Unidad 9

Colecciones y tipos abstractos de datos

Programación

9

Índice

Introducción

Las arrays nos permiten trabajar con agrupaciones de datos, pero su uso queda limitado si no conocemos el total de datos que disponemos y nos obliga a trabajar estáticamente haciendo ineficiente su uso. Pensar en un ejemplo de una carnicería que tiene que limitar el pase solo a 100 clientes para entrar a comprar, ¿Los 100 clientes tiene que estar todos a la misma hora?, ¿Hay alguna ordenación o prioridad? ¿Qué sucede si vienen más de 100 clientes?

Todas las cuestiones del ejemplo quedan resueltas en este tema. Cuando tenemos agrupaciones de datos dinámicamente que varían sus elementos en tiempo tenemos que hacer uso de un nuevo concepto, las colecciones.

Al finalizar esta unidad

- Sabremos que es una colección
- Conoceremos las diferentes colecciones que existen
- Analizaremos que colección es mejor

9.1. Colecciones y tipos abstractos de datos

Una colección es una estructura dinámica que nos permite agrupar elementos de una misma tipología y variar su tamaño según las necesidades que sean requeridas. A diferencia de los arrays, las colecciones ya tienen implementadas funcionalidades a través de sus interfaces que facilitan el trabajo con ellas, como, por ejemplo, los métodos de inserción, eliminación, ordenación, etc.

Existen diferentes tipos de colecciones, en java se dispone de las interfaces Collection con tres tipos diferentes Set, List y Queue y a parte tenemos Map que permite albergar colecciones haciendo uso de claves valor por cada elemento.

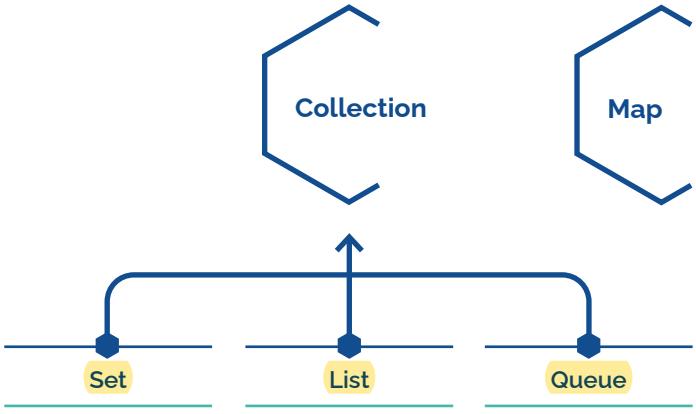

Imagen 1. Tipos de colecciones en java

El uso de un tipo de colección u otra dependerá si queremos que la colección sea ordenada, su acceso sea secuencial, exista una prioridad, pueda haber valores duplicados, etc.

9.2. Interfaz Collection

La interfaz Collection es la raíz para las colecciones Set, List y Queue.

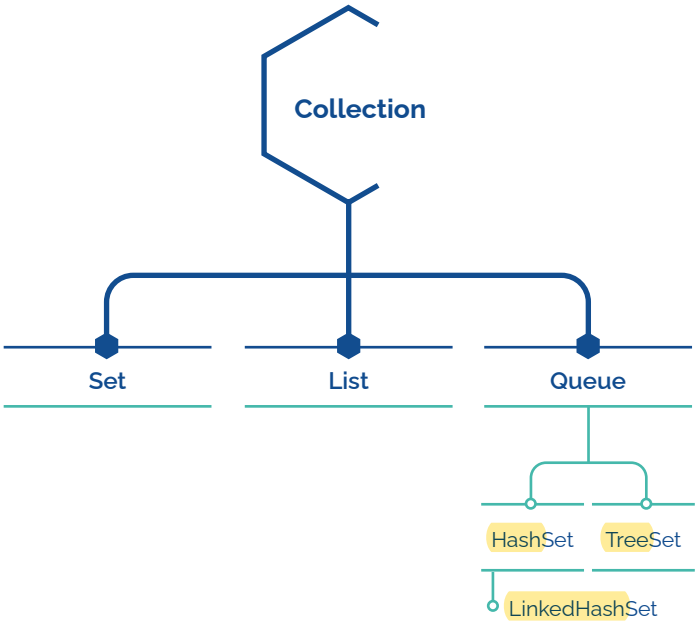

Imagen 2. Tipos de colecciones de la interfaz Collection en java

Collection hereda de la interfaz Iterable y dando la posibilidad de hacer uso de iteradores para recorrer sus elementos. A continuación, se detallan los métodos principales que facilita la interfaz y pueden hacer uso desde Set, List y Queue:

Principales métodos de la interfaz Collection	
Método	**Descripción**
int size()	Devuelve el número de elementos contenido en la colección.
boolean isEmpty()	Devuelve true si no hay elementos. false si hay elementos
void clear()	Borra todos los elementos
boolean contains(E elemento)	Devuelve true si existe el elemento, false en caso contrario
boalean add(E elemento)	Añade el elemento. Devuelve true si ha sido añadido
boalean remove(E elemento)	Elimina el elemento. Devuelve true si lo ha borrado.
Iterator<E> iterator()	Devuelve un iterador de la colección para recorrerla

9.2.1. Set

Set es una interfaz que usa conjunto de datos con la principal característica que no permite elementos duplicados. Las clases que hacen uso de la interfaz Set son, HashSet, LinkedHashSet y TreeSet.

HashSet

HashSet es una colección que hace uso interno de una tabla hash para almacenar sus elementos. Sus principales características:

- Su acceso es muy rápido.
- Los elementos no están ordenados
- Puede tener elementos null
- No permite accesos secuenciales.

```
HashSet<String> clientes = new  HashSet<>();
clientes.add("Asdaiop S.A");
clientes.add("Juan A. Escaleras");
clientes.add("JindaX S.A.");
clientes.add("Copisteria El Arbol");
clientes.add(null);
clientes.add("TodoFurgo2287 S.L.");
clientes.add("JindaX S.A.");
clientes.add("A. Ay. y cia");

Iterator<String> iterador = clientes.iterator();

System.out.println("Resultado:");

while(iterador.hasNext()) {
    System.out.println(iterador.next());
}
```

```
Resultado:
null
TodoFurgo2287 S.L.
Juan A. Escaleras
JindaX S.A.
A. Ay. y cia
Copisteria El Arbol
Asdaiop S.A
```

Imagen 3. Ejemplo de HashSet

LinkedHashSet

Es similar a HashSet hace uso interno de una tabla hash para almacenar sus elementos, pero se diferencia en que usa enlaces o punteros para guardar la dirección de memoria del anterior y posterior elementos. Sus principales características:

- Los elementos están ordenados según el orden de inserción
- Puede tener elementos null
- Acceso secuencial
- Su rendimiento es inferior a HashSet, aun así, su acceso es rápido.

```java
LinkedHashSet<String> clientes = new  LinkedHashSet<>();
clientes.add("Asdaiop S.A");
clientes.add("Juan A. Escaleras");
clientes.add("JindaX S.A.");
clientes.add("Copisteria El Arbol");
clientes.add(null);
clientes.add("TodoFurgo2287 S.L.");
clientes.add("JindaX S.A.");
clientes.add("A. Ay. y cia");

Iterator<String> iterador = clientes.iterator();

System.out.println("Resultado:");

while(iterador.hasNext()) {
    System.out.println(iterador.next());
}
```

```
Resultado:
Asdaiop S.A
Juan A. Escaleras
JindaX S.A.
Copisteria El Arbol
null
TodoFurgo2287 S.L.
A. Ay. y cia
```

Imagen 4. Ejemplo de LinkedHashSet

TreeSet

Es una estructura en árbol binario solo puede haber dos ramificaciones por cada nodo, permitiendo así tener ordenados sus elementos según un criterio. Sus principales características:

- Los elementos están ordenados ascendentemente según sus valores
- No puede tener elementos null.
- No permite el acceso secuencial
- Su rendimiento es lento.

Al usar TreeSet se pueden usar otros métodos exclusivos. A continuación, se detallan los principales:

Principales métodos de TreeSet	
Método	**Descripción**
E first()	Devuelve el menor elemento.
E last()	Devuelve el mayor elemento.
E floor(E e)	Devuelve el mayor elemento que sea menor o igual al elemento dado, null si no existe.
E ceiling(E e)	Devuelve el menor elemento que sea menor o igual al elemento dado, null si no existe.
E higher(E e)	Devuelve el menor elemento que sea mayor al elemento dado, null si no existe.
E lower(E e)	Devuelve el menor elemento que sea menor al elemento dado, null si no existe.
E pollFirst()	Recupera y eliminar el primer elemento menor
E pollLast()	Recupera y elimina el último elemento mayor

```
TreeSet<String> clientesTree = new TreeSet<>();
clientesTree.add("Asdaiop S.A");
clientesTree.add("Juan A. Escaleras");
clientesTree.add("JindaX S.A.");
clientesTree.add("Copisteria El Arbol");
clientesTree.add("TodoFurgo2287 S.L.");
clientesTree.add("JindaX S.A.");
clientesTree.add("A. Ay. y cia");

System.out.println("Resultado:");
for (String cliente : clientesTree) {
    System.out.println(cliente);
}
```

```
Resultado:
A. Ay. y cia
Asdaiop S.A
Copisteria El Arbol
JindaX S.A.
Juan A. Escaleras
TodoFurgo2287 S.L.
```

Imagen 5. Ejemplo de TreeSet

El criterio de ordenación viene establecido por el tipo del elemento, si son tipos de clases envoltorio, los ordena según orden alfabético, numérico o lexicográfico. Si son tipo clase, habrá que definir el criterio de ordenación usando la clase Comparator

```
public static void main(String[] args) {

    TreeSet<String> clientesTree = new TreeSet<>(new comparador());
    clientesTree.add("Asdaiop S.A");
    clientesTree.add("Juan A. Escaleras");
    clientesTree.add("JindaX S.A.");
    clientesTree.add("Copisteria El Arbol");
    clientesTree.add("TodoFurgo2287 S.L.");
    clientesTree.add("JindaX S.A.");
    clientesTree.add("A. Ay. y cia");

    System.out.println("Resultado:");
    for (String cliente : clientesTree) {
        System.out.println(cliente);
    }

}
static class comparador implements Comparator<String> {
    public int compare(String cadena1, String cadena2) {
        int resultado = cadena1.compareTo(cadena2);
        // Cambiamos a orden descendente
        if(resultado < 0)
            resultado = 1;
        else
            resultado = -1;

        return resultado;
    }

}
```

```
Resultado:
TodoFurgo2287 S.L.
Juan A. Escaleras
JindaX S.A.
JindaX S.A.
Copisteria El Arbol
Asdaiop S.A
A. Ay. y cia
```

Imagen 6. Ejemplo de TreeSet con Comparator

9.2.2. List

List es una interfaz que usa una sucesión de elementos sin necesidad de que sea un conjunto. Sus principales características es que permite elementos duplicados y su acceso puede ser secuencial. Las clases que hacen uso de la interfaz List son, ArrayList y LinkedList.

Haciendo uso de List se puede implementar sistemas de cola (FIFO) y pila (LIFO) dependiendo de cómo se inserte y extraiga elementos. Cola (FIFO First in First out), las extracciones de elementos se harán del principio de la lista y sus inserciones se harán al final de la lista. Pila (LIFO Last in First Out), las extracciones e inserciones se harán desde el final, con lo cual, el último elemento en extraerse siempre será el primero que se insertó.

ArrayList

Es la implementación de un array dinámico haciendo uso de índices y añadiendo la mejora que puede variar su tamaño en ejecución. Sus principales características:

- Su acceso es secuencial
- Puede tener elementos null.
- Los elementos están ordenados según el orden de inserción
- Su rendimiento es muy bueno para el acceso secuencial y aleatorio, en cambio para insertar o eliminar tiene un coste muy alto.

```
ArrayList<String> clientes = new  ArrayList<>();
clientes.add("Asdaiop S.A");
clientes.add("Juan A. Escaleras");
clientes.add("JindaX S.A.");
clientes.add("Copisteria El Arbol");
clientes.add(null);
clientes.add("TodoFurgo2287 S.L.");
clientes.add("JindaX S.A.");
clientes.add("A. Ay. y cia");

System.out.println("Resultado:");
for (String cliente : clientes) {
    System.out.println(cliente);
}
```

```
Resultado:
Asdaiop S.A
Juan A. Escaleras
JindaX S.A.
Copisteria El Arbol
null
TodoFurgo2287 S.L.
JindaX S.A.
A. Ay. y cia
```

Imagen 7. Ejemplo de ArrayList

A continuación, se detallan los métodos principales:

Principales métodos de ArrayList	
Método	**Descripción**
E get(int posición)	Devuelve el elemento según la posición dada
int indexOf(E Elemento)	Devuelve la posición de la primera ocurrencia del elemento
int LastIndexOf(E Elemento)	Devuelve la posición de la última ocurrencia del elemento
E set(int posición, E elemento)	Remplaza el elemento según la posición dada
sort(comparator<? super E> c)	Ordena los elementos según el comparador facilitado.

LinkedList

Es un ArrayList pero mejorando su rendimiento debido a su doble lista enlazada. Guarda la dirección de memoria del siguiente y anterior elemento y facilita su inserción y borrado. Sus principales características:

- Su acceso es secuencial
- Puede tener elementos null.
- Los elementos están ordenados según el orden de inserción
- Su rendimiento es muy bueno para el acceso secuencial y mejora su rendimiento para insertar o eliminar de un ArrayList, en cambio, empeora su rendimiento para acceder a un elemento aleatoriamente, debido a que hay que recorrer toda la lista.

```
LinkedList<String> clientes = new  LinkedList<>();
clientes.add("Asdaiop S.A");
clientes.add("Juan A. Escaleras");
clientes.add("JindaX S.A.");
clientes.add("Copisteria El Arbol");
clientes.add(null);
clientes.add("TodoFurgo2287 S.L.");
clientes.add("JindaX S.A.");
clientes.add("A. Ay. y cia");

System.out.println("Resultado:");
for (String cliente : clientes) {
    System.out.println(cliente);
}
```

```
Resultado:
Asdaiop S.A
Juan A. Escaleras
JindaX S.A.
Copisteria El Arbol
null
TodoFurgo2287 S.L.
JindaX S.A.
A. Ay. y cia
```

Imagen 8. Ejemplo de LinkedList

LinkedList hace uso de los métodos de Collection y ArrayList, además añaden los siguientes:

Principales métodos de LinkedList	
Método	**Descripción**
addFirst (E e)	Inserta el elemento al principio.
push (E e)	Inserta el elemento al principio.
addLast(E e)	Inserta el elemento al final.
boolean offer(E e)	Inserta el elemento al final.
E peek()	Devuelve el elemento del principio
E peekLast()	Devuelve el elemento del final
E poll()	Devuelve y elimina el elemento del principio
E pollLast()	Devuelve y elimina el elemento del final

9.2.3. Queue

La intefaz queue o cola es un símil de la de interfaz List con la implementación FIFO. Sus características permiten tener elemento duplicados y el acceso no es secuencial. Las clases que hacen uso de la interfaz Queue son, PriorityQueue y ArrayDeque.

Sus principales métodos son los indicados anteriormente para LinkedList.

PriorityQueue

Es una cola con un orden de prioridad. La prioridad es marcada por el orden natural del tipo y es posible definir una prioridad propia usando Comparable. Sus principales características:

- Orden no secuencial
- No permite valores nulos
- Los elementos están ordenados según el orden de inserción.

```
PriorityQueue<Integer> turnoClientes = new  PriorityQueue<>();
turnoClientes.add(9);
turnoClientes.add(55);
turnoClientes.add(4);
turnoClientes.add(1);
turnoClientes.add(67);
turnoClientes.add(3);

Integer elemento = turnoClientes.poll();
while(elemento!=null) {
    System.out.println(elemento);
    elemento = turnoClientes.poll();
}
```

```
Resultado:
1
3
4
9
55
67
```

Imagen 9. Ejemplo de PriorityQueue

ArrayDeque

Es un array similar a ArrayList que puede implementar una cola, pila o ambos a la vez, ya que permite añadir por delante y por detrás. Sus principales características:

- Orden secuencial
- No permite valores nulos
- Acceso rápido

```
ArrayDeque<Integer> turnoClientes = new  ArrayDeque<>();
turnoClientes.add(9);
turnoClientes.add(55);
turnoClientes.add(4);
turnoClientes.add(1);
turnoClientes.add(67);
turnoClientes.add(3);

System.out.println("Resultado: ");
Integer elemento = turnoClientes.poll();
while(elemento!=null) {
    System.out.println(elemento);
    elemento = turnoClientes.poll();
}
```

```
Resultado:
9
55
4
1
67
3
```

Imagen 10. Ejemplo de ArrayDeque

9.3. Interfaz Map

Map difiere de las colecciones de la interfaz Collection que hasta ahora se habían visto. Permite agrupar datos y conjuntos teniendo que hacer uso de una clave para cada elemento (Clave-Valor), es decir, va a tener siempre dos tipos, un tipo para representar la clave y otro para el elemento. La clave asocia siempre al elemento y no puede haber claves duplicadas. Las clases que hacen uso de la interfaz Map son, HashMap, LinkedHashMap y TreeMap.

A continuación, se detallan los métodos principales que facilita la interfaz:

Principales métodos de la interfaz Map

Método	Descripción
int size()	Devuelve el número de elementos contenido en el mapa.
boolean isEmpty()	Devuelve true si el mapa esta vacío. false si hay algún par clave-valor.
void clear()	Borra todas los pares clave-valor.
set<K> keySet()	Obtiene todas las claves del mapa.
Collection<V> values()	Obtiene todos los valores del mapa.
boolean containsKey(K clave)	Devuelve true si existe la clave, false en caso contrario.
boolean containsValue(V valor)	Devuelve true si existe el valor, false en caso contrario.
Object get(Object clave)	Obtiene el valor según su clave.
Object put(K clave, V valor)	Añade el par clave-valor. Si el par ya existía devuelve el objeto, si no, devuelve null
Object remove(K clave)	Elimina el par clave-valor según su clave. Y lo devuelve si lo ha podido eliminar.
boolean remove(K clave, V valor)	Elimina el par clave-valor según su clave y valor. Devuelve true si lo ha eliminado.
E replace(K clave, V valor)	Remplaza el valor del par clave-valor. Devuelve el valor anterior si lo remplaza, null, en caso contrario.

9.3.1. HashMap

Es una estructura de mapa o matriz que almacena las claves en una tabla hash. Sus principales características:

- No existe ningún orden
- Permite poner la clave como su valor nulo
- El rendimiento es muy bueno. Es recomendable inicializar el tamaño que tendrá para que el rendimiento no se vea afectado.

```
HashMap<Integer, String> clientes = new  HashMap<Integer, String>();
clientes.put(1,"Asdaiop S.A");
clientes.put(2,"Juan A. Escaleras");
clientes.put(4,"JindaX S.A.");
clientes.put(7,"Copisteria El Arbol");
clientes.put(null, null);
clientes.put(5,"TodoFurgo2287 S.L.");
clientes.put(6, "JindaX S.A.");
clientes.put(3, "A. Ay. y cia");

System.out.println("Resultado:");
for (String cliente : clientes.values()) {
    System.out.println(cliente);
}
```

```
Resultado:
null
Asdaiop S.A
Juan A. Escaleras
A. Ay. y cia
JindaX S.A.
TodoFurgo2287 S.L.
JindaX S.A.
Copisteria El Arbol
```

Imagen 11. Ejemplo de HashMap

9.3.2. LinkedHashMap

Igual a HashMap pero añade los enlaces para apuntar al siguiente y anterior elemento. Sus principales características:

- Mantiene el orden de inserción
- Permite poner la clave como su valor nulo
- Su rendimiento es inferior a HashMap. Facilita la inserción en cualquier momento y no hace falta establecer un tamaño inicial.

```
LinkedHashMap<Integer, String> clientes = new  LinkedHashMap<Integer, String>();
clientes.put(1,"Asdaiop S.A");
clientes.put(2,"Juan A. Escaleras");
clientes.put(4,"JindaX S.A.");
clientes.put(7,"Copisteria El Arbol");
clientes.put(null, null);
clientes.put(5,"TodoFurgo2287 S.L.");
clientes.put(6, "JindaX S.A.");
clientes.put(3, "A. Ay. y cia");

System.out.println("Resultado:");
for (String cliente : clientes.values()) {
    System.out.println(cliente);
}
```

```
Resultado:
Asdaiop S.A
Juan A. Escaleras
JindaX S.A.
Copisteria El Arbol
null
TodoFurgo2287 S.L.
JindaX S.A.
A. Ay. y cia
```

Imagen 12. Ejemplo de LinkedHashMap

9.3.3. TreeMap

Realiza un mapa con una estructura en árbol, ordenado ascendentemente por las claves del conjunto. Se puede establecer un criterio diferente de ordenación mediante el uso de Comparable. Sus principales características:

- Realiza un orden por la clave.
- No permite poner la clave a null, en cambio sus valores sí.
- Tiene un peor rendimiento, debido a la ordenación que se realiza por cada inserción. En cambio, las búsquedas aleatorias son las más rápidas.

```
TreeMap<Integer, String> clientesTreeMap = new  TreeMap<Integer, String>();
clientesTreeMap.put(0,"Asdaiop S.A");
clientesTreeMap.put(1,"Juan A. Escaleras");
clientesTreeMap.put(3,"JindaX S.A.");
clientesTreeMap.put(4,"Copisteria El Arbol");
clientesTreeMap.put(2,null);
clientesTreeMap.put(5,"TodoFurgo2287 S.L.");
clientesTreeMap.put(7,"JindaX S.A.");
clientesTreeMap.put(6,"A. Ay. y cia");

System.out.println("Resultado:");
for (int i = 0; i<clientesTreeMap.size();i++) {
    System.out.println(clientesTreeMap.get(i));
}
```

```
Resultado:
Asdaiop S.A
Juan A. Escaleras
null
JindaX S.A.
Copisteria El Arbol
TodoFurgo2287 S.L.
A. Ay. y cia
JindaX S.A.
```

Imagen 13. Ejemplo de TreeMap

A continuación, se detallan los métodos principales:

K,V -> Clave/ Valor

Principales métodos de TreeMap	
Método	**Descripción**
Map.Entry<K,V> firstEntry()	Devuelve el par clave-valor asociado a la menor clave
Map.Entry<K,V> LastEntry()	Devuelve el par clave-valor asociado a la mayor clave
K firstKey()	Devuelve la menor clave
K lastKey()	Devuelve la mayor clave
Map.Entry<K,V> higherEntry(K clave	Devuelve el par clave-valor asociado a la menor clave de todas aquellas claves que son mayores o igual a la pasada por argumento
Map.Entry<K,V> floorEntry(K clave)	Devuelve el par clave-valor asociado a la mayor clave de todas aquellas claves que son menores o igual a la pasada por argumento

www.universae.com