Tipos genéricos

en Java

Verónica E. Arriola-Rios

Facultad de Ciencias, UNAM

20 de enero de 2021

Temas

Tipos genéricos

Temas

Genéricos

- Proveen un mecanismo para crear métodos y estructuras que funcionan sobre tipos distintos sin necesidad de utilizar conversión explícita (*casting*).
- Permiten detectar un mayor número de errores en tiempo de compilación, en lugar de en tiempo de ejecución[1].
- En `Java` los tipos genéricos funcionan **exclusivamente en tiempo de compilación**.

[1] Siempre y cuando no se abuse de ellos.

Ejemplo de uso

Antes

```
1 Lista l = new Lista();
2 l.agrega(new Numero());
3 Object o = l.lee(0);
4 Numero n = (Numero) l.lee(0);
```

vs

Ahora

```
1 Lista<Numero> l = new Lista<>();
2 l.agrega(new Numero());
3 Numero n = l.lee(0);
```

Ejemplo de declaración

```
1  public class Lista <E> {
2    private E dato;
3    private Lista<E> sig;
4
5    public Lista(E dato) {
6      this.dato = dato;
7    }
8
9    public void agrega(E dato){
10     if(sig != null) sig.agrega(dato);
11     else {
12       // Usando el diamante <>
13       sig = new Lista<>(dato);
14     }
15   }
16 }
```

Convensiones de nombrado

- Se utiliza una sola letra mayúscula.
- Nombres comunes son:

E	elemento (almacenado en una estructura de datos).
K	llave en un mapa o diccionario (del inglés *key*).
N	número
T	tipo (cualquiera)
V	valor
S,U,V,...	cuando se requieren más variables.

Las cosas que no se pueden hacer

No se puede:

1. asociar tipos primitivos a variables genéricas (pero se pueden usar *wrappers*),
2. crear objetos a partir de parámetros de tipo,
3. crear campos estáticos con tipos genéricos,
4. *usar conversiones de tipo o usar `instanceof` con tipos parametrizados,
5. crear arreglos de tipos parametrizados,
6. crear, cachar o lanzar objetos de tipos parametrizados,
7. sobrecargar métodos donde los parámetros formales de tipos se borren a los mismos tipos.

Temas

Métodos genéricos

- Si requerimos utilizar el tipo del argumento:
- Es posible declarar y utilizar una variable genérica que se utilice únicamente en el método:

```
1 public static <T> void fromArrayToCollection
2         (T[] a, Collection<T> c) {
3     for (T o : a) {
4         c.add(o); // Correcto
5     }
6 }
```

Temas

Contenedores de subtipos vs subtipos de contenedores

¿Qué sucede si queremos utilizar subtipos[2] y genéricos?

- x Un contenedor de tipo `T` no es lo mismo que un contenedor de supertipos de `T`, pues el contenedor de supertipos puede contener elementos que el otro no.

```
1 Lista<String> l = new Lista<>();
2 Lista<Object> lo = ls;  // Error
3 lo.agrega(new Object());
```

[2] Herencia de clases e implementación de interfaces

Comodines

- Para crear métodos que funcionen con varias clases con tipos genéricos, se requieren *comodines* (*wildcards*).

```
1 public void printIterable(Iterable<?> c) {
2   for (Object e : c) {
3     System.out.println(e);
4   }
5 }
```

Comodines acotados superiormente

- Si queremos limitar los tipos que pueden ser aceptados por un método utilizamos comodines acotados.
- Cota superior (una superclase):

```
1 public Real sumaTodos
2         (Iterable<? extends Real> numeros) {
3   Real r = new Doble(0);
4   for(Numero n: numeros) {
5     r = r.suma(n);
6   }
7   return r;
8 }
```

¡OJO! Dado que se desconoce el tipo exacto de `numeros` ahora el compilador prohibe utilizar aquellos métodos que requieran conocer el tipo genérico exacto, por ej. `public void agrega(T e)`.

Cotas múltiples

```
1 Class A { /* ... */ }
2 interface B { /* ... */ }
3 interface C { /* ... */ }
4
5 class D <T extends A & B & C> { /* ... */ }
```

La clase de la cual se debe heredar debe ser la primera en ser especificada.

- Otro ejemplo con cotas superiores:

```
1 public class Collections {
2   public static <T> void copy
3     (List<T> dest, List<? extends T> src) {
4     ...
5   }
6 }
```

Comodines acotados inferiormente

- Cota inferior. Por ejemplo, nos sirve para métodos genéricos que escriben.

```
1  interface Sink<T> {
2      flush(T t);
3  }
4  public static <T> T writeAll(Collection<T> coll,
5                               Sink<? super T> snk) {
6    T last;
7    for (T t : coll) {
8      last = t;
9      snk.flush(last);
10   }
11   return last;
12 }
13 Sink<Object> s; // ? es un supertipo de T
14 Collection<String> cs;
15 String str = writeAll(cs, s); // Yes!
```

Los genéricos ayudan únicamente en tiempo de compilación

Temas

2. Los genéricos ayudan únicamente en tiempo de compilación
   - Borrado de tipo
   - Donde el asunto puede ir muy mal

Borrado de tipo *(Type erasure)*

- Las variables genéricas existen únicamente en tiempo de compilación.
- Se puede ver como si el código fuente fuera convertido a otro código fuente **sin genéricos** donde:
  - `T` es reemplazado por `Object`, `T extends Número` es reemplazado por `Número` y `T super Número` es reemplazado por `Object`.
  - Los castings requeridos son insertados automáticamente.

```
1  public class ConDato <T extends Entero> {
2    private T datoT;
3    public T get() {
4      return datoT;
5    }
6  }
```

```
1  public class ConDato {
2    private Entero datoT;
3    public Entero get() {
4      return (Entero) datoT;
5    }
6  }
```

- Se crean métodos puente para preservar el polimorfismo entre clases genéricas extendidas.

Código 1: Código con genéricos

```
1  public class Node<T> {
2      public T data;
3      public Node(T data) { this.data = data; }
4      public void setData(T data) {
5          System.out.println("Node.setData");
6          this.data = data;
7      }
8  }
9
10 public class MyNode extends Node<Integer> {
11     public MyNode(Integer data) { super(data);
           -> }
12     public void setData(Integer data) {
13         System.out.println("MyNode.setData");
14         super.setData(data);
15     }
16 }
```

Código 2: Después del borrado de tipos

```
1  public class Node {
2      public Object data;
3      public Node(Object data) { this.data = data; }
4      public void setData(Object data) {
5          System.out.println("Node.setData");
6          this.data = data;
7      }
8  }
9
10 public class MyNode extends Node {
11     public MyNode(Integer data) { super(data); }
12     // Bridge method generated by the compiler
13     public void setData(Object data) {
14         setData((Integer) data);
15     }
16     public void setData(Integer data) {
17         System.out.println("MyNode.setData");
18         super.setData(data);
19     }
20 }
```

Si los tipos genéricos fueron borrados...

En tiempo de ejecución:

- Sólo quedan los tipos ordinarios.

```
1  List <String> l1 = new ArrayList<String>();
2  List <Integer> l2 = new ArrayList<Integer>();
3  System.out.println
4    (l1.getClass() == l2.getClass()); // true
```

- No podemos usar `instanceof` con genéricos:

```
1  java.util.Collection cs = new ArrayList<String>();
2  // Illegal.
3  if (cs instanceof java.util.Collection<Integer>) {
4      System.out.print("Yes it is");
5  }
6  // Legal
7  if (cs instanceof java.util.Collection) {
8      System.out.print("Yes it is");
9  }
```

Temas

2. Los genéricos ayudan únicamente en tiempo de compilación
   - Borrado de tipo
   - Donde el asunto puede ir muy mal

Regla Golem #2

- Se supone que esto lanza una advertencia (el compilador puede estar configurado para **no** lanzarla):

```
1  Lista<String> cstr = (Lista<String>) cs;
```

```
1  <T> T badCast(T t, Object o) {
2      return (T) o;
3  }
```

Infiltrando al enemigo: contaminación del heap

```
1  import java.util.ArrayList;
2  import java.util.LinkedList; // Can trick too
3
4  public class ArrayListTrick {
5
6    /** Let's errase the identity of any object. */
7    public static <T,U> T deceiver(U something) {
8      return (T)something;
9    }
10
11   /** Abusing the type erasure system... */
12   public static void main(String[] args) {
13     // Only numbers
14     ArrayList<Number> lista = new ArrayList<>();
15     //LinkedList<Number> lista = new LinkedList<>();
16     String intruder = "Hola";
17     // ArrayList stores its stuff inside an Object[] array.
18     lista.add(deceiver(intruder)); // T is cleaverly inferred to be "
           ->Number".
19     lista.add(new Double(24.5));
20     lista.add(new Float(50.3));
21     lista.add(new Integer(100));
22     for(int i = 0; i < lista.size(); i++) {
23       // OK, just don't make it explicit there is an intruder there.
24       // If I call anything specific to Number someone will figure out
             ->.
25       System.out.println(lista.get(i));
26     }
27     // The compiler won't let me cast a Number into String, so let's
           ->use a trick.
28     String cameBackAlive = deceiver(lista.get(0));
29     System.out.println(cameBackAlive + "␣made␣it␣back!");
30   }
31 }
```

Infiltrando al enemigo (¡descubierto!)

```java
import java.util.ArrayList;
import java.util.LinkedList; // Can trick too

public class ArrayListTrick {

  /** Let's errase the identity of any object. */
  public static <T,U> T deceiver(U something) {
    return (T)something;
  }

  /** Abusing the type erasure system... */
  public static void main(String[] args) {
    // Only numbers
    ArrayList<Number> lista = new ArrayList<>();
    //LinkedList<Number> lista = new LinkedList<>();
    String intruder = "Hola";
    // ArrayList stores its stuff inside an Object[] array.
    lista.add(deceiver(intruder)); // T is cleaverly inferred to be "
        ->Number".
    lista.add(new Double(24.5));
    lista.add(new Float(50.3));
    lista.add(new Integer(100));
    Number n = 0;
    for(int i = 0; i < lista.size(); i++) {
      // Oops!!!
      n = n.doubleValue() + lista.get(i).doubleValue();
    }
    // The compiler won't let me cast a Number into String, so let's
        ->use a trick.
    String cameBackAlive = deceiver(lista.get(0));
    System.out.println(cameBackAlive + " made it back!");
  }
}
```

Regla Golem #3. Arreglos

- El tipo de los objetos almacenados en un arreglo no puede ser:
  - Un tipo genérico (ej. `T`).
    `T[] a = new T[100];` ✗
  - Un tipo parametrizado (ej. `Lista<T>`).
    `List<String>[] lsa = new List<String>[10];` ✗
    a menos que sea un comodín no acotado `?`:
    `List<?>[] lsa = new List<?>[10];` ✓

Regla Golem #3. Arreglos (continuación)

- ¡Sí se pueden declarar estos tipos, pero no crear esos objetos!
  - Permite pasar arreglos como parámetros, sin que se conozca su tipo.

```
1 ...
2   public static <T> void imprime(T[] arreglo) {
3     for(T o : arreglo) System.out.println(o);
4   }
5   public static void main(String[] args) {
6     Integer[] is = {1,2,3,4,5};
7     imprime(is);
8   }
9 ...
```

  - Esta regla tienta a que se hagan conversiones explícitas, pero no impide el *posible* error en tiempo de ejecución.
  - x Se presta a usos con consecuencias impredecibles como:
    `T[] a = (T[]) new Object[10];`

Compatibilidad

Temas

3. Compatibilidad
   - Interactuando con código viejo
   - *Del pasado al presente, que salga equivalente (o actualizando código viejo)

Tipos puros (*raw types*)

- Si no se especifica el tipo actual de una clase genérica, se dice que el *tipo es puro*[3].
- Un tipo puro se refiere a cualquier genérico, como si los genéricos no hubieran existido.
- Cada declaración en el código siguiente significa algo distinto:

```
1  Lista l; // Tipo puro
2  Lista<Numero> ln;
3  Lista<Object> lo;
4  Lista<?> lq;
```

[3] En inglés *raw type*.

Regla Golem #1

- Los tipos puros permiten interactuar con código viejo realizando asignaciones inseguras como:

```
1 public interface Assembly {
2   // Returns a collection of Parts
3   Collection getParts();
4 }
5 ...
6 Collection<Part> k =
7   Inventory.getAssembly("thingee").getParts();
```

"Shouldn't this be an error? Theoretically speaking, yes; but practically speaking, if generic code is going to call legacy code, this has to be allowed."

Temas

3. Compatibilidad
   - Interactuando con código viejo
   - *Del pasado al presente, que salga equivalente (o actualizando código viejo)

Cotas múltiples

- Si se quiere mantener la compatibilidad con otras clases el código nuevo debe:
  - Funcionar para los mismos argumentos y tipos de regreso.
  - Mantener compatibilidad a nivel binario (debe seguir ejecutando correctamente con otro código compilado).
- La regla general es: al borrar los tipos genéricos debe quedar la fima del método original sin genéricos.

Borrando a la firma anterior

- Por ejemplo:
  ```
  public static <T extends Comparable<? super T>> T
  max(Collection<T> coll)
  ```
- Se borra a:
  ```
  public static Comparable max(Collection coll)
  ```
- La firma original era:
  ```
  public static Object max(Collection coll)
  ```
- Se arregla forzando a que la clase más general sea la especificada como primer argumento en:
  ```
  public static <T extends Object & Comparable<? super T>> T
  max(Collection<T> coll)
  ```

Parchando

Creando arreglos de tipo T

Usar el método de la API `Arrays`:

```
1  public static <T> T[] copyOf(T[] original, int newLength);
```

Por ejemplo:

```
1  /** Recibe como parámetro un arreglo de tamaño cero,
2  el cual usa para el crear otro arreglo del tipo del
3  que se pasó como parámetro. */
4  public static <T> T[] creaArregloT(T[] crea, int length) {
5    return Arrays.copyOf(crea, length);
6  }
7  ...
8  public static void main(String[] args) {
9    String[] arr = creaArreglo(new String[0], 10);
10 }
```

Referencias

Genéricos (en detalle)

https://docs.oracle.com/javase/tutorial/java/generics/index.html

Genéricos (directo a lo avanzado)

https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Licencia