в стилі хвостової рекурсії

1. Визначити функцію. Назвемо її *основною* (`factorialWithTailRecursion`).

2. Усередині основної функції прописати ще одну функцію. Назвемо її *допоміжною* (`loop`).

3. Як аргументи допоміжної функції вказати *число викликів* (`x`) (цей аргумент збігається з одним з аргументів основної функції – `n`), плюс *акумулятор* (`accumulator`), який міститиме кінцевий результат.

4. Прописати код з рекурсією в тілі допоміжної функції з використанням акумулятора так, щоб значення поступово накопичувалося (`x*accumulator`).

5. Викликати допоміжну функцію з основної функції (`loop(n, 1)`).

Не забути вказати початкове значення акумулятора, що визначає відправну точку всіх обчислень. Використовувати аргумент за умовчанням для оптимізації коду (`accumulator: Int = 1`). Тоді виклик допоміжної функцію з основної функції буде наступним `loop(n)`.

Як перевірити функцію на предмет хвостової рекурсії

Якщо немає упевненості, чи дійсно імплементували хвостову рекурсію – можна скористатися анотацією **`@tailrec`.** Цим вказується компілятору, що функція має бути *tail recursive*, і якщо це не так, код просто не запуститься.

Для цього потрібно імпортувати

```
import scala.annotation.tailrec,
```

а сам `@tailrec` прописати перед рекурсивною функцією:

```
def factorialWithTailRecursion(n: Int): Int = {
    @tailrec
 def loop(x: Int, accumulator: Int = 1): Int = {
      if (x <= 1) accumulator
      else loop(x-1, x*accumulator)
    }
    loop(n)
  }
```

***Приклад* 8**. Виведення слова `n` разів

Наприклад, таким `while`-циклом можна надрукувати слово `n` разів.

```
  var i = 0
  while (i < 3) {
    println("hello")
    i += 1
  }
```

Цей приклад, що використовує хвостову рекурсію:

```
def repeatWord(word: String, n: Int): String = {
    @tailrec
 def loop(i: Int, acc: String = word): String = {
      if (i <= 1) acc
      else loop(i - 1, s"$word $acc")
 }
    loop(n)
}
println(rec(3,"hello"))   // hello hello hello
```

***Завдання до практичної роботи* 1**

1.1 Змусити функцію `someFunc` завжди видавати результат множення `x` на 2. Для цього:

– написати метод `multiply`, який нескінченно множить результат на 2;

– виправити код функцій, де це треба.

```
def aCondition(): Boolean =
    if ( 1 > 2) true
    else false

def someFunnc(x: Int, y: Int): Unit = {
  if (aCondition) x * 3
  else  multiply(y)
}
// Виклик функції має бути наступним
println(someFunnc(3, multiply(4)))          // 6
```

1.2 Написати функцію `powerOfTwo`, яка обчислює ступінь двійки.

У коді треба передбачите виконання наступних умов:

– потрібна для обчислення ступінь (тип `Int`) передається в функцію як аргумент;

– функція повертає число;

– використовується *хвостова рекурсія*;

– код видає коректний результат навіть для ступенів вище 32.

Зауваження:

– прописати в потрібному місці коду `@tailrec` ;

– акумуляторі вказати дефолтне значення;

– вернути увагу на тип `BigInt`;

– достатньо реалізувати обчислення лише позитивних ступенів.

```
Sample Input: 2
Sample Output: 4
```

1.3 Написати програму, яка:

– збільшує задане число `x` на число `y` `n`-у кількість разів (x=10, y=2, n=5 – *в завданні*) ;

– виводить на екран отримане на кроці 1 число стільки раз, скільки в ньому цифр, і фразу `is the result`

Зауваження:

– обмежитися цілими числами `Int`;

– дану програму потрібно написати, використовуючи *лише хвостову рекурсію.*

```
Sample Input: 10 2 5
Sample Output: 20 20 is the result
```

1.4 Модифікувати поданий на вхід рядок так, щоб слова в ньому розмістилися в зворотному порядку. Наприклад, рядок `I like Scala` повинен трансформуватися в `Scala like I`

Зауваження:

– вернутися до рядка, що подається на вхід, можна через змінну `input`. Прочитати інформацію з вхідного потоку можна оператором

```
val input = StdIn.readLine
```

– надрукувати кінцевий результат;

– передбачити видалення всіх зайвих пропусків з вихідного рядка;

– перевірити, щоб на початку і кінці рядка пропусків взагалі не було.

```
Sample Input: I like Scala
Sample Output: Scala like I
```

Контрольні запитання

1. Які інтерпретатори використовуються для команд Scala?
2. Чи відрізняються специфікатори `val` і `var`?
3. Чи відрізняються специфікатори `val` і `def`?
4. Що таке типи `AnyRef` та `AnyVal`?
5. Якою командою можна перевірити тип змінної?
6. Який метод застосовується для зміни типів?
7. Коли використовується тип `Unit`?
8. Які основні функції роботи з рядками?
9. Що таке s-інтерполятор?
10. Що таке raw-інтерполятор?
11. Які основні властивості виразів в Scala?
12. Який синтаксис та особливості if-виразів?
13. Що таке блок коду (Code block)?
14. Який синтаксис та особливості циклів `while` і `for`?
15. Як зіставляти із зразком будь-які типи даних (`Pattern Matching`)?
16. Чим відрізняється виклик параметрів за ім'ям (call-by-name) та виклик за значенням (call-by-value)?
17. Що таке "хвостова рекурсія"?

Практична робота 2

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ В SCALA

Мета: Одержання практичних навичок при використанні класів та об'єктів у мові Scala

Теми для попереднього опрацьовування

– Вступ до класів
– Конструктори
– Перевантаження методів класу
– Об'єкти, компаньйони

Теоретичні відомості

1. Класи

Для оголошення класу використовується ключове слово `class`, новий екземпляр оголошується через `new`. Методи класу є функціями, оголошеними в його тілі, а поля класу вказуються відразу ж після імені як список аргументів. роте за умовчанням вони оголошуються як `private val`, тому, якщо не вказати жодних модифікаторів, вказане поле буде доступне лише усередині класу і буде незмінним. Але головна відмінність від Java – відсутність конструктора. Код, який повинен виконуватися під час створення об'єкту, пишеться безпосередньо в тілі класу.

Клас оголошується за допомогою ключового слова `class`, за яким слідує ім'я класу:

```
class Student
```

Створення екземпляра класу відбувається за сприяння ключового слова `new`:

```
val student = new Student
```

Приклад використання класу в Scala:

```
class Foo(x: Int) {
  def bar(y: Int) = x + y
}
```

```
val foo = new Foo(1)
foo.bar(1)                  // 2
```

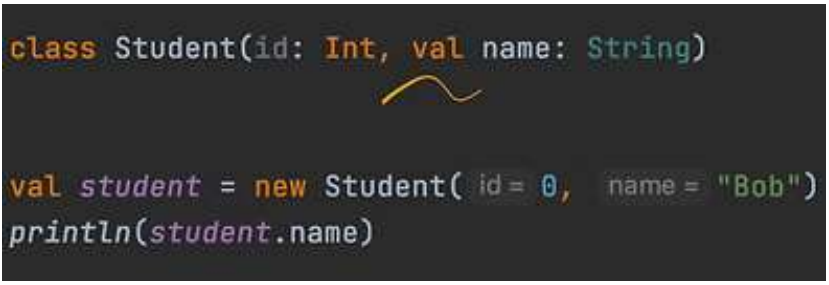
Конструктор класу

Розглянемо клас з параметрами `id` і `name`:

```
class Student(id: Int, name: String)
```

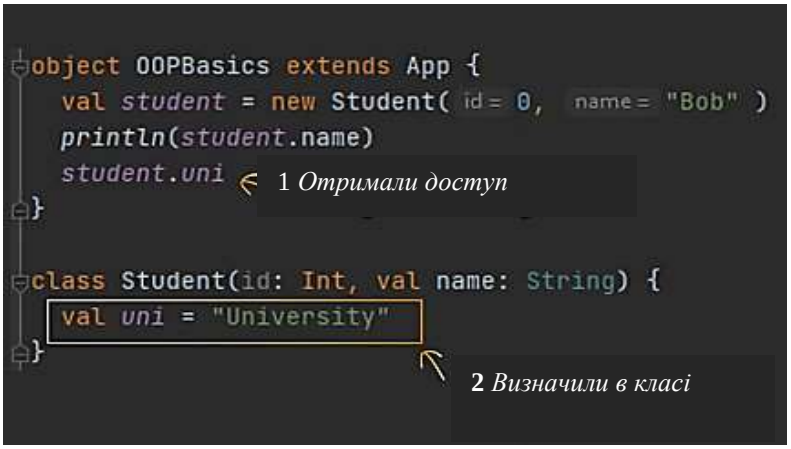
Часть `Student(id: Int, name: String)` називається конструктором.

У студента з'явилося ім'я, до якого потрібно звернутися. Але не можна написати `student.name`. Оскільки *параметр класу ще не член класу*, доступ до якого можна отримати через крапку. Щоб цей доступ зробити можливим (*щоб поле стало членом класу*) – необхідно додати ключове слово `val` або `var`, прописавши його перед параметром. Тоді отримуємо поле класу, до якого можна при необхідності звернутися (*без* `val`/`var` *поля дійсно* `private`*, якщо використані у функції усередині класу, але наявність* `val`/`var` *не міняє* `private` *на* `public`*, а дає доступ до однойменного геттера).*

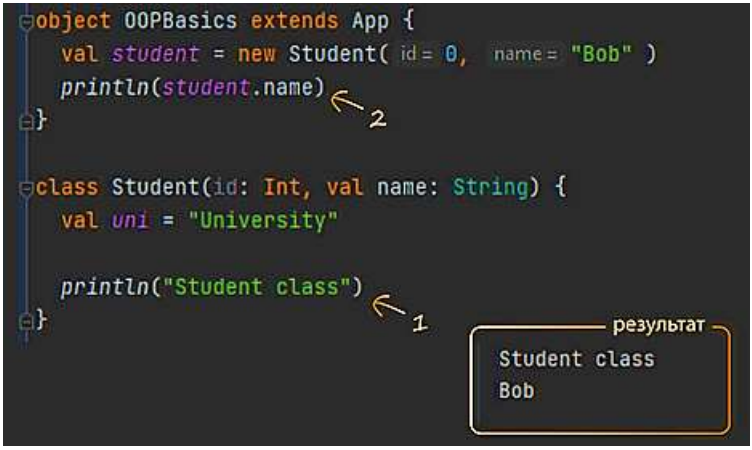


Тіло класу

Тіло класу описуємо у фігурних дужках (як і блоки коду). До всього, що визначається в тілі класу, можна дістати доступ через крапку (*тобто змінні, описані усередині класу, є полями класу!*).



Пропишемо оператора `print` в тілі класу. А іншим оператором `print` виведемо значення змінної `student.name`.



Значення `name` друкується *останнім*. Це пов'язано з тим, що при створенні екземпляра класу *автоматично виконуються всі конструкції, описані усередині класу*:

```
class Student(id: Int,val name: String){
    val uni = "Universiry"
    println("Student class")
    println (uni)
  }
val student = new Student(id = 0, name = "Bob")
  println(student.name)   // Student class
                          // Universiry
                          // Bob
```

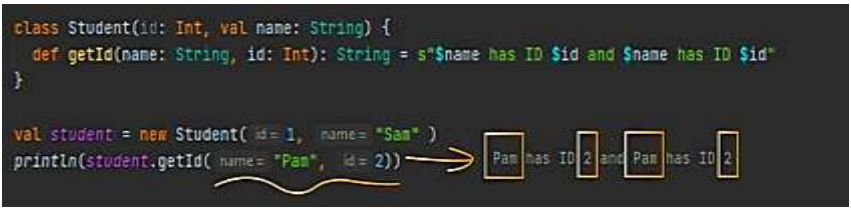
Методи класу

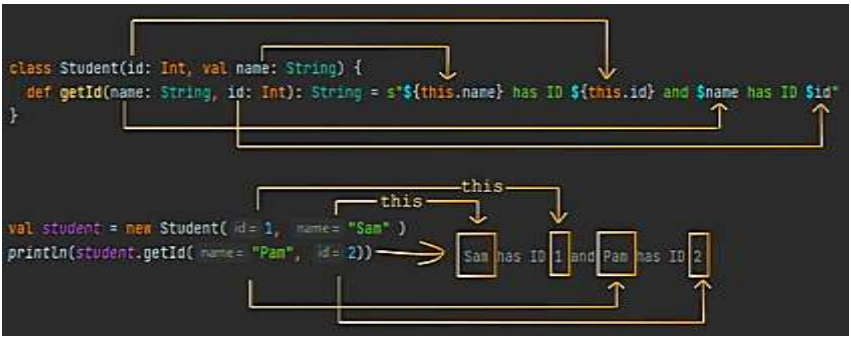
Для виклику методу також застосовується точкова нотація.

Ключовий момент, який варто запам'ятати, – це те, як використання ключового слова `this` впливає на результат програми. Розглянемо дію `this` на прикладі `getId`:

```
class Student (id:Int,val name:String){
  def getID(name:String, id:Int):String =
  s"${this.name} - has ID ${this.id} and $name -
has ID $id"
  }

val student = new Student(id = 0,name = "Bob")
println(student.getID("Sam",11))
                / Bob - has ID 0 and Sam - has ID 11
```

Т.ч. з параметром `this` використовуються параметри *класу*, а без `this` – параметри *методу класу*.





***Приклад* 1**. Нехай існує клас:

```
class SomeClass(a: Int, b: Int, val c: Int) {
  def someFunc_b(): Int = b
  println(a)
}
```

`a` – існує лише як *локальна* змінна конструктора. Тоді доступність/недоступність пояснюється зоною видимості (у тілі класу);

`b` – використано у *функції* (ключове слово – функція) усередині класу, тому доступність/недоступність пояснюється тим, що `b` є `private` полем;

`c` – наявність `val` робить полем класу, що і пояснює доступність поза класом.

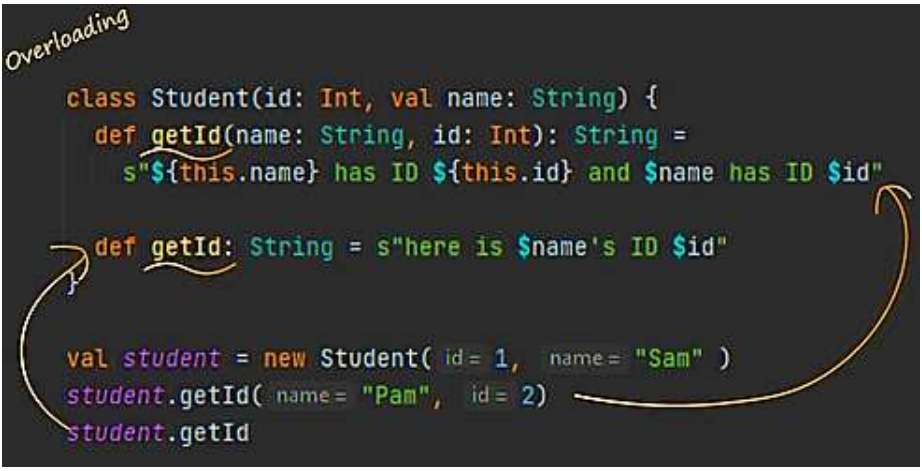
```
val ob = new SomeClass(1,2,3)
                    //1 (println(a) у  класі)
println(ob.someFunc_b())
                    //2 (someFunc_b виклик ф-ції)
println(ob.c)       //3  (друк поля public)
```

***Приклад* 2**. Все в Scala є виразом. Так що такий код цілком допустим:

```
class A(val a: String)
class B(val someVal: A)
val a = new A("value a")
val b = new B(a)
println(b.someVal.a)      // value a
```

Перевантаження методу (Overloading)

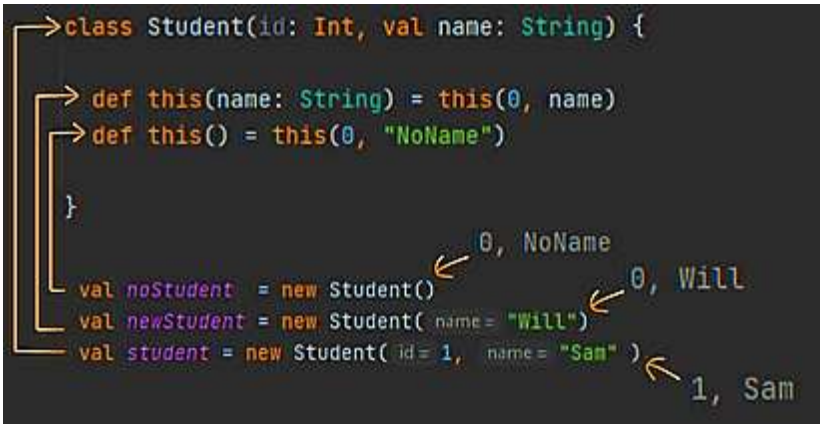
Дозволяє мати функції з однаковою назвою. Єдина умова – щоб набір аргументів і(або) їх тип був *різний*, щоб компілятор міг зрозуміти, виклик якої саме функції потрібний.



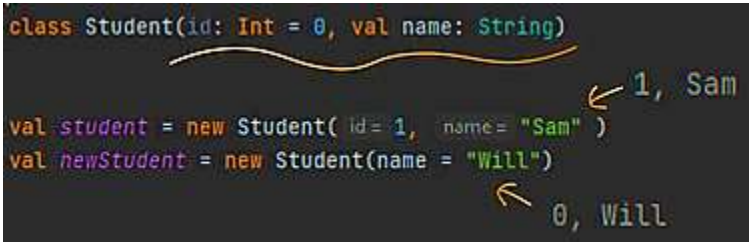

```
class Student (id:Int,val name:String){
  def getID(name:String,id:Int):String =
    s"${this.name} - has ID ${this.id} and $name
- has ID $id"
  def getID:String = s"here $name - has ID $id"
}
val student = new Student(id = 0,name = "Bob")
println(student.getID("Sam",11))
        // Bob - has ID 0 and Sam - has ID 11
println(student.getID)
        // here Bob - has ID 0
```

Перевантажені конструктори

Клас може мати декілька конструкторів. Це досягається за рахунок використання `def this`. Припустимо, потрібен конструктор, який би за умовчанням використовував 0 як значення `id`:



Альтернативою буде завдання *значення за умовчанням* в конструкторі класу:




```
class Student (id:Int = 0,val name:String){
   def this (name:String) = this(0,name)
   def this () = this(0,"NoName")
}
val noStudent = new Student()
println(noStudent.name)                // NoName
val newStudent = new Student("Sam")
println(newStudent.name)               // Sam
val student = new Student(id = 0,name = "Bob")
println(student.name)                  // Bob
```

Якби клас приймав не 2, а скажемо 4 різних параметра (`id, name, surname, date_of_birth`), то можна було б створити по одному конструктору `def this` на кожен випадок.

***Приклад* 3**. Дано наступне визначення класу:

```
class Employee(val name: String, var salary: Double) {
def this() { this("John Q. Public", 0.0) }
// или def this() = this("John Q. Public", 0.0)
}
```

Необхідно переписати код так, щоб клас містив явні визначення полів і мав головний конструктор за умовчанням.

```
class Employee(private val name: String =
"John Q. Public", private var salary: Double =
 0.0) {}

 var e = new Employee("Jane", 1000.0)
```

2. Об'єкти

Створення об'єкту дуже нагадує створення класу лише з ключовим словом `object`:

```
object Number
```

Використовується, якщо потрібно створити змінні або методи, доступні у будь-який час, причому *без додаткового оголошення* екземплярів класу (щось таке універсальне).

Наприклад, наступний код дозволить використовувати задане число `Pi` в будь-якому місці призначеної для користувача коду.

```
object Number {
  val Pi = 3.14
}
println(Number.Pi)                    // 3.14
```

В Scala `public` вказувати не обов'язково, і аргументи конструктора доступні у всьому класі, а локальне приватне поле створювати теж не обов'язково. Додатково до цього, в Scala можна відразу оголосити об'єкт без створення класу, використовуючи ключове слово `object`. В результаті реалізується патерн Singleton:

```
object Singleton(field: String) {
  def squareMe(x: Int) = x*x
}
Singleton.squareMe(2)                          // 4
```

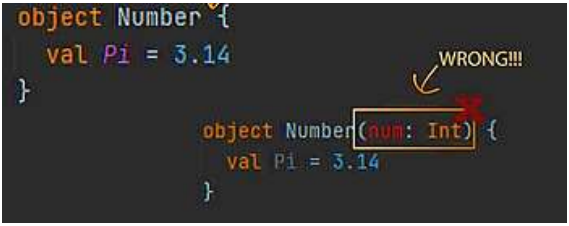
В Scala постаралися дотримуватися чистого ООП – немає даних поза об'єктами. Отже, немає статичних полів. Є класи (конструктори об'єктів) і об'єкти-синглтони. Визначення `val`, `var`, `def`, `lazy val` дозволені лише усередині об'єкту/класу/метода/функції.

Об'єкт – аналог статичного поля (наприклад, зручно застосовувати для створення лічильника кількості екземплярів класу).

Відмінними рисами об'єктів є те, що вони:

– не мають параметрів;

– є одинаками (Singleton Object).



Коли говорять "одинаки" – мають на увазі, що об'єкти існують ***в єдиному екземплярі***. У них свій тип (наприклад, об'єкт `Number`, узятий як приклад, типу `Number`).

Твердження про одинаків може бути легко перевірене:

```
object Number
val numA = Number
val numB = Number
println(numA == numB)     // true
```

`True`, виведене в результаті, свідчить про те, що `numA` і `numB` посилаються на один і той же `Number`.

А ось якщо подібний експеримент провести для класу, то результат буде негативним, оскільки тепер справу матимемо з двома різними екземплярами класу.

```
class Number
val numA = new Number
val numB = new Number
println(numA == numB) // false
```

Компаньйони

Якщо в одному і тому ж файлі і під одним і тим же ім'ям оголосити об'єкт і клас, то у такому разі їх можна назвати *компаньйонами*.

Об'єкт має доступ до всіх полів і методів свого класу-компаньйона (у тому числі і `private`).

```
class MyString (val str:String){       //клас
  private var extra = "extraData"
  def getString: String = str + extra
}

object MyString{                       // об'єкт
  def apply(base: String, extras: String): MyString
   ={
     val s = new MyString(base)
     s.extra = extras
              // для екземпляра класу s
              // міняємо значення private extra
      s
     }
   def apply(base: String) = new MyString(base)

}

println(MyString.apply("hello","world").getString)
                            // helloworld

println(MyString.apply("welcome").getString)
                            //welcomeextraData
```

Якщо оптимізувати код, вийде так:

```
class MyString(val str: String) {
  private var extra = "extraData"
  override def toString: String = str + extra
}
object MyString {
  def apply(base: String, extras: String):
  MyString = {
    val s = new MyString(base)
    s.extra = extras
    s
  }
  def apply(base: String) = new MyString(base)
}
```

Опускаємо найменування методу `apply` і відразу пишемо необхідні для цього методу параметри

```
println(MyString("hello", "world"))
                              // helloworld
println(MyString("welcome"))
                              // welcomeextraData
```

Тут два ключові моменти, обидва пов'язані з *синтаксичним цукром* (детально пізніше):

1) `apply` можна не прописувати, а відразу після імені об'єкту в дужках вказувати параметри `apply` методу;

2) перейменувавши `getString` в `toString`, позбавилися від необхідності взагалі прописувати ім'я методу для його виклику.

Виклик `toString` передбачений в імплементації `print` и `println`.

Додаткова ілюстрація (код той же, але мета – детальніше показати роботу методів, що викликаються):

Фабричний метод (Factory method)

У об'єкта *відсутні параметри, що передаються*.

Але якщо все ж треба якось імплементувати ускладнений інтерфейс для створення об'єкту, а конструктор класу не міняти, то допоможе фабричний метод.

Фабричний метод застосовується:

– коли заздалегідь невідомо, об'єкти яких типів необхідно створювати;

– коли система має бути незалежною від процесу створення нових об'єктів і розширюваною: у неї можна легко вводити нові класи, об'єкти яких система повинна створювати;

– коли створення нових об'єктів необхідно делегувати з базового класу класам спадкоємцям.

```
class Number(val num: Int)

object Number {
  val Pi = 3.14
  // Factory method
  def newNum(x: Number, y: Number): Number =
              new Number(x.num + y.num)
          //Ускладнили процес створення
          //екземпляра класу Number, тепер
          //задіяли кілька представників класу Number
}
val numA = new Number(1)       // екземпляр класу
val numB = new Number(2)       // екземпляр класу
val numC = Number.newNum(numA, numB)   // об'єкт

println(numA.num)                 // 1
println(numB.num)                 // 2
println(numC.num)                 // 3
```

Можна зробити код ще лаконічнішим, якщо перейменувати `newNum` в `apply`, що дозволить опускати ім'я методу `apply` при виклику (це на практиці і робиться):

```
class Number(val num: Int)

object Number {
  val Pi = 3.14
  def apply(x: Number, y: Number): Number =
            new Number(x.num + y.num)
}

val numA = new Number(1)     // екземпляр класу
val numB = new Number(2)     // екземпляр класу
val numC = Number(numA, numB) // об'єкт
```

```
 // застосовується apply,
 // компілятор знає про apply і що з ним робити,
 // на відміну від методу з довільним ім'ям

  println(numA.num) // 1
  println(numB.num) // 2
  println(numC.num) // 3
```

***Приклад* 4**. Можливість передавати різну кількість параметрів.

```
class Number(val n: Int) {
    private var _double: Double = 3.14
    def double(): Double = _double
 }


object Number {
    def apply(n: Int): Number = new Number(n)
    def apply(n: Int, dN: Double): Number = {
        val number = new Number(n)
        number._double = dN
        number
     }
}

// екземпляр класу
  val v1 = new Number(10)
  println(v1.n+" "+v1.double())       // 10 3.14
// об'єкт без apply
  val v2 = Number(20)
  println(v2.n+" "+v2.double())       // 20 3.14
// об'єкт з apply
  val v3 = Number.apply(30, 33.33)
  println(v3.n+" "+v3.double())       // 30 33.33
```

***Завдання до практичної роботи* 2**

2.1 Напишіть клас `Time` з полями `hours` и `minutes`, доступними лише для читання, і методом `before(other: Time): Boolean`, який перевіряє, чи передує час `this` часу `other`.

Об'єкт `Time` повинен конструюватися як `new Time(hrs, min)`, где `hrs` – час в 24-годинному форматі.

Запуск функції:

```
println(time2.before(time1))     // true (або false)
```

2.2 Напишіть клас `Person` з головним конструктором, що приймає рядок, який містить ім'я, пропуск і прізвище, наприклад: `new Person("Fred Smith")`. Зробіть поля `firstName` і `lastName` доступними лише для читання. Реалізуйте в класі методи, які повертають окремо ім'я і окремо прізвище.

2.3 Напишіть клас `Car` з полями, що визначають виробника, назву моделі і рік виробництва, які доступні лише для читання, і поле з реєстраційним номером автомобіля, доступним для читання/запису.

Додайте чотири конструктори. Всі вони повинні приймати назву *виробника* і назву *моделі*. При необхідності у виклику конструктора можуть також вказуватися *рік* і *реєстраційний номер*. Якщо рік не вказаний, він повинен встановлюватися рівним 1, а за відсутності реєстраційного номера повинен встановлюватися порожній рядок.

2.4 Завдання по створенню двох класів – клас `Instructor`(який містить *id, ім'я і прізвище інструктора*) и клас `Course`.

Клас `Instructor: id, name, surname`

Методи класу:

– `fullName`: повертає повне ім'я у вигляді Ім'я Прізвище.

Перші букви імені і прізвища обов'язково мають бути заголовними, а на вхід цілком можуть передаватися дані, що складаються повністю як з великих букв, так і з маленьких.

Клас `Course: courseID, title, releaseYear, instructor`

Методи класу:

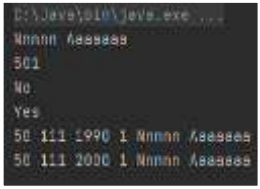
– `getID`: повертає `id` в форматі `courseID + instructor.id` (наприклад, якщо `courseId = 1`, а `instructor.id = 2`, то метод поверне 12);

– `isTaughtBy(instructor)`: перевіряє, чи є вказана людина інструктором курсу;

– `copyCourse(newReleaseYear)`: повертає новий екземпляр класу `Course` з оновленим роком реліза, в якому як значення `releaseYear` використовується `newReleaseYear`;

– `print()`: функція друку всього.

```
// Типи, які мають бути використані
String: name, surname, fullName, title, releaseYear,
getID;
Int: courseId, id;
Instructor: instructor (в конструкторі класу Course),
instructor (у методі isTaughtBy);
Boolean: isTaughtBy;
Course: copyCourse.

// Результати роботи класів мають бути такими
val instr1 = new Instructor(1,"nNNnn","aaAAAaa")
val instr2 = new Instructor(2,"nNNnn","aaAAAaa")
val ob1 = new  Course(50,"111","1990",instr1)
val ob2 = new  Course(50,"111","1990",instr1)
println( instr1.fullName())
println(ob1.getID())
  if (ob2.isTaughtBy(instr2))
       println("Yes")
  else println("No")
  if (ob2.isTaughtBy(instr1))
       println("Yes")
  else println("No")
ob1.print()
val ob3 = ob1.copyCourse("2000")
ob3.print()
```



2.5 Напишіть об'єкт `Conversions` з методами `inchesToCentimeters`, `gallonsToLiters` и `milesToKilometers`, які конвертують значення (дюйм-сантиметр, галон-літр, миля-кілометр).

Значення всіх вхідних (дюйм, галон, миля) змінних дорівнюють 3.5, результати мають бути наступними:

8.89 см

13.24 л

5.63269 км

2.6 Визначите клас `Point` з об'єктом-компаньйоном (з методом `apply`), щоб можна було конструювати екземпляри `Point`, как `Point`(3, 4), без ключового слова `new`.

Перевантажити в об'єкті-компаньйоні метод `apply`, який приймав би на вхід існуючий об'єкт класу `Point` і повертав суму координат.

Контрольні запитання

1. Як оголошується клас та екземпляр класу у мові Scala?
2. Як отримати доступ до поля класу?
3. Як викликаються методи класу?
4. Як використання ключового слова `this` впливає на результат програми?
5. Що таке перевантаження методів класу?
6. Як перевантажити конструктори класу?
7. Що таке об'єкт? Які відмінні риси об'єктів
8. Коли клас і об'єкт можна назвати компаньйонами?
9. Що таке фабричний метод?
10. Коли застосовується фабричний метод?

Практична робота 3

КЛАСИ ЗРАЗКИ В SCALA

Мета: Одержання практичних навичок при використанні класів-зразків та об'єктів у мові Scala

Теми для попереднього опрацьовування

– Вступ до класів

– Конструктори

– Перевантаження методів класу

– Об'єкти, компаньйони

Теоретичні відомості

Однією з основних ідей функціонального програмування є те, що прагнуть розділити структури даних і операції над ними. Тобто всі функції зазвичай поміщаються в трейти і об'єкти, але ніяк не в звичайні класи. Їх взагалі бажано уникати, якщо того дозволяє бізнес-логіка додатка, вдаючись до `case` класів.

Однак існує набір методів, які завжди можуть бути застосовані для будь-якого класу. Саме доступ до таких методів і дають класи зразки, позбавляючи необхідності імплементувати ці методи вручну.

Оголосити клас зразок дуже легко. Треба всього лише дописати ключове слово `case`:

```
case class Person(name: String, occupation: String)
```

Переваги `case` класів:

– іммутабельність (імутабельний (незмінний, `immutable`) клас – це клас, який після ініціалізації *не може змінити свій стан*);

– вивід в консолі, в читабельному вигляді;

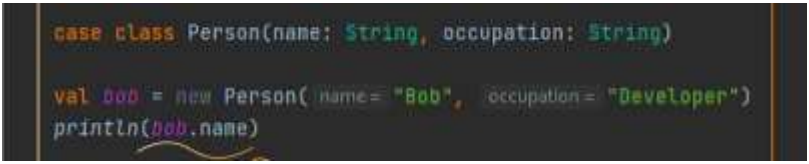
– поелементне порівняння;

– можливість копіювання.

1. Параметри `case` класу за умовчанням є `val` полями

Це означає, що доступ до них отримують автоматично без додаткових маніпуляцій.

Клас-зразок має автоматичний доступ до полів класу:

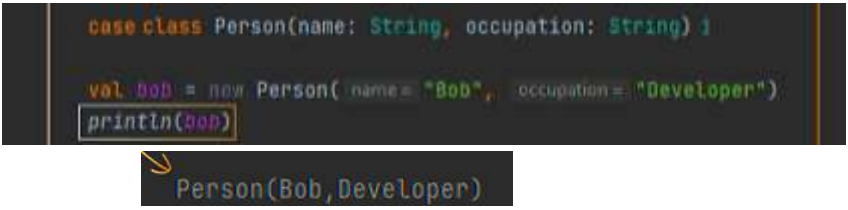


Звичайний клас – без специфікатора `val` доступу не буде:

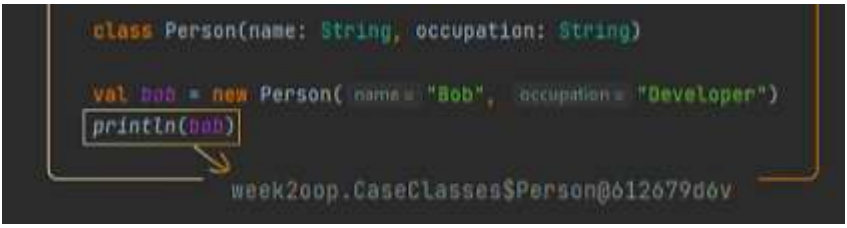
2. Інформація відразу виводиться в зрозумілому вигляді

Метод `toString` (який має свій синтаксичний цукор, тому можна обійтися взагалі без явного прописування імені цього методу в коді) позбавляє від абракадабри, що виводиться в звичайному класі.

Клас-зразок :



Звичайний клас:



3. Доступний метод `equals`

Зазвичай при обговоренні методу `equals( == )` прийнято говорити про `reference level equality` (рівність еталонного рівня) і `content level equality` (рівність на рівні змісту).

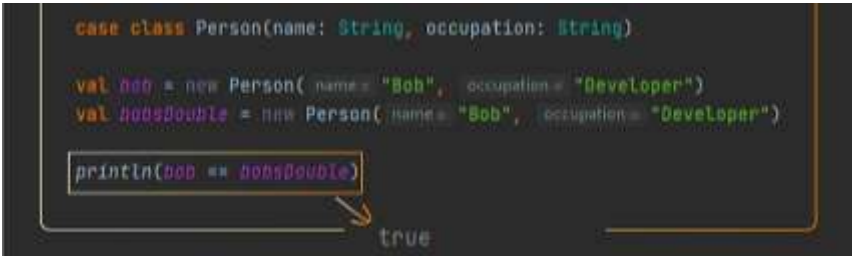
Однією з властивостей `case` класів є *обов'язкова наявність списку параметрів* (нехай навіть порожнього), що пояснює існування прописаного спеціально для них методу `equals`, що представляє `content level equality` і що дозволяє виробляти порівняння по структурі, а не за посиланнями.

Наприклад:

```
case class Person(name: String, age: Int)
val personA = Person("a", 32)
val personB = Person("b", 32)
val personC = Person("a", 32)
println(personA.equals(personB))
                    // false ("a", 32) и ("b", 32)
println(personA.equals(personC))
                    // true
```

Тому, якщо створити двійника Боба (`bobsDouble`), а потім порівняти двійника з оригіналом, то у випадку `case` класу – отримаємо `true`, хоча звичайний клас видасть `false`.

Клас-зразок :



Звичайний клас:


```
case  class Person(name: String, occupation: String)
val bob = new Person("Bob","Developer")
val bob2 = new Person("Bob","Developer")
println(bob == bob2)                    //true
```

4. Доступний метод `copy`

Метод дозволяє як *повністю* скопіювати екземпляр класу, так і скопіювати із *зміненими аргументами* конструктора:

```
case class Person(name: String, occupation: String)
val bob = Person("Bob", "Developer")
                // без new, т.к. будь-який
                // case клас має об'єкт-компаньйон
val anotherBob = bob.copy()
println(bob)       // Person(Bob,Developer)
println(anotherBob)   // Person(Bob,Developer)
println(bob == anotherBob)    // true

val bobsTwin = bob.copy(name = "John")
println(bobsTwin)       // Person(John,Developer)
```

5. Будь-який `case` клас має об'єкт-компаньйон (`class` + `object` = `companion`)

У такому об'єкті-компаньйонові завжди присутній метод `apply`, яким можна скористатися і зробити так:

```
case class Person(name: String, occupation: String)
val alice = Person("Alice", "Engineer")
                          // метод apply в дії
```

Тобто був створений екземпляр класу *без використання ключового слова* `new`.

При створенні `case class` завжди автоматично створюється об'єкт-компаньйон + методи `apply()`, `toString()`, `hashCode`, `equals()`. Якщо необхідно доповнити об'єкт додатковим функціоналом – потрібно прописати об'єкт явно в коді з необхідними доповненнями.

***Приклад* 1.** Коли необхідно застосовувати ключове слово `new` при створенні об'єктів.

1. Використовуйте ключове слово `new`, якщо необхідно послатися на власний конструктор `class`:

```
class Foo { }
val f = new Foo
```

2. Пропустіть `new`, якщо необхідно послатися на метод `apply` супутнього об'єкту:

```
class Foo { }
object Foo {
    def apply() = new Foo
}
// Both of these are legal
val f = new Foo
val f2 = Foo()
```

3. Якщо створено `case` клас:

```
case class Foo()
```

Scala за замовчуванням створює об'єкт-компаньйон, перетворюючи його на такий:

```
class Foo { }
object Foo {
    def apply() = new Foo
}
val f  = Foo()
```

4. Немає правила, яке свідчить, що супутній `apply` має бути проксі-сервером для конструктора:

```
class Foo { }
object Foo {
    def apply() = 7
}
// These do different things
println(new Foo())  // test@5c79cc94
println(Foo())      // 7
                    // об'єкт, схожий на функцію
                    // це вживання методу apply()
```

Завдання до практичної роботи 3

3.1 Написати код класу `Logger`, який:

– дасть користувачу можливість отримати поточну кількість повідомлень (зробити параметр `msgNum` полем класу, задати значення за умовчанням для `msgNum` = 10 конструкторі при створенні класу);

– містить в собі метод `info`, що виводить на екран повідомлення `INFO New Message` і що збільшує число повідомлень в системі на 1;

– містить перевантажений метод `info`, що виводить вказане вище повідомлення `INFO New Message` задане число разів (*використовувати рекурсію*);

–містить метод `print` для отримання поточної кількості повідомлень (`def print: Unit`).

Реалізувати за допомогою:

– компаньйонів (`object`, де реалізовано метод `apply`);
– `case`-класів.

У основній програмі робота з класом `Logger` виглядає так:

```
 val logger = new Logger
logger.print       // 10
logger.info.print         // INFO New Message
                          // 11
logger.info(3).print      // INFO New Message
                          // INFO New Message
                          // INFO New Message
                          // 13
```

Контрольні запитання

1. Що таке класи- зразки?
2. Як для класу-зрзазку отримати доступ до полів класу?
3. Як використовується метод `toString` для класу-зразку?
4. Як для `case`-класу перевантажен метод `equals`?
5. Як для `case`-класу перевантажен метод `copy`?
6. Чому для `case`-класу завжди присутній метод `apply`?

Практична робота 4

НАСЛІДУВАННЯ В SCALA

Мета: Одержання практичних навичок при використанні наслідування у мові Scala

Теми для попереднього опрацьовування

– Перевантаження методів класу

– Об'єкти, компаньйони

– Класи-зразки

Теоретичні відомості

Завдяки наслідуванню – клас може отримати доступ до полів та методів іншого класу (з умовою, що ті не відзначені як `private`). Наслідування відбувається за допомогою ключового слова **`extends`**.

Основні теми цієї глави:

– ключові слова `extends` і `final` діють так само, як в Java;

– при перевизначенні методів необхідно використовувати модифікатор **`override`**;

– конструктор суперкласу можна викликати лише з головного конструктора;

– поля можуть перевизначитися.

Для прикладу було оголошено клас `Person`, у якому було створено метод `greet`. І потім було створено новий клас `Student`, який розширює раніше створений клас `Person`.

```
class Person {
  def greet: String = "Hello"
}
class Student extends Person
val student = new Student
println(student.greet)
```

Як видно, якщо створити екземпляр класу `Student`, можна буде звернутися до методу `greet` класу `Person`, завдяки наслідуванню.

Множинне наслідування в Scala не підтримується! Для цього є **трейти** (*будуть розглянуті далі*). Трейт можна сприймати як заміну абстрактному класу.

`protected` vs `private`

При наслідуванні з'являється доступ до всіх полів і методів батьківського класу, які *не є* `private`.

Ключове слово `private`, написане перед методом та/або змінною, робить цей метод та/або змінну доступним тільки для того класу, в якому він та/або вона були описані:

Ключове слово `protected` працює трохи інакше, роблячи зазначені поля та методи доступними для класу та його підкласу, але недоступними *поза їхніми тілами*.

***Приклад* 1.** Необхідно правильно створити змінну *`pizza`*, щоб виводилося на екран повідомлення `This is pizza`.

```
class Pizza private {
    override def toString = "This is pizza"
  }

object Pizza {
    val pizza = new Pizza
    def getInstance: Pizza = {
      pizza
    }
  }

   val pizza = .....            // пропущений код
   println(pizza)
```

Примітки:

– за допомогою `private` контролюється видимість конструктора класу, тобто дозволяється створення екземпляра класу *лише через методи;*

– `override` використовується, щоб перевизначити стандартний метод `toString` в метод класу `Pizza`;

– для `toString` існує синтаксичний цукор, тому `println(pizza)` насправді `println(pizza.toString)`

```
new Pizza      // не можна створити об'єкт класу,
               // т.к. private конструктор
Pizza          // можна(об'єкт),
               // але не надрукує правильно
Pizza.getInstance     // так вірно!
```

Можна було ще так:

```
val pizza =  Pizza
println(pizza.getInstance.toString)
//  або
println(pizza.getInstance)// синтаксичний цукор
```

Саме тому удається через об'єкт створити екземпляр класу `Pizza` оскільки об'єкт і клас є *компаньйонами*.

Є правила коду, згідно з якими змінні *не можуть* називатися з *великої* літери (тому не повинно виникати ситуації, коли побачивши змінну, написану з великої літери, потрібно перевіряти, чи є такий клас, і чи є такий об'єкт). У документації про об'єкти компаньйони є така примітка – "якщо у класу або об'єкта є компаньйон, вони *повинні бути розміщені в тому самому файлі*".

Використання `private` і `protected`

Це застосовно, як правило, за наявності складних конструкторів.

Нехай є клас:

```
class Student(email: String, name: String)
```

Цілком логічно, що студент може вказати будь-який `email` і `name`.

Можна доповнити:

```
class Student(email: String, name: String, country: String)
```

Тоді можна вказати будь-яке значення і для `country`, неіснуючу країну теж. А це повинно бути недопустимо. І мета – надати функціонал, що дозволяє створювати `Student` без явної вказівки `country`. Для цього і буде потрібно `private` або `protected` для конструктора.

`Private:`

– забороняє створення класу за допомогою ключового слова **`new`** і вимушує використовувати інші методи. Наприклад, `apply`

```
class Student private(email: String, name: String, country: String)
```

Тоді буде потрібно `object`:

```
class Student private(email: String, name: String, country: String)
```

```
object Student {
  def createRuStudent(email: String, name: String): Student = new Student(email, name, "usa")
}

val student = Student.createRuStudent("jd@ex.com", "John")
```

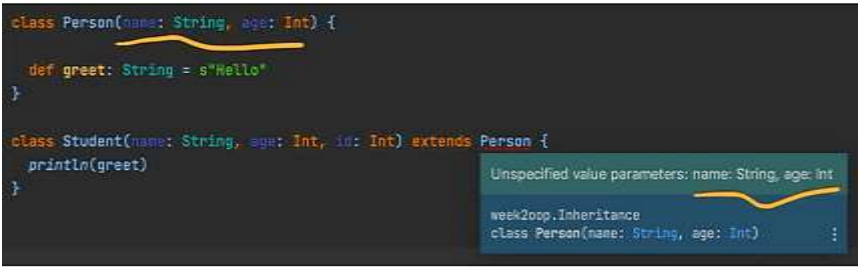
Protected:

– альтернатива `private`, допускає `extend` і робить `new` доступним для використання:

```
class Student protected(email : String, name : String, country: String)

class StudentUsa(email: String, name: String) extends Student(email, name, "usa")

val student = new StudentUsa("jd@ex.com", "John")
```

Ключове слово extend для класу с параметрами

Тепер можна побачити, як працюють конструктори. У наступному коді є помилка:



Причиною *помилки* є те, що при створенні екземпляра класу `Student` – спочатку потрібно буде зробити виклик конструктора з батьківського класу.

Необхідно вказати пропущені параметри:

```
class Person(name: String, age: Int) {

  def greet: String = s"Hello"
}

class Student(name: String, age: Int, id: Int) extends Person(name, age) {
  println(greet)
```

Або можна використовувати *допоміжний конструктор* `def this`. І тепер компілятор знаходитиме конструктор класу `Person`, який не вимагає жодних параметрів, і все працюватиме:

```
class Person(name: String, age: Int) {
  def greet: String = s"Hello"

  def this() = this("UnknownPerson", 0)
}

class Student(name: String, age: Int, id: Int) extends Person {
  println(greet)
}
```

Використовуємо конструктор без параметрів

Перевизначення (`override`)

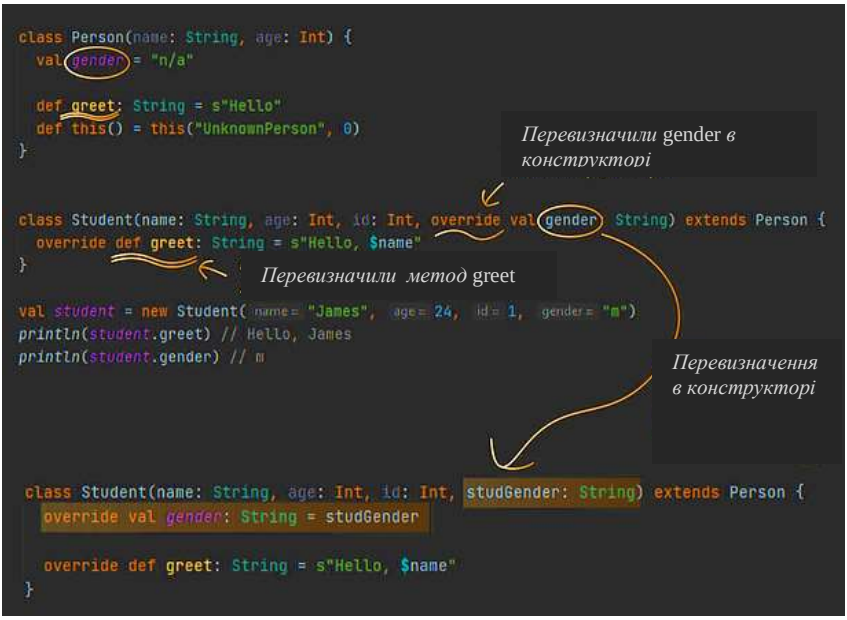
Застосовується, коли для *підкласу* потрібно застосувати метод з *батьківського* класу, але з імплементацією, відмінною від початкової.

Або коли необхідно *змінити значення змінної батьківського класу*.

Для перевизначення методу або змінної використовуємо ключове слово `override`. Наприклад:

```
public class Person {
    ...
    override def toString =
       s"${getClass.getName}[name=$name]"
}
```

Причому поля, на відміну від методів, можуть бути перевизначені в конструкторі класу:



Значення змінної `val gender` буде змінено лише для екземплярів класу `Student`, якщо працюватимемо з екземплярами класу `Person`, значення буде "`n/a`"

```
val mark: Person = new Person("Mark", 24)
println(mark.gender)                    // n/a
```

`Super` – виклик методу супер-класу

Корисно, коли хочемо скористатися імплементацією методу або значенням змінної з батьківського класу. І в той же час додати до цього щось своє. Для цього всього-лише треба застосувати ключове слово `super`. Наприклад, інструкція `super.toString` викличе метод `toString` суперкласу, тобто `Person.toString`:

```
public class Person {
    ...
    override def toString =
       s"${getClass.getName}[name=$name]"
}
```

```
public class Employee extends Person {
   ...
   override def toString =
      s"${super.toString}[salary=$salary]"
}
```

Поліморфізм (polymorphism)

Поліморфізм має на увазі *заміну типів*. Наприклад:

```
class Person(name: String, age: Int){
 val gender = "n/a"
 def greet:String=s"Hello"
 def this()=this("UnknownPerson",0)
}

class Student (name:String, age:Int,id:Int,
         studGender:String)extends Person {
override val gender = studGender
override  def greet:String =
     s"${super.greet}, $name"
}
```

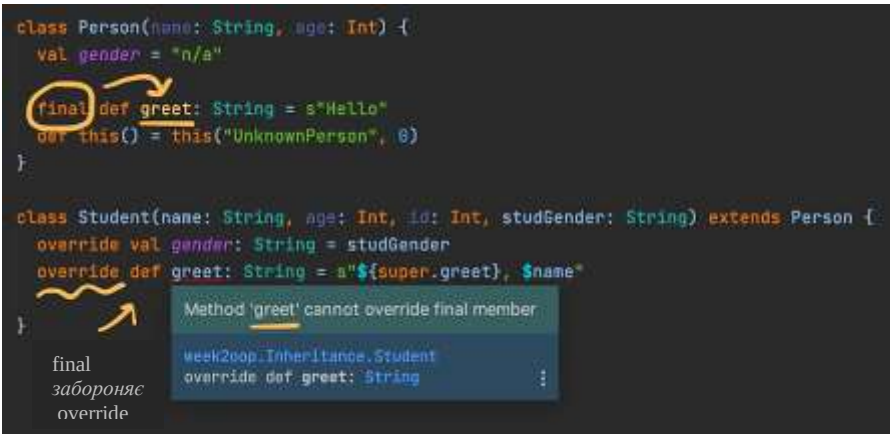
```
val student = new Student("James",24,1,"m")
println(student.greet)          // Hello, James
println(student.gender)         // m
val aPerson:Person = new Student("Alice",24,2,"f")
println(aPerson.greet)          // Hello, Alice
```

Поле `gender` у `aPerson` буде `f` як в переданих аргументах (а не `n/a` як при звичайному створенні екземпляра `Person`), оскільки для `aPerson` будуть задіяні перевизначені методи і значення з класу `Student`. Тобто, якщо йде наслідування від `Person`, то можна створювати для `val aPerson:Person` новий екземпляр `Student`.

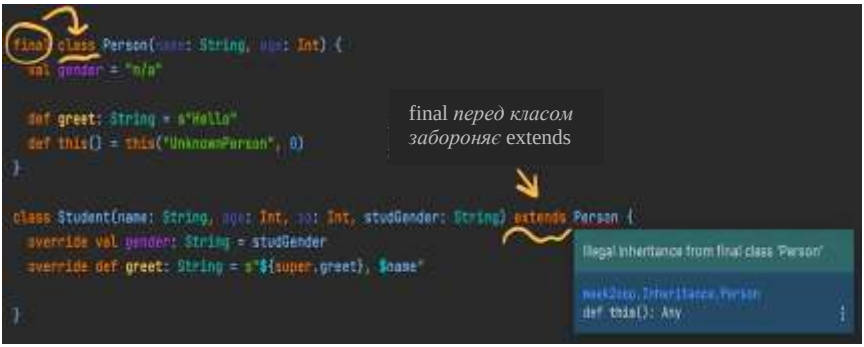
Захист від перевизначення

Є декілька способів:

1) використання ключового слова `final` перед членами класу, для яких потрібно заборонити `override`:



2) використання ключового слова `final` для цілого класу (що взагалі заборонить `extends` цього класу):



3) використання ключового слова `sealed` для класу (це м'якша версія `final`, тому допускається `extends` у поточному файлі, але забороняється поза цим файлом)

`sealed` використовується, коли треба гарантувати існування підкласів, визначуваних лише в поточному файлі.

Приклад 2. Необхідно описати дні тижня. Використання `sealed` гарантує, що існувати будуть лише ті дні, що конкретизовані в поточному файлі, і ніде раптом випадково не спливе невідомий день тижня.

Те, що будете відомо, які дні доступні – створює сприятливі умови для вживання шаблонів, якщо їх буде потрібно.

Примітка: ні шаблони, ні абстрактні класи ще не вивчалися.

```
sealed abstract class DayOfTheWeek(val name:String,
                    val isWeekend: Boolean)
case object Monday extends
                DayOfTheWeek("Monday", false)
case object Tuesday extends
                DayOfTheWeek("Tuesday",  false)
case object Wednesday extends
                DayOfTheWeek("Wednesday", false)
case object Thursday extends
                DayOfTheWeek("Thursday", false)
case object Friday extends
                DayOfTheWeek("Friday", false)
case object Saturday extends
                DayOfTheWeek("Saturday", true)
case object Sunday extends
                DayOfTheWeek("Sunday", true)
```

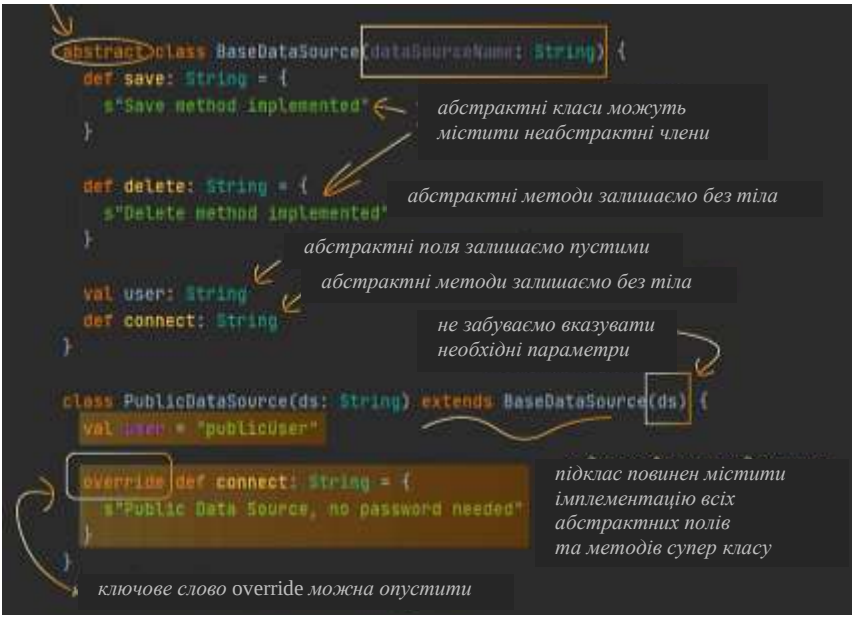
Приклад дослідження типів:

```
val mon = Monday
println(mon.getClass.getSimpleName)    // Monday
```

Абстрактні класи (abstract classes)

Бувають ситуації, коли необхідно в класі задати лише поле або метод, а імплементацію прописувати в підкласах, підлаштовуючи її під кожен конкретний випадок.

У абстрактному класі абстрактні поля залишаються *порожніми*, а абстрактні методи *без тіла*. Це означає, що *не можна створити екземпляр абстрактного класу*, поки абстрактні методи і поля не прописані належним чином.



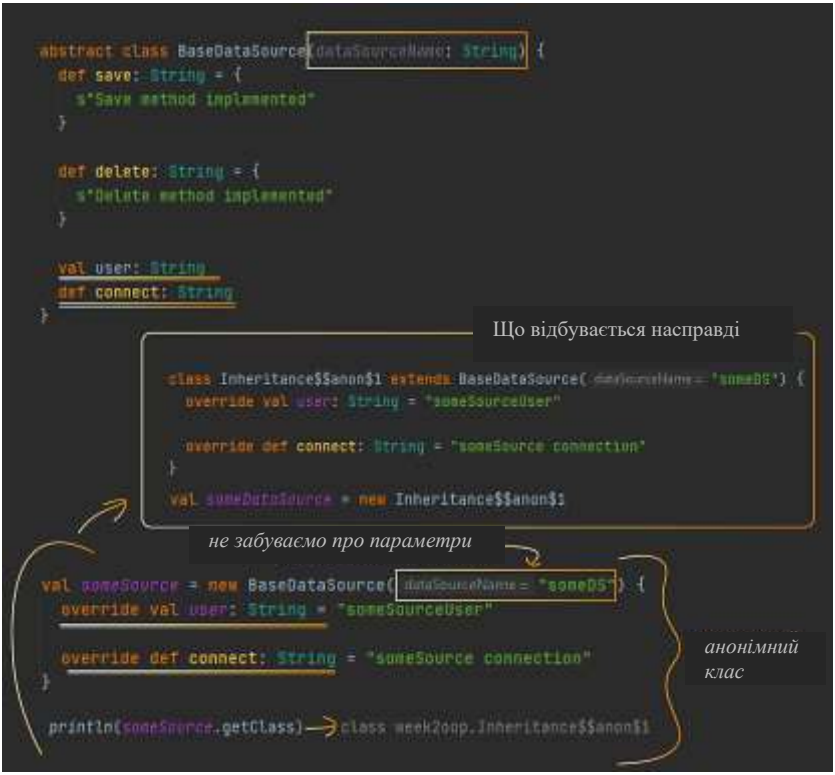
Параметр `ds` передаємо в частину `class PublicDataSource(ds: String) extends BaseDataSource(ds)`, тому що у батьківського класу `BaseDataSource` конструктор з одним параметром `String`. Параметр `ds` визначений у конструкторі класу `PublicDataSource(ds:String)`.

Обов'язково потрібно вказувати тип, який повертається методом абстрактного класу. Інакше компілятор сам додумає тип (зазвичай, `Unit`).

Анонімні класи (Anonymous classes)

Не можна створити екземпляр абстрактного класу, поки абстрактні методи і поля не прописані належним чином. Якщо створити змінну абстрактного класу, де прописуємо реалізацію анонімних змінних і методів, уникнувши створення підкласу, то отримуємо *анонімний підклас*:

```
val someSource = new BaseDataSource("someDS") {
    override val user: String = "someSourceUser"
    override def connect: String =
                "someSource connection"
}
```



Для чого використовуються анонімні класи

Щоб скоротити код. Наприклад, іноді екземпляр класу потрібний як аргумент, але відомо, що більше ніде в коді імплементація цього класу не потрібно – тоді зручніше прописати анонімний клас, ніж створювати окремий.

Просто при швидкому скануванні чужого коду буває зручнішим побачити анонімний клас, зрозуміти, що це одиничний випадок його використання і забути про нього, чим зустріти повністю прописаний клас і стежити, де він далі в коді буде.

Приклад 3. Клас хоч би з одним абстрактним методом або абстрактним полем має бути оголошений абстрактним.

```
abstract class Person {
  val id: Int              // Абстрактне поле
  var name: String         // Абстрактне поле
  def get_id:Int           // Абстрактне метод
}
```

Цей клас визначає абстрактні методи читання для полів `id` і `name` і абстрактний метод `get_id`. Кожен конкретний підклас класу `Person` повинен визначити реалізацію методу `get_id` і реалізувати конкретні поля. При перевизначенні абстрактного методу в підкласі не потрібно використовувати ключове слово `override`.

```
class Employee(val id:Int) extends Person {
  var name = "Hello"
  def get_id = id //  override не потрібно
}
```

Абстрактне поле завжди можна визначати без вказівки типу.

Якщо створити змінну абстрактного класу, де прописуємо реалізацію анонімних змінних і методів, уникнувши створення підкласу, то отримуємо *анонімний підклас*:

```
val v1 = new Person {     // Анонімний підклас
  val id = 1000           // ініціалізація id
  var name = "Hi"         // ініціалізація name
  def get_id =id + 1000
                      // реалізація def  get_id
}
println(v1.id+" "+v1.name+" "+v1.get_id)
                      // 1000 Hi 2000
```

Якщо створити змінну підкласу `Employee`:

```
val v2 = new Employee(3000)
println(v2.id+" "+v2.name+" "+v2.get_id)
                    // 3000 Hello 3000
```

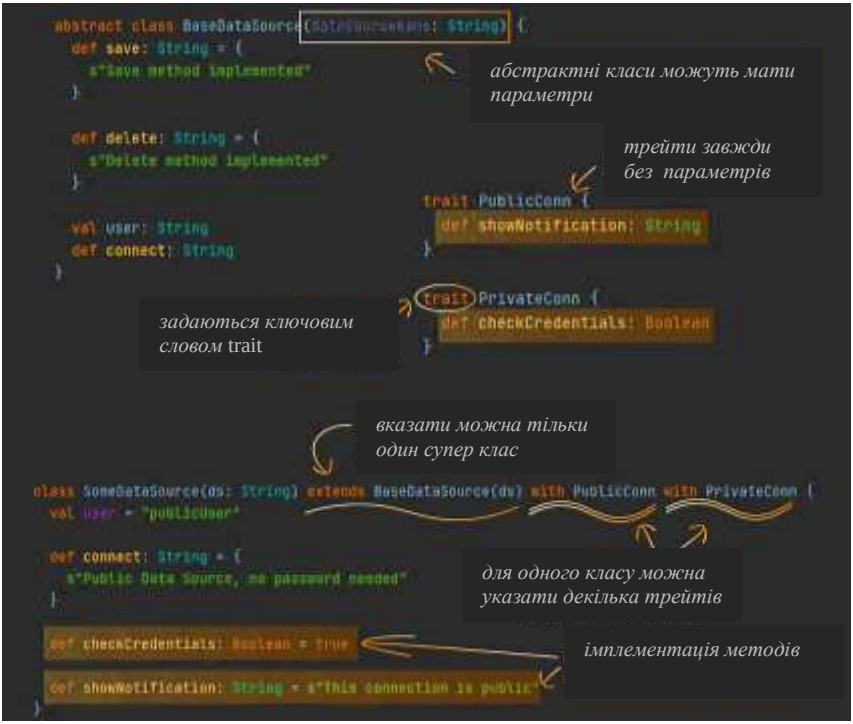
Трейти (traits)

Дуже схожі на абстрактні класи. Як і абстрактні класи, трейти можуть включати неабстрактних членів. Але все таки трейти – це не абстрактні класи. І причин цьому декілька:

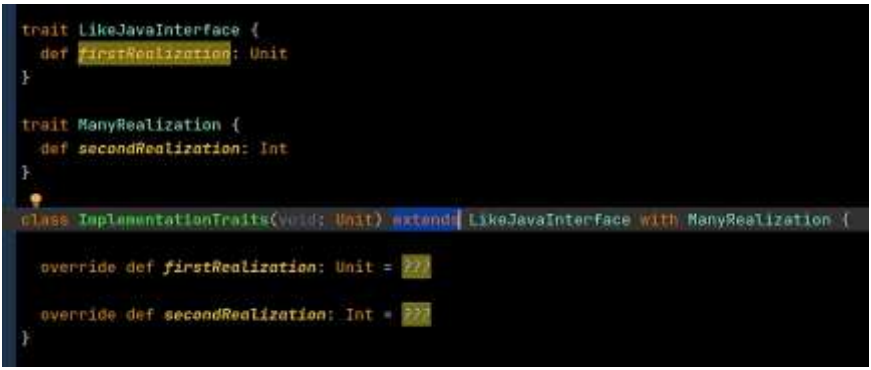
– трейти не можуть задаватися з параметрами;

– можна вказати декілька трейтів для одного класу;

– трейти описують конкретну поведінку для конкретної ситуації.



При реалізації трейтів, перед першим трейтом (за відсутності наслідування від класів) також йде ключове слово `extends`:



Наприклад, є абстрактний клас, у ньому описані `n` обов'язкових параметрів. Є допустимо п'ять класів, що наслідуються від абстрактного, але у кожного по `n+1` параметру, причому кожен "зайвий" параметр унікальний. Щоб не реалізовувати в абстрактному класі всі п'ять параметрів, і не ставити на них заглушки – реалізуємо їх через трейти.

Приклад 4. Створити власний простий приклад ієрархії трейтів, що демонструє багаторівневі трейти, конкретні і абстрактні методи, а також конкретні і абстрактні поля.

```
// Трейт "Тварини"
trait Animal {
 var IQ: Int  // рівень інтелекту - абстрактне поле
 def says: String            // абстрактний метод
}
trait Mammals extends Animal {
 val warmBloded = true
           // ознака теплокровності, конкретне поле
}
trait Dogs extends Mammals {
 val bites = true        // конкретне поле
 override def says: String = "bow-wow!"
                          // конкретний метод
}
```

```
trait Horse extends Mammals {
 val kick = true          // конкретне поле
 override def says: String = "neigh!"
                          //конкретний метод
}
trait Dogtaur extends Horse with Dogs  {
                      // багаторівневий трейт
 val strange = true
}

class Henry extends Dogtaur { // анонімний клас
   var IQ = 100
   def stinks = true
 }
 val henry = new Henry
 assert(henry.strange)
              // догатавр вийшов дивною істотою
 println(henry.strange)   // true
 println(henry.says)      // який гавкає
                          // bow-wow!
 assert(henry.kick)       // брикає
 println(henry.kick)      // true
 assert(henry.bites)      // кусає
 println(henry.bites)     // true
 assert(henry.stinks)     // і страшно смердить
 println(henry.stinks)    // true
```

***Завдання до практичної роботи* 4**

4.1 Реалізувати три класи: `Button`, `RoundedButton` і `TestButton`.

Для класу `Button`:

– передбачити можливість вказівки `label` типа `String` при створенні екземпляра класу;

– прописати метод `def click(): String`, що повертає рядок, в якому вказаний `label`, заданий при створенні кнопки: *`button -label- has been clicked.`*

Для класу `RoundedButton`:

– передбачити наслідування от `Button`;

– реалізувати метод `click`, який схожий на батьківський метод `click`, але на відміну від нього містить слово `rounded` перед `button`: *`rounded button -label- has been clicked.`*

Для класу `TestButton`:

– реалізувати наслідування у вигляді `class TestButton extends Button`;

– метод `click` завжди виводить *`test button -test- has been clicked.`*

Примітка: використовувати `super` в коді.

Результат виконання має бути наступний:

```
val x = new Button("save")
println(x.click())
   //button -save- has been clicked

val y = new RoundedButton("save")
println(y.click())
  //rounded button -save- has been clicked
val z = new TestButton
println(z.click())
  //test button -test- has been clicked
```

4.2 У системі задані класи Event и Listener і створена змінна notification. Для цього використаний наступний код:

```
abstract class Event {
  def trigger(eventName: String): Unit
}
```

```
class Listener(val eventName: String,
                          var event: Event) {
   def register(evt: Event) { event = evt }
   def sendNotification() {
     event.trigger(eventName)
   }
}
val notification: Listener =
          new Listener("mousedown", null)
```

Необхідно, використавши знання анонімних класів, доповнити код так, щоб спрацював виклик:

```
notification.sendNotification
```

В результаті (для випадку, коли в якості `eventName` задано `mousedown`) повинен з'явитися рядок: *`trigger mousedown event`*

Примітка: для вирішення потрібно лише викликати `notification.register` з необхідним параметром. Для цього потрібно використовувати анонімний клас (наприклад, з використанням *об'єкту* `aclass` или *без объекта, тобто як параметр функції* `register`), у якому потрібно перевантажити функцію `trigger` .

Результат виконання має бути наступний:

```
notification.register(aclass)
notification.sendNotification
                    //trigger mousedown event
```

4.3 Визначте клас `CheckingAccount`, який наслідує клас `BankAccount`, який стягує $1 комісійних за кожну операцію поповнення або списання.

```
class BankAccount(initialBalance: Double) {
 private var balance = initialBalance
 def currentBalance = balance
 def deposit(amount: Double) = {
             balance += amount; balance }
 def withdraw(amount: Double) = {
             balance -= amount; balance }
}
```

4.4 Визначте клас `SavingsAccount`, який наслідує клас `BankAccount` з попередньої вправи, який нараховує відсотки кожного місяця (викликом методу `earnMonthlyInterest`) і дозволяє безкоштовно виконувати три операції зарахування або списання кожного місяця. Метод `earnMonthlyInterest` повинен скидати лічильник транзакцій.

4.5 Спроектуйте клас точки `Point`, значення координат `x` і `y` якої передаються конструктору. Реалізуйте підклас точки з підписом `LabeledPoint`, конструктор якого прийматиме рядок з підписом і значення координат `x` и `y`, наприклад:

```
new LabeledPoint("Black Thursday", 1929, 230.07)
```

Реалізувати функцію друку всіх заданих значень.

4.6 Використовувати напрацювання Завдання 4.5. Визначити абстрактний клас геометричної фігури `Shape` з абстрактним методом `def centerPoint:Point` (який розраховує центр фігури) і підкласи прямокутника і кола, `Rectangle` і `Circle`. Реалізувати відповідні конструктори в підкласах і перевизначити метод `centerPoint` у кожному підкласі.

Підказка:

– використовуйте функцію знаходження центру для `Rectangle`:

```
def centerPoint: Point = {
     Point( (lowerLeft.x + width)/2,
            (lowerLeft.y + height)/2 ),
```

де `lowerLeft: Point` – координати лівого верхнього кута;

– для `Circle` – при створенні класу – задавайте *центр кола* і *радіус* в конструкторі і просто виводьте координати центру.

Результат виконання має бути наступний:

```
var rectangle = new Rectangle(
            Point(1.0, 2.5), 4.0, 3.0)
println(rectangle.centerPoint.x)       //2.5
println(rectangle.centerPoint.y)       //2.75
```

```
var circle = new Circle(
           Point(3.5, 5.0), 2.25)
 println(circle.centerPoint.x)     //3.5
 println(circle.centerPoint.y)     //5.0
```

Контрольні запитання

1. За допомогою якого ключового слова відбувається наслідування в Scala?

2. Який модифікатор необхідно використовувати при перевизначенні методів?

3. Чим відрізняється використання специфікаторів `private` і `protected`?

4. Як працюють конструктори за умовчанням при наслідуванні?

5. Як працюють конструктори з параметрами при наслідуванні?

6. Як за допомогою допоміжного конструктору `def this` об'єднати використання наслідування за умовчанням та з параметрами?

7. Коли застосовується перевизначення методів класу?

8. Що таке `Super` -клас?

9. Як зробити виклик методу `Super`-класу?

10. Що таке поліморфізм?

11. Як виконати захист від перевизначення методів класу?

12. Чим відрізняється ключове слово `sealed` від `final`?

13. Що таке абстрактні класи?

14. Що таке анонімні класи? Для чого використовуються анонімні класи?

15. Що таке трейти?

Практична робота 5

ВИКЛЮЧЕННЯ. УЗАГАЛЬНЕННЯ. СИНТАКСИЧНИЙ ЦУКОР В SCALA

Мета: Одержання практичних навичок при роботі з помилками, узагальненими класами у мові Scala

Теми для попереднього опрацьовування

- Перевантаження методів класу
- Об'єкти, компаньйони
- Класи зразки
- Наслідування

Теоретичні відомості

1. Виключення

Помилки (Errors) vs Виключення (Exceptions)

Помилки

Вказують на те, що щось не так з системою:

– StackOverflowError (з нею мали справу в темі рекурсій) помилка пов'язана зі `stack` пам'яттю, яка задіюється при виклику методу (під кожен виклик методу в стеку створюється новий блок, що зберігає інформацію про параметри, змінні методу);

– OutOfMemoryError свідчить, що вичерпали доступну пам'ять. Наприклад, якщо в масиві раптом стало надто багато елементів. Тільки цього разу помилка пов'язана з `heap` пам'яттю, яка використовується для виділення пам'яті під об'єкти.

Виключення

Дають знати, що щось не так з програмою:

– `NullPointerException` – виникає, якщо намагаємося отримати доступ до чогось, чого немає:

```
val x: String = null
println(x.length)      // NullPointerException
```

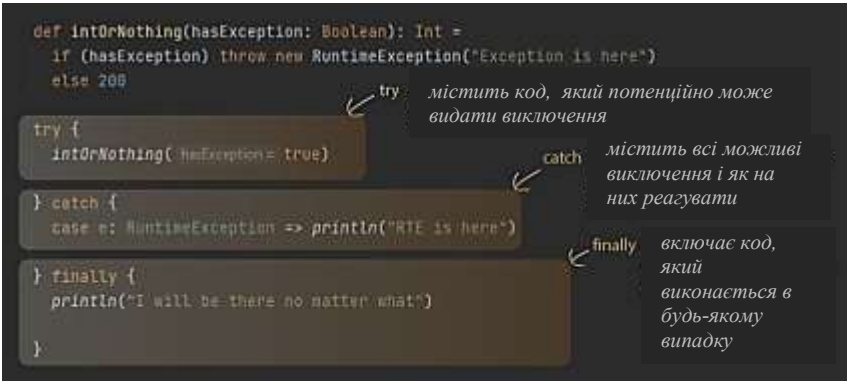
– `RuntimeException` – виникає, коли необхідно вказати на якісь логічні помилки програми.

Якщо в коді необхідно кинути помилку (тобто потрібно штучно видати виключення) – досить використовувати ключове слово `throw`.

Наприклад: є програма, при запуску якої користувач повинен задати свої аргументи, але певні значення не повинні прийматися. Тоді при виявленні таких значень має сенс кинути виключення і зупинити тим самим виконання програми:

```
throw new NullPointerException
```

За допомогою `try-catch-finally` можна зловити виключення. `finally` не повинен містити в собі нічого, окрім побічних ефектів.



Щоб створити своє власне виключення, потрібно пригадати тему наслідування класів. І створити свій клас, який `extends` клас `Exception`. Далі створюється екземпляр класу, який при потребі можна кидати (`throw`) де завгодно:

```
class MyException extends Exception

val exception = new MyException
throw exception
```

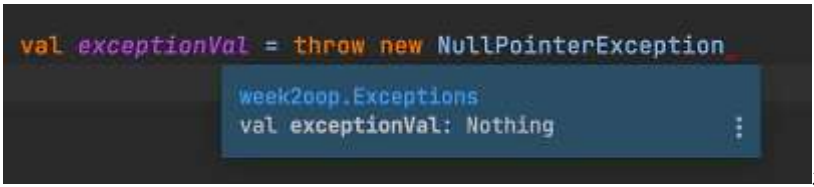
Тип `Nothing`

Все в Scala є виразом. Тому якщо написати так:

```
 val exceptionVal = throw new
NullPointerException
```

а потім спробувати взнати тип змінної, то буде надруковано `Nothing`

```
 println(exceptionVal.getClass.getName)
```



5

`Nothing` означає, що нічого не немає – порожнечу. Тому можна як тип змінної вказати `Int` або `String`:

```
 val exceptionVal: Int = throw new
NullPointerException
 val exceptionVal: String = throw new
NullPointerException
```

Тип `AnyVal`

Нехай є код:

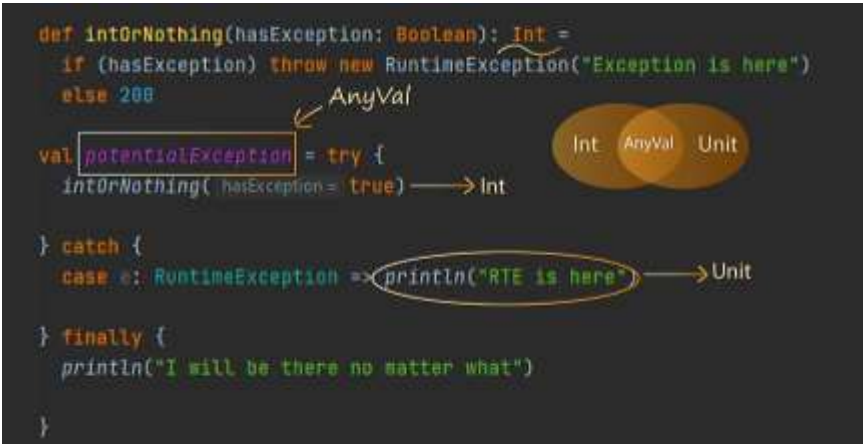
```
def intOrNothing (hasException:Boolean):Int =
  if (hasException)
    throw new RuntimeException ("Exception is here")
  else 2000

 val potentialException = try {
    intOrNothing(false)
 } catch {
  case e: RuntimeException => println("RTE is here")
  } finally {
    println("I will be there no matter what")
}
```

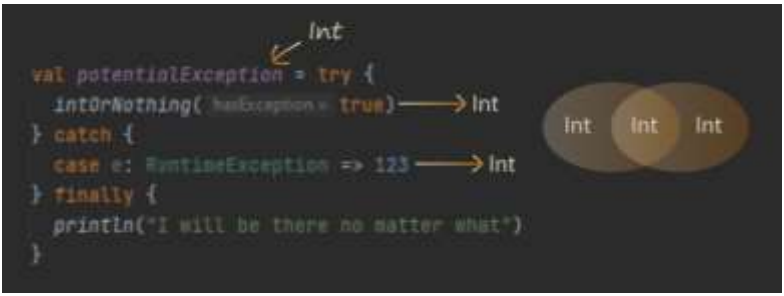
Результат виконання коду: `I will be there no matter what`

Змінна `potentialException` на цей раз матиме типа `AnyVal`:

Це пояснюється тим, що необхідно догодити і тому, що буде повернено у блоці `try`, і тому, що повернеться в `catch` (`finally` не враховується). Тобто в нашому прикладі необхідно використовувати універсального типа, який дозволив би змінною зберігати в собі як значення типу `Int`, так і `Unit`. І цим типом якраз буде `AnyVal` (можна запам'ятати як "готовий до будь-якого значення").

У наступному прикладі – і `try`, і `catch` – обидва повертають `Int`, тим самим диктуючи `potentialException` тип `Int`.





2. Узагальнення (Generics)

Допустимо, є програма, яка розраховує суму наявних накопичень. Програма чудово працює з числами типу `Int`. Але, наприклад, необхідно збільшити точність розрахунків і використовувати тип `Double` або `Float`.

Для цього використовують узагальнення.

Узагальнення – це можливість використання одного і того ж коду, але для різних типів даних. Тобто. змусити `метод` приймати на вхід будь-які типи (цілі числа, рядки тощо). Тоді створюється не просто клас, а клас з узагальненнями. Узагальнення вказують на те, що тип змінних, з якими працюватимуть методи класу, стане відомий лише у момент ініціалізації.

Для завдання узагальненого класу після імені класу в дужках задається параметр типа:

```
class SomeClass[T]
```

Тоді замість конкретного типу, скрізь прописується T:

```
class SomeClass[T] {
  def someFunc(aVal: T): Unit =
      println(s"Generic type is used.
         Received value $aVal of type
         ${aVal.getClass.getSimpleName}")
}
```

Конкретний тип задаватиметься у момент створення екземпляра класу:

```
val anInstance = new SomeClass[Int]
anInstance.someFunc(1)
//Generic type is used. Received value 1 of type Integer

  val anotherInstance = new SomeClass[String]
  anotherInstance.someFunc("some string")
//Generic type is used. Received value some string of
// type String

  val doubleInstance = new SomeClass[Double]
  doubleInstance.someFunc(2.0)
//Generic type is used. Received value 2.0 of type
// Double
```

Розглянемо приклад із списками. Допустимо, відомо, що доведеться зберігати дані у вигляді списку. Ось тільки тип цих даних доки невідомий. А це – відмінний привід скористатися узагальненнями. Тоді клас для роботи із списками створюватимемо ось так:

```
class MyList[A]
```

`[A]` – означає тип даних, які можуть зберігатись у списку. Цей тип буде конкретизовано безпосередньо у момент створення списку:

```
val listOfStrings = new MyList[String]
val listOfDoubles = new MyList[Double]
val listOfInts = new MyList[Int]
```

Примітка:

– за угодою в дужках [ ] зазвичай вказуються заголовні букви типу `A`, `B`, `C`, ... (але узагальнення можна називати не лише однією буквою, наприклад `Map[Key, Value]`);

– узагальнення застосовуються до класів (`class`), трейтів (`trait`) але *не до об'єктів* (object).

Узагальнені методи

Припустимо, є метод, який рандомно вибирає число з цілого списку:

```
def randomInt(items: List[Int]): Int = {
  val randomIndex =
     scala.util.Random.nextInt(items.length)
  items(randomIndex)
}
println(randomInt(List(1, 2, 3, 4, 5)))
```

Якщо необхідно на вхід подавати списки будь-якого типу, тоді має сенс створити узагальнений метод, переписавши функцію ось так:

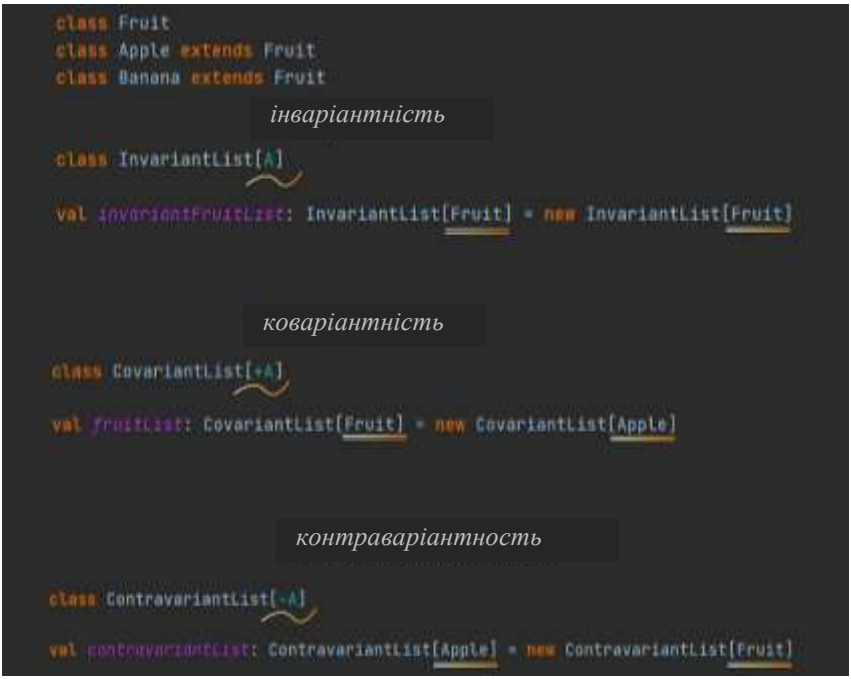
```
def randomItem[A](items: List[A]): A = {
  val randomIndex =
     scala.util.Random.nextInt(items.length)
  items(randomIndex)
}

println(randomItem(List("a", "bc", "def")))
println(randomItem(List(1.5, 2.75, 3.8)))
```

Варіативність (Variance problem)

Нехай є ієрархія класів:

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit
```

Якщо `Apple` розширює клас `Fruit`, то чи можна подібне стверджувати стосовно списків `List[Apple]` і `List[Fruit]`? Відповідей на дане питання цілих три. І зв'язані вони з коваріантністю, інваріантністю і контраваріантностью.



Коваріантність і контраваріантність використовуються для забезпечення безпеки типів.

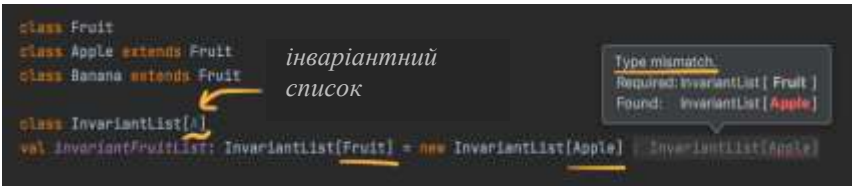
Інваріантність (Invariance)

Інваріантність диктує, що `List[Apple]` и `List[Fruit]` – це абсолютно різні речі, не зв'язані жодними родинними зв'язками.

Тип, вказаний в дужках без яких-небудь додаткових знаків, свідчить про інваріантність:

```
class InvariantList[A]
```

Тому наступний код видасть помилку (в цьому випадку беремо фрукти і лише фрукти, жодні яблука і банани ми із-за інваріантності не визнаємо):



Тобто тип, вказаний зліва, повинен збігатися з типом в правій частині:

```
class InvariantList[A]
val invariantFruitList: InvariantList[Fruit] =
new InvariantList[Fruit]
```

Контраваріантность (Contravariance)

Контраваріантность передбачає знак мінус поряд з типом:

```
class ContravariantList[-A]
```

І тоді створюємо ось таку змінну:

```
val contravariantList:
ContravariantList[Apple] =
 new ContravariantList[Fruit]   // фрукти, а не
                                // лише яблука
```

Приклад з фруктами не дуже інтуїтивний:

```
class Person[-A]
val person: Person[ScalaDeveloper] =
         new Person[Developer]
                          // Developer, а не
                          // лише Scaladeveloper
```

Уявіть ситуацію: ви шукали Scala-розробника до себе в команду, а найняли відмінного розробника, знання якого не обмежені лише мовою Scala. Адже ви від цього тільки виграєте: у працівника більше знань, більше умінь, а значить – більше завдань, які він зможе успішно виконати.

Обмежені типи (Bounded types)

Можна обмежити типи, що підходять для узагальненого класу, вказавши використовувати або тільки підтипи, або тільки супер типи.

Верхнє обмеження типу T <: A

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

class Food[T <: Fruit](fruit: T)
val food = new Food(new Banana)
```

Таким чином – ми говоримо, що до класу `Food` підходять лише параметри типу T, причому цей тип має бути підтипом `Fruit`. Оскільки `Banana extends Fruit`, то `Banana` відмінно підійде.

Оскільки жодних зв'язків між `Pizza` і `Fruit` не задано, то такий код – невірний:

```
class Food[T <: Fruit](fruit: T)
val food = new Food(new Banana)

class Pizza
val moreFood = new Food(new Pizza)
```

І хоча заздалегідь жодного повідомлення про помилку не буде, код не запуститься. Буде видано повідомлення *`inferred type arguments do not conform to value <local Food>'s type parameter bounds...`*

Нижнє обмеження типу T >: A

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit

class Food[T >: Fruit](fruit: T)
```

Тепер як тип T підходять тільки супертипи типу `Fruit` (*тобто батьки класу `Fruit` – якийсь клас "рослинна їжа", наприклад, за умови що `Fruit` успадковується від нього*).

Коваріантність (Covariance)

Коваріантність має на увазі, що раз `Apple` успадковується від `Fruit`, то і `List[Apple]` можна розглядати як нащадка `List[Fruit]`.

Для позначення коваріантного списку – треба додати плюс перед типом:

```
class CovariantList[+A]
```

Тоді можна сказати, що раз справедливий ось такий код:

```
class Fruit
class Apple extends Fruit
class Banana extends Fruit
  val fruit: Fruit = new Apple
```

то буде працювати і ось такий:

```
val fruitList: CovariantList[Fruit] = new CovariantList[Apple]
```

Але тоді питання: чи зможемо ми додати `Banana` в наш список, зробивши ось так:

```
fruitList.add(new Banana)
```

Адже хоча у нас і список `Fruit`, але насправді він створювався для одних яблук (`new CovariantList[Apple]`) і що буде, якщо додати банан? Буде все вірно. Просто із списку яблук – список перетвориться на те, чим і мав бути спочатку, в список фруктів.

Напишемо імплементацію методу `add`, щоб точно бачити – які типи, як і де вказувати.

```
class List[+A] {
  def add[B >: A](element: B): List[B] = ???
}
```

Цим кодом ми говоримо, що якщо в список типа `A` буде доданий елемент типа `B`, то список типа `A` перетвориться на список типа `B`, причому `B` є супер типом для `A`.

3. Синтаксичний цукор (Syntactic Sugar)

Синтаксичний цукор – це альтернативний спосіб написання коду, який більше нагадує природну мову. Тому може бути приємніший людському оку.

Інфіксна нотація (Infix notation)

Підходить лише для методів *з одним параметром*.

Створимо клас `Person`, у якому пропишемо метод `worksAs`:

```
class Person(val name: String,
             occupation: String) {
  def worksAs(jobName: String): Boolean =
            jobName == occupation
}
```

Далі можна створити екземпляр класу `Person`:

```
val bob = new Person("Bob", "Developer")
```

Якщо потрібно для `bob` викликати метод `worksAs`, то можна це зробити звичним способом – через *точкову* нотацію:

```
println(bob.worksAs("Developer"))
                              // точкова нотація
```

Але оскільки `worksAs` приймає на вхід всього один аргумент, то виклик цього методу можна переписати в *інфіксній* нотації:

```
println(bob worksAs "Developer")
                             // інфіксна нотація
```

Результат виклику функції від цього ніскільки не змінився, але код став нагадувати звичайне речення. Тобто позбавилися від крапки і дужок, звернувшись до пропусків.

Постфіксна нотація (Postfix notation)

Підходить лише для методів *без параметрів*.

Продовжимо працювати із створеним раніше класом `Person`. Додамо в нього метод `speaksEnglish`:

```
class Person(val name: String,
             occupation: String) {
  def worksAs(jobName: String): Boolean =
       jobName == occupation
  def speaksEnglish: Boolean = true
}
  val bob = new Person("Bob", "Developer")
```

Постфіксна нотація аналогічна інфіксною, просто використовується *для методів без параметрів*:

```
  println(bob.speaksEnglish)
                          // точкова нотація
  println(bob speaksEnglish)
                          // постфіксна нотація
```

Примітка: на практиці інфіксна і постфіксна нотації застосовуються досить рідко. І перевага віддається точковій нотації. Крім того, у версіях Scala, починаючи з 2.13, поступово уникають постфіксних нотацій, що вимагає додаткового імпортування `import scala.language.postfixOps`.

Оператори

Перш ніж перейти до префіксної нотації, потрібно пригадати про операторів.

1. Оператори в Scala насправді є методами. А це означає, що до них теж можна звернутися як через точку:

```
  println(1 + 2)    // звичайне звернення
  println(1.+(2))   // звернення через точку
```

2. Оператори можна використовувати як ім'я методу.

Можна розширити розглянутий раніше клас `Person` і додати в нього метод `&`

```
class Person(val name: String,
             occupation: String) {
 def worksAs(jobName: String): Boolean =
       jobName == occupation
 def speaksEnglish: Boolean = true
 def &(person: Person): String =
   s"${this.name} and ${person.name} are colleagues"
}
val bob = new Person("Bob", "Developer")
```

Раз в методу & всього *один аргумент* – стає доступна інфіксна нотація:

```
  val alice = new Person("Alice", "Data Engineer")

  println(bob.&(alice))   // точкова нотація
  println(bob & alice)    // інфіксна нотація
```

Префіксна нотація (Prefix notation)

Нехай оголосили і проініціалізували змінну:

```
  val x = -1
```

Для запису вище існує еквівалент з використанням `unary_`

```
val x = 1.unary_-
```

Префікс `unary_` личить лише для операцій `+ - ~ !`

Оператор з `unary_` також можна використовувати як ім'я функції:

```
class Person(val name: String,
             occupation: String) {
 def worksAs(jobName: String): Boolean =
       jobName == occupation
 def speaksEnglish: Boolean = true
 def &(person: Person): String =
   s"${this.name} and ${person.name} are colleagues"
     def unary_! : String = s"$name is not real"
}
```

```
  val bob = new Person("Bob", "Developer")
```

Примітка: у функції `unary_!` немає аргументів, тому *був поставлений пропуск* і лише після нього йшла двокрапка і дописувався тип значення, що повертається. Це було зроблено для того, щоб компілятор не сприймав `unary_!:` як ім'я функції, а використовував як ім'я лише `unary_!` – *без двокрапки.*

Тепер викликати `unary_!` можна трьома способами:

```
  println(!bob)           // префіксна нотація
  println(bob.unary_!)    // точкова нотація
  println(bob unary_!)    // постфіксна нотація
```

Метод apply

Спершу потрібний додати метод `apply` в клас `Person` (те, що метод робитиме – не має значення):

```
class Person(val name: String,
      occupation: String) {
 def worksAs(jobName: String): Boolean =
       jobName == occupation
 def speaksEnglish: Boolean = true
 def &(person: Person): String =
  s"${this.name} and ${person.name} are colleagues"
 def unary_! : String = s"$name is not real"
 def apply(): String =
        s"$name works as a $occupation"
}
  val bob = new Person("Bob", "Developer")
```

Можна використовувати два способи виклику `apply`:

```
  println(bob.apply())     // точкова нотація
  println(bob())           // використовується
                           // найчастіше
```

Причому другий спосіб найбільш популярний. Потрібно запам'ятати: як тільки бачимо якийсь *об'єкт класу*, який *схожий на функцію (є дужки)*, – це неявний виклик `apply`.

Результат виконання:

```
Bob works as a Developer
Bob works as a Developer
```

***Завдання до практичної роботи* 5**

5.1 Реалізувати клас `Person` з функцією, яка спрацьовуватиме на виклик:

```
val person = new Person("Bob")
println((+person).name)
```

і виводить на екран `Bob NoSurname`

Примітка:

– конструктор класу `Person` містить лише одне поле: `name`;

– `name` може бути будь-яким – залежить від того, що програма отримає на вхід при тестуванні;

– проте у всіх буде до імені `name` додано `NoSurname` – незалежно від того, яке ім'я передали на вхід.

Підказка: увага на `(+person).name` – *дужки означають, що .name застосовується до того результату, що дає функція, застосована до* `person` *у префіксній нотації, тобто до unary_+).*

Контрольні запитання

1. Що таке помилки, а що виключення?
2. Який синтаксис механізму обробки виключних ситуацій?
3. Що таке узагальнені класи?
4. Що таке узагальнені методи?
5. Які є типи варіативності і чим вони відрізняються?
6. Що таке синтаксичний цукор?
7. Що таке постфіксна, префіксна та точкова нотація?
8. Як і коли використовується метод `apply`?

Практична робота 6

ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ В SCALA. ВИКОРИСТАННЯ `Option`. ОБРОБКА ВИКЛЮЧНИХ СИТУАЦІЙ SCALA

Мета: Одержання практичних навичок при використанні `options` та механізму обробки виключних ситуацій у мові Scala

Теми для попереднього опрацьовування

– Вступ до функціонального програмування

– Використання `options`

– Обробка виключних ситуацій

Теоретичні відомості

`Option` – є структурою даних, яка використовується за наявності невизначеності в програмі, коли немає упевненості, що якесь обчислення поверне значення. В разі `Option` можуть виникнути дві ситуації:

– значення відсутнє;

– значення присутнє.

Ці ситуації описуються двома підкласами: `Some` (зазвичай означає наявність значення в `Option`) і `None`.

Це всього лише спосіб захисту від виключення `NullPointerException`

Не слід вручну робити перевірки на `Null` – можна помістити в `Option` змінну, яку потрібно перевірити на відсутність значення:

```
def unsafeMethod(): String = null
def safeMethod(): String = "There is a String"
val unsafeRes = Option(unsafeMethod())
val safeRes = Option(safeMethod())

println(unsafeRes) // None
println(safeRes)   // Some(There is a String)
```

Тобто Option поверне None *за відсутності значення* і Some *із значенням, якщо значення є.*

Можна перевірити, чи є Option Some або None, використовуючи наступні методи:

isDefined – значення true, якщо об'єкт має значення Some;

nonEmpty – значення true, якщо об'єкт має значення Some;

isEmpty – значення true, якщо об'єкту привласнено значення None.

Нехай визначені змінні:

```
val someOption: Option[String] = Some("Success")
val noneOption: Option[Int] = None
```

```
  println(someOption.isDefined)   //true
  println(someOption.nonEmpty)    //true
  println(someOption.isEmpty)     //false

  println(noneOption.isDefined)   //false
  println(noneOption.nonEmpty)    //false
  println(noneOption.isEmpty)     //true
```

Отримання *змісту* Option

Можна отримати значення Option за допомогою метода get. Якщо викликати метод get для екземпляра None, то буде створено *виключення* NoSuchElementException.

```
println(someOption.get)   // Success
println(someOption)       // Some(Success)

println(noneOption.get)
              //виключення NoSuchElementException
println(noneOption)      // None
```

Значення `Option` *за умовчанням*

Існує також кілька інших методів вилучення:

`getOrElse` – витягує значення, якщо об'єкт має *значення* `Some`, інакше повертає значення *за умовчанням* (у прикладі – 0);

`orElse` – витягує *параметр,* якщо він має *значення* `Some`, інакше повертає *альтернативний параметр*:

```
println(someOption.getOrElse(0))      // Success
println(noneOption.getOrElse(0))      // 0

println(someOption.orElse(noneOption))
                                  //Some(Success)
println(noneOption.orElse(someOption).get)
                                  //Success
```

***Приклад* 1**. Використання `getOrElse`. Метод об'єднує в собі перевірку і виведення значення (через `get`):

```
println(noneOption getOrElse(someOption))
                                    // Some(Success)
```

Тут `noneOption` не має значення, отже `get` не спрацьовує, програма бере значення, вказане для випадку `Else`, а це `someOption`, значення якого `Some(Success)`.

```
println(someOption getOrElse(noneOption))
                                  // Success
```

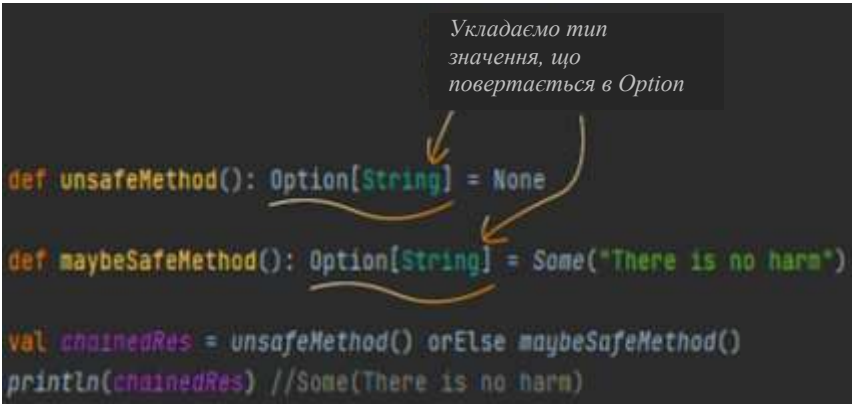
В цьому випадку значення в `someOption` є, тому спрацьовує `get` і повертається само значення `Success`.

***Приклад* 2**. Коли є необхідність протестувати декілька методів одночасно з допомогою `Option`, зробити код більш лаконічним допоможе конструкція `orElse` (тут для виклику методу застосований *синтаксичний цукор*):

```
def unsafeMethod(): String = null
```

```
def maybeSafeMethod(): String =
                    "There is no harm"
val chainedResult =
Option(unsafeMethod()) orElse (Option(maybeSafeMethod()))
println(chainedResult) // Some(There is no harm)
```

Як альтернатива – можна закласти перевірку безпосередньо в описуваний метод:




```
def unsafeMethod1(): Option[String] = None
def maybeSafeMethod1(): Option[String] =
                        Some("Everything is Ok")
val chainedRes = unsafeMethod1() orElse
                 maybeSafeMethod1()
println(chainedRes)    // Some(Everything is Ok)
```

***Приклад* 3**. Яким буде результат наступного коду?

```
scala> val someVal = Some(2)
```

someVal: Some[Int] = Some(2)

```
scala> someVal.map(_ * 2)
```

res1: Option[Int] = Some(4)

Коротке визначення класу Some:

```
final case class Some[+A](val value : A) extends scala.Option[A]
```

***Приклад* 4**. `Option` найчастіше використовується для охорони (`guard`) при пошуку якихось даних по ключу. Наприклад, це може бути витягання значень з `Map` (також і з будь-якої бази даних). Якщо в програмі станеться виклик елементу з неіснуючим ключем, станеться виключення (повідомлення про помилку з остановом). Для запобігання такій небажаній ситуації при витяганні слід застосовувати метод `get`.

Нехай визначена змінна:

```
val h = Map(1 ->"aa", 2->"bb", 3->"cc")

println(h(2))       // bb
println(h(9))       // NoSuchElementException
```

Тепер застосуємо `get`. Функція `get` повертає `Option` і не допускає виключення:

```
println(h.get(2))   // Some(bb)
println(h.get(9))   // None

scala> h.get(2)
res1: Option[String] = Some(bb)
scala> h.get(4)
res2: Option[String] = None
```

Щоб витягти з `Some(bb)` сам елемент, треба ще раз застосувати `get`:

```
println(h.get(2).get)   // bb
println(h.get(9).get)   // NoSuchElementException
```

Проте, і це ще не всі проблеми, оскільки вживання `get` до `None` теж викличе виключення. З цієї причини замість другого `get` краще застосовувати метод `getOrElse`, який в разі `None` повертатиме задане значення за умовчанням замість виключення.

***Приклад* 5**.

```
val p = Map(1 -> "David", 9 -> "Elwood", 8 -> "Archer")
println(p.getOrElse(99, "Nobody"))    // Nobody
println(p.getOrElse(9, "Nobody"))     //  Elwood
```

Ключа 99 немає і метод `getOrElse` повернув задане значення за умовчанням. Ключ 8 є, і отримали потрібне значення. Таким чином, роблячи в програмі перевірку на `None`, можна уникнути помилкових ситуацій.

Виключення `Try`

З ООП відомо про існування механізму обробки виключних ситуацій за допомогою `try-catch-finally`. У функціональному програмуванні використовують клас `Try`.

Щоб `Try` став доступний, його необхідно заздалегідь імпортувати:

```
import scala.util.Try
```

Тип `Try` представляє обчислення, яке може або привести до виключення, або повернути успішно обчислене значення.

Екземпляр `Try[T]` може бути або `Success(v)`, де `v` – значення типу `T`, або `Failure(ex)`, де `ex` – `Throwable` (екземпляр класу `java.lang.Throwable`).

Синтаксис: пишемо `Try`, а потім в круглих дужках вказуємо метод, який було б бажано перевірити:

```
def unsafeMethod(): String =
  throw new RuntimeException("Sorry, not your day")
val potentialFailure = Try(unsafeMethod())
println(potentialFailure)
     // Failure(java.lang.RuntimeException:
     // Sorry, not your day)

def myMethod(): String = "My method is valid"
println(Try(myMethod()))
     // Success(My method is valid)
```

Як альтернатива – можна писати `Try`, потім через пропуск відкривати фігурні дужки, в яких прописувати код для тестування.

```
val anotherPotentialFailure = Try {
              // код, що містить виключення
}
```

Визначити успішне чи неуспішне завершення обчислень можна також за допомогою методів `isSuccess` і `isFailure` об'єкту `Try`. Можна заздалегідь перевірити, чи містить метод виключення, отримавши у відповідь `true` або `false`:

```
println(Try(unsafeMethod()).isSuccess) // false
println(Try(myMethod()).isSuccess)     // true
println(Try(unsafeMethod()).isFailure )// true
println(Try(myMethod()).isFailure )    // false
```

`Try` і `orElse`

Коли є необхідність протестувати декілька методів одночасно з допомогою `Try`, зробити код лаконічніше допоможе конструкція `orElse`.

Наприклад:

```
def unsafeMethod(): String =
   throw new RuntimeException("Sorry, not your day")
def myMethod(): String = "My method is valid"

println(Try(unsafeMethod()))
       // Failure(java.lang.RuntimeException:
       // Sorry, not your day)

println(Try(myMethod()))
       // Success(My method is valid)

println(Try(unsafeMethod()).orElse
        (Try(myMethod())))
       //  Success(My method is valid)
```

Ще одним варіантом буде обернути в `Try` результат роботи методу.

```
import scala.util.{Try, Failure, Success}

def methodWhichFails(): Try[String] =
       Failure(new RuntimeException("Ooops..."))
def methodWhichSucceeds(): Try[String] =
       Success("Everything is OK")

val tryMethods = methodWhichFails() orElse
                 methodWhichSucceeds()
println(tryMethods)  // Success(Everything is OK)
```

***Завдання до практичної роботи* 6**

6.1 Зробити функцію `divide`, яка ділить одне число на інше і повертає типа `Option`. Друк результату ділення здійснити за допомогою функції `showDivide`, яка при діленні на 0 повертає "`null division`".

Результати виконання:

```
println(showDivide(10, 5))    // 10 / 5  = 2
println(showDivide(10, 0))    // null division
```

6.2 Стандартний метод `List.find(p: Char => Boolean):Option[Char]` намагається знайти перший символ, що задовольняє предикату `p`.

Написати функцію `def foo(list: List[Int]): Int`, яка находить в списку `List` перше число, яке ділиться на 3, і множить його на 2. Якщо таких чисел нема – функція повертає 0. Використовувати функцію `find` і методи `Option`.

Результат виконання повинен бути наступним:

```
println(foo(List(1,2,4,5,7,8)))       //0
println(foo(List(1,2,9,5,7,6)))       //18
```

Контрольні запитання

1. Що таке `Option`?

2. Якими двома підкласами описуються ситуації присутності та відсутності значення?

3. Як використовуються методи `isDefined`, `nonEmpty` та `isEmpty`?

4. Як отримати зміст `Option`?

5. Як отримати значення `Option` за умовчанням?

6. Чим відрізняються методи `getOrElse` та `orElse`?

7. В яких ситуаціях слід застосовувати метод `get`?

8. Як у функціональному програмуванні використовують клас `Try`?

9. Яке використання методів `isSuccess` і `isFailure` об'єкту `Try`?

Практична робота 7

ШАБЛОНИ (PATTERN MATCHING) В SCALA. ЧАСТКОВІ ФУНКЦІЇ (PARTIAL FUNCTIONS)

Мета: Одержання практичних навичок при використанні Pattern Matching та часткових функцій у мові Scala

Теми для попереднього опрацьовування

− Вступ до функціонального програмування

− Використання шаблонів Pattern Matching

− Використання часткових функцій

Теоретичні відомості

Шаблони **Pattern Matching** застосовуються тоді, коли для конкретних значень потрібно визначити свій варіант дії програми (можна провести паралелі з `switch-case` з інших мов).

Наприклад:

```
val someVal = 3
val description = someVal match {
  case 1 => "action 1"
  case 2 => "action 2"
  case 3 => "action three"
  case _ => "default action"
}
println(description) // action three
```

Рядом з `case` задаються шаблони, які порівнюються із значенням змінної `someVal`. А потім => вказується бажаний результат для заданого випадку.

Символ _ зазвичай описує ситуацію, коли збігів серед шаблонів не знайдено тобто результат *за умовчанням*.

Види шаблонів

Як шаблони можна використовувати:

– *константи*

```
  val x: Any = "Scala"
  val constants = x match {
    case 1 => "число"
    case "Scala" => "рядок"
    case true => "булеве значення"
  }
println(constants)                    // рядок
```

Зверніть увагу на тип `Any`. Він знадобився через те, що як `x` у цьому прикладі може бути використане і *число*, і *рядок*, і *булеве значення*.

– *`Tuple` (кортежи)*

```
  val aTuple = (5,2)
  val matchATuple = aTuple match {
    case (1, 1)         => "повний збіг"
    case (something, 2) =>
                    s"я знайшов $something"
  }
  println(matchATuple)          // я знайшов 5
```

Вкладені `Tuple` також не є проблемою:

```
  val nestedTuple = (1, (2, 3))
  val matchNestedTuple = nestedTuple match {
    case (_, (2, v)) => s"тут є число $v"
  }
  println(matchNestedTuple)     // тут є число 3
```

– *списки*

```
  val aStandardList = List(5, 4)
  val listMatching = aStandardList match {
    case List(1, _, _, _) => "список з 4
        елементів, де перший елемент дорівнює 1"
    case List(2, _*) => "список довільної
        довжини, де перший елемент дорівнює 2"
```

```
  case List(3, 2, 1) :+ 0 =>
         "список List(3, 2, 1, 0)"
  case 5 :: List(_) => "список, де першим йде
       число 5 і ще один елемент"
}
println(listMatching)
     // список, де першим йде число 5
     // і ще один елемент
```

Оператор `::` служить для додавання елементу в початок списку. Код `5 :: List(_)` можна переписати в точковій нотації `List(1).::(5)`, де:

– якщо вказано `_` , то після 5 повинен йти точно один елемент;

– якби вказали `_*`, то розглядали б список, де першим елементом йшло число 5, а далі – довільна кількість елементів, власне, список `List(5)` теж би підійшов під умову (*тобто полягаючий мінімум з одного елементу, причому першим елементом має бути число* 5).

При виконанні програми виконується `5 :: List(_)` і вже отриманий результат `List(5, _)` використовується порівняно із заданим для `match` значенням.

– *типи*

```
val unknown: Any = List(1, 2  )
val typeMatch = unknown match {
  case list: List[Int] => "list of INTs"
  case _ => "default"
}
println(typeMatch)            // list of INTs
```

– *класи-зразки*

```
case class Person(name: String, age: Int)

val bob = Person("Bob", 30)
val greeting = bob match {
```

```
  case Person(n, a) if a < 18 =>
              s"I'm $n and I'm under 18"
  case Person(n, a) =>
              s"I'm $n and I am $a years old"
  case _ => "I have no name"
}
println(greeting)
           // I'm Bob and I am 30 years old
```

Прив'язка імен (Name Binding)

Використовується, якщо потрібно якось звернутися до знайденого збігу. Задається ім'я збігу, ставиться знак `@`, після якого прописується сам шаблон:

```
val nameBindingMatch = List(6, 2) match {
  case nonEmptyList@List(1, _, _, _) =>
     s"нашли $nonEmptyList"
  case someList@List(6, _*) =>
     s"нашли список $someList"
}

println(nameBindingMatch)
                    // нашли список List(6, 2)
```

***Приклад* 1**. Нехай є код:

```
val numbers = List(1, 2, 3)
val numbersMatch = numbers match {
  case listOfStrings: List[String] =>
            "a list of strings"
  case listOfNumbers: List[Int] =>
            "a list of integers"
  case _ => "default case"
}

println(numbersMatch)
```

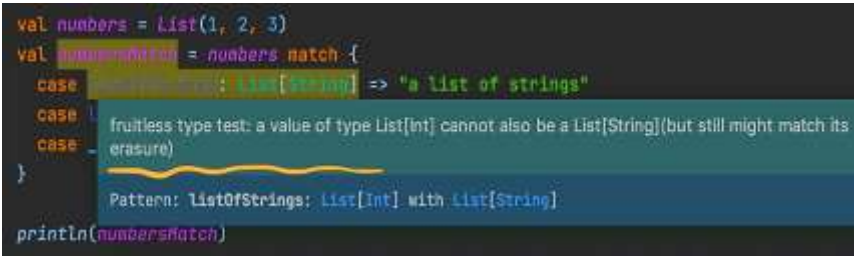
Якщо код запустити, то на екрані можна буде побачити `a list of strings`. І це не дивлячись на те, що на вхід був поданий цілочисельний список.

Причина цьому – JVM. Компілятор насправді бачить код без *generic*-тіпів. Тобто для компілятора код виглядає ось так:

```
val numbersMatch = numbers match {
  case listOfStrings: List =>
              "a list of strings"
  case listOfNumbers: List =>
              "a list of integers"
  case _ =>   "default case"
}
```

А оскільки шаблони перевіряються по порядку – один за іншим, то перший же шаблон вважатиметься відповідним, і на екрані буде `a list of strings`.

Саме цим і пояснюється те, що було видано попередження:



***Приклад* 2.** Який патерн підійде для пошуку `List`, що складається з *одного* або *декількох* елементів?

```
List(_, _*)
```

***Приклад* 3**. Реакція програми на числове значення, яке приймає змінна:

```
val count = -5
count match {
 case 1 => println("one, a lonely number")
 case x if x == 2 || x == 3
        => println("two's company, three's a crowd")
```

```
      case x if x > 3
             => println("4+, that's a party")
      case _ => println("i'm guessing your number is
zero or less")
}
      // i'm guessing your number is zero or less
```

Часткові функції (Partial functions)

Часто буває так, що потрібно обмежити значення, які можна подавати на вхід функції. Це можна зробити через шаблони. Так, наприклад, в коді допускається лише значення `mon, fri, sun`:

```
       val whatToDo = (d: String) => d match {
         case "mon" => "Work!"
         case "fri" => "Party Time"
         case "sun" => "Relax a little"
       }
     println(whatToDo("fri")) // Party Time
     println(whatToDo("sat")) //виключення MatchError
```

Проте є і інше рішення, яке засноване на вживанні `Partial functions` – функція, яка може бути визначена не для всіх вхідних значень. Такі функції є екземплярами класу `PartialFunction[A, B]`, де (`A` – тип параметру, `B` – *тип значення, що повертається*.)

Цей клас має два методи:

– `apply`, що обчислює значення функції із зіставлення із зразком;

– `isDefinedAt`, повертаючий `true`, якщо вхідне значення збігається хоч би з одним зразком.

```
     val  aPartialFunction:  PartialFunction[String,
String] = {
         case "mon" => "Work!"
         case "fri" => "Party Time"
         case "sun" => "Relax a little"
       }
```

Якщо запустити функцію, то результат буде таким:

```
println(aPartialFunction("sun"))
                        // Relax a little
println(aPartialFunction("sat"))
                        // виключення MatchError
```

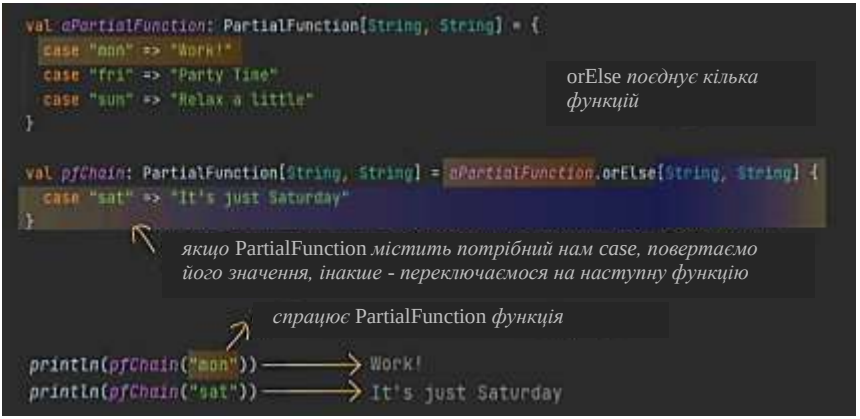
Для недопустимого значення отримуємо `MatchError`. Це говорить про те, що часткові функції засновані на збігу шаблонів.

***Приклад* 4**. Використання функцій `apply` і `isDefinedAt` (перевірка аргументу).

```
val f: PartialFunction[Char, Int] = {
 case '+' => 1 ;
 case '-' => -1
}
f('-')        // Виклик f.apply('-'), поверне -1
              // тут - синтаксичний цукор
f.apply('+')              // 1
f.isDefinedAt('0')        // false
f('0')                    //виключення MatchError
```

Часткові функції і `orElse`

Об'єднати декілька функцій в ланцюг можна з допомогою `orElse`:



```
    val aPartialFunction: PartialFunction[String,
String] = {
    case "mon" => "Work!"
    case "fri" => "Party Time"
    case "sun" => "Relax a little"
  }
val pfChain:PartialFunction[String,String]=
  aPartialFunction.orElse[String,String]{
    case "sat" => "It's just Saturday"
  }
  println(pfChain("mon"))     // Work
  println(pfChain("sat"))     // It's just Saturday
```

Ліфтінг

Можна сказати, що ліфтінг дозволяє підняти часткову функцію на наступний рівень. Метод `lift` перетворить `PartialFunction[A, B]` у звичайну функцію, тобто після використання `lift` – функція повертатиме значення типу `Option[B]` (тобто вирішується проблема `MatchError`)

```
      val aPartialFunction: PartialFunction[String,
String] = {
        case "mon" => "Work!"
        case "fri" => "Party Time"
        case "sun" => "Relax a little"
      }
      val lifted = aPartialFunction.lift
                       // тепер на виході маємо
                       // тип Option[String]
      println(lifted("fri")) // Some(Party Time)
      println(lifted("thu")) // None
```

***Приклад* 5**. Використання метода `lift`

```
val f: PartialFunction[Char, Int] = {
 case '+' => 1 ;
 case '-' => -1
}
val g = f.lift                    //  Функція  типу
Char => Option[Int]
println(g('-'))                   // Some(-1)
println(g.apply('&'))             // None
```

***Завдання до практичної роботи* 7**

7.1 Що потрібно дописати на місці пропущеного коду, щоб отримати бажаний результат:

```
1 – непарне число
2 – парне число
3 – непарне число
4 – парне число
5 – непарне число

for (i <- 1 to 5) {
    i match {
      // тут пропущений код
    }
}
```

7.2 Використовуючи зіставлення із зразком, напишіть функцію `swap`, яка приймає пару цілих чисел і повертає ту ж пару, помінявши компоненти місцями.

7.3 Написати метод `guard`, який отримує на вхід два параметри: `data` і `maxLength`:

- `data` – це вхідний параметр для перевірки. Це може бути все, що завгодно: число, рядок, список, вектор – але що саме, заздалегідь *не відомо*;
- `maxLength` – максимально допустима довжина рядка або списку;
- типом значення, що повертається для `guard` є `String`.

Усередині методу необхідно прописати шаблон, що складається з п'яти `case`. Результати перевірок на відповідність шаблону наступні:

1) задано список `List` допустимої довжини;
2) довжина списку більше максимально допустимого значення;
3) довжина рядка не перевищує максимально допустимого значення;
4) отримано рядок недопустимої довжини;
5) що це? Це не рядок і не список.

Результати виконання:

```
println(guard(List(1,2,3,4,5,6,7),7))
  // задано список List допустимої довжини
println(guard(List(1,2,3,4,5,6,7),3))
  // довжина списку більше максимально
  // допустимого значення
println(guard("Hello",7))
 // довжина рядка не перевищує максимально
 // допустимого значення
println(guard("Hello",2))
 // отримано рядок недопустимої довжини
println(guard(1,2))
 // що це? Це не рядок і не список
```

7.4 Необхідно створити чат-бот. Бот реагує лише на три фрази:

```
hello
bye
what's up
```

Відповідно, його відповідями будуть:

```
Hi, I'm Bot
Bye-bye
sup-sup
```

Примітка:

– використовувати `PartialFunction`;

– передбачити, щоб жодних виключень не виникало (задати лише допустимі значення і не вказувати значення за умовчанням);

– до функції звертатися через змінну типу `Option[String]` з ім'ям `chatbot`;

– подумати, як вирішити завдання не через `isDefinedAt()`.

Підказка:

1) виклик має бути таким: `println(chatbot("hello"))`

2) щоб отримати повідомлення з `Some`, використовувати `getOrElse` (повертає значення `Option` або, якщо воно відсутнє, повертає задане за умовчанням значення).

Контрольні запитання

1. Коли застосовуються шаблони Pattern Matching?
2. Які існують види шаблонів?
3. Що таке прив'язка імен (Name Binding)?
4. Коли використовуються часткові функції (Partial functions)?
5. Як використовуються методи `apply` та `isDefinedAt`?
6. Що таке ліфтінг?

Практична робота 8

ФУНКЦІЇ. КОЛЕКЦІЇ В SCALA

Мета: Одержання практичних навичок при використанні функцій та колекцій у мові Scala

Теми для попереднього опрацьовування

– Функції, чисті функції

– Анонімні функції або лямбда-функції

– Функції вищого порядку

– Колекції

Теоретичні відомості

Функціональне програмування – це написання коду з використанням лише *чистих функцій* і *незмінних змінних*.

Чиста функція (pure function) – це коли:

– результат роботи функції залежить лише від того, що вона отримує на вхід, і описаного усередині неї алгоритму. А значить, скільки б разів така функція не була викликана, наприклад, для параметра `x` – результат її роботи буде незмінним;

– немає ні читання, ні запису у файл або будь-якої іншої взаємодії із зовнішнім джерелом даних (файлом, базою даних, UI).

Прикладом чистої функції може служити функція `toUpperCase`. Функція `System.nanoTime()` – це не чиста функція.

Наприклад, є код:

```
val a = f(x)
val b = y(a)
val c = z(b)
```

Такий стиль дає ряд переваг: так відомо, що `a` – це завжди `f(x)`, тому `b` можна представити як `y(f(x))`. Аналогічні дії можна виконати і з `c`. Отже код нагадуватиме наступний:

```
val c = z(y(f(x)))
```

Функції

При роботі з функціями можна:

– використовувати функції як параметри;

– повертати функції з функцій.

Допустимо, необхідна функція *множення числа на константу*. Пригадавши ООП, де працюємо з класами і їх екземплярами, можна написати ось такий код:

```
class Multiplication {
  def multiply(x: Int): Int = x * 2
}
```

Можна, застосувавши узагальнені класи, зняти обмеження у використанні типів (*у квадратних дужках вказуються типи, робота з узагальненнями*):

```
trait Multiplication[A, B] {
  def multiplyh(x: A): B
}
```

Ще є `apply`:

```
trait Multiplication[A, B] {
  def apply(x: A): B
}
```

Тепер, можна оголосити змінну `res` (як анонімний клас з перевантаженим методом `apply`):

```
val res = new Multiplication[Int, Int] {
  override def apply(x: Int): Int = x * 2
}
```

Звертатися до неї можна так, що виглядати це буде – як звернення до функції:

```
println(res(2))  // звернення до змінної, як до
                 // функції – неявний виклик
                 // функції apply
```

Потрібно врахувати, що подібні трейти вже прописані в Scala. Це `Function0, Function1, Function2, Function3` і так до `Function22` включно. Числа 0, 1, 2, 3, ..., 22 в конці слова `Function` – вказують на кількість параметрів, що задаються.

Таким чином, для даної задачі з *одним* параметром, що передається, підійде трейт `Function1[A, B]`:

```
val res = new Function1[Int, Int] {
  override def apply(x: Int): Int = x * 2
}
```

Відповідно, якщо потрібно передати два параметри, використовується `Function2` (третій `Int` в дужках у `Function2` – позначає тип значення, що повертається):

```
val product = new Function2[Int, Int, Int] {
  override def apply(x: Int, y: Int): Int = x * y
}
println(product(3, 4) )              // 12
```

Анонімні функції або лямбда-функції

У Scala необов'язково давати імена всім функціям, як необов'язково давати імена всім числам. Нижче демонструється анонімна функція:

```
(x: Double) => 3 * x
```

Цю функцію можна було б зберегти в змінній:

```
val triple = (x: Double) => 3 * x
```

Це рівносильно використанню визначення `def`:

```
def triple(x: Double) = 3 * x
```

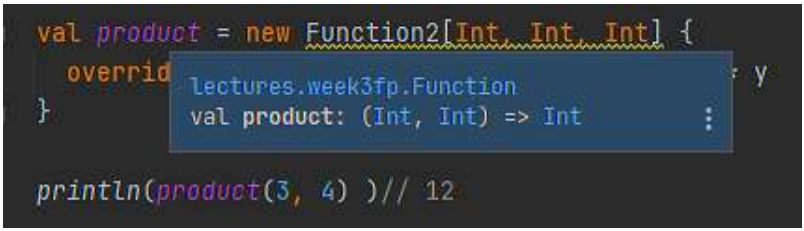
Функції можна просто передавати іншим функціям:

```
Array(3.14, 1.42, 2.0).map((x: Double) => 3 * x)
                    // Array(9.42, 4.26, 6.0)
```

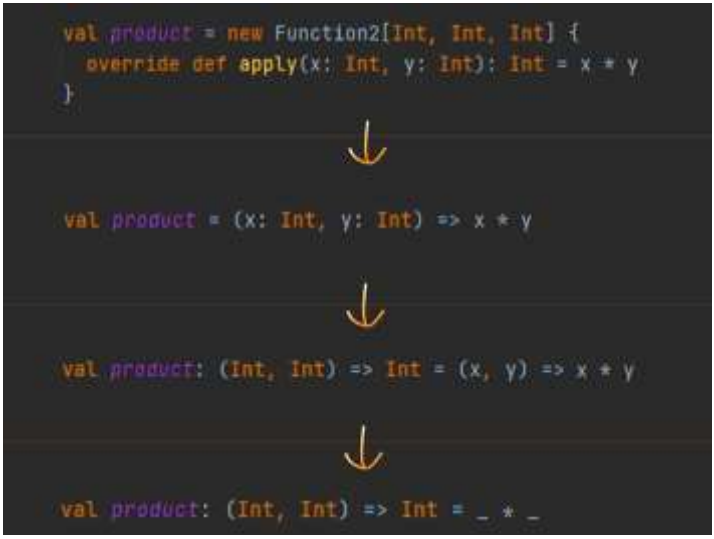
Для функції, розглянутої вище:

```
val product = new Function2[Int, Int, Int] {
  override def apply(x: Int, y: Int): Int = x * y
}
```

Якщо навести мишку і поглянути на *тип змінної* `product`, то видно `(Int, Int) => Int`. А це є ще одним *синтаксичним цукром*:

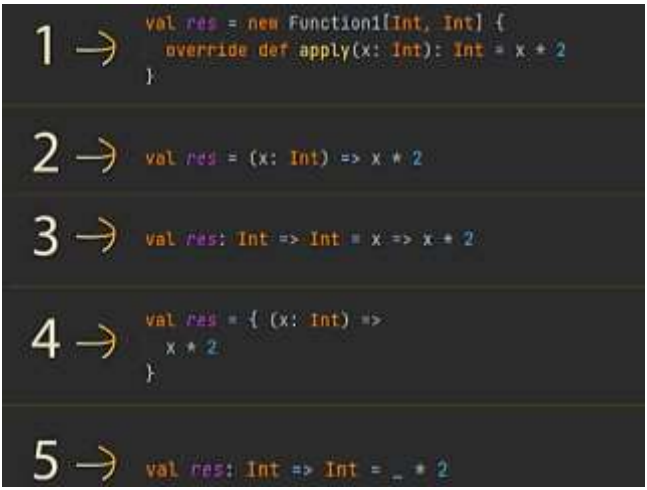


Способів написання коду з використанням анонімних функцій (або лямбди-виразів) декілька (і всі вони дадуть один і той же результат):



Останній метод дозволяє компілятору самому підставити змінні замість _. Зручно, але можна сплутати зі змінними, що передаються.

***Приклад* 1**. Функція з одним параметром. Код під номером 4 – це ще один спосіб написання анонімних функцій – у фігурних дужках. Такий спосіб корисний, коли необхідно визначити багаторядкову анонімну функцію.




```
val product = new Function2 [Int, Int, Int] {
  override def apply(x: Int, y: Int): Int =
    if (x > y) x else y
}

  val product1 = (x:Int,y:Int) =>
    if (x > y) x else y

  val product2: (Int, Int) => Int = (x,y) =>
    if (x > y) x else y
```

При зверненні до лямбди-функцій обов'язково використовувати дужки – інакше отримують саму функцію, а не результат її виконання.

Наприклад:

```
val product1 = () => println("Анонімна функція ")
```

```
println(product1)
  //WordCount$$$Lambda$6/1504109395@56ac3a89
println(product1()) // Анонімна функція
                    //()
val product3 = {
  (x: Int, y: Int) =>
   if (x > y) x else y
  }
println(product3 (5,4))   //5
println(product3)
           //Prim$$$Lambda$6/1908316405@72d818d1
```

Функції вищого порядку (High Order Functions)

Під функціями вищого порядку розуміють такі функції, які *на вхід отримують іншу функцію або ж, як результат, повертають функцію*.

Як приклад потрібно написати функцію `nTimes`, яка отримує на вхід три параметри: `f, x, n,` де `f` – це та функція, яка буде застосована до параметра `x` вказану кількість разів (тут `n` разів).

Функцію `nTimes` можна написати декількома способами.

Спосіб 1. Передаємо всі три аргументи відразу – в одних дужках:

```
@tailrec
def nTimes(f: Int => Int, x: Int,  n: Int): Int = {
  if (n <= 0) x
  else nTimes(f, f(x), n - 1)
}
```

Як функцію `f`, яка буде передаватися як аргумент, взята *анонімна функція, що збільшує число на одиницю*

```
  val increment = (x: Int) => x + 1
```

Тоді запуск функції `nTimes` буде виглядати таким чином:

```
 println(nTimes(increment, 0, 3))          // 3
```

Операція каррування для функцій вищого порядку (Currying Functions)

Перед тим, як розглянути *Спосіб* 2 написання функції `nTimes`, необхідно розібратися, що таке *каррування*.

Під каррованою функцією мають на увазі функцію, яка на вхід приймає декілька аргументів (причому – можна сказати, що аргументи розбиті на групи). А в тілі цієї функції відбувається серія викликів функцій, кожна з яких приймає *єдиний* аргумент.

Звертаючись до алгебри, розписати весь процес можна приблизно так:

```
f1 = f(x)
f2 = f1(y)     //f(x)(y)
result = f2(r)       //f(x)(y)(r)
```

Інакше кажучи:

```
result = f(x)(y)(r)
```

Розглянемо приклад функції складання. Можна написати так:

```
  def add(x: Int, y: Int) = x + y
  println(add(1, 2))                          // 3
```

А можна так (увага на аргументи):

```
  def add(x: Int):Int => Int = (y: Int) => x + y
```

Або

```
  def add(x: Int) = (y: Int) => x + y

println(add(1)(2)) // 3
```

Тобто результатом `add(1)` є функція `(y: Int) => 1 + y`. Ця функція передає число 2 і повертає 3.

Або *скорочена* форма визначення таких каррованих функцій в Scala:

```
 def add(x: Int)(y: Int) = x + y
  println(add(1)(2))
```

Примітка. Все, що визначається за допомогою `def`, є методом, а не функцією. Визначаючи карровані методи за допомогою `def`, можна використовувати декілька пар круглих дужок:

```
scala> def mulOneAtATime(x: Int)(y: Int) = x * y
```

mulOneAtATime: (x: Int)(y: Int)Int

У даному прикладі метод має тип `(x:Int)(y:Int)Int`.

Навпаки, визначаючи функцію, потрібно використовувати *декілька стрілок*, а не пар круглих дужок:

```
scala> val mulOneAtATime = (x: Int) => (y: Int)
=> x * y
```

mulOneAtATime: Int => (Int => Int)

Спосіб 2. Використання каррування `curryingNTimes:(f, n) (x)`

Виглядати це тепер буде так:

```
def    им'яФункції(аргумент1,    агрумент2)    =
(аргумент3) => операція
```

Або, може бути ось такий запис:

```
curryingNTimes(f,n) = x => f( f(f...f(x) ) )
```

І сам код:

```
  def curryingNTimes(f: Int => Int, n: Int): Int
=> Int = {
    if (n <= 0) (x: Int) => x
                // лямбда-функція,
                // яка просто
                // бере і повертає значення
    else (x: Int) => curryingNTimes(f, n-1)(f(x))
  }
```

Сам виклик тепер матиме наступний формат:

```
  println(curryingNTimes(increment, 3)(0))
```

***Приклад* 2**. Розглянемо ще одну функцію, де множаться два числа.

Для цього на вхід подається число `Int`, повертається ще одна функція, на вхід якої подається число і яка в результаті повертає ціле число (результат множення):


```
val multiplier:Int => Int => Int = x => y => x*y

  val multiplyBy2 = multiplier(2)
             // Int => Int y => 2 * y -
             // передали функції перший аргумент
  println(multiplyBy2(5))             //10
            // залишилось передати другий аргумент
  println(multiplier(2)(5))           //10
```

***Приклад* 3**. У прикладі визначення функції `succ` є аналогічним анонімному визначенню функції `anonfun1`:

```
val succ = (x: Int) => x + 1
val anonfun1 = new Function1[Int, Int] {
      def apply(x: Int): Int = x + 1
    }
  println(succ(2))              //3
  println(anonfun1(2))          //3
```

***Приклад* 4**. Є функція, яка на вхід приймає ціле число, а повертає функцію, яка повертає суму двох чисел.

```
def someFunc: Int => Function1[Int, Int] =
   new Function1[Int, Function1[Int, Int]] {
```

```
    override def apply(x: Int): Function1[Int, Int] =
        new Function1[Int, Int] {
          override def apply(y: Int): Int = x + y
        }
  }
```

Переписати дану функцію можна наступними способами:

```
def someFunc = (x: Int) => (y: Int) => x + y
def someFunc: Int => (Int => Int) = (x) => (y) => x + y
def someFunc: Int => ((Int) => Int) = (x: Int) =>
                     (y: Int) => x + y
def someFunc: Int => (Int => Int) = (x: Int) =>
                     (y: Int) => x + y
```

Якщо

```
    val res = someFunc(1)
```

то

```
    println(res)
        // WordCount$$$Lambda$7/1682092198@4b952a2d
    println(res(4))                          //5
    println(someFunc(3)(4))                  //7
```

Колекції (Collections)

Основні трейти колекцій

Всі колекції поділяються на три основні категорії: *множини*, *послідовності* та *асоціативні масиви*:

`Seq` – послідовності, що мають певний порядок (тобто в кожного елементу свій індекс, наприклад – `Vector, Range, List, Array`);

`Set` – множина, що не містить однакових елементів;

`Map` – асоціативні масиви (це сховище ключ-значення).

На рис. 8.1 представлені найбільш важливі трейти, що утворюють ієрархію колекцій у Scala.

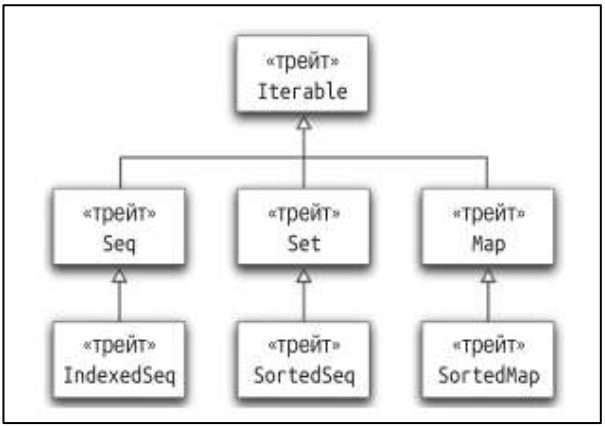


Рисунок 8.1 – Основні трейти, що утворюють ієрархію колекцій Scala

`Iterable` – це будь-яка колекція, здатна повертати ітератор `Iterator`, що забезпечує доступ до всіх елементів в колекції.

`Seq` – це впорядкована колекція значень, така як масив або список. `IndexedSeq` забезпечує швидкий довільний доступ з використанням цілочисельних індексів. Наприклад,, `ArrayBuffer` забезпечує доступ до елементів по індексах, а зв'язаний список – ні.

`Set` – невпорядкована колекція значень. Обхід елементів у `SortedSet` завжди виконується в порядку сортування.

`Map` – множина пар (ключ, значення). `SortedMap` забезпечує обхід елементів в порядку сортування ключів.

Кожен трейт або клас колекції в Scala має об'єкт-компаньйон з методом `apply`, що створює екземпляри колекцій. Це називається «Принципом одноманітності створення».

Наприклад:

```
Iterable(0xFF, 0xFF00, 0xFF0000)
Set(Color.RED, Color.GREEN, Color.BLUE)
Map(Color.RED -> 0xFF0000, Color.GREEN -> 0xFF00)
SortedSet("Hello", "World")
```

Існують також методи `toSeq`, `toSet`, `toMap` та інші і узагальнений метод `to[C]`, які можна використовувати для перетворення колекцій з одного типа в іншій. Наприклад:

```
val coll = Seq(1, 1, 2, 3, 5, 8, 13)
val set = coll.toSet
val buffer = coll.to[ArrayBuffer]
```

`Seq` – послідовності, що мають певний порядок.

```
val aSequence = Seq(1,3,2,4)
println(aSequence)                // List(1,3,2,4)
println(aSequence.length)         // 4
println(println(aSequence++Seq(6,7,5)))
                                  // List(1,3,2,4,6,7,5)
println(aSequence.reverse)
                                  // List(4,2,3,1)
println(aSequence.sorted)
                                  // List(1,2,3,4)
println(aSequence(1))                    // 3
            // отримання елементу по індексу
println(aSequence.search(3))         // Found(1)
            // повертає індекс елемента, що
            // був знайдений
```

`Set` – множина, що не містить однакових елементів (колекція елементів одного типу). За умовчанням створюється незмінний `Set`.

```
val emptySet:Set[Int]= Set()
val aSet:Set[Int]= Set(10,20,30,40)
val anotherSet:Set[Int]= Set(30,40,50,60)
println(aSet.isEmpty)                  // false
println(emptySet.isEmpty)              // true
println(aSet.head)                     // 10
println(aSet.tail)          // Set(20,30 40)
println(aSet.min)                      // 10
println(aSet.max)                      // 40
println(aSet.intersect(anotherSet))
                                  // Set(30, 40)
println(aSet.&(anotherSet))  // Set(30, 40)
```

```
println(aSet.++(anotherSet))
             // Set(10, 20, 60, 50, 40, 30)
println(aSet++ anotherSet)
             // Set(10, 20, 60, 50, 40, 30)
```

Map – асоціативні масиви (це сховище ключ-значення).

```
val aMap:Map[String,Int]=Map()
       //створили пустий Map
       //(викликали метод apply)
val colors:Map[String,String]=
    Map(("black","#00000"),
     "blue"->"#00001", //синтаксичний цукор
    ("Blue","#00002"))
     .withDefaultValue("na") //допомогає
         // уникнути NoSuchElementException
println(colors)
       // Map(black -> #00000,
       // blue -> #00001, Blue -> #00002)

//два способи перевірити, чи є в Map ключ
println(colors.contains("black")) // true
println(colors("black"))       // #00000
println(colors("red"))               // na
       //оскільки є withDefaultValue

val color:(String,String)=
         "green"->"#00004"
                           //додали нову пару
val newColors:Map[String,String]=
           colors + color
println(newColors)
       // Map(black -> #00000, blue -> #00001,
       // Blue -> #00002, green -> #00004)
```

```
 println(colors.toList)
       // List((black,#00000),
       // (blue,#00001), (Blue,#00002))
println(List(("white", "#00005")).toMap)
                      //Конвертація
                      // Map(white -> #00005)
//-------------------------------------------
val color: Map[String, String] =
       Map("green" -> "#339616")
println(color.getClass.getName)
  // scala.collection.immutable.Map$Map1

val color1: (String, String) =
                    "blue" -> "#0d1ad1"
println(color1.getClass.getName)
                                 // scala.Tuple2
val newcolor:Map[String, String] =
                    color + color1
println(newcolor.getClass.getName)
println(newcolor)
 // scala.collection.immutable.Map$Map2
 // Map(green -> #339616, blue -> #0d1ad1)
```

Змінні і незмінні колекції

Scala підтримує змінні і незмінні колекції. Незмінні колекції ніколи не змінюються, тому їх без побоювання можна передавати за посиланням і використовувати навіть в багатопотоковому середовищі виконання. Наприклад, існують типи `scala.collection.mutable.Map` і `scala.collection.immutable.Map`. Обидва мають загальний супертип `scala.collection.Map`.

У мові Scala перевага віддається *незмінним* колекціям. Об'єкти-компаньйони в пакеті `scala.collection` виробляють незмінні колекції. Наприклад, `scala.collection.Map("Hello" -> 42)` – це незмінна колекція.

Порада.

Додавши інструкцію `import scala.collection.mutable` можна створювати незмінні асоціативні масиви, використовуючи псевдонім `Map`, а змінні – `mutable.Map`.

Array - масиви

```
val anArray : Array[String] =
             Array("h","e","l","l","o",".")
anArray(5) = "!" //синтаксичний цукор,
       //повна версія - anArray.update(5, "!")
println(anArray.mkString(""))   // hello!
println(anArray.mkString("-")) // h-e-l-l-o-!

val twoElements: Array[Boolean] =
             Array.ofDim[Boolean](2)
    //створюється пустий масив заданого розміру
twoElements.foreach(println)          // false
                                      // false

val numberSeq:Seq[String] = anArray
                          //перетворення типу
println(numberSeq)  // WrappedArray(h,e,l,l,o,!)
```

Tuple - кортежи

```
val aTuple:(Int,String)= (100,"Scala")
       //містить елементи різних типів
val sameTuple:(Int,String)= Tuple2(100,"Scala")
       // Tuple1, Tuple2,... Tuple22 – тобто
       // може містити до 22 елементів

println(aTuple)                 // (100,Scala)
println(aTuple._1)              // 100
println(aTuple._2)              // Scala

val copy:(Int,String) =
                   aTuple.copy(_2 = "Python")
```

```
        // копіювання, причому при копіюванні
        // можна замінити елементи
println(copy)                    // (100,Python)
println(aTuple.swap)             // (Scala,100)
```

Range - діапазон

```
val aRange:Seq[Int]= 1 until 3
                    //число 3 не буде включено
aRange.foreach(print)                      // 12

(1 to 3).foreach(x => print("Hello"))
                              // HelloHelloHello
        //можна використовувати замість рекурсії

val aRangeToVector:Vector[Int]=
                     (1 to 5).toVector
        // Range допомагає створювати колекції
println(aRangeToVector)  // Vector(1,2,3,4,5)
```

Загальні методи

foreach

```
val list = List(1,2,3)
list.foreach(print)                 // 123

for {        //код, еквівалентний foreach
n <- list
} print(n)                          // 123
```

Конструкція `for(i <- range)` забезпечує послідовне привласнення змінній `i` всіх значень діапазону.

Метод `until` класу `RichInt` повертає всі числа у вказаному діапазоні, не включаючи верхньої межі. Наприклад:

```
for (i <- 0 until 10) print(i)   // 0123456789
```

Якщо в тілі циклу індекси не потрібні, можна реалізувати обхід елементів безпосередньо:

```
for (elem <- array) print(i)
```

map, flatMap, filter

Сенс `map` полягає в тому, що задана функція застосовується *до кожного елементу* списку.

`flatMap` дуже схож на `map`, лише він перетворить кожен елемент в цілий список елементів і виконує дії вже з ними, а потім результат збирає в одне ціле.

Наприклад:

```
val fruits = List("apple", "banana")

val mapResult = fruits.map(_.toUpperCase)
val flatResult = fruits.flatMap(_.toUpperCase)

println(mapResult)  // List(APPLE, BANANA)
println(flatResult)
        // List(A, P, P, L, E, B, A, N, A, N, A)
```

Саме із-за того, як працює `flatMap`, якщо потрібно проставити крапку після кожного символу рядка і на виході отримати модифікований рядок, використовувати доведеться саме його.

Наприклад:

```
  val s = "Hello"
  val newStr: String = s.flatMap(c => (c + "."))
  println(newStr) // H.e.l.l.o.
```

`map` теж спрацює, лише поверне вже не рядок

```
  println(s.map(c => (c + ".")))
                // ArraySeq(H., e., l., l., o.)
```

Комбінуючи `map` та `flatMap`, можна дістати можливість пройтися за списком. Альтернативою є вживання `for`:

```
val list1 = List(1,2)
val list2 = List("a","b")
val combinations =
    list1.flatMap(n=>list2.map(c=>c+n))
 //кожний елемент list1 комбінуємо
 // з кожним елементом list2
println(combinations)
                    // List(a1, b1, a2, b2)

val forcombinations =   //аналогічний код з for
for{
    n<- list1
    c<- list2
  } yield c+n
  println(forcombinations)
                    // List(a1, b1, a2, b2)
```

Вище дано код с `flatMap` і з `map`. І аналогічний код, написаний з вживанням `for`. Якщо провести паралелі, можна побачити, що `yield` – альтернатива `map` служить для повернення результуючого значення.

`yield` це `append` у виразі `for`.

Цикл `for (...) yield` створить нову колекцію того ж типу, що і оригінал. Якщо у вираженні бере участь масив, на виході виходить інший масив. Масив, що вийшов, міститиме значення виразу, наступного за `yield`, обчислюваного в кожній ітерації.

***Приклад* 5**. Дан код:

```
val list = List(1, 2, 3)
val doubler = (x: Int) => List(x, x * 2)
                                        //функція
```

Що буде надруковане, при використанні наступних операторів:

1) `println(doubler(5))  // List(5, 10)`

2) `println(list.map(doubler))`
`// List(List(1, 2), List(2, 4), List(3, 6))`

3) `println(list.flatMap(doubler))`
`// List(1, 2, 2, 4, 3, 6)`

***Приклад* 6**. Допустимо, є наступний список:

```
val progLanguages =
       List("java", "scala", "python")
```

Для кожного елементу списку необхідно обчислити довжину рядка. В *результаті* повинен вийти список кортежів

`List((java,4), (scala,5), (python,6)),`

де кожен тупл складається з двох елементів: назва мови програмування і обчислена довжина рядка.

Отримати список кортежів можна таким чином:

```
val out = for {
   lng <- progLanguages
 } yield (lng, lng.length)
println(out)
         //List((java,4), (scala,5), (python,6))
```

Або, як альтернативу `for` виразу, можна використовувати наступний код:

```
 val out =
    progLanguages.map(lng => (lng, lng.length))
 println(out)
   //List((java,4), (scala,5), (python,6))
```

А тепер необхідно отримати просто список мов програмування, назви яких написані великими буквами:

`List(JAVA, SCALA, PYTHON).`

Це можна зробити наступними способами.

Спосіб 1.

```
val x:List[String] = for {
    lng <- progLanguages
  } yield lng.toUpperCase
  println(x)
        //List(JAVA, SCALA, PYTHON)
```

Спосіб 2.

```
println(progLanguages.map(_.toUpperCase))
                     //List(JAVA, SCALA, PYTHON)
```

Надрукувати у вигляді:

```
//JAVA
//SCALA
//PYTHON
```

дозволяють наступні фрагменти кодів:

```
progLanguages.map(_.toUpperCase).foreach(println)
```

```
for (lng <- progLanguages)
       println(lng.toUpperCase)
```

```
for {
   lng <- progLanguages
  } println(lng.toUpperCase)
```

***Завдання до практичної роботи* 8**

8.1 Написати не менше 5 різних анонімних функцій, що обчислюють *довжину поданого їй на вхід рядка*.

Примітка: код повинен уміщатися в одну строчку.

Sample Input: `Hello, world!` Sample Output: `13`

8.2 Написати анонімну функцію, що обчислюють *довжину поданого їй на вхід рядка.*

Примітка: використовувати перевантажений метод `apply` і рекурсію.

8.3 Дано список мов програмування і список скорочених назв цих мов:

```
val progLanguages =
              List("java", "scala", "python")
val lngAbbrev = List("JA", "SCA", "PY")
```

Потрібно отримати список

```
List((JA,java), (SCA,scala), (PY,python)),
```

у якому назва мови об'єднана лише з відповідною цієї мові скороченою назвою (умовимося, що скорочення відповідає назві мови в тому випадку, якщо перші букви абревіатури і назви збігаються).

Кінцевий результат всіх перетворень має бути записаний в змінну `out`.

8.4 Дано списки:

```
val nums1 = List(1, 2, 3)
val nums2 = List(4, 6, 7)
val nums3 = List(10, 100, 1000)
```

Написати коди програм, які дозволять отримати наступні результати:

1) використовувати `for`

```
List(50, 500, 5000, 70, 700, 7000, 80, 800, 8000,
60, 600, 6000, 80, 800, 8000, 90, 900, 9000, 70, 700,
7000, 90, 900, 9000, 100, 1000, 10000)
```

2) а) використовувати `for;`

б) використовувати *`flatMap, filter, map`*

```
List(80, 800, 8000, 90, 900, 9000, 100, 1000,
10000)
```

3) использовать *`flatMap, filter, map`*

```
List(71, 701, 7001, 72, 702, 7002, 73, 703, 7003)
```

8.5 Написати функцію

`values(fun: (Int) => Int, low: Int, high: Int),` `яка` повертає колекцію із значень у вказаному діапазоні.

Наприклад, виклик

```
println(values(x => x * x, -1, 3))
```

повинен повернути колекцію пар

```
Vector((-1,1), (0,0), (1,1), (2,4), (3,9)).
```

Контрольні запитання

1. Що таке чиста функція?
2. Що можна робити при використанні функцій в Scala?
3. Що таке анонімні функції?
4. Що таке функції вищого порядку?
5. Як виконується операція каррування для функцій вищого порядку?
6. На які три категорії поділяються всі колекції?
7. Які основні функції `Seq`?
8. Які основні функції `Set`?
9. Які основні функції `Map`?
10. Що таке змінні і незмінні колекції в Scala?
11. Які основні функції `Array`?
12. Які основні функції `Tuple`?
13. Які основні функції `Range`?
14. Який синтаксис методу `foreach`?
15. Який синтаксис методів `map`, `flatMap`, `filter`?

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Odersky M., Spoon L., Venners B. – Programming in Scala. – Artima Inc, ISBN: 0981531601, 2021. – 776 p.
2. Horstmann C.S. Scala for the Impatient. – Addison-Wesley. ISBN: 978-0-13-454056-6, 2017. – 369 p.
3. Wampler D. Programming Scala. – O'Reilly. ISBN: 978-1-492-07789-3, 2021. – 556 p.
4. S-99: Ninety-Nine Scala Problems [Електронний ресурс]. – Режим доступу: https://aperiodic.net/phil/scala/s-99/ – Дата звернення: 03.10.2023.
5. First Steps to Scala [Електронний ресурс]. – Режим доступу: https://www.artima.com/articles/first-steps-to-scala – Дата звернення: 03.10.2023.

ЗМІСТ

Навчальне видання

ЛЮБЧЕНКО Наталія Юріївна
ПОДОРОЖНЯК Андрій Олексійович
ЧЕРНИХ Олена Петрівна

**ОСНОВИ МОВИ
ПРОГРАМУВАННЯ SCALA**

Навчально-методичний посібник
для студентів комп'ютерних спеціальностей
вищих навчальних закладів

Роботу рекомендував до видання проф. Заполовський М.Й.
Відповідальний за випуск проф. Солощук М.М.

Дизайн обкладинки

В авторській редакції

План 2023 р., поз. 143
Підписано до друку 12.10.23. Формат 60x84 1/16. Папір офсет.
Друк – ризографія. Гарнітура Times New Roman. Ум. друк. арк.3,5
Наклад 20 прим. Зам №. Ціна договірна

Видавничий центр НТУ "ХПІ" 61002, Харків, вул. Кирпичова, 2.
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.