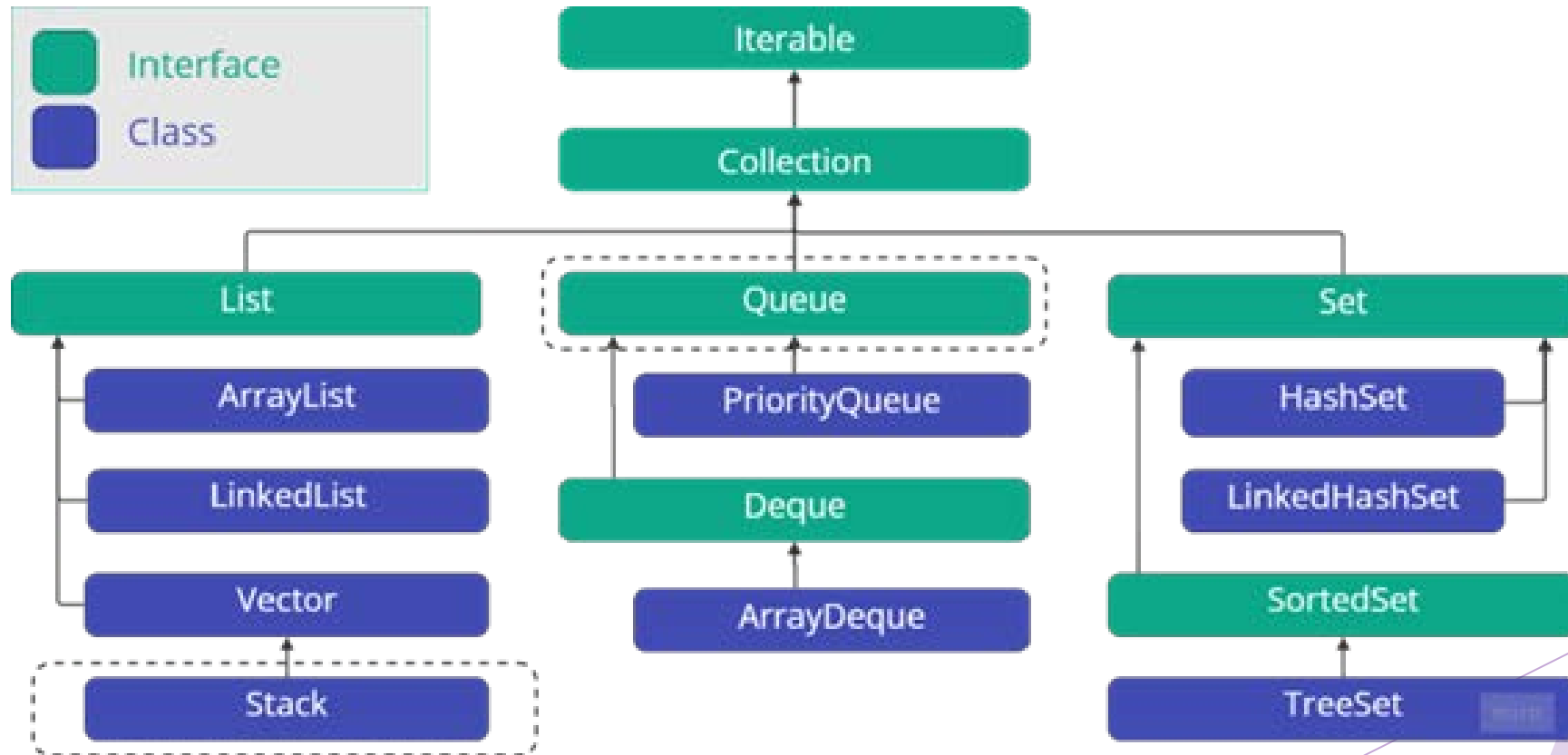


# Estructuras Java

- ▶ Materia Algoritmos y Estructuras de Datos - CB100
- ▶ Cátedra Schmidt
- ▶ Estructuras nativas

Vamos a explorar las estructuras de datos más comunes en Java, como listas, pilas, colas, árboles, y mapas/diccionarios. Te explicaré cómo se usan, cómo insertar elementos, recorrerlos, verificar si contienen un elemento, y eliminarlos, todo siguiendo las mejores prácticas modernas




# 1. Lista (List)

## Características

- Permite elementos duplicados.
- Los elementos tienen un **orden definido**.
- Se implementa comúnmente con:
  - `ArrayList` (basado en un array, rápido para acceso aleatorio).
  - `LinkedList` (basado en nodos enlazados, eficiente para inserciones/eliminaciones).

## Ejemplo básico con ArrayList

java

 Copiar código

```
import java.util.*;

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

        // Insertar elementos
        lista.add("Manzana");
        lista.add("Banana");
        lista.add("Cereza");

        // Recorrer con for-each
        System.out.println("Recorrido con for-each:");
        for (String fruta : lista) {
            System.out.println(fruta);
        }

        // Recorrer con iterador
        System.out.println("Recorrido con Iterator:");
        Iterator<String> iterador = lista.iterator();
        while (iterador.hasNext()) {
            System.out.println(iterador.next());
        }
    }
}
```

```
// Recorrer con iterador
System.out.println("Recorrido con Iterator:");
Iterator<String> iterador = lista.iterator();
while (iterador.hasNext()) {
    System.out.println(iterador.next());
}

// Verificar si contiene un elemento
System.out.println("¿Contiene Banana? " + lista.contains("Banana"));

// Eliminar un elemento
lista.remove("Banana");
System.out.println("Lista después de eliminar Banana: " + lista);
}
}
```


## 2. Pila (Stack)

### Características

- Sigue la regla **LIFO** (último en entrar, primero en salir).
- Modernamente, se usa la clase `Deque` como una mejor alternativa a `Stack`.

### Ejemplo con `ArrayDeque`

java

 Copiar código

```
import java.util.*;

public class PilaEjemplo {
    public static void main(String[] args) {
        Deque<Integer> pila = new ArrayDeque<>();

        // Insertar elementos (push)
        pila.push(10);
        pila.push(20);
        pila.push(30);

        // Ver elemento en la cima (peek)
        System.out.println("Cima de la pila: " + pila.peek());
    }
}
```

```
// Recorrer la pila  
System.out.println("Recorrido de la pila:");  
for (Integer num : pila) {  
    System.out.println(num);  
}  
  
// Eliminar elementos (pop)  
System.out.println("Elemento eliminado: " + pila.pop());  
System.out.println("Pila después del pop: " + pila);  
}  
}
```

### 3. Cola (Queue)

#### Características

- Sigue la regla **FIFO** (primero en entrar, primero en salir).
- Implementaciones comunes:
  - `LinkedList` (como cola básica).
  - `PriorityQueue` (para colas con prioridades).
  - `ArrayDeque` (cola eficiente).

#### Ejemplo con `ArrayDeque`


```
java

import java.util.*;

public class ColaEjemplo {
    public static void main(String[] args) {
        Queue<String> cola = new ArrayDeque<>();

        // Insertar elementos
        cola.offer("A");
        cola.offer("B");
        cola.offer("C");

        // Ver el elemento en la cabeza (peek)
        System.out.println("Elemento en la cabeza: " + cola.peek());
    }
}
```

 Copiar código



```
// Recorrer la cola
System.out.println("Recorrido de la cola:");
for (String letra : cola) {
    System.out.println(letra);
}

// Eliminar elementos (poll)
System.out.println("Elemento eliminado: " + cola.poll());
System.out.println("Cola después del poll: " + cola);
}
}
```

## 4. Árboles (TreeSet)

### Características

- Basado en un árbol binario de búsqueda.
- Los elementos se mantienen ordenados de forma natural o mediante un comparador.
- No permite duplicados.

### Ejemplo con TreeSet

```
java

import java.util.*;

public class ArbolEjemplo {
    public static void main(String[] args) {
        Set<Integer> arbol = new TreeSet<>();

        // Insertar elementos
        arbol.add(50);
        arbol.add(30);
        arbol.add(70);
        arbol.add(10);

        // Recorrer el árbol (ya está ordenado)
        System.out.println("Elementos en orden:");
        for (Integer num : arbol) {
            System.out.println(num);
        }
    }
}
```

 Copiar código

```
// Verificar si contiene un elemento  
System.out.println("¿Contiene 30? " + arbol.contains(30));  
  
// Eliminar un elemento  
arbol.remove(30);  
System.out.println("Árbol después de eliminar 30: " + arbol);  
}  
}
```

---


## 5. Mapas y Diccionarios (HashMap, TreeMap)

### Características

- Almacenan pares clave-valor.
- `HashMap` : No garantiza el orden.
- `TreeMap` : Mantiene las claves ordenadas.
- Operaciones típicas: insertar ( `put` ), obtener ( `get` ), verificar ( `containsKey` ).

### Ejemplo con `HashMap`

java

 Copiar código

```
import java.util.*;

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

        // Insertar pares clave-valor
        mapa.put("Juan", 25);
        mapa.put("Ana", 30);
        mapa.put("Luis", 20);
    }
}
```

```
// Recorrer el mapa
System.out.println("Recorrido del mapa:");
for (Map.Entry<String, Integer> entrada : mapa.entrySet()) {
    System.out.println(entrada.getKey() + " -> " + entrada.getValue());
}

// Verificar si contiene una clave
System.out.println("¿Contiene clave 'Ana'? " + mapa.containsKey("Ana"));

// Obtener un valor
System.out.println("Valor asociado a 'Luis': " + mapa.get("Luis"));

// Eliminar un par clave-valor
mapa.remove("Ana");
System.out.println("Mapa después de eliminar 'Ana': " + mapa);
}
}
```

## Resumen de Mejores Prácticas

1. **Usar interfaces:** Declara las variables usando interfaces ( `List` , `Queue` , `Map` ) en lugar de implementaciones concretas ( `ArrayList` , `LinkedList` ).
2. **Preferir `Deque` a `Stack`:** `Deque` es más moderna y eficiente.
3. **Elegir la estructura adecuada:**
  - `ArrayList` para acceso rápido a índices.
  - `LinkedList` para inserciones/eliminaciones frecuentes.
  - `TreeSet` / `TreeMap` si necesitas mantener el orden.
  - `HashSet` / `HashMap` para acceso rápido sin orden.
4. **Usar iteradores** cuando se necesite modificar la colección mientras se recorre.

# 1. Lista (List) con Streams

Ejemplo básico:

```
java Copiar código

import java.util.*;
import java.util.stream.*;

public class ListaStream {
    public static void main(String[] args) {
        List<String> lista = Arrays.asList("Manzana", "Banana", "Cereza");

        // Recorrer e imprimir usando stream
        System.out.println("Recorrido de la lista con Stream:");
        lista.stream().forEach(System.out::println);

        // Filtrar elementos que contienen "a"
        System.out.println("Elementos que contienen 'a':");
        lista.stream()
            .filter(fruta -> fruta.contains("a"))
            .forEach(System.out::println);
    }
}
```

Operaciones comunes:


1. **Filtrar** ( `filter` ): Excluir elementos que no cumplen una condición.
2. **Transformar** ( `map` ): Modificar los elementos.
3. **Reducir** ( `reduce` ): Combinar elementos en un único valor.

## 2. Pila (Deque) con Streams

Aunque `Deque` no implementa directamente la interfaz `Stream`, podemos convertirla fácilmente en un stream utilizando el método `stream()` de la colección.

### Ejemplo:

java

 Copiar código

```
import java.util.*;
import java.util.stream.*;

public class PilaStream {
    public static void main(String[] args) {
        Deque<Integer> pila = new ArrayDeque<>(Arrays.asList(10, 20, 30, 40));

        // Recorrer e imprimir usando stream
        System.out.println("Recorrido de la pila con Stream:");
        pila.stream().forEach(System.out::println);

        // Multiplicar por 2 los elementos y mostrarlos
        System.out.println("Elementos multiplicados por 2:");
        pila.stream()
            .map(num -> num * 2)
            .forEach(System.out::println);

        // Sumar todos los elementos
        int suma = pila.stream().reduce(0, Integer::sum);
        System.out.println("Suma de los elementos: " + suma);
    }
}
```



### 3. Cola ( Queue ) con Streams

Con `Queue`, también se puede recorrer directamente con streams.

Ejemplo:

```
java Copiar código

import java.util.*;
import java.util.stream.*;

public class ColaStream {
    public static void main(String[] args) {
        Queue<String> cola = new ArrayDeque<>(Arrays.asList("A", "B", "C"));

        // Recorrer e imprimir usando stream
        System.out.println("Recorrido de la cola con Stream:");
        cola.stream().forEach(System.out::println);


        // Contar elementos que no son "B"
        long count = cola.stream().filter(letra -> !letra.equals("B")).count();
        System.out.println("Número de elementos diferentes de 'B': " + count);
    }
}
```

## 4. Árbol (TreeSet) con Streams

Con `TreeSet`, los elementos ya están ordenados, lo cual se combina perfectamente con el procesamiento de streams.

### Ejemplo:

java

 Copiar código

```
import java.util.*;
import java.util.stream.*;

public class ArbolStream {
    public static void main(String[] args) {
        Set<Integer> arbol = new TreeSet<>(Arrays.asList(10, 5, 20, 15));

        // Recorrer e imprimir usando stream
        System.out.println("Recorrido del árbol con Stream:");
        arbol.stream().forEach(System.out::println);

        // Obtener los números mayores a 10
        System.out.println("Números mayores a 10:");
        arbol.stream()
            .filter(num -> num > 10)
            .forEach(System.out::println);

        // Buscar el número máximo
        int max = arbol.stream().max(Integer::compare).orElse(-1);
        System.out.println("Número máximo: " + max);
    }
}
```

## 5. Mapas (HashMap, TreeMap) con Streams

Los mapas tienen métodos adicionales para trabajar con streams en sus claves, valores o entradas.

Ejemplo:

java

 Copiar código

```
import java.util.*;
import java.util.stream.*;

public class MapaStream {
    public static void main(String[] args) {
        Map<String, Integer> mapa = new HashMap<>();
        mapa.put("Juan", 25);
        mapa.put("Ana", 30);
        mapa.put("Luis", 20);

        // Recorrer Las claves
        System.out.println("Claves del mapa:");
        mapa.keySet().stream().forEach(System.out::println);

        // Recorrer Los valores
        System.out.println("Valores del mapa:");
        mapa.values().stream().forEach(System.out::println);

        // Recorrer Las entradas clave-valor
        System.out.println("Entradas del mapa:");
        mapa.entrySet().stream()
            .forEach(entrada -> System.out.println(entrada.getKey() + " -> " + entrada.getValue()));
    }
}
```

```
// Filtrar y transformar: Claves cuyos valores sean mayores a 20  
System.out.println("Claves con valores mayores a 20:");  
mapa.entrySet().stream()  
    .filter(entrada -> entrada.getValue() > 20)  
    .map(Map.Entry::getKey)  
    .forEach(System.out::println);  
}  
}
```

# Resumen de Operaciones Clave en Streams

Operación	Descripción	Ejemplo
<code>filter()</code>	Filtra elementos que cumplen una condición.	<code>stream.filter(x -&gt; x &gt; 10)</code>
<code>map()</code>	Transforma elementos.	<code>stream.map(x -&gt; x * 2)</code>
<code>forEach()</code>	Ejecuta una acción para cada elemento.	<code>stream.forEach(System.out::println)</code>
<code>reduce()</code>	Combina elementos en un único valor.	<code>stream.reduce(0, Integer::sum)</code>
<code>count()</code>	Cuenta los elementos.	<code>stream.count()</code>
<code>sorted()</code>	Ordena los elementos.	<code>stream.sorted()</code>
<code>collect()</code>	Convierte el stream en una colección o resultado.	<code>stream.collect(Collectors.toList())</code>
<code>max()</code> / <code>min()</code>	Encuentra el elemento máximo/mínimo.	<code>stream.max(Comparator.naturalOrder())</code>

Los streams son ideales para procesamiento complejo con colecciones de datos de manera declarativa y paralelizable si es necesario.

Fin