Entornos de Desarrollo
Bloque 2
Tema 8: Uso de colecciones

por Nacho Iborra

Hasta ahora, siempre que hemos necesitado trabajar con un conjunto de elementos de un determinado tipo hemos utilizado *arrays*. Sin embargo, los *arrays* son estáticos, es decir, no podemos modificar su tamaño, por lo que no son adecuados para aplicaciones en las que la cantidad de objetos a almacenar puede variar a medida que se ejecuta la aplicación. En estos casos deberíamos utilizar una **colección**.

2.8.1. Introducción a las colecciones

Una colección es una estructura de datos **dinámica** que contiene objetos, los cuales queremos almacenar y operar con ellos de una manera concreta. El tamaño de la colección podrá variar durante la ejecución de un programa según vayamos añadiendo o eliminando elementos de la colección.

En Java existen varios tipos de colecciones, aunque nos centraremos en tres subtipos concretos:

- **Listas (*Lists*):** colecciones en las cuales cada elemento tiene asignado un índice numérico (como en un *array*), y podemos explorar toda la colección yendo de un índice al siguiente.
- **Mapas (*Maps*):** también llamados tablas *hash* o diccionarios, almacenan pares clave-valor, en los cuales, dando la clave podemos obtener el valor correspondiente asociado. La clave puede ser de cualquier tipo (numérico, cadena o incluso objetos complejos), por lo que no tenemos ningún índice numérico para explorar la colección.
- **Conjuntos (*Sets*):** una colección de elementos sin duplicados. Los elementos no tienen un índice numérico asociado y, generalmente exploraremos el conjunto utilizando un *"iterador"*.

Dependiendo de la aplicación que estemos desarrollando, deberemos elegir uno de estos tipos (o varios). En los siguientes apartados vamos a conocer las clases más populares que podemos utilizar en Java para representar y trabajar con cada tipo de colección. La mayor parte de estas clases pertenecen al paquete `java.util`, por lo tanto necesitaremos importarlo en nuestros proyectos.

2.8.2. Listas

Las listas son colecciones en las cuales los elementos están indexados, por lo que pueden ser ordenadas e iteradas. En Java, cada tipo de lista implementa una interfaz global llamada `List`, aquí puedes consultar el API de la interfaz.

Si echas un vistazo al API, existen algunos métodos que podemos utilizar con cualquier tipo de lista, como los siguientes:

- `add(element)` o `add(index, element)`: para añadir un elemento, al final de la lista o, en un índice dado.

- `clear()`: limpia la lista (elimina todos sus elementos).
- `contains(element)`: comprueba si el elemento indicado ya existe en la lista (siempre que la clase sobrecargue el método `equals` para saber como comprobar si dos elementos son iguales o no).
- `get(index)`: obtiene el elemento de la posición indicada.
- `indexOf(element)`: obtiene el índice de la primera ocurrencia encontrada del elemento especificado (siempre que la clase implemente el método `equals`). Si el elemento no existe en la lista, devolverá -1.
- `remove(index)`: elimina el elemento del índice especificado.
- `remove(element)`: eliminará la primera ocurrencia del elemento indicado en la lista (siempre que la clase sobrecargue el método `equals`).
- `size()`: devuelve el número total de elementos almacenados en la lista.

2.8.2.1. ArrayList

El subtipo más popular de lista que podemos utilizar en Java es la clase `ArrayList` (API). Como implementa el interfaz `List`, podemos utilizar todos sus métodos.

Podemos crear un ArrayList definiendo una variable `ArrayList` o una variable `List` (utilizando polimorfismo):

```
List myList = new ArrayList();
ArrayList myOtherList = new ArrayList();
```

Por lo que, podemos añadir, obtener, eliminar... elementos de esta lista. Estos elementos pueden ser de cualquier tipo, por lo que deberemos tener cuidado cuando obtengamos elementos de la lista, y asegurarnos que sean del tipo apropiado.

```
myList.add("Hello");
myList.add(new Person("Nacho", 40));
...
if (myList.get(1) instanceof Person)
{
    Person p = (Person)(myList.get(1));
    System.out.println(p.getName());
}
```

2.8.2.2. Usando genéricos

Si utilizamos listas como hemos visto en el ejemplo anterior, podemos tener algunos problemas, ya que podemos añadir cualquier tipo de objeto en la lista, por lo que necesitamos comprobar y convertir los

tipos antes de trabajar con ellos.

Para evitar estas comprobaciones, podemos utilizar **genéricos**. Los genéricos son la forma de personalizar un determinado objeto o colección y trabajar así con un determinado tipo de datos concreto. Este tipo de datos se expresará entre los símbolos `<` y `>` después del tipo de colección.

Por ejemplo, si queremos trabajar con una lista de cadenas utilizando genéricos, inicializaremos la lista de la siguiente forma (tanto si usamos `List` como `ArrayList`):

```
List<String> stringList = new ArrayList<>();
ArrayList<String> anotherStringList = new ArrayList<>();
```

Desde este momento, cada elemento que añadamos a la colección necesita ser una cadena, y cada vez que obtengamos un elemento de la colección, podemos estar seguros que será una cadena, por lo que no será necesario comprobar y convertir.

```
stringList.add("Hello");
stringList.add("Goodbye");
System.out.println(stringList.get(1).toUpperCase()); // GOODBYE
```

Ten en cuenta que podemos utilizar **polimorfismo** con genéricos, por lo que, si creamos una lista de objetos `Animal`, por ejemplo, puede haber cualquier tipo de animal en esta lista (perros, patos, etc).

2.8.2.3. Otros subtipos de listas

Hay otros subtipos de listas disponibles en el API de Java, aunque no trabajaremos con ellos en este tema. Aquí puedes dar un vistazo rápido:

- La clase `Vector` es similar a `ArrayList`, pero ofrece mayor seguridad para procesos (es decir, es adecuada para trabajar con múltiples procesos, está sincronizada), pero no es tan eficiente como ArrayList.
- La clase `LinkedList` es otro subtipo de lista en la cual cada elemento no sólo está enlazado con el siguiente, sino con el anterior, por lo que podemos explorar la lista desde cualquiera de sus lados con la misma eficiencia.
- La clase `Stack` es un subtipo especial de lista para definir una pila, es decir, una lista en la que los elementos se añaden y eliminan siempre desde el mismo lado. Tiene una estructura LIFO (*Last In, First Out*, último en entrar, primero en salir).
- **Queues** es otro subtipo especial de lista para definir colas, es decir, una lista en la que los elementos se añaden por un lado y se eliminan por el opuesto. Es una estructura FIFO (*First In, First Out*, primero en entrar, primero en salir). Realmente, `Queue` no es una subclase de lista en Java, sino una interface, pero su significado y comportamiento es similar a una lista.

2.8.2.4. Ordenar listas automáticamente

De la misma manera que utilizamos los interfaces `Comparator` o `Comparable` para determinar como clasificar objetos complejos en un *array* a través del método `Arrays.sort`. Existe el método `Collections.sort` (en la clase `java.util.Collections`) que nos permite ordenar cualquier tipo de colección utilizando un comparador.

Todo lo que necesitamos es definir el método de comparación (ya bien sea en la propia clase a ordenar, o en otra clase a través de un `Comparator`), y llamar entonces a este método para ordenar la colección. Por ejemplo, si tenemos una lista u objetos `Person` como el *array* utilizado en el tema anterior, podemos ordenarlos utilizando la misma clase comparador:

```
List<Person> people = new ArrayList<>();
... // Fill list
Collections.sort(people, new Comparator<Person>()
{
    @Override
    public int compare(Person p1, Person p2)
    {
        return Integer.compare(p2.getAge(), p1.getAge());
    }
});
```

Si la propia clase (`Person` en este ejemplo) tiene el método de comparación en su código, podremos llamar a `Collections.sort` con sólo un parámetro (la colección a ordenar):

```
Collections.sort(people);
```

> **Ejercicios propuestos**
>
> **2.8.2.1.** Volviendo al ejercicio 2.6.4.1. del tema 6. Sustituye el *array* de videojuegos en el método `main` por un `ArrayList` genérico. Una vez hecho, añade algunos videojuegos a la lista (ya sean objetos del tipo `VideoJuego` o `PCVideoJuego`), explora y muestra la lista con un `for` y pregunta al usuario:
>
> - Buscar videojuegos por título: el usuario escribirá un título y el programa deberá mostrar todos los videojuegos que contengan dicho título (ignorando mayúsculas y minúsculas).
> - Eliminar un videojuego de la lista: el usuario introducirá el índice del videojuego a eliminar, y si el índice es válido, el videojuego que esté en ese índice será eliminado.
>
> **2.8.2.2.** Ordena la lista anterior por precio ascendentemente utilizando `Collections.sort` y el método de comparación apropiado. Realiza este paso antes de mostrar la lista en pantalla.

2.8.3. Mapas

Los mapas son un tipo dinámico de colección, en los cuales, cada elemento o valor está referenciado por una clave. También son llamados tablas *hash* o diccionarios, porque funcionan como un diccionario: si conocemos la palabra que estamos buscando, podemos saltar a ella (sin tener que explorar todas las palabras previas) y comprobar su significado.

En Java, cada tipo de mapa implementa un interfaz global llamado `Map`, aquí puedes ver la API de su interfaz.

Si echas un vistazo al API, hay algunos métodos que podemos utilizar en cualquier tipo de lista, como:

- `clear()`: limpia el mapa (elimina todos sus elementos).
- `containsKey(key)`: comprueba si el mapa contiene la clave indicada.
- `get(key)`: obtiene el valor asociado a la clave indicada.
- `put(key, value)`: añade el par clave-valor especificado en el mapa.
- `remove(key)`: elimina el par clave-valor identificado por la clave dada.
- `size()`: obtiene el número total de elementos almacenados en el mapa.

2.8.3.1. HashMap

El subtipo de mapa más popular que podemos utilizar en Java es la clase `HashMap` (API). Como implementa la interfaz de `Map`, podemos utilizar todos sus métodos.

Podemos crear un *HashMap* definiendo una variable `HashMap`o una variable `Map` (utilizando polimorfismo). En ambos casos, deberíamos utilizar genéricos (aunque no es obligatorio), para establecer el tipo de datos tanto de la clave como del valor. En el ejemplo siguiente, definimos un mapa en el cual las claves son *strings* y lo valores son objetos `Person`. Luego añadimos algunos elementos al mapa (fíjate como indicamos tanto la clave como el valor en el mapa), y buscamos un elemento dado por su clave:

```
Map<String, Person> myMap = new HashMap<>();
myMap.put("11223344A", new Person("Nacho", 40));
myMap.put("22334455B", new Person("Arturo", 35));
...
// Print the name of the Person with key = 11223344A
System.out.println(myMap.get("11223344A").getName());
```

2.8.3.2. Explorar mapas

Si queremos explorar un mapa, no podemos utilizar el tradicional `for` e incrementar un índice para ir a cada posición, ya que no hay un índice numérico. En su lugar, necesitamos obtener el conjunto de claves (con el método `keySet`), y explorarlo con un `foreach`:

```
for(String key : myMap.keySet())
{
    System.out.println(myMap.get(key).getName());
}
```

> **Ejercicios propuestos**
>
> **2.8.3.1.** Crea un proyecto llamado **Biblioteca**, con una clase principal llamada *Biblioteca* y otra clase llamada *Libro*, en la cual iremos almacenando información sobre cada libro: el *id* (cadena), el título y el nombre del autor. En la aplicación principal, definiremos un mapa de libros, en el cual las claves serán sus correspondientes *ids*. Añade manualmente algunos libros en el mapa, y luego explora el mapa mostrarndo algunos libros en pantalla (sobrescribe el método `toString` de la clase `Libro` correctamente para mostrar la información).

2.8.4. Conjuntos

Un conjunto es una colección de elementos sin duplicados. En Java, cada tipo de conjunto implementa una interfaz global llamada `Set` (aquí puedes ver el API de su interfaz).

Si le echas un vistazo a la API, hay algunos métodos que podemos utilizar con cualquier tipo de lista, como:

- `add(element)`: añade un elemento al conjunto, siempre que no exista previamente.
- `clear()`: limpia el conjunto (elimina todos sus elementos).
- `contains(element)`: comprueba si el elemento indicado ya existe en el conjunto (siempre que se haya sobrecargado el método `equals` para saber como comprobar si dos elementos son iguales o no).
- `iterator()`: obtiene un *"iterador"* y poder así explorar los elementos del conjunto con él.
- `remove(element)`: elimina del conjunto el elemento indicado (siempre que esté sobrecargado el método `equals`).
- `size()`: obtiene el número total de elementos almacenados en el conjunto.

Como puedes ver, no hay un índice que especifique la posición de los elementos en el conjunto, por que debemos explorarlo con un *"iterador"*. Vamos a suponer que tenemos un conjunto de cadenas, podemos explorarlo de la siguiente forma:

```
Set<String> mySet = ...;
Iterator<String> it = mySet.iterator();
while(it.hasNext())
{
    String s = it.next();
    ...
}
```

2.8.4.1. HashSet

Una de las clases más populares para trabajar con conjuntos es la clase `HashSet` (API). Funciona como la clase *HashMap* vista anteriormente, pero sólo especificaremos las claves del mapa (no los valores). Como implementa la interfaz `Set`, podemos utilizar todos sus métodos.

Podemos crear un *HashSet* definiendo una variable `HashSet` o una variable `Set` (utilizando polimorfismo). En ambos casos, deberíamos utilizar genéricos (aunque no es obligatorio), para establece el tipo de datos de la clave. En el siguiente ejemplo, definimos un conjunto de cadenas y añadimos algunos valores. Luego eliminamos una cadena dada del conjunto:

```
Set<String> mySet = new HashSet<>();
mySet.add("Hello");
mySet.add("Hello"); // Will not work, "Hello" already exits
mySet.add("Goobye");
...
mySet.remove("Goodbye");
```

Ejercicios propuestos

2.8.4.1. Crea un proyecto llamado *FairyTaleSet* con una clase principal que almacene un conjunto de cuentos de hadas. Para cada cuento de hadas, vamos a almacenar su título y el número de páginas, por lo que deberás definir una clase *FairyTale* con los dos atributos correspondientes, un constructor y sus correspondientes *getters* y *setters*.

También sobrecarga el método `equals` para determinar cuando dos cuentos de hadas estarán considerados iguales: serán iguales cuando tengan el mismo título.

Luego, en la aplicación principal, define un conjunto de objetos `FairyTale`, y añade algunos a la lista. Prueba a añadir algunos cuentos de hadas, incluyendo algunos con el mismo título, para ver cuantos de ellos son añadidos finalmente al conjunto. Después, explora el conjunto con un *"iterador"* y muestra en pantalla la información (también puedes sobrecargar el método `toString`).

NOTA: para determinar si dos cuentos son el mismo por el título, además de redefinir el método `equals`, necesitamos redefinir también el método `hashcode` para generar un código hash con el título (sin incluir el número de páginas). De este modo, Java puede comparar dos códigos hash de dos cuentos diferentes y comprobar si son el mismo en cuanto a título.