{desafío} latam_

Colecciones y API - Parte I

Iteradores

Objetivos

- Conocer iteradores para recuperar elementos de colecciones y sus métodos
- Iterar elementos de colecciones con cada iterador
- Diferenciar entre los métodos que ofrecen distintos los iteradores

Introducción

Los iteradores se utilizan sobre Colecciones en Java para recuperar elementos uno por uno. Pero ¿qué son las colecciones?. Son simplemente objetos que agrupan multiples elementos dentro de una entidad, esta entidad generalmente representa un grupo natural, por ejemplo una mano de cartas que vendría siendo una colección de cartas, o una carpeta de correo que sería una colección de emails. A continuación se mencionarán tres iteradores.

- **Enumeration**
- **Iterator**
- **ListIterator**

Enumeration

Interfaz utilizada para obtener elementos de colecciones heredadas (Vector, Hashtable). Enumeration llamando al método elements() de la clase vector en cualquier objeto vector:

Se inicializa un vector con números del 1 al 9. Luego se iteran con Enumeration.

```
Vector vector = new Vector();
for (int i = 0; i < 10; i++) vector.add(i);
System.out.println(vector);

Enumeration e = vector.elements();
while(e.hasMoreElements()) {
    System.out.print(e.nextElement()+" ");
}
```

Iterator

Es un iterador universal, ya que se puede aplicar a cualquier objeto Collection. Al usar Iterator, se pueden realizar operaciones de lectura y eliminación. Es una versión mejorada de Enumeration con funcionalidad adicional de capacidad de eliminación de un elemento. Iterator debe usarse siempre que queramos enumerar elementos en todas las interfaces implementadas de Collection las cuales veremos mas adelante como Set, List, Queue, Deque y también en todas las clases implementadas de la interfaz Map. Iterator es el único cursor disponible para todo Collection.

El objeto iterator se puede crear llamando al método iterator() presente en la interfaz de Collection.

A continuación se inicializa un ArrayList con números del 1 al 9 y posterior mente se itera con Iterator, luego se eliminan los números impares.

```
ArrayList al = new ArrayList();
for (int i = 0; i < 10; i++) al.add(i);
System.out.println(al);

Iterator it = al.iterator();
while (it.hasNext()) {
    if ((int) it.next() % 2 != 0) it.remove();
}
System.out.println("\n"+al);
```

ListIterator

Solo es aplicable a las clases implementadas de la colección de List como ArrayList, LinkedList, etc. Proporciona iteración bidireccional. ListIterator debe usarse cuando queremos enumerar elementos de List. Este cursor tiene más funcionalidad (métodos) que Iterator. El objeto ListIterator se puede crear llamando al método listIterator() presente en la interfaz List.

A continuación se inicializa un ArrayList con números del 1 al 9 y posterior mente se itera con ListIterator, luego se eliminan los números pares.

```
ArrayList al = new ArrayList();
for (int i = 0; i < 10; i++) al.add(i);
System.out.println(al);

ListIterator lit = al.listIterator();
while (lit.hasNext()) {
    if ((int) lit.next() % 2 == 0) lit.remove();
}
System.out.println("\n"+al);
```

Alcance de las variables locales

También es importante que recordemos que las variables tienen alcance y si las definimos dentro del bloque solo existirán dentro del bloque.

```
ListIterator lit = al.listIterator();
while (lit.hasNext()) {
    int x = (int) lit.next();
}
```

En el caso de los métodos, el alcance de las variables locales tiene límite entre `{` y `}`

Colecciones

Una colección es un grupo de objetos individuales representados como una sola unidad. Java proporciona Framework de colecciones que define varias clases e interfaces para representar un grupo de objetos como una sola unidad.

La interfaz Collection (java.util.Collection) y la interfaz Map (java.util.Map) son las dos principales interfaces "raíz" de las clases de recopilación de Java.

Jerarquía de Collection Framework

Las interfaces de la colección principal encapsulan diferentes tipos de colecciones, que se muestran en la figura a continuación. Estas interfaces permiten manipular las colecciones independientemente de los detalles de su representación. Las interfaces de la colección principal son la base del marco de colecciones de Java. Como puede ver en la siguiente figura, las interfaces de la colección principal forman una jerarquía.

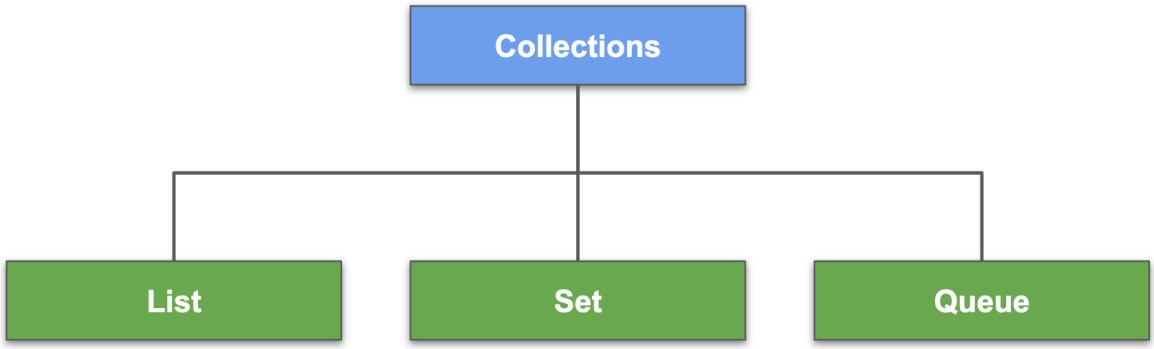

Imagen 1. Jerarquía de colecciones.

Introducción a List

Objetivos

- Conocer la utilidad de una lista
- Crear una lista
- Acceder a elementos dentro de una lista
- Iterar lista

¿Qué es List?

Una lista es una colección ordenada (a veces llamada secuencia). Las listas pueden contener elementos duplicados. La plataforma Java contiene dos implementaciones de Lista de propósito general. ArrayList, que suele ser la implementación con mejor rendimiento, y LinkedList que ofrece un mejor rendimiento en determinadas circunstancias, ambas serán vistas a continuación.

Además de las operaciones heredadas de Collection, la interfaz List incluye operaciones para lo siguiente, las que se verán mas adelante:

- Acceso posicional: manipula elementos en función de su posición numérica en la lista. Esto incluye métodos como get, set, add, addAll y remove.
- Buscar: busca un objeto específico en la lista y devuelve su posición numérica. Los métodos de búsqueda incluyen indexOf y lastIndexOf.
- Iteración: extiende la semántica del iterador para aprovechar la naturaleza secuencial de la lista. Los métodos listIterator proporcionan este comportamiento.

Creando un ArrayList

Implementación de matriz-redimensionable de la interfaz List. Permite todos los elementos, incluido nulo. Se pude ver en la imagen, que implementa List.

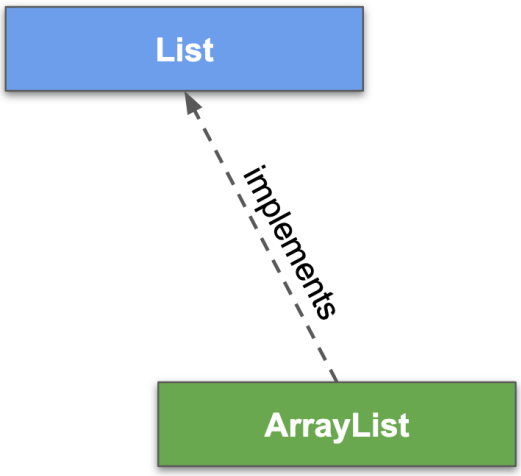

Imagen 2. Implementación de un ArrayList.

Para crear un nuevo ArrayList:

```
List<String> list = new ArrayList<>();
list.add("Java");
list.add("Scala");
list.add("Kotlin");
System.out.println(list); //[Java, Scala, Kotlin]
```

Índices

Los elementos tienen una posición, a esta posición se le llamada índice. Se puede acceder a los valores pasando el índice como parámetro del método get.

Los índices van de cero hasta `n-1`, donde n es la cantidad de elementos de la colección. Si una lista contiene 5 elementos, el primer elemento estará en la posición cero, y el último en la cuarta posición.

```
System.out.println(list.get(0)); //Java
System.out.println(list.get(2)); //Kotlin
```

Índices fuera de borde

En caso de que el índice sea mayor o igual a la cantidad de elementos se obtendrá una excepción ArrayIndexOutOfBoundsException.

Métodos de acesso posicional

List permite agregar, quitar, obtener y establecer operaciones basadas en posiciones numéricas de elementos en la lista. Estos son métodos usados sobre listas:

- **void add(int index, Object o)** : este método agrega un elemento en el índice especificado.
- **addAll boolean(int int, Collection c)**: este método agrega todos los elementos de la colección especificada a la lista. El primer elemento se inserta en el índice dado. Si ya hay un elemento en esa posición, ese elemento y otros elementos posteriores (si los hay) se desplazan hacia la derecha al aumentar su índice.
- **Object remove(int index)**: este método elimina un elemento del índice especificado. Desplaza los elementos posteriores (si los hay) a la izquierda y disminuye sus índices en 1.
- **Object get(int index)**: este método devuelve el elemento en el índice especificado.
- **Object set(int index, Object new)**: este método reemplaza el elemento en un índice dado con un nuevo elemento. Esta función devuelve el elemento que acaba de ser reemplazado por un nuevo elemento.

Métodos de búsqueda

List proporciona métodos para buscar elementos y devuelve su posición numérica. Los siguientes dos métodos son compatibles con List para esta operación:

- **int indexOf(Object o)**: este método devuelve la primera aparición del elemento dado o -1 si el elemento no está presente en la lista.
- **int lastIndexOf(Object o)**: este método devuelve la última aparición del elemento dado o -1 si el elemento no está presente en la lista.

Buscar índice de elemento con .indexOf

Podemos saber si un elemento se encuentra dentro de una lista utilizando el método .indexOf

```
list.indexOf("Kotlin") //2
```

Remover un elemento

El método remove entrega el elemento removido si se le pasa el índice como parámetro o lanza una excepción si no se encuentra. Adicionalmente se puede pasar como parámetro el objeto al método remove y este devolverá true si lo eliminó o false si no lo encontró.

```
list.remove("Kotlisn") //false
list.remove(1) //Scala
```

Ejemplos de iteración con forEach de streams

```
List<String> langs = Arrays.asList("Java", "Scala", "Groovy", "Rust",
"Kotlin");
langs.forEach(System.out::println); //Imprime en pantalla los lenguajes
```

La salida del programa:

```
Java
Scala
Groovy
Rust
Kotlin
```

Transformación con .map

```
List<String> langs = Arrays.asList("Java", "Scala", "Rust");
langs.stream().map(lang -> lang+" Lang").forEach(System.out::println);
```

La salida del programa es:

```
Java Lang
Scala Lang
Rust Lang
```

Filtrando con .filter

```
List<String> langs = Arrays.asList("JVM Java","JVM Scala", "C++", "Rust", "JVM
Kotlin", "C#");
langs.stream().filter(lang ->
lang.startsWith("JVM")).forEach(System.out::println);
```

La salida del programa es:

```
JVM Java
JVM Scala
JVM Kotlin
```

Creando una nueva lista a partir de otra con .collect

El método .map devuelve una secuencia que consta de los resultados de aplicar la función dada a los elementos de esta secuencia. Esta es una operación intermedia, posterior a eso los valores se recolectan y se almacenan en una lista.

```
List<Integer> ns = Arrays.asList(12, 3, 4, 5, 42);
List<Integer> dobles = ns.stream().map(n -> n*2).collect(Collectors.toList());
System.out.println(dobles);
```

Reduciendo con .reduce

```
List<Integer> numeros = Arrays.asList(12, 3, 4, 5, 42);
int suma = numeros.stream().reduce(0, (x,y) -> x+y);
System.out.println(suma); //66
```

O también a través de un método reference en lugar de una función lambda, llamando al método con **::**.

```
List<Integer> numeros = Arrays.asList(12, 3, 4, 5, 42);
int suma = numeros.stream().reduce(0, Integer::sum);
System.out.println(suma); //66
```

Esto realiza una reducción en los elementos de la secuencia llamada numeros, utilizando el valor de identidad proporcionado y una función de acumulación asociativa, y devuelve el valor reducido. Esta es una operación de reducción.

Una operación de reducción (también llamada pliegue) toma una secuencia de elementos de entrada y los combina en un único resultado de resumen mediante la aplicación repetida de una operación de combinación, como encontrar la suma o el máximo de un conjunto de números, o acumular elementos en una lista . Las clases de secuencias tienen múltiples formas de operaciones de reducción generales, llamadas reduce() y collect(), así como múltiples formas de reducción especializadas como sum(), max() o count().

Por supuesto, tales operaciones pueden implementarse fácilmente como simples bucles secuenciales, como en:

```
int suma = 0;
for (int numero : numeros) {
   suma += numero;
}
```

Introducción a Set

Objetivos

- Crear un Set
- Iterar un Set
- Realizar intersecciones entre Sets
- Eliminar datos de un Set en otro

¿Qué es Set?

Set es una interfaz que extiende de Collection. Es una colección desordenada de objetos, en donde a diferencia de List no se pueden almacenar valores duplicados. Set también agrega un contrato más fuerte sobre el comportamiento de las operaciones equals y hashCode, permitiendo que las instancias de Set se comparen significativamente incluso si sus tipos de implementación difieren. Dos instancias de Set son iguales si contienen los mismos elementos.

La plataforma Java contiene tres implementaciones de Set de propósito general: HashSet, TreeSet y LinkedHashSet. Set tiene varios métodos como add, remove, clear, size, etc. para mejorar el uso de esta interfaz.

Creando un HashSet

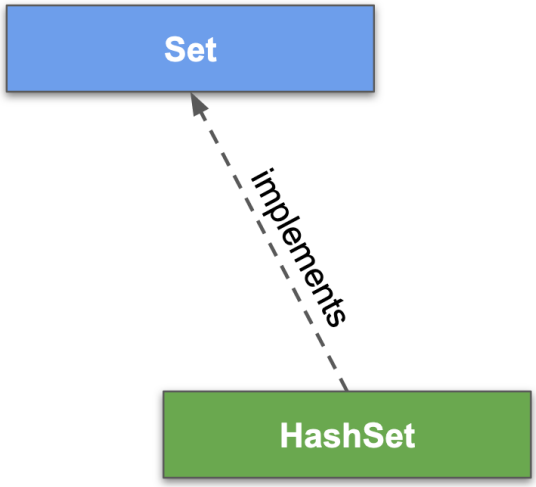

Imagen 3. HashSet implementa Set.

HashSet, almacena sus elementos en una tabla hash, es la implementación con mejor rendimiento; sin embargo, no garantiza el orden de iteración.

```
Set<String> langs = new HashSet<>();
langs.add("Java");
langs.add("Go");
langs.add("Erlang");
langs.add("Java");
langs.add("Elixir");
langs.add("Fortran");
System.out.println(langs); //sin duplicados
```

La salida del programa:

```
[Java, Go, Erlang, Elixir, Fortran]
```

Se ha ingresado "Java" dos veces, pero no se muestra en la salida. Además, se pueden ordenar directamente las entradas pasando el conjunto desordenado como parámetro de TreeSet. A diferencia de un HashSet los elementos del TreeSet van ordenados.

Creando un TreeSet

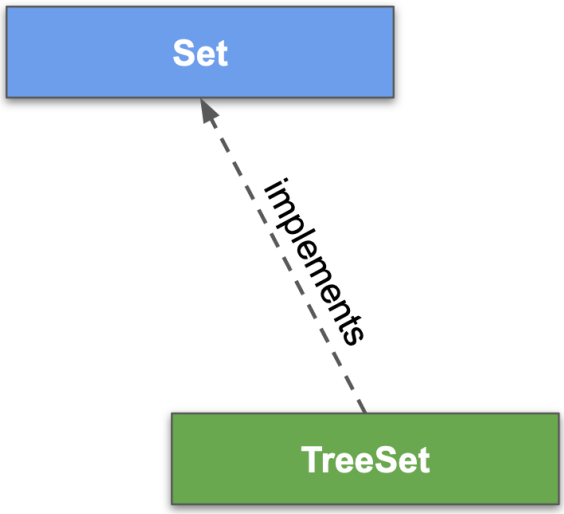

Imagen 4. TreeSet implementa Set.

TreeSet, almacena sus elementos en un árbol rojo-negro, ordena sus elementos en función de sus valores; Es sustancialmente más lento que HashSet.

```
Set<String> sortedLangs = new TreeSet<>(langs);
System.out.println(sortedLangs); //ordenados
```

La salida del programa:

```
[Elixir, Erlang, Fortran, Go, Java]
```

Uniones

Dados estos conjuntos de números:

```
1, 13, 8, 2, 4, 9, 5
1, 3, 7, 5, 14, 0, 7, 5
```

Se unen sin repetidos y ordenados usando un TreeSet

```
Set<Integer> lon = new HashSet<>(Arrays.asList(1, 13, 8, 2, 4, 9, 5));
Set<Integer> lat = new HashSet<>(Arrays.asList(1, 3, 7, 5, 14, 0, 7, 5));
Set<Integer> union = new TreeSet<>(lon);
union.addAll(lat);
System.out.println(union); //[0, 1, 2, 3, 4, 5, 7, 8, 9, 13, 14]
```

Intersecciones

Para encontrar valores en común en ambas colecciones, se usa retainAll.

```
Set<Integer> lon = new HashSet<>(Arrays.asList(1, 13, 8, 2, 4, 9, 5));
Set<Integer> lat = new HashSet<>(Arrays.asList(1, 3, 7, 5, 14, 0, 7, 5));
Set<Integer> intersections = new HashSet<>(lon);
intersections.retainAll(lat);
System.out.println(intersections); //[1, 5]
```

Diferencias dentro de un Set

Para remover todos los valores de un Set, en el otro. Se usa removeAll.

```
Set<Integer> lon = new HashSet<>(Arrays.asList(1, 13, 8, 2, 4, 9, 5));
Set<Integer> lat = new HashSet<>(Arrays.asList(1, 3, 7, 5, 14, 0, 7, 5));
Set<Integer> diff = new HashSet<>(lat);
diff.removeAll(lon);
System.out.println(diff); //[2, 4, 8, 9, 13]
```

Creando un LinkedHashSet

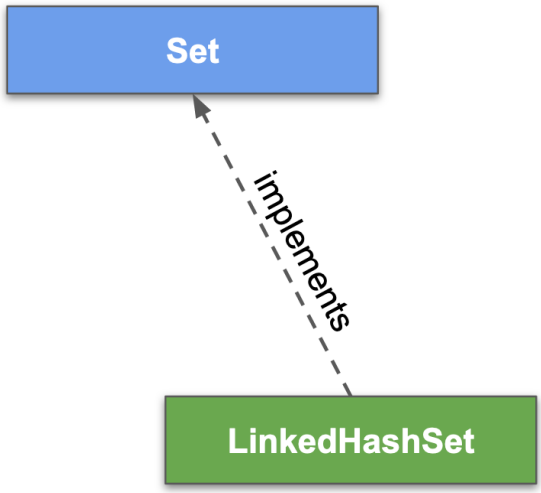

Imagen 5. Implementación de LinkedHashSet.

LinkedHashSet se implementa como una tabla hash con una lista vinculada que lo ejecuta, ordena sus elementos según el orden en que se insertaron en el conjunto (orden de inserción).

```
Set<String> programmers = new LinkedHashSet<>();
programmers.add("James Gosling");
programmers.add("Martin Odersky");
programmers.add("Rich Hickey");
programmers.add("Larry Wall");
programmers.add("Graydon Hoare");
System.out.println(programmers); //[James Gosling, Martin Odersky, Rich Hickey,
Larry Wall, Graydon Hoare]
```

Las iteraciones, transformaciones, filtrado, creación de nuevas colecciones y reducciones (pliegues) usadas con List, siguen siendo aplicadas de la misma forma en Set. Al igual que los métodos vistos en Set como addAll, retainAll y removeAll pueden ser usados en listas.

Introducción a Queue

Objetivos

- Crear un Queue
- Eliminar encabezados en Queues
- Obtener datos de Queue

Una cola es una colección ideal para contener elementos antes del procesamiento. Es una estructura lineal que sigue un orden particular en el que se realizan las operaciones. Un buen ejemplo de una cola es cualquier cola de consumidores para un recurso donde el consumidor que vino primero se sirve primero, como una cola de supermercado.

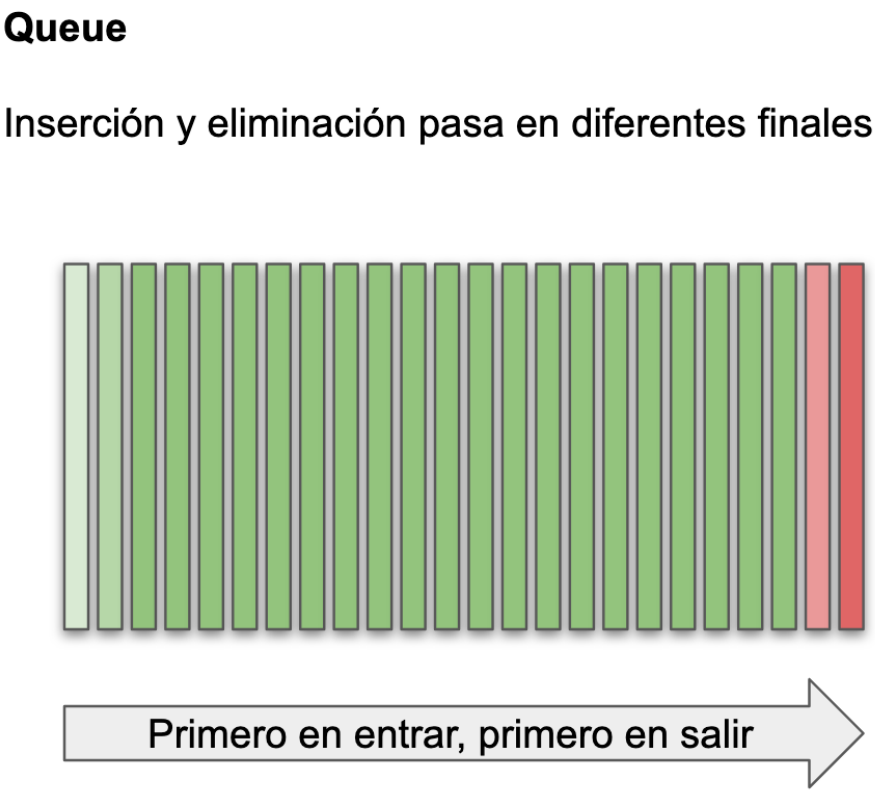

Imagen 6. Primero en entrar, primero en salir.

La colección Queue se utiliza para mantener los elementos a punto de ser procesados y proporciona varias operaciones como la inserción, eliminación, etc. Es una lista ordenada de objetos con su uso limitado para insertar elementos al final de la lista y eliminar elementos desde el principio de lista, es decir, sigue el principio Primero en entrar, primero en salir FIFO (First In First Out). Al ser una interfaz, la cola necesita una clase concreta para la declaración y las clases más comunes son PriorityQueue y LinkedList en Java. Como recomendación cabe señalar que ambas implementaciones no son seguras para subprocesos. Y que PriorityBlockingQueue es una implementación alternativa si se necesitara una implementación segura para subprocesos.

Las colas típicamente, pero no necesariamente, ordenan elementos de manera FIFO (primero en entrar, primero en salir). Entre las excepciones se encuentran las colas de prioridad, que ordenan los elementos de acuerdo con sus valores.

Métodos en Queue

- **add()**: Este método se usa para agregar elementos en la cola de la cola. Más específicamente, al final si se usa LinkedList, o según la prioridad en caso de implementación de PriorityQueue.
- **remove()**: Este método elimina y devuelve el encabezado de la cola. Lanza NoSuchElementException cuando la cola está vacía.
- **poll()**: Este método elimina y devuelve el encabezado de la cola. Devuelve nulo si la cola está vacía.
- **peek()**: Este método se utiliza para ver el encabezado de la cola sin eliminarlo. Devuelve nulo si la cola está vacía.
- **element()**: Este método es similar a peek(). Lanza NoSuchElementException cuando la cola está vacía.
- **size()**: Este método devuelve el número de elementos en la cola.

> Dado que es un subtipo de la clase Collections, hereda todos los métodos, a saber, size(), isEmpty(), contains(), etc.

Creando una cola con Queue

```
Queue tickets = new LinkedList<>();
for (int i = 1; i < 20; i+=3) tickets.add(i);
System.out.println(tickets); //[1, 4, 7, 10, 13, 16, 19]
```

Eliminando encabezado con .remove

En caso de que la cola esté vacía, remove lanza una excepción.

```
Queue tickets = new LinkedList<>();
for (int i = 2; i < 20; i+=3) tickets.add(i); //[2, 5, 8, 11, 14, 17]
System.out.println(tickets.remove()); //2
System.out.println(tickets.remove()); //5
```

Eliminando encabezado .poll

En caso de que la cola esté vacía, poll devuelve null.

```
Queue tickets = new LinkedList<>();
for (int i = 1; i < 3; i++) tickets.add(i); //[1, 2]
System.out.println(tickets.poll()); //1
System.out.println(tickets.poll()); //2
System.out.println(tickets.poll()); //null
```

Obtener encabezado de la cola con .peek

```
Queue tickets = new LinkedList<>();
for (int i = 30; i < 40; i++) tickets.add(i);
System.out.println("peek : "+tickets.peek()); //30
```

Introducción a Map

Objetivos

- Conocer la utilidad de un map.
- Crear un map.
- Conocer los conceptos de llave y valor.
- Acceder a elementos dentro de un map a través de la clave.
- Iterar maps

Map

La interfaz Map representa una asignación entre una clave y un valor. La interfaz de mapa no es un subtipo de Collection. Por lo tanto, se comporta un poco diferente del resto de los tipos de colección.

- Un mapa no puede contener claves duplicadas y cada clave puede mapearse a mas de un valor. Algunas implementaciones permiten clave nula y valor nulo como HashMap y LinkedHashMap, pero otras no como TreeMap.
- El orden de un mapa depende de la implementación, por ejemplo, TreeMap y LinkedHashMap tienen un orden predecible, mientras que HashMap no.
- Hay dos interfaces para implementar Map en Java: Map y SortedMap, y tres clases: HashMap, TreeMap y LinkedHashMap.

Jerarquía en Map

La plataforma Java contiene tres implementaciones de mapas de uso general: HashMap, TreeMap y LinkedHashMap. Su comportamiento y rendimiento son precisamente análogos a HashSet, TreeSet y LinkedHashSet.

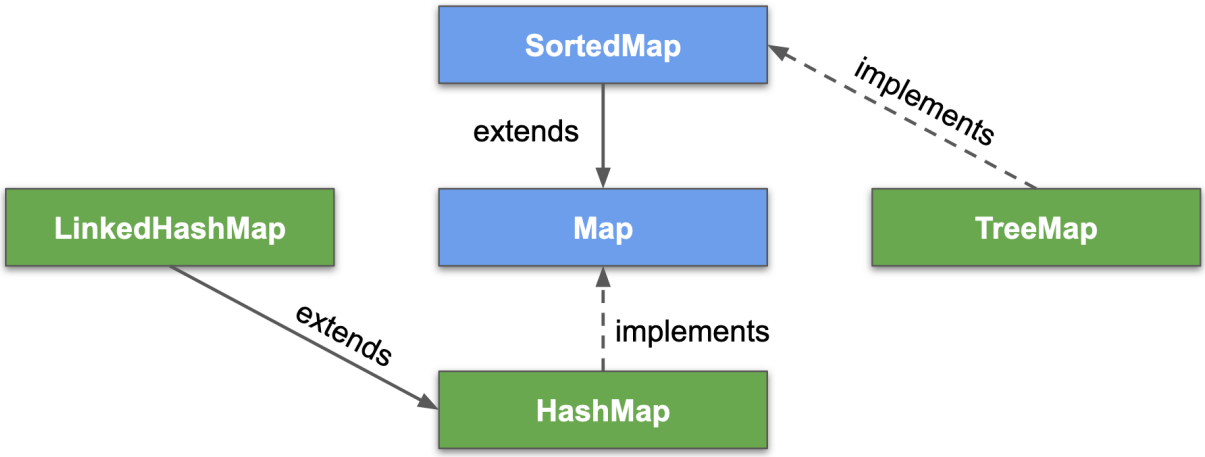

Imagen 7. Jerarquía de Map.

¿Por qué y cuándo usar Maps?

Los mapas son perfectos para usar con el mapeo de asociación de valores clave, como los diccionarios. Los mapas se utilizan para realizar la búsqueda por claves o cuando desean recuperar y actualizar elementos por claves. Algunos ejemplos son:

- Un mapa de códigos de error y sus descripciones.
- Un mapa de códigos postales y ciudades.
- Un mapa de clases y estudiantes. Cada clase está asociada con una lista de estudiantes.

Métodos en la interfaz del mapa:

- Object put(Object key, Object value): este método se utiliza para insertar nuevos valores, con clave y valor.
- void putAll(Map map): este método se utiliza para insertar múltiples valores, a través de un mapa como parámetro.
- Object remove(Object key): este método se utiliza para eliminar una entrada para la clave especificada.
- Object get(Object key): este método se utiliza para devolver el valor de la clave especificada.
- boolean containsKey(Object key): este método se utiliza para buscar la clave especificada.
- Set keySet(): este método se utiliza para devolver la vista tipo Set que contiene todas las claves.
- Set entrySet(): este método se utiliza para devolver la vista tipo Set que contiene todas las claves y valores.

HashMap

Proporciona la implementación básica de la interfaz Map. Almacena los datos en pares clave, valor. Para acceder a un valor se debe conocer su clave. HashMap se conoce como "HashMap" porque utiliza una técnica llamada Hashing.

Hashing es una técnica para convertir un String grande en un String pequeño que representa el misma String. Un valor más corto ayuda en la indexación y búsquedas más rápidas.

Algunas características importantes de HashMap son:

- HashMap no permite claves duplicadas pero permite valores duplicados.
- Esta clase no garantiza el orden del mapa; en particular, no garantiza que el pedido se mantendrá constante en el tiempo.

Creando Map con HashMap

```
Map<String, List<String>> continentes = new HashMap<>();
System.out.println(continentes);
```

Agregando elementos con .put

```
continentes.put("America", Arrays.asList("Chile", "Peru", "Argentina",
"Colombia"));
continentes.put("Europa",Arrays.asList("España", "Portugal","Suecia","Suiza"));
System.out.println(continentes);
```

La salida del programa:

```
{America=[Chile, Peru, Argentina, Colombia], Europa=[España, Portugal, Suecia,
Suiza]}
```

Cambiando los valores para una clave

```
continentes.put("Europa", Arrays.asList("Alemania",
"Luxemburgo","Belgica","Holanda"));
```

La salida del programa:

```
{America=[Chile, Peru, Argentina, Colombia], Europa=[Alemania, Luxemburgo,
Belgica, Holanda]}
```

Retornando sus claves con .keySet

```
continentes.keySet().forEach(System.out::println);
```

La salida del programa:

```
America
Europa
```

Retornando los valores del map con .entrySet

```
continentes.entrySet().forEach(continente ->
continente.getValue().forEach(System.out::println));
```

La salida del programa:

```
Chile
Peru
Argentina
Colombia
Alemania
Luxemburgo
Belgica
Holanda
```

Iterando maps con .forEach

Se puede iterar usando forEach, usando una función lambda que recibe como parámetros, la clave y el valor del mapa. Luego itera el valor e imprime en pantalla.

```
Map<String, List<String>> continentes = new HashMap<>();
continentes.put("America", Arrays.asList("Chile", "Peru", "Argentina",
"Colombia"));
continentes.put("Europa", Arrays.asList("España",
"Portugal","Suecia","Suiza"));
continentes.forEach((key, value) -> value.forEach(pais ->
System.out.println(key + "\t - " + pais)));
```

La salida del programa:

```
America  - Chile
America  - Peru
America  - Argentina
America  - Colombia
Europa   - España
Europa   - Portugal
Europa   - Suecia
Europa   - Suiza
```

TreeMap

El mapa se ordena de acuerdo con al orden de las claves, o a un comparador proporcionado en el momento de la creación del mapa, según el constructor utilizado. Esto demuestra ser una forma eficiente de clasificar y almacenar los pares de clave-valor.

Algunas características importantes del TreeMap son:

Creando TreeMap

```
Map<String, Integer> libros = new TreeMap<>();
System.out.println(libros); //{}
```

Agregando valores

```
Map<String, Integer> libros = new TreeMap<>();
libros.put("Head First Java", 25000);
libros.put("The Rust Programming Language", 23000);
libros.put("Programming In Scala Second Edition", 30000);
libros.put("Cracking The Coding Interview", 28000);
libros.put("Effective Java Third Edition", 340000);
libros.put("Kafka The Definitive Guide", null); //Valores no pueden ser null
libros.put("Kafka The Definitive Guide", 24400);
libros.forEach((k,v) -> System.out.println(k+": $"+v));
```

La salida del programa de forma ordenada:

```
Cracking The Coding Interview: $28000
Effective Java Third Edition: $340000
Head First Java: $25000
Kafka The Definitive Guide: $24400
Programming In Scala Second Edition: $30000
The Rust Programming Language: $23000
```

Eliminando un valor por su clave

```
libros.remove("Cracking The Coding Interview");
```

LinkedHashMap

LinkedHashMap es como HashMap con una característica adicional de mantener un orden de cosas insertadas en él. HashMap ofrece la ventaja de una rapida inserción, búsqueda y eliminación.

Algunas características importantes de LinkedHashMap son las siguientes:

- Puede tener una clave nula y varios valores nulos.
- Es lo mismo que un HashMap con la características de que mantiene el orden de inserción.

Creando LinkedHashMap

```
Map<String, Double> sismos = new LinkedHashMap<>();
```

Agregando datos

```
sismos.put("31 km al NO de Camiña", 4.5);
sismos.put("73 km al N de Calama", 4.2);
sismos.put("68 km al SO de Tongoy", 3.8);
sismos.put("52 km al N de Mejillones", 3.6);
sismos.put("68 km al SO de Tongoy", 3.6);
sismos.put("121 km al O de Pichilemu", 3.2);
sismos.put("126 km al O de Pichilemu", 3.5);
sismos.put("84 km al NE de San Pedro de Atacama", 3.0);
sismos.put("63 km al SO de Ollagüe", 3.1);
sismos.put("68 km al E de Combarbalá", 3.4);
sismos.put("29 km al N de Tongoy", 3.0);
sismos.put("51 km al SO de Los Vilos", 4.0);
sismos.put("51 km al SO de Los Vilos", 3.5);
sismos.forEach((k,v) -> System.out.println(k+" -> "+v));
```

La salida del programa, en el orden que se insertaron los datos:

```
31 km al NO de Camiña -> 4.5
73 km al N de Calama -> 4.2
68 km al SO de Tongoy -> 3.6
52 km al N de Mejillones -> 3.6
121 km al O de Pichilemu -> 3.2
126 km al O de Pichilemu -> 3.5
84 km al NE de San Pedro de Atacama -> 3.0
63 km al SO de Ollagüe -> 3.1
68 km al E de Combarbalá -> 3.4
29 km al N de Tongoy -> 3.0
51 km al SO de Los Vilos -> 3.5
```

Filtrando con .filter

Los sismos mayores a 3.5 grados

```
sismos.entrySet().stream().filter(mag -> mag.getValue() >
3.5).forEach(System.out::println);
```

La salida del programa:

```
31 km al NO de Camiña=4.5
73 km al N de Calama=4.2
68 km al SO de Tongoy=3.6
52 km al N de Mejillones=3.6
```

Resumiendo

La gran variedad implementaciones para colecciones de Java, hace que sea complejo recordar la especialidad de cada una, a continuación una tabla con las características que ofrecen.

Clase/Interfaz	Notas	Duplicados
Collection	La interfaz principal de todo el marco de colecciones.	Depende
List	Le permite poner elementos en un orden específico. Cada elemento está asociado con un índice.	Si
Set	Prohibe elementos duplicados. Prueba fácil de membresía. No puedes ordenar los elementos.	No
Queue	Solo la "cabeza" (siguiente) está disponible.	Si
Map	Mapa de claves y valores	Claves:No, Valores: Si

Tabla 1. Resumen implementación de colecciones Java