POO-TRABAJO TEÓRICO

junio, 2003

TIPOS GENÉRICOS EN JAVA

Celia Fontanillo Fontanillo

Antonio Garrote Hernández

Departamento de Informática y Automática
Universidad de Salamanca

Información de los autores:

Celia Fontanillo Fontanillo
Estudiante de Ingeniería Técnica en Informática de Sistemas
Facultad de Ciencias – Universidad de Salamanca
cfontanillo@eresmas.com

Antonio Garrote Hernández
Estudiante de Ingeniería Técnica en Informática de Sistemas
Facultad de Ciencias – Universidad de Salamanca
tonino@beograd.com

Resumen

En este documento se pretenden presentar los tipos genéricos en Java que están siendo desarrollados para su pintroducción en la próxima versión del lenguaje, Java 1.5.

La primera parte del trabajo está pensada como una introducción general de los tipos genéricos al lector, por si no está familiarizado. Se realiza para ello una breve descripción de los mismos, mostrando sus ventajas basándose en ejemplos, además de hacer una revisión histórica de la evolución de la especificación hasta el momento actual y de los motivos que han impulsado a introducir tipos genéricos en Java.

En la segunda parte se trata a cerca de las limitaciones en la implementación y de las características de la nueva especificación desarrollada, comentándose también las principales diferencias entre los tipos parametrizados en C++ (plantillas) y los tipos genéricos en Java.

Por último se presentan las especificaciones así como la sintaxis de los tipos genéricos publicada por el grupo encargado de su desarrollo, el JSR14.

Abstract

Generics Java, which is now being developed to its insertion into the next language version (Java1.5), is expected to be presented in this document.

The first part is thought as a general introduction to Generics in case the reader isn't used to them. A short description is carried out, and its advantages are showed having grounds in examples. Also, a historical revision of the evolution of the specification until the most recent events is done, presenting the main reasons which have led to the introduction of generics in Java.

The second part describes the limitations in the implementation and the main characteristics of the new specification. After all this, generic types in C++ (templates) and generics in Java are compared, showing the differences between them.

And finally, specifications as well as generics syntax are presented. The specifications included in this document are the ones extracted from the JSR14, the group in charge of adding generic types in Java.

Tabla de Contenidos

1.Introducción a los tipos Genéricos

1.1.¿Qué son tipos genéricos?

En el momento de escribir una clase se debe conocer con que tipo de datos va a interactuar, sin embargo esto no siempre es conocido. Por lo tanto se debería poder definir una clase con la ayuda de un "contenedor" al cual nos referimos como si fuera el tipo sobre el que opera la clase. La definición actual de la clase es creada una vez que declaramos un objeto en particular.

Los tipos genéricos, también llamados tipos parametrizados, permiten definir un tipo sin especificar todos los tipos que éste usa. Estos serán suministrados como parámetros en punto de instanciación. Los cambios son realizados por tanto en tiempo de compilación.

Los tipos parametrizados se utilizan especialmente para implementar tipos abstractos de datos: pilas, colas, anillos, bolsas y otros que permiten almacenar distintos tipos de elementos según sean instanciados en tiempo de compilación.

Los tipos genéricos han sido usados por otros lenguajes durante años, como por ejemplo la plantillas en C++, tipos genéricos en Ada, polimorfismo paramétrico en ML y Haskell, y ahora, debido a la demanda popular , los tipos genéricos están siendo añadidos en Java 1.5.

1.2.Cómo se programan los tipos genéricos en la versión actual de Java.

Con los tipos de genéricos, los tipos que contienen datos, como por ejemplo las listas, no están definidas para operar sobre tipos de datos específicos sino que operan sobre un conjunto homogéneo donde el tipo del conjunto es definido en la declaración. Sin embargo sin los tipos genéricos es mucho más difícil lidiar con los tipos de datos que contienen otros tipos.

Para poder hacer esto es necesario declarar una lista que acepte objetos de tipo Object. Puesto que en Java todas las clases heredan de Object la clase Lista (List) puede almacenar cualquier tipo de elementos mediante una sustitución polimórfica.

Luego, al recuperar estos elementos de la lista, es necesario añadir los *castings* pertinentes dependiendo del tipo de dato que haya sido introducido en ella.

Un ejemplo que orienta el manejo de tipos que contienen a otros tipos de datos podría ser el siguiente. En él se declara una lista de objetos de tipo Object en la que introducimos un número

de enteros obteniendo lo que ha sido introducido posteriormente. Esto, como ya ha sido comentado, ha sido posible gracias a la sustitución polimórfica al heredar en Java todos los objetos de la clase Object.

```
List integerList = new LinkedList();
integerList.add( new Integer(1));
integerList.add( new Integer(2));
Iterator listIterator = integerList.iterator();
While(listIterator.hasNext())
{
      Integer item = (Integer) listIterator.next();
}
```

1.3.Problemas de la forma de programar actual.

En el método usado en la versión actual de Java se pueden observar algunos problemas:

- *El programador debe recordar que tipo de elemento hay almacenado en la lista* y realizar el *casting* al tipo apropiado cuando lo extraiga de la lista. Extraer un elemento de una lista requiere por tanto dos *castings*.

- *No hay comprobación de tipos en tiempo de compilación*. Para el compilador los elementos de todas las listas son de tipo Object por lo que no se pueden comprobar los tipos en tiempo de compilación.

- *Necesaria comprobación explicita de tipos en tiempo de ejecución*. Al no diferenciar los tipos en tiempo de compilación se hace necesaria una comprobación de los mismos en tiempo de ejecución para poder detectar posibles errores de tipos. Esto significa que si el desarrollador confunde las dos listas y hace un *casting* ilegal en un elemento el error no será detectado hasta el tiempo de ejecución.

- *Aparición de excepciones (ClassCastException) en tiempo de ejecución*. Al aparecer en tiempo de ejecución se hace mucho más difícil la eliminación de los errores de *casting* que los provocan que si aparecieran en tiempo de compilación.

- *Se permite la existencia de clases contenedoras con objetos heterogéneos* lo que dará problemas al intentar recuperar los elementos de las listas.

En el siguiente fragmento de código se muestras algunos ejemplos de los errores anteriores.

```
1..
2.  List stringList  = new LinkedList();
3.  List integerList = new LinkedList();
4.
5.  integerList.add(new Integer(1));
6.  integerList.add(new Integer(2));
7.
8.  stringList.add(new String("I am a String"));
9.
10. // Nada restringe los elementos a un conjunto homogéneo.
11. stringList.add(new Integer(1));
12.
13. Iterator listIterator = integerList.iterator();
14.
15. // El compilador no se da cuenta del tipo de retorno de la lista y
del casting ilegal.
16. // El desarrollador pretende iterar en la lista de cadenas.
17. while(listIterator.hasNext()) {
18.
19.   // Casting ilegal detectado en tiempo de ejecución.
20.   String item = (String)listIterator.next();
21. }
22.
23. listIterator = stringList.iterator();
24. // No garantiza un contenedor homogéneo.
25. while (listIterator.hasNext()) {
26.   //Fallo en tiempo de ejecución debido a que el conjunto no es
heterogéneo.
27.   String item = (String)listIterator.next();
28. }
29..
```

De la línea 13 a la 20 el programador puede estar pensando que trabaja con una lista de objetos Integer cuando en realidad está trabajando con una lista de Strings siendo mostrado el error en tiempo de ejecución.

En la línea 11 y de la 22 a la 27 el programador puede pensar que trabaja con un conjunto homogéneo de String cuando en realidad está trabajando con un conjunto heterogéneo de String e Integer mezclado al haber introducido enteros en la lista que originalmente había sido declarada para cadenas. Así que a menos que se cree una nueva subclase lista para cada tipo de elemento (lo cual disminuiría las ventajas de la reutilización de la OO), no hay ningún otro modo de poder manejar estáticamente un conjunto heterogéneo.

Este ejemplo es muy sencillo y permite detectar los errores con rapidez, sin embargo estos errores en un programa mayor serían muy difíciles de detectar al producirse en tiempo de ejecución.

1.4. Soluciones y ventajas derivadas del uso de tipos genéricos.

Algunos de los beneficios de los tipos genéricos en Java son:

- *Comprobación estricta de tipos* manteniendo la misma flexibilidad que el enlazado dinámico. Con tipos genéricos se puede alcanzar un polimorfismo similar al usado en el ejemplo anterior pero con una comprobación estricta de tipos que permite detectar errores en tiempo de compilación. El compilador conoce que los tipos de listas son diferentes porque contienen distintos elementos. Los errores al ser detectados en tiempo de compilación son mucho más fáciles de detectar que los errores en tiempo de ejecución.

- *No es necesaria la comprobación de tipos en tiempo de ejecución* , lo que redunda en un código con menos *castings* y por lo tanto más legible y menos propenso a errores. En lugar de confiar en la memoria del usuario los parámetros marcan el tipo de los elementos obtenidos de la lista.

- Los tipos genéricos *garantizan que las listas contienen solo un conjunto homogéneo de elementos* eliminando los errores derivados de la aparición de listas heterogéneas.

- *Hace que el código sea menos ambiguo y más fácil de mantener.*

El ejemplo mostrado a continuación es una traducción del ejemplo previo utilizando las extensiones de tipos genéricos a Java que están siendo desarrolladas para ilustrar las ventajas derivadas de su uso.

```
1..
2.                  import                java.util.LinkedList;
3.                 import               java.util.Collections;
4.                   import                  java.util.Iterator;
5.
6.            public          class          genericsExample2{
7.
8.     static    public    void    main(String[]    args)    {
9.          LinkedList<String>       stringList       =      new
LinkedList<String>();
10.           LinkedList<Integer>     integerList     =     new
LinkedList<Integer>();
11.
12.                    integerList.add(new         Integer(1));
13.                    integerList.add(new         Integer(2));
14.
15.        stringList.add(new   String("I   am   a   String"));
16.    stringList.add(new Integer(1)); // provoca un error de
compilación
17.
18.      /*  genericsExample2.java:16:  cannot  resolve  symbol
19.        **   symbol   :   method   add   (java.lang.Integer)
20.                                                          */
21.
22.               Iterator<Integer>       listIterator       =
integerList.iterator();
23.                                  String                item;
24.                    while(listIterator.hasNext())         {
25.        item = listIterator.next(); // provoca un error de
compilación
26.
27.          /*  genericsExample2.java:25:  incompatible  types
28.                    **     found     :     java.lang.Integer
29.                      **      required:      java.lang.String
```

```
  30.                                                              */
  31.                                                               }
  32.
  33.     listIterator = stringList.iterator(); // provoca un
error                      de                       compilación
  34.
  35.       /*  genericsExample2.java:33:  incompatible  types
  36.       **  found  :  java.util.Iterator<java.lang.String>
  37.       **  required:  java.util.Iterator<java.lang.Integer>
  38.                                                              */
  39.    // se garantiza que el iterador recorre una lista de
elementos                                              homogéneos
  40.               while      (listIterator.hasNext())      {
  41.                   item       =       listIterator.next();
  42.
  43.          /*   genericsEx2.java:41:   incompatible   types
  44.               **      found      :      java.lang.Integer
  45.                 **       required:       java.lang.String
  46.                                                              */
  47.                                                               }
  48.                 }                 //                 main
  49. } // class genericsExample2
```

Comparando los dos ejemplos se observa que ha sido incluida información adicional sobre los tipos de datos en el código usando tipos genéricos, lo que lleva al compilador a conocer que tipo contiene cada contenedor de elementos.

Se puede observar también en el ejemplo que todos los errores de *casting* provocan una excepción en tiempo de compilación con lo que resultan muy sencillos de detectar y corregir.

2.Evolución temporal de la especificación

2.1.Motivos para iniciar el desarrollo de tipos genéricos en Java

La ausencia de soporte para el uso de tipos genéricos fue una de las principales críticas que se realizaron al lenguaje Java desde sus primeras versiones. Esta característica estaba pensada originalmente para formar parte de la especificación del lenguaje, pero debido a la falta de

tiempo y a la complejidad e inmadurez de la propuesta realizada por Gosling y Joy, autores de la especificación del lenguaje, la inclusión de genéricos no pudo ser posible.

El interés porque Java incluyese un soporte para los tipos genéricos siempre ha sido sin embargo muy alto. Buena prueba de ello son las propuestas para su inclusión que se realizaron en distintas conferencias como la ECOOP y la OOPSLA, otra prueba es que dentro del programa *"bug parade*" (http://developer.java.sun.com/developer/bugStats/top25rfes.shtml), puesto en marcha por Sun para que los usuarios de Java votasen cuales son los fallos o limitaciones del lenguaje que más les gustaría ver solucionadas, una propuesta para incluir tipos genéricos acabo entre las veinticinco más votadas.

Como respuesta a este interés en 1996 un grupo de programadores bajo el nombre de "*Pizza Group"* comienza el proyecto GJ *("Generic Java*") (http://www.research.avayalabs.com/user/wadler/gj/) con el fin de incluir esta característica al lenguaje.

En mayo de 1999, Sun hace una propuesta para la inclusión de los tipos genéricos dentro del lenguaje Java en el marco del "*Java Community Process (JCP)",* que se basa en el trabajo realizado por el proyecto GJ. La propuesta es aceptada y se crea un grupo de trabajo, el "*Java Specification Request 14 (*JSR14)" (http://jcp.org/en/jsr/detail?id=14) para que lleve a cabo la nueva especificación. El grupo estará dirigido por Gilad Bracha miembro del proyecto GJ.

Los trabajos del JSR14 van pasando por las fases especificadas por el JCP, de tal modo que en mayo de 2001 está disponible un prototipo de compilador que soporta los tipos genéricos definidos en la especificación que realiza el JSR14, especificación que finalmente será publicada en un documento de revisión pública en agosto de 2001.

Desde entonces el JSR14 ha realizado labores de actualización de la especificación, que se espera sea incluida en la próxima gran revisión del lenguaje Java, la versión 1.5 del JDK.

2.2. Desarrolladores actuales

El JSR14 esta compuesto por las siguientes personas:

- Líder de la especificación: Gilad Bracha (Sun Microsystems).
- Grupo de expertos:
  - Stefan Marx (Borland Software Corporation).
  - Martín Odersky (IBM).

- Kersten Krab Thorup (A.G. Software).

3. Limitaciones a la implementación de tipos genéricos en Java

La principal restricción que se impuso al JSR14 a la hora de elaborar su especificación de tipos genéricos, era que el nuevo código genérico debería funcionar perfectamente con el viejo código que no soportaba todavía esta característica.

Por ejemplo, código antiguo programado para usar las clases "`collection`" no genéricas, debería funcionar perfectamente con las nuevas clases "`collection`" con soporte para la genericidad, esto es posible mediante el uso de los llamados tipos crudos ("*raw types*") como veremos más adelante, ya que la nueva implementación de la máquina virtual no necesita información extra sobre el tipo de los objetos. Así, una llamada al método `getClass()` sobre un contenedor genérico, no indicará el tipo que guarda el contenedor. Esta restricción nos permite sin embargo que el código antiguo sea compilable sin ninguna modificación.

4. Características

Las principales características de la especificación de los tipos genéricos en Java realizada por el JSR14 son:

- Permite el polimorfismo restringido, es decir, se puede especificar por ejemplo, que un parámetro deba implementar una determinada interfaz para poder ser un parámetro válido.
- Permite el polimorfismo f-restringido, es decir, se puede especificar que un parámetro deba implementar una interfaz que a su vez es también genérica.
- No permite los "*mix-ins*", un parámetro no se puede especificar como un supertipo.
- Permite el uso de métodos genéricos.
- Mínimos cambios en la máquina virtual.
- Traducción homogénea. Se compila la clase genérica en un único archivo .class con extensiones sólo para el compilador. Esto tiene como ventaja que sólo es necesario un archivo en disco y una única instancia en tiempo de ejecución pero al precio de penalizaciones en la

velocidad de ejecución, ya que no se puede optimizaciones de velocidad en el código y la necesidad de "*castings*".

- Soporte limitado a la reflexión como se ha visto anteriormente.
- No se permite la instanciación de tipos genéricos con sustitución de parámetros por tipos primitivos.
- No permite sustitución de parámetros por "no tipos": constantes, "functors"…
- Garantiza la compatibilidad con código antiguo no preparado para el uso con tipos genéricos.

5. Diferencias entre los tipos genéricos en Java y las plantillas de C++

Los tipos genéricos de Java no son plantillas, hay una serie de diferencias entre ambos enfoques:

En la declaración de un tipo genérico en Java existe comprobación de tipos no así en C++, en Java los tipos genéricos se compilan una sola vez, el código del tipo no se muestra al usuario del tipo.

En C++ se utiliza la "traducción textual" para instanciar los tipos, la declaración de la plantilla se utiliza como una macro, por lo tanto se necesita el código fuente de la plantilla para su instanciación, cada vez que se utiliza una plantilla esta se recompila.

Esta aproximación es "grande y rápida" ya que cada instanciación de la plantilla consume almacenamiento primario y secundario, sin embargo son posibles optimizaciones de velocidad en tiempo de ejecución. También permite un completo soporte para la reflexión. Por el contrario la aproximación usada en la especificación de tipos genéricos en Java se utiliza la "traducción homogénea", esta es una aproximación "pequeña y lenta" sólo se requiere un archivo "*bytecode*" por tipo genérico, con el ahorro de memoria y además se oculta el código fuente del tipo, pero imposibilita las optimizaciones de velocidad y se limita el soporte a la reflexión.

6. Especificación del JSR14

6.1. Sintaxis de tipos:

tipo parametrizado: C <T1,…,Tn>

Identificadores no cualificados introducidos por la declaración de una clase o una interfaz parametrizada y declaraciones de tipos polimórficos.

Donde:

C ::= clase o interfaz parametrizada.

<T1, … ,Tn> ::= lista de parámetros que deben coincidir con el número de parámetros declarados en la sección de parámetros de la clase o interfaz parmetrizada. El tipo de los parámetros de la lista debe ser un subtipo de las restricciones expresadas en la sección de parámetros.

Los tipos primitivos no pueden ser especificados como parámetros.

e.g:

```
MiClase<A,B>


MiClase<String,MiClase<A,B>>

MiClase<String,Integer>

MiClase<int,String> //error tipo básico no permitido como parámetro

MiClase<String,Integer,Bolean> //error número de parámetros
```

Una clase o interfaz parametrizado define todo un conjunto de tipos para cada posible implementación de tipos en la lista de parámetros. Todo este conjunto de tipos comparten la misma clase o interfaz en tiempo de ejecución:

```
MiClase<Integer,String> x = new MiClase<Integer,String>();

MiClase<Boolean,Boolean> y = new MiClase<Boolean,Boolean>();

Return (x.getClass() == y.getClass()); //devuelve true
```

La sección de parámetros sigue al nombre de la clase o interfaz parametrizado y consisten en una sucesión de parámetros limitados por los caracteres “<” y “>”.

Cada parámetro puede llevar asociada una restricción que puede consistir en una clase o interfaz y opcionalmente una serie de interfaces.

Si no se especifica ninguna interfaz se considera `java.lang.Object` como restricción por defecto.

Si un parámetro tiene más de una restricción, entonces no se puede acceder a atributos o métodos de ese parámetro, que no sean del tipo `Object`, sin realizar un *casting* explícito al tipo del método o atributo.

El ámbito de un tipo de parámetro es el de toda la clase pero incluyendo la sección de parámetros, por lo que pueden aparecer como parte de sus propias restricciones o como restricciones de otros tipos de parámetros en la sección de parámetros.

e.g:

```
interface ConvertibleA<A> (abre llave)
      A convertir();
}
    MiClase<B implements ConvertibleA<A>,A implements
ConvertibleA<B>>
```

Las declaraciones parametrizadas pueden estar anidadas dentro de otras declaraciones.

Como los mecanismos de *throw* y *catch* funcionan sólo con tipos no parametrizados es obligatorio que los tipos parametrizados no hereden ni directa ni indirectamente de `java.lang.Throwable`.

Ha sido necesario refinar la gramática del lenguaje para que no haya confusiones por parte del analizador sintáctico entre el final de una sección de parámetros con tipos parametrizados y los operadores “>>” y “>>>”.

6.2. Subtipos, supertipos y tipos miembro.

Dada una clase o interfaz parametrizado C, con una lista de parámetros A1 … An y con las restricciones B1, … , Bn, esa clase o interfaz define un conjunto de tipos C<T1, … , Tn>, donde cada argumento Ti incluye todos los tipos que son subtipos de la restricciones Bi. Es decir, para cada restricción tipo Si en Bi, Ti es un subtipo de:

Si [T1 := A1, … , Tn:=An]

Donde T:=A significa substitución directa del tipo variable A por el tipo T.

Esta definición incluye lo tipos de variable y los tipos parametrizados.

Dada una clase o interfaz parametrizado C<A1, … , An> sus supertipos directos son:

- El tipo dado en la cláusula `extends` de la declaración de la clase o el tipo `java.lang.Object` si esta no está presente.
- El conjunto de tipos dado en la clausula `implements` de la declaración de la clase si esta está presente.

Los supertipos directos de un tipo C<T1, … , Tn> son D<U1theta, … , Un theta> donde:

- D<U1, … , Uk> es supertipo directo de C<A1, … , An>
- Theta es la substitución [T1:=A1, … ,Tn:=An]

Los supertipos de un tipo variable son los tipos de su restricción .

Los supertipos de un tipo se obtienen mediante el cierre transitivo del conjunto de supertipos. Los subtipos de un tipo T son todos los tipos U tales que T es supertipo de U.

Los subtipos no se heredan a través de los tipos parametrizados.

e.g:

Si T es un subtipo de U no implica que C<T> sea un subtipo de C<U>

Para permitir la traducción directa a través de la eliminación de tipos, se impone la restricción de que una clase o variable tipo no puede ser al mismo tiempo extensión de dos interfaces tipos que son diferentes parametrizaciones de la misma interfaz.

e.g:

```
class B implements InterfaceA<Integer>

class C extends class B implements
InterfaceA<Integer> // error herencia
```

Consecuencia del concepto de tipos parametrizados, es que el tipo de las variables miembro en la definición de una clase parametrizada no es fijo, depende de la parametrización de la clase en un tipo.

e.g:

Sea C<A1, … ,An> y M un miembro de C cuyo tipo se ha declarado en la parametrización T. Entonces el tipo de M en el tipo C<T1, … ,Tn> es T[A1:=T1, … ,An:=Tn]

6.3. Tipos “crudos”

Los tipos “crudos” son un mecanismo por el cual se permite el uso como tipos válidos de tipos parametrizados mediante el borrado de sus parámetros. Estos tipos son utilizados para permitir el uso de código antiguo que no usa parámetros.

Es posible por ejemplo asignar un tipo `Vector<String>` a un tipo `Vector`. La asignación inversa es posible pero no se recomienda ya que no es segura desde el punto de vista del uso de genéricos.

Los supertipos y subtipos de un tipo “crudo” son las versiones “crudas” del supertipo y subtipo parametrizado.

El tipo de un miembro M de la clase parametrizada es borrado y el acceso a estos miembros provocará mensajes de advertencia del tipo “*unchecked*”.

e.g:

```
class Cell<A> (abre llave) {
 A value;
 Cell (A v) { value = v; }
 A get() { return value; }
 Void set(A v) { value = v; }
}

Cell x = new Cell<String>(“abc”);
x.value;    //Correcto, tiene tipo Object
x.get();    //Correcto tiene tipo Object
x.put(“def”);    //deprecated
```

6.4. Métodos genéricos

Los métodos pueden tener parámetros al igual que las clases e interfaces.

La sección de parámetros de los métodos se sitúa delante del tipo de retorno en la declaración del método.

e.g:

```
 static <E> void mostrarPorPantalla(E[] array) {
       for(int i =0;i<array.length;i++) {
            System.out.println(array[i]);
       }
}
interface Formateado {
 void mostrarConFormato();
}
<E implements Formateado> void mostrarConFormato(E[]
array) {
       for (int i=0;i<array.length;i++) {
       array[i].mostrarConFormato();
      }
}
```

No está permitido declarar dos métodos genéricos con el mismo número de parámetros y con los mismos tipos en las restricciones y dará lugar a un error del tipo "*have the same argument types*".

La sobrescritura de métodos se ha relajado en la especificación de tipos genéricos. Ahora se permite que la sobrescritura de un método cambie el tipo de retorno del método que sobrescribe.

e.g:

```
//Estas declaraciones son legales según la especificación del JSR14 pero
//ilegales para la JLS (Java Language Specifacation)
class C implementa Clonable {
 C copy() { (C) return clonar(); }
 …
```

```
    }
    class D extends C implements Clonable {
    D copy() { (D) return clonar(); }
…
}
```

También se han relajado las reglas del lenguaje para las clases abstractas que implementan interfaces, ahora una clase puede heredar múltiple métodos con el mismo nombre y argumentos siempre y cuando: sean todos abstractos, la clase no los herede o los sobrescriba y uno de los métodos tenga un tipo de retorno que sea subtipo de todos los demás tipos de retorno.

6.5. Excepciones

No se permite el uso de parámetros en las cláusulas `catch` pero sí se permite su uso en las listas de parámetros `throws`.

e.g:

```
//ejemplo de cláusulas throws genéricas:
interface ExcepcionDeAccionPriviliegiada <E extends
 Exception> {
 void run() throws E;
}
class ControladorDeAcceso {
 public static <E extends Exception>  Object
    hacerAccionPrivilegiada(
 ExcepcionDeAccionPrivilegiada<E> accion) throws  E {
 … }
}
class PruebaExcepciones {
 public static void main ( String args[]) {
      try {
```

```
    ControladorDeAcceso.hacerAccionPrivilegiada (
                    new
ExcepcionDeAccionPrivilegiada<FileNotFoundException>() {
                        public      void      run      throws
FileNotFoundExecption {
                                … Se borra un archivo …
                        } } );
                } catch ( FileNotFoundException e) {
                    e.getStackTrace();
                }
        }
}
```

6.6. Expresiones

6.6.1. Instanciación:

Una expresión de instanciación de clase para una clase parametrizada consiste en el identificador completo del tipo que va a ser creado y argumento para los constructores de ese tipo.

e.g:

```
new Vector<String>(new String("hola"));
```

6.6.2. Arrays:

El tipo de elemento en la creación de arrays genéricos es un tipo parametrizado, intentar utilizar una variable da como resultado una advertencia del tipo "*unchecked*".

e.g:

```
new Vector<String>[10];
new Seq<Carácter>[10][20][];
```

6.6.3. Castings:

Se aplican las reglas comunes de los *castings* en Java, pero con el inconveniente de que la información del tipo de los parámetros no es mantenida en tiempo de ejecución y por tanto la corrección de los mismos debe ser determinada estáticamente. Para ello se refinan las reglas de *casting* de modo que un tipo S puede ser convertido a un tipo T si una de las dos condiciones siguientes se cumple:

- S es un subtipo de T y no hay otros subtipos de S con el mismo borrado que T
- T es un supertipo de S, conversión segura por las restricciones a la herencia de multiples interfaces.

e.g:

Dado:

```
class Diccionario<A,B> extends Object  (abre llave) …
};

class HashTable<A,B> extends Disccionario<A,B> (abre
llave) … };

Diccionario<String,Integer> dic;

Object o;
```

Entonces:

```
(HashTable<String,Integer>)dic;  //correcto,  es  un
supertipo

(HashTable<Integer,Integer>)o //incorrecto, no existe
el subtipo
```

6.6.4. Invocación de métodos genéricos:

La invocación de métodos genéricos no tiene una sintaxis especial, se hace la llamada omitiendo la sección de parámetros y la sustitución de estos se hace de forma automática según la declaración del método.

6.6.5. Operador de comparación de tipos:

Las operaciones de comparación de tipos pueden involucrar también tipos genéricos.

e.g:

```
class Seq<A> implements List<A> {
   static boolean isSeq(List<A> x) {
         return  x instanceof  Seq<A>
   }
   static boolean isSeq(Object x) {
         return x instanceof Seq
   }
   static boolean isSeqArray(Object x) {
         return x instanceof Seq[]
   }
}
```

6.7. Inferencia de tipos de parámetros

La inferencia de los tipos de los argumentos en la llamada a un método genérico, se obtienen de los valores de los parámetros de la llamada.

Mediante la inferencia de tipos se inserta el tipo más específico de los parámetros tal que:

1. Cada argumento real es un subtipo del argumento formal.

2. Ningún tipo de variable que ocurre más de una vez en el tipo resultado del método es instanciado en un tipo que contiene el tipo ***,** donde el tipo * es el tipo nulo que no puede ser escrito en ningún programa Java pero que se utiliza en la especificación para explicar la inferencia de tipos. El tipo * es un subtipo del tipo de cualquier referencia.

3. Se produce un error si los tipos más específicos de los parámetros que se determinan no existen o no son únicos.

Sobre la segunda restricción mencionada anteriormente, decir que algunas reglas de covarianza se aplican a los tipos que contienen el tipo **.** Para cualquier tipo de contexto TC, tipo U, TC[*] es un subtipo de TC[U].

La covarianza general conduce a sistemas de tipos poco seguros. Por lo tanto hay que discutir con cuidado como este sistema de tipos con covarianza general sigue siendo seguro.

La argumentación es la siguiente: como nadie puede declarar variables de tipo TC<*>, todo lo que se puede hacer con un valor de ese tipo es pasarlo o asignarlo una vez a una variable o a un parámetro de algún otro tipo.

Aparecen ahora tres posibles situaciones dependiendo del tipo de la variable o parámetro al que se le asigna el valor:

- El tipo es un supertipo desparametrizado del tipo crudo TC. En este caso la asignación es claramente segura.

- El tipo de la variable es TC'<T>, para algún tipo T y algún supertipo TC' de TC. Ahora el único valor de * es null, que es también de un valor de cualquier referencia de tipo T. Por lo tanto, cualquier valor del tipo TC<*> es también un valor del tipo TC'<T> y la asignación es segura.

- El valor es un parámetro p cuyo tipo es un tipo variable A. Entonces el código que accede a p en el cuerpo del método no es sensible al tipo del parámetro real, por lo tanto no existe peligro de que el código del método por sí mismo de lugar a errores de tipo. Mas aún, gracias a las restricciones anteriores se garantiza que el tipo resultado del método contendrá como mucho una ocurrencia del tipo A, así que el tipo real del método aplicación es de nuevo de la forma TC'<*> donde TC' es un tipo contexto.

Sin la segunda restricción la seguridad de tipos quedaría en entredicho.

6.8. Traducción de tipos

Para la traducción de un tipo genérico el compilador calcula el "borrado de tipos" (*type erasure*) de un tipo genérico. Este borrado de tipos consiste en calcular todos los posibles tipos no genéricos que se pueden obtener del tipo genérico.

Si |T| es el borrado de tipos de un tipo T(T1, … ,Tn) entonces:

- El borrado del tipo T.C es |T|.C.

- El borrado de l tipo T[] es |T|[].

- El borrado de un tipo variable es el borrado de las clases de sus posibles restricciones y si la restricción consiste sólo en interfaces, el borrado consiste en el interfaz entre todos los borrados de las interfaces que tenga el mínimo nombre canónico usando ordenación lexicográfica.

- El borrado de cualquier otro tipo es el mismo tipo.

6.9. Traducción de métodos

Cada método T(T1, … ,Tn) throws S1, … ,Sm es traducido por el compilador en un método con el mismo nombre cuyo tipo de retorno, tipos de argumentos y tipos de la cláusula `trhows` son el borrado de los tipos en el correspondiente método genérico. Además si el método m pertenece a una clase o interfaz C que es heredado por otra subclase D, quizás sea necesario la generación de un nuevo "método puente" en D. Las reglas precisas son las siguientes:

- Si un método C.m es sobrescrito por un método D.m y el borrado del tipo de retorno de C.m es diferente al de D.m, entonces un método puente tiene que ser generado.

- Un método puente también tiene que ser generado si no existe un método D.m que sobrescriba directamente al método C.m, excepto en el caso de que C.m sea abstracto.

El tipo del método puente es el tipo del borrado del método en la clase o interfaz base C. En el cuerpo del método puente todos los parámetros del método seran convertidos al borrado del tipo de la clase que hereda D, después de lo cual cualquier llamada sera pasada al método sobrescrito D.m (si existe) o al método original C.m (en cualquier otro caso). No se necesitan cambios extras en el manejo de los tipos de las cláusulas "`trhown`", ya que estas no son comprobadas en tiempo de carga o ejecución.

Esta forma de traducción puede crear métodos con el mismo nombre y tipos de argumentos pero con diferentes tipos de retorno dentro de la misma clase. Un compilador podría rechazar esta doble declaración de métodos, pero su representación en formato "*bytecode*" es correcta ya que en este formato se referencia a los métodos usando su signatura completa, pudiéndose distinguir entre las ocurrencias repetidas por su tipo de retorno.

La misma técnica es usada para implementar métodos sobrescritos con tipos de retorno covariantes.

Como nuestra traducción de métodos borra los tipos, es posible que métodos con diferentes tipos pero el mismo nombre sean "borrados" en el mismo tipo.

Este caso se considera un error en el código original y debe ser rechazado por el compilador Java. Hay tres reglas para evitar problemas con la signatura causados por la traducción. Sea un método M que sobrescribe indirectamente la declaración de otro método M', Si hay una (posiblemente vacía) ruta entre las declaraciones de los métodos:

M = M0,M1,…,Mn = M'

Tal que el método Mi sobrescribe al método Mi+1, tenemos que:

1. Métodos declarados en la misma clase con el mismo nombre deben tener "borrados" diferentes.

2. Si el método M en la clase C tiene el mismo nombre y el mismo "borrado" que el método M' de la superclase D de C, entonces el método M debe sobrescribir indirectamente el método M' en D.

3. Si una declaración de método M en una superclase D de C tiene el mismo borrado de tipo que un método M' de una interfaz I implementada por C, entonces debe haber una declaración de método M'' en C o en una de sus superclases que indirectamente sobrescriba M en D e implemente M' en I.

6.10. Traducción de expresiones

La traducción de expresiones se realiza de acuerdo con lo especificado en la "*Java Language Specificacition*" excepto por el hecho de que se introducen conversiones de tipo en aquellos lugares donde resulte necesario:

- Accesos a campos donde el miembro accedido sea un tipo parámetro.
- Retorno de métodos donde el tipo devuelto sea un tipo parámetro.

7. Conclusiones

Después de realizar este recorrido por la futura especificación de los tipos genéricos en Java, podemos destacar los siguientes aspectos: En primer lugar, hemos observado las ventajas que el uso de genéricos aporta al desarrollo de código Java, haciéndolo más legible y seguro, pero el uso de genéricos no se limita solamente a estos aspectos beneficiosos, su mayor importancia radica en que permite que el uso de una herramienta de diseño, tan poderosa como la genericidad ,tenga soporte directo en el lenguaje. De esta manera, problemas de diseño, que anteriormente tenían que ser abordados de forma indirecta dando lugar a soluciones poco

satisfactorias, puedan ser resueltos mediante soluciones de diseño simples y elegantes con el uso de la genericidad. Buena prueba de lo anterior es el rediseño de las clases colección de Java.

Sin embargo, en el mismo ejemplo de las clases colección podemos ver también el que a nuestro juicio es el punto más débil de esta especificación, la necesidad de mantener la compatibilidad con el código antiguo. Esta restricción ha determinado, como se ha visto, el tipo de mecanismo para realizar la implementación de los tipos genéricos, la traducción homogénea. Esto ha significado tener que renunciar a características tan importantes como el soporte a la reflexión.

Habiéndose examinado en la realización de este trabajo distinto métodos para implementar la genericidad en lenguajes de programación, así como las características teóricas que no se han incluido en esta especificación, encontramos que aunque el precio pagado por las restricciones derivadas de una adopción tardía de la genericidad es elevado, la solución a la que se ha llegado es muy satisfactoria, resistiendo la comparación con un enfoque opuesto al mismo concepto como son las plantillas de C++.

En definitiva, la especifiación realizada por el JSR-14 viene a resolver una de las carencias más demandadas por la comunidad de desarrolladores Java, solución que se ha realizado dentro del marco de esa misma comunidad, de una forma eficiente que beneficia tanto a los desarrolladores como al futuro del mismo lenguaje.

8. Bibliografía recomendada

- **Adding Generics To The Java Programming Language: Participant Draft Specification**, JSR14 Abril 2001: http://objectclub.esm.co.jp/evolutionalDesignWithPatterns/
- **Preparing for Generics**, Paul Mingardi, Noviembre 2002: http://developer.java.sun.com/developer/technicalArticles/releases/generics/

- **Generics in Java and Beyond**, Martín Büchi, 2001: http://www.ociweb.com/javasig/knowledgebase/2001May/JavaGenerics.pdf
- **Java One Sun's 2001 Worlwide Java Developer Conference**, Adding Generics To The Java Programming Language, Gilad Bracha: http://servlet.java.sun.com/javaone/conf/sessions/2733/0-sf2001.jsp
- **JSR-14 public draft**: http://www.jcp.org/aboutJava/communityprocess/review/jsr014/index.html
- **GJ website y tutorial**: http://www.cs.bell-labs.com/who/wadler/pizza/gj
- **OOPSLA 98 GJ paper**: http://www.research.avayalabs.com/user/wadler/topics/gj.html
- **Construcción de software orientado a objetos** 2ª Edición Bertrand Meyer., PRENTICE mayo 1999.
- **El Lenguaje de Programación C++**, Bjarne Stroustrup PEARSON EDUCACIÓN 2002.

Apéndice A:

El prototipo experimental de compilador

Existe actualmente un prototipo experimental de compilador que produce archivos *bytecode* a partir de código que cumpla con la especificación actual contenida en el *public review draft specification* documento publicado por el JSR-14.

Los archivos generados son ejecutables en cualquier máquina virtual cuya versión sea superior a la 1.3.

Para instalar el compilador es necesario tener instalada la plataforma J2SE-1.4

El compilador esta disponible para su libre descarga desde los repositorios de Sun, en su sección *Java developers connection* siendo únicamente necesario registrarse para proceder a la descarga.

El prototipo se actualiza continuamente (la última actualización data del 23 de mayo de 2003), y está pensado para que los usuarios que lo descarguen puedan a través de su experiencia y comentarios colaborar en el desarrollo del proyecto. Es por esto que el prototipo soporta

características que ni siquiera forman parte de la especificación actual del JSR-14, como la varianza de tipos, y otras mejoras al lenguaje Java que están siendo desarrolladas por diferentes JSR, en este caso las propuestas por el JSR-201(*Extending the Java(TM) Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import*). Los autores del prototipo tampoco pueden garantizar por lo tanto compatibilidad con el compilador definitivo que se pretende sea incluido en el JDK1.5.

La distribución del compilador esta formado por una serie de archivos: el código del compilador, una serie *scripts* para la generación del compilador, una serie de clases que deben sustituir a clases de la plataforma estándar que no soporta genericidad, las nuevas clases colección genéricas y las fuentes de estas clases, además de distintos ejemplos de código donde se usa la genericidad listos para ser compilados. También se incluye distintos archivos con documentación así como una copia del documento de especifcación del JSR-14.

La instalación se realiza mediante la ejecución del *script bootstrap* una vez se ha definido una variable de entorno J2SE14 para que apunte al directorio donde se instalado la distribución de la plataforma J2SE-1.4.

Una vez terminada la instalación la compilación del código genérico se realiza normalmente y para su ejecución basta con utilizar el *flag* "–source 1.5".