1. Refuerzo de Java

Anexo II. Más sobre colecciones

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Algunos subtipos más de Collection

En el contenido principal de esta parte de la unidad, se ha visto una descripción general de algunos de los subtipos de *Collection* más importantes, con respecto a listas, mapas y conjuntos. Pero hay otros subtipos que pueden ser útiles en ciertas aplicaciones, por tanto, también se les echará un vistazo.

1.1. Algunos subtipos de listas

Aparte del *ArrayList* o *LinkedList*, existen otros subtipos de listas que puedes utilizar en muchas aplicaciones, como las pilas (*stacks*), colas (*queues*) o vectores (*vectors*).

1.1.1. Vectores

Los vectores son una implementación de lista muy similar al *ArrayList*, pero con la principal diferencia de que las operaciones son todas **sincronizadas**, por lo que son una buena opción cuando se utiliza más de 1 hilo para operar con la lista. De lo contrario, se recomienda usar *ArrayList* en su lugar (más eficiente).

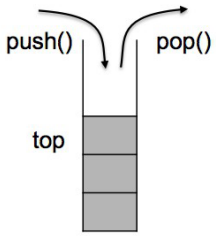
También se puede crear un *ArrayList* sincronizado (*thread safe*) en vez de usar un Vector:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

1.1.2. Pilas y colas

Una pila (*stack*) es una implementación de una lista LIFO (último en entrar, primero en salir). Hay una pila que extiende la clase *Vector*, por lo que tiene los mismos métodos más cinco métodos adicionales, que permiten que esta lista se comporte como una lista LIFO (métodos como *push*, *pop*, *peek*, …).



https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

Una cola (*queue*) es una implementación de una lista FIFO (primero en entrar, primero en salir). No hay una clase *Queue* en la API de Java, pero existe una interfaz *Queue*, y se puede usar cualquier clase que lo implemente para simular una cola. De hecho, es mejor utilizar una implementación de la interfaz *Queue* o *Deque* para simular una cola o una pila, en lugar de usar la clase *Stack* o cualquier otra clase antigua.

- ***Queue*** es un tipo especial de lista que está diseñado para añadir elementos siempre al final de la lista (métodos *add()* y *offer()*), y realizar las operaciones de obtención y eliminación al principio (FIFO o cola) o al final (LIFO o pila) de la lista. No está diseñada para acceder a posiciones con un índice (se debería usar una lista para eso). Cuando se desea limitar el uso a las operaciones al principio o al final de una lista, es más eficiente usar *Queue* (o *Deque*) que *List*.

  https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

- ***Deque*** es un tipo especial de cola que permite insertar y eliminar al principio o al final de la cola (cola de doble final), y que también se trata de una interfaz. Permite recuperar o eliminar un objeto que no está en el extremo de la cola.

  https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html

- Algunas de las clases que implementan estas interfaces son *ArrayDeque* y *LinkedList* (se habló de esto anteriormente).

  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html

1.2. Algunos subtipos de mapas

Con respecto a los mapas, existen algunos subtipos distintos de *Hashtable* o *HashMap*, aunque no son tan populares.

1.2.1. LinkedHashMap

La diferencia en esta implementación en comparación con *HashMap* es que, además de tener una estructura de mapa, se mantiene una lista interna enlazada entre los valores insertados. Esta lista vinculada se ordena manteniendo el mismo orden en que se insertaron los valores. El coste adicional de esta implementación es tener dos referencias adicionales por cada valor insertado (artículo anterior y siguiente). La ventaja es que se puede iterar a través de los elementos fácilmente en el mismo orden en que se añadieron, o generar una nueva copia de un *LinkedHashMap*, sabiendo que se mantendrá el orden.

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

1.2.2. TreeMap

Esta implementación no utiliza una función *hash*. En su lugar, almacena las claves y valores en una estructura de árbol (árbol rojo-negro[1]). En esta estructura, las claves deben mantener un orden, por lo que deben ser comparables. Por defecto, utiliza el orden natural para comparar claves (*Integers* → numérico, *Strings* → alfabético, …), pero con objetos más complejos, puedes implementar un ***Comparator*** para hacerlo (por ejemplo, objetos Persona que deben ordenarse por su número de DNI, …).

Esta implementación suele ser más costosa que un *HashMap* cuando se busca e inserta elementos, pero se compensa al no tener que calcular una función *hash* cada vez que se usa una clave, por lo que tiene sus ventajas.

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

1.3. Algunos subtipos de conjuntos

Finalmente, hay algunos subtipos de conjuntos (*sets*) que puedes utilizar, además del *HashSet*.

1 Árbol rojo-negro (https://es.wikipedia.org/wiki/Árbol_rojo-negro)

1.3.1. TreeSet

Se basa en *TreeMap* internamente con sus pros y sus contras. Los valores se almacenan como claves en el *TreeMap* interno.

https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

1.3.2. LinkedHashSet

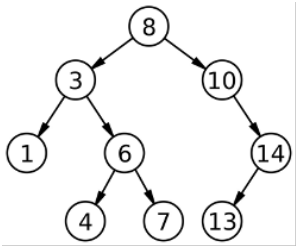
Es equivalente a *LinkedHashMap*, mantiene el orden interno en el que se insertaron los elementos.

https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html

2. Trabajar con árboles

Algunas de las implementaciones de mapas y conjuntos utilizan árboles para almacenar los valores, como *TreeMap* o *TreeSet*. Por lo general, los árboles son una estructura poco habitual en nuestros programas, aunque tienen algunas ventajas.

Un árbol es una estructura jerárquica con un valor raíz a partir del cual se puede encontrar un conjunto de nodos vinculados. Cada uno de estos nodos es un sub-árbol de hecho, con todos los nodos vinculados desde la raíz. Hay algunas restricciones: ningún nodo puede estar duplicado, no hay ciclos y no hay puntos de nodo a la raíz.



Este sería el ejemplo de un árbol. El nodo raíz está representado por el número 8 y tiene dos sub-árboles: uno representado por el número 3 y otro representado por el nodo con el número 10. El sub-árbol número 3 tiene dos sub-árboles más en su interior: los números 1 y 6, y así sucesivamente …

Se puede definir un árbol explorando sus nodos de forma recursiva (puede ser útil, por ejemplo, para explorar una estructura de directorios), y también se puede definir un árbol como una estructura ordenada, de modo que todos los nodos a la izquierda de un nodo sean inferiores a este nodo, y todos los nodos de su derecha sean mayores. El árbol que se muestra en la imagen de arriba es un árbol ordenado (todos los nodos del sub-árbol izquierdo son inferiores a 8, y todos los nodos del sub-árbol derecho son mayores que 8, y se puede decir lo mismo con cualquier sub-árbol). La principal ventaja de estos árboles ordenados, asumiendo que están equilibrados (es decir, todas las ramas tienen más o menos la misma profundidad), es que la complejidad de buscar un nodo dado es más simple que en una lista. Si una lista tiene N elementos, la complejidad de buscar un elemento es O(N); significa que se podría explorar hasta N elementos antes de encontrar el deseado. Sin embargo, si tenemos un árbol balanceado con N nodos, donde cada nodo tiene hasta M ramas, la complejidad de buscar un elemento en este árbol es $O(\log_M N)$, que es menor que O(N).

Existen varios tipos de estructuras de árbol, como árboles binarios (cada nodo tiene 2 ramas), árboles rojo-negro (árbol binario equilibrado), etc. No nos centraremos en implementar este tipo de árboles, sino en utilizar los que proporciona Java. Observa cómo funcionan con un ejemplo.

2.1. Ejemplo

Siguiendo con el ejemplo que se muestra en el Anexo I (el comparador de formas geométricas). Teníamos una estructura de clases con una interfaz *IShape*, y algunas clases que lo implementan, como *Circle*, *Triangle* y *Rectangle*. Todos tienen un método *getArea()* que calcula el área de cada forma. Se podría sobrecargar el método *toString()* en cada subclase (*Circle*, *Triangle* y *Rectangle*) para que impriman el tipo de forma y el área. Por ejemplo, en la clase *Rectangle* tendríamos este método:

```
@Override
public String toString() {
    return String.format("Rectangle -> Area: %.2f", getArea());
}
```

A partir de esta estructura de clases, se podría crear un *TreeSet* que almacene un conjunto de formas, ordenándolas automáticamente por su área. El programa principal podría ser como se muestra:

```
Set<IShape> set = new TreeSet<>(new Comparator<IShape>() {
    @Override
    public int compare(IShape s1, IShape s2) {
        return Double.compare(s1.getArea(), s2.getArea());
    }
});

set.add(new Triangle(6.25, 8));
set.add(new Triangle(7.2, 6.78));
set.add(new Circle(4.3));
set.add(new Circle(1.88));
set.add(new Rectangle(9.25, 7.6));
set.add(new Rectangle(9, 2.55));

for (IShape s : set) {
    System.out.println(s.toString());
}
```

Ten en cuenta que debes crear un nuevo objeto *Comparator* en una clase anónima e implementar el método *compare*, que comparará dos formas por su área. Luego, se añadirán algunas formas al árbol (desordenadas). Si ejecutas el programa, deberías obtener algo parecido a esto:

```
Circle -> Area: 11,10
Rectangle -> Area: 22,95
Triangle -> Area: 24,41
Triangle -> Area: 25,00
Circle -> Area: 58,09
Rectangle -> Area: 70,30
```

Ejercicio 1

Crea un proyecto llamado ***Companies*** con estas clases (incluido Main):

- ***Company***: Contendrá los atributos *name* y *money* (*double*) (valor establecido en el constructor).
- ***Person***: Tendrá los atributos *name* y *age* (valor establecido en el constructor).

En el ***main***, crea un *TreeMap* (ordenado por el dinero de las empresas) en el que la clave es la compañía y el valor es un *TreeSet* de personas (ordenado por edad). Completa el *TreeMap* con 3 empresas (no las añadas ordenadas por dinero) y cada empresa tendrá una lista de 3 personas (no ordenadas en el momento de añadirlas).

Recorre el *TreeMap* (use el método *entrySet()* para obtener un Conjunto ordenado de claves) y muestra las empresas con su personal (ambos deben ordenarse por dinero y edad).