



Introducción a *Generics* (en Java)

CARLOS LORIA-SAENZ.

EIF400 OCTUBRE 2017

Objetivos

2

- ▶ Revisar “`generics`” en Java
- ▶ Revisar tipificación estática (varianza)
- ▶ Justificación de `generics`
- ▶ Destacar ventajas en reuso
- ▶ Limitaciones (borrado de tipos)
- ▶ Colateralmente revisar Java-FP (uso extensivo de Generics)

Referencias

- ▶ G. Bracha. Generics in the Java Programming Language. 2004. (se consigue en la Web)
- ▶ [Doc de Oracle](#)

Material

4

► En el sitio

Generics (aka. polimorfismo paramétrico): Justificación

- ▶ Ofrece mejor reuso de algoritmos que no dependen de los tipos
- ▶ Es común en colecciones, listas.
- ▶ Ejemplo: una pila. Las operaciones `push`, `pop`, etc no dependen del tipo de objeto en la pila
- ▶ Permite más análisis al sistema de tipos (tiene más información)
- ▶ Eliminar castings en el código fuente
- ▶ Menos código significa menos mantenimiento
- ▶ Más legibilidad del código

Sintaxis: Parámetros de tipos (ver Pila)

6

```
public interface Pila<E>{  
    E pop();  
    void push(E e);  
    E top();  
    boolean isEmpty();  
}
```

E es un parámetro de tipo
Pueden usarse en clases,
interfaces y métodos

Convenciones (APIs de JDK)

7

- ▶ Tipos son identificadores normales, pero se ponen en mayúscula, una sola letra
- ▶ T: tipo primario general.
- ▶ S tipo secundario
- ▶ R: tipo de retorno
- ▶ E: elemento de una colección
- ▶ K: tipo de una llave (Key)
- ▶ V: valor (Value)
- ▶ N: número

Ejemplo: java.util.HashMap

8

```
java.lang.Object  
    java.util.AbstractMap<K,V>  
        java.util.HashMap<K,V>
```

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>

Ejemplo: java.util.function.Function

9

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.
static <T> Function <T,T>	identity () Returns a function that always returns its input argument.

Function es el tipo de una
lambda de un argumento

Ejemplo: eliminación de casting en fuente

10

Casting en
código fuente

El sistema de tipos
“sabe” que lista
tiene String: no
hace falta el casting

```
public static void sinGenerics() {  
    List lista = new ArrayList();  
    lista.add("csh");  
    String so = (String) lista.get(0);  
}  
public static void conGenerics() {  
    List<String> lista = new ArrayList<>();  
    lista.add("csh");  
    String sg = lista.get(0);  
}
```

Sistema de tipos

11

- ▶ Analizador de semántica estática: “*type safety*” (nada malo ocurre en `runtime`)
- ▶ Debe velar porque un programa no corrompa (polucione) los objetos en tiempo de ejecución: que el tipo “*prometido*” en compilación sea protegido en `runtime`
- ▶ La JVM tiene tipos dinámicos y con eso evita “*polución*” de la memoria
- ▶ **RECORDAR**: Es indecidible saber que tipo exactamente tendrá una variable en tiempo de ejecución

Restricciones inesperadas

12

- ▶ Haga lo indicado en [slide 14](#)

Regla básica: Asignación

13

- ▶ Escribimos $x : A$ para decir x es de tipo A .
- ▶ Escribimos $B <: A$: $B = A$ o es subtipo de A (o A es supertipo de B) (en Java $B \text{ extends } A$, en Kotlin $B : A$)
- ▶ Regla asignación: $x = y$ es válida en compilación sólo si $x : A$, $y : B$ y $B <: A$
- ▶ Aplica a llamadas de métodos y funciones también.
- ▶ Si $R \text{ f}(\dots, A \ x, \dots)$ entonces $\text{f}(\dots, y, \dots)$ es válida si $y : B$ y si $B <: A$
- ▶ Un Objetivo del sistema de tipos si $x = y$ fue válida en tiempo de compilación entonces si $x = y$ fuera inválida en ejecución entonces la JVM podrá reportar una excepción.

Ejemplo: “engañando al sistema de tipos” (compilador) (Ver Tipos)

Esto compila. Si quitamos el casting no compila

```
public class Tipo{  
    public static void test(Object obj, Integer[] a){  
        a[0] = (Integer)obj;  
    }  
    public static void main(String[] args){  
        Integer[] a = {1,2,3};  
        test("csh", a);  
    }  
}
```

Pero se cae en runtime: no se “corrompe” (poluciona) el arreglo


```
PP:java Tipo  
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lan  
g.Integer  
    at Tipo.test(Tipo.java:3)  
    at Tipo.main(Tipo.java:7)
```

Problema con covarianza y generics (ver J/J.java) (volver)

```
import java.util.*;
public class J{
    public void fooArray(){
        String[] as = new String[0];
        Object[] os = as;
    }
    public void fooList(){
        List<String> ls = new ArrayList();
        List<Object> lo = ls;
    }
    public static void main(String[] args){
    }
}
```

No compila
Aunque String es subtipo de Object
List<String> NO ES SUBTIPO de List<Object>

¿Por qué no?



```
PP:javac J.java
J.java:17: error: incompatible types: List<String> cannot be converted to List<Object>
        List<Object> lo = ls;
                        ^
Note: J.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

Varianza en tipos genéricos

16

- ▶ Si un tipo T en $K<T>$ preserva la relación de subtipo (es decir si $A <: B$ implica $K<A> <: K$) se dice que T es covariante en K .
- ▶ Si $A <: B$ implica $K<A> <: K$ dice contravariante (se invierte la relación)
- ▶ Si sólo puede sustituirse por T mismo para que haya relación entonces se dice invariante. ($A <: B$ no implica $K<A> <: K$ ni implica $K<A> <: K$; sólo si $A=B$)
- ▶ Por ejemplo: `String[] <: Object[]` porque `String <: Object`. Es decir T en $T[]$ es covariante.
- ▶ Pero en `List<T>` el tipo T es invariante. `List<String>` no es subtipo de `List<Object>` (ni viceversa). `List<String>` sólo es subtipo de si mismo.

Varianza en métodos

17

- ▶ Sea $R \text{ } f(T_1 \text{ } x_1, \dots, \mathbf{A_i \text{ } x_i}, \dots, T_n \text{ } x_n) \{ \dots \}$ un método declarado en una clase C .
- ▶ Decimos que f es covariante en parámetro en A_i si cuando extendemos C con una clase D y cambiamos A_i por B_i con $B_i <: A_i$ entonces eso provoca la sobrescritura (sustitución) del f de C por el f de D en esa clase D .
- ▶ Contravariante si la sobrescritura ocurre si $A_i <: B_i$
- ▶ Invariante si la sobre escritura sólo ocurre si $A_i = B_i$
- ▶ Similares definiciones sobre el tipo R de retorno (se dice covariante, contravariante o invariante en retorno).
- ▶ **Nota:** Java es invariante en todo parámetro. Y covariante en retorno (jdk>5). No hay contravarianza en métodos.

Covarianza (ver directorio Tipos)

18

Covarianza.java

```
public static void main(String[] args){  
    Integer[] a = {1,2,3};  
    Object[] o = a;  
}
```

Esto si compila
Integer[] si es
subtipo de Object[]

Se dice que esos tipos
covarían

Ejemplo: Bolsa.java (ver Bolsa_01)

- ▶ Simple ejemplo con generics
- ▶ Revisar y compilar
- ▶ Descomentar línea 33

```
C:\Windows\System32\cmd.exe
PP:javac Bolsa.java
PP:java Bolsa
Nums: [1, 2, 3, 4]
Hileras: [a, b, c]
```

```
PP:javac Bolsa.java
Bolsa.java:33: error: incompatible types: Bolsa<String> cannot be converted to Bolsa<Object>
    Bolsa<Object> objetos = hileras;
                        ^
1 error
```

Regla de convariancia

20

- ▶ Tipo de objeto genérico es un tipo de objeto parametrizado como `Bolsa<T>`
- ▶ Tipo de Objeto genérico: No son covariantes
 - ▶ `Bolsa <: Bolsa<A>` sólo si `A == B`
- ▶ Se dice que son invariantes
- ▶ La única excepción:
 - ▶ Arrays sí son covariantes `B[] <: A[]` si `B <: A`

En gráficos: regla de asignación

21

- $A \leftarrow B$
- Lea B como A
- Escriba B en A

sii



sii

B extends A

Si B = ?

$A \leftarrow ?$
Lea ? como A

sii



sii

? extends A

Si A = ?

$? \leftarrow B$
Escriba B en ?

sii



sii

~~B extends ?~~

? super B

Wildcards (comodines)

22

- ▶ Son expresiones de Java para poder lograr covarianza
- ▶ Son “?”, “? extends” y “? super”
- ▶ Explicamos con ejemplos, pero en resumen

? extends A es un tipo desconocido del cuál lo único que sabemos es que es subtipo de A

? super A es un tipo desconocido del cuál lo único que sabemos es que es supertipo de A

? equivale a ? extends Object

Wildcard “?” (ver Bolsa_02)

23

- ▶ Queremos un método estático en `Bolsa` que sirva para imprimir bolsas de cualquier tipo
- ▶ Esto no compilaría `T` en contexto estático

```
public static void imprimaUnaBolsa(Bolsa<T> bolsa){
```

- ▶ Falla pues `S` no es conocido

```
public static void imprimaUnaBolsa(Bolsa<S> bolsa){
```

- ▶ Esto sólo aceptaría `Object`

```
public static void imprimaUnaBolsa(Bolsa<Object> bolsa){
```


Una solución: (Bolsa_02)

24

- Usar el wildcard “?”

```
// Pruebe cambiando ? por T, S, Object y compila
public static void imprimaUnaBolsa(Bolsa<?> bolsa) {
    IntStream.range(0, bolsa.largo())
        .forEach( i -> System.out.println(bolsa.sacar(i)));
}
```

Bolsa<?> sí es supertipo de
Bolsa<Integer>, y de Bolsa<String>

Limitación: «readonly»

25

- ▶ ? Sólo sirve para sacar (leer) cosas como Object.
- ▶ Recuerde que equivale a ? extends Object
- ▶ No permite guardar objects

No compila.
Polucionaría b

```
Bolsa<?> b = new Bolsa<String>();  
b.guardar(new Object());
```

Otra solución: métodos parametrizados: ver Bolsa_03

26

```
public static <S> void imprimaUnaBolsa(Bolsa<S> bolsa){  
    for (int i = 0 ; i < bolsa.largo(); i++)  
        System.out.println(bolsa.sacar(i));  
}
```

Inferencia de
tipo en
contexto

```
Bolsa<Integer> nums = new Bolsa<>(); // Integer  
nums.guardar(Arrays.asList(1,2,3,4));  
System.out.println("Nums:"+nums);  
Bolsa<String> hileras = new Bolsa<>();  
hileras.guardar(Arrays.asList("a", "b", "c"));  
System.out.println("Hileras:"+hileras);  
Bolsa.imprimaUnaBolsa(nums);  
Bolsa.imprimaUnaBolsa(hileras);
```

Wildcard “? extends” (Bolsa_04)

27

```
public boolean compararConOtraBolsaEnPos(Bolsa<? extends T> otra, int pos){  
    // Por simpleza por ==  
    return this.largo() == otra.largo() && this.sacar(pos) == otra.sacar(pos);  
}
```

? extends A: significa X extends A para un tipo X fijo que es desconocido; solo se sabe que es subtipo de A o puede ser A mismo

Wildcard “? super” (Bolsa_04)

28

```
public void pasarAOtraBolsaPorPos(Bolsa<? super T> otra, int pos){  
    otra.guardar(this.sacar(pos));  
}
```

? super A: significa A extends X para un tipo X fijo que es desconocido; solo se sabe que es supertipo de A o puede ser A mismo

- ▶ Use `extends` si es para lectura (entrada)
- ▶ Use `super` si es para escritura (salida)
- ▶ O use métodos parametrizados: Ejemplo `java.util.Collections`

```
static <T> void
```

```
copy(List<? super T> dest, List<? extends T> src)
```

Copies all of the elements from one list into another.

Ejemplos: java.util.Collection

30

Method and Type	Method and Description
boolean	<code>add(E e)</code> Ensures that this collection contains the specified element (optional operation).
boolean	<code>addAll(Collection<? extends E> c)</code> Adds all of the elements in the specified collection to this collection (optional operation).
void	<code>clear()</code> Removes all of the elements from this collection (optional operation).
boolean	<code>contains(Object o)</code> Returns true if this collection contains the specified element.
boolean	<code>containsAll(Collection<?> c)</code> Returns true if this collection contains all of the elements in the specified collection.

Casos potencialmente inseguros (ver Legado.java)

31

```
PP: javac Legado.java
Note: Legado.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

PP: javac -Xlint:unchecked Legado.java
Legado.java:26: warning: [unchecked] unchecked conversion
        List<Dato> datosDespues = sg.demeDatos();
                                   ^
    required: List<Dato>
    found:    List
1 warning

PP:
```

- Compilador no puede dar un error porque sino ya no se podría usar la clase “legada”.
- Da un warning

Borrado (ver Borrado.java)

32

- ▶ Java no tiene tipos genéricos en su JVM
- ▶ Los “borra” durante compilación. No existen en tiempo de ejecución

```
// Uso de generics normal
public static String conGenerics(){
    List<String> lista = new ArrayList<>();
    lista.add("csh");
    return lista.get(0);
}

// Así lo trata el compilador
public static String sinGenerics(){
    List lista = new ArrayList();
    lista.add("csh");
    return (String) lista.get(0);
}
```

Limitaciones (ver NoSeVale.java)

33

- ▶ Toda clase $C<T>$ en compilación termina siendo sólo C en ejecución
Una única clase por cada instancia de T
- ▶ Eso limita ciertos casos
- ▶ Se revisan con ejemplos en NoSeVale

Ejercicio: java.util.function.Function

34

- Estudie estos métodos. Explique sus tipos

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.
static <T> Function <T,T>	identity () Returns a function that always returns its input argument.

Ejercicio

35

- ▶ Conteste las preguntas en Ejercicios/Assign.java

Ejercicio

36

- ▶ Explique la salida de Ejercicios/Return.java
 - ▶ ¿Por qué no es "**C::foo**".

```
PP:javac Return.java
PP:java Return
D::foo
PP:
```

Ejercicio: Haga lo pedido en Ejercicios\ejercicio_js_to_java.js

Se trata de traducir de ES6
a Java8

```
PP:node ejercicio_js_to_java.js  
[ 10, 20, 30 ] ' ---> ' [ 456976, 470596, 484416 ]
```