CFGS APLICACIONES MULTIPLATAFORMA - PROGRAMACIÓN

UNIDAD 9

TIPOS DE DATOS GENÉRICOS

DIEGO VALERO ARÉVALO
BASADO EN APUNTES Y EJERCICIOS PROPORCIONADOS POR
Mª CARMEN DÍAZ GONZÁLEZ - IES VIRGEN DE LA PALOMA

U9 - **TIPOS DE DATOS GENÉRICOS**
ÍNDICE

9.1 - INTRODUCCIÓN A LOS TIPOS GENÉRICOS

El uso de los **tipos genéricos (`T`)** obedece a la **necesidad de disponer de clases, interfaces o métodos** que **se puedan usar con muchos tipos de datos distintos**, pero haciendo **comprobaciones del tipo durante la compilación**.

>> 9.1.1- COMPARACIÓN GENÉRICA: `COMPARETO()` Y `COMPARE()`

Ejemplos importantes de estructuras que pueden adaptarse a cualquier tipo de dato son los **métodos de comparación** `compareTo()` y `compare()`, que aparecen en las interfaces `Comparable` y `Comparator` respectivamente, que tendremos que implementar al usar esos métodos. Ambas interfaces están pensadas para **comparar objetos de cualquier clase**. De hecho, **tendremos que implementar** `Comparable` para **cualquier clase de objetos** que insertemos en cualquier **tabla** o **colección** que pretendamos ordenar. Por ejemplo, si tenemos una tabla de objetos de tipo `Cliente` que queremos ordenar por su atributo `DNI`, implementamos la interfaz `Comparable` y su único método `compareTo()`.

Antes de que aparecieran los genéricos con **Java 5**, estos métodos recibían como parámetros variables de tipo `Object`, que es la clase **más general de todas**. Aunque hoy día siguen soportando esta implementación antigua, **debe evitarse cuando sea posible**, ya que eso sería **renunciar a la ventaja de los tipos genéricos**.

9.2 - USO DE TIPOS GENÉRICOS

>> 9.2.1- CLASES CON PARÁMETROS GENÉRICOS

Supongamos que queremos definir la clase `Contenedor`, con las siguientes características:

CONTENEDOR.JAVA
• Permitirá **guardar un solo objeto** de **cualquier tipo**. • Los únicos métodos serán `guardar()` y `extraer()`. • Habrá **un único atributo llamado** `objeto` que, en principio, podría ser de tipo `Object`, para que el objeto para guardar **pueda ser de cualquier clase**.

Pero tenemos un problema: Así **no tenemos control sobre el tipo del objeto guardado**. Desde luego, siempre **podríamos implementar una clase** `Contenedor` **para** `Integer`**, otra para** `Double` **y así sucesivamente**. Pero si lo que queremos es una clase `Contenedor` que **sirva para todo tipo de objetos** y que, a la vez, **permita controlar en cada caso ese tipo**, tenemos que recurrir a los **tipos genéricos**.

· 9.2.1.1- DECLARAR CLASES GENÉRICAS

Para declarar una clase como **genérica**, se añade el **operador diamante (<>)** y dentro el **tipo genérico `T`** en el nombre de la clase:

```
public class nombreClase<T>{
    ...
}
```

Se suele usar la letra `T` para el **tipo genérico**, pero **puede ser cualquier otra**, aunque es costumbre que algunas se reserven para otros usos, como `B` para elementos de colecciones, `K` para claves, `V` para valores o `N` para números.

Una clase `Contenedor` con tipo genérico `T` podría ser:

EJEMPLO

`Contenedor.java`

```
1  public class Contenedor<T>{
2
3      private T[] objeto;
4
5      public void guardar(T nuevo){
6          objeto = nuevo;
7      }
8
9      public T extraer(){
10         T res = objeto;
11         objeto = null;
12         return res;
13     }
14 }
```

Ya que **no tenemos un constructor**, `objeto` se inicializa él solo con un valor `null`. Con `guardar()` podemos añadir **cualquier tipo de dato a este**, y con `extraer()`, hacemos que `objeto` vuelva a estar vacío.

`T` representa el **tipo de datos que se va a usar en la clase** en **cada declaración concreta**, y este debe ser una **clase** o **interfaz**, ya que los genéricos **no aceptan primitivas**.

Recuerda que para usar primitivas en estos casos deberemos recurrir a los **wrappers** (`Integer`, `Boolean`, `Character`,...).

Para poder **declarar un objeto con nuestra clase genérica**, usamos la siguiente sintaxis:

```
Clase<Tipo> nombreObjeto = new Clase<[Tipo]>();
```

`Clase`	El tipo de nuestro objeto será de la **clase que hemos creado**.
`<Tipo>`	Como la clase es **genérica**, debemos **especificar el tipo de dato** que le vamos a pasar.
`nombreObjeto`	Asignamos un **nombre** al mismo.
`new`	El operador `new` asigna durante la ejecución del programa un **espacio de memoria al objeto**, y almacena esta **referencia** en una variable.
`Clase<[Tipo]>()`	La clase en la que se va a basar el objeto, que será la **que hemos creado**. Podemos **declarar el tipo de dato aquí también** u **omitirlo**, ya que JAVA **automáticamente refiere** a la primera declaración para asignarlo.

EJEMPLO

ContenedorMain.java

```java
public class ContenedorMain{
   public static void main(String[] args){

      Contenedor<Integer> miContenedor = new Contenedor<Integer>();

      miContenedor.guardar(5);

      Integer n = miContenedor.extraer();

      System.out.println(n);
   }
}
```

Console

```
5
```

El compilador comprueba el **tipo del valor** que pasamos al método `guardar()`, que tiene que ser `Integer`. Si hubiéramos pasado el valor `7.4` o la cadena `"silla"`, habría dado **un error en la compilación**.

También **hace una comprobación de tipos** en la **asignación a la variable** `n`, que se ha declarado `Integer`. Si hubiéramos implementado la clase `Contenedor` con una variable de tipo `Object` en vez de un tipo genérico `T`, habríamos tenido que hacer un casting implícito a tipo `Integer` delante de `miContenedor.extraer()`, y si el objeto devuelto **fuera de tipo distinto a** `Integer`, el **error** se habría producido **durante la ejecución del programa**.

Con los tipos genéricos podemos **declarar objetos de cualquier tipo**, incluso clases propias:

EJEMPLO

ContenedorMain.java

```java
Contenedor<Cliente> cliente = new Contenedor<>();
```

· 9.2.1.2- DECLARAR CLASES CON MÚLTIPLES PARÁMETROS GENÉRICOS

En la implementación de una clase **pueden intervenir más de un tipo genérico**. Para ello, se especifican los parámetros separados por comas. A cada uno se le suele dar una letra diferente como `U`, `V`,... .

```java
public class nombreClase<T1, ..., Tn>{
   ...
}
```

· 9.2.1.3- USO DE CLASES GENÉRICAS SIN ESPECIFICACIÓN DE TIPOS

Las clases definidas con tipos genéricos, como nuestro `Contenedor`, también pueden usarse **sin declarar el tipo de dato que van a guardar**, en cuyo caso el compilador asigna por defecto a las variables el tipo `Object`. Eso significa que **no hace comprobaciones de tipos** y se pueden guardar objetos de **distintas clases mezclados**. Por ejemplo:

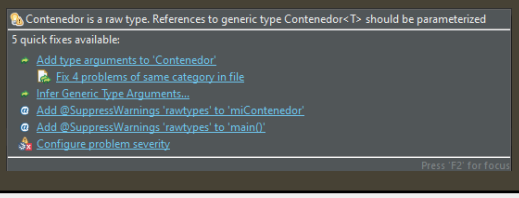
EJEMPLO

ContenedorMain.java

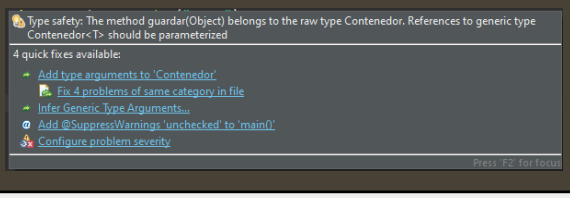
```java
public class ContenedorMain{
```

```
2      public static void main(String[] args){
3
4          Contenedor miContenedor = new Contenedor();
5
6          miContenedor.guardar(5);
7          miContenedor.guardar("mesa");
8
9          Double x = (Double) miContenedor.extraer();
10
11         System.out.println(n);
12     }
13 }
```

Contenedor



miContenedor.guardar(5)



El compilador se limita a **avisarnos** de que estamos realizando **operaciones sin comprobación de tipo**.

Console

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be cast to class java.lang.Double (java.lang.String and java.lang.Double are in module java.base of loader 'bootstrap')
        at ContenedorMain.main(Main.java:9)
```

El error se producirá al ejecutar el programa, con una excepción **`ClassCastException`** al aplicar el **casting**.

>> 9.2.2- INTERFACES CON PARÁMETROS GENÉRICOS

Las **interfaces genéricas** funcionan exactamente igual que las **clases genéricas**.

```
interface nombreInterfaz<T>{
    ...
}
```

Tenemos como ejemplo la interfaz **`Comparable`**, clase responsable del método **`compareTo()`** que ya hemos visto. Para usarla especificando el tipo de dato con el que va a trabajar podemos escribir la clase como parámetro. Por ejemplo:

EJEMPLO

Cliente.java

```
1  public class Cliente implements Comparable<Cliente>{
2
3      private String dni;
4
5      public int compareTo(Cliente obj){
6          return dni.compareTo(obj.dni);
7      }
8  }
```

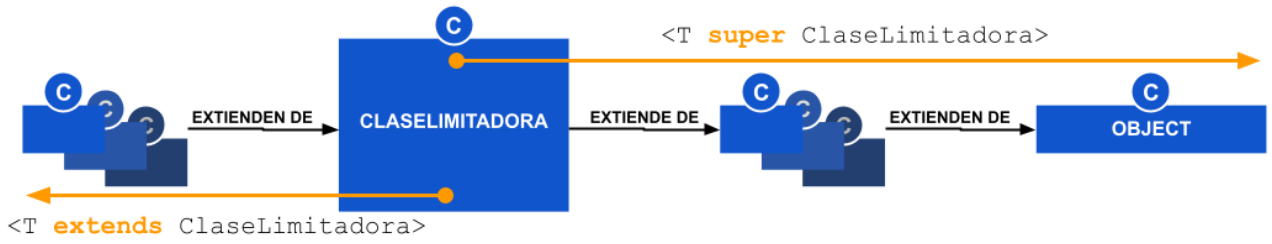
>> 9.2.3- LIMITAR LOS TIPOS GENÉRICOS

· 9.2.3.1- LIMITACIÓN A CLASES

Hay ocasiones en las que aunque **un método use un tipo genérico `T`**, sus operaciones **sólo pueden usar cierto tipo de datos**. Por ejemplo, un método en el que se operen valores numéricos no podrá usar datos de tipo **`String`** u otra clase. Para ello lo que hacemos es **limitar los tipos `T` a una determinada clase o a sus superclases o subclases**, lo cual se especifica **extendiendo** el tipo **`T`**.

```
public class nombreClase<T extends / super nombreClaseLimitadora>{
    ...
}
```

`<T extends nombreClaseLimitadora>`	De esta manera limitamos a que los datos **`T`** que pasemos sólo puedan ser del **tipo de la clase que extendemos y sus respectivas subclases**.
`<T super nombreClaseLimitadora>`	Al contrario que **`extends`**, **`T`** sólo podrá ser **del tipo de la clase que extendemos y las superclases de las que esta hereda**.



EJEMPLO

Sumar.java

```
1  public class Calculadora<T extends Number>{
2
3      ...
4  }
```

Main.java

```
1  public class Main{
2     public static void main(String[] args){
3
4        Calculadora<Double> obj1 = new Calculadora<>();
5
7        Calculadora<String> obj2 = new Calculadora<>();
8     }
9  }
```

```
Bound mismatch: The type String is not a valid substitute for the bounded parameter <T extends Number> of the type Calculadora<T>
Press 'F2' for focus
```

Al declarar un objeto donde su tipo **no corresponda con la clase limitadora**, el compilador **nos lanzará un error**. En este caso puedes leer en el error que **no acepta** el tipo **`String`** para sustituir el parámetro `T`, ya que está delimitado por **`Number`** en nuestro tipo **`Calculadora`**.

· 9.2.3.1- LIMITACIÓN A INTERFACES

También se pueden **limitar** los tipos genéricos a aquellos que implementan **una o más interfaces**. En este caso no se usa la palabra **`implements`**, sino **`extends`** (como si fuera una herencia). Esta es una particularidad exclusiva de la sintaxis de los parámetros genéricos.

```
public class NombreClase<T extends NombreInterfaz>{
    ...
}
```

Haciendo esto sólo podremos definir objetos de esta clase sólo con tipos que extiendan de la interfaz especificada.

>> 9.2.4- MÉTODOS GENÉRICOS

Los **parámetros genéricos** de una **clase** o **interfaz** suelen aparecer en los **métodos implementados** dentro de ella. Sin embargo, dentro de cualquier clase, tanto si está definida con tipos genéricos como si no, **podemos implementar métodos con sus propios parámetros genéricos**, distintos de los que pueda tener la clase, llamados **métodos genéricos**. El **tipo genérico** se declara en la **definición del método**, **justo antes del tipo devuelto**.

```
public [static] <T> tipo nombreDelMetodo(T nombreParametro){
    return valorReturn;
}
```

EJEMPLO

Ejemplo.java

```
1  public class Ejemplo{
2     public static void main(String[] args){
3        String[] miTabla = {"hola", "adios", "manzana", null, "casa"};
4
5        System.out.println(numeroNulls(miTabla));
7     }
8
9     public static <U> int numeroNulls(U[] tabla){
10       int contador = 0;
11       for(U e : tabla){
12          if(e == null){
13             contador++;
14          }
15       }
16       return contador;
17    }
18 }
```

Console

```
1
```

Este método se puede incluir **en cualquier clase** y **no depende** de los parámetros propios de la misma.

El tipo asociado a un método genérico **también puede estar limitado**. Por ejemplo, si quisiéramos que nuestro método **`numeroNulls()`** sólo funcionase con tablas numéricas, lo declararíamos como **`<U extends Number>`** en vez de **`<U>`**.

>> 9.2.5- COMODINES (WILDCARDS)

Como no podemos pasar un tipo `T` como argumento de un tipo genérico, usamos los **comodines**, que son representados con el **símbolo de interrogación (`?`)**. Estos significan **cualquier tipo**, y se suelen usar en la **declaración de atributos**, **variables locales** o **parámetros** que se vayan a pasar a un **método** cuando no sabemos de qué tipo van a ser.

```
Clase<?> nombreObjeto = new Clase<>();
```

EJEMPLO

Main.java

```
1  public class Main{
2     public static void main(String[] args){
3        Contenedor<?> miContenedor = new Contenedor<>();
4     }
5  }
```

Al declarar el objeto `miContenedor` de la clase `Contenedor` con un tipo comodín, este puede ser de cualquier tipo perteneciente a `Contenedor` , lo que significa que puede ser `Contenedor<Integer>`, `Contenedor<String>`, `Contenedor<Boolean>`, … o la que sea. Esto significa que todos los objetos `Contenedor` pertenecen a alguna subclase de `Contenedor<?>`.

Los comodines para tipos genéricos son la solución específica para un problema muy concreto: **leer y escribir en colecciones genéricas**, justo lo que acabamos de ver. Existen **3 comodines** para tipos genéricos que podemos aplicar:

Para tipos desconocidos	Para tipos derivados	Para supertipos
`<?>`	`<? extends T>`	`<? super T>`

· 9.2.5.1- PARA TIPOS DESCONOCIDOS

Son el tipo **más simple de comodines genéricos**, y también **el más limitado**, paradójicamente porque **no tiene límites**. Nos permiten indicar al compilador que **no sabemos el tipo exacto que se va a pasar** para **procesar** en nuestro tipo genérico.

EJEMPLO

Main.java

```
1  public static void mostrarElementos(List<?> listado){
2     for(Object elemento : listado){
3        System.out.println(elemento);
4     }
5  }
```

En este método, el tipo `List<?>` indica a este que el contenido de la colección que se pasa como parámetro puede ser **de cualquier tipo**.

En el fondo es casi como si **estuviésemos usando la solución tradicional con `Object`**. De hecho, los elementos de una colección de este tipo **los tendríamos que tratar como objetos `Object`**, así que le podríamos pasar cualquier cosa. No es la mejor solución, pero tiene sus usos, así que conviene conocerla.

· 9.2.5.2- PARA TIPOS DERIVADOS

¿Recuerdas la limitación a parámetros genéricos? Pues esto funciona exactamente igual. El comodín **`extends`** quiere decir que **se admiten todos los objetos que hereden de la clase especificada**.

EJEMPLO

Ejemplo.java

```java
public static void mostrarElementos(List<? extends Contenedor> listado){
    for(Contenedor elemento : listado){
        System.out.println(elemento);
    }
}
```

Ahora nuestro método sólo admite objetos de tipo **`List`** donde el contenido de la colección **debe ser de tipo `Contenedor`** o de aquellas **que hereden de esta**.

Fíjate en que, dado que cualquier clase que herede de **`Contenedor`** se puede convertir (*casting*) a **`Contenedor`**, entonces podemos recibir listas genéricas de ellas y convertir cada elemento a **`Contenedor`** antes de utilizarlo. Es por esto que como **variable del bucle** se puede usar un **elemento de tipo `Contenedor`**. Y es por esto que podemos pasarle sin problemas nuestra lista, que se imprimirá al igual que cualquier otra lista genérica. Es decir, **el comodín para tipos derivados actúa como límite superior en la jerarquía de clases** que admite la lista genérica, ya que cualquier clase que descienda de **`Contenedor`**, a cualquier nivel, se admitirá sin problemas.

· 9.2.5.3- PARA SUPERTIPOS

Exactamente al contrario que para derivados, con **`super`** hacemos que las clases admitidas sean **la que especificamos** y **aquellas de las que herede esta**.

EJEMPLO

Ejemplo.java

```java
public static void mostrarElementos(List<? super Contenedor> listado){
    for(Contenedor elemento : listado){
        System.out.println(elemento);
    }
}
```

Ahora nuestro método sólo admite objetos de tipo **`List`** donde **los objetos que contiene** tienen que ser de la clase **`Contenedor`** o de aquellos de los **que hereda la misma**.

>> 9.2.6- RESTRICCIONES DEL USO DE PARÁMETROS GENÉRICOS

A pesar de todas las aportaciones de las clases genéricas a la programación en JAVA, por la forma en que estas han sido implementadas, tienen una serie de **limitaciones**, algunas de las cuales puede que sean subsanadas en el futuro. En concreto, hay **varias operaciones** que, de momento, están **prohibidas**:

- Los tipos genéricos **nunca pueden ser primitivos**. Para usarlos debemos colocar **wrappers** (**`Integer`**, **`Character`**, ...).
- No se pueden crear **instancias de tipo genérico**.

  ✖
  ```java
  T miGenerico = new T();
  ```
- Acompañando a la restricción anterior, no se pueden crear **arrays de tipos genéricos**.

```
✖  T[] miArrayGenerico = new T[10];
```

Cuando necesitemos un array de **un tipo concreto**, debemos pasar este **como argumento al método donde van a ser usadas**, que puede ser un **constructor**, o deberán ser devueltos por algún **método definido fuera de la clase**. Los arrays genéricos **siempre deben construirse fuera de la clase**, **interfaz** o **método genérico**.

Tampoco se pueden crear **arrays de clases parametrizadas**

```
✖  T Contenedor<Integer>[] miArrayGenerico = new Contenedor<Integer>[10];
```

No se pueden usar **excepciones genéricas**. Simplemente con declararlas, el compilador lanza un **error**.

U9 - **BATERÍA DE EJERCICIOS**

1	Implementar, con tipos genéricos, la clase Contenedor, donde podremos guardar tantos objetos como deseemos. Para ello utilizaremos una tabla, que inicialmente tendrá tamaño cero y se irá redimensionando según añadamos o eliminemos elementos. La clase, además del constructor y toString(), tendrá los siguientes métodos: a. void insertarAlPrincipio(T nuevo) b. void insertarAlFinal(T nuevo) c. T extraerDelPrincipio() d. T extraerDelFinal() e. void ordenar()
2	A partir de la clase Contenedor definida en el ejercicio anterior, implementa una aplicación donde se guardan 30 enteros aleatorios entre 1 y 10 y luego se ordenan de mayor a menor. La aplicación debe mostrar el contenedor antes y después de ordenar.
3	Añade a la clase contenedor el método: void ordenar(Comparator<T> c) Que ordena los elementos del contenedor según el criterio de c.
4	Añade a la clase contenedor el método: T get (int índice) Que devuelve el elemento que ocupa el lugar índice dentro del contenedor.
5	Implementa la clase Cola genérica utilizando un objeto ArrayList para guardar los elementos. Una cola es del orden FIFO (First In First Out: el primero que llega, el primero que sale).
6	Implementa la clase Pila genérica utilizando un objeto ArrayList para guardar los elementos. Una pila es del orden FILO (First In Last Out: el primero que llega, el último que sale).