9.A. Introducción a las estructuras de almacenamiento.

Sitio: Aula Virtual CIERD (CIDEAD)
Curso: Programación_DAM
Libro: 9.A. Introducción a las estructuras de almacenamiento.
Imprimido por: LUIS PUJOL
Día: jueves, 12 de marzo de 2020, 13:45

Tabla de contenidos

1 Introducción.

Anteriormente, en este curso, ya clasificamos las estructuras de almacenamiento en función de si pueden o no cambiar de tamaño de forma dinámica:

- **Estructuras cuyo tamaño se establece en el momento de la creación o definición** y su tamaño no puede variar después. Ejemplos de estas estructuras son los **arrays** y las matrices (arrays multidimensionales). Este grupo ya lo vimos en una unidad anterior.

- **Estructuras cuyo tamaño es variable (conocidas como estructuras dinámicas)**. Su tamaño crece o decrece según las necesidades de forma dinámica. Es el caso de las listas, árboles, conjuntos y algunos tipos de cadenas de caracteres. Este grupo es el que trataremos en esta unidad de contenidos.

Además, atendiendo a la forma en la que los datos se ordenan dentro de la estructura, ya diferenciamos varios tipos de estructuras:

- **Estructuras que no se ordenan de por sí**, y debe ser el programador el encargado de ordenar los datos si fuera necesario. Un ejemplo de estas estructuras son los arrays. Este grupo ya lo vimos en una unidad anterior.

- **Estructuras ordenadas**. Se trata de estructuras que al incorporar un dato nuevo a todos los datos existentes, este se almacena en una posición concreta que irá en función del orden. El orden establecido en la estructura puede variar dependiendo de las necesidades del programa: alfabético, orden numérico de mayor a menor, momento de inserción, etc. Este grupo es el que trataremos en esta unidad de contenidos.

2 Clases y métodos genéricos (I).

Una particularidad esencial de las estructuras dinámicas es la utilización de lo que se conoce como clases y métodos genéricos.

¿Sabes por qué se suele aprender el uso de los genéricos? Pues porque se necesita para usar las listas, aunque realmente los genéricos son una herramienta muy potente y que nos puede ahorrar tareas de programación repetitivas.

Imagen procedente de curso Progragamación del MECD

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes de programación. Su objetivo es claro: facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos, evitando incomodas y engorrosas conversiones de tipos. Su inicio se remonta a las plantillas (templates) de C++, un gran avance en el mundo de programación sin duda. En lenguajes de más alto nivel como Java o C# se ha transformado en lo que se denomina "genéricos". Veamos un ejemplo sencillo de como transformar un método normal en genérico:

Versiones genérica y no genérica del método de compararTamano.	
Versión no genérica	**Versión genérica del método**

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

Los dos métodos anteriores tienen un claro objetivo: permitir comprobar si un array es mayor que otro. Retornarán 0 si ambos arrays son iguales, un número mayor de cero si el array `b` es mayor, y un número menor de cero si el array `a` es mayor, pero uno es genérico y el otro no. La versión genérica del módulo incluye la expresión "`<T>`", justo antes del tipo retornado 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. Como veremos más adelante, 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**.

Supongamos que el tipo base es `Integer`, pues 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.

Invocaciones de las versiones genéricas y no genéricas de un método.	
Invocación del método no genérico.	**Invocación del método genérico.**
`Integer []a={0,1,2,3,4};`	`Integer []a={0,1,2,3,4};`
`Integer []b={0,1,2,3,4,5};`	`Integer []b={0,1,2,3,4,5};`
`util.compararTamano ((Object[])a, (Object[])b);`	`util.<Integer>compararTamano (a, b);`

3 Clases y métodos genéricos (II).

¿Crees qué el código es más legible al utilizar genéricos o que se complica? La verdad es que al principio cuesta, pero después, el código se entiende mejor que si se empieza a insertar conversiones de tipo.

Las clases genéricas son equivalentes a los métodos genéricos pero a nivel de clase, permiten definir un parámetro de tipo o genérico que se podrá usar a lo largo de toda la clase, facilitando así crear 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, la clase `Util` contiene el método invertir cuya función 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 dicha clase especificando el tipo base entre los símbolos menor que ("<") y mayor que (">"), justo detrás del nombre de la clase. Veamos un 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.println(numeros[i]);
```

Como puedes observar, el uso de genéricos es sencillo, tanto a nivel de clase como a nivel de método. Simplemente, a la hora de crear una instancia de una clase genérica, hay que especificar el tipo, tanto en la definición (`Util <integer> u`) como en la creación (`new Util<Integer>()`).

Los genéricos los vamos a usar ampliamente a partir de ahora, aplicados a un montón de clases genéricas que tiene Java y que son de gran utilidad, por lo que es conveniente que aprendas bien a usar una clase genérica.

Los parámetros de tipo de las clases genéricas solo pueden ser clases, no pueden ser jamás tipos de datos primitivos como `int, short, double`, etc. En su lugar, debemos usar sus clases envoltorio `Integer, Short, Double`, etc.

Autoevaluación

Dada la siguiente clase, donde el código del método `prueba` carece de importancia, ¿podrías decir cuál de las siguientes invocaciones es la correcta?

```
public class Util {
```

```
        public static <T> int prueba (T t) { … }
```

```
};
```

- `Util.<int>prueba(4);`
- `Util.<Integer>prueba(new Integer(4));`
- `Util u=new Util(); u.<int>prueba(4);`

/