UNIDAD 6. ESTRUCTURAS DE DATOS INTERNAS. MEMORIA

1º DAW. PROGRAMACIÓN

6.1 CLASES Y MÉTODOS GENÉRICOS

Las clases y los métodos genéricos son un recurso de programación disponible en muchos lenguajes. Su objetivo es facilitar la reutilización del software, creando métodos y clases que puedan trabajar con diferentes tipos de objetos indistintamente.

Clases

Una clase genérica se define utilizando parámetros de tipo. Estos parámetros de tipo actúan como marcadores para los tipos reales que se utilizarán cuando se instancie la clase.

Métodos

Los métodos genéricos son similares a las clases genéricas, pero se aplican a métodos individuales en lugar de a clases enteras. Pueden utilizarse parámetros de tipo en la declaración de un método para hacerlo genérico.

6.1.1 EJEMPLO CLASE GENÉRICA

```
public class Caja<T> {
    private T contenido;

    public void ponerContenido(T contenido) {
        this.contenido = contenido;
    }

    public T obtenerContenido() {
        return contenido;
    }
}
```

En este ejemplo, T es un parámetro de tipo que representa el tipo del contenido de la caja.

6.1.1 EJEMPLO CLASE GENÉRICA

El parámetro de tipo T puede usarse con distintos tipos de datos

```
Caja<Integer> cajaEntero = new Caja<>();
cajaEntero.ponerContenido(42);
System.out.println(cajaEntero.obtenerContenido());  // Imprime 42

Caja<String> cajaTexto = new Caja<>();
cajaTexto.ponerContenido("Hola, mundo");
System.out.println(cajaTexto.obtenerContenido());  // Imprime Hola, mundo
```

6.1.2 EJEMPLO MÉTODO GENÉRICO

```
public class Utilidades {
    public <T> void imprimirArray(T[] array) {
        for (T elemento : array) {
            System.out.print(elemento + " ");
        }
        System.out.println();
    }
}
```

Este método **imprimirArray** es genérico y puede imprimir arrays de cualquier tipo. Puede usarse con arrays de enteros, cadenas u otros tipos

6.1.2 EJEMPLO MÉTODO GENÉRICO

```
Integer[] arrayEnteros = {1, 2, 3, 4, 5};
String[] arrayCadenas = {"uno", "dos", "tres"};


Utilidades util = new Utilidades();
util.imprimirArray(arrayEnteros);  // Imprime 1 2 3 4 5
util.imprimirArray(arrayCadenas);   // Imprime uno dos tres
```

Este método imprime indistintamente los enteros y las cadenas de texto, como así lo haría para cualquier tipo de dato.

En resumen, las clases y métodos genéricos en Java permiten escribir código **más flexible y reutilizable** al trabajar con diferentes tipos de datos sin sacrificar la seguridad de tipo.

6.2 COLECCIONES

Una **colección** representa un grupo de objetos. Estos objetos son conocidos como elementos.

Cuando se quiere trabajar con un conjunto de elementos, es necesario un almacén donde poder guardarlos.

En Java, se emplea la interfaz genérica **Collection** para este propósito que expone una serie de métodos comunes para trabajar con ellas: añadir, eliminar, ver tamaño, etc.

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

6.3 TIPOS DE COLECCIONES

En la plataforma de Java se encuentran por defecto una serie de tipos de colecciones, de los cuales analizaremos los principales y más usados.

6.3.1 CONJUNTOS. 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.

Para comprobar duplicidades, los elementos tienen que tener correctamente implementados los **métodos equals y hashCode.**

Implementaciones de Set:

- **HashSet**
- **TreeSet.** Ordena según valores. Los elementos deben implementar a Comparable.
- **LinkedHashSet.** Ordena según inserción

6.3.2 LISTAS. 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 la mejora en algunas tareas como el acceso posicional a elementos, las búsquedas y la iteración sobre los elementos.

Implementaciones de List:

- **ArrayList.** Es la implementación más típica. Es un array redimensionable.
- **LinkedList.** Mejora la interación entre los elementos en ciertas ocasiones.

6.3.3 FUNCIONES CLAVE-VALOR. MAP

La interfaz Map asocia claves a valores. Esta interfaz no puede contener claves duplicadas y cada clave sólo puede tener asociado como máximo un valor.

Realmente, el conjunto de claves es un Set, por lo que debemos considerarlo como tal.

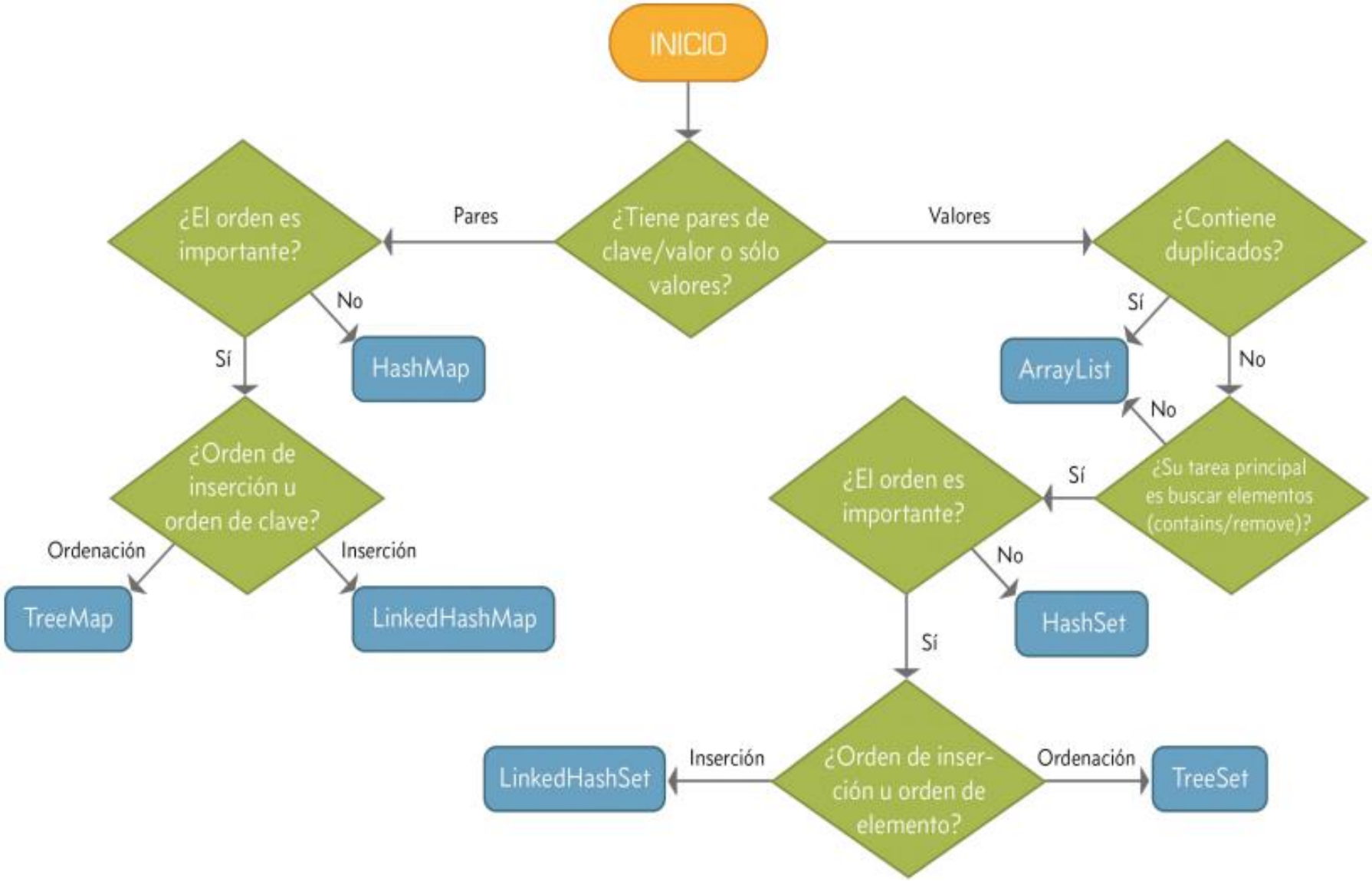
Implementaciones de Map:

- **HashMap.** Almacena las claves en una tabla y no guarda ningún orden.
- **TreeMap.** Almacena las claves ordenándolas según sus valores (Comparable).
- **LinkedHashMap.** Almacena las claves ordenadas en función de su orden de inserción.

6.3.4 CONCLUSIONES

Como hemos visto, existen una variedad de estructuras para almacenar datos. Estas estructuras, ofrecen diversas funcionalidades: ordenación de elementos, mejora de rendimiento, rango de operaciones... Es importante conocer cada una de ellas para saber cuál es la mejor situación para utilizarlas ya que dicha elección influirá en el rendimiento de nuestra aplicación.

6.3.5 DIAGRAMA DE DECISIÓN



6.4 CLASE DE UTILIDAD COLLECTIONS

La clase **Collections** en Java proporciona métodos estáticos que operan con colecciones para realizar operaciones útiles.

Algunos de los métodos más usados son:

- **sort(List<T> list)** Ordena los elementos ascendentemente
- **reverse(List<?> list)** Invierte el orden de los elementos en la lista
- **shuffle(List<?> list)** Reorganiza aleatoriamente los elementos en la lista
- **binarySearch(List<? extends Comparable<? super T>> list, T key)** Realiza una búsqueda binaria en la lista para encontrar la posición de la clave dada
- **addAll(Collection<? super T> c, T... elements)** Agrega los elementos a la colección
- **max(Collection<? extends T> coll)** Devuelve el elemento máximo
- **min(Collection<? extends T> coll)** Devuelve el elemento mínimo
- **frequency(Collection<?> c, Object o)** Devuelve el número de veces que el elemento especificado aparece en la colección
- **replaceAll(List<T> list, T oldVal, T newVal)** Reemplaza todas las ocurrencias del valor antiguo con el nuevo valor en la lista
- **copy(List<? super T> dest, List<? extends T> src)** Copia todos los elementos de la lista fuente en la lista de destino

6.5 ITERADORES

A menudo, será útil recorrer las colecciones.

Los **iteradores** son un mecanismo que nos permite recorrer todos los elementos de una colección de forma secuencial, sencilla y segura.

Los iteradores permiten recorrer las colecciones de dos formas: bucles foreach y a través de un bucle normal usando el iterador.

Los iteradores se crean invocando al método **iterator() de la interfaz Iterable.**

La clase Iterable expone los métodos:

- **hasNext().** Devuelve si existen más elementos
- **next().** Devuelve el siguiente elemento
- **remove().** Borra de la colección el elemento actual