10 Genéricos

1. Introducción

Desde su versión original 1.0, muchas nuevas características han sido añadidas a Java. Todas han mejorado y ampliado el alcance del lenguaje, pero una que ha tenido un impacto especialmente profundo y de gran alcance es el uso de genéricos porque sus efectos se sintieron en todo el lenguaje Java ya que añadieron un elemento de sintaxis completamente nuevo y causaron cambios en muchas de las clases y métodos de la API principal. Los genéricos de Java están basados en los famosos *templates* de C++ y se introdujeron en la versión 5 de Java.

En su esencia, el término **genérico** significa **tipo parametrizado**, es decir, el tipo de datos sobre el que se opera se especifica como parámetro. Muchos algoritmos son lógicamente los mismos, independientemente del tipo de datos a los que se apliquen. Por ejemplo, un algoritmo de ordenación es el mismo si está ordenando elementos de tipo entero, decimales o cadena. Con los genéricos, se puede definir un algoritmo una vez, independientemente del tipo de datos y luego aplicar ese algoritmo a una amplia variedad de tipos de datos sin ningún esfuerzo adicional.

Lo que se use en un genérico debe ser un objeto, por lo tanto, los genéricos no funcionan con datos primitivos. Para resolver esta situación, la API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos.

Los genéricos permiten crear clases, métodos e interfaces en los que los tipos de datos sobre los que actúan son un parámetro más, de manera que pueden adaptarse de forma automática para emplear dichos tipos.

2. Clases genéricas

Cuando una clase tiene algún parámetro de tipo se denomina **clase genérica**. Al crear objetos de esa clase, se debe indicar el tipo real por el que se va a sustituir el genérico. A partir de ese momento, donde se utilice el genérico, se sustituirá por el tipo real con el que se ha creado el objeto.

Veamos un ejemplo de clase genérica:

```
package tema10_Genericos;

public class GenericClass<T> {//Clase genérica de tipo T

    private T attribute;//Declaramos un atributo de tipo T

    public GenericClass(T attribute) {
        this.attribute = attribute;

    }

    public T getAttribute() {
        return attribute;
    }

    public void show() {
        System.out.printf("El tipo de T es %s",
attribute.getClass().getSimpleName());
        System.out.printf("  --->  Valor del atributo: %s\n",
getAttribute());
    }

    public static void main(String[] args) {

        //T es un parámetro de tipo que se sustituye por un tipo real
al crear un objeto de la clase
        GenericClass<Integer> genInteger;//Se sustituye T por Integer
        GenericClass<Double> genDouble;//Se sustituye T por Double
        GenericClass<String> genString;//Se sustituye T por String
        GenericClass<Vehicle> genVehicle;//Se sustituye T por Vehicle

        genInteger = new GenericClass<Integer>(80);
        genDouble = new GenericClass<Double>(78.6);
        genString = new GenericClass<String>("Clases genéricas");
        genVehicle = new GenericClass<Vehicle>(new Vehicle(4,
"azul"));

        genInteger.show();
        genDouble.show();
        genString.show();
        genVehicle.show();
```

```
    }

}
```

Salida por consola:

```
El tipo de T es Integer  --->  Valor del atributo: 80
El tipo de T es Double  --->  Valor del atributo: 78.6
El tipo de T es String  --->  Valor del atributo: Clases genéricas
El tipo de T es Vehicle  --->  Valor del atributo: Vehicle
[wheelCount=4, speed=0.0, colour=azul]
```

Podemos observar que la clase tiene un atributo de tipo `T`, pero no se permiten variables estáticas de tipo `T`.

Una de las ventajas del uso de genéricos es que el compilador realiza comprobaciones de tipos y no permite asignar a un objeto la referencia de otro con distinta traducción del tipo de genérico:

```
genInteger = genDouble;//Error de compilación
```

El compilador informa con el siguiente mensaje de error: No puede convertir de `GenericClass<Double>` a `GenericClass<Integer>`. Si ambos hubieran sido del mismo tipo, sí se podría haber realizado la asignación.

A partir de Java 7, cuando instanciamos un objeto de una clase genérica, no es necesario especificar el tipo con el que queremos trabajar siempre y cuando el compilador pueda inferirlo (adivinarlo) a partir del contexto. Por lo tanto, el siguiente código:

```
genInteger = new GenericClass<Integer>(80);
genDouble = new GenericClass<Double>(78.6);
genString = new GenericClass<String>("Clases genéricas");
genVehicle = new GenericClass<Vehicle>(new Vehicle(4, "azul"));
```

también se puede realizar de la siguiente manera:

```
genInteger = new GenericClass<>(80);//Como la variable a la que se
asigna es GenericClass<Integer> es capaz de inferir que queremos crear
un GenericClass<Integer> al hacer el new
genDouble = new GenericClass<>(78.6);
genString = new GenericClass<>("Clases genéricas");
genVehicle = new GenericClass<>(new Vehicle(4, "azul"));
```

A esto se le conoce como **operador diamante**, dado que al quitar el tipo de la instanciación nos queda `<>` (que presenta una forma de diamante).

3. Métodos genéricos

Los métodos de una clase genérica pueden usar el parámetro del tipo de una clase y por tanto son genéricos de forma automática. Sin embargo, también podemos declarar métodos genéricos dentro de clases no genéricas, tanto estáticos como dinámicos, que reciban uno o más parámetros genéricos.

Los genéricos en un método se declaran en la firma del método delante del tipo de devolución.

Veamos un ejemplo de un método genérico estático:

```
package tema10_Genericos;

public class GenericMethod {

    public static <T> String showType(T parameter) {//El genérico <T>
se declara delante del tipo de devolución String
        return parameter.getClass().getSimpleName();
    }

    public static void main(String[] args) {

        System.out.printf("El tipo del parámetro es %s\n",
GenericMethod.showType(80));
        System.out.printf("El tipo del parámetro es %s\n",
GenericMethod.showType(78.6));
        System.out.printf("El tipo del parámetro es %s\n",
GenericMethod.showType("Métodos genéricos"));
        System.out.printf("El tipo del parámetro es %s\n",
GenericMethod.showType(new Vehicle(4, "azul")));

    }

}
```

Salida por consola:

```
El tipo del parámetro es Integer
El tipo del parámetro es Double
El tipo del parámetro es String
El tipo del parámetro es Vehicle
```

4. Interfaces genéricas

Las interfaces también pueden contener parámetros de tipo, denominándose interfaces genéricas. Las clases que implementan la interfaz, tienen dos posibilidades:

- Implementar la interface traduciendo el genérico por un tipo concreto.
- Implementar la interface de manera genérica convirtiéndose en una clase genérica.

Veamos ejemplos de ambos casos:

Ejemplo de implementación de la interface traduciendo el genérico por un tipo concreto:

```
package tema10_Genericos;

public interface GenericInterface<T> {

    T first(T[] array);

    T last(T[] array);

}
```

```
package tema10_Genericos;

public class MyClass implements GenericInterface<Vehicle> {

    @Override
    public Vehicle first(Vehicle[] array) {
        return array[0];
    }

    @Override
    public Vehicle last(Vehicle[] array) {
        return array[array.length - 1];
    }

    public static void main(String[] args) {

        MyClass myClass = new MyClass();
        Vehicle[] array = new Vehicle[3];
        array[0] = new Vehicle(4, "azul");
        array[1] = new Vehicle(4, "blanco");
        array[2] = new Vehicle(2, "rojo");
        System.out.printf("Primer vehículo del array: %s\n",
myClass.first(array));
        System.out.printf("Último vehículo del array: %s",
myClass.last(array));
```

```
    }

}
```

Salida por consola:

```
Primer vehículo del array: Vehicle [wheelCount=4, speed=0.0,
colour=azul]
Último vehículo del array: Vehicle [wheelCount=2, speed=0.0,
colour=rojo]
```

Ejemplo de implementación de la interface de manera genérica convirtiéndose en una clase genérica:

```
package tema10_Genericos;

public class GenericClass2<T> implements GenericInterface<T> {

    @Override
    public T first(T[] array) {
        return array[0];
    }

    @Override
    public T last(T[] array) {
        return array[array.length - 1];
    }

    public static void main(String[] args) {

        Vehicle[] array = new Vehicle[3];
        array[0] = new Vehicle(4, "azul");
        array[1] = new Vehicle(4, "blanco");
        array[2] = new Vehicle(2, "rojo");
        GenericClass2<Vehicle> genVehicle = new GenericClass2<>();
        System.out.printf("Primer vehículo del array: %s\n",
genVehicle.first(array));
        System.out.printf("Último vehículo del array: %s",
genVehicle.last(array));

    }

}
```

Salida por consola:

```
Primer vehículo del array: Vehicle [wheelCount=4, speed=0.0,
colour=azul]
Último vehículo del array: Vehicle [wheelCount=2, speed=0.0,
colour=rojo]
```

5. Uso de varios genéricos

Se pueden utilizar varios genéricos indicándolos separados por comas dentro del operador diamante.

```
package tema10_Genericos;

public class GenericClass3<T, V> {

    private T attribute1;
    private V attribute2;

    public GenericClass3(T attribute1, V attribute2) {
        this.attribute1 = attribute1;
        this.attribute2 = attribute2;
    }

    public T getAttribute1() {
        return attribute1;
    }

    public V getAttribute2() {
        return attribute2;
    }

    public void show() {
        System.out.printf("El tipo de T es %s",
attribute1.getClass().getSimpleName());
        System.out.printf("  --->  Valor del atributo: %s\n",
getAttribute1());
        System.out.printf("El tipo de V es %s",
attribute2.getClass().getSimpleName());
        System.out.printf("  --->  Valor del atributo: %s\n\n",
getAttribute2());

    }

    public static void main(String[] args) {

        GenericClass3<Integer, Integer> variousGenerics1 = new
GenericClass3<>(80, 56);
        GenericClass3<Integer, Double> variousGenerics2 = new
GenericClass3<>(5, 78.6);
        GenericClass3<String, Vehicle> variousGenerics3 = new
GenericClass3<>("Clases genéricas",
```

```
                new Vehicle(4, "azul"));
        variousGenerics1.show();
        variousGenerics2.show();
        variousGenerics3.show();
    }

}
```

Salida por consola:

```
El tipo de T es Integer  --->  Valor del atributo: 80
El tipo de V es Integer  --->  Valor del atributo: 56

El tipo de T es Integer  --->  Valor del atributo: 5
El tipo de V es Double  --->  Valor del atributo: 78.6

El tipo de T es String  --->  Valor del atributo: Clases genéricas
El tipo de V es Vehicle  --->  Valor del atributo: Vehicle
[wheelCount=4, speed=0.0, colour=azul]
```

El uso de genéricos puede crear situaciones de ambigüedad, sobretodo en casos de sobrecarga de métodos:

```
class Gen<T, V> {
    // Estos dos métodos se sobrecargan pero dado que T y V pueden ser
del mismo tipo, se generarían dos métodos iguales por lo que el
compilador genera un error y este código no compila.
    void get(T ob) {}

    void get(V ob) {}
}
```

6. Tipos vinculados (o limitados)

Java ofrece los tipos vinculados que permiten, al especificar un parámetro de tipo, crear un vínculo superior que declare la superclase de la que deben derivarse todos los argumentos de tipo. Para ello usamos la cláusula *extends* al especificar los parámetros de tipo: `<T extends superclass>`. Esto especifica que `T` solo se puede reemplazar por *superclass* o por subclases de *superclass*, por tanto *superclass* define un **límite superior e inclusivo**.

La cláusula *extends* también puede referirse a interfaces: `<T extends interface>`. En este caso, `T` solo se puede reemplazar por una clase que implemente dicha interfaz.

Los tipos numéricos heredan de la clase abstracta *Number*, por lo tanto, podemos limitar los parámetros de tipo a únicamente tipos numéricos: `<T extends Number>`. Esto nos permitiría utilizar métodos de la clase *Number*:

```java
package tema10_Genericos;

public class GenNumeric<T extends Number> { //Limitamos T a tipos
numéricos

    private T attribute;

    public GenNumeric(T attribute) {
        this.attribute = attribute;

    }

    public T getAttribute() {
        return attribute;
    }

    double fraction() {
        return attribute.doubleValue() - attribute.intValue(); //Se
pueden emplear métodos de la clase Number
    }

    public static void main(String[] args) {

        GenNumeric<Integer> genInteger;
        GenNumeric<Double> genDouble;

        genInteger = new GenNumeric<Integer>(80);
        genDouble = new GenNumeric<Double>(78.63);

        System.out.printf("La parte decimal de %s es %.2f\n",
genInteger.getAttribute(), genInteger.fraction());
        System.out.printf("La parte decimal de %s es %.2f\n",
genDouble.getAttribute(), genDouble.fraction());

    }

}
```

Salida por consola:

```
La parte decimal de 80 es 0,00
La parte decimal de 78.63 es 0,63
```

Si intentamos utilizar una clase que no sea descendiente de *Number*, nos da un error de compilación:

```java
GenNumeric<String> genString; //Error de compilación
```

Los tipos vinculados resultan especialmente útiles para garantizar que un parámetro sea compatible con otro:

```
class Pair<T, V extends T> { //V debe tener el mismo tipo que T o ser una subclase de T
    ....
}
```

V debe tener el mismo tipo que T o ser una subclase de T. Veamos ejemplos a la hora de crear objetos de la clase *Pair*:

```
Pair<Integer, Integer> pair1 = new Pair<Integer, Integer>();//Correcto
Pair<Number, Integer> pair2 = new Pair<Number, Integer>();//Correcto, Integer es una subclase de Number
Pair<Integer, String> pair3 = new Pair<Integer, String>();
//Incorrecto, String no es una subclase de Integer
```

7. Tipos comodín

El signo de interrogación `?` se conoce como comodín (wildcard) y representa un tipo desconocido. Es necesario utilizarlo cuando mezclamos objetos de la misma clase pero con distinta traducción del genérico. Veamos un ejemplo sin tipo comodín para ver dónde nos podría hacer falta:

```
package tema10_Genericos;

import java.util.Arrays;

public class GenericClass4<T> {

    private T[] array;

    GenericClass4(T[] array) {
        this.array = array;
    }

    public void showArray() {
        System.out.println(Arrays.toString(array));
    }

    public boolean equalSize(GenericClass4<T> gen) {
        return array.length == gen.array.length;
    }

    public static void main(String[] args) {

        Integer[] array1 = { 8, 7, 9 };
        Integer[] array2 = { 3, 5, 12 };
```

```
        GenericClass4<Integer> gen1 = new GenericClass4<>(array1);
        GenericClass4<Integer> gen2 = new GenericClass4<>(array2);
        gen1.showArray();
        gen2.showArray();
        System.out.printf("%s tienen el mismo tamaño",
gen1.equalSize(gen2) ? "Sí" : "No");

    }

}
```

Salida por consola:

```
[8, 7, 9]
[3, 5, 12]
Sí tienen el mismo tamaño
```

El método `equalSize` recibe un objeto por parámetro y devuelve si ambos objetos tienen el mismo tamaño del array. En este caso funciona porque tienen la misma traducción del genérico (Integer). Pero ¿y si le pasamos por ejemplo uno de Double?:

```
package tema10_Genericos;

public class Main {

    public static void main(String[] args) {

        Integer[] array1 = { 8, 7, 9 };
        Double[] array2 = { 3.6, 5.4, 12.42 };
        GenericClass4<Integer> gen1 = new GenericClass4<>(array1);
        GenericClass4<Double> gen2 = new GenericClass4<>(array2);
        gen1.showArray();
        gen2.showArray();
        System.out.printf("%s tienen el mismo tamaño",
gen1.equalSize(gen2) ? "Sí" : "No");

    }

}
```

Nos da un error de compilación porque en el objeto `gen1` se ha traducido el genérico por Integer y le estamos pasando como argumento `gen2` que se ha traducido por Double. Entonces, en este caso, como lo que queremos en el método `equalSize` es simplemente comparar el tamaño de dos arrays y no nos importa que los arrays sean de distinto tipo, lo que podemos hacer es utilizar el tipo comodín `?` en el parámetro de `equalSize` para indicar que nos tienen que pasar un objeto de un tipo pero que no tiene por qué necesariamente coincidir con `T`:

```java
package tema10_Genericos;

import java.util.Arrays;

public class GenericClass5<T> {

    private T[] array;

    GenericClass5(T[] array) {
        this.array = array;
    }

    public void showArray() {
        System.out.println(Arrays.toString(array));
    }

    public boolean equalSize(GenericClass5<?> gen) {//Tipo comodín
        return array.length == gen.array.length;
    }

    public static void main(String[] args) {

        Integer[] array1 = { 8, 7, 9 };
        Double[] array2 = { 3.6, 5.4, 12.42 };
        GenericClass5<Integer> gen1 = new GenericClass5<>(array1);
        GenericClass5<Double> gen2 = new GenericClass5<>(array2);
        gen1.showArray();
        gen2.showArray();
        System.out.printf("%s tienen el mismo tamaño",
gen1.equalSize(gen2) ? "Sí" : "No");

    }

}
```

Salida por consola:

```
[8, 7, 9]
[3.6, 5.4, 12.42]
Sí tienen el mismo tamaño
```

Los tipos comodín se pueden limitar de varias maneras:

- `<? extends superclass>`: solo se puede reemplazar por *superclass* o por subclases de *superclass*, por tanto *superclass* define un límite superior e inclusivo.
- `<? extends interface>`: solo se puede reemplazar por una clase que implemente dicha interfaz.
- `<? super subclass>:` solo se puede reemplazar por superclases de *subclass* sin incluir a *subclass*, por tanto *subclass* define un límite inferior no inclusivo.