Capítulo 1: Qué es programación funcional?

Expresión transparente referencialmente e es *referencialmente transparente (RT)* si, para todo programa p, todas las ocurrencias de e en p pueden ser reemplazadas por el resultado de evaluar e sin afectar el significado de p.

Función pura Una función f es pura si la expresión $f(x)$ es RT para toda expresión x RT.

Conclusiones:
El diseño funcional permite que los programas sean pensados y diseñados de una forma mucho más modular y local gracias a la transparencia referencial. Cuando una función debe ser modificada o símplemente comprendida, no es necesario simular mentalmente secuencias de actualizaciones de estado. El entendimiento se limita a un *razonamiento local* en donde todo lo que importa es la entrada y salida de la función. Además de esto, y gracias a que el código no tiene efectos laterales, el código se limita a ser una serie de composiciones.

Garantizar tales cosas hacen que las labores de testeo, revisión y modificación del código se vuelvan radicalmente más sencillas, aumentando la mantenibilidad y vida del código.

Capítulo 2: Comenzando con la FP en Scala

Objects En Java se suelen utilizar clases anónimas para crear rápidamente una instancia de una clase (la clase anónima) que implemente una interfaz. Este comportamiento se puede simular al que tienen los `object` en Scala. Estos `object` corresponden formalmente a un *tipo* singleton, pero pueden entenderse como clases que tienen una única instancia.

Nota: Estos `object` permiten asemejar el comportamiento de la keyword `static` en Java, en Scala.

- 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 1-ários 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*) `@annotation.tailrec`.

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._`.

Capítulo 3: