9.G. Algoritmos sobre listas y arrays.

Tabla de contenidos

1 Algoritmos (I).

La palabra algoritmo seguro que te suena, pero, ¿a qué se refiere en el contexto de las colecciones y de otras estructuras de datos? Las colecciones, los arrays e incluso las cadenas, tienen un conjunto de operaciones típicas asociadas que son habituales. Algunas de estas operaciones ya las hemos visto antes, pero otras no. Veamos para qué nos pueden servir estas operaciones:

- Ordenar listas y arrays.
- Desordenar listas y arrays.
- Búsqueda binaria en listas y arrays.
- Conversión de arrays a listas y de listas a array.
- Partir cadenas y almacenar el resultado en un array.

Estos algoritmos están en su mayoría recogidos como métodos estáticos de las `clases java.util.Collections` y `java.util.Arrays`, salvo los referentes a cadenas obviamente.

Los algoritmos de ordenación ordenan los elementos en orden natural, siempre que Java sepa como ordenarlos. Como se explico en el apartado de conjuntos, cuando se desea que la ordenación siga un orden diferente, o simplemente los elementos no son ordenables de forma natural, hay que facilitar un mecanismo para que se pueda producir la ordenación. Los tipos "ordenables" de forma natural son los enteros, las cadenas (orden alfabético) y las fechas, y por defecto su orden es ascendente.

La clase `Collections` y la clase `Arrays` facilitan el método `sort`, que permiten ordenar respectivamente listas y arrays. Los siguientes ejemplos ordenarían los números de forma ascendente (de menor a mayor):

Ordenación natural en listas y arrays.	
Ejemplo de ordenación de un array de números	**Ejemplo de ordenación de una lista con números**
	`ArrayList<Integer> lista=new ArrayList<Integer> ();`
`Integer[] array={10,9,99,3,5};`	`lista.add(10); lista.add(9);lista.add(99);`
`Arrays.sort(array);`	`lista.add(3); lista.add(5);`
	`Collections.sort(lista);`

2 Algoritmos (II).

En Java hay dos mecanismos para cambiar la forma en la que los elementos se ordenan. Volvamos al ejemplo tratado ya en alguna ocasión en el curso, relativo a un conjunto de artículos y que quisiéramos ordenar por código del artículo. Imagina que tienes los artículos almacenados en una lista llamada "`articulos`", y que cada artículo se almacena en la siguiente clase (fíjate que el código de artículo es una cadena y no un número):

```
class Articulo {
```

```
    public String codArticulo; // Código de artículo
```

```
    public String descripcion; // Descripción del artículo.
```

```
    public int cantidad; // Cantidad a proveer del artículo.
```

```
}
```

La primera forma de ordenar consiste en crear una clase que implemente la interfaz `java.util.Comparator`, y en ende, el método método `compare` definido en dicha interfaz. Esto se explicó en el apartado de conjuntos, al explicar el `TreeSet`, así que no vamos a profundizar en ello. No obstante, el comparador para ese caso podría ser así:

```
class comparadorArticulos implements Comparator<Articulo>
```

```
{
```

```
    @Override
```

```
    public int compare( Articulo o1, Articulo o2) { return o1.codArticulo.compareTo(o2.codArticu
lo); }
```

```
}
```

Una vez creada esta clase, ordenar los elementos es muy sencillo, simplemente se pasa como segundo parámetro del método `sort` una instancia del comparador creado:

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma es quizás más sencilla cuando se trata de objetos cuya ordenación no existe de forma natural, pero requiere modificar la clase `Articulo`. Consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz `java.util.Comparable`. **Todos los objetos que**

implementan la interfaz `Comparable` son "ordenables" y se puede invocar el método `sort` sin indicar un comparador para ordenarlos. La interfaz `comparable` solo requiere implementar el método `compareTo`:

```
class Articulo implements Comparable<Articulo>{
```

```
    public String codArticulo;
```

```
    public String descripcion;
```

```
    public int cantidad;
```

```

```

```
    @Override
```

```
    public int compareTo(Articulo o) { return codArticulo.compareTo(o.codArticulo); }
```

```
}
```

Del ejemplo anterior se pueden denotar dos cosas importantes: que la interfaz `Comparable` es genérica y que para que funcione sin problemas es conveniente indicar el tipo base sobre el que se permite la comparación (en este caso, el objeto `Articulo` debe compararse consigo mismo), y que el método `compareTo` solo admite un parámetro, dado que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método `compareTo` es el mismo que el método compare de la interfaz `Comparator`: si la clase que se pasa por parámetro es igual al objeto, se tendría que retornar 0; si es menor o anterior, se debería retornar un número menor que cero; si es mayor o posterior, se debería retornar un número mayor que 0.

Ordenar ahora la lista de artículos es sencillo, fíjate que fácil: "`Collections.sort(articulos);`"

Autoevaluación

Si tienes que ordenar los elementos de una lista de tres formas diferentes, ¿cuál de los métodos anteriores es más conveniente?

- ○ Usar comparadores, a través de la interfaz java.util.Comparator.
- ○ Implementar la interfaz comparable en el objeto almacenado en la lista.

3 Algoritmos (III).

¿Qué más ofrece las `clases java.util.Collections` y `java.util.Arrays` de Java? Una vez vista la ordenación, quizás lo más complicado, veamos algunas operaciones adicionales. En los ejemplos, la variable "`array`" es un array y la variable "lista" es una lista de cualquier tipo de elemento:

Operaciones adicionales sobre listas y arrays.		
Operación	**Descripción**	**Ejemplos**
Desordenar una lista.	Desordena una lista, este método no está disponible para arrays.	`Collections.shuffle (lista);`
Rellenar una lista o array.	Rellena una lista o array copiando el mismo valor en todos los elementos del array o lista. Útil para reiniciar una lista o array.	`Collections.fill (lista,elemento);` `Arrays.fill (array,elemento);`
Búsqueda binaria.	Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.	`Collections.binarySearch(lista, elemento);` `Arrays.binarySearch(array, elemento);`

Convertir un array a lista.	Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es `ArrayList` ni `LinkedList`), solo se especifica que retorna una lista que implementa la interfaz `java.util.List`.	`List lista=Arrays.asList(array);`
		Si el tipo de dato almacenado en el array es conocido (`Integer` por ejemplo), es conveniente especificar el tipo de objeto de la lista:
		`List<Integer>lista = Arrays.asList(array);`

Convertir una lista a array.	Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase `Collections`, sino propio de la interfaz `Collection`. Es conveniente que sepas de su existencia.	Para este ejemplo, supondremos que los elementos de la lista son números, dado que hay que crear un array del tipo almacenado en la lista, y del tamaño de la lista:
		`Integer[] array=new Integer[lista.size()];`
		`lista.toArray(array)`

Dar la vuelta.	Da la vuelta a una lista, poniéndola en orden inverso al que tiene.	`Collections.reverse(lista);`

Otra operación que no se ha visto hasta ahora es la dividir una cadena en partes. Cuando una cadena está formada internamente por trozos de texto claramente delimitados por un separador (una coma, un punto y coma o cualquier otro), es posible dividir la cadena y obtener cada uno de los trozos de texto por separado en un array de cadenas. Es una operación sencilla, pero dado que es necesario conocer el funcionamiento de los arrays y de las expresiones regulares para su uso, no se ha podido ver hasta ahora. Para poder realizar esta operación, usaremos el método `split` de la clase `String`. El delimitador o separador es una expresión regular, único argumento del método `split`, y puede ser obviamente todo lo complejo que sea necesario:

```
String texto="Z,B,A,X,M,O,P,U";
```

```
String []partes=texto.split(",");
```

```
Arrays.sort(partes);
```

En el ejemplo anterior la cadena `texto` contiene una serie de letras separadas por comas. La cadena se ha dividido con el método `split`, y se ha guardado cada carácter por separado en un array. Después se ha ordenado el array. ¡Increíble lo que se puede llegar a hacer con solo tres líneas de código!