# Colecciones

## 1. ¿QUÉ ES UNA COLECCIÓN?.

Una colección es un objeto que representa un conjunto de elementos. Estos elementos pueden ser de cualquier tipo de objeto. Las colecciones se utilizan para almacenar, recuperar y manipular conjuntos de datos en una aplicación Java.

En Java, se emplea la interfaz genérica **Collection,** que se encuentra en el paquete **java.util**. Recordemos que las interfaces definen una serie de métodos que tienen que tener las clases que las implementen pero no dicen cómo se implementan. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser:

- **boolean *add(E elemento)***: agrega un elemento a la colección.
- **boolean *remove(E elemento)***: elimina un elemento de la colección.
- **int size():** devuelve el número de elementos en la colección.
- **boolean isEmpty():** devuelve true si la colección no contiene elementos.
- **boolean contains(Object objeto):** devuelve true si la colección contiene el elemento especificado.
- **iterator():** devuelve un iterador sobre los elementos en la colección.

Además tiene métodos para combinar colecciones:

- boolean **addAll**(Collection<?> c) : Añade todos los elementos de la colección c.
- boolean **removeAll**(Collection<?> c): Elimina todos los elementos de la colección c.
- boolean **containsAll**(Collection<?> c): Comprueba si coinciden las colecciones.
- boolean **retainAll**(Collection<?> c): Elimina todos los elementos a no ser que estén en c. (obtiene la intersección).

Partiendo de la interfaz genérica Collection extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior. A su vez estas interfaces serán implementadas por las clases con las que vamos a trabajar, ver Figura 1.

NOTA: La interfaz **Iterable** define un único método, iterator(), que devuelve un iterador sobre los elementos de la colección. Esta interfaz es implementada por cualquier clase que permita que sus objetos sean iterados en un bucle "for-each". Esto significa que cualquier clase que implemente la interfaz Iterable puede ser utilizada en un bucle "for-each" en Java, es decir, podremos usar este tipo de bucles cuando trabajamos con las colecciones.

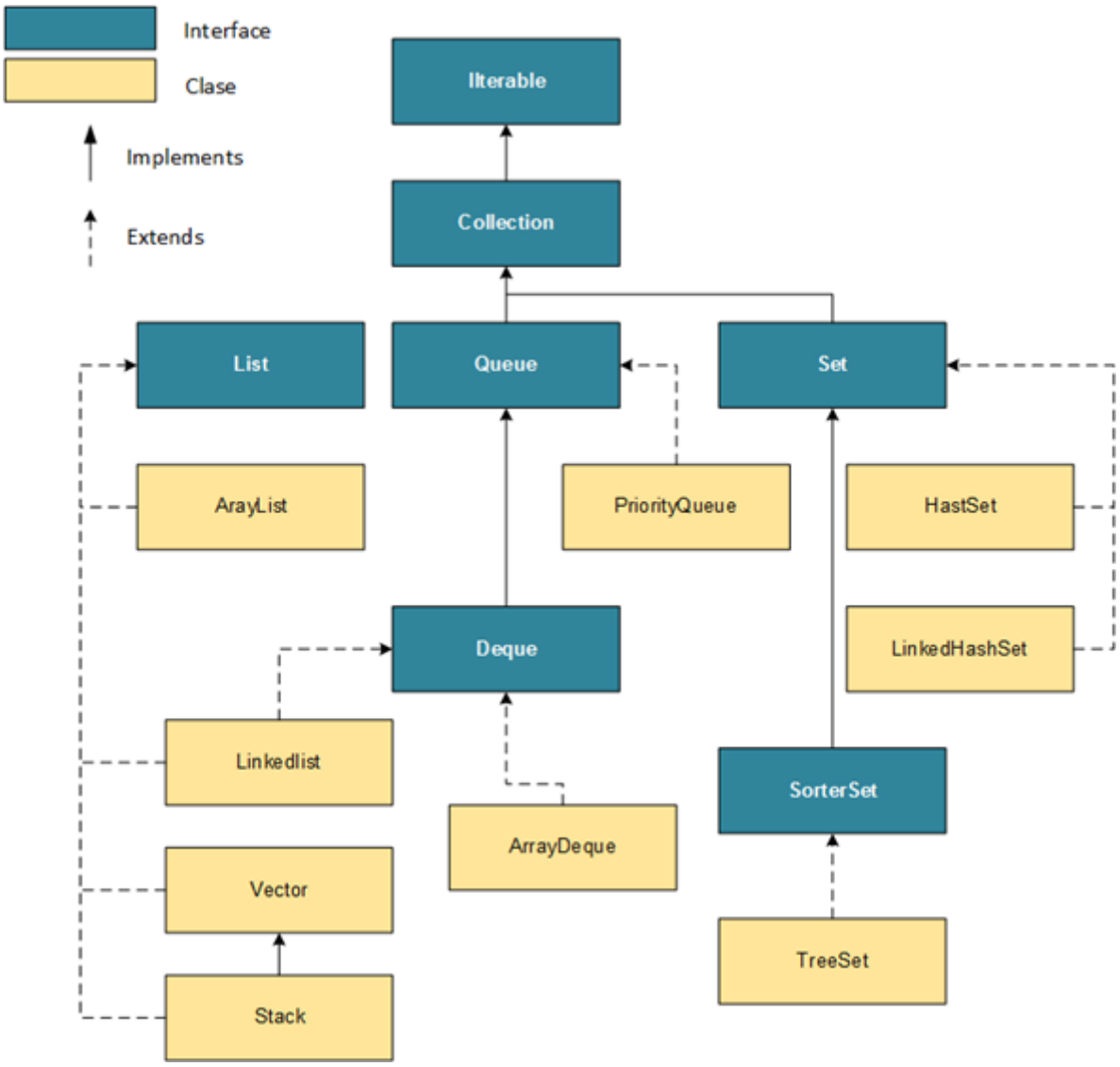


Figura 1.

Como observamos en la Figura 1 hay tres interfaces que extienden la interfaz Collection. A continuación vamos a ver cada una de ellas más detalladamente, así como las clases que las implementan.

## 2. LIST (LISTAS)

La interfaz **List** es una colección ordenada de manera secuencial, cada elemento tendrá su índice, además esta interfaz permite elementos duplicados. Es decir, puede tener múltiples elementos iguales y se puede acceder a ellos a través de un índice entero.

Los métodos que implementa esta interfaz son (algunos son heredados de colletion):

- Métodos de inserción:
  - **boolean add(objeto):** Añade un nuevo elemento al final de la lista.
  - **boolean add(int indice,objeto):** Inserta un nuevo elemento en una posición. El elemento que estaba en esta posición y los siguientes se moverán un índice hacia atrás.
  - **objeto set(int indice, objeto):** Pone un nuevo elemento en la posición indicada. Devuelve el elemento que se encontraba en dicha posición anteriormente, es decir, "machaca" lo que había antes.
- Métodos de eliminación::
  - **boolean remove(objeto):** Elimina la primera ocurrencia del elemento indicado.
  - **objeto remove(int indice):** Elimina el elemento de la posición indicada y devuelve el objeto eliminado.
- **objeto get(int indice)**: Devuelve el elemento en la posición especificada.
- **boolean contains(objeto):** Comprueba si el elemento especificado está en la colección.
- **int indexOf(objeto):** Primera posición en la que se encuentra un elemento; -1 si no está.
- **int lastIndexOf(objeto):** Última posición del elemento especificado; o -1 si no está.
- **void clear():** Elimina todos los elementos de la colección.
- **int size():** Devuelve el número de elementos en la colección.
- **boolean isEmpty():** Comprueba si la colección está vacía.

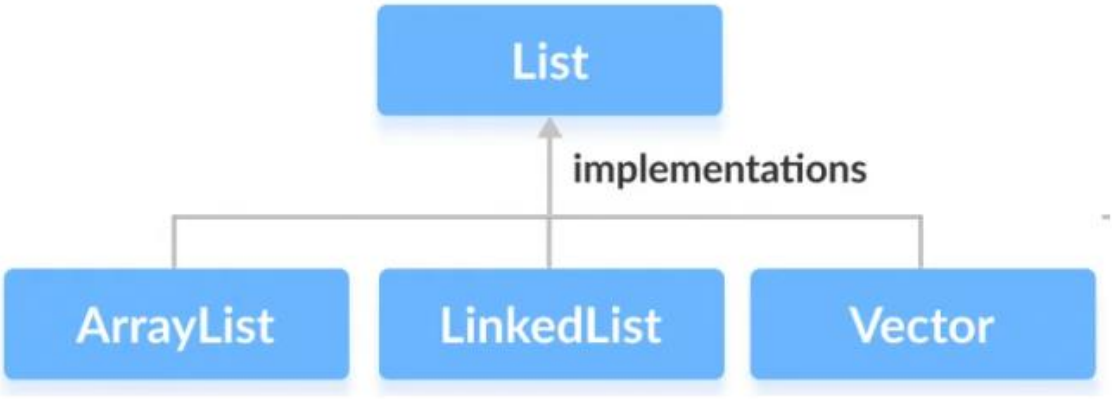


Figura 2

En la figura 2 podemos ver las clases que implementan la interfaz List. Tanto ArrayList como LinkedList implementan la interfaz List en Java, lo que significa que comparten muchos métodos comunes, como add(), remove(), get(), set(), size(), clear(), entre otros. Recordemos que List es una interfaz que dice que métodos hay y qué hacen pero no cómo, de manera que su funcionamiento será diferente dependiendo de la clase que usemos.

Además estas clases tienen unos métodos exclusivos que veremos más adelante.

## 2.1 ArrayList.

Almacena los elementos en un array dinámico de tamaño variable, lo que significa que los elementos están ubicados en memoria de forma contigua. Para acceder a un elemento en una posición específica, ArrayList puede acceder directamente a esa posición en memoria utilizando un índice. Esto hace que ArrayList sea más rápido para el acceso aleatorio y la modificación de elementos en la lista, pero menos eficiente para la inserción y eliminación de elementos en posiciones intermedias de la lista, ya que puede requerir la reubicación de los elementos en la lista. (ver figura 3)

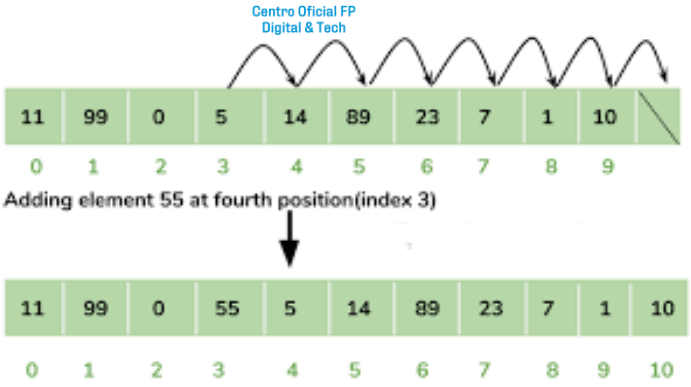


Figura 3

Para crear un objeto de esta clase haremos:

```
ArrayList<TipoDeDato> nombreLista = new ArrayList<TipoDeDato>();

ArrayList<TipoDeDato> nombres = new ArrayList<>();    //Desde Java 7
```

Veamos un ejemplo de esto:

```
import java.util.ArrayList;

public class EjemploArrayList {

    public static void main(String[] args) {

        ArrayList<String> lista = new ArrayList<>();

        // Agregar elementos a la lista
        lista.add("Juan");
        lista.add("María");
        lista.add("Pedro");
        lista.add("Lucía");

        // Obtener el tamaño de la lista
        int tamano = lista.size();
        System.out.println("Tamaño de la lista: " + tamano);

        // Obtener un elemento de la lista por su índice
        String elemento = lista.get(2);
        System.out.println("Elemento en la posición 2: " + elemento);
```

```
        // Quitar un elemento de la lista
        lista.remove(1);
        System.out.println("Lista después de remover el elemento en la posición 1: " + lista);

        // Verificar si la lista contiene un elemento determinado
        boolean contiene = lista.contains("Lucía");
        System.out.println("¿La lista contiene a Lucía? " + contiene);

        // Limpiar la lista
        lista.clear();
        System.out.println("Lista después de limpiarla: " + lista);

        // Verificar si la lista está vacía
        boolean vacia = lista.isEmpty();
        System.out.println("¿La lista está vacía? " + vacia);
    }
}
```

## 2.2 LinkedList.

Almacena los elementos en una lista doblemente enlazada, lo que significa que cada elemento tiene un enlace a su predecesor y sucesor. Para acceder a un elemento en una posición específica, LinkedList debe recorrer la lista enlazada desde el inicio o el final hasta la posición deseada. Esto hace que LinkedList sea más lento para el acceso aleatorio y la modificación de elementos en la lista, pero más eficiente para la inserción y eliminación de elementos en cualquier posición de la lista, ya que solo se requiere la modificación de los enlaces. (ver figura 4)

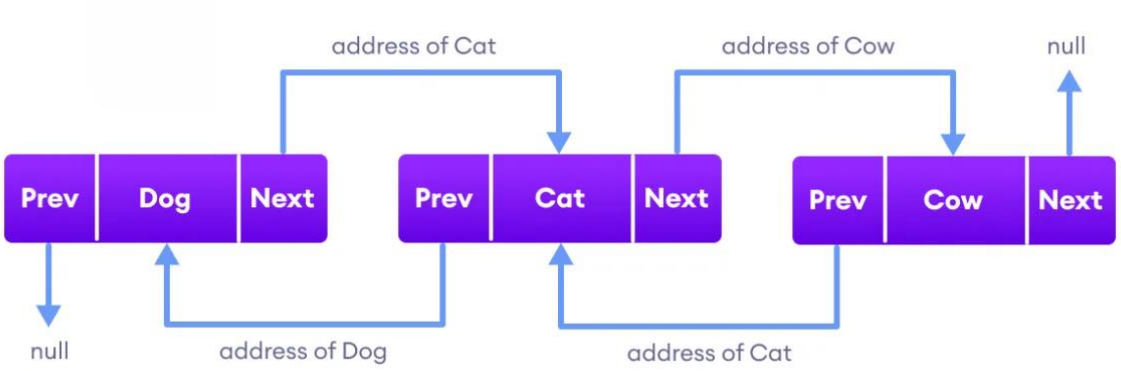


Figura 4

Para crear un objeto de esta clase haremos:

```
LinkedList<TipoDeDato> nombreLista = new LinkedList<TipoDeDato>();

LinkedList<TipoDeDato> nombres = new LinkedList<>();    //Desde Java 7
```

```
import java.util.LinkedList;

public class EjemploLinkedList {
    public static void main(String[] args) {
        LinkedList<String> nombres = new LinkedList<>();

        nombres.add("Ana");
        nombres.add("Juan");
        nombres.add("María");
        nombres.add("Luis");

        System.out.println("Lista de nombres: " + nombres);

        // Inserta "Pedro" en la posición 2
        nombres.add(2, "Pedro");
        System.out.println("Lista después de insertar a Pedro: " + nombres);

      // Elimina el elemento en la posición 1
         nombres.remove(1);
    System.out.println("Lista después de eliminar el segundo elemento: " +
nombres);

         // Reemplaza el elemento en la posición 3 por "Carlos"
         nombres.set(3, "Carlos");
      System.out.println("Lista después de reemplazar el cuarto elemento: " +
nombres);
```

```
        // Verifica si la lista contiene "María"
        if( nombres.contains("María")){
                System.out.println("La lista contiene a María);
        else{
          System.out.println("La lista NO  contiene a María);
        }

      // Obtiene el índice del elemento "Luis"
      int indiceLuis = nombres.indexOf("Luis");
      System.out.println("El índice de Luis es: " + indiceLuis);
    }
}
```

## 2.3 Vector.

Es una clase antigua, ya obsoleta, es similar a ArrayList, pero es sincronizada, lo que significa que es segura para su uso en entornos multi-hilo. Sin embargo, debido a su naturaleza sincronizada, puede ser menos eficiente que ArrayList en algunos casos de uso.

# 3. QUEUE (COLAS)

La interfaz **Queue** representa una estructura de datos de cola ordenada en la que los elementos se agregan al final y se eliminan del principio. Esta estructura de datos se basa en el principio de "primero en entrar, primero en salir" (FIFO). En estas estructuras podrá haber o no elementos repetidos dependiendo de la clase que estemos usando.

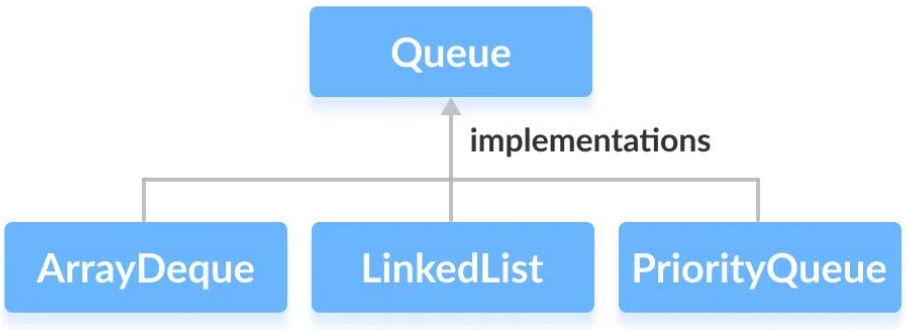


Figura 5

La interfaz Queue presenta los siguientes métodos:

- Métodos de inserción:
  - **void add(E e):**Agrega un elemento al final de la cola. Si la cola está llena, lanzará una excepción.
  - **boolean offer(E e)**: Agrega un elemento al final de la cola. Si la cola está llena, devuelve false.

- Métodos de eliminación:
  - **Element remove()**: Retira y devuelve el elemento al principio de la cola. Si la cola está vacía, lanzará una excepción..
  - **Element poll()**: Retira y devuelve el elemento al principio de la cola. Si la cola está vacía, devuelve null.

- Métodos de consulta:
  - **Element element()**: Devuelve el elemento al principio de la cola sin retirarlo. Si la cola está vacía, lanzará una excepción.
  - **Element peek()**: Devuelve el elemento al principio de la cola sin retirarlo. Si la cola está vacía, devuelve null.

## 3.1 LinkedList.

La clase LinkedList implementa tanto Queue como List por lo que se puede usar con los métodos de ambos. Además si nos fijamos en la figura 1 vemos que LinkedList implementa también DeQueue.

DeQueue hace referencia a una cola de doble extremo (double-ended queue), es decir, que se puede insertar y retirar por ambos extremos.

Deque

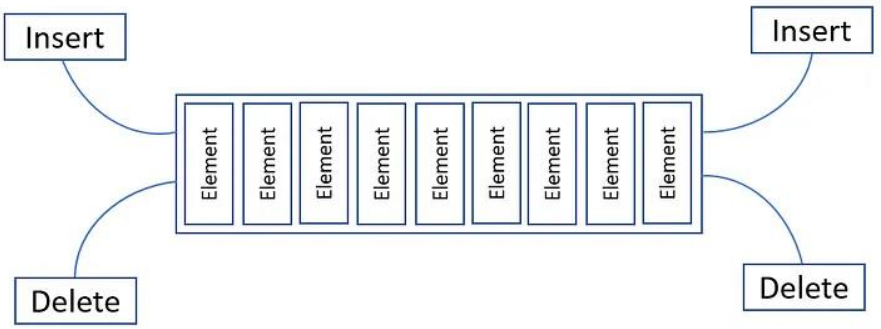


Figura 6

Al implementar DeQueue LinkedList presenta los siguientes métodos:

- Métodos de inserción:
  - **void addFirst(E e)**: Agrega un elemento al principio de la cola. Si la cola está llena, lanzará una excepción.
  - **void addLast(E e):**Agrega un elemento al final de la cola. Si la cola está llena, lanzará una excepción.
  - **boolean offerFirst(E e)**: Agrega un elemento al principio de la cola. Si la cola está llena, devuelve false.
  - **boolean offerLast(E e)**: Agrega un elemento al final de la cola. Si la cola está llena, devuelve false.

- Métodos de eliminación:
  - **Element removeFirst()**: Retira y devuelve el elemento al principio de la cola. Si la cola está vacía, lanzará una excepción.
  - **Element removeLast()**: Retira y devuelve el elemento al final de la cola. Si la cola está vacía, lanzará una excepción.
  - **Element pollFirst()**: Retira y devuelve el elemento al principio de la cola. Si la cola está vacía, devuelve null.
  - **Element pollLast()**: Retira y devuelve el elemento al final de la cola. Si la cola está vacía, devuelve null.

- Métodos de consulta:
  - **Element elementFirst()**: Devuelve el elemento al principio de la cola sin retirarlo. Si la cola está vacía, lanzará una excepción.
  - **Element elementLast()**: Devuelve el elemento al final de la cola sin retirarlo. Si la cola está vacía, lanzará una excepción.

- **Element peekFirst()**: Devuelve el elemento al principio de la cola sin retirarlo. Si la cola está vacía, devuelve null.
- **Element peekLast()**: Devuelve el elemento al final de la cola sin retirarlo. Si la cola está vacía, devuelve null.

Como se puede observar LinkedList tendrá métodos implementados de Queu y de DeQueue que hagan lo mismo como add() y addLast() o offer() y offerFirst(). Podemos usar ambos, si bien cuando LinkedList se usa como una cola FIFO se usan los métodos de Queue y cuando es una cola de doble extremo los de DeQueue dejando de forma clara si estamos trabajando con el principio o el final.

Un ejemplo de LinkedList usada como cola de doble extremo:

```
import java.util.LinkedList;


public class EjemploLinkedList {
    public static void main(String[] args) {

        LinkedList<String> nombres = new LinkedList<>();


        // Agregamos elementos a la lista
        nombres.addLast("Juan");
        nombres.addLast("María");
        nombres.addLast("Pedro");
        nombres.addLast("Ana");


       // Imprimimos la lista.
        System.out.println("Lista con nuevo elemento al principio: " + nombres);



        // Agregamos un elemento al principio de la lista
        nombres.addFirst("Luis");


        // Eliminamos el ultimo elemento de la lista
        nombres.removeLast();


        // Imprimimos la lista nuevamente para ver que cambio
        System.out.println("Lista con el primer elemento eliminado: " + nombres);


        // Obtenemos y eliminamos el primer elemento de la lista
```

```
        String ultimoElemento = nombres.pollFirst();

        // Imprimimos la lista y el elemento sacado
        System.out.println("Lista con el último elemento eliminado: " + nombres);
        System.out.println("Último elemento eliminado: " + ultimoElemento);
    }
}
```

## 3.2 PriorityQueue.

PriorityQueue se trata de una cola donde los elementos no se ordenan por "llegada" sino por una prioridad establecida. Esta prioridad, sino se especifica, viene dada por el orden natural de las clases, por ejemplo, si es Integer se ordenará de menor a mayor, si es String se ordenará alfabéticamente.

```
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new PriorityQueue<>();
        numbers.add(5);
        numbers.add(3);
        numbers.add(8);
        numbers.add(1);
        numbers.add(2);

        // Remueve y devuelve el primer elemento de la cola. No se usa un foreach porque no saldría ordenado, es necesario hacerlo con esta estructura.
        while (!numbers.isEmpty()) {
            System.out.println(numbers.poll());
        }
    }
}
```

Pero, ¿qué ocurre si tenemos una clase, por ejemplo Cliente, donde tenemos los atributos de nombre, apellidos, codigoCliente? ¿la ordenamos por código de cliente? ¿por nombre?. En ese caso haremos que la clase Cliente implemente la interfaz **Comparable** y sobreescriba el método **compareTo(Elemento elemento).** Este método debe devolver:

- un **número negativo** si el objeto actual es "**menor**" que el objeto pasado como argumento.
- **cero** si son "**iguales**".
- un **número positivo** si el objeto actual es "**mayor**".

Veamos esto, por ejemplo ordenar alfabéticamente por nombre:

```
public class Cliente implements Comparable<Cliente> {
    private String nombre;
    private String apellidos;
    private int codigoCliente;

    public Cliente(String nombre, String apellidos, int codigoCliente) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.codigoCliente = codigoCliente;
    }

    public String getNombre() {
        return nombre;
    }

    public String getApellidos() {
        return apellidos;
    }

    public int getCodigoCliente() {
        return codigoCliente;
    }

    @Override
    public int compareTo(Cliente otroCliente) {
        return this.nombre.compareTo(otroCliente.nombre);
    }

    @Override
    public String toString() {
```

```
        return "Cliente{" +
                "nombre='" + nombre + '\'' +
                ", apellidos='" + apellidos + '\'' +
                ", codigoCliente=" + codigoCliente +
                '}';
    }
}
```

Luego ya podemos usar esto, por ejemplo:

```
import java.util.PriorityQueue;

public class GestorClientes {
    private PriorityQueue<Cliente> colaClientes = new PriorityQueue<>();

    public GestorClientes() {
    }

    public void agregarCliente(Cliente cliente) {
        colaClientes.offer(cliente);
    }

    public void imprimirClientes() {
        while (!colaClientes.isEmpty()) {
            System.out.println(colaClientes.poll());
        }
    }
}
```

Si quisiéramos ordenar por el código de cliente tendremos que modificar el método compareTo() de la siguiente manera:

```
 @Override
    public int compareTo(Cliente otroCliente) {
```

```
    return
this.codigoCliente.compareTo(otroCliente.codigoCliente);
  }
```

Ten en cuenta que para hacer esto codigoCliente debe ser definido como Integer para que sea una clase y tenga un método compareTo(). Si fuera un int podríamos usar el método de la siguiente manera:

```
@Override
public int compareTo(Tarea otra) {
  return Integer.compare(this.prioridad, otra.prioridad);
}
```

## 4. SET

La interfaz **Set** es una colección no ordenada que no permite elementos duplicados, es decir, solo puede tener un elemento de cada tipo. Se usa para trabajar con conjuntos y, como tal, tiene métodos para realizar operaciones de conjunto como la unión, la intersección y la diferencia.

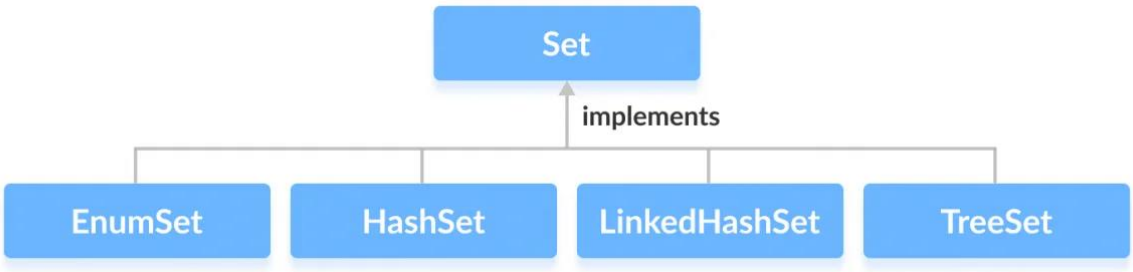


Figura 6

Como se aprecia en la figura 6 hay varias clases en java que implementan la interfaz Set: HashSet, TreeSet y LinkedHashSet.

HashSet es la implementación más común y se basa en la tabla hash (asociación de llaves y claves). Es muy eficiente para realizar operaciones de inserción, eliminación y búsqueda de elementos. Sin embargo, los elementos no se almacenan en ningún orden específico.

TreeSet, por otro lado, se basa en un árbol y almacena los elementos en orden ascendente o descendente. Esto hace que sea más adecuado para aplicaciones que requieren que los elementos estén ordenados.

LinkedHashSet es una implementación que mantiene el orden de inserción de los elementos, lo que significa que los elementos se mantienen en el orden en que se agregaron al conjunto.

EnumSet se utiliza para almacenar elementos de un tipo enumerado específico. Es muy eficiente en términos de memoria y velocidad, y es ideal para conjuntos grandes de elementos enumerados.

## 5. RECORRER COLECCIONES

Para recorrer las colecciones se usa la estructura del bucle for-each, este permite recorrer cada elemento de una colección sin necesidad de usar índices ni iteradores manualmente.

Su estructura es la siguiente:

```
for (Tipo elemento : coleccion) {
   // instrucciones usando  el objeto elemento
}
```

Veamos un ejemplo a continuación:

```
import java.util.ArrayList;

public class Ejemplo {
   public static void main(String[] args) {
      ArrayList<String> frutas = new ArrayList<>();
      frutas.add("Manzana");
      frutas.add("Banana");
      frutas.add("Cereza");

      for (String fruta : frutas) {
         System.out.println(fruta);
      }
   }
}
```

El problema que presente este tipo de bucles es que la colección no puede ser modificada en el propio bucle, es decir, no podemos usar métodos como add() o remove(), ya que lanzaría la excepción java.util.ConcurrentModificationException.

Entonces, si necesitamos borrar o añadir un elemento ¿cómo se hace? pues para ello podemos crear un objeto Iterator, que recorra la colección y usar su método remove()

```
public class BorrarConIterator {
    public static void main(String[] args) {
        ArrayList<String> nombres = new ArrayList<>();
        nombres.add("Ana");
        nombres.add("Luis");
        nombres.add("Pedro");
        nombres.add("Luis");


        // Eliminamos todos los elementos que sean "Luis"
        Iterator<String> it = nombres.iterator();
        while (it.hasNext()) {
            String nombre = it.next();
            if (nombre.equals("Luis")) {
                it.remove(); //  Se hace remove sobre el iterador, no sobre la colección
            }
        }


        System.out.println("Lista final: " + nombres);
    }
}
```

Como vemos un Iterador no es más que un objeto que te permite recorrer los elementos de una colección uno a uno, de forma segura y controlada. Viene de la interfaz del paquete `java.util` que define métodos para recorrer estructuras de datos como listas, conjuntos, etc., sin exponer su estructura interna.

La interfaz Iterator<E> incluye tres métodos clave:

- hasNext(): Devuelve true si hay otro elemento por iterar.
- next(): Devuelve el siguiente elemento.
- remove():Elimina el último elemento devuelto por next() (opcional).

Como vemos no sirve para añadir elementos mientras se recorre el bucle. Si queremos hacer esto podemos buscar la posición, guardar el indice y posteriormente, fuera del bucle, hacer el add().