Capítulo 2: Comenzando con la FP en Scala

Conceptos y observaciones

Objects Son un *tipo* singletón que se pueden interpretar como clases con una única instancia.

Higher order function Funciones que aceptan o retornan otras funciones.

Función polimórfica Funciones que generalizan sobre los tipos que puede manipular la función, brindando argumentos de tipo.

Trait Son el equivalente de Scala a las interfaces de Java, aunque estas también permiten definir implementaciones concretas de sus métodos.

- En Java se suelen utilizar clases anónimas para crear e instanciar rápidamente una clase que implementa una interfaz particular. Este comportamiento es similar al que tienen los `object` en Scala.

  Nota: Los `object` pueden ser utilizados para simular el comportamiento de la keyword `static` de Java.

- Conforme la complejidad de un proyecto crece, se hace inviable gestionar la compilación (scalac) y administración de cada uno de los archivos del proyecto. Para cubrir esta necesidad se suele usar el Scala Build Tool (*sbt*), que tras declarar y configurar su archivo, trackea y compila apropiadamente los archivos.

- Scala no tiene nociones especiales de *operadores*, y a efectos prácticos, **todo** es un objeto (no hay primitivos como en Java, por ejemplo). Por lo tanto `2 + 1` no es más que azucar sintáctico para `2.+(1)`.

- Los métodos unarios pueden usar notación infija. Si `abs` calcula el valor absoluto, `abs(42) == abs 42`.

- La noción de *namespace* corresponde a ver cada nombre en la notación clásica de punto (`Object1.Object2.val`), como un espacio que almacena *miembros*. Así, en la expresión `Object1.Object2.val`, `Object2` tiene como *namespace* `Object1` ya que es un miembro de él. Así mismo, `val` tiene como *namespace* a `Object1.Object2` y es miembro de `Object2` (y, posiblemente, de `Object1` también).

  Un objeto cuya única finalidad es ser el namespace de un conjunto de miembros, es llamado *módulo* (de aquí que tenga sentido llamar también módulo a los `object`).

- Es preferible hacer iteraciones mediante funciones recursivas que mediante loops o whiles. En Scala el performance de estos dos últimos es idéntico al obtenido por una función recursiva **cuando** dicha función usa *tail recursion*. (Esto evita añadir llamados a la pila de llamadas (call stack)).

  El compilador de Scala es capaz de generar un error si una función que debería usar tail recursion, no la está haciendo. Para esto se usa la anotación `@annotation.tailrec`.

- La versión más general de función hasta el momento, es la de una *función polimórfica*; estas funciones tienen 2 tipos de argumentos:

  1. Variables de tipo, usables en el resto de la firma de la función por las variables ordinarias o el retorno.
  2. Variables ordinarias, usables en el cuerpo de la función.

  Note tambien que las funciones polimórficas o genéricas añaden una restricción importante: *El cuerpo de las funciones sólo puede usar, en general, funciones que les han sido pasadas como argumentos.* Tal característica hace qué, en ocasiones, baste saber la firma con los tipos de una función, para poder construir una implementación *"canónica"* de esa función. El ejemplo de la aplicación parcial con firma

  ```
  def partial1[A,B,C](a: A, f: (A,B) => C): B => C
  ```

  es particularmente ilustrativa sobre esto.

- Las funciones anónimas de Scala no son más que la creación de un `object` con método `apply`. Este método permite llamar al objeto como si él mismo fuera el método apply. Estos objetos son creados a partir del trait `Function<n>`, un trait de Scala que define el método `apply` n-ario.

Conclusiones

- Los `object` de Scala son un 'shortcut' para crear una clase que se instancia inmediatamente.
- Las importaciones de un módulo completo usan la notación de variable anónima, por lo qué, para importar todos los miembros de un módulo, basta usar `import MyModule._`.
- Las funciones de Scala pueden tener parámetros tanto de tipo como de la función.

- Los tipos en funciones genéricas tienen el potencial para determinar su implementación en base únicamente a su firma. Esto es, los tipos permiten derivar una implementación.