Clases y métodos genéricos

Clases y métodos genéricos

1 Métodos genéricos

El uso de los genéricos se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas

Las clases y los métodos genéricos se usan en muchos lenguajes de programación para facilitar la reutilización del software, creando métodos y Clases que puedan trabajar con diferentes tipos de objetos, evitando así las conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina “genéricos”.

Veamos un ejemplo sencillo de cómo transformar un método normal en genérico:

Versión no genérica del método:

```
public class Util {

   public static int compararTamanyo(Object[] a, Object[] b) {
       return a.length-b.length;
   }
}
```

Versión genérica del método:

```
public class Util {

   public static <T> int compararTamanyo(T[] a, T[] b) {
       return a.length-b.length;
   }
}
```

Los dos métodos anteriores permiten comprobar si un array es mayor que otro.

Devuelven 0 si ambos arrays son iguales, un número mayor que cero si el array a es mayor, y un número menor de cero si el array b es mayor, pero uno es genérico y el otro no. La **versión genérica** del módulo incluye la expresión "**<T>**", **justo antes del tipo devuelto** por el método. "**<T>**" es la definición de una variable o parámetro formal de tipo de la clase o método genérico, al que podemos llamar simplemente **parámetro de tipo** o **parámetro genérico** . Este parámetro genérico (T) se puede usar a lo largo de todo el método o clase, dependiendo del ámbito de definición, y hará referencia a cualquier clase con la que nuestro algoritmo tenga que trabajar. Puede haber más de un parámetro genérico.

Utilizar genéricos tiene claras ventajas. Para invocar un método genérico, sólo hay que realizar una invocación de tipo genérico, olvidándonos de las conversiones de tipo. Esto consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico ("**<T>**"), para después, pasándole los argumentos correspondientes, ejecutar el algoritmo. Cada clase o interfaz concreta, la podemos denominar tipo o **tipo base** y se da por sentado que los argumentos pasados al método genérico serán también de dicho tipo base.

Si el tipo base es Integer, para realizar la invocación del método genérico anterior basta con indicar el tipo, entre los símbolos de menor qué y mayor qué (" <Integer> "), justo antes del nombre del método.

Invocación del método NO genérico

```
Integer []a={0,1,2,3,4};
Integer []b={0,1,2,3,4,5};
Util.compararTamanyo ((Object[])a, (Object[])b);
```

Invocación del método genérico

```
Integer []a={0,1,2,3,4};
Integer []b={0,1,2,3,4,5};
Util.<Integer>compararTamanyo (a, b);
```

2 Clases genéricas

Son equivalentes a los métodos genéricos pero en el ámbito de las clases. Permiten definir un parámetro de tipo genérico que se podrá usar a lo largo de toda la clase. Así se crean clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase.

```
public class Util<T>{
	T t1;

	public void invertir(T[] array) {
		for(int i =0 ; i< array.length/2; i++) {
			t1=array[i];
			array[i] = array[array.length - i - 1];
			array[array.length - i -1] = t1;
		}
	}
}
```

En el ejemplo anterior el método invertir lo que hace es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de esa clase especificando el tipo base entre los símbolos "**<**" y "**>**", justo detrás del nombre de la clase. Ejemplo:

```
Integer[] numeros={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
Util<Integer> u = new Util<Integer>();
u.invertir(numeros);
for(int i =0; i<numeros.length; i++)
	System.out.print(numeros[i] + " ");
```

Hay que especificar el tipo tanto al declarar la referencia al objeto como al instanciarlo.

Los parámetros de tipo de las clases genéricas sólo pueden ser clases, no pueden ser tipos de datos primitivos como `int`, `double`… En su lugar tendremos que usar sus clases envoltorio: `Integer`, `Double`, …

3 Otras cuestiones a tener en cuenta al trabajar con clases y métodos genéricos.

3.1 Dos o más parámetros de tipo

Si un **método** genérico necesita utilizar dos o más parámetros genéricos se indican separándolos por comas.

Ejemplo de método que devuelve la suma de las longitudes de dos arrays que pueden ser de tipos distintos:

```
public static <T, M> int sumaDeLongitudes(T[] a, M[] b){
	return a.length + b.length;
}
```

Ejemplo de **clase** que almacena tres elementos de distinto tipo base que están relacionados entre sí:

```
public class Terna<A, B, C> {
	A varA; B varB; C varC;

	public Terna(A varA, B varB, C varC){
		this.varA=varA;
		this.varB=varB;
		this.varC=varC;
	}

	public A getVarA() {
		return varA;
	}
	public B getVarB() {
		return varB;
	}
	public C getVarC() {
		return varC;
	}

}
```

3.2 Inferencia de tipos.

La **inferencia de tipos** asigna automáticamente un **tipo** de datos a una función sin necesidad de que el programador lo escriba.

A partir de Java 7 no es obligatorio indicar los tipos al invocar a un método genérico.

```
Integer[] vecInteger={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
Double[] vecDouble={0d, 1d, 2d, 3d, 4d, 5d, 6d, 7d, 8d, 9d};
Util.<Integer, Double>sumaDeLongitudes(vecInteger, vecDouble);
Util.sumaDeLongitudes(vecInteger, vecDouble);
```

Las dos formas de llamar al método ***sumaDeLongitudes*** son válidas.

A partir de Java 7 se puede usar el operador diamante (<>) para **instanciar objetos de clases genéricas**.

Ejemplo:

```
Integer integer1=0; Double double1=1.2d; Float float1=1.43f;
Terna<Integer, Double, Float> unaTerna = new Terna<>(integer1, double1, float1);
```

3.3 Limitación de tipos

Se pueden limitar el conjunto de tipos que se pueden usar con una clase o método genérico usando el operador "extends". Este operador permite indicar que la clase que se pasa como parámetro genérico tiene que derivar de una clase específica. En el ejemplo que sigue no se admitirá ninguna clase que no derive de Number, así nos aseguramos de que se puede realizar la suma:

```
public class Util{
	public static <T extends Number> Double Sumar(T t1, T t2){
		return new Double(t1.doubleValue() + t2.doubleValue());
	}
}
```

3.4 Paso de clases genéricas como parámetro

Cuando un método tiene como parámetro una clase genérica se puede especificar cuál debe ser el tipo base usado en la instancia de la clase genérica que se le pasa como argumento. De esta forma se pueden crear diferentes versiones de un mismo método (sobrecarga) y dependiendo del tipo base usado en la instancia de la clase genérica se ejecutará una versión u otra:

```
public class Ejemplo<A>{
	public A a;
}
...
void test(Ejemplo<Integer> e) { ... }
```

3.5 Paso de clases genéricas como parámetro. Carácter comodín.

Cuando un método admite como parámetro una clase genérica, en la que no importa el tipo de clase, podemos usar el interrogante para indicar "cualquier tipo".

```
public class Ejemplo<A>{
	public A a;
}
...
void test(Ejemplo<?> e) { ... }
```

También se puede limitar el conjunto de tipos que una clase genérica puede usar a través del operador extends. En el ejemplo que sigue es como decir **"cualquier tipo que derive de Number"**.

```
public class Ejemplo<A>{
	public A a;
}
...
void test(Ejemplo<? extends Number> e) { ... }
```