El lenguaje *Scala*

Salvador Carrillo Fuentes, Ricardo Holthausen Bermejo

Enero 2019

1 ¿Qué es Scala?

Scala es un lenguaje de programación de propósito general. Se trata de un lenguaje estrechamente relacionado con *Java*, puesto que, al igual que éste, es compilado a *Java Bytecode*.

Scala viene a poner en relación dos paradigmas de la programación: La programación orientada a objetos (OOP, por sus siglas en inglés) y la programación funcional (FP, por sus siglas en inglés). Con respecto al primero de ellos, *Scala* es un lenguaje de OOP puro, en el sentido de que todos los valores son objetos (por ejemplo, los tipos primitivos y las funciones).

En el caso de la FP, *Scala* incluye características relacionadas con este paradigma, como pueden ser:

- No distinción entre instrucciones y expresiones (todo son expresiones en *Scala*. En caso de que la expresión no evalúe a nada, devuelve un objeto `Unit`).
- Inferencia de tipos.
- Inmutabilidad de variables (`val` se emplea para declarar valores fijos).
- Evaluación perezosa
- Funciones de orden superior: Las funciones pueden recibir otras funciones como parámetros y devolverlas como resultados.
- Funciones anidadas.
- *Currying*

El nombre de *Scala* proviene de "***Sca****lable* ***La****nguage*", es decir, lenguaje escalable. Esto se debe a que la idea principal que tenía Martin Odersky (su creador) cuando lo desarrollaba era precisamente que fuese un lenguaje de programación que creciese con las necesidades de sus usuarios.*Scala* comenzó a desarrollarse en la École Polytechnique Fédérale de Lausanne (EPFL) (Lausanne, Suiza) en 2001, aunque no fue publicado hasta 2004.

Scala es un lenguaje de tipado estático, que se refiere a que una variable está ligada a un tipo particular durante su tiempo de vida. No se le puede cambiar el tipo y solo puede tener valores compatibles con ese tipo. En un lenguaje de tipado dinámico, se liga el tipo con el valor actual referenciado por la variable.

Uno de los propósitos que persigue *Scala* es "conseguir más con menos líneas de código". *Java* puede ser muy redundante en determinadas ocasiones. Características como la inferencia de tipos, o la propia sintaxis de *Scala* minimizan el exceso de información innecesaria.

El sistema de tipos de *Scala* extiende al de *Java*, siendo más flexible y añadiendo construcciones más avanzadas. El hecho de que *Scala* se compile a *javabytecode* y se ejecute en la *Java Virtual Machine* (*JVM*) proporciona un buen rendimiento a la par que la ventaja de la portabilidad. Por otro lado, la existencia de diferentes entornos de desarrollo y extensiones para editores de textos son indicativos de la actividad de la comunidad detrás del lenguaje, y de la búsqueda por la sencillez y eficiencia a la hora de trabajar con *Scala*. Además, soporta concurrencia y, debido a la posibilidad de aplicar el paradigma funcional, se puede eliminar la necesidad de sincronizar el acceso a estados mutables.

2 Bases de Scala

En el siguiente apartado se procederá a explicar determinados elementos básicos de *Scala*, como son su jerarquía de tipos, su sintaxis, el enfoque que tiene con respecto a la OOP, y su relación con la máquina virtual de *Java*.

2.1 Jerarquía de tipos

En *Scala*, los tipos `Any`, `AnyRef` y `AnyVal` son la raíz de la jerarquía de tipos, siendo `Any` la raíz absoluta. De `AnyVal` descienden los tipos `value`, es decir, aquellos que son la base para representar datos (los tipos numéricos, `Char`, `Boolean` y `Unit`). Los tipos que heredan de `AnyRef` son las colecciones, las clases y los *strings*. En lo más bajo de la jerarquía se encuentran los tipos `Nothing` y `Null`. `Nothing` es subtipo de todos los demás subtipos, y `Null` es subtipo de todos los tipos `AnyRef`.

2.2 Sintaxis

Se procede a exponer los elementos más representativos de la sintaxis de *Scala* comparándolos con la de *Java*, en primer lugar por la relación que existe entre ellos, y en segundo lugar porque se trata de uno de los lenguajes más extendidos. Las características más representativas de la sintaxis de *Scala* son:

- No se hace uso del punto y coma.
- La primera letra de los tipos primitivos se escribe en mayúscula (`Int`, `Double`, ...)

- La definición de los métodos se realiza con la partícula `def`.
- Los tipos de los parámetros y las salidas de las funciones se escriben de forma similar al lenguaje *Pascal* (`nombreVar: TipoVar`).
- Dada la inferencia de tipos que lleva a cabo el compilador de *Scala*, para declarar variables y valores fijos se emplean las partículas `var` y `val`, respectivamente.
- No es necesario devolver (`return`) explícitamente al final de las funciones; La última expresión evaluada es la que devuelve la función.
- No existe el `void`. Lo sustituye la clase `Unit`.
- Los tipos genéricos se colocan entre corchetes, en lugar de paréntesis angulares.

3 Programación orientada a objetos en Scala

Como bien es sabido, el elemento central en la OOP son las clases. En *Scala*, para crear una clase empleamos la partícula `class` seguida del nombre de la clase y, opcionalmente y entre paréntesis, los parámetros que usará el constructor. En caso de que queramos que estos parámetros pasen a ser campos de la clase (es decir, que puedan luego ser utilizados por los métodos), deben ir precedidos por las partículas `val` o `var`.

```
class <identifier> [extends <identifier>] [ fields,
              methods, and classes ]
```

```
class Student(n: String) {
   val name: String = n
   def greeting = s"Hi, I am $name"
   override def toString = s"Student($name)"
}
```

3.1 Notación para métodos

Scala ofrece azúcar sintáctico en cuanto a la notación para llamar métodos. Esto hace que los programas tengan más parecido al lenguaje natural y sea más fácil la tarea de programar.

En *Scala* todos los operadores son métodos. Es decir, $1 + 2$ es equivalente a `1.+(2)`. *Scala* ofrece notación infija (solo para métodos con 1 parámetro), prefija (para operadores unarios) y postfija (para métodos sin parámetros).

Podemos definir operadores unarios con `unary_<operador>`, siendo el operador alguno de entre `{+, -, ~, !}`. El método `apply()` es muy importante en *Scala* y es el que rompe las barreras entre la OOP y la FP. Este método

permite llamar a las instancias de una clase como si fueran funciones. Se pueden ver ejemplos en el fuente *MethodNotation.scala*.

3.2 Objetos en Scala

`Object` es una sentencia bastante característica de *Scala*. Existen dos tipos de `Object`. En primer lugar, los `Standalone Object` son un tipo especial de clase que sólo tienen una instancia. Vendría a ser, pues, el patrón de diseño *Singleton* (sin tener nosotros que desarrollar nada). En segundo lugar encontramos los `Companion Object`. En este caso, este object viene a cubrir la tarea que llevan a cabo los miembros `static` de una clase en *Java*.

Una clase y un objecto con el mismo nombre pueden acceder mutuamente a sus miembros privados. De forma casi análoga al caso anterior, podemos implementar otro patrón de diseño de forma bastante sencilla. El patrón en cuestión es la factoría (*Factory*). Es decir, dadas varias clases que heredan de una misma, podemos ahorrarnos el crear distintos constructores para cada una de ellas si creamos una factoría. Se pueden ver ejemplos en el fuente *ObjectsEnScala.scala*.

3.3 Herencia

Scala soporta herencia simple, es decir, solo se puede extender una única clase. Al usar la *keyword* `extends`, se heredan todos los campos y métodos no privados. Los miembros `private` sólo son accesibles de la propia clase, los `protected` son accesibles dentro de la clase y las subclases y aquellos por defecto son declarados `public`, accesibles sin restricción.

Al extender una clase con parámetros, se debe proporcionar el constructor de la superclase, ya que la *JVM* llama a este constructor primero. Es posible hacer *override* de campos y métodos en el cuerpo de la clase, aunque también en el constructor en el caso de los campos. En casos de polimorfismo, se ejecutan los métodos de las subclases primero. Se puede usar la *keyword* `super` para hacer referencia a la superclase.

Es posible prevenir los *overrides* de tres maneras:

1. Declarando como `final` un miembro
2. Declarando como `final` la clase entera (no podrá ser extendida)
3. Sellando la clase (`sealed`), permitiendo la herencia dentro del archivo en el que se encuentra pero no desde otros archivos.

Se pueden consultar los ejemplos en el fuente *Inheritance.scala*.

3.4 Clases abstractas y *traits*

Se pueden declarar clases parcialmente implementadas (abstractas) en *Scala*. Estas clases, naturalmente, no pueden ser instanciadas puesto que una subclase debe implementar los métodos y rellenar los campos que estén vacíos.

Por otro lado, están los `traits`, que al contrario que las clases, no permiten el paso de parámetros en el constructor. Además, la diferencia más notable entre clases abstractas y `traits` es que una clase puede heredar de múltiples `traits`, mientras que sólo puede tener una superclase. Como norma informal, se usarán `traits` para describir comportamiento y clases para describir la naturaleza de la entidad en cuestión.

Por último, si la eficiencia es importante o si el código va a ser llamado desde código *Java*, conviene usar clases. Se pueden consultar ejemplos en *AbstractClassesAndTraits.scala*.

3.5 Clases anónimas, tipos genéricos y *case classes*

Para reusar código es muy útil disponer de soporte para la creación de colecciones que acepten un tipo arbitrario como parámetro. Es posible definir clases que acepten tantos parámetros genéricos como se desee y también funciona para los `traits`.

Además, existen métodos genéricos, que reciben como parámetro un tipo genérico. Gracias a las clases anónimas podemos crear una instancia en el momento. Es posible instanciar una clase (abstracta o no) o un `trait` con algún miembro sin implementar e implementarlo directamente en la instanciación. Lo que realmente hace el compilador por detrás es crear una clase anónima que hereda de la clase o `trait` que deseamos instanciar y crea la instancia de esa clase.

Cabe resaltar que es necesario pasar los parámetros en caso de ser necesarios e implementar todos los miembros que no estén implementados. Las *case classes* son realmente útiles cuando trabajamos con estructuras de datos ligeras. Al declarar una clase como *"case"*, se implementan automáticamente algunas funcionalidades y métodos como:

- los parámetros pasan a ser campos (podemos acceder a ellos con la notación punto).
- método `toString`, métodos `equals` y `hashCode`.
- método `copy` (que acepta parámetros)
- creación de un *companion object* (con método *factory*, que hace, en este caso, el mismo papel que el constructor)
- las *case classes* son serializables (muy útil en sistemas distribuidos ya que se pueden enviar instancias por la red y entre *JVMs*)
- pueden ser usadas en *pattern matching* y también existen *case objects* (aunque no tienen *companion object* al ser ya objetos).

Se pueden consultar los ejemplos en el fuente *Generics_ Anonymous_ Case.scala*.

4 Programación funcional con Scala

4.1 Recursión y *tail-recursion*

Sabemos que en FP se hace uso de la recursividad cuando en el paradigma imperativo emplearíamos bucles. Para ejecutar una función recursiva, la *JVM* mantiene internamente una pila de llamadas (*call stack*) que calcula los resultados parciales que se van componiendo hasta obtener el resultado final. Este mecanismo puede verse en detalle al ejecutar una función recursiva en *debugger* del entorno de desarrollo.

Una vez que se llega al caso base, cada resultado parcial se va alimentando del resultado parcial anterior. La aproximación anteriormente descrita tiene un problema, la memoria.

La *JVM* mantiene todas las llamadas recursivas en la pila de llamadas (`Stack`), con las limitaciones que esto conlleva. El error que aparecerá cuando excedemos el número de llamadas recursivas que puede soportar la pila es conocido como `StackOverflowError`. Para solucionar este problema, debemos hacer que nuestra función sea *tail-recursive*, haciendo así uso del `heap`.

Se dice que una función es *tail-recursive* cuando vamos arrastrando el resultado actual en cada llamada. De este modo, no necesitamos ir apilando las llamadas (también se puede observar en el *debugger*) puesto que no dependemos de resultados que aún no se han calculado como ocurre en la recursión simple.

Cada nueva llamada va reemplazando a la anterior con el valor acumulado actualizado. Se usa la llamada recursiva como la última expresión. Finalmente, al llegar al caso base, se retorna el parámetro acumulador, que contendrá el resultado final.

Se pueden ver ejemplos en el archivo *RecursionAndTailRecursion.scala*.

4.2 Funciones

El objetivo es usar funciones como elementos de primera clase, es decir, trabajar con funciones del mismo modo en el que lo hacemos con valores y variables. El problema es que venimos del mundo orientado a objetos en el que todo es un objeto, una instancia de alguna clase, así es como la *JVM* fue diseñada originalmente para *Java*.

Se simula FP usando clases y con el método `apply()`. *Scala* usa algunos ajustes para hacer que el lenguaje parezca realmente funcional. El método `apply()` nos permite llamar a las instancias como si fueran funciones.

No hace falta que creemos los tipos de las funciones, ya que *Scala* los incorpora por defecto. Si queremos una función que reciba una parámetro y devuelva un resultado, usaremos una instancia de `Function1[A, B]`. *Scala* soporta este tipo para funciones de hasta 22 parámetros.

Además, *Scala* añade azúcar sintáctico para hacer el lenguaje aún más funcional:

`Function2[Int, Int, Int]` es equivalente a `(Int, Int) => Int`

Cabe resaltar que en *Scala* todas las funciones son objetos o instancias de clases derivadas de `Function1`, `Function2`, ... Las diferentes características sintácticas que incorpora *Scala*, lo convierten en un lenguaje funcional aun teniendo en cuenta que en el diseño original de la *JVM* nunca se pensó en la FP.

Se pueden encontrar ejemplos en el archivo *WhatsAFunction.scala*.

4.3 Funciones anónimas

Definir funciones usando los tipos para funciones `FunctionX` es incómodo y además, demasiado parecido al mundo orientado a objetos. Por ello, *Scala* introduce más azúcar sintáctico para instanciar la funciones.

Ejemplos disponibles en *AnonymousFunctions.scala*.

4.4 Funciones de orden superior y *currificación*

Un concepto importante es el de función de orden superior. Estas funciones o reciben alguna función como parámetro o devuelven alguna función como resultado. *"Currificar"* consiste en transformar una función que toma más de un parámetro en una secuencia de funciones que toma un sólo parámetro.

Además, *Scala* permite definir funciones con varias listas de parámetros, que pueden actuar como funciones *currificadas*.

Cabe resaltar que `def` y `val` no son equivalente ya que `def` evalúa en la llamada y crea una instancia nueva cada vez, mientras que `val` evalúa una vez que se define. Por ello, `def` ofrecerá menor rendimiento al crear una instancia nueva cada vez.

Hay ejemplos disponibles en el fichero *HOFs_ Curries.scala*.

4.5 Estructuras de datos funcionales

A continuación se expondrán diversas estructuras de datos que se encuentran estrechamente relacionadas con el paradigma de la programación funcional, y se explicará cómo se utilizan en el lenguaje *Scala*.

4.5.1 *Lists*

Las listas vienen a ser una secuencia de elementos del mismo tipo. Para mantener su inmutabilidad, cada vez que se "añade un elemento" a una lista, una nueva lista es creada. La sintaxis para una operación frecuente como puede ser añadir un elemento al principio de una lista es la siguiente:

```
val list1 = List("1", "2")
val list2 = "0" :: list1
```

La sintaxis es muy similar a la de *Haskell*. No obstante, resulta interesante mencionar que, dada la naturaleza orientada a objetos de *Scala*, la segunda línea del ejemplo anterior es equivalente a la siguiente:

```
val list2 = list1.::("0")
```

4.5.2 *Maps*

Los mapas son otra estructura de datos bastante popular y muy relacionada con la programación funcional. Se trata de conjuntos de parejas de claves y valores. Un ejemplo de creación de un mapa en *Scala* es el siguiente:

```
val capitales = Map(
   "Alemania" -> "Berlin"
   "Francia" -> "Paris"
   "Paises Bajos" -> "Amsterdam"
   "Espana" -> "Madrid")
```

Existe una amplia variedad de funciones y operaciones que pueden llevarse a cabo sobre mapas (añadir o eliminar valores, retener valores con una determinada condición, reducir el mapa, obtener el conjunto de claves o de valores como una subcolección, etc.). El abanico de operaciones se abre cuando dejamos de utilizar mapas inmutables y empezamos a usar los mutables. La contrapartida de esto sería la aparición de *side-effects* y, por tanto, el abandono del paradigma funcional.

4.5.3 *Options*

Si bien las dos estructuras anteriormente mencionadas pueden encontrarse en la mayoría de lenguajes en la actualidad, no sucede así con las opciones (`Options`). Una `Option` es un envoltorio (*wrapper*) cuyo contenido puede existir (`Some`) o no (`None`). Las operaciones funcionales sobre `Options` nos libran de implementar expresiones condicionales adicionales. Un ejemplo de su uso sería el siguiente:

```
val someNumber = Some(5)
val noneNumber = None
for (option <- List(noneNumber, someNumber)) {
   option.map(n => println(n * 5))
}
```

En caso de encontrarse con un valor vacío (`None`), no se aplicará la función, de forma que el desarrollador se está librando del manejo de excepciones.

4.5.4 Otras estructuras

Además de las listas, mapas y opciones, escogidas por su relación con la programación funcional en los primeros dos casos, y por una cierta "exclusividad" en el tercero, *Scala* también proporciona otra serie de estructuras como pueden ser los *arrays*, *sets*, tuplas, etc.

4.6 Operaciones funcionales

Del mismo modo que hay estructuras de datos íntimamente relacionadas con la programación funcional (a pesar de que también son usadas en otros paradigmas), existen operaciones con estas mismas características. Se exponen a continuación las más importantes:

- `map`: aplica una función a todos los elementos de la lista
- `flatMap`: aplica una función que devuelve una *sequence* para cada elemento de la lista y luego aplana (concatena) las *sequence* obtenidas. Es una combinación de `map` y `flatten`.
- `filter`: selecciona aquellos elementos que satisfacen un predicado.
- `foreach`: aplica una función f a todos los elementos, pero devuelve `Unit`. Es el método estándar para iterar por estructuras de datos en *Scala*. Cabe resaltar que genera efectos secundarios, por lo que no es un elemento puramente funcional.
- *for-comprehensions*: hace más legible cadenas del tipo:

  ```
  xs.flatMap(x => ys.flatMap(y => zs.map(z => ...)))
  ```

  Realmente, aunque usemos *for-comprehensions*, el compilador lo reescribe como cadenas de `map`, `flatMap` y `filter`. También permite el uso de guardas.
- `fold`: toma un operador binario y un valor inicial y reduce la estructura a un solo valor.

Se pueden consultar ejemplos en *MapFlatMapFilterFor.scala*.

5 *Pattern matching*

Una de las características más útiles e interesantes que proporciona *Scala* es el *Pattern matching* (PM). Se trata de un elemento similar al *"switch"* que proporcionan otros lenguajes como *C* y *Java*, donde un elemento de entrada es evaluado, y el primer patrón (*Pattern*) con el que se le encuentre coincidencia (*Matching*) es ejecutado.

Al igual que en el caso de los lenguajes mencionados, *Scala* proporciona soporte para evaluar expresiones "por defecto" (mediante el uso de *wildcards*), pero, por el contrario, sólo un patrón (o cero) pueden ser ejecutados (dado que una vez se evalúe un patrón como cierto se ejecutará y terminará la evaluación - en *C* y *Java*, si no aparece la sentencia *break*, se pueden ejecutar varios patrones).

Otra diferencia con la sentencia *"switch"* es que sólo puede haber coincidencias en función del valor, mientras que las expresiones de *Scala* proporcionan una

mayor flexibilidad (coincidencia de tipos, de expresiones regulares), mientras se mantiene una sintaxis concisa y clara.

Por último, cabe destacar que en el caso del manejo de excepciones también se puede hacer uso del PM.

Se pueden ver ejemplos en el fuente *PatternMatchingEnScala.scala*.

6 ¿Para qué se utiliza Scala?

Si bien *Scala* es un lenguaje de propósito general, hay determinadas tareas en las que destaca el uso de este lenguaje. Si se echa un vistazo a los *frameworks* de *Scala* con mayor popularidad nos podemos hacer una idea de los ámbitos donde es más usado. Así, encontramos:

- Ciencia de datos a través del *framework Apache Spark*. *Apache Spark* es un *framework* de propósito general ideado para la computación en clústers. Diversas características de *Scala*, como son la propia escalabilidad del lenguaje (que repercute en una mantenibilidad del código más sencilla), o la posibilidad de serialización de los objetos (permitiendo así la computación distribuida), hacen de este lenguaje un candidato de interés para su uso junto a *Spark*.

- Desarrollo web mediante el *framework Play*. *Play* está escrito en *Scala*, y sigue el patrón Modelo-Vista-Controlador. Su objetivo es la optimización de la productividad a partir de características como la Convención sobre Configuración o la recarga de código en caliente (*Hot Code Reloading*).

- Además de las dos áreas desarrolladas, *Scala* encuentra aplicación en otras tareas como son (a) aplicaciones concurrentes y distribuidas con el *toolkit Akka*, (b) ejecución en paralelo de trabajos por lotes o incluso (c) *scripts* simples desarrollados *ad-hoc* (ejecutados en *REPL*).

7 Bibliografía

- "Learning Scala by Jason Swartz (O'Reilly). Copyright 2015 Jason Swartz, 978-1-449-36793-0."

- "Scala Cookbook by Alvin Alexander (O'Reilly). Copyright 2013 Alvin Alexander, 978-1-449-33961-6."

- "Programming Scala by Dean Wampler and Alex Payne. Copyright 2009 Dean Wampler and Alex Payne, 978-0-596-15595-7."

- https://www.scala-lang.org/

- https://en.wikipedia.org/wiki/Scala_(programming_language)

- https://www.udemy.com/rock-the-jvm-scala-for-beginners/

- https://www.techfak.uni-bielefeld.de/ags/pi/lehre/ProgSem13/oopScalaPrint.pdf
- https://insights.stackoverflow.com/survey/2018/
- http://allaboutscala.com/tutorials/chapter-3-beginner-tutorial-using-classes-scala/scala-tutorial-learn-companion-objects-factory-apply-method-inheritance/
- https://stackoverflow.com/questions/33923/what-is-tail-recursion
- https://stackoverflow.com/questions/36314/what-is-currying/36321
- https://stackoverflow.com/questions/18887264/what-is-the-difference-between-def-and-val-to-define-a-function
- https://en.wikipedia.org/wiki/Apache_Spark
- https://en.wikipedia.org/wiki/Play_Framework
- https://en.wikipedia.org/wiki/Currying
- https://commitlogs.com/2016/09/10/scala-fold-foldleft-and-foldright/