Colecciones y API (Parte I)

Introducción a Colecciones

Competencias

- Comprender el uso de arreglos orientados a objetos para resolver problemas cotidianos dentro del mundo de la programación.
- Distinguir las distintas interfaces de colecciones para implementarlas en problemas de código.

Motivación

La programación es un lenguaje excepcional que nos da la capacidad de abstraer objetos reales y convertirlos en objetos virtuales para diversos usos. Sin embargo, si contamos con diversos elementos y no los agrupamos tendremos un código poco legible y desordenado.

Es por ello que la pregunta es ¿cómo guardamos estos elementos y los utilizamos posteriormente? La respuesta es simple: Colecciones. Pero, ¿qué son las colecciones? Pues bien, las colecciones son objetos/elementos que se van incorporando dentro de una entidad mayor, generalmente representada por un grupo natural.

Por ejemplo, una carpeta o bandeja de entrada en el correo pasaría a ser una colección de mails, ¿por qué? Porque agrupa a cada elemento (mails) en una entidad mayor llamada bandeja de entrada. Entonces, ¿para qué es necesario saber colecciones? Para solucionar problemas de la vida cotidiana de un programador/a y dar orden a estas variables que hemos ido viendo capítulo a capítulo.

Colecciones

Una colección es un grupo de objetos individuales representados como una sola unidad. El lenguaje Java, por ejemplo, nos proporciona Frameworks de colecciones mediante 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 son la base del marco de colecciones en Java. Estas interfaces permiten manipular colecciones independientemente de los detalles de su representación.
Como se puede ver en la siguiente figura, las interfaces de la colección principal forman una jerarquía:

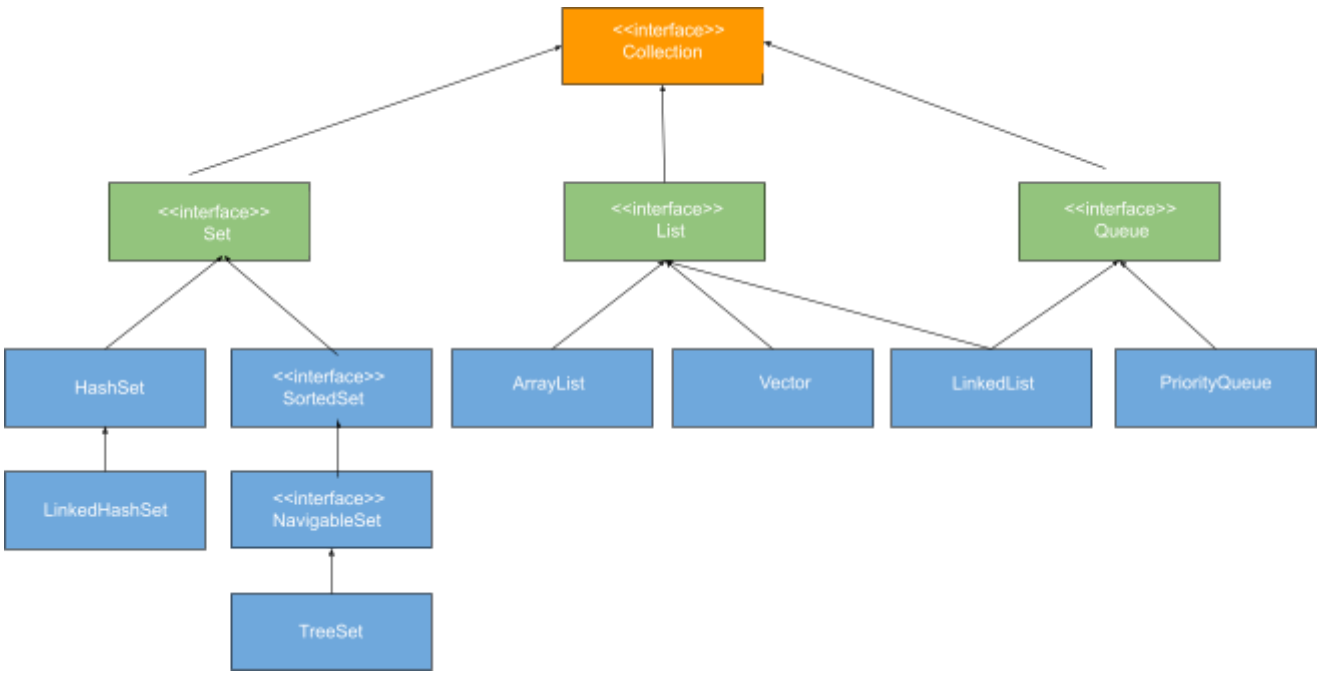

Imagen 1. Elementos que contiene la interfaz Collections en Java.
Fuente: Desafío Latam

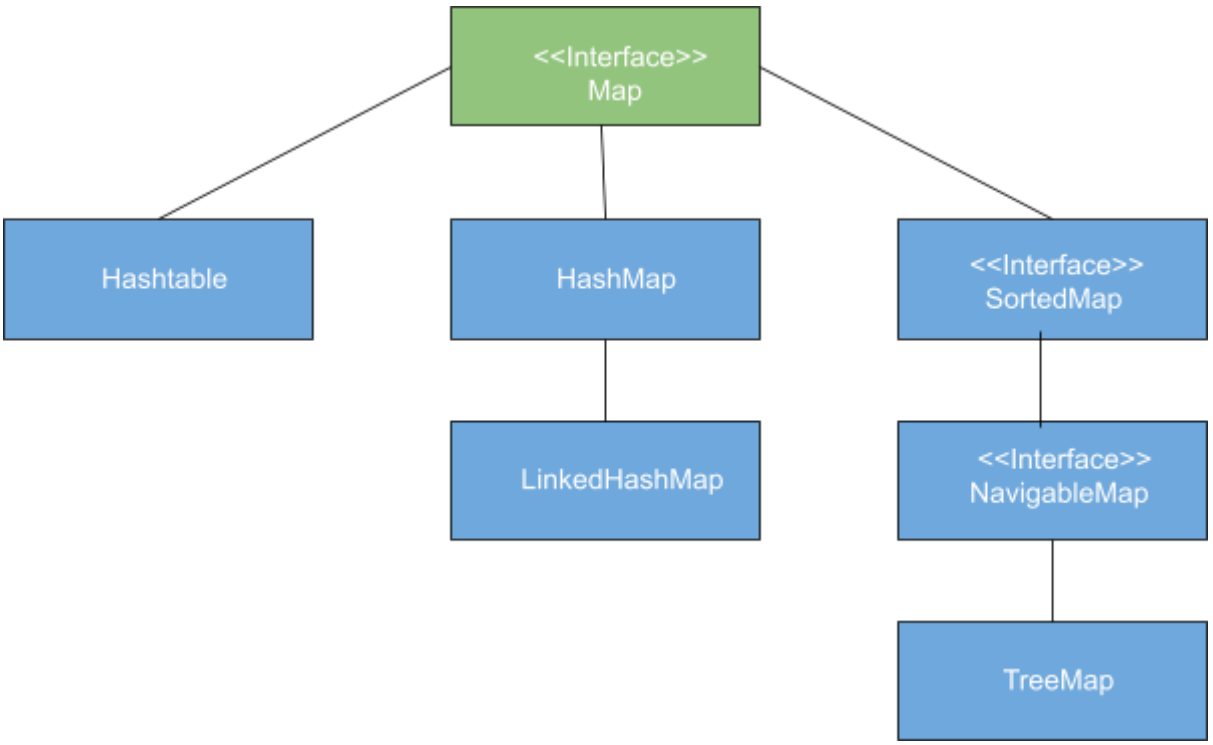

Imagen 2. Elementos que contiene la interfaz Map en Java
Fuente: Desafío Latam

Introducción a List

Competencias

- Aplicar el uso de List para resolver problemas cotidianos dentro del mundo de la programación.
- Crear distintas implementaciones de tipo List para ocupar métodos como agregar, eliminar y recorrer datos.

Introducción

¿Qué es List?

Una lista es una colección ordenada, a veces llamada secuencia, que contiene elementos en su interior. 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 fueron definidas previamente en los gráficos anteriores.

Además de algunas operaciones heredadas desde Collection, la interfaz List también incluye operaciones de uso propio. Algunas de estas son:

- Acceso posicional (get, set, add, addAll, remove y size)
- Búsqueda (indexOf y lastIndexOf)
- Iteración (listIterator)

Más adelante veremos cómo se usa cada una de estas operaciones, pero primero partiremos creando un ArrayList para entender cómo funcionan y desde dónde se implementan.

Creando un ArrayList

Un arrayList es una estructura de datos que puede estirarse para acomodar elementos adicionales dentro de sí mismos y reducirse a un tamaño más pequeño cuando se eliminan elementos. Es una implementación de matriz-redimensionable de la interfaz List. Además, permite a todos los elementos, incluyendo el valor nulo. Se puede ver en la imagen la implementación de este método y que implementa desde List.

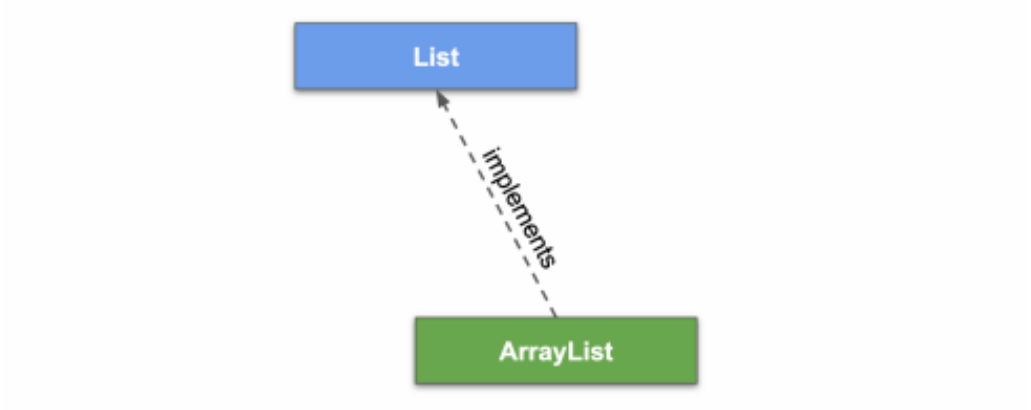

Imagen 3. Implementación ArrayList en List
Fuente: Desafío Latam

Ahora, vayamos a un ejemplo de cómo crear un ArrayList en Java. Para esto, partiremos definiendo una List del tipo String, cuyo nombre será list e instanciamos como una ArrayList.

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

Es importante destacar que los elementos de la lista tienen una posición y se les llama índices. Se puede acceder a los valores pasando el índice como parámetro del método get. En el ejemplo podemos ver como al llamar la posición número cero esta llama al elemento agregado “Java”.

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

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

Ejercicio guiado: Métodos de acceso posicional (Parte I)

Como se mencionó previamente, tenemos distintos métodos para manejar el acceso posicional de elementos. La colección List permite agregar, quitar, obtener y establecer operaciones basadas en posiciones numéricas de elementos en la lista. A continuación, veremos algunos de los métodos más usados sobre estas listas con un ejemplo de las ciudades de Chile:

- **Paso 1:** Para crear una ArrayList<>() hay que importar su implementación desde "`util.java.ArrayList`" en la parte superior de la clase y luego instanciarla como se muestra en la siguiente imagen.

```
import java.util.ArrayList

      ArrayList<String> ciudades = new ArrayList<>();
```

- **Paso 2:** Para incorporar elementos a la lista, debemos ocupar el método void `add()`, el cual ocupa el nombre de la lista para ir incorporando elementos al ArrayList previamente definido.

```
      ArrayList<String> ciudades = new ArrayList<>();
      ciudades.add("Santiago");
      ciudades.add("Iquique");
      ciudades.add("Arica");
      ciudades.add("Concepción");
      ciudades.add("La Serena");
      ciudades.add("Puerto Montt");
      System.out.print(ciudades);

      ---------------------------
      Impresión en pantalla:

      [Santiago, Iquique, Arica, Concepción, La Serena, Puerto Montt]
```

- **Paso 3:** Para incorporar elementos desde una colección específica a la lista de ciudades que tenemos, podemos ocupar el método `addAll()`. Este método agrega todos los elementos de la colección a la lista. El primer elemento se inserta en el índice dado. Si ya hay un elemento en esa posición, ese elemento y los otros elementos posteriores (si los hay) se desplazan hacia la derecha al aumentar su índice.

```
        //Incorporación de ciudades al ArrayList
        ArrayList<String> ciudades = new ArrayList<>();
        ciudades.add("Santiago");
        ciudades.add("Iquique");
        ciudades.add("Arica");
        ciudades.add("Concepción");
        ciudades.add("La Serena");
        ciudades.add("Puerto Montt");

        //Incorporación de más ciudades desde una colección distinta
llamada otrasCiudades

        ArrayList<String> otrasCiudades = new ArrayList();
        otrasCiudades.add("Rancagua");
        otrasCiudades.add("Punta Arenas");
        ciudades.addAll(otrasCiudades);
        System.out.print(ciudades);

        ---------------------------
        Impresión en pantalla:

        [Santiago, Iquique, Arica, Concepción, La Serena, Puerto Montt,
Rancagua, Punta Arenas]
```

- **Paso 4:** Para obtener un elemento en base a su posición, podemos hacer uso del get y buscar este índice al interior de la lista. Si queremos saber de qué elemento se está hablando, usaremos el método `get()`. Este método devuelve el elemento en el índice especificado partiendo. Cabe recordar que la lista de los índices parte desde el número cero.

```
System.out.print(list.get(0));
System.out.print(list.get(4));
---------------------------
Impresión en pantalla:

[Santiago]
[Puerto Montt]
```

- **Paso 5:** Para remover un elemento específico desde la ArrayList previamente hecha, podemos utilizar el método `remove()`. Este método elimina un elemento del índice especificado. Desplaza los elementos posteriores (si los hay) a la izquierda y disminuye sus índices en 1.

```
        //Para eliminar a Puerto Montt por ejemplo, se puede remover
usando su posición que la número 4 al interior de la lista
       ciudades.remove(4);
       System.out.print(ciudades);
       ---------------------------
       Impresión en pantalla:

       [Santiago, Iquique, Arica, Concepción, La Serena, Rancagua, Punta
Arenas]

        //Para comprobar si se elimina el elemento, podemos usar el
mismo método usando su nombre y este nos arrojará false si no lo
encontró o true si lo elimino

        ciudades.remove("Puerto Montt");
```

- **Paso 6:** Para modificar un elemento al interior de la lista en base a su índice correspondiente, podemos usar el método `set()`. 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.

```
      ciudades.set(2, "Talca");
      System.out.print(ciudades);
      --------------------------
      Impresión en pantalla:

      [Santiago, Iquique, Talca, Concepción, La Serena, Rancagua, Punta
Arenas]
```

- **Paso 7:** Para encontrar la cantidad exacta de elementos que contiene la lista, podemos utilizar el método `size()`. Este método devuelve la cantidad de elementos al interior de la lista.

```
      System.out.print(ciudades.size());
      --------------------------
      Impresión en pantalla:

      [7]
```

Ejercicio guiado: Métodos de búsqueda

Por su parte, List también proporciona métodos para buscar elementos y los devuelve en base a su posición numérica. Los siguientes dos métodos son compatibles con List para esta operación:

- **Paso 8:** Para buscar en base a contenido de un elemento, podemos usar el método `indexOf()`. Este método devuelve la posición del elemento dado, si el elemento aún está en la lista o (-1) si el elemento no está presente en la lista.

```
System.out.print(ciudades.indexOf("Puerto Montt"));
System.out.print(ciudades.indexOf("Santiago"));
---------------------------
Impresión en pantalla:

[-1] //Fuera de la lista
[0]  //Devuelve su posición que es 0
```

- **Paso 9:** Para buscar en base al último contenido que tuvo un elemento, podemos usar el método `lastIndexOf()`. Este método devuelve la posición del elemento dado, si el elemento aún está en la lista o (-1) si el elemento no está presente en la lista.

```
System.out.print(ciudades.lastIndexOf("Puerto Montt"));
System.out.print(ciudades.lastIndexOf("Santiago"));
---------------------------
Impresión en pantalla:

[-1] //Fuera de la lista
[0]  //Devuelve su posición que es 0
```

Ejercicio guiado: Métodos de iteración

Como se mencionó también al inicio de este capítulo, contamos con un método para iterar en una lista mediante `listIterator()`. A continuación, veremos cómo ocupamos este método siguiendo con el ejemplo de las ciudades de Chile:

- Paso 10: Para crear una ArrayList<>() hay que importar su implementación desde "util.java.ArrayList" en la parte superior de la clase y luego instanciarla como se muestra en la siguiente imagen:

```
import java.util.ListIterator;

	//Devuelve la lista ordenada tal cual llega
	List<String> ciudades = new ArrayList<>();
	ciudades.add("Santiago");
	ciudades.add("Iquique");
	ciudades.add("Arica");
	ciudades.add("Concepción");
	ciudades.add("La Serena");
	ciudades.add("Puerto Montt");

	ListIterator<String> ciudadesIterator = ciudades.listIterator();
	System.out.print(ciudadesIterator.hasNext());
	---------------------------
	Impresión en pantalla:
	true
	//El true es el equivalente a la lista recorrida y ordenada
	//[Santiago, Iquique, Arica, Concepción, La Serena, Puerto Montt]



	//Devuelve la lista ordenada al revés
	List<String> ciudades = new ArrayList<>();
	ciudades.add("Santiago");
	ciudades.add("Iquique");
	ciudades.add("Arica");
	ciudades.add("Concepción");
	ciudades.add("La Serena");
	ciudades.add("Puerto Montt");

	ListIterator ciudadesIterator = ciudades.listIterator();
	System.out.print(ciudadesIterator.hasPrevious());
```

```
       --------------------------
       Impresión en pantalla:
       [false]
       //El false es equivalente a la lista recorrida pero ordenada de
manera inversa
       //[Puerto Montt,La Serena, Concepción,  Arica,  Iquique,
Santiago]
```

Ejercicio propuesto (1)

Incorporar las notas que ha obtenido durante el último curso de Java:

- Agregar las siguientes notas como Float.

```
   [6.9, 6.7, 4.5, 5.5, 5.8, 6.1, 7.0]
```

- Eliminar la nota más baja del listado.
- Sacar el promedio final del curso Java.

Introducción a Set

Competencias

- Aplicar el uso de Set para resolver problemas cotidianos dentro del mundo de la programación.
- Crear distintas implementaciones de tipo Set para utilizar métodos como agregar, eliminar, interceptar y recorrer datos.

Introducción

¿Qué es Set?

Set es una interfaz que extiende desde Collections. Es uno de los tipo de colecciones más desordenado en relación a los 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. Además, Set tiene varios métodos de acceso posicional como add, remove, clear y size, entre muchos otros, que permiten mejorar el uso de esta interfaz.

Más adelante veremos cómo se usa cada una de estas operaciones, pero primero partiremos creando un HashSet para entender cómo funcionan y desde donde se implementan.

Creando un Set

Un Set es una interfaz presente en Java que extiende de Collections, se comporta como una colección de objetos desordenada en donde los valores duplicados no pueden ser almacenados. Además, dentro de las posibles implementaciones presentadas por Set, encontramos a HashSet. Esta implementación almacena elementos en una tabla llamada hash, cuyos rendimientos son mejores que el resto de las implementaciones, sin embargo, no garantiza el orden de iteración.

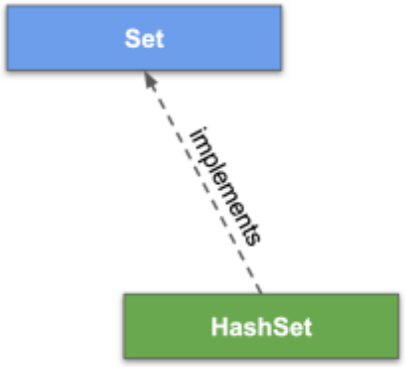

Imagen 4. Implementación de HashSet en Set.
Fuente: Desafio Latam

Ahora, vayamos a un ejemplo de cómo crear un HashSet en Java. Para esto, partiremos definiendo una Set del tipo String, cuyo nombre será capitales e instanciamos como una HashSet. Luego le agregaremos algunos lenguajes de programación para mostrar que el elemento "Java" no se repite.

```
import java.util.Set;
import java.util.HashSet;
       Set<String> languages = new HashSet<>();
       languages.add("Java");
       languages.add("Go");
       languages.add("Erlang");
       languages.add("Java");
       languages.add("Elixir");
       languages.add("Fortran");
       System.out.println(languages); //[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 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).

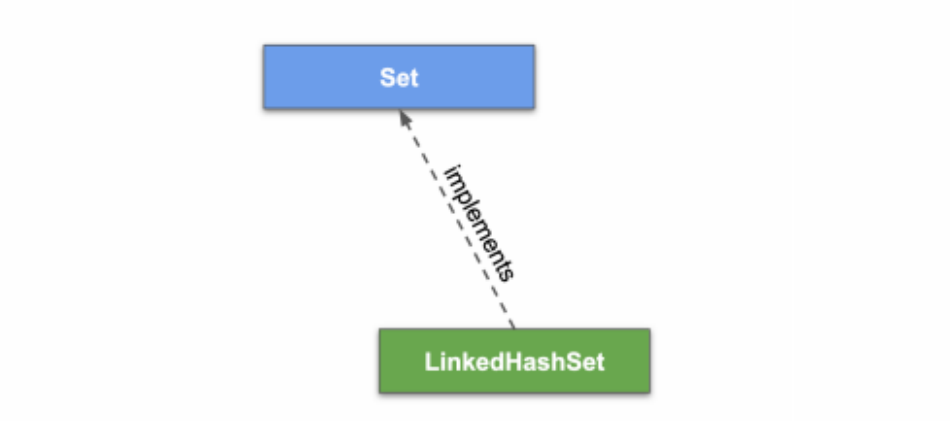

Imagen 6. Implementación de LinkedHashSet en Set.
Fuente: Desafío Latam

En el ejemplo que se muestra a continuación, se puede ver cómo estos programadores a medida que se van incorporando, se van imprimiendo en consola.

```
import java.util.LinkedHashSet;

       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]
```

Creando un TreeSet

Como se mencionó en el párrafo previo, TreeSet almacena sus elementos en un árbol rojo-negro, es decir, ordena sus elementos en función de sus valores. Es sustancialmente más lento que HashSet.

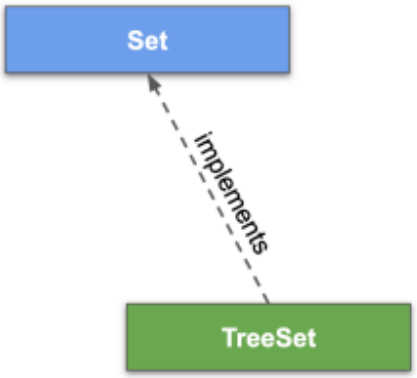

Imagen 5.Implementación de TreeSet en Set.
Fuente: Desafio Latam

Ejercicio guiado: Capitales del mundo

- **Paso 1:** Para explicar lo que acabamos de mencionar, realizaremos un ejemplo guiado con las capitales sudamericanas. Cabe destacar que repetiremos "Brasilia" a propósito para mostrar que al imprimir capitales, no aparecerá repetida.

```
import java.util.TreeSet;

        Set<String> capitales = new TreeSet<>();
        capitales.add("Buenos Aires");
        capitales.add("Brasilia");
        capitales.add("Asunción");
        capitales.add("Lima");
        System.out.println(capitales); //[Asunción, Brasilia, Buenos
Aires, Lima]
```

- **Paso 2**: Ahora, veremos cómo se conforma la unión de 2 colecciones mediante `TreeSet`. Para ello crearemos otro `HashSet` llamado capitales2 y le incorporaremos 5 capitales más. Luego con un `TreeSet` juntaremos las capitales e imprimimos que resulta del Set capitalesUnidas.

```
      Set<String> capitales = new TreeSet<>();
      capitales.add("Buenos Aires");
      capitales.add("Brasilia");
      capitales.add("Asunción");
      capitales.add("Lima");

      Set<String> capitales2 = new HashSet<>(Arrays.asList("Caracas",
"Bogotá", "Montevideo","Quito", "Brasilia"));

      Set<String> capitalesUnidas = new TreeSet<>(capitales);
      capitalesUnidas.addAll(capitales2);
      System.out.println(capitalesUnidas);
      ---------------------------
      Impresión en pantalla:

      [Asunción, Bogotá ,Brasilia, Buenos Aires, Caracas, Lima,
Montevideo, Quito]
```

- **Paso 3:** Si queremos borrar una colección completa de la lista, debemos usar el método `removeAll()`. Al igual que en `ArrayList()`, remove es un método de acceso posicional que sirve para eliminar elementos desde una colección general, en este caso para eliminar una colección llamada capitales2 (visto en el ejemplo anterior).

```
      Set<String> removerCapitales = new HashSet<>(capitales);
      removerCapitales.removeAll(capitales2);
      System.out.println(removerCapitales);
      ---------------------------
      Impresión en pantalla:

      [Asunción,Brasilia, Buenos Aires, Lima]
```

- **Paso 4**: Para encontrar valores en común entre colecciones, se puede usar el método retainAll(). Para nuestro caso, usaremos las colecciones previas de capitales y capitales2. Es decir, buscaremos la intersección de ambas colecciones.

```
Set<String> interseccionCapitales = new HashSet<>(capitales);
interseccionCapitales.retainAll(capitales2);
System.out.println(interseccionCapitales);
---------------------------
Impresión en pantalla:
[Brasilia]
```

Ejercicio propuesto (2)

Controlar los precios de productos de una tienda.

- Agregar los siguientes precios de dulces.

```
[100, 200, 100, 500, 400]
```

- Volver a agregar precios, pero de la categoría bebidas. Esta vez será una nueva colección y deberá unir los precios con la colección anterior.

```
[200, 400, 30000]
```

- Eliminar el precio 30000 ya que fue un error de tipeo.

Introducción a Queue

Competencias

- Aplicar el uso de Queue para resolver problemas cotidianos dentro del mundo de la programación.
- Crear distintas implementaciones de Queue para ocupar métodos como agregar, eliminar y recorrer datos.

Introducción

¿Qué es Queue?

Debemos partir definiendo que Queue es una palabra proveniente del inglés que significa "cola" o "colocar en la lista". 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 es una cola para un local de comida rápida, donde el primero en pagar es el primero en servirse.

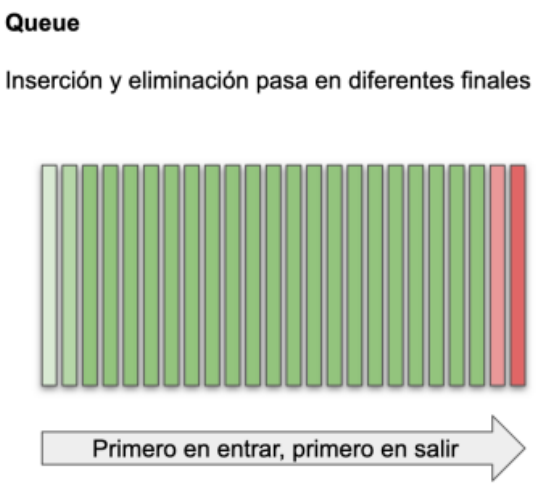

Imagen 7. Funcionamiento de un encolamiento.
Fuente: Desafio Latam

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.

Ejercicio guiado: Métodos de acceso posicional

Para generar un ejemplo, seguiremos usando la geografía como referencia, pero esta vez con enfoque a los continentes.

- **Paso 1**: Para crear una LinkedList<>() del tipo Queue hay que importar su implementación desde "util.java.LinkedList" en la parte superior de la clase y luego instanciarla como se muestra en la siguiente imagen.

```
import java.util.LinkedList;

       Queue continentes = new LinkedList<>();
```

- **Paso 2**: Para agregar elementos a encolar, podemos utilizar el método add(). Este método se usa para agregar elementos a la cola, más específicamente, al final de la cola. Se usa LinkedList para este caso pero dependerá de la prioridad según sea el caso de implementación de PriorityQueue.

```
            Queue continentes = new LinkedList<>();
            continentes.add("África");
            continentes.add("América");
            continentes.add("Europa");
            continentes.add("Oceanía");
```

```
  continentes.add("Asia");
  continentes.add("Antártica");
--------------------------
Impresión en pantalla:
[África, América, Europa, Oceanía, Asia, Antártica]
```

- **Paso 3**: Para poner un elemento específico del encolamiento continentes, podemos usar el método remove(). Este método elimina y devuelve el encabezado de la cola. Lanza NoSuchElementException cuando la cola está vacía.

```
System.out.println(continentes.remove("Antártica"));
System.out.println(continentes);
--------------------------
Impresión en pantalla:
[África, América, Europa, Oceanía, Asia]
```

- **Paso 4**: Para eliminar un encabezado, es decir, el primero de la lista se elimina, podemos usar el método poll(). Este método elimina y devuelve el encabezado de la cola. Devuelve nulo si la cola está vacía.

```
System.out.println(continentes.poll());
System.out.println(continentes);
--------------------------
Impresión en pantalla:
[América, Europa, Oceanía, Asia]
```

- **Paso 5**: Para obtener el encabezado de la cola sin eliminarlo podemos usar el método peek(). Este método se utiliza para ver el encabezado de la cola sin eliminarlo, devuelve nulo si la cola está vacía.

```
System.out.println("peek : " + continentes.peek());
System.out.println(continentes);
--------------------------
Impresión en pantalla:
peek: [América]
```

- **Paso 6**: Para encontrar un elemento, al igual que peek se puede hacer sin eliminar el objeto con el método element(): Este método es similar a peek(), pero lanza NoSuchElementException cuando la cola está vacía.

```
System.out.println("element: "+continentes.element());
System.out.println(continentes);
---------------------------
Impresión en pantalla:
element: [América]
```

- **Paso 7**: Para encontrar el tamaño de un encolado, podemos utilizar el método size(). Este método devuelve el número de elementos en la cola.

```
System.out.println(continentes.size());
---------------------------
Impresión en pantalla:
[4]
```

Ejercicio propuesto (3)

Llevar el orden de los clientes que visitan el negocio.

- Agregar los siguientes clientes que dejaron su nombre en el negocio.

```
  Armando Casas, Pedro del Campo, Silvana Susana, Natalia Ivanovic,
Roger Federer, Dominic Toretto
```

- Eliminar a Roger Federer por ser una figura pública.
- Encontrar el tamaño de la lista total de clientes.

Introducción a Map

Competencias

- Aplicar el uso de Map para resolver problemas cotidianos dentro del mundo de la programación.
- Crear distintas implementaciones de Map para ocupar métodos como agregar, eliminar y recorrer datos.

Introducción

¿Qué es Map?

La interfaz Map representa una asignación entre una clave y un valor. La interfaz de map no es un subtipo de Collection. Por lo tanto, se comporta un poco diferente del resto de los tipos de colecciones. Un map no puede contener claves duplicadas y cada clave puede mapearse a más de un valor.

Algunas implementaciones permiten una 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.

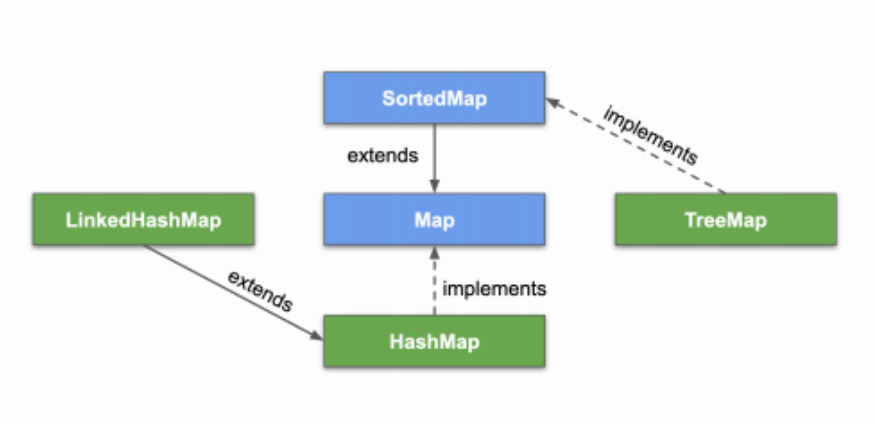

Imagen 8. Jerarquía de Map.
Fuente: Desafío Latam

¿Por qué y cuándo usar Maps?

Los mapas son perfectos para usar con la asociación de valores clave. Por ejemplo, cuando uno busca una palabra en el diccionario, nos guiamos por la letra del abecedario para buscar la palabra específica. Map hace algo similar y utiliza claves para realizar la búsqueda cuando desean recuperar y actualizar elementos. 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.

Resumiendo

La gran variedad de implementaciones que hemos visto previamente, nos sirve para manejar distintas estructuras de datos e ir definiendo su funcionalidad en base a las características que ofrecen. A Continuación mostraremos un resumen de cuándo ocupar cada uno de estos métodos.

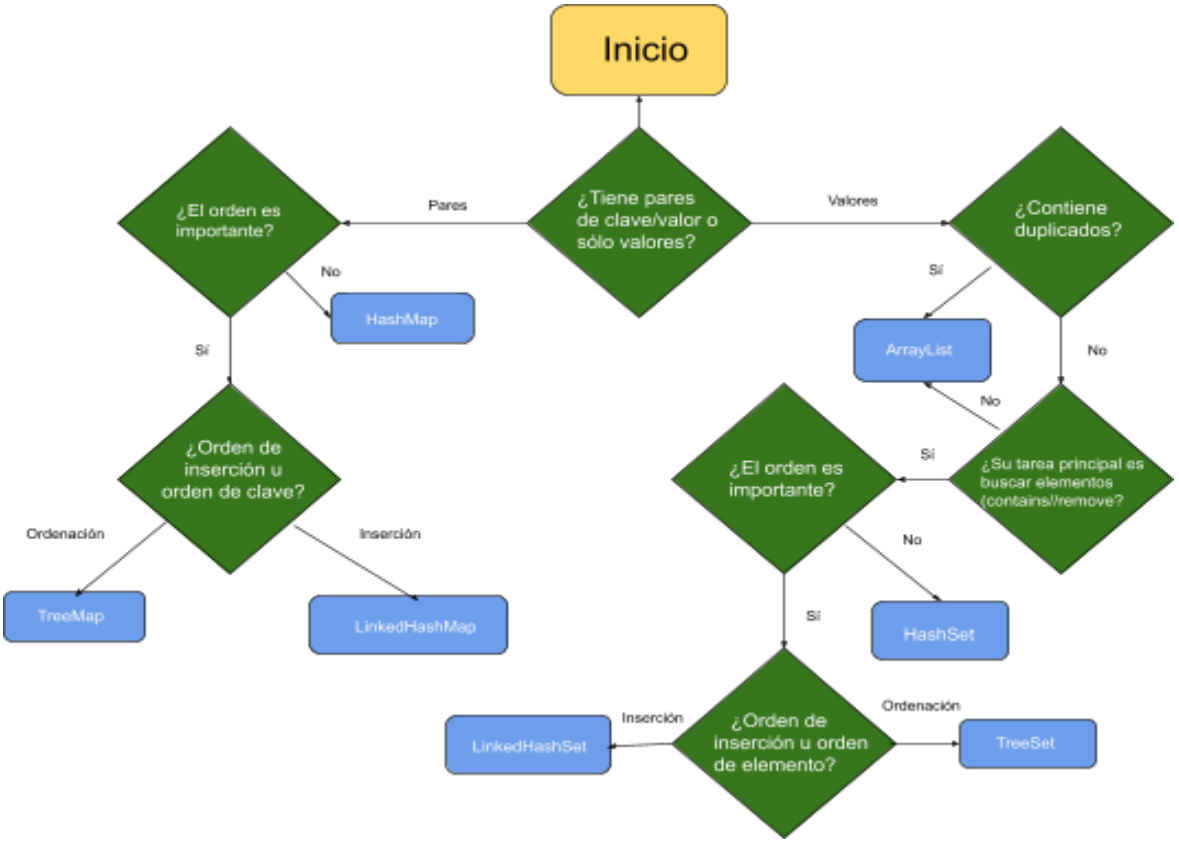

Imagen 9. Resumen de funcionalidades.
Fuente: Desafío Latam

Ejercicio guiado: Métodos en la interfaz de Map

Seguiremos usando referencia geoespacial, pero esta vez con enfoque a los planetas.

- **Paso 1**: Para crear una TreeMap<>() del tipo Map, hay que importar su implementación desde "util.java.TreeMap" en la parte superior de la clase y luego instanciarla, como se muestra en la siguiente imagen:

```
import java.util.TreeMap;
import java.util.Map;

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

- **Paso 2**: Para incorporar elementos a este TreeMap, podemos utilizar el método put(). Este método se utiliza para insertar nuevos valores, con una clave y asignándole un valor. Por ejemplo, la clave sería el nombre del planeta y los valores a entregar serían los años luz que se encuentran de la tierra.

```
            Map<String, Integer> planetas = new TreeMap<>();
            planetas.put("Mercurio", 10);
            planetas.put("Venus", 20);
            planetas.put("Marte", 15);
            planetas.put("Jupiter", 50);
            System.out.println(planetas);
            ---------------------------
            Impresión en pantalla:
            [Jupiter = 50, Marte = 15, Mercurio = 10, Venus = 20]
```

- **Paso 3**: Para eliminar un objeto, este caso un planeta, podemos usar el método remove(). Este método se utiliza para eliminar la entrada de una clave específica.

```
            planetas.remove("Venus");
            System.out.println(planetas);
            ---------------------------
            Impresión en pantalla:
            [Jupiter = 50, Marte = 15, Mercurio = 10]
```

- **Paso 4**: Para obtener un elemento desde Map, podemos utilizar el método get(). Este método se utiliza para devolver el valor de una clave específica. En este caso, nos devolverá 50 que son los años luz desde la tierra a Júpiter.

```
System.out.println(planetas.get("Jupiter"));
---------------------------
Impresión en pantalla:
[50]
```

- **Paso 5**: Para ver si una clave determinada aún está presente en la colección, podemos usar el método containsKey(). Este método se utiliza para buscar una clave específica, en este caso buscaremos el planeta Tierra y Júpiter.

```
System.out.print(planetas.containsKey(“Tierra”));
System.out.print(planetas.containsKey(“Jupiter”));
---------------------------
Impresión en pantalla:
[false] //Significa que no está en la colección
[true]  //La podemos encontrar en la colección
```

- **Paso 6**: Para retornar una o más claves del Map, podemos usar el método keySet(). Este método se utiliza para devolver la vista tipo Set que contiene todas las claves. En este caso por ejemplo ocuparemos un forEach() para recorrer toda la colección.

```
planetas.keySet().forEach(System.out.print);
---------------------------
Impresión en pantalla:
[Júpiter, Marte, Mercurio]
```

- **Paso 7**: Para obtener aquellos valores de planetas que tengan una distancia menor a 16 años luz, podemos usar el método entrySet(). Este método se utiliza para devolver la vista tipo Set que contiene todas las claves y valores, en este caso por ejemplo ocuparemos un forEach() para recorrer toda la colección e imprimir aquellos elementos que cumplan con el requisito de filtrado.

```
            planetas.entrySet().stream().filter(aniosLuz->
aniosLuz.getValue()<16).forEach(System.out::print);
           --------------------------
           Impresión en pantalla:
           [Marte = 15, Mercurio = 10]
```

Ejercicio propuesto (4)

Dado los eventos telúricos que enfrenta nuestro país a diario, la ONEMI le entrega los siguientes datos para que usted pueda encontrar esta información:

```
   "31 km al NO de Camiña", 5.7
   "73 km al N de Calama", 5.9
   "68 km al SO de Tongoy", 6.8
   "52 km al N de Mejillones", 6.6
   "68 km al SO de Tongoy", 6.9
```

- ¿Cuál ha sido el temblor con mayor magnitud?
- ¿Cuál ha sido el temblor con menor magnitud?
- Devuelve todos los temblores mayores a 6.5 grados.
- La ONEMI le avisa que se equivocaron en la escala del temblor “52 km al N de Mejillones” y le pide actualizar a 6.8.

Soluciones ejercicios propuestos

Solución Ejercicio Propuesto (1)

Contexto

1.- Generamos una List del tipo `ArrayList` y agregamos cada uno de los elementos con `add`.

```
       ArrayList<Float> notas = new ArrayList<>();
                   notas.add(6.9F);
                   notas.add(6.7F);
                   notas.add(4.5F);
                   notas.add(5.5F);
                   notas.add(5.8F);
                   notas.add(6.1F);
                   notas.add(7.0F);
                   System.out.println(notas);
```

2.- Eliminamos la nota más baja con `remove`.

```
              notas.remove(4.5F);
              System.out.println(notas);
```

3.- Sacamos el promedio de las notas ocupando la suma de las notas con `get` y luego dividiendo con `size`.

```
                   Float sum = notas.get(0) + notas.get(1) +
notas.get(2) + notas.get(3) + notas.get(4) + notas.get(5);
                   Float prom = sum/notas.size();
                   System.out.println(prom);
```

Solución Ejercicio Propuesto (2)

Contexto

1.- Generamos un Set del tipo HashSet y agregamos cada uno de los elementos con `add`.

```
            Set<Integer> precios = new HashSet<>();
            precios.add(100);
            precios.add(200);
            precios.add(100);
            precios.add(500);
            precios.add(400);
```

2.- Generar la nueva colección y unirla a la anterior mediante un nuevo `TreeSet`.

```
            Set<Integer> otrosPrecios = new
TreeSet<>(Arrays.asList(200,400,30000));
            Set<Integer> preciosUnidos = new TreeSet<>(precios);
            preciosUnidos.addAll(otrosPrecios);
```

3.- Eliminar el elemento que contiene 30000 con `remove`.

```
            preciosUnidos.remove(30000);
```

Solución Ejercicio Propuesto (3)

Contexto

1.- Generamos un Queue del tipo `LinkedList` y agregamos cada uno de los elementos con `add`.

```
            Queue clientes = new LinkedList<>();
            clientes.add("Armando Casas");
            clientes.add("Pedro del Campo");
            clientes.add("Silvana Susana");
            clientes.add("Natalia Ivanovic");
            clientes.add("Roger Federer");
            clientes.add("Dominic Toretto");
```

2.- Ocupar `remove` para remover a Roger Federer de la lista.

```
            System.out.println(clientes.remove("Roger Federer"));
```

3.- Ocupar size para encontrar la cantidad total de clientes y se imprimen todos en pantalla.

```
            System.out.println(clientes.size());
            System.out.println(clientes);
```

Solución Ejercicio Propuesto (4)

Contexto

1.- Generamos un Map, del tipo `TreeMap` y agregamos cada uno de los elementos con `put`.

```
            Map<String, Float> sismos = new TreeMap<>();
            sismos.put("31 km al NO de Camiña", 5.7F);
            sismos.put("73 km al N de Calama", 5.9F);
            sismos.put("68 km al SO de Tongoy", 6.9F);
            sismos.put("52 km al N de Mejillones", 6.6F);
            sismos.put("121 km al O de Pichilemu", 6.9F);
```

2.- Ocupar `Collections.max` para encontrar el máximo valor.

```
            System.out.println(Collections.max(sismos.values()));
```

3.- Ocupar `Collections.min` para encontrar el mínimo valor.

```
            System.out.println(Collections.min(sismos.values()));
```

4.-Filtrar los temblores mayores a 6.5 con `filter`.

```
            sismos.entrySet().stream().filter(magnitud ->
magnitud.getValue()>6.5F).forEach(System.out::print);
```