Collection en Java.

1. ¿Qué son las colecciones?

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica Collection para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección, etc.

Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

En este tema vamos a ver qué son y los distintos tipos de colecciones más usados que existen (List, Set y Map). También, vamos a ver que cada uno de los distintos tipos de colecciones puede tener, además, distintas implementaciones, lo que ofrece funcionalidad distinta.

2. Tipos de colecciones.

2.1 List.

La interfaz **List** define una sucesión de elementos. La interfaz **List** admite elementos duplicados. A parte de los métodos heredados de **Collection**, añade métodos que permiten mejorar los siguientes puntos:

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

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

- **ArrayList**: esta es la implementación típica. Se basa en un *array* redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.
- **LinkedList**: esta implementación permite que mejore el rendimiento en ciertas ocasiones. Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

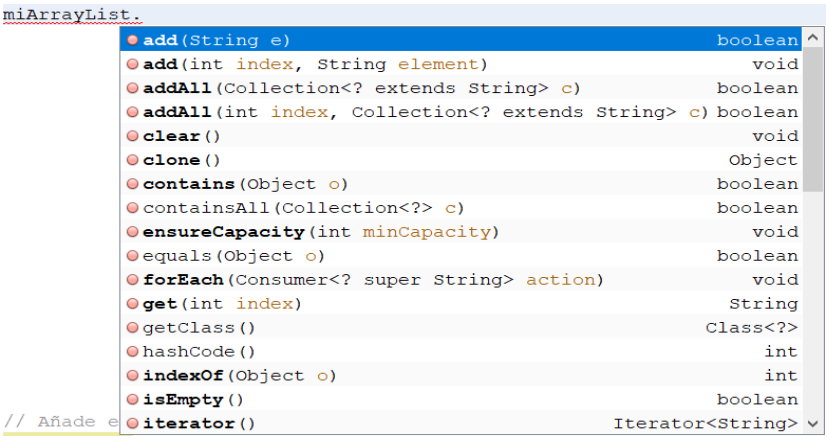
El cuándo usar una implementación u otra de List variará en función de la situación en la que nos encontremos. Generalmente, ArrayList será la implementación que usemos en la mayoría de situaciones. Sobre todo, varían los tiempos de inserción, búsqueda y eliminación de elementos, siendo en unos casos una solución más óptima que la otra.

2.1.1 ArrayList.

Empezamos declarando un ArrayList.

```
// Declaración de un ArrayList de "String". Puede ser de cualquier otro elemento un Objeto (float, Boolean, Object, ...)
ArrayList<String> miArrayList = new ArrayList<>();
```

A continuación, vamos a describir alguno de sus métodos.



// Añade el elemento al ArrayList

miArrayList.add("Elemento");

// Añade el elemento al ArrayList en la posición 'n'

miArrayList.add(1, "Elemento 2");

// Devuelve el numero de elementos del ArrayList

miArrayList.size();

// Devuelve el elemento que esta en la posición '2' del ArrayList

miArrayList.get(2);

// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro

miArrayList.contains("Elemento");

// Devuelve la posición de la primera ocurrencia ('Elemento') en el ArrayList

miArrayList.indexOf("Elemento");

// Devuelve la posición de la última ocurrencia ('Elemento') en el ArrayList

miArrayList.lastIndexOf("Elemento");

// Borra el elemento de la posición '5' del ArrayList

miArrayList.remove(5);

// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.

miArrayList.remove("Elemento");

//Borra todos los elementos de ArrayList

miArrayList.clear();

// Devuelve True si el ArrayList esta vacio. Sino Devuelve False

miArrayList.isEmpty();

// Copiar un ArrayList

ArrayList arrayListCopia = (ArrayList) miArrayList.clone();

// Pasa el ArrayList a un Array

Object[] objarray = miArrayList.toArray();

2.1.1.1 Iteradores.

Una cuestión muy importante a la hora de trabajar con los ArrayList son los "Iteradores" (Iterator). Los Iteradores sirven para recorrer los ArrayList y poder trabajar con ellos. Los Iteradores solo tienen tres métodos:

- "hasNext()" para comprobar que siguen quedando elementos en el iterador. "next()" para que nos dé el siguiente elemento del iterador.
- "remove()" que sirve para eliminar el elemento del Iterador.

```
ArrayList<String> miArrayList = new ArrayList<>();
miArrayList.add(0, "Elemento 0");
miArrayList.add(1, "Elemento 1");
miArrayList.add(2, "Elemento 2");

// Declaramos el Iterador e imprimimos los Elementos del ArrayList
Iterator<String> nombreIterator = miArrayList.iterator();
while(nombreIterator.hasNext()){
        String elemento = nombreIterator.next();
        System.out.print(elemento+" / ");
}
```

2.1.1.2 Bucle for each con arrays.

Esta estructura nos permite recorrer una Colección o un array de elementos de una forma sencilla. Evitando el uso de Iteradores o de un bucle for normal.

La sintaxis es la siguiente:

```
for (TipoBase variable: ArrayDeTiposBase) {..}
```

Un ejemplo de su uso del for each.

```
ArrayList<String> lista = new ArrayList<>();
 // Añadimos elementos
lista.add("Juana");
lista.add("Alicia");
lista.add("Gloria");
 for (String nombre: lista)
  System.out.println(nombre);
```

El algoritmo anterior lo podemos modificar por el método foreach de la clase ArrayList, quedando de la siguiente manera.

```
ArrayList<String> lista = new ArrayList<>();
 // Añadimos elementos
lista.add("Juana");
lista.add("Alicia");
lista.add("Gloria");
lista.forEach((nombre) -> {
   System.out.println(nombre);
```

Realizamos a continuación, un programa que contenga varios métodos utilizados con ArrayList.

Desarrolla un programa que registre una lista de 10 nombres diferentes mediante un ArrayList. Posteriormente deberá mostrar los valores de la lista en pantalla e indicar el tamaño de la lista. Una vez mostrado la lista añadiremos dos nombres más a la lista, que deben coincidir con uno que ya exista. Dichos nombres los guardaremos en la posición 3 y la posición 8 (ejemplo Juan).

En el siguiente paso mostramos de nuevo la lista y a continuación eliminamos los registros que no coinciden con el nombre añadido como repetido (ejemplo Juan)

```
public static void cargarLista(ArrayList milista){
      Scanner sc = new Scanner(System.in);
     // Añadimos 10 Elementos en el ArrayList
      for (int i = 1; i <= 10; i++) {
         System.out.print("Ingrese un nombre " + i + ":");
         milista.add(sc.nextLine());
      }
   }
    public static void mostrarLista(ArrayList milista){
     //Utilizamos el método forEach para leer la lista.
      milista.forEach((nombre) ->
      System.out.println(nombre));
   }
  public static void eliminarNombreDistintos(ArrayList milista,String nombre){
     //Utilizamos un iterator para recorrer el array.
   Iterator<String> miIterator = milista.iterator();
    while(miIterator.hasNext()){
         String elemento = miIterator.next();
         if(!elemento.equals(nombre))
                  miIterator.remove(); // Eliminamos el Elemento
}   }
```

```
public static void main(String[] args) {
    ArrayList<String> lista = new ArrayList<>();
    cargarLista(lista); //método que carga los valores en la lista
    mostrarLista(lista); //método que muestra en pantalla la lista
    System.out.println("El tamaño de la lista es: " + lista.size());
    //Añadir el usuario Juan en la posición 3
    lista.add(3, "Juan");
    //Añadir el usuario Juan en la posición 8
    lista.add(8, "Juan");
    System.out.println("Listado después de añadir los dos nombres.");
    mostrarLista(lista);
    eliminarNombreDistintos(lista, "Juan");
    System.out.println("Listado final después de eliminar nombres de la lista.");
    mostrarLista(lista);
}
```

Actividad Propuesta:

Se desea desarrollar un programa que registre las incidencias detectadas por los inspectores de calidad de una empresa de automóviles.

La información a registrar será la siguiente: Código de Incidencia, Código de Inspector, Descripción de incidencia, Fecha de la Incidencia, Nivel de Error.

Ejemplo de un registro sería: 1234, 001A, “Fallo de colocación luneta trasera”, 01/02/2019, ”Medio”.

Para desarrollar este programa se necesita crear una clase denominada Incidencias que contendrá los atributos y métodos necesarios para controlar y registrar la información. También se necesita crear otra clase denominada ListarIncidencias que contendrá todos los métodos para almacenar la lista de incidencias producidas en la empresa de automóviles (cargar incidencias, listar incidencias, añadir valores a la lista, eliminar valores de la lista, comprobar sí un incidencia existe o no. Para el desarrollo de este programa se utilizará la clase ArrayList.

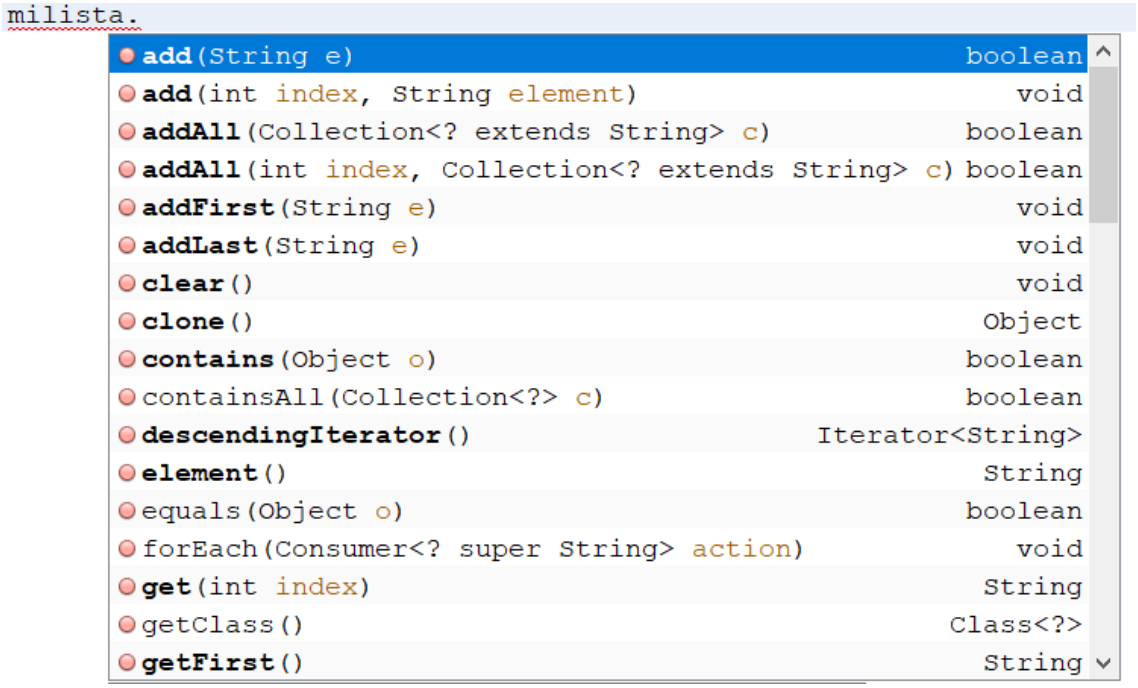
2.1.1 LinkedList.

Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

Empezamos declarando un LinkedList.

```
// Declaración de un LinkedList de "String". Puede ser de cualquier otro elemento un Objeto (float, Boolean, Object, ...)
LinkedList<String> milista= new LinkedList<>();
```

A continuación, vamos a describir alguno de sus métodos.



```
// Añade el elemento al LinkedList
milista.add("Elemento");
// Añade el elemento al LinkedList en la posición 'n'
milista.add(1, "Elemento 2");
//añade un elemento al principio de la lista
milista.addFirst("Elemento 1");
//añade un elemento al final de la lista
milista.addLast("Elemento final");
// Devuelve el numero de elementos del LinkedList
milista.size();
// Devuelve el elemento que esta en la posición '2' del LinkedList
milista.get(2);
// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro
milista.contains("Elemento");
// Devuelve la posición de la primera ocurrencia ('Elemento') en el LinkedList
milista.indexOf("Elemento");
// Devuelve la posición de la última ocurrencia ('Elemento') en el LinkedList
milista.lastIndexOf("Elemento");
// Borra el elemento de la posición '2' del LinkedList
milista.remove(2);
// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.
milista.remove("Elemento");
//Borra todos los elementos de LinkedList
milista.clear();
// Devuelve True si el LinkedList esta vacio. Sino Devuelve False
milista.isEmpty();
// Copiar un LinkedList
LinkedList linkedListCopia = (LinkedList) milista.clone();
```

Ejemplo de una Lista con LinkedList.

```
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
  int num;
 int op;
  LinkedList lista = new LinkedList<>();
 do{
    System.out.println( "\t Menú \t" );
    System.out.println( "Operaciones con listas" );
    System.out.println( "1.- Insertar al principio" );
    System.out.println( "2.- Insertar al final" );
    System.out.println( "3.- Borrar al principio" );
    System.out.println( "4.- Borrar al final" );
    System.out.println( "5.- Mostrar la lista" );
    System.out.println( "6.- Salir" );
    System.out.println( "\n" );
    System.out.println( "Elija la operación que desee" );
```

```
        switch(op){
           case 1:
               System.out.println( "Inserte numero" );
               num = sc.nextInt();
               lista.addFirst(num);
               break;
           case 2:
               System.out.println( "Inserte numero" );
               num = sc.nextInt();
               lista.addLast(num);
               break;
           case 3:
               System.out.println( "Se borrará el primer elemento" );
               lista.removeFirst();
               break;
           case 4:
               System.out.println( "Se borrará el último elemento" );
               lista.removeLast();
               break;
           case 5:
               System.out.println( "La lista es la siguiente" );
                System.out.println(lista.toString());
                break;
           case 6:
               System.out.println( "Gracias" );
               break;
           }
        }

     while( op != 6 );
}
```

ArrayList y LinkedList son diferentes implementaciones de List. Tienen cosas en común así como cosas similares. Las cosas que los distinguen son las siguientes.

ArrayList	**LinkedList**
Es basada en índices	Es basada en nudos
Buscar un elemento es más rápido en ArrayList	Buscar un elemento es más lento en LinkedList
Insertar un elemento es más lento enArrayList	Insertar un elemento es más rápido en LinkedList
Usa menos memoria	Usa más memoria

Actividad Propuesta:

Crear un programa que registre 20 valores ordenados de mayor a menor en una lista con LinkedList. Posteriormente utilizando la clase LinkedList cree un método que busque un valor en la lista y me indique el valor anterior y posterior a dicho valor a buscar.

Actividad Propuesta:

Desarrolla un programa que inserte 1000 registros en un ArrayList y en un LinkedList. Posteriormente calcule el tiempo en que tarda cada una de las listas en insertar un valor en la posición 100;

2.2. Set

La interfaz **Set** define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de **Collection** añadiendo la restricción de que los elementos duplicados están prohibidos. Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**. Para comprobar si dos **Set** son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

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

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

Declaración de la interfaz Set y sus implementaciones.

```
// Un HashSet simple de String
HashSet<String> mihashSet = new HashSet<>();
//Un TreeSet simple de Integer
TreeSet<Integer> mitree = new TreeSet<Integer>();
// LinkedHashSet simple de Double
LinkedHashSet<Double> milhset = new LinkedHashSet<>();
```

A continuación, vamos a describir alguno de sus métodos la interfaz Set.

```
HashSet<String> milista= new HashSet<>();
// Añade el elemento
milista.add("Elemento");
// Devuelve el número de elementos
milista.size();
// Comprueba se existe del elemento ('Elemento') que se le pasa como parametro
milista.contains("Elemento");
// Borra la primera ocurrencia del 'Elemento' que se le pasa como parametro.
milista.remove("Elemento");
//Borra todos los elementos
milista.clear();
// Devuelve True si  está vacío. Sino Devuelve False
milista.isEmpty();
```

Ejemplo con **HashSet**: Lista varias edades, no ordenadas y no repetidas.

```
public static void main(String[] args) {
    HashSet<Integer> edades = new HashSet<>();
    edades.add(12);
    edades.add(43);
    edades.add(100);
    edades.add(80);
    edades.add(12);
    System.out.println("Edades: " + edades);
}
```

Ejemplo con **TreeSet:** Lista varias frutas y ordena las frutas por valor alfabético.

```
public static void main(String[] args) {
    TreeSet<String> frutas = new TreeSet<>();
    frutas.add("Platano");
    frutas.add("Manzana");
    frutas.add("Piña");
    frutas.add("Naranja");
    frutas.add("Albaricoque");
    System.out.println("Fruits Set : " + frutas);
}
```

Ejemplo con **LinkedHashSet:** Lista varias letras, ordenadas según su insercción .

```
public static void main(String[] args) {
    LinkedHashSet<String> alfabeto = new LinkedHashSet<>();
    alfabeto.add("Z");
    alfabeto.add("P");
    alfabeto.add("N");
    alfabeto.add("O");
    alfabeto.add("K");
    alfabeto.add("A");
    System.out.println(alfabeto); }
```

Una vez explicados los distintos tipos de Set, veremos cómo se crean y mostraremos sus diferencias en los tiempos de inserción. Como hemos visto anteriormente, el más rápido

debería ser HashSet mientras que, por otro lado, el más lento debería ser TreeSet. Vamos a comprobarlo con el siguiente código:

```
public static void main(String[] args) {

    HashSet<Integer> hashSet = new HashSet<>();
    Long startHashSetTime = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {   hashSet.add(i);    }
    Long endHashSetTime = System.currentTimeMillis();

    System.out.println("Tiempo: " + (endHashSetTime - startHashSetTime));

    TreeSet<Integer> treeSet = new TreeSet<>();
    Long startTreeSetTime = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) { treeSet.add(i);    }
    Long endTreeSetTime = System.currentTimeMillis();

    System.out.println("Tiempo: " + (endTreeSetTime - startTreeSetTime));

    LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();
    Long startLinkedHashSetTime = System.currentTimeMillis();
    for (int i = 0; i < 1000000; i++) {  linkedHashSet.add(i);   }
    Long endLinkedHashSetTime = System.currentTimeMillis();

    System.out.println("Tiempo: " + (endLinkedHashSetTime - startTreeSetTime));}
```

A continuación, el resultado de los tiempos obtenidos:

Tiempo: 63
Tiempo: 208
Tiempo: 255

Los tiempos obtenidos demuestran que, efectivamente, el tiempo de inserción es menor en HashSet y mayor en TreeSet. Es importante destacar que la inicialización del tamaño inicial del Set a la hora de su creación es importante ya que, en caso de insertar un gran número de elementos, podrían aumentar el número de colisiones y; con ello, el tiempo de inserción.

Actividad Propuesta:

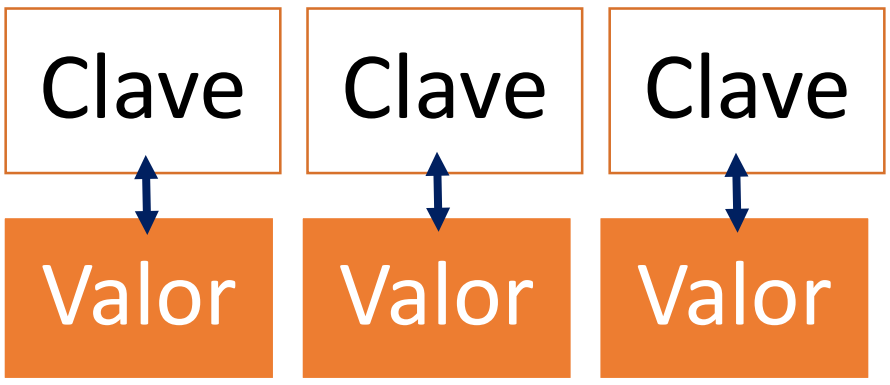
Desarrolla un programa utilizando las implementaciones de la interfaz Set que cree una clase denominada Apartamentos con los atributos cod_apartamento (long), zona(String), y precio (double). Los valores para zona podrán ser "playa","montaña" o "rural". El precio supondremos que es un dato en euros que podrá tomar valores entre 40,00 y 150,00.

Crea una clase con el método main donde se cree un conjunto sin ordenar de 20 apartamentos. El programa nos mostrará por consola este conjunto de apartamentos y nos preguntará en qué zona queremos el apartamento.

Tras ésto el programa creará un conjunto ordenado por precio con los apartamentos cuya zona corresponda con la zona elegida y los mostrará por pantalla.

2.3. Map

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



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

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

A continuación, vamos a ver cómo se crean los distintos tipos de interfaces:

```
// Un HashMap una clave Integer y una lista de String
HashMap<Integer, List<String>> mihashMap = new HashMap<>();
//Un TreeMap una clave Integer y una lista de String
TreeMap<Integer, List<String>> mitreeMap = new TreeMap<>();
// LinkedHashMap una clave Integer y una lista de String
LinkedHashMap<Integer, List<String>> milhashMap = new LinkedHashMap<>();
```

Definición de los métodos utilizados de la interfaz Map:

```
// Declaración de un Map (un HashMap) con clave "Integer" y Valor "String".
//Las claves pueden ser de cualquier tipo de objetos, aunque los más
// utilizados  como clave son los objetos predefinidos de Java como String,
// Integer, Double ...  los Map no permiten datos atómicos
Map<Integer, String> miMap = new HashMap<Integer, String>();
miMap.size(); // Devuelve el número de elementos del Map
miMap.isEmpty();
// Devuelve true si no hay elementos en el Map y false si si los hay
miMap.put(K clave,V valor); // Añade un elemento al Map
miMap.get(K clave);
// Devuelve el valor de la clave que se le pasa como parámetro o 'null'
//si la clave no existe
miMap.clear(); // Borra todos los componentes del Map
miMap.remove(K clave);
// Borra el par clave/valor de la clave que se le pasa como parámetro
miMap.containsKey(K clave);
// Devuelve true si en el map hay una clave que coincide con K
miMap.containsValue(V valor);
// Devuelve true si en el map hay un Valor que coincide con V
miMap.values();
// Devuelve una "Collection" con los valores del Map
```

Ejemplo con **HashMap**: Lista varias características de un vehículo.

```
public static void main(String[] args) {
    HashMap<String, String> mihashMap = new HashMap();
    mihashMap.put("Nombre", "Suzuki");
    mihashMap.put("Potencia", "220");
    mihashMap.put("Tipo", "2-wheeler");
    mihashMap.put("Precio", "85000");
    System.out.println("Elementos del mapa:" + mihashMap);

}
```

Ejemplo con **TreeMap**: Lista varios nombres de futbolistas, no ordenados.

```
public static void main(String[] args) {
TreeMap<Integer, String> mitreeMap = new TreeMap<>();
mitreeMap.put(1, "Casillas");        mitreeMap.put(15, "Ramos");
mitreeMap.put(3, "Pique");           mitreeMap.put(5, "Puyol");
mitreeMap.put(11, "Capdevila");      mitreeMap.put(14, "Xabi Alonso");
mitreeMap.put(16, "Busquets");       mitreeMap.put(8, "Xavi Hernandez");
mitreeMap.put(18, "Pedrito");        mitreeMap.put(6, "Iniesta");
mitreeMap.put(7, "Villa");

// Mostramos el TreeMap con un Iterador que ya hemos instanciado
anteriormente
Iterator iterador = mitreeMap.keySet().iterator();
while(iterador.hasNext()){
  Integer key = (Integer) iterador.next();
  System.out.println("Clave: " + key + " -> Valor: " + mitreeMap.get(key));
}

}
```

Si ejecutamos el algoritmo podemos observar que nos ordena el listado por orden de clave.

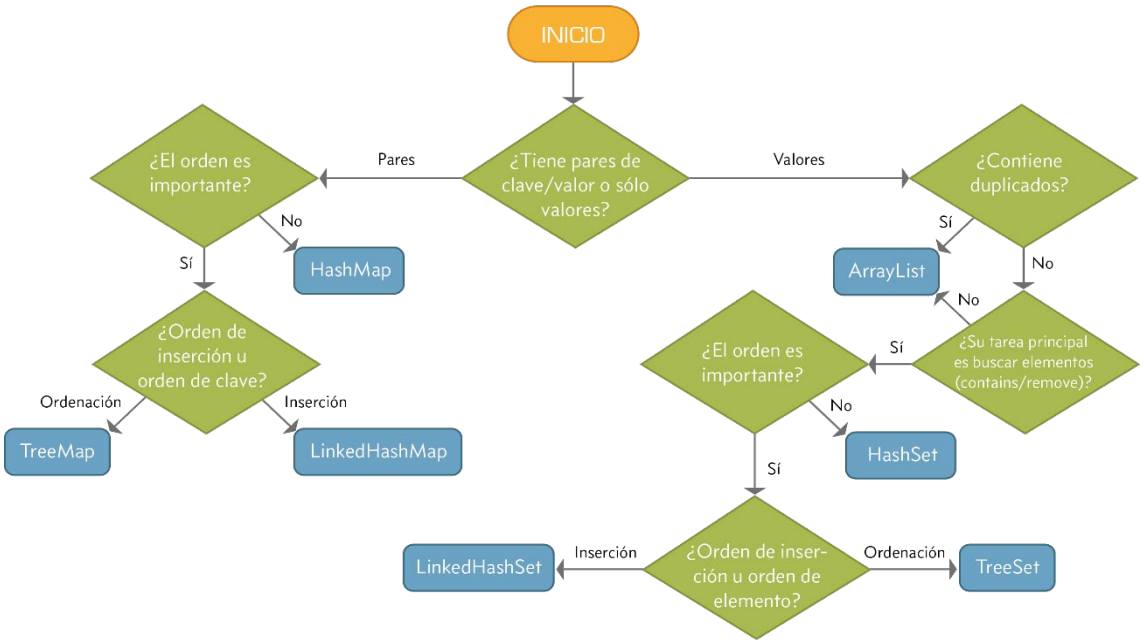
Ejemplo con **LinkedHashMap**: Lista varios nombres de futbolistas, no ordenados.

```
 public static void main(String[] args) {
LinkedHashMap<Integer, String> milinkedHashMap = new LinkedHashMap<>();
milinkedHashMap.put(1, "Casillas");          milinkedHashMap.put(15, "Ramos");
milinkedHashMap.put(3, "Pique");             milinkedHashMap.put(5, "Puyol");
milinkedHashMap.put(11, "Capdevila");        milinkedHashMap.put(14, "Xabi Alonso");
milinkedHashMap.put(16, "Busquets");         milinkedHashMap.put(8, "Xavi
Hernandez");
milinkedHashMap.put(18, "Pedrito");          milinkedHashMap.put(6, "Iniesta");
milinkedHashMap.put(7, "Villa");

// Mostramos el LinkedHashMap con un Iterador que ya hemos instanciado
anteriormente
Iterator iterador = milinkedHashMap.keySet().iterator();
while(iterador.hasNext()){
  Integer key = (Integer) iterador.next();
  System.out.println("Clave: " + key + " -> Valor: " + milinkedHashMap.get(key));
}

}
```

Si ejecutamos el algoritmo podemos observar que nos ordena en el mismo orden de inserción.

El cuándo usar una implementación u otra de Map variará en función de la situación en la que nos encontremos. Generalmente, HashMap será la implementación que usemos en la mayoría de situaciones. HashMap es la implementación con mejor rendimiento (como se ha podido comprobar en el análisis de Set), pero en algunas ocasiones podemos decidir renunciar a este rendimiento a favor de cierta funcionalidad como la ordenación de sus elementos.

Diagrama de decisión para uso de colecciones Java



Actividad Propuesta:

Se desea desarrolla una agenda de contactos personales. Para ello nos solicitan implementar un algoritmo que inserta los valores de los contactos. Dicha información estará compuesta por el nombre del contacto y su teléfono. Para ello utilizaremos la interfaz **Map** con la implementación **TreeMap**. Los métodos a implementar serán: introducir contactos, mostrar contactos, buscar contacto.

Actividad Propuesta:

Una empresa de videojuegos para móviles desea desarrollar el juego de piedra, papel o tijera, que permita jugar a sólo 2 jugadores en n partidas. Para ello nos solicitan implementar un algoritmo que cumpla las siguientes condiciones:

Limitar el número de jugadores a 2 en cada partida.

Registrar en cada partida el valor introducido. Ejemplo jugador 1: piedra; jugador 2: papel. En partida 1

Registrar el ganador de cada partida.

Crear un método para mostrar las n partidas jugadas con su ganador.

Crear un método que indique que jugador ha ganado más partida en la n partidas jugadas.