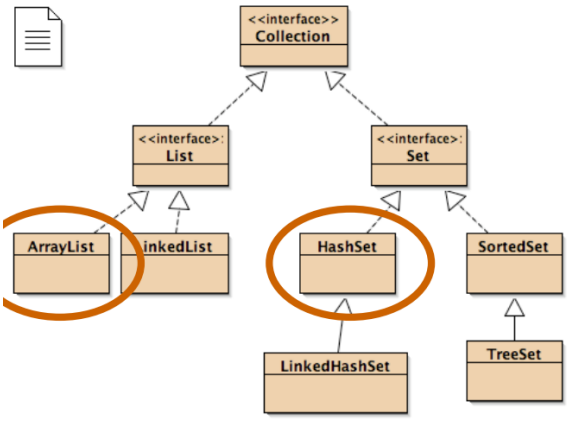
José Enrique Estrada Palomares **Colecciones** 5-D

**Introducción:**

En este reporte vamos a profundizar en las colecciones, se sabe que existen en todos los lenguajes de programación, pero nos basaremos en Java. Vamos a ver qué son y los distintos tipos de colecciones más usados que existen (Set, List y Map, Vector). También, vamos a ver que cada uno de los distintos tipos de colecciones puede tener, además, distintas implementaciones, lo que ofrece funcionalidad distinta.

**Desarrollo:**

**¿Qué es una Colección en Java?**



*Jerarquía de Colecciones*

Pero comenzando, ¿Que es una colección en Java?, básicamente las colecciones son una representación de un grupo de objetos, y a estos objetos se les conoce como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos, en Java, se emplea la interfaz genérica Collection para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección… Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

**¿Qué tipo de colecciones hay en Java?**

Existen 3 tipos de Colecciones en Java, las *Set, Map,* pero agregaremos los *vectores*, ya que estos también son de importancia.

**Set.**

La interfaz Set define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode.

Dentro de la interfaz Set existen varios tipos de implementaciones realizadas dentro de la plataforma Java. Vamos a analizar cada una de ellas:

- HashSet: esta implementación almacena los elementos en una tabla hash. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- TreeSet: esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento que HashSet. Los elementos almacenados deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- LinkedHashSet: esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que HashSet.

```java
final Set<Integer> hashSet = new HashSet<Integer>(1_000_000);
final Long startHashSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    hashSet.add(i);
}
final Long endHashSetTime = System.currentTimeMillis();
System.out.println("Time spent by HashSet: " + (endHashSetTime - startHashSetTime));

final Set<Integer> treeSet = new TreeSet<Integer>();
final Long startTreeSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    treeSet.add(i);
}
final Long endTreeSetTime = System.currentTimeMillis();
System.out.println("Time spent by TreeSet: " + (endTreeSetTime - startTreeSetTime));

final Set<Integer> linkedHashSet = new LinkedHashSet<Integer>(1_000_000);
final Long startLinkedHashSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    linkedHashSet.add(i);
}
final Long endLinkedHashSetTime = System.currentTimeMillis();
System.out.println("Time spent by LinkedHashSet: " + (endLinkedHashSetTime - startLinkedHashSetTim
e));
```

Este es un ejemple de los tres tipos de set que hay.

**List.**

a interfaz List define una sucesión de elementos. A diferencia de la interfaz Set, la interfaz List sí admite elementos duplicados. A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

- Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
- Búsqueda de elementos: busca un elemento concreto de la lista y devuelve su posición.
- Iteración sobre elementos: mejora el Iterator por defecto.
- Rango de operación: permite realizar ciertas operaciones sobre ragos de elementos dentro de la propia lista.

Dentro de la interfaz List existen varios tipos de implementaciones realizadas dentro de Java, las cuales son las siguientes:

- ArrayList: esta es la implementación típica. Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de las situaciones.

- LinkedList: esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

De esta forma se declara un ArrayList y un LinkedList.

```
final List<Integer> arrayList = new ArrayList<Integer>();
final List<Integer> linkedList = new LinkedList<Integer>();
```

**Map.**

Continuemos con la colección Map; la interfaz Map asocia claves a valores. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.

Dentro de la interfaz Map existen varios tipos de implementaciones realizadas dentro de Java, las cuales son las siguientes:

- HashMap: esta implementación almacena las claves en una tabla hash. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.
- TreeMap: esta implementación almacena las claves ordenándolas en función de sus valores. Es bastante más lento que HashMap. Las claves almacenadas deben implementar la interfaz Comparable. Esta implementación garantiza, siempre, un rendimiento de log(N) en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
- LinkedHashMap: esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.

Ahora veamos como se declaran las diferentes tipos de Map en Java.

```
final Map<Integer, List<String>> hashMap = new HashMap<Integer, List<String>>();
final Map<Integer, List<String>> treeMap = new TreeMap<Integer, List<String>>();
final Map<Integer, List<String>> linkedHashMap = new LinkedHashMap<Integer, List<String>>();
```

**Vector.**

Y por último llegamos a la colección que todos utilizamos cuando aprendemos a programar en cualquier lenguaje de programación, los Vectores o Arrays; un vector es una zona de almacenamiento continuo, que contiene una serie de elementos del mismo tipo.

La forma de crear un vector en java es sencilla, es suficiente con agregar los paréntesis cuadrados, "[ ]", al final del tipo de nuestra variable, como ejemplo vamos a crear un vector de Strings, uno de enteros y otro de Object, se procede de la siguiente manera:

```
String[] vectorDeStrings;
int[] vectorDeEnteros;
Object[] vectorDeObjetos;
```

**Conclusión:**

Java proporciona una serie de estructura muy variadas para almacenar datos. Estas estructuras, ofrecen diversas funcionalidades: ordenación de elementos, mejora de rendimiento, rango de operaciones… Es importante conocer cada una de ellas para saber cuál es la mejor situación para utilizarlas. Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación.

**Referencias:**


https://www.adictosaltrabajo.com/tutoriales/introduccion-a-colecciones-en-java/

http://franvarvil.blogspot.com/2012/04/vectores-parte-i-programacion-en-java.html