Introducción a *Generics* (en Kotlin)

CARLOS LORIA-SAENZ.

EIF400 OCTUBRE 2017

Objetivos

- Revisar "`generics`" en Kotlin
- Enfoque de varianza en Kotlin
- Contraste con Java

Referencias

- Generics en Kotlin (reference)

Material

- En el sitio

Generics en Kotlin

- Mismos principios de generics (polimorfismo paramétrico) que Java
- Realización es distinta (no wildcards "`? extends` " "`? super`")
- Especificación de varianza "*en declaración*" y "en sitio" (Java sólo en sitio con proyecciones)
- Estrategia subyacente la misma: "*borrado de tipos*"
- Más flexibilidad con "re-ificacion" en el caso de inlining

Tipos invariantes

- En Java los tipos paramétricos son en su declaración siempre <u>invariantes</u>

compact1, compact2, compact3
java.util

Interface List<E>

Type Parameters:

E - the type of elements in this list

E es declarado invariante

Consecuencia

```
List<String> ls = new Arrays.asList("a", "b");
List<Object> lo = ls; // No compila
```

`K<T>` invariante en `T`

`A <: B` <u>no implica</u> `K<A> <: K<B>` ni `K<B> <: K<A>`

Tipos invariantes (JDK8)

java.util.function

Interface Function<T,R>

Type Parameters:

T - the type of the input to the function

R - the type of the result of the function

T y **R** invariantes

Primer parámetro de `after` es contravariante en R

Segundo es covariante en V

default <V> **Function**<T,V>	**andThen**(**Function**<? super R,? extends V> after) Returns a composed function that first applies this function to its input, and then applies the after function to the result.
R	**apply**(T t) Applies this function to the given argument.
default <V> **Function**<V,R>	**compose**(**Function**<? super V,? extends T> before) Returns a composed function that first applies the before function to its input, and then applies this function to the result.

Ejercicio

- Revise work/Wildcards.java
- Justifique por qué compila

Productor-Consumidor

- Un tipo `K<T>` es productor para `T` si sólo hay métodos de `K<T>` que retornen `T` (a lo más)
- Similarmente `K<T>` es consumidor para `T` si sólo hay métodos que usan `T` (a lo más)
- Note que se podría generalizar y restringir K<T> a ser un productor si uno se compromete a sólo usar métodos productores
- Similarmente con consumidores.
- El compromiso sería con el sistema de tipos.

Ejercicio

- Revise el API de `List<E>` de `JDK8` y determine en cada método si es consumidor o productor
- Por ejemplo: ¿`add`?, ¿`get`? y ¿`clear`?

PECS

- En Java `K<? extends T>` es un productor en `T`
- Similarmente `K<? super T>` es un `consumidor` en T
- Producer-is-Extends, Consumer-is-Super (PECS)
- En Java `K<T>` es subtipo de `K<? extends T>` (covaría)
- Y `K<T>` es subtipo de `K<? super T>` (contravaría)
- Otra manera es decir que `K<? extends T>` es *read-only* en T y `K<? super T>` *write-only* en T

Anotación `out`

- Si `K<T>` es productor en `T`, entonces Kotlin permite denotar eso como `K<out T>`
- Ventaja: `K<T>` se vuelve covariante en `T`.

Ejercicio

- Con un cambio minimalista logre que work/Animal.kt compile y ejecute

Anotación `in`

- Si `K<T>` es consumidor en `T`, entonces Kotlin permite denotar eso como `K<in T>`
- Ventaja: `K<T>` se vuelve contravariante en `T`.

Ejercicio

- Explique por qué se permite compilar con un Comparator de `Any` una lista de `String` en `work/Sorting.kt`.

Proyecciones

- Si una función `f` usa `K<T>` (que no fue declarado como `K<out T>`) pero `f` sólo lo usa a `K<T>` como consumidor entonces se puede declarar `K<out T>` en `f` y entonces `f` se vuelve covariante en esa posición
- Análogamente con `in`.

Ejercicio

- Haga un cambio minimalista en `work/Copy.kt` para que sí compile (no modifique `main`)

Proyección *

- Una función `f` usa un parámetro `x: K<T>` pero no usa nada que requiera `T` o saber qué es `T`. Entonces `x` se puede declarar en `f` como `x:K<*>`
- `K<*>` significa que hay un tipo pero no se sabe cuál es (y en el contexto no se necesita saberlo)

Ejemplo

- Revise `work/First.kt`