Tema 6. Colecciones (*Java Collections Framework*)

Collection

Es un interface que representa la funcionalidad de un grupo de objetos del mismo tipo a los que se denomina *elementos* (de la colección). No tiene una implementación directa (aunque si es implementada indirectamente a través de las implementaciones de sus interfaces hijas). Se utiliza especialmente para mover objetos de tipo colección a lo largo de las clases y métodos, en aquellos lugares donde se desee la máxima generalidad. Se trata de la **clase raíz** de la jerarquía de colecciones de la API de Java.

Un ejemplo:

```
Collection<Object> coleccion = new ArrayList();
```

Para una *Collection* el número de elementos que tiene no es una constante (como sucede en los *arrays* estáticos) sino que varía en el tiempo. Es decir, son entes dinámicos, que pueden aumentar (e incluso disminuir) su tamaño a petición.

Dos posibles criterios para clasificar las colecciones son:

- Unicidad. Según si permiten o no la existencia de elementos repetidos.

- Orden. Según si garantizan o no que los elementos estén ordenados.

El recorrido de los elementos de toda *Collection* se puede realizar de dos formas principales. O bien mediante bucles *forEach*, o bien mediante *iteradores*. Existen alternativas para recorrerlas, como por ejemplo empleando recursividad.

Todas las implementaciones de propósito general de una *Collection* deberían proporcionar al menos 2 constructores distintos. Uno vacío (sin parámetros), que crea una colección vacía, y otro con un solo parámetro de tipo *Collection*, que crea una nueva colección con los mismos elementos que la colección pasada como parámetro. Sin embargo, dado que una interfaz no puede implementar constructores, no se puede asegurar esta propiedad deseable de los 2 constructores, solo aconsejar.

Set

Es un subtipo de *Collection* que se caracteriza porque:

- No admite la repetición de elementos (no puede haber dos elementos idénticos). Más formalmente: no contiene un par de elementos *e1* y *e2* tal que *e1.equals(e2)*.

- Al recorrer sus elementos, no se obtienen en el mismo orden en el que se añadieron.

- Una diferencia con la interfaz *Collection* es que define **propiedades** especiales acerca de sus **constructores**, como es el hecho de que cualquier constructor de una de las implementaciones de *Set* debe crear siempre un conjunto que no contenga elementos repetidos. Así, por ejemplo, si creamos una colección *Set* a partir de otra *Collection* (un *ArrayList*, entre otros), se debe asegurar que, aunque el *ArrayList* tenga

elementos repetidos, el conjunto final *Set* que se construye no debe tenerlos.

- Otra diferencia es que el método ***add* no añadirá elementos repetidos**, porque la propia estructura no los permite. Por tanto, en caso de añadir un elemento repetido, no producirá un error, pero no modificará el *Set*, y devolverá *false*. En caso de que se intente añadir un elemento no repetido, y además se consiga, devolverá *true*.

Una implementación típica de la interfaz *Set* sería:

HashSet. Organiza los elementos en una tabla hash. Una tabla hash (o matriz asociativa) es una estructura de datos que asocia llaves o claves con valores. Es imprescindible que el tipo de datos de los elementos tenga redefinido los métodos *equals*() y *hashCode*() heredados de *Object*. De esta forma, con *equals* se podrá verificar la unicidad del conjunto, y con *hashCode* se creará la clave única necesaria para la tabla hash. Esta implementación permite elementos *null*, aunque en la práctica, solo contendrá 1 (si se añade), puesto que al ser un conjunto, aunque intentáramos añadir más elementos *null*, serían repetidos, y por tanto no se añadirían.

SortedSet

Interfaz que representa un conjunto de elementos ordenados. Extiende (*extends*) a la interfaz *Set*. Por tanto, tampoco permite elementos repetidos, pero en este caso, los elementos que contiene el conjunto están ordenados en base a un criterio.

TreeSet. Implementación que ordena sus elementos en un árbol binario. Los elementos se obtienen según el orden indicado que se establezca:

* Orden natural del elemento: es necesario que el tipo de los elementos que contiene el árbol implemente la interfaz *Comparable*.

* Orden total: establecido a través de un objeto *Comparator* que se fija en el constructor del *TreeSet*. Se utiliza cuando los elementos no son comparables o se quiere ordenar los elementos por otro criterio distinto al de su orden natural.

Algo a tener en cuenta, es que *TreeSet* **no siempre permite el valor *null***. Si utiliza el orden natural de los elementos (no se le pasa un *Comparator*), no permitirá el valor *null*, lanzando un *NullPointerException* (*NPE*) en caso de intentar añadir o utlizar uno.

Por otro lado, si el *TreeSet* se forma mediante un constructor al que se le pasa un comparador al inicio, y, solo si dicho comparador implementa un manejo apropiado para los elementos *null*, el *TreeSet* podrá admitir dicho valor.

Esto se debe a que, al añadir elementos al conjunto, el *TreeSet* compara el elemento actual con el resto de elementos del árbol, para poder insertarlo ya ordenado. Al comparar, llama o bien al método *compareTo* (orden natural - interfaz *Comparable*) o bien al método *compare* (orden total – interfaz *Comparator*). Por tanto, si se intenta invocar uno de esos 2 métodos sobre un objeto nulo, o pasando un objeto nulo como parámetro, se producirá un error y lanzará una excepción (*NPE*).

Queue

Es un subtipo de *Collection* que se caracteriza porque:

- Admite la repetición de elementos.

- Típicamente, ordena sus elementos con una **estructura FIFO** (*First In, First Out*). Algunas implementaciones concretas, como por ejemplo las colas de prioridad, ordenan los elementos de forma acorde al comparador que se le proporcione, o bien al orden natural de los elementos que esté encolando. Por otro lado, están las *LIFO queues* (*Last In, First Out*), también conocidas como *stacks* (pilas), que siguen un orden inverso a las colas tradicionales.

  Sea cual sea el orden de encolado, los elementos siempre se añaden por el final de la cola (*tail*), y se consultan o extraen por el principio de la cola (*head*).

Además de las operaciones comunes a toda colección, ofrece de forma complementaria la inserción, consulta o extracción de elementos. Cada una de estas operaciones tiene 2 formas: si la operación falla, en una de ellas lanza una excepción, y en la otra, devuelve un valor especial, que puede ser *null* o *false*.

Operación	Comportamiento ante **fallo** de la operación	
	Lanza excepción	***Devuelve valor especial***
Insert	*add(E e)*	*offer(E e)* (false)
Remove	*remove()*	*poll()* (null)
Examine	*element()*	*peek()* (null)

En relación a la operación de inserción, es preferible utilizar la forma *offer(E e)* a utilizar la forma *add(E e)*, cuando se trata de una *Queue* de capacidad restringida. En la mayoria de casos, cuando la capacidad es dinámica o ilimitada, la operación de inserción simplemente no puede fallar, y se utiliza *add(E e)*.

En general, las implementaciones de *Queue* no permiten la inserción de elementos *null* como uno más de la colección, aunque algunas implementaciones concretas, como *LinkedList*, no prohiben expresamente hacer esto. Pero, incluso aunque la implementación lo permitiese, los elementos de este tipo no deberían insertarse dentro de la *Queue*, puesto que *null* es también utilizado como tipo de retorno de operaciones particulares de las colas, como *pick()* o *poll()*, para inidicar que la cola no contiene elementos, y, por tanto, podría llevar a confusiones.

List

Es un subtipo de *Collection* que se caracteriza porque:

- Se admite la repetición de elementos.

- Cuando se recorren los elementos, estos se obtienen en el mismo orden en que fueron añadidos (de ahí que a una lista también se la conozca como una secuencia).

- Se puede acceder a un elemento concreto facilitando su posición mediante un índice de tipo entero (el 0 representa la posición del primer elemento).

- Admiten el elemento *null*, y, además, multiples instancias del mismo, si las hay.

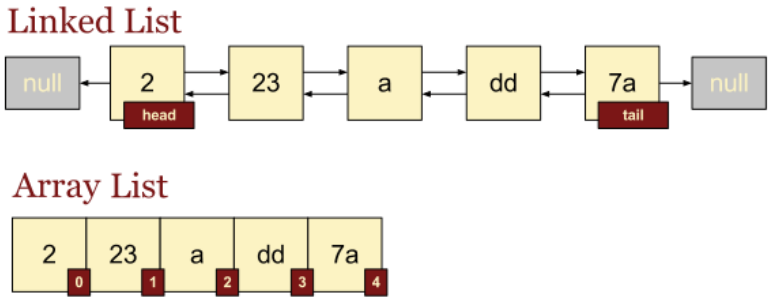
- Pueden contenerse a si mismas como elementos. Es decir, los elementos de una lista, pueden ser a su vez listas, pudiendo formar listas de listas. Esto permite algunas implementaciones de *List* recursivas, como el caso de *LinkedList*, donde una lista enlazada puede estar vacía, o bien tener un elemento seguido de otra sublista enlazada (la cual a su vez puede ser vacía, o bien tener otro par elemento-subLista).

Implementaciones de *List* especializadas **según el tipo de acceso que se realiza**:

ArrayList. Es la implementación que debería utilizarse cuando el acceso a los elementos es aleatorio (mediante un índice).

LinkedList. Es la implementación que debería utilizarse cuando el acceso a los elemento se produce secuencialmente (del primero al último o del último al primero).

Básicamente, ambas implementaciones se diferencian en el coste computacional (complejidad ciclomática) a la hora de ejecutarse, distinguiéndose entre el coste en memoria y en tiempo de ejecución. En la mayoría de los casos, es preferible utilizar *ArrayList*, puesto que el tiempo de sus operaciones y uso de memoria es inferior.



Algunos métodos interesantes que operan con *List* son:

* método *asList(T... values)* de la clase *Arrays*. Devuelve un objeto *List<T>* compuesto por todos los elementos pasados como parámetros.

```
List<String> nombresRios = Arrays.asList("Duero", "Sil", "Tajo");
```

* método *sort(List<T> list)* de la clase *Collections*. Ordena los elementos de la lista según su orden natural. Por tanto, es necesario que los elementos a ordenar implementen la interfaz *Comparable*. Modifica la lista original.

* método *sort(List<T> list, Comparator<T> comparator)* de la clase *Collections*. Ordena los elementos de la lista según el orden establecido por el comparador pasado como parámetro. También modifica la lista original.

Iteradores (*Iterable* e *Iterator*)

En primer lugar, vamos a hablar de la interfaz *Iterable*. Es una interfaz muy simple, que, al igual que, por ejemplo, la interfaz *Comparable*, simplemente define una "propiedad" o característica de la clase que la implementa. Viene a decir que los objetos de la clase que implemente dicha interfaz, son iterables.

Además, la clase que la implemente debe redefinir la implementación de un único

método: *iterator()*, que precisamente devuelve un iterador (objeto de tipo *Iterator*) sobre los elementos del tipo de la clase que sea *Iterable*.

La interfaz *Iterator* permite crear objetos de tipo iterador, los cuales sirven para recorrer colecciones secuencialmente, utilizando una copia de las mismas. Esta interfaz obliga, a la clase que la implemente, a redefinir 2 métodos:

- *hasNext()*. Devuelve *true* si el iterador tiene un siguiente elemento, y *false* en caso contrario. Es decir, sirve para comprobar si quedan elementos por recorrer.

- *next()*. Devuelve el elemento siguiente en el recorrido.

La gran **ventaja** de trabajar con iteradores es que se trabaja sobre copias, en lugar de hacerlo sobre las colecciones originales, y además, permiten recorrer cualquier colección dentro de la jerarquía de colecciones, sea la que sea.

Al trabajar sobre copias, no modificamos ni destruímos la original, y permite trabajar igualmente realizando consultas o modificaciones, al tener disponibles las referencias (punteros) a los objetos de la colección.

Por otro lado, se debe tener en cuenta que, por ejemplo, no todas las colecciones de objetos en Java tienen un índice entero asociado a cada objeto, con lo cual no todas se pueden recorrer basándose en un índice. En cambio, siempre se podrá utilizar un iterador para recorrer una colección.

ListIterator

Un tipo especial de iterador es el de la interfaz *ListIterator*, la cual extiende la interfaz *Iterator*. Se trata básicamente de un iterador que solo puede operar sobre listas, y permite algunas cosas que no permiten los iteradores normales:

- Recorrer la lista en dos direccciones, hacia delante y hacia atrás.

- Añadir un elemento a la lista durante el recorrido (método *add(E e)*).

- Modificar un elemento de la lista durante el recorrido (método *set(E e)*).

- Obtener la posición actual del iterador en el recorrido que hace de la lista.

Map

Un *Map* es un tipo de objeto que asocia claves (*keys*) con valores (*values*). Al definir un objeto mapa, es preciso indicar el tipo de las claves (*K*) y el de los valores (*V*).

A cada uno de los elementos de un *Map* se le denomina genéricamente como "entrada". Las entradas de un mapa están formadas por la pareja (clave, valor) y su tipo de dato es *Map.Entry<K, V>*. En un *Map* no puede haber dos entradas con la misma clave, así como una misma clave no puede referenciar a dos valores distintos.

La interfaz *Map* no permite contener elementos de tipo *Map* como claves para sus entradas. Sin embargo, si que permite contener elementos *Map* como valores de sus entradas. Esto permite, de forma similar a como ocurría con las listas de listas, crear mapas que a su vez contengan otros mapas como valores de las entradas.

Cualquier implementación de *Map* proporciona 3 vistas diferentes de sus elementos:

- Como un conjunto de claves (método *keySet*). Devuelve un *Set*, precisamente, porque en un mapa no puede haber claves repetidas.

- Como una colección de valores (método *values*).

- Como un conjunto de asociaciones clave-valor, es decir, entradas (método *entrySet*).

La implementación más conocida para la interfaz *Map* es:

HashMap. En ella las entradas del mapa se organizan en una tabla hash. Es por tanto necesario que el tipo de objetos establecido para las claves redefina los métodos *equals()* y *hashCode()* de la clase *Object*. Esta implementación no garantiza ningún orden para sus entradas. Además, permite *null*, tanto como valores de las entradas, así como también permite una clave de tipo *null*.

SortedMap

Interfaz que representa un mapa de entradas ordenados. Extiende (*extends*) a la interfaz *Map*. Por tanto, también permite valores repetidos, pero no claves, las cuales han de ser únicas. Sin embargo, la diferencia está en que, en este caso, las entradas que contiene el mapa están ordenados en base a un criterio.

TreeMap. Implementación del mapa que ordena sus elementos (entradas) en un árbol binario, donde las claves del *Map* serán las que dicten el orden de dichas entradas. Se podrá establecer el orden de 2 formas:

* Orden natural: es necesario que las claves implementen la interfaz *Comparable<K>*.

* Orden total: establecido a través de un objeto *Comparator<K>* fijado en el constructor del *TreeMap*, el cual será el que determine cómo deben compararse las claves. Se utiliza cuando los elementos no son comparables o se quiere ordenar las entradas por otro criterio distinto al del orden natural de sus claves.

Collections

Es una clase final (no interfaz), que consta únicamente de métodos estáticos, los cuales operan sobre objetos, o devuelve objetos, que son de tipo colección (*Collection*). Se trata básicamente de un conjunto de utilidades para las colecciones.

Algunos de los métodos que contiene son polimórficos, porque sirven para operar indistintamente sobre cualquier tipo de colección, y otros operan con, o sobre, colecciones concretas. Además, todos los métodos de la clase lanzan una excepción del tipo *NullPointerException* si alguno de los parámetros pasados es de tipo *null*.

Algunos de los métodos principales o más usados son:

• *addAll(Collection<T> c, T... elements)*.

Permite añadir todos los elementos especificados a la colección especificada. Los elementos a añadir se pueden pasar individualmente o como un array estático.

```
Collections.addAll(listaRios, "Duero", "Ebro", "Tajo");

Collections.addAll(listaRios, String[]);
```

● *copy(List<T> destination, List<T> source)*.

Copia todos los elementos de la segunda lista a la primera lista. La lista de destino debe ser igual o más grande que la lista de origen (igual o mayor n° de elementos).

```
Collections.copy(listaDestino, listaOrigen);
```

● *disjoint(Collection<T> c1, Collection<T> c2)*.

Devuelve *false* si ambas colecciones tiene algún elemento en común, y devuelve *true* en caso de que no tengan ningún elemento en común (sean disjuntas).

```
Collections.disjoint(listaRios, riosNuevo);
```

● *shuffle(List<T> list)*.

Permuta aleatoriamente los elementos de una lista usando una fuente de aleatoriedad por defecto. Se debe tener en cuenta que es un método "destructivo", puesto que modifica la lista original pasada como parámetro. Todas las permutaciones ocurren aproximadamente con la misma probabilidad.

```
Collections.shuffle(listaSinMezclar);
```

Otros métodos interesante son el método ***frequency*** (devuelve el n° de elementos iguales a uno dado que existen en la lista que se indique, es decir, su frecuencia), el método ***reverse*** (que invierte el orden actual de la lista que se indique), el método ***sort*** (que ya hemos visto, y que está sobrecargado, admitiendo solo una lista – orden natural, o admitiendo una lista y un comparador – orden total) o por ejemplo, por último, el método ***swap*** (que intercambia los elementos de dos posiciones especificadas sobre una lista concreta).