



Paradigmas de Programación (EIF-400) Realizaciones de FP:estático y Kotlin

CARLOS LORÍA-SÁENZ LORACARLOS@GMAIL.COM

SETIEMBRE 2017

EIF/UNA

Objetivos

2

- ▶ Estudiar el paradigma de FP en casos de lenguajes estáticos
- ▶ Contrastar con realizaciones dinámicas
- ▶ Usar Kotlin como caso de estudio

Objetivos Específicos

3

- ▶ Los mismos principios del caso dinámico
- ▶ Resaltar retos asociados con el caso estático
- ▶ Entender ventajas y desventajas

Material

4

- ▶ En el sitio
- ▶ [Kotlin referencia](#)

Repaso: FP

5

- ▶ Función es objeto abstrae computación
- ▶ Programar = Razonar/Combinar funciones
 - ▶ Combinadores explícitos en objetos
 - ▶ Recursión
- ▶ Evaluar = simplificar expresiones
- ▶ Minimizar efectos-secundarios (SE): transparencia referencial
- ▶ Literalidad (Knuth)
- ▶ Agilidad: DRY

Hipótesis

6

- ▶ FP da más expresión declarativa que programación imperativa

OOP-FP

7

- ▶ El objeto es datos (sustantivo)
- ▶ Pero puede ser controlador
- ▶ Controlador = tiene verbos de computación
- ▶ En “imperativo” el controlador es la máquina
- ▶ EN OOP-FP movemos patrones de control a objetos
- ▶ La función es la tarea a realizar

FP control al objeto

8

- ▶ Recursión
- ▶ Combinadores: `map`, `reduce`, `filter`, etc
- ▶ Le dan el control a la colección

Tipificación estática

9

- ▶ Sistema formal de razonamiento que dadas las restricciones de tipo en tiempo (estático) de compilación busca type-safety
- ▶ *Type-safety*: prevenir en tiempo de compilación que “*algo malo*” ocurra en tiempo de ejecución (runtime)
- ▶ Ejemplos:
 - ▶ Llamar a una función/operación con parámetros no especificados para la misma
 - ▶ Conversiones entre datos no relacionados (casting)

Tipificación estática

10

- ▶ Cada objeto (dato, función) tiene asignado un tipo (por el programador o por el compilador)
- ▶ Los tipos cumplen relaciones (reglas) lógicas (como sub-tipo, super-tipo, alias herencia)
- ▶ Cada expresión debe tener un único tipo. Normalmente el más “simple posible”
- ▶ El programa pasa la tipificación sino se viola ninguna regla de tipos.
- ▶ No se genera código si la tipificación falla.

Notaciones de tipos

11

- ▶ Estilo C: `String x (C, Java, C++)`
- ▶ Estilo Pascal `x:String` (Kotlin, Typescript)
- ▶ `String[] a` es un array de `String`
- ▶ `a: Array<String>` en Kotlin
- ▶ `String foo(String[] a)` Java
- ▶ **fun** `foo(a : Array<String>) : String`
Kotlin

Teorema

12

- ▶ Calcular el tipo exacto que va a tener un dato en un programa cualquiera (Turing computable) en tiempo de ejecución no es Turing computable.
- ▶ Si t es el tipo exacto el compilador a los más podrá calcular un super tipo de t , en general.

Tipificación estática: limitaciones

13

- ▶ Es conservativa: asume cualquier caso puede ocurrir
- ▶ En este ejemplo la asignación nunca se hará en runtime.
- ▶ Pero el compilador rechaza la asignación en compilación

```
int x = 0;  
if ( x - 1 > 0 )  
    x = "";
```

Tipificación limitaciones

14

- Este ejemplo pasa compilación. Pero el runtime lo captura "CastException"

```
String foo(Object x){  
    return (String)x;  
}
```

Pasa los
tipos

```
// ...  
foo( new Integer(666) ); // OK
```


Inferencia de tipos

15

- El sistema de tipos es capaz de deducir el tipo más apropiado de una expresión según el contexto

```
Arrays.asList(1, 2, 3)
    .stream()
    .map( n -> String.format("n=%d", n) )
    .forEach( s -> System.out.println(s) );
```


Ventajas tipos estáticos

16

- ▶ Código más robusto (menos errores)
- ▶ Código más documentado (el IDE ayuda en desarrollo)
- ▶ No es necesario ver el detalle del código para ver que se espera de un dato
- ▶ Código más eficiente (se evitan preguntas en runtime que el compilador ya garantizó)

Desventajas

17

- ▶ EL tiempo de desarrollo tiende a ser mayor
- ▶ Menos ágil
- ▶ No se evita el testing (siempre pueden haber pulgas: `ClassCastException`, `NullPointerException`, etc)

Tendencia: un punto intermedio

- ▶ Inferencia de tipos crea “*la ilusión*” de que no hay tipos estáticos en muchos casos (aunque los hay)
- ▶ Se mantienen las ventajas del análisis estático
- ▶ Lenguajes como Java8, C#, Scala y Kotlin se adhieren a ese modelo intermedio
- ▶ Se añaden los principios DRY también: sintaxis más breve (por ejemplo no poner siempre ;)
- ▶ Convenciones que simplifican la expresión de ideas

Parametricidad (generics)

19

- ▶ Tipos constantes: por ejemplo String, Object
- ▶ Tipos variables (parámetros de tipos)

The screenshot shows the Java IDE interface for the `Stack` class. At the top, there are tabs for `PREV CLASS`, `NEXT CLASS`, `FRAMES`, `NO FRAMES`, and `ALL CLASSES`. Below these is a summary bar with `SUMMARY: NESTED | FIELD | CONSTR | METHOD` and `DETAIL: FIELD | CONSTR | METHOD`. The main content area displays the class hierarchy for `Stack`. It starts with `compact1, compact2, compact3` and `java.util`. The class `Stack<E>` is highlighted in yellow. Below it, the hierarchy continues with `java.lang.Object`, `java.util.AbstractCollection<E>`, `java.util.AbstractList<E>`, `java.util.Vector<E>`, and `java.util.Stack<E>`. Under the heading `All Implemented Interfaces:`, the following interfaces are listed: `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, and `RandomAccess`. At the bottom, the class declaration is shown: `public class Stack<E>` and `extends Vector<E>`. Two green arrows point from the text in the list above to the `Stack<E>` class and its generic parameter `E` in the screenshot.

```
PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
java.util
Class Stack<E>
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.Vector<E>
        java.util.Stack<E>

All Implemented Interfaces:
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

public class Stack<E>
extends Vector<E>
```

Generics

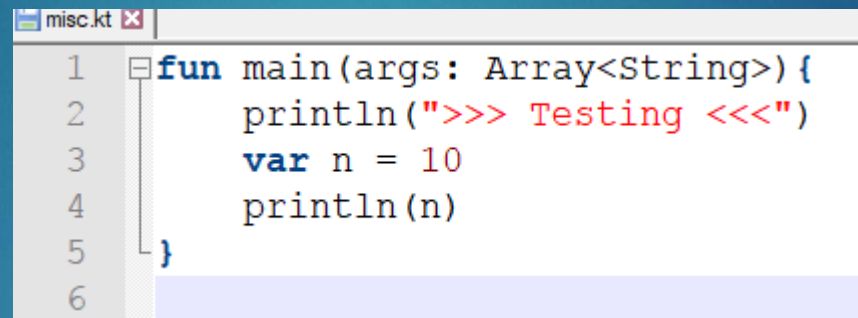
20

- ▶ Tendremos clase aparte

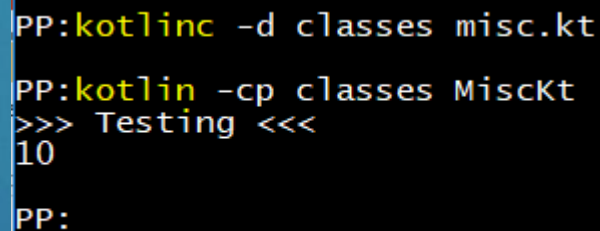
Ejercicio: kotlinc y kotlin

21

- Haga un archivo misc.kt



```
misc.kt x
1 fun main(args: Array<String>){
2     println(">>> Testing <<<")
3     var n = 10
4     println(n)
5 }
6
```



```
PP:kotlinc -d classes misc.kt
PP:kotlin -cp classes MiscKt
>>> Testing <<<
10
PP:
```

Ejercicio

22

- ▶ Estudie y compile el proyecto Java-Kotlin adjunto.
- ▶ Logre que corra

```
PP:bats\test_with_kotlin.bat
Prueba de Kotlin
>>> Saludos desde Kotlin! <<<
n=1
n=2
n=3
A(666).value = 666

PP:bats\test_with_java.bat
Prueba de Kotlin
>>> Saludos desde Java! <<<
n=1
n=2
n=3
PP:
```