Java Collections

Autor: José Guillermo Sandoval Huerta

Fecha: 15 noviembre 2023

Índice

¿Qué son las colecciones?

Las colecciones (del inglés Collections) son un conjunto de interfaces y clases que proporcionan estructuras de datos para almacenar, organizar y manipular grupos de objetos. Son parte del paquete java.util. La principal ventaja de las colecciones en Java es que simplifican la manipulación y gestión de datos en programas, ofreciendo implementaciones de estructuras de datos comunes como list, sets y queue [1].

Historia

La evolución de las colecciones se ha tenido a partir de la liberación de las diferentes versiones del JDK, a continuación, se expone esta evolución.

- **Versiones Iniciales (antes de JDK 1.2)**: En las primeras versiones de Java, las únicas estructuras de datos disponibles eran los arrays, que eran limitados en términos de flexibilidad y funcionalidad.
- **JDK 1.2 (lanzado en 1998)**: La versión 1.2 de Java introdujo el paquete java.util, que contenía las primeras implementaciones de las interfaces de colecciones, como Collection, List, Set, y Map. Esta versión presentó clases como ArrayList, LinkedList, HashSet, y HashMap.
- **JDK 1.5 (lanzado en 2004)**: También conocido como Java 5 o Java 5.0, esta versión introdujo mejoras significativas en el sistema de tipo de Java y trajo consigo la introducción del framework de colecciones mejorado conocido como "Generics". Las colecciones ahora podían ser parametrizadas con tipos específicos, lo que mejoraba la seguridad y la legibilidad del código.
- **JDK 1.6 y 1.7 (lanzados en 2006 y 2011, respectivamente)**: Estas versiones incluyeron mejoras adicionales en términos de rendimiento y funcionalidad para las colecciones en Java.
- **JDK 1.8 (lanzado en 2014)**: La versión 8 de Java trajo consigo la introducción de expresiones lambda y el paquete java.util.stream, que proporciona operaciones de transmisión para procesar colecciones de manera más concisa y funcional.
- **JDK 9 (lanzado en 2017)**: Esta versión no introdujo cambios significativos en el marco de colecciones, pero hubo mejoras en términos de eficiencia y funcionalidad.
- **JDK 10, 11, y posteriores**: Las versiones más recientes de Java han continuado introduciendo mejoras y optimizaciones en el lenguaje y la plataforma, pero no han realizado cambios radicales en el marco de colecciones.

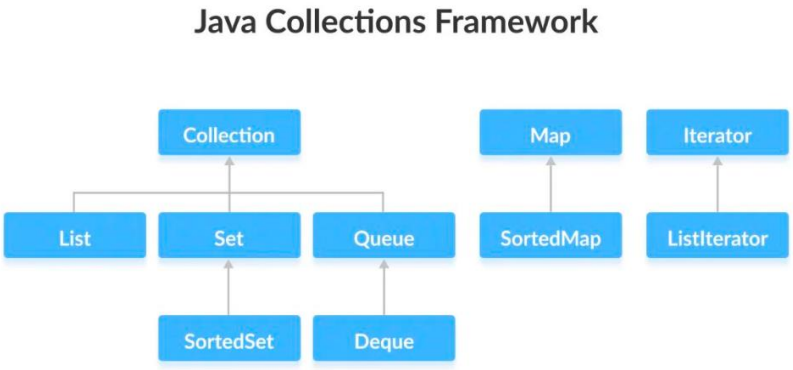

Imagen 1. Estructura y composición de los Collection, Map e Iterator.

List

Del inglés “List”, está representa una **secuencia ordenada de elementos** permitiendo duplicados y mantienen el orden de inserción de los elementos. La interfaz List proporciona métodos adicionales específicos para trabajar con índices y elementos en una secuencia [2].

Entre las implementaciones más comunes encontramos:

ArrayList

Un ArrayList maneja una implementación respaldada por un array dinámico que crece automáticamente según sea necesario. Ofrece un acceso rápido a los elementos mediante índices y mantiene un buen rendimiento en acceso y búsqueda, pero puede ser menos eficiente en inserciones y eliminaciones en el medio de la lista.

Se emplea de la siguiente forma:

List<String> arrayList = new ArrayList<>();

Código 1. Ejemplo ArrayList

```
import java.util.ArrayList;
import java.util.List;

public class EjemploColeccion {

    public static void main(String[] args) {
        // Crear una lista de cadenas de caracteres
        List<String> miLista = new ArrayList<>();

        // Agregar elementos a la lista
        miLista.add("Elemento 1");
        miLista.add("Elemento 2");
        miLista.add("Elemento 3");

        // Mostrar los elementos de la lista
        System.out.println("Elementos de la lista:");
        for (String elemento : miLista) {
            System.out.println(elemento);
        }
    }
}
```

LinkedList

Un LinkedList (o lista enlazada) es una estructura de datos que organiza y almacena elementos de manera secuencial. A diferencia de un array, donde los elementos están almacenados en ubicaciones de memoria contiguas, en una lista enlazada, cada elemento (llamado nodo) contiene un valor y una referencia al siguiente nodo en la secuencia. La última referencia de la lista enlazada generalmente apunta a un valor nulo para indicar el final. Pero el acceso a los elementos puede ser más lento que en ArrayList.

List<String> linkedList = new LinkedList<>();

Vector

Un Vector es una clase que implementa una lista dinámica que puede crecer o decrecer automáticamente según sea necesario. Aunque proporciona operaciones de lista básicas, su uso ha sido en gran medida reemplazado por las implementaciones más modernas de la interfaz List, como ArrayList y LinkedList, que son más eficientes y flexibles en muchos casos.

List<String> vector = new Vector<>();

Stack

La clase Stack extiende la clase Vector con métodos que permiten un acceso fácil a las operaciones de pila. Sin embargo, cabe señalar que, según las prácticas recomendadas, se prefiere el uso de la interfaz Deque (extiende la interfaz Queue) y sus implementaciones, como ArrayDeque, sobre Stack debido a mejoras en la concurrencia y en términos de rendimiento.

Stack<String> stack = new Stack<>();

Set

La interfaz Set representa una colección de elementos únicos, en otras palabras, una colección en la que no se permiten elementos duplicados. La interfaz Set **no garantiza ningún orden específico de los elementos**. Proporciona métodos específicos para la manipulación de conjuntos, como add para agregar elementos, remove para eliminar elementos, contains para verificar la presencia de un elemento, entre otros [3].

Algunas de las implementaciones son:

HashSet

La principal característica de HashSet es que garantiza la unicidad de elementos, es decir, no permite elementos duplicados. Además, HashSet no garantiza ningún orden específico de los elementos.

Set<String> hashSet = new HashSet<>();

Código 2. Ejemplo de Set

```
import java.util.HashSet;
import java.util.Set;

public class EjemploSet {

    public static void main(String[] args) {
        // Crear un conjunto de cadenas de caracteres
        Set<String> miSet = new HashSet<>();
```

```
        // Agregar elementos al conjunto
        miSet.add("Elemento 1");
        miSet.add("Elemento 2");
        miSet.add("Elemento 3");
        miSet.add("Elemento 2"); // No se permiten duplicados, esta adición será ignorada

        // Mostrar los elementos del conjunto
        System.out.println("Elementos del conjunto:");
        for (String elemento : miSet) {
            System.out.println(elemento);
        }
    }
}
```

TreeSet

Esta implementación mantiene los elementos ordenados según su orden natural o según un comparador proporcionado al construir el TreeSet. Garantiza un orden ascendente.

Set<String> treeSet = new TreeSet<>();

LinkedHashSet

Esta implementación es similar a HashSet, pero mantiene un orden de inserción de los elementos. Es decir, cuando iteras sobre un LinkedHashSet, se obtrendrá los elementos en el orden en que fueron insertados.

Set<String> linkedHashSet = new LinkedHashSet<>();

SortedSet

La interfaz SortedSet es una subinterfaz de la interfaz Set en Java. Un conjunto ordenado garantiza que los elementos se almacenen y se recuperen en un orden específico. Al igual que Set, SortedSet no permite duplicados.

SortedSet<String> sortedSet = new TreeSet<>();

Queue

En Java, un Queue representa una cola, una estructura de datos que sigue el principio de "primero en entrar, primero en salir" (FIFO). La interfaz Queue extiende la interfaz Collection y proporciona métodos específicos para operaciones de cola, como la inserción de elementos al final de la cola y la eliminación de elementos desde el principio de la cola [4].

Las operaciones más comunes por Queue incluyen:

- **add(E elemento):** Agrega un elemento al final de la cola.

- **offer(E elemento):** Intenta agregar un elemento al final de la cola. Retorna true si la operación tuvo éxito, false si no hay espacio suficiente, etc.
- **remove():** Elimina y retorna el elemento en la cabeza de la cola. Lanza una excepción si la cola está vacía.
- **poll():** Intenta eliminar y retornar el elemento en la cabeza de la cola. Retorna null si la cola está vacía.
- **element():** Retorna, pero no elimina, el elemento en la cabeza de la cola. Lanza una excepción si la cola está vacía.
- **peek():** Retorna, pero no elimina, el elemento en la cabeza de la cola. Retorna null si la cola está vacía.

Entre las implementaciones están:

LinkedList

La clase LinkedList implementa la interfaz Queue y proporciona una implementación de cola basada en listas enlazadas.

Queue<String> cola = new LinkedList<>();

Código 3. Ejemplo Queue

```
import java.util.LinkedList;
import java.util.Queue;

public class EjemploQueue {

    public static void main(String[] args) {
        // Crear una cola de enteros
        Queue<Integer> miCola = new LinkedList<>();

        // Agregar elementos a la cola
        miCola.offer(1);
        miCola.offer(2);
        miCola.offer(3);

        // Mostrar los elementos de la cola
        System.out.println("Elementos de la cola:");
        for (Integer elemento : miCola) {
            System.out.println(elemento);
        }
    }
}
```

PriorityQueue

Esta implementación de cola utiliza un ordenamiento basado en prioridades para ordenar los elementos. El elemento con la mayor prioridad será el primero en ser retirado.

Queue<String> colaPrioridad = new PriorityQueue<>();

¿Qué son los Map?

La interfaz Map representa una colección de pares clave-valor, donde cada clave está asociada con exactamente un valor. Esto significa que cada clave en un Map es única, y puedes usar la clave para recuperar su valor correspondiente [5].

Algunas de las operaciones más comunes proporcionadas por la interfaz Map incluyen:

- **put(K clave, V valor):** Asocia el valor especificado con la clave especificada en el mapa. Si la clave ya está presente, el valor anterior asociado con esa clave se reemplaza.
- **get(Object clave):** Retorna el valor asociado con la clave especificada, o null si la clave no está presente en el mapa.
- **containsKey(Object clave):** Retorna true si el mapa contiene una asignación para la clave especificada.
- **containsValue(Object valor):** Retorna true si el mapa contiene al menos una clave asociada con el valor especificado.
- **remove(Object clave):** Elimina la asignación para la clave especificada, si está presente.
- **keySet():** Retorna un conjunto de todas las claves presentes en el mapa.
- **values():** Retorna una colección de todos los valores presentes en el mapa.
- **entrySet():** Retorna un conjunto de pares clave-valor (entradas) presentes en el mapa.

Entre las implementaciones de la interfaz Map, se tiene:

HashMap

Esta implementación utiliza una tabla de dispersión para almacenar las entradas del mapa, ofreciendo un rendimiento constante en términos de tiempo promedio para operaciones básicas como put y get.

Map<String, Integer> hashMap = new HashMap<>();

Código 4. Ejemplo Map

```
import java.util.HashMap;
import java.util.Map;

public class EjemploMap {

    public static void main(String[] args) {
        // Crear un mapa de cadenas de caracteres a enteros
        Map<String, Integer> miMapa = new HashMap<>();
```

```
        // Agregar pares clave-valor al mapa
        miMapa.put("Clave1", 10);
        miMapa.put("Clave2", 20);
        miMapa.put("Clave3", 30);

        // Mostrar los pares clave-valor del mapa
        System.out.println("Pares clave-valor del mapa:");
        for (Map.Entry<String, Integer> entry : miMapa.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

TreeMap

Esta implementación mantiene las entradas ordenadas según el orden natural de las claves o un comparador proporcionado.

Map<String, Integer> treeMap = new TreeMap<>();

LinkedHashMap

Similar a HashMap, pero mantiene el orden de inserción de las entradas.

Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

Hashtable

Una implementación antigua que es similar a HashMap, pero es sincronizada, lo que significa que es segura para operaciones concurrentes. Sin embargo, se recomienda el uso de HashMap junto con mecanismos de sincronización externos para operaciones concurrentes en versiones más recientes de Java.

Map<String, Integer> hashtable = new Hashtable<>();

SortedMap

La interfaz SortedMap representa un mapa cuyas claves están ordenadas. Cada clave en un SortedMap se asocia con un valor, y las claves se ordenan según su orden natural o según un comparador proporcionado durante la creación del mapa.

Entre sus implementaciones están:

- **TreeMap:** Esta implementación de SortedMap utiliza un árbol (generalmente un árbol rojo-negro) para almacenar las entradas del mapa, manteniendo las claves ordenadas.

  SortedMap<String, Integer> treeMap = new TreeMap<>();

- **ConcurrentSkipListMap:** Esta implementación proporciona una versión concurrente y basada en listas de salto de SortedMap.

```
SortedMap<String, Integer> skipListMap = new ConcurrentSkipListMap<>();
```

¿Qué son los Iterator?

Los Iterator es una interfaz que se utiliza para recorrer elementos en una colección. Permite acceder a los elementos de una colección de manera secuencial, sin exponer la estructura interna de la colección.

La interfaz Iterator define tres métodos principales:

- **boolean hasNext():** Retorna true si hay más elementos en la colección que aún no se han iterado, y false si no hay más elementos disponibles.

- **E next():** Retorna el próximo elemento de la colección y avanza el cursor al siguiente elemento. Si no hay más elementos, lanza una excepción NoSuchElementException.

- **void remove():** Remueve el último elemento retornado por next() de la colección. No todos los iteradores soportan esta operación, y si no es soportada, lanzará una excepción UnsupportedOperationException.

La mayoría de las colecciones en Java, como List, Set y Map, implementan la interfaz Iterable, lo que significa que pueden proporcionar un iterador a través del método iterator().

Código 5. Ejemplo Iterator.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class EjemploIterator {

    public static void main(String[] args) {
        // Crear una lista de cadenas de caracteres
        List<String> miLista = new ArrayList<>();

        // Agregar elementos a la lista
        miLista.add("Elemento 1");
        miLista.add("Elemento 2");
        miLista.add("Elemento 3");

        // Obtener un Iterator para la lista
        Iterator<String> iterador = miLista.iterator();

        // Recorrer la lista usando el Iterator
        System.out.println("Recorriendo la lista con Iterator:");
        while (iterador.hasNext()) {
            String elemento = iterador.next();
            System.out.println(elemento);
        }
    }
}
```

Referencias

[1] C. Álvarez, «Java Collections Framework y su estructura,» 22 Enero 2022. [En línea]. Available: https://www.arquitecturajava.com/java-collections-framework-y-su-estructura/#:~:text=Un%20collection%20es%20un%20conjunto,cualquier%20posici%C3%B3n%20de%20la%20lista.. [Último acceso: 15 Noviembre 2023].

[2] Y. Kemal, «Tutorial de métodos de lista de Java: ejemplo de API de lista de utilidades,» freeCodeCamp, 21 Febrero 2023. [En línea]. Available: https://www.freecodecamp.org/espanol/news/metodos-de-lista-de-java/. [Último acceso: 15 Noviembre 2023].

[3] C. Álvarez, «Java Collections List vs Set,» 17 Febrero 2015. [En línea]. Available: https://www.arquitecturajava.com/java-collections-list-vs-set/. [Último acceso: 15 Noviembre 2023].

[4] «72 - Colecciones: Queue y PriorityQueue,» [En línea]. Available: https://www.tutorialesprogramacionya.com/javaya/detalleconcepto.php?punto=72&codigo=150&inicio=60. [Último acceso: 15 Noviembre 2023].

[5] O. Blancarte, «Explicando la función map de las colecciones Java,» Codmid, 10 Diciembre 2022. [En línea]. Available: https://blog.codmind.com/explicando-la-funcion-map-de-las-colecciones-java/#:~:text=1%20min%20read-,La%20funci%C3%B3n%20map%20de%20Java%20permite%20aplicar%20una%20funci%C3%B3n%20dada,de%20datos%20en%20una%20colecci%C3%B3n.. [Último acceso: 15 Noviembre 2023].