APUNTE DE COLECCIONES EN JAVA

Árbol y Jerarquía de Colecciones en Java

En Java, las colecciones son estructuras de datos diseñadas para almacenar y manipular grupos de objetos. La plataforma Java proporciona un conjunto robusto de clases e interfaces para este propósito, todas agrupadas en el paquete

java.util. La jerarquía de colecciones se basa principalmente en interfaces que definen el comportamiento, y clases que proporcionan implementaciones concretas de esas interfaces.

I. Interfaces de Colecciones (Comportamiento)

Estas interfaces establecen el contrato para las diferentes categorías de colecciones.

- **Collection**: Es la interfaz raíz en la jerarquía de colecciones. Define los métodos comunes que todas las colecciones deben implementar, como

add(), remove(), size(), isEmpty(), contains(), etc.. No se puede instanciar directamente.

  - **List**: Hereda de Collection. Representa una colección **ordenada** (secuencia) que permite **elementos duplicados**. Los elementos en una

    List tienen un índice.

    - Implementaciones comunes:
      - ArrayList
      - LinkedList

  - **Set**: Hereda de Collection. Representa una colección que **no permite elementos duplicados**. El orden de los elementos no está garantizado, a menos que se use una implementación específica.
    - **SortedSet**: Extiende Set. Mantiene sus elementos **ordenados**.
      - **NavigableSet**: Extiende SortedSet. Proporciona métodos adicionales para navegar por los elementos, como lower(), floor(), ceiling(), higher().
    - Implementaciones comunes:
      - HashSet (No mantiene un orden específico)
      - TreeSet (Implementa SortedSet y NavigableSet, mantiene los elementos ordenados)
      - LinkedHashSet (Mantiene el orden de inserción)

- **Queue**: Hereda de Collection. Diseñada para contener elementos antes de ser procesados, siguiendo generalmente un principio FIFO (First-In, First-Out).
    - **Deque (Double Ended Queue)**: Extiende Queue. Permite la inserción y eliminación de elementos en **ambos extremos**.
    - Implementaciones comunes:
        - PriorityQueue (Implementa Queue, basada en un *heap* para prioridad)
        - ArrayDeque (Implementa Deque, una cola de doble extremo basada en un *array* redimensionable)

- **Map**: **No extiende de Collection** , pero es una parte fundamental del *framework* de colecciones. Representa un objeto que **mapea claves a valores**. No permite **claves duplicadas**; si se inserta una clave existente, el valor asociado se actualiza.
    - **SortedMap**: Extiende Map. Mantiene sus entradas **ordenadas por la clave**.
        - **NavigableMap**: Extiende SortedMap. Proporciona métodos adicionales para navegar por las entradas del mapa.
    - Implementaciones comunes:
        - HashMap (No mantiene un orden específico)
        - TreeMap (Implementa SortedMap y NavigableMap, mantiene las entradas ordenadas por clave)
        - LinkedHashMap (Mantiene un orden de inserción o de acceso)

II. Clases de Colecciones (Implementaciones Concretas)

Estas clases son las implementaciones tangibles de las interfaces mencionadas anteriormente.

- **ArrayList**: Implementa la interfaz List. Es una lista redimensionable basada en un *array*. Ideal para acceso por índice rápido.
- **LinkedList**: Implementa las interfaces List y Deque. Es una lista doblemente enlazada. Eficiente para inserciones y eliminaciones en los extremos o el medio.
- **HashSet**: Implementa la interfaz Set. Utiliza una tabla *hash* para almacenar elementos. No garantiza un orden.
- **TreeSet**: Implementa las interfaces SortedSet y NavigableSet. Utiliza un árbol rojo-negro para almacenar elementos, manteniéndolos ordenados.

- **LinkedHashSet**: Implementa la interfaz Set. Mantiene el orden de inserción de los elementos utilizando una tabla *hash* y una lista enlazada.
- **PriorityQueue**: Implementa la interfaz Queue. Proporciona una cola con prioridad basada en un *heap*.
- **ArrayDeque**: Implementa la interfaz Deque. Proporciona una cola de doble extremo basada en un *array* redimensionable.
- **HashMap**: Implementa la interfaz Map. Utiliza una tabla *hash* para almacenar entradas (pares clave-valor). No garantiza un orden.
- **TreeMap**: Implementa las interfaces SortedMap y NavigableMap. Utiliza un árbol rojo-negro para almacenar entradas, manteniéndolas ordenadas por la clave.
- **LinkedHashMap**: Implementa la interfaz Map. Mantiene un orden de inserción o de acceso utilizando una tabla *hash* y una lista enlazada.

III. Otras Clases e Interfaces Utilitarias

Además de las colecciones principales, Java proporciona herramientas para interactuar con ellas.

- **Iterator**: Una interfaz fundamental para recorrer los elementos de cualquier Collection.
- **ListIterator**: Una interfaz que extiende Iterator, específica para recorrer los elementos de una List en ambas direcciones.
- **Collections**: Una clase de utilidad que proporciona métodos estáticos para operar en colecciones, como ordenar, buscar o sincronizar.
- **Arrays**: Una clase de utilidad que proporciona métodos estáticos para operar directamente en *arrays*.

Esta estructura facilita la comprensión de cómo se organizan y se relacionan las diferentes herramientas de colecciones en Java, permitiendo elegir la implementación más adecuada para cada necesidad de almacenamiento y manipulación de datos. Estas interfaces y clases proporcionan una amplia gama de funcionalidades para manejar diferentes tipos de datos y escenarios, lo que las convierte en una parte esencial de la programación en Java.

Ejemplos Prácticos de Colecciones

A continuación, se presentan ejemplos de código para HashSet y HashMap, dos de las colecciones más utilizadas en Java.

Ejemplo de HashSet

Un HashSet es una colección que no permite elementos duplicados y no garantiza un orden específico de los elementos.

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

public class EjemploHashSet { //
    public static void main(String[] args) { //
        // Crear un nuevo HashSet
        Set<String> set = new HashSet<>();

        // Agregar elementos al HashSet
        set.add("Manzana"); //
        set.add("Banana"); //
        set.add("Cereza"); //
        set.add("Manzana"); // Esto no se agregará porque es un duplicado

        // Imprimir el HashSet
        System.out.println("HashSet: " + set); //

        // Verificar si un elemento está presente en el HashSet
        boolean contieneBanana = set.contains("Banana"); //
        System.out.println("Contiene 'Banana': " + contieneBanana); //

        // Eliminar un elemento del HashSet
        boolean eliminado = set.remove("Cereza"); //
        System.out.println("Eliminado 'Cereza': " + eliminado);

        // Imprimir el HashSet después de la eliminación
```

```
        System.out.println("HashSet después de la eliminación: " + set);

        // Obtener el tamaño del HashSet
        int tamaño = set.size();
        System.out.println("Tamaño del HashSet: " + tamaño);

        // Iterar sobre los elementos del HashSet [cite: 101]
        System.out.println("Elementos del HashSet:"); //
        for (String elemento : set) { //
            System.out.println(elemento); //
        }

        // Limpiar el HashSet
        set.clear(); //
        System.out.println("HashSet después de limpiar: " + set);
    }
}
```

Explicación del Código HashSet:

- **Creación del HashSet**: Se instancia un nuevo HashSet para almacenar elementos de tipo String.
- **Agregar elementos**: Se añaden varios elementos, notando que un duplicado ("Manzana") no se agrega, ya que los Set no permiten elementos repetidos.
- **Imprimir el HashSet**: Muestra los elementos actualmente en el conjunto.
- **Verificar si contiene un elemento**: Se utiliza contains() para comprobar la existencia de un elemento específico.
- **Eliminar un elemento**: remove() quita un elemento del conjunto.
- **Obtener el tamaño**: size() retorna el número de elementos en el HashSet.
- **Iterar sobre los elementos**: Un bucle *for-each* permite recorrer e imprimir cada elemento del conjunto.
- **Limpiar el HashSet**: clear() elimina todos los elementos del conjunto.

Este ejemplo demuestra las operaciones básicas de un HashSet en Java.

Ejemplo de HashMap

Un HashMap es una colección que almacena pares clave-valor y no permite claves duplicadas, actualizando el valor si se intenta insertar una clave ya existente.

```
import java.util.HashMap; // cite:
import java.util.Map; //
public class EjemploHashMap { //
    public static void main(String[] args) { //
        // Crear un nuevo HashMap
        Map<String, Integer> map = new HashMap<>();

        // Agregar pares clave-valor al HashMap
        map.put("Manzana", 10);
        map.put("Banana", 5);
        map.put("Cereza", 20);
        map.put("Manzana", 15); // Esto actualizará el valor de "Manzana"
        // Imprimir el HashMap
        System.out.println("HashMap: " + map);

        // Obtener el valor asociado a una clave
        int cantidadManzana = map.get("Manzana");
        System.out.println("Cantidad de Manzanas: " + cantidadManzana); //

        // Verificar si una clave está presente en el HashMap
        boolean contieneBanana = map.containsKey("Banana"); //
        System.out.println("Contiene 'Banana': " + contieneBanana); //

        // Eliminar un par clave-valor del HashMap
        Integer eliminado = map.remove("Cereza"); //
```

```
        System.out.println("Eliminado 'Cereza': " + eliminado); //

        // Imprimir el HashMap después de la eliminación
        System.out.println("HashMap después de la eliminación: " + map); //

        // Obtener el tamaño del HashMap
        int tamaño = map.size(); //
        System.out.println("Tamaño del HashMap: " + tamaño); //

        // Iterar sobre los pares clave-valor del HashMap
        System.out.println("Pares clave-valor del HashMap:"); //
        for (Map.Entry<String, Integer> entry : map.entrySet()) { //
            System.out.println("Clave: " + entry.getKey() + ", Valor: " + entry.getValue());
//
        }

        // Limpiar el HashMap
        map.clear(); //
        System.out.println("HashMap después de limpiar: " + map); //
    }
}
```

Explicación del Código HashMap:

- **Creación del HashMap**: Se inicializa un nuevo HashMap para almacenar pares de clave (String) y valor (Integer).
- **Agregar pares clave-valor**: Se añaden elementos usando put(). Si se inserta una clave existente, el valor asociado se actualiza.
- **Imprimir el HashMap**: Muestra el contenido actual del mapa.
- **Obtener el valor asociado a una clave**: get() recupera el valor de una clave específica.
- **Verificar si contiene una clave**: containsKey() verifica la presencia de una clave.

- **Eliminar un par clave-valor**: remove() elimina la entrada asociada a una clave.
- **Obtener el tamaño**: size() devuelve el número de pares clave-valor en el mapa.
- **Iterar sobre los pares clave-valor**: entrySet() permite recorrer cada par de clave-valor en el mapa.
- **Limpiar el HashMap**: clear() vacía completamente el mapa.

Este ejemplo ilustra las operaciones comunes que se pueden realizar con un HashMap en Java.