Посібник по Scala
для Java розробників

Версія 1.3

30 березня 2011 р.

Michel Schinz, Philipp Haller, пер. Віталій Яковчук

Programming Methods Laboratory
EPFL
Switzerland

1 Вступ

Цей документ дає коротку інформацію стосовно мови програмування і компілятора Scala. Документ розрахований на людей які вже мають досвід програмування і хочуть ознайомитись з можливостями Scala. Передбачається що читач знайомий з основами об'єктно-орієнтовного програмування, особливо за допомогою мови програмування Java.

2 Перший приклад

Для початку розглянемо стандартний приклад програми *Hello world* (*Привіт світ*). Даний приклад слабо показує переваги Scala, але дозволяє продемонструвати базові конструкції мови:

```
object HelloWorld {
 def main(args: Array[String]) {
  println("Hello, world!")
 }
}
```

Структура даної програми схожа на структуру програми на мові програмування Java. Програма складається з одного метода, який називається main, методу передається перелік текстових параметрів, надісланих програмі за допомогою командної стрічки. Тіло метода містить один виклик стандартного метода println, якому передається привітальне повідомлення. Метод main не повертає ніякого значення, тому немає сенсу вказувати тип, що повертається даним методом.

Набагато менш знайомим для Java програмістів тут, являється оголошення так званого *одиничного об'єкта* за допомогою ключового слова **object** - класу, що має одну реалізацію. Таким чином, конструкція **object** оголошує одночасно і клас HelloWorld і його реалізацію HelloWorld. Реалізація створюється, тоді, коли в ній є потреба - під час першого звернення до *одиничного об'єкта*.

Уважний читач міг звернути увагу, що метод main не був оголошений статичним. Справа в тому, що в Scala відсутні статичні члени (поля і методи). Для створення статичних членів їх необхідно оголосити всередині *одиничного об'єкта*.

2.1 Компіляція прикладу

Для компіляції Scala програм використовується утиліта `scalac`. Утиліта `scalac` працює як і більшість інших компіляторів. З командної стрічки їй передається шлях до файла з початковим кодом і, при необхідності, додаткові пара-

метри. В результаті виконання компіляції, `scalac` створить один або декілька об'єктних (бінарних) файлів. Об'єктні файли являються стандартними **class** файлами Java.

Якщо ми збережемо попередній приклад в файл HelloWorld.scala, то зможемо відкомпілювати його за допомогою наступної команди (тут символ більше '>' позначає початок введення команди в консолі).

```
> scalac HelloWorld.scala
```

Дана команда сформує декілька `class` файлів, в поточній директорії. Один з цих файлів буде називатись `HelloWorld.class`. Даний файл може бути використаний командою `scala` для запуску відкомпільованої програми, як це показано в наступному пункті.

2.2 Запуск прикладу

Після того, як приклад відкомпільовано, його можна запускати, за допомогою команди `scala`. Використання команди `scala` дуже схоже на використання команди `java`, яка використовується для запуску Java програм і приймає такі ж параметри. Наступний приклад показує, як запустити відкомпільовану програму, яка в результаті виведе привітальний текст.

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Взаємодія з Java

Дуже важлива особливість Scala полягає в простій інтеграції програмного коду з бібліотеками Java. Всі класи з пакету `java.lang` по замовчуванню імпортуються, тоді як інші пакети Java, необхідно імпортувати окремо.

Давайте розглянемо приклад, який демонструє імпорт пакетів Java. Нехай нам необхідно дізнатися поточну дату і відформатувати її згідно специфікацій певної країна, скажімо Франції[1].

Стандартні бібліотеки Java надають широкі можливості, для роботи з датами, зокрема, за допомогою класів `Date` і `DateFormat`. Так як Scala тісно інтегрована з Java, не має жодної необхідності створювати аналогічні класи в Scala, достатньо просто імпортувати зазначені класи з бібліотеки Java:

```
import java.util.{Date, Locale}
```

[1]Слід зазначити, що таке форматування може використовуватись і на інших територіях, крім Франції, де використовується французька мова.

```
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
 def main(args: Array[String]) {
  val now = new Date
  val df = getDateInstance(LONG, Locale.FRANCE)
  println(df format now)
 }
}
```

Конструкція призначена для імпортування класів в Scala, виглядає схоже на аналогічну конструкцію в Java, проте Scala надає більш потужний механізм імпорту класів. Декілька класів одного пакету, можуть бути імпортовані шляхом об'єднання в фігурні дужки, як це було в першому рядку. При цьому, якщо необхідно, імпортувати декілька класів одного пакету, або декілька статичних членів певного класу в мові програмування Scala використовується символ (_), замість аналогічного символу (*) в мові програмування Java. Це пов'язано з тим, що в Scala символ (*) може бути використаний в якості ідентифікатора (наприклад, за допомогою даного символу можна називати методи, що описано нижче).

Важлива конструкція імпортування задана в третьому рядку. З її допомогою були імпортовані статичні члени класу `DateFormat`. Після зазначення такої конструкції, в програмі можна використовувати статичний метод `getDateInstance` і статичне поле `LONG`, класу `DateFormat` на пряму[2].

Всередині методу `main` ми спочатку створили реалізацію Java-класу `Date`, який по замовчуванню містить поточну дату. Після цього ми створили реалізацію класу `DateFormat` для форматування дат, використовуючи статичний метод `getDateInstance`, який в свою чергу було імпортовано раніше. І нарешті, ми вивели поточну дату згідно відповідних французьких правил. Рядок `println(df format now)` демонструє цікаву особливість синтаксису мови програмування Scala: для виклику методів з одним аргументом можна використовувати спрощену конструкцію. Вираз:

```
df format now
```

являється менш громіздкою альтернативою виразу:

```
df.format(now)
```

На перший погляд, така деталь може здатись маловажливою, але далі буде продемонстровано, які важливі особливості можна отримати з її допомогою.

[2]В Java, починаючи з версії 1.5, була введена аналогічна конструкція **import static**, що дозволяє імпортувати статичні члени класів.

На завершення даного пункту слід зазначити, що класи в Scala можуть наслідуватись від класів Java, а також реалізовувати інтерфейси Java.

4 Все об'єкт

Scala повністю об'єктно-орієнтовна мова, в тому сенсі, що *все* в ній являється об'єктом, включачи числа і функції. Дана особливість відмінна від Java тим, що в Java присутні примітивні типи, такі як **int** або **boolean**, а методами в Java не можна маніпулювати як змінними.

4.1 Числа це об'єкти

Так як числа являються об'єктами, вони теж можуть мати свої методи. Фактично, арифметичні вирази, подібні до наступної:

```
1 + 2 * 3 / x
```

складаються виключно з викликів методів, так як даний вираз повністю еквівалентний наступному виразу:

```
(1).+(((2).*(3))./(x))
```

Це також означає, що символи +, -, *, / можна використовувати в ідентифікаторах (назвах змінних, методів, тощо).

Дужки навколо чисел обов'язкові, так як синтаксис Scala передбачає використання найдовшого співпадання записів елементарних елементів. Мається на увазі, що запис 1. буде інтерпретований, не як ціле число і крапка (крапка, що розділяє об'єкт і метод), а як дійсне число 1.0. Тому запис:

```
(1).+(2)
```

дозволяє уникнути ситуації, інтерпретації цілого числа 1 (з типом `Int`), в кості дійсного числа 1.0 (з типом `Double`).

4.2 Функції це об'єкти

Java програмісти мабуть буду здивовані, дізнавшись що функції в Scala також являються об'єктами. Таким чином, функції можуть бути передані як аргументи, збереженні в змінних, повернуті, як результат. Така можливість являється основою дуже цікавої парадигми програмування – функціонального програмування.

Для того щоб пояснити корисність використання функцій в якості об'єктів, розглянемо тривіальний приклад функції-таймеру. Мета даної функції викликати певний програмний код кожну секунду. Логічно припустити, що

такий програмний код варто помістити в певну функцію, яку в свою чергу зручно передати функції-таймеру в якості параметру. Багато розробників могли зіткатись з подібними задачами, особливо в під час розробки графічних додатків, де слід було зареєструвати call-back[3], для того, щоб виконувати певний код, якщо наступить певна подія.

В наступному прикладі функція-таймер називається onePerSecond, їй передається функція, яка буде викликатись кожну секунду. Тип функції, що передається записується наспупним чином: () => Unit, він являється типом всіх функцій, що не мають параметрів і не повертають жодного результату (тип Unit подібний типу **void**) в C/C++).

Головна функція даної програми просто викликає функцію-таймер передаючи їй в якості параметру іншу функцію, яка виводить на екран повідомлення "time flies like an arrow...", яке в результаті цього буде виводитись кожну секунду.

```
object Timer {
 def oncePerSecond(callback: () ⇒ Unit) {
  while (true) { callback(); Thread sleep 1000 }
 }
 def timeFlies() {
  println("time flies like an arrow...")
 }
 def main(args: Array[String]) {
  oncePerSecond(timeFlies)
 }
}
```

Слід відмітити, що для виводу повідомлення на екран був використаний вже оголошений Scala метод println, замість використання аналогічного методу з System.out.

4.2.1 Анонімні функції

Наведений вище приклад програми можна спростити, при цьому не ускладнюючи розуміння принципу її роботи. Перш за все, слід зазначити, що функція timeFlies була оголошена лише для того, щоб потім бути переданою функції oncePerSecond в якості параметру. Маючи функцію, назва якої була використана лише один раз, можливо, можна було б взагалі уникнути необхідності в такій назві. В Scala таке можливо за допомогою використання анонімних функцій: функцій, що не мають назви. Версія програми, що вико-

[3]Близький переклад терміну *call-back*: "Зворотній виклик" – функція що викликається "ззовні" або перехоплювач якогось сигналу. Наприклад, в мережевих додатках під терміном call-back, як правило, розуміють виклик якоїсь функції на стороні клієнта, по "ініціаціативі" сервера.

ристовує анонімну функцію замість оголошеної функції timeFlies буде мати наступний вигляд:

```
object TimerAnonymous {
 def oncePerSecond(callback: () ⇒ Unit) {
  while (true) { callback(); Thread sleep 1000 }
 }
 def main(args: Array[String]) {
  oncePerSecond(() ⇒
   println("time flies like an arrow..."))
 }
}
```

Наявність анонімної функції задається за допомогою стрілки направленої вправо '=>', яка відділяє перелік аргументів функції від її тіла. В даному прикладі аргументи функції відсутні, це задано оголошенням пустої пари дужок зліва від стрілки. Тіло функції ідентичне до того, яким воно було і в функції timeFlies в попередній версії програми.

5 Класи

Як було показано раніше, Scala- об'єктно орієнтовна мова програмування, тому в ній присутнє поняття класу [4]. Класи в Scala оголошуються за допомогою конструкції близької до синтаксису Java. Важлива відмінність оголошення: класи в Scala можуть мати параметри, що проілюстровано в наступному коді на прикладі класу комплексного числа:

```
class Complex(real: Double, imaginary: Double) {
 def re() = real
 def im() = imaginary
}
```

Даному класу в якості параметрів передаються два аргументи, які відповідають за дійсну і уявну складові комплексного числа. Дані аргументи мають бути передані при створені реалізації класу Complex, наступним чином: new Complex(1.5, 2.3). Клас містить два методи (re та im) за допомогою яких можна отримати доступ до складових комплексного числа.

Слід відмітити, що в даному випадку тип результату функції не був заданий явним чином. Тип буде визначений автоматично компілятором, який використовує інформацію з правої сторони виразу для того, щоб позначити обидва методи з типом результату Double.

[4] Для повноти слід зазначити, що існують об'єктно-орієнтовні мови програмування в яких не має поняття класу, Scala не відноситься до таких мов.

Компілятор не завжди може визначити тип результату, як це робиться в даному випадку і, на жаль, не існує простих правил, за допомогою яких можна точно зрозуміти, чи визначить компілятор тип результату автоматично чи ні. Але на практиці, зазвичай, це не проблема так як компілятор видає повідомлення про помилку, у випадку коли тип результату метода визначити неможливо. Для початківців, можна дати наступне правило: завжди намагатись не задавати тип результату метода в подібних випадках. З досвідом програмісти починають відчувати в яких випадках компілятор може автоматично визначити тип результату методу, а в яких ні і, відповідно, коли саме слід цей тип задавати явно.

5.1 Методи без аргументів

Невелика проблема методів im та re полягає в тому, що кожного разу коли ми хочемо викликати дані методи, нам необхідно поставити пусті дужки, як це показано в наступному прикладі:

```
object ComplexNumbers {
 def main(args: Array[String]) {
  val c = new Complex(1.2, 3.4)
  println("imaginary part: " + c.im())
 }
}
```

Було б краще, якби ми могли мати доступ до дійсної і уявної частини так, ніби ми маємо доступ до полів. В Scala є можливість оголошувати методи без списку аргументів. Відмінність таких методів від методів з пустим переліком аргументів полягає в тому, що такі методи не мають пустих дужок після їх назви, як в частині їх оголошення так і в частині виклику. Наш клас Complex може бути переписаний наступним чином:

```
class Complex(real: Double, imaginary: Double) {
 def re = real
 def im = imaginary
}
```

5.2 Наслідування і заміщення

Всі класи в Scala наслідуються від батьківського класу. Якщо батьківський клас не вказаний, як це було в попередньому прикладі з класом Complex, то в якості батьківського класу використовується клас scala.AnyRef.

Методи, що були присутні в батьківському класі можна замістити [5]. При цьому заміщення в Scala обов'язково має бути позначене модифікатором методу

[5]Заміщення, один з прийомів, що відповідає концепції поліморфізму в об'єктно-орієнтовному програмуванні.

override. Такий підхід використовується для того, щоб уникнути випадкового заміщення методів. Для прикладу додамо в наш клас Complex метод toString, який буде заміщати відповідний методу класу Object:

```
class Complex(real: Double, imaginary: Double) {
 def re = real
 def im = imaginary
 override def toString() =
  "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Класи-випадки і співпадіння за зразком

В програмах часто використовується структура даних: дерево. Для прикладу інтерпретатори і компілятори, як правило, обробляють програму як дерево; XML документи являються деревами; різні види контейнерів являються деревами, наприклад, "червоно-чорні" дерева.

Далі буде описано, як такі дерева використовуються в Scala на прикладі програми – простого калькулятора. Ціль цієї програми обробляти прості арифметичні вирази, які складаються з суми, цілочисельних констант і змінних. Прості вирази можуть виглядати наступним чином: $1+2$ або $(x+x)+(7+y)$.

Перш за, все ми маємо визначити яким чином обробляти подібні вирази. Найбільш зручно представляти такі вирази у вигляді дерева, де гілками дерева будуть операції (в даному випадку додавання) а листками дерева будуть змінні (в даному випадку константи або змінні).

В Java такі структури зручно було б представити у вигляді абстрактного батьківського класу для елементів дерева, а також за допомогою дочірніх класів для гілок і листків. В функціональних мовах програмування варто було б використати структури для відповідних алгебраїчних елементів. Scala надає концепцію *класів-випадків*, які являються чимось середнім між зазначеними підходами. Наступний приклад демонструє як класи-випадки можуть бути використані для оголошення типів для побудови дерева:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Оголошення класів Sum, Var і Const в якості класів-випадків, означає що ці класи мають ряд відмінностей порівняно зі стандартними класами:

- Ключове слово **new** не являється обов'язковим для створення реалізації класу (тобто можна написати Const(5) замість **new** Const(5));

- Функції для отримання параметрів конструктора автоматично присутні в такому класі (тобто можна отримати параметр конструктора v для деякої реалізації класу c використовуючи запис c.v);
- По замовчуванню в класі автоматично присутні методи equals і hashCode, які використовують дані структури, а не ідентифікатори об'єктів (внутрішні посилання на об'єкти);
- По замовчуванню в класі присутній метод toString, який друкує значення змінної в "початковій формі" (тобто дерево для виразу $x + 1$ буде надруковано у вигляді Sum(Var(x),Const(1)));
- Реалізації класів-випадків можуть бути розкладені за допомогою "співпадіння за зразком", як це буде показано далі.

Ми маємо тип даних, що відповідає заявленим арифметичним операціям, тепер можна реалізовувати операції для маніпуляції з ними. Спочатку реалізуємо функцію що буде розраховувати вираз в деякому *середовищі* (середовищі змінних). Мета середовища: інтерпритувати змінну в її значення. Наприклад, вираз $x + 1$, що розраховується в середовищі, яке в свою чергу асоціює зі змінною x значення 5, за допомогою запису $\{x \to 5\}$, дасть в результаті нам 6.

І так, на потрібно вибрати шлях, для реалізації середовища. Звичайно, ми могли б для цього використати якусь асоціативну структуру, наприклад, хеш-таблицю, але в даному випадку ми можемо використати функції на пряму! Середовище, це лише функція, яка ставить у відповідність до назви змінної її значення. Середовище $\{x \to 5\}$ показане вище в Scala може бути записане наступним чином:

```
{ case "x" ⇒ 5 }
```

Даний запис оголошує функцію яка, якщо їй буде передане значення x"" в якості аргументу поверне ціле число 5, інакше зазнає невдачі виконання і згенерує виключну ситуацію.

Перед тим як реалізовувати функцію по розрахунку виразів, задамо ім'я для середовищ зі змінною. Звичайно ми завжди можемо використовувати тип String => Int для середовищ, але оголошення цього типу зі своєю окремою назвою спростить програму і дасть змогу більш простими шляхами вносити подальші зміни. Оголошення такого типу в Scala буде виглядати наступним чином:

```
type Environment = String ⇒ Int
```

Тепер тип Environment може використовуватись в якості псевдоніма для функцій що перетворюють значення типу String в Int, для повернення цілочисельного значення змінної за допомогою її текстового позначення.

Тепер ми можемо реалізувати функцію, що розраховує значення виразів. Концептуально це дуже просто: значення суми двох виразів рівне сумі значень цих виразів; значення змінних отримуються з деякого середовища и нарешті, значення константи рівне цій константі. Записати це все за допомогою Scala так само просто:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r)      => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)       => v
}
```

Функція розрахунку виразу працює використовуючи *співпадіння за зразком* в дереві t. Коротко, даний програмний код значить наступне:

1. Спочатку відбувається перевірка чи дерево t являється реалізацією класу Sum, якщо так, то ліве піддерево (лівий доданок) зв'язується зі змінною l, а праве піддерево зі змінною r. Далі відбувається обробка зв'язаних зі змінними r і l піддерев по правилу, що знаходиться справа від стрілки '=>', а саме піддерева знову обробляються як дерева, а результат обробки сумується.

2. Якщо перша перевірка хибна (дерево t не являється реалізацією класу Sum), відбувається перевірка чи дерево t являється реалізацією класу Var, якщо так то результат обробки буде повернення значення змінної за її назвою.

3. Остання умова виконається якщо дерево t являється реалізацією класу Const, в такому випадку результатом обробки буде константа виразу v.

4. І нарешті, якщо жодна умова з трьох не виконається згенерується виключна ситуація, таке може статись лишу якщо від класу Tree наслідується ще якийсь клас крім трьох передбачених класів у нашому прикладі.

Головна ідея співпадіння за зразком: порівняти змінну до набору можливих класів-зразків, і виконати той програмний код, який може використовувати поля класу-зразку.

В мові програмування Java цей приклад можна було б реалізувати зробивши клас Tree абстрактним з абстрактним методом eval, додавши свою окрему реалізацію методу eval для підкласів Sum, Var і Const. Досвідчений програміст, звикший використовувати об'єктно-орієнтований підхід, може здивуватись, чому в даному випадку не був використаний підхід з абстрактним класом. І дійсно, ми могли б реалізувати даний приклад з використанням абстрактного методу, так як Scala підтримує оголошення методів в класах-випадках, так само як і в "нормальних" класах. Вибір чи використовувати співпадіння

за зразком чи методів в великій мірі справа смаку, але разом з тим це суттєво впливає на можливість подальшого розширення початкового програмного коду:

- використання методів дозволяє легко додавати нові типи гілок з новою логікою поведінки (наприклад, до нашого прикладу в мабутньому ми захочемо додати функцію віднімання). Для цього нам треба буде створити новий підклас класа Tree, з іншого боку для для додавання нових функцій для маніпуляцій з деревом потребуватиме зміни всіх підкласів класу Tree (наприклад, якщо ми захочемо в майбутньому мати функцію спрощення виразу);
- співпадіння за зразком напроти потребуватиме зміни всіх частин де використовувалась дана конструкція у випадку, якщо ми захочемо додати новий тип гілки, але у випадку, якщо ми захочемо мати нові функції по обробці вже існуючих гілок, нам не треба буде міняти всі підкласи класу Tree.

Спробуємо реалізувати функцію пошуку похідної наших виразів. Читач може пам'ятати наступні правила пошуку похідних:

- похідна суми рівна сумі похідних;
- похідна змінної v рівна 1 одиниці, якщо похідна береться по v інакше рівна нулю;
- похідна константи рівна нулю.

Ці правила можуть бути перекладені на мову програмування Scala майже буквально за допомогою наступної функція:

```
def derive(t: Tree, v: String): Tree = t match {
 case Sum(l, r) => Sum(derive(l, v), derive(r, v))
 case Var(n) if (v == n) => Const(1)
 case _ => Const(0)
}
```

Ця функція демонструє дві нові концепції пов'язані зі співпадінням за зразком. Перш за все, вираз **case** перевідряється за допомогою ключового слова **if**. Дана перевірка запобігає виконанню співпадінню до тих пір, поки відповідний вираз не рівний **true**. В данному випадку використовується перевірка, для того щоб перетворити похідну змінною в одиницю тільки у випадку якщо похідна береться по цій же змінній. Також, в даній конструкції використовується обробка непередбаченого вище випадку за допомогою знаку _.

Ми не дослідили всієї потужності використання співпадіння за зразком, але зупинимось на даному прикладі, щоб зберегти даний посібник невеликим.

Перейдемо відразу до демонстрації використання двох описаних функції на реальному прикладі. Давайте напишемо невелику функцію main яка здійснює обробку виразу $(x + x) + (7 + y)$ різним чином: спочатку порахуємо значення виразу, для $x \to 5, y \to 7$, потім візьмемо похідну виразу по x і по y.

```
def main(args: Array[String]) {
 val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
 val env: Environment = { case "x" ⇒ 5 case "y" ⇒ 7 }
 println("Expression: " + exp)
 println("Evaluation with x=5, y=7: " + eval(exp, env))
 println("Derivative relative to x:\n " + derive(exp, "x"))
 println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Виконавши дану програму, ми отримаємо наступний вивід:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Як бачимо, вигляд виразів маже бути спрощений перед виводом. Реалізація функції по спрощенню виразу перед друком, використовуючи співпадіння за зразком цікава задача і потребує специфічної винахідливості. Дана задача залишена читачу в якості вправи.

7 Типажі

Окрім наслідування програмного коду від батьківського класу в мові програмування Scala є можливість імпорту програмного коду з типажів [6] .

Найпростіший спосіб Java-програмісту зрозуміти, що таке типаж, уявити собі інтерфейс, який містить методи разом з їхньою реалізацією. В Scala, якщо клас наслідується від типажу він одночасно реалізує методи інтерфейсу типажу, а також імпортує програмний код описаний в типажі.

Щоб продемонструвати користь від використання типажів, розглянемо класичний випадок об'єктів, які можна відсортувати. Такі класи часто дуже зручно безпосередньо порівнювати між собою. В Java об'єкти, які можна

[6] Слово типаж запропоноване, як найближчий варіант перекладу англійського слова trait.

порівнювати реалізують інтерфейс Comparable. В Scala можливо дещо краще реалізувати поведінку порівняння об'єктів за допомогою типажів, ніж це зазвичай робиться в Java. Для цього створемо типаж з назвою Ord.

Для порівняння об'єктів зручно використовувати шість різних предикатів: менше, менше рівно, рівно, не рівно, більше рівно і, більше. Однак кожного разу реалізовувати шість методів досить громіздко в ряді випадків, особливо якщо врахувати, що чотири з них можуть бути реалізовані двох інших. Наприклад предикат рівно і менше, може бути використано для означення решти чотирьох предикатів. В Scala даний випадок можна помістити в один типаж наступним чином:

```
trait Ord{
 def < (that:Any): Boolean
 def <=(that:Any): Boolean = (this < that) || (this == that)
 def > (that:Any): Boolean = !(this <= that)
 def >=(that:Any): Boolean = !(this < that)
}
```

Таке оголошення створює новий тип з назвою Ord, який виконує одночасно роль Java інтерфейсу Comparable і реалізовуючи три предиката з чотирьох, один з яких абстрактний. Предикати рівності і нерівності в даному випадку не оголошуються так як вони вже присутні в усіх об'єктах.

Використаний вище тип Any, являється батьківським типом для всіх типів в мові програмування Scala. Даний тип більш загальна версія класу Object в Java, так як він загальний в тому числі для простих типів таких, як Int, Float і інших.

Щоб зробити об'єкти класу такими, що можуть порівнюватись в класі достатньо реалізувати абстрактний метод-предикат "менше", а також унаслідувати клас від типажу Ord. Для прикладу, створемо клас Date для збереження дат Григоріанського календаря. Такі дати будуть складатись з дня, місяця і року, які будуть зберігатись в цілочисельних полях. Тому оголосимо типаж наступним чином:

```
 class Date(y: Int, m: Int, d: Int) extends Ord {
 def year = y
 def month = m
 def day = d

 override def toString(): String = year + "-" + month + "-" + day
```

Вижливо звернути увагу на запис **extends** Ord, який задає, що клас Date наслідується від типажу Ord.

Тепер ми переоголосимо метод equals з Java класу Object і реалізуємо в ньому порівняння всі полів класу Date. Стандартний метод equals в даному випадку

нам не підходить, так як в Java він порівнює фізичні адреси об'єктів:

```
override def equals(that: Any): Boolean =
 that.isInstanceOf[Date] && {
  val o = that.asInstanceOf[Date]
   o.day == day && o.month == month && o.year == year
}
```

Даний метод використовує стандартні методи Scala isInstanceOf і asInstanceOf. Перший isInstanceOf аналогія оператору Java s isInstanceinstanceof, який повертає **true** у випадку, якщо об'єкт наслідується від класу переданого в квадратних дужка. Метод asInstanceOf це приведення об'єкту, для якого викликається метод до типу переданого в квадратних дужках. Якщо яб'єкт не може бути приведений до відповідного типу, то згенерується виключна ситуація ClassCastException, для цього ми й перевіряли спочатку чи наслідується переданий об'єкт від класу Date.

Нарешті, останній метод, який нам треба реалізувати, це предикат "менше". Даний метод використовує інший стандартний метод об'єкта для мови програмування Scala error, який генерує виключну ситуацію з переданим текстом повідомленням про помилку.

```
def <(that: Any): Boolean = {
 if (!that.isInstanceOf[Date])
  error("cannot compare " + that + " and a Date")

 val o = that.asInstanceOf[Date]
 (year < o.year) ||
 (year == o.year && (month < o.month ||
          (month == o.month && day < o.day)))
}
```

Даний метод завершує реалізацію нашого класу Date. Тепер всі об'єкти класу Date можуть порівнюватись за допомогою шести різних операцій порівняння, два з яких (рівно та менше) ми реалізували в класі Date, а решта чотири унаслідувались від типажу Ord.

Звичайно, типажі можуть використовуватись в ряді інших випадків крім показаного тут, але це виходить за рамки розмірів накладених на даний документ.

8 Узагальнене програмування

На останок в даному документі буде розглянута можливість використання узагальненого програмування в Scala. Java програмісти можуть бути знайомі з узагальненими класами (generic класами) в Java.

Узагальнене програмування це методика, яка дозволяє писати програмний код, що може бути в результаті застосований до різних типів даних. Наприклад програміст може створити універсальний клас-список, для збереження списку певних об'єктів. При цьому даний клас хотілось би використовувати не для одного типу даних (скажімо Int), а для цілого набору різних типів даних.

Java програмісти можуть вирішити дану проблему використавши обробку об'єктів працюючи з ними, як з об'єктами класу Object [7], при цьому такий підхід досить далекий від ідеального, так як він всерівно не дозволяє працювати з простими типами (int, float і так далі).

Scala дозволяє використовувати узагальнені класи (і методи), для вирішення вищеокреслених задач. Розглянемо простий клас-контейнер: посилання на об'єкт, яке може бути або пустим, або посилатись на об'єкт довільного типу.

```
class Reference[T] {
 private var contents: T = _

 def set(value: T) { contents = value }
 def get: T = contents
}
```

Клас Reference використовує параметр тиру T, який являється узагальненим типом, що використовується в межах класу, зокрема як параметр функції set і результат функції get.

Даний приклад програмного коду використовує оголошення змінної, якій присвоюється значення _ - значення по замовчуванню. Це значення по замовчуванню рівне 0 для числових типів, **false** для типу Boolean, () для типу Unit і **null** для всіх інших типів.

Для використання класу Reference, необхідно вказати, який саме тип буде використовуватись для параметру T. Для прикладу, щоб зберегти цілочисельну змінну в нашому контейнері, можна продемонструвати наступний програмний код:

```
object IntegerReference {
 def main(args: Array[String]) {
  val cell = new Reference[Int]
  cell.set(13)
  println("Reference contains the half of " + (cell.get * 2))
 }
}
```

[7] В Java починаючи з версії 1.5 також з'явилась можливість використання узагальненого програмування, воно реалізоване лише на рівні мови, тобто будь-який узагальнений клас після компіляції програмного коду розглядається як клас Object, при цьому компілятор автоматично приводить тип об'єкта до потрібного нам класу, де це потрібно

Як видно з прикладу, немає необхідності приводити тип змінної поверненої методом get, для того щоб використовувати її як ціле число. Так само неможливо зберегти в контейнері cell нічого крім цілого числа, так як даний контейнер був оголошений таким, що використовує цілочисельний тип.

9 Підсумок

Даний документ дав вам короткий огляд мови програмування Scala і показав декілька простих прикладів. Зацікавлений користувач може ознайомитись з супутнім документом *Scala By Example*, де розглянуті більш складні приклади. Детальну інформацію про мову програмування Scala можна отримати в документі *Scala Language Specification*.