Anexo II: Estructuras de datos.

Índice de contenidos

1 Listas.

Las listas en *Java* **son flexibles y se utilizan ampliamente debido a su capacidad para manejar colecciones de elementos** con diferentes necesidades de acceso y manipulación. **Se implementan principalmente mediante la interfaz `List`** de la biblioteca estándar.

Las listas **mantienen el orden en el que se añaden los elementos**. Se puede acceder a los elementos por su índice y los elementos se pueden reordenar usando el método `sort()`.

A diferencia de los conjuntos (`Set`), las listas **permiten elementos duplicados**. Se puede agregar el mismo elemento varias veces.

Las listas **pueden cambiar de tamaño dinámicamente**. No se necesita especificar un tamaño fijo al crear una lista.

Existen varias implementaciones comunes de esta interfaz `List` en la **colección `java.util`**. Las implementaciones más usadas son:

- **`ArrayList`**: es una implementación de lista basada en un **array dinámico**. Es muy **eficiente para acceder a los elementos por índice**.

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();

        lista.add("Elemento 1");
        lista.add("Elemento 2");
        lista.add("Elemento 3");
        System.out.println("Lista: " + lista);
    }
}
```

- **`LinkedList`**: es una implementación de lista basada en una **lista doblemente enlazada**. Es más **eficiente para operaciones de inserción y eliminación** en comparación con `ArrayList`.

```java
import java.util.LinkedList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new LinkedList<>();

        lista.add("Elemento 1");
        lista.add("Elemento 2");
        lista.add("Elemento 3");
        System.out.println("Lista: " + lista);
    }
}
```

- **`Vector`**: es una implementación de **lista sincronizada**, lo que significa que **es segura para el uso concurrente**. Sin embargo, en la mayoría de los casos, se recomienda usar `ArrayList` en lugar de `Vector` debido a que `ArrayList` es generalmente más eficiente.

```java
import java.util.Vector;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new Vector<>();

        lista.add("Elemento 1");
        lista.add("Elemento 2");
        lista.add("Elemento 3");
        System.out.println("Lista: " + lista);
    }
}
```

La elección entre `ArrayList` y `LinkedList` dependerá de las operaciones que se planeen realizar con mayor frecuencia en la aplicación. Si se necesita un acceso rápido por índice y no se realizan muchas inserciones o eliminaciones en el medio de la lista, `ArrayList` podría ser más apropiado. Por otro lado, si las operaciones de inserción y eliminación son frecuentes o si se necesita mantener un orden específico, `LinkedList` podría ser la mejor opción.

Tabla que resume los **constructores y operaciones más comunes** para las principales implementaciones de listas:

Operación	ArrayList	LinkedList	Vector
Constructor vacío	`ArrayList<>()`	`LinkedList<>()`	`Vector<>()`
Constructor con capacidad inicial	`ArrayList<>(int initialCapacity)`	`LinkedList<>()` (no admite capacidad inicial)	`Vector<>(int initialCapacity)`
Constructor con colección	`ArrayList<>(Collection<? extends E> c)`	`LinkedList<>(Collection<? extends E> c)`	`Vector<>(Collection<? extends E> c)`
Agregar elemento al final	`boolean add(E e)`	`boolean add(E e)`	`boolean add(E e)`
Acceder a elemento por índice	`E get(int index)`	`E get(int index)`	`E get(int index)`
Modificar elemento en una posición específica	`E set(int index, E element)`	`E set(int index, E element)`	`E set(int index, E element)`
Eliminar elemento por índice	`E remove(int index)`	`E remove(int index)`	`E remove(int index)`
Eliminar primer aparición de elemento	`boolean remove(Object o)`	`boolean remove(Object o)`	`boolean remove(Object o)`
Tamaño de la lista	`int size()`	`int size()`	`int size()`
Verificar si está vacía	`boolean isEmpty()`	`boolean isEmpty()`	`boolean isEmpty()`
Iterar sobre la lista	`void forEach(Consumer<? super E> action)`	`void forEach(Consumer<? super E> action)`	`void forEach(Consumer<? super E> action)`
Agregar elemento en una posición específica	`void add(int index, E element)`	`void add(int index, E element)`	`void add(int index, E element)`
Obtener sublista	`List<E> subList(int fromIndex, int toIndex)`	`List<E> subList(int fromIndex, int toIndex)`	`List<E> subList(int fromIndex, int toIndex)`
Agregar elementos en el inicio	No es específico, requiere usar `addFirst(E e)`	`void addFirst(E e)`	No es específico, requiere usar `addElement(E e)`
Eliminar el primer elemento	No es específico, requiere usar `remove(0)`	`E removeFirst()`	No es específico, requiere usar `remove(0)`
Eliminar el último elemento	`E remove(size() - 1)`	`E removeLast()`	`E remove(size() - 1)`
Obtener el primer elemento	No es específico, requiere usar `get(0)`	`E getFirst()`	No es específico, requiere usar `firstElement()`
Obtener el último elemento	`E get(size() - 1)`	`E getLast()`	`E lastElement()`
Eliminar todos los elementos	`void clear()`	`void clear()`	`void clear()`
Convertir a array	`Object[] toArray()`	`Object[] toArray()`	`Object[] toArray()`
Convertir a array genérico	`T[] toArray(T[] a)`	`T[] toArray(T[] a)`	`T[] toArray(T[] a)`
Índice de primer aparición de elemento (-1 si no lo encuentra)	`int indexOf(Object o)`	`int indexOf(Object o)`	`int indexOf(Object o)`
Contiene un elemento	`boolean contains(Object o)`	`boolean contains(Object o)`	`boolean contains(Object o)`

El método **`Collections.sort()`** se utiliza para **ordenar elementos en una lista**. El método `sort()` está disponible en la clase `Collections` para ordenar listas que implementan la interfaz `List`, como `ArrayList`, `LinkedList` y `Vector`. Detalles del método `sort()`:

- `Collections.sort(List<T> list)`:
  - Ordena la lista `list` en orden ascendente según el orden natural de los elementos.
  - Requiere que los elementos de la lista implementen la interfaz `Comparable` (por ejemplo, `Integer`, `String`).
- `Collections.sort(List<T> list, Comparator<? super T> c)`:
  - Ordena la lista `list` según el comparador proporcionado `c`.
  - Permite especificar un criterio de ordenación personalizado mediante un `Comparator`.

El método **`Arrays.asList()`** se utiliza para **convertir un array en una lista fija** (no modificable en cuanto al tamaño) **que implementa la interfaz `List`**. Sintaxis:

```
List<T> list = Arrays.asList(array);
```

Como parámetro acepta un array o una lista de elementos separados por comas y devuelve una lista respaldada por el array original. La lista devuelta tiene un tamaño fijo; **no se pueden añadir o eliminar elementos, aunque sí modificar los elementos existentes**.

Ejemplos de **uso de los constructores de `ArrayList`, `LinkedList` y `Vector`**:

Tipo Lista	Constructor	Ejemplo
ArrayList	Constructor vacío	`ArrayList<String> list1 = new ArrayList<>();`
	Constructor con capacidad inicial	`ArrayList<Integer> list2 = new ArrayList<>(10);`
	Constructor con colección	`ArrayList<Double> list3 = new ArrayList<>(Arrays.asList(1.1, 2.2, 3.3));`
	Constructor con clase específica	`ArrayList<MyClass> list4 = new ArrayList<>();`
LinkedList	Constructor vacío	`LinkedList<String> list1 = new LinkedList<>();`
	Constructor con colección	`LinkedList<Integer> list2 = new LinkedList<>(Arrays.asList(10, 20, 30));`
	Constructor con clase específica	`LinkedList<MyClass> list3 = new LinkedList<>();`
Vector	Constructor vacío	`Vector<String> vector1 = new Vector<>();`
	Constructor con capacidad inicial	`Vector<Integer> vector2 = new Vector<>(5);`
	Constructor con colección	`Vector<Double> vector3 = new Vector<>(Arrays.asList(1.5, 2.5, 3.5));`
	Constructor con clase específica	`Vector<MyClass> vector4 = new Vector<>();`

Recorrer una lista se puede hacer de varias maneras dependiendo del propósito y del tipo de lista que se esté

utilizando. Aquí se proponen varios ejemplos de cómo recorrer una lista utilizando `ArrayList`, pero los mismos métodos aplican a `LinkedList` y `Vector`:

1. Uso de un **bucle `for` tradicional** (con índices): esta forma es útil cuando se necesita el índice del elemento mientras se recorre la lista.

```
import java.util.ArrayList;
import java.util.List;

public class ForLoopExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        // Recorrer con un bucle for tradicional
        for (int i = 0; i < list.size(); i++)
            System.out.println("Elemento en índice " + i + ": " + list.get(i));
    }
}
```

2. Uso de un **bucle `for-each`**: es más conciso y se usa cuando no se necesita el índice, solo se quiere acceder a cada elemento.

```
import java.util.ArrayList;
import java.util.List;

public class ForEachExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("X");
        list.add("Y");
        list.add("Z");

        // Recorrer con un bucle for-each
        for (String element : list)
            System.out.println("Elemento: " + element);
    }
}
```

3. Uso de un **`Iterator`**: proporciona más control sobre el recorrido, incluyendo la capacidad de eliminar elementos de la lista de manera segura durante la iteración.

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorExample {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(30);

        // Recorrer con un Iterator
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()) {
            Integer element = iterator.next();
            System.out.println("Elemento: " + element);
        }
    }
}
```

4. Uso de **`ListIterator`** (para listas bidireccionales): permite iterar en ambas direcciones y modificar la lista durante el recorrido.

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Uno");
        list.add("Dos");
        list.add("Tres");
```

```
        // Recorrer con un ListIterator
        ListIterator<String> listIterator = list.listIterator();
        while (listIterator.hasNext()) {
            String element = listIterator.next();
            System.out.println("Elemento: " + element);
        }
    }
}
```

5. Uso de **forEach con expresiones lambda**: más moderno y conciso para recorrer y aplicar acciones a los elementos.

```
import java.util.ArrayList;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<Double> list = new ArrayList<>();
        list.add(1.1);
        list.add(2.2);
        list.add(3.3);

        // Recorrer con for-each y expresiones lambda
        list.forEach(element -> System.out.println("Elemento: " + element));
    }
}
```

Ejemplo:

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<>();

        // Agregar elementos
        lista.add("Manzana");
        lista.add("Banana");
        lista.add("Cereza");
        String primerElemento = lista.get(0); // Acceder a un elemento
        lista.set(1, "Mango"); // Modificar un elemento
        lista.remove("Cereza"); // Eliminar un elemento
        int tamaño = lista.size(); // Tamaño de la lista
        System.out.println("Lista: " + lista); // Imprimir la lista
    }
}
```

Ejemplo:

```
import java.util.Arrays;
import java.util.List;

public class ArraysAsListExample {
    public static void main(String[] args) {
        // Modificar elementos dentro de la lista
        String[] array = {"A", "B", "C"};
        List<String> list = Arrays.asList(array);

        list.set(1, "D"); // Modificar el segundo elemento
        System.out.println("Lista modificada: " + list);
        System.out.println("Array modificado: " + Arrays.toString(array)); // Refleja cambios
    }
}
```

Ejemplo: uso con un comparador personalizado.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Crear una lista de cadenas
        List<String> lista = new ArrayList<>();
        lista.add("banana");
        lista.add("apple");
        lista.add("orange");
        lista.add("kiwi");

        // Ordenar la lista en orden descendente usando un comparador personalizado
```

```
        Collections.sort(lista, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s2.compareTo(s1); // Orden descendente
            }
        });
        System.out.println("Lista ordenada: " + lista); // Imprimir la lista ordenada
    }
}
```

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html, https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html y https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Vector.html

2 Conjuntos.

Los conjuntos (o sets) son **colecciones que no permiten elementos duplicados y no mantienen un orden específico de los elementos**. Java proporciona la **interfaz `Set`**, que es parte del **paquete `java.util`**, y varias implementaciones:

- **`HashSet`**: es la implementación más común de `Set`. No garantiza ningún orden de los elementos.

```
import java.util.HashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> frutas = new HashSet<>();
        frutas.add("Manzana");
        frutas.add("Banana");
        frutas.add("Pera");
        frutas.add("Manzana"); // Intento de agregar un duplicado

        System.out.println(frutas); // Salida: [Manzana, Banana, Pera] (sin orden específico)
    }
}
```

- **`LinkedHashSet`**: mantiene el orden de inserción de los elementos. Es útil si se necesita que los elementos aparezcan en el orden en el que fueron añadidos.

```
import java.util.LinkedHashSet;
import java.util.Set;

public class Main {
    public static void main(String[] args) {
        Set<String> frutas = new LinkedHashSet<>();
        frutas.add("Manzana");
        frutas.add("Banana");
        frutas.add("Pera");

        System.out.println(frutas); // Salida: [Manzana, Banana, Pera] (mantiene el orden)
    }
}
```

- **`TreeSet`**: mantiene los elementos ordenados de forma natural (o por un comparador personalizado). Es más lento que `HashSet` y `LinkedHashSet`, pero útil cuando el orden es importante.

```
import java.util.Set;
import java.util.TreeSet;

public class Main {
    public static void main(String[] args) {
        Set<String> frutas = new TreeSet<>();
        frutas.add("Manzana");
        frutas.add("Banana");
        frutas.add("Pera");

        System.out.println(frutas); // Salida: [Banana, Manzana, Pera] (orden natural)
    }
}
```

La elección entre `HashSet`, `LinkedHashSet` y `TreeSet` depende principalmente de los requisitos de la aplicación en cuanto a la ordenación, rendimiento y uso de memoria. Elegir **`HashSet` cuando el rendimiento sea la prioridad y no importe el orden de los elementos**, ya que es rápido y eficiente en memoria. Optar por **`LinkedHashSet` si se necesita mantener el orden de inserción con un rendimiento similar a `HashSet`** pero con un ligero costo adicional de memoria. Usar **`TreeSet` cuando se requiera que los elementos estén ordenados de forma natural o específica, aunque es más lento y consume más memoria** debido a su estructura de árbol.

Tabla que resume los **constructores y operaciones más comunes** para las principales implementaciones de conjuntos:

Característica	HashSet	LinkedHashSet	TreeSet
Constructor por defecto	`HashSet<>()`	`LinkedHashSet<>()`	`TreeSet<>()`
Constructor con capacidad inicial	`HashSet(int initialCapacity)`	`LinkedHashSet(int initialCapacity)`	`TreeSet(Comparator<? super E> comparator)`
Constructor con capacidad inicial y factor de carga	`HashSet(int initialCapacity, float loadFactor)`	`LinkedHashSet(int initialCapacity, float loadFactor)`	`TreeSet(Collection<? extends E> c)`
Añadir elemento	`boolean add(E e)`	`boolean add(E e)`	`boolean add(E e)`
Eliminar elemento	`boolean remove(Object o)`	`boolean remove(Object o)`	`boolean remove(Object o)`
Contar elementos	`int size()`	`int size()`	`int size()`
Verificar existencia	`boolean contains(Object o)`	`boolean contains(Object o)`	`boolean contains(Object o)`
Vaciar conjunto	`void clear()`	`void clear()`	`void clear()`
Verificar vacío	`boolean isEmpty()`	`boolean isEmpty()`	`boolean isEmpty()`
Convertir a arreglo	`Object[] toArray()`	`Object[] toArray()`	`Object[] toArray()`
Convertir a arreglo con tipo específico	`<T> T[] toArray(T[] a)`	`<T> T[] toArray(T[] a)`	`<T> T[] toArray(T[] a)`
Eliminar todos los elementos especificados	`boolean removeAll(Collection<?> c)`	`boolean removeAll(Collection<?> c)`	`boolean removeAll(Collection<?> c)`
Añadir todos los elementos especificados	`boolean addAll(Collection<? extends E> c)`	`boolean addAll(Collection<? extends E> c)`	`boolean addAll(Collection<? extends E> c)`
Ordenar elementos	No	Mantiene orden de inserción	Ordenado (natural o por comparador)
Iterar sobre elementos	`Iterator<E> iterator()`	`Iterator<E> iterator()`	`Iterator<E> iterator()`
Operaciones de rango	No	No	`NavigableSet<E> subSet(E fromElement, E toElement), NavigableSet<E> headSet(E toElement), NavigableSet<E> tailSet(E fromElement)`
Convertir a lista y ordenar	`List<E> lista = new ArrayList<>(set); Collections.sort(lista);`	`List<E> lista = new ArrayList<>(set); Collections.sort(lista);`	`List<E> lista = new ArrayList<>(set); Collections.sort(lista);`
Convertir de arreglo a conjunto	`Set<E> set = new HashSet<>(Arrays.asList(array));`	`Set<E> set = new LinkedHashSet<>(Arrays.asList(array));`	`Set<E> set = new TreeSet<>(Arrays.asList(array));`
Añadir todos los elementos especificados	`boolean addAll(Collection<? extends E> c)`	`boolean addAll(Collection<? extends E> c)`	`boolean addAll(Collection<? extends E> c)`
Obtener primer elemento	No	No	`E first()`
Eliminar y obtener primer elemento	No	No	`E pollFirst()`
Obtener último elemento	No	No	`E last()`
Eliminar y obtener último elemento	No	No	`E pollLast()`

Para ordenar los elementos de un conjunto, primero se debe convertirlo en una lista, ordenar la lista y, si es necesario, volver a convertirla en un conjunto. Para convertir un arreglo a un conjunto, se puede utilizar `Arrays.asList` y luego construir el conjunto a partir de la lista resultante.

Recorrer un conjunto y una lista es conceptualmente similar y se puede hacer utilizando un bucle `for-each`, un iterador o una expresión lambda con `forEach`. La principal diferencia radica en el orden y la capacidad de acceso por índice (las listas permiten el acceso a elementos por índice, mientras que los conjuntos no lo permiten). Formas:

1. Uso de un **bucle `for-each`**:

```java
Set<String> conjunto = new HashSet<>(Arrays.asList("Plátano", "Manzana", "Pera"));
for (String fruta : conjunto)
    System.out.println(fruta);
```

2. Uso de un **iterador**:

```java
Set<String> conjunto = new HashSet<>(Arrays.asList("Plátano", "Manzana", "Pera"));
Iterator<String> iterador = conjunto.iterator();
while (iterador.hasNext())
    System.out.println(iterador.next());
```

3. Uso de **`forEach` y expresiones lambda**:

```java
Set<String> conjunto = new HashSet<>(Arrays.asList("Plátano", "Manzana", "Pera"));
```

```
        conjunto.forEach(fruta -> System.out.println(fruta));
```

Ejemplo: utiliza `HashSet`, pero puede adaptarse fácilmente para `LinkedHashSet` o `TreeSet`.

```
import java.util.*;
public class SetEjemplo {
    public static void main(String[] args) {
        // Crear un conjunto
        Set<String> frutas = new HashSet<>();

        // Añadir elementos
        frutas.add("Manzana");
        frutas.add("Plátano");
        frutas.add("Pera");
        frutas.add("Naranja");

        // Intentar añadir un duplicado
        boolean added = frutas.add("Manzana");
        System.out.println("Se añadió 'Manzana': " + added); // Salida: false

        // Eliminar un elemento
        boolean removed = frutas.remove("Plátano");
        System.out.println("Se eliminó 'Plátano': " + removed); // Salida: true

        // Verificar si el conjunto contiene un elemento
        boolean contains = frutas.contains("Pera");
        System.out.println("El conjunto contiene 'Pera': " + contains); // Salida: true

        // Contar el número de elementos
        int size = frutas.size();
        System.out.println("Número de elementos en el conjunto: " + size); // Salida: 3

        // Verificar si el conjunto está vacío
        boolean isEmpty = frutas.isEmpty();
        System.out.println("El conjunto está vacío: " + isEmpty); // Salida: false

        // Convertir el conjunto a un arreglo
        Object[] frutasArray = frutas.toArray();
        System.out.println("Elementos en el arreglo:");
        for (Object fruta : frutasArray)
            System.out.println(fruta);

        // Convertir el conjunto a un arreglo con tipo específico
        String[] frutasArrayTyped = frutas.toArray(new String[0]);
        System.out.println("Elementos en el arreglo con tipo específico:");
        for (String fruta : frutasArrayTyped)
            System.out.println(fruta);

        // Eliminar todos los elementos de otro conjunto
        Set<String> frutasParaEliminar = new HashSet<>(Arrays.asList("Pera", "Naranja"));
        boolean removedAll = frutas.removeAll(frutasParaEliminar);
        System.out.println("Se eliminaron todos los elementos especificados: " + removedAll); // Salida: true

        // Iterar sobre los elementos del conjunto usando for-each
        System.out.println("Elementos en el conjunto después de eliminar:");
        for (String fruta : frutas)
            System.out.println(fruta);

        // Iterar sobre los elementos del conjunto usando un iterador
        Iterator<String> iterador = frutas.iterator();
        System.out.println("Elementos en el conjunto usando un iterador:");
        while (iterador.hasNext())
            System.out.println(iterador.next());

        // Añadir elementos nuevamente y ordenar usando lista (solo para HashSet)
        frutas.add("Manzana");
        frutas.add("Plátano");
        frutas.add("Pera");
        List<String> lista = new ArrayList<>(frutas);
        Collections.sort(lista);
        System.out.println("Elementos ordenados:");
        for (String fruta : lista)
            System.out.println(fruta);

        // Vaciar el conjunto
        frutas.clear();
        System.out.println("El conjunto está vacío después de clear(): " + frutas.isEmpty()); // Salida: true
    }
}
```

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html, https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html y https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedHashSet.html

3 Mapas.

Un mapa es una **estructura de datos que almacena pares clave-valor**. La idea es que cada clave única en el mapa está asociada con un valor específico. Los mapas **permiten acceder a los valores de manera rápida usando las claves**, en lugar de tener que buscar en una lista o en otro tipo de colección.

La mayoría de las implementaciones de la **interfaz `Map`** (como `HashMap`) no garantizan ningún orden en el almacenamiento de las claves y los valores. Sin embargo, algunas implementaciones (como `LinkedHashMap` y `TreeMap`) mantienen un orden específico.

Implementaciones en *Java*:

- **`HashMap`**: basado en una tabla hash, es rápido en operaciones de inserción, eliminación y búsqueda. No mantiene un orden específico.

```
import java.util.HashMap;
import java.util.Map;

public class EjemploHashMap {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<>();

        // Añadir pares clave-valor
        hashMap.put("Uno", 1);
        hashMap.put("Dos", 2);
        hashMap.put("Tres", 3);

        // Obtener un valor
        System.out.println("Valor para 'Dos': " + hashMap.get("Dos"));

        // Iterar sobre las entradas
        for (Map.Entry<String, Integer> entrada : hashMap.entrySet())
            System.out.println(entrada.getKey() + ": " + entrada.getValue());
    }
}
```

- **`LinkedHashMap`**: extiende `HashMap` y mantiene un orden de inserción o el orden de acceso a las claves.

```
import java.util.LinkedHashMap;
import java.util.Map;

public class EjemploLinkedHashMap {
    public static void main(String[] args) {
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

        // Añadir pares clave-valor
        linkedHashMap.put("Uno", 1);
        linkedHashMap.put("Dos", 2);
        linkedHashMap.put("Tres", 3);

        // Obtener un valor
        System.out.println("Valor para 'Dos': " + linkedHashMap.get("Dos"));

        // Iterar sobre las entradas
        for (Map.Entry<String, Integer> entrada : linkedHashMap.entrySet())
            System.out.println(entrada.getKey() + ": " + entrada.getValue());
    }
}
```

- **`TreeMap`**: basado en un árbol binario auto-balanceado, mantiene las claves ordenadas de acuerdo con su orden natural o un comparador proporcionado.

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

public class EjemploTreeMap {
    public static void main(String[] args) {
        Map<String, Integer> treeMap = new TreeMap<>();
```

```
        // Añadir pares clave-valor
        treeMap.put("Tres", 3);
        treeMap.put("Uno", 1);
        treeMap.put("Dos", 2);

        // Obtener un valor
        System.out.println("Valor para 'Dos': " + treeMap.get("Dos"));

        // Iterar sobre las entradas
        for (Map.Entry<String, Integer> entrada : treeMap.entrySet())
            System.out.println(entrada.getKey() + ": " + entrada.getValue());
    }
}
```

Elegir **HashMap cuando se necesite una implementación rápida y no importe el orden de las claves**, ya que ofrece rendimiento óptimo en operaciones de inserción, eliminación y búsqueda sin mantener ningún orden específico. Optar por **LinkedHashMap si se necesita preservar el orden de inserción de las claves o el orden de acceso**, ya que mantiene el orden en que se añadieron los elementos, lo que es útil para iteraciones predecibles. Usar **TreeMap cuando se requiera que las claves estén ordenadas de manera natural o personalizada mediante un comparador**, ya que garantiza el orden de las claves y facilita la navegación por un rango de claves ordenadas, aunque puede ser más lento en comparación con HashMap y LinkedHashMap.

Tabla que resume los **constructores y operaciones más comunes** para las principales implementaciones de mapas:

Operación/Constructor	HashMap	LinkedHashMap	TreeMap
Constructor por defecto	HashMap<>()	LinkedHashMap<>()	TreeMap<>()
Constructor con capacidad inicial	HashMap(int initialCapacity)	LinkedHashMap(int initialCapacity)	TreeMap(Comparator<? super K> comparator)
Constructor con capacidad y factor de carga	HashMap(int initialCapacity, float loadFactor)	LinkedHashMap(int initialCapacity, float loadFactor)	TreeMap(Map<? extends K, ? extends V> m)
Constructor copiando elementos de otro mapa	HashMap(Map<? extends K, ? extends V> m)	LinkedHashMap(Map<? extends K, ? extends V> m)	TreeMap(Map<? extends K, ? extends V> m)
Añadir elemento	V put(K key, V value)	V put(K key, V value)	V put(K key, V value)
Obtener valor	V get(Object key)	V get(Object key)	V get(Object key)
Eliminar elemento	V remove(Object key)	V remove(Object key)	V remove(Object key)
Contar elementos	int size()	int size()	int size()
Verificar existencia de clave	boolean containsKey(Object key)	boolean containsKey(Object key)	boolean containsKey(Object key)
Verificar existencia de valor	boolean containsValue(Object value)	boolean containsValue(Object value)	boolean containsValue(Object value)
Iterar sobre claves	Set<K> keySet()	Set<K> keySet()	Set<K> keySet()
Iterar sobre valores	Collection<V> values()	Collection<V> values()	Collection<V> values()
Iterar sobre entradas	Set<Map.Entry<K, V>> entrySet()	Set<Map.Entry<K, V>> entrySet()	Set<Map.Entry<K, V>> entrySet()
Verificar si está vacío	boolean isEmpty()	boolean isEmpty()	boolean isEmpty()
Limpiar mapa	void clear()	void clear()	void clear()

Métodos más comunes para **recorrer un Map**:

1. **Iterar sobre las claves**: para recorrer las claves del mapa, se puede usar el método keySet() que devuelve un conjunto (Set) de todas las claves.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class RecorrerMapas {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Uno", 1);
        map.put("Dos", 2);
        map.put("Tres", 3);

        // Iterar sobre las claves
        Set<String> claves = map.keySet();
        for (String clave : claves)
            System.out.println("Clave: " + clave + ", Valor: " + map.get(clave));
    }
}
```

2. **Iterar sobre los valores**: para recorrer los valores del mapa, se puede usar el método values() que devuelve una colección (Collection) de todos los valores.

```
import java.util.HashMap;
```

```
import java.util.Map;
import java.util.Collection;

public class RecorrerMapas {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Uno", 1);
        map.put("Dos", 2);
        map.put("Tres", 3);

        // Iterar sobre los valores
        Collection<Integer> valores = map.values();
        for (Integer valor : valores)
            System.out.println("Valor: " + valor);
    }
}
```

3. **Iterar sobre las entradas**: para recorrer las entradas (pares clave-valor) del mapa, puedes usar el método `entrySet()` que devuelve un conjunto (`Set`) de entradas (`Map.Entry`).

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class RecorrerMapas {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Uno", 1);
        map.put("Dos", 2);
        map.put("Tres", 3);

        // Iterar sobre las entradas
        Set<Map.Entry<String, Integer>> entradas = map.entrySet();
        for (Map.Entry<String, Integer> entrada : entradas)
            System.out.println("Clave: " + entrada.getKey() + ", Valor: " + entrada.getValue());
    }
}
```

4. **Usando un iterador**: también puede usarse un `Iterator` para recorrer un `Map`. Esto puede ser útil si se necesita controlar más detalladamente la iteración.

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;

public class RecorrerMapas {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Uno", 1);
        map.put("Dos", 2);
        map.put("Tres", 3);

        // Iterar usando un iterador sobre las entradas
        Iterator<Entry<String, Integer>> iterator = map.entrySet().iterator();
        while (iterator.hasNext()) {
            Entry<String, Integer> entry = iterator.next();
            System.out.println("Clave: " + entry.getKey() + ", Valor: " + entry.getValue());
        }
    }
}
```

Ejemplo: `HashMap` donde las claves sean `String` y los valores sean objetos de la clase `Persona`.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

class Persona {
    private String nombre;
    private int edad;

    // Constructor
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Métodos getter
```

```
    public String getNombre() {
        return nombre;
    }

    public int getEdad() {
        return edad;
    }

    // Método toString para imprimir la información de la persona
    @Override
    public String toString() {
        return "Nombre: " + nombre + ", Edad: " + edad;
    }
}

public class EjemploMapConObjetos {
    public static void main(String[] args) {
        // Crear un HashMap con String como clave y Persona como valor
        Map<String, Persona> mapaPersonas = new HashMap<>();

        // Agregar elementos al mapa
        mapaPersonas.put("ID001", new Persona("Alice", 30));
        mapaPersonas.put("ID002", new Persona("Bob", 25));
        mapaPersonas.put("ID003", new Persona("Charlie", 35));

        // Iterar sobre las claves
        System.out.println("Iterando sobre las claves:");
        for (String id : mapaPersonas.keySet()) {
            Persona persona = mapaPersonas.get(id);
            System.out.println("Clave: " + id + ", " + persona);
        }

        // Iterar sobre los valores
        System.out.println("\nIterando sobre los valores:");
        for (Persona persona : mapaPersonas.values())
            System.out.println(persona);

        // Iterar sobre las entradas
        System.out.println("\nIterando sobre las entradas:");
        for (Map.Entry<String, Persona> entry : mapaPersonas.entrySet()) {
            String id = entry.getKey();
            Persona persona = entry.getValue();
            System.out.println("Clave: " + id + ", " + persona);
        }

        // Verificar si el mapa está vacío
        System.out.println("\nEl mapa está vacío? " + mapaPersonas.isEmpty());

        // Contar el número de elementos
        System.out.println("Número de elementos en el mapa: " + mapaPersonas.size());

        // Limpiar el mapa
        mapaPersonas.clear();
        System.out.println("Número de elementos después de limpiar el mapa: " + mapaPersonas.size());
    }
}
```

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html, https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html y https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedHashMap.html

4 Pilas.

Una pila es una **estructura de datos que sigue el principio de Last In, First Out** (LIFO), **es decir, el último elemento en entrar es el primero en salir**. En *Java*, se puede implementar una pila utilizando la **clase `Stack`** de la biblioteca estándar o usando una **implementación propia basada en otras estructuras de datos como `LinkedList`**.

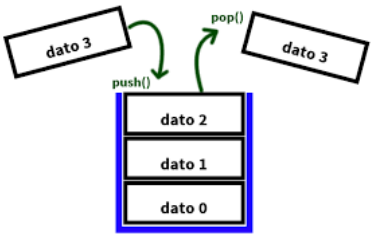

Tabla con los **constructores y operaciones** para la clase `Stack` y la implementación de pila usando `LinkedList`:

Descripción	Stack	LinkedList
Crear una pila vacía	`new Stack<>()`	`new LinkedList<>()`
Añadir un elemento al tope de la pila	`void push(E item)`	`void addFirst(E item)`
Eliminar y devolver el elemento en el tope de la pila	`E pop()`	`E removeFirst()`
Obtener el elemento en el tope de la pila sin eliminarlo	`E peek()`	`E getFirst()`
Comprobar si la pila está vacía	`boolean isEmpty()`	`boolean isEmpty()`
Obtener el número de elementos en la pila	`int size()`	`int size()`

Utilizar la **clase `Stack` cuando se necesite una implementación de pila estándar** con métodos especializados para operaciones LIFO (Last In, First Out). Es ideal para aplicaciones que requieren una pila con una interfaz simple y directa. Optar por **`LinkedList` si se busca una mayor flexibilidad y un rendimiento mejorado** en operaciones de inserción y eliminación, especialmente cuando también podrías necesitar otras funcionalidades de listas, como la manipulación de elementos en ambos extremos. En general, `LinkedList` puede ser preferible cuando la pila se usa en un contexto más amplio donde se requieren capacidades adicionales de lista.

Para **recorrer una pila usando la clase `Stack`, se puede usar un bucle `while` o `for` junto con la operación `pop()`** para eliminar y procesar cada elemento. Ten en cuenta que esto vaciará la pila. **Si se necesita preservar la pila original, puede usarse un `Iterator`**.

Ejemplo:

```
import java.util.Stack;
import java.util.LinkedList;

public class StackAndLinkedListExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        // LinkedList<String> linkedListStack = new LinkedList<>();

        // Añadir elementos a la pila
        stack.push("Elemento 1"); // linkedListStack.addFirst("Elemento 1");
        stack.push("Elemento 2"); // linkedListStack.addFirst("Elemento 2");
        stack.push("Elemento 3"); // linkedListStack.addFirst("Elemento 3");

        // Mostrar el tamaño de la pila
        System.out.println("Tamaño de la pila: " + stack.size());
        // System.out.println("Tamaño de la pila: " + linkedListStack.size());

        // Obtener y mostrar el elemento en la cima de la pila sin eliminarlo
        System.out.println("Elemento en la cima: " + stack.peek());
        // System.out.println("Elemento en la cima: " + linkedListStack.getFirst());

        // Eliminar y mostrar el elemento en la cima de la pila
        System.out.println("Elemento eliminado: " + stack.pop());
        // System.out.println("Elemento eliminado: " + linkedListStack.removeFirst());

        // Verificar si la pila está vacía
        System.out.println("¿Está vacía la pila?: " + stack.isEmpty());
        // System.out.println("¿Está vacía la pila?: " + linkedListStack.isEmpty());

        // Mostrar el tamaño de la pila después de eliminar un elemento
        System.out.println("Tamaño de la pila después de eliminar un elemento: " + stack.size());
        // System.out.println("Tamaño de la pila después de eliminar un elemento: " + linkedListStack.size());
    }
}
```

Ejemplo: recorrer una pila con `pop()`.

```
import java.util.Stack;

public class StackTraversalExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("Elemento 1");
        stack.push("Elemento 2");
        stack.push("Elemento 3");

        // Recorrer la pila usando pop (vaciará la pila)
        System.out.println("Recorriendo pila con pop:");
        while (!stack.isEmpty())
            System.out.println(stack.pop());
    }
}
```

Ejemplo: recorrer una pila con `Iterator` (sin vaciar la pila).

```
import java.util.Stack;
```

```
import java.util.Iterator;

public class StackTraversalExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("Elemento 1");
        stack.push("Elemento 2");
        stack.push("Elemento 3");

        // Recorrer la pila usando un Iterator (sin vaciar la pila)
        System.out.println("Recorriendo pila con Iterator:");
        Iterator<String> iterator = stack.iterator();
        while (iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Stack.html

5 Colas.

Una cola (queue) es una **estructura de datos que sigue el principio FIFO (First In, First Out), donde los elementos se añaden al final de la cola y se eliminan del frente**. Java proporciona varias implementaciones de colas a través de la **interfaz `Queue`**, que pertenece al **paquete `java.util`**. Algunas de las **implementaciones** más comunes son **`LinkedList`**, **`PriorityQueue`** y **`ArrayDeque`**.

Tabla con los **constructores y operaciones básicas** de las principales implementaciones de colas (Queue):

Operación	LinkedList	PriorityQueue	ArrayDeque
Constructores	`LinkedList<>()`	`PriorityQueue<>()`	`ArrayDeque<>()`
	`LinkedList<>(Collection<? extends E> c)`	`PriorityQueue<>(int initialCapacity)`	`ArrayDeque<>(int numElements)`
		`PriorityQueue<>(int initialCapacity, Comparator)`	`ArrayDeque<>(Collection<? extends E> c)`
		`PriorityQueue<>(Collection<? extends E> c)`	
		`PriorityQueue<>(PriorityQueue <? extends E> c)`	
		`PriorityQueue<>(SortedSet<? extends E> c)`	
Añadir elemento	`boolean offer(E e)`	`boolean offer(E e)`	`boolean offer(E e)`
	`boolean add(E e)`	`boolean add(E e)`	`boolean add(E e)`
Remover y devolver primer elemento	`E poll()`	`E poll()`	`E poll()`
	`E remove()`	`E remove()`	`E remove()`
Ver primer elemento sin remover	`E peek()`	`E peek()`	`E peek()`
	`E element()`	`E element()`	`E element()`
Verificar si está vacía	`boolean isEmpty()`	`boolean isEmpty()`	`boolean isEmpty()`
Obtener el tamaño	`int size()`	`int size()`	`int size()`

Para elegir la implementación de cola adecuada en *Java*, considerar el uso y las necesidades específicas de la aplicación: **usar `LinkedList` cuando se necesite una cola o lista doblemente enlazada que permita elementos nulos y requiera operaciones frecuentes de inserción y eliminación en ambos extremos**. **`PriorityQueue` es ideal cuando se necesita una cola que mantenga los elementos en orden**, ya sea natural o definido por un comparador, pero **no permite elementos nulos y es perfecta para gestionar prioridades**, como en algoritmos de búsqueda o planificación. Por último, **`ArrayDeque` es la mejor opción cuando buscas una implementación eficiente de cola o pila sin permitir elementos nulos**, siendo más rápida que `LinkedList` para colas cortas debido a su implementación basada en arrays. Elegir `ArrayDeque` para un rendimiento óptimo en operaciones de inserción y eliminación en los extremos, especialmente en contextos de uso intensivo de memoria.

Recorrer una cola implica iterar sobre sus elementos desde el primero hasta el último sin alterar el orden de los mismos. Las colas pueden ser recorridas usando varios métodos, como un **bucle `for-each`**, un **iterador** o **convirtiendo la cola en una lista**.

Ejemplo: `import java.util.*;`

```
public class QueueTraversal {
    public static void main(String[] args) {
        Queue<Integer> linkedListQueue = new LinkedList<>(); // Ejemplo con LinkedList
        linkedListQueue.offer(1);
        linkedListQueue.offer(2);
        linkedListQueue.offer(3);

        Queue<Integer> priorityQueue = new PriorityQueue<>(); // Ejemplo con PriorityQueue
        priorityQueue.offer(3);
        priorityQueue.offer(1);
        priorityQueue.offer(2);

        Queue<Integer> arrayDequeQueue = new ArrayDeque<>(); // Ejemplo con ArrayDeque
        arrayDequeQueue.offer(10);
        arrayDequeQueue.offer(20);
        arrayDequeQueue.offer(30);

        // Recorrido usando for-each
        System.out.println("Recorrido LinkedList:");
        for (Integer element : linkedListQueue)
            System.out.println(element);

        System.out.println("\nRecorrido PriorityQueue:");
        for (Integer element : priorityQueue)
            System.out.println(element); // Ordenado según prioridad

        System.out.println("\nRecorrido ArrayDeque:");
        for (Integer element : arrayDequeQueue)
            System.out.println(element);

        // Recorrido usando un iterador
        System.out.println("\nRecorrido con Iterator (LinkedList):");
        Iterator<Integer> iterator = linkedListQueue.iterator();
        while (iterator.hasNext())
            System.out.println(iterator.next());
    }
}
```

Ejemplo: sistema de atención al cliente con prioridades, donde los clientes con mayor prioridad (menor valor numérico) se atienden antes.

```
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Comparator;

// Clase Cliente para gestionar la información de cada cliente
class Cliente {
    String nombre;
    int prioridad; // Prioridad: cuanto menor es el número, mayor es la prioridad

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

    @Override
    public String toString() {
        return nombre + " (Prioridad: " + prioridad + ")";
    }
}

public class BancoConPrioridad {
    public static void main(String[] args) {
        // Cola con prioridad usando PriorityQueue
        Queue<Cliente> colaClientes = new PriorityQueue<>(Comparator.comparingInt(c -> c.prioridad));

        // Añadiendo clientes a la cola
        colaClientes.offer(new Cliente("Carlos", 1)); // Cliente VIP con alta prioridad
        colaClientes.offer(new Cliente("María", 2)); // Cliente VIP
        colaClientes.offer(new Cliente("Pedro", 5)); // Cliente regular
        colaClientes.offer(new Cliente("Sofía", 3)); // Cliente regular con prioridad media
        colaClientes.offer(new Cliente("Juan", 4)); // Cliente regular
        colaClientes.offer(new Cliente("Ana", 6)); // Cliente regular con baja prioridad

        // Simulación de atención al cliente
        System.out.println("=== Sistema de Atención al Cliente con Prioridades ===");

        // Atender a todos los clientes según su prioridad
        while (!colaClientes.isEmpty()) {
            Cliente cliente = colaClientes.poll();
            System.out.println("Atendiendo a: " + cliente);
```

```
            }
            // Verificar si la cola está vacía
            System.out.println("\n¿La cola está vacía?: " + colaClientes.isEmpty());
        }
    }
```

Ejemplo: aplicación de uso de una cola en la que los pedidos se procesan en función de su precio.

```
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Comparator;

class Pedido {
    private String producto;
    private double precio;

    public Pedido(String producto, double precio) {
        this.producto = producto;
        this.precio = precio;
    }

    public String getProducto() { return producto; }
    public double getPrecio() { return precio; }
}

class ComparadorPedidos implements Comparator<Pedido> {
    @Override
    public int compare(Pedido pedido1, Pedido pedido2) {
        return Double.compare(pedido1.getPrecio(), pedido2.getPrecio());
    }
}

public class EjemploPriorityQueuePedidos {
    public static void main(String[] args) {
        Queue<Pedido> colaPedidos = new PriorityQueue<>(new ComparadorPedidos());

        colaPedidos.offer(new Pedido("Portátil", 1200));
        colaPedidos.offer(new Pedido("Smartphone", 800));
        colaPedidos.offer(new Pedido("Tablet", 500));
        colaPedidos.offer(new Pedido("Monitor", 250));

        System.out.println("Procesando pedidos por precio:");
        while (!colaPedidos.isEmpty()) {
            Pedido pedido = colaPedidos.poll();
            System.out.println("Pedido: " + pedido.getProducto() + " (Precio: €" + pedido.getPrecio() + ")");
        }
    }
}
```

Más información en https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/PriorityQueue.html y https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayDeque.html