Estructuras Java

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

1. Vector (Array Dinámico)

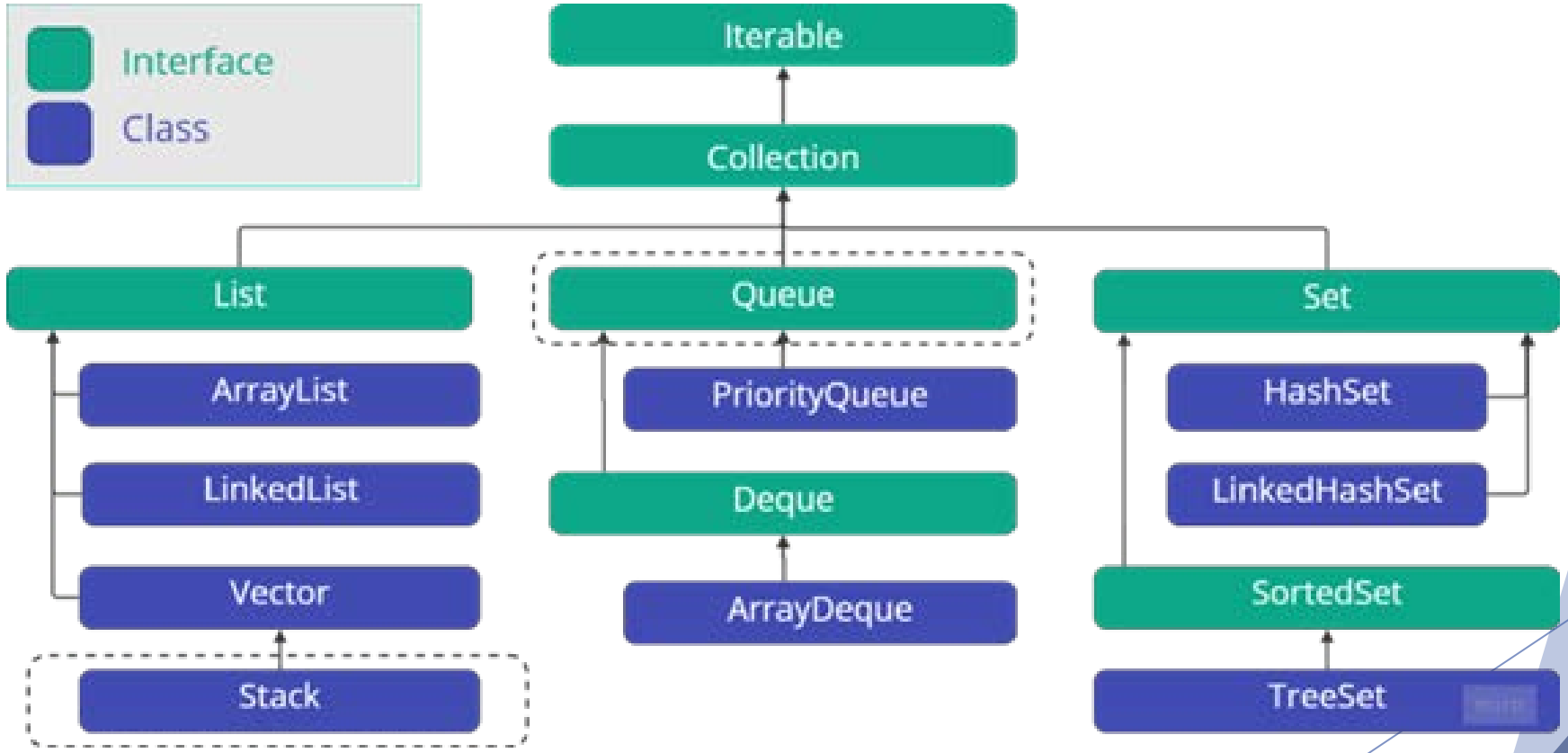
Estructura de datos que almacena elementos de un mismo tipo en posiciones de memoria contiguas. En Java, Vector es la clase original sincronizada, pero se recomienda usar **ArrayList** por ser más moderna y no sincronizada (más rápida en entornos *single-thread*).

Aspecto	**java.util.Vector**	**java.util.ArrayList (Recomendada)**
Implementación	Array redimensionable y **sincronizado** (thread-safe).	Array redimensionable y **no sincronizado** (más rápida).
Ventajas	Acceso a elementos en O(1) (por índice). Crecimiento automático. Sincronización integrada.	Rápida en entornos *single-thread*. Acceso $O(1)$.
Desventajas	Lenta debido a la sobrecarga de la sincronización.	Las operaciones de inserción/eliminación en el medio son $O(n)$. No es thread-safe.

```java
public static void vector() {
    {
        Vector<String> vector = new Vector<>();
        vector.add("Manzana");      // O(1) amortizado
        vector.add(0, "Pera");
        String fruta = vector.get(1); // O(1)
        System.out.println(fruta);
        for(String item: vector) {
            System.out.println(item);
        }
    }

    {
        ArrayList<String> lista = new ArrayList<>();
        lista.add("Manzana");      // O(1) amortizado
        lista.add(0, "Pera");      // O(n)
        String fruta = lista.get(1); // O(1)
        System.out.println(fruta); // Salida: Manzana
    }
}
```

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

```java
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. Lista (Listas Enlazadas)

Colección lineal donde los elementos no están necesariamente en posiciones de memoria contiguas. Cada nodo contiene el dato y una referencia al siguiente nodo (y al anterior, en listas doblemente enlazadas).

Aspecto	**java.util.LinkedList (Doble Enlazada)**
Implementación	Lista doblemente enlazada.
Ventajas	Inserción y eliminación eficientes ($O(1)$) en cualquier extremo o en el medio (si se tiene la referencia al nodo).
Desventajas	Acceso por índice es lento ($O(n)$) porque requiere recorrer la lista desde el principio. Mayor consumo de memoria por almacenar punteros.

✅ Problema típico con listas (nivel inicial/intermedio)

Enunciado:

Dada una lista de números enteros, realizar las siguientes operaciones utilizando una lista enlazada (`LinkedList`):

1. Agregar números a la lista hasta que se ingrese un número negativo (no incluirlo).
2. Mostrar todos los números ingresados.
3. Eliminar todos los números pares de la lista.
4. Mostrar la lista resultante.
5. Calcular el promedio de los valores restantes.

```java
public static void linkedList() {

    Scanner scanner = new Scanner(System.in);
    LinkedList<Integer> numeros = new LinkedList<>();

    // 1. Cargar números hasta que llegue uno negativo
    System.out.println("Ingrese números enteros (negativo para terminar):");
    while (true) {
        int n = scanner.nextInt();
        if (n < 0) break;
        numeros.add(n);
    }

    // 2. Mostrar todos los números ingresados
    System.out.println("Lista original:");
    System.out.println(numeros);
```

```java
// 2. Mostrar todos los números ingresados
System.out.println("Lista original:");
System.out.println(numeros);

// 3. Eliminar números pares (usando ListIterator)
ListIterator<Integer> it = numeros.listIterator();
while (it.hasNext()) {
    int valor = it.next();
    if (valor % 2 == 0) {
        it.remove();
    }
}

// 4. Mostrar la lista final
System.out.println("Lista sin números pares:");
System.out.println(numeros);

// 5. Calcular promedio
if (numeros.isEmpty()) {
    System.out.println("No quedan números impares para calcular promedio.");
} else {
    int suma = 0;
    for (int v : numeros) {
        suma += v;
    }
    double promedio = (double) suma / numeros.size();
    System.out.println("Promedio de los valores restantes: " + promedio);
}
```

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

```java
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 Copiar código

```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());
```

```
        // 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);
    }
}
```

5. Conjunto (Set)

Colección que no permite elementos duplicados. Modela el concepto matemático de conjunto.

Aspecto	java.util.HashSet	java.util.TreeSet
Implementación	Basado en **Tablas Hash** (usa un HashMap internamente).	Basado en **Árbol Binario de Búsqueda Auto-Balanceado** (usa un TreeMap internamente).
Ventajas	Inserción, eliminación y búsqueda en $O(1)$ en promedio. Muy rápido.	Mantiene los elementos **ordenados**. Búsqueda $O(\log n)$.
Desventajas	No mantiene un orden predecible.	Más lento que HashSet ($O(\log n)$ en lugar de $O(1)$).

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

Copiar código

```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);
        }
```

```
        // 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);
    }
}
```

```java
public static void hashSet() {
    HashSet<String> colores = new HashSet<>();
    colores.add("Rojo");
    colores.add("Azul");
    colores.add("Rojo"); // Ignorado (duplicado)
    boolean existe = colores.contains("Azul"); // O(1)
    System.out.println(existe + ", Tamaño: " + colores.size()); // Salida: true, Tamaño: 2
}
```

✅ Problema típico usando `TreeMap`

Enunciado:

Una empresa quiere almacenar la cantidad de ventas diarias de un producto. Cada registro tiene:

- **Fecha** (como `LocalDate`)
- **Cantidad vendida** (entero)

Se pide:

1. Cargar ventas para distintas fechas.
2. Mostrar todas las ventas ordenadas por fecha (automático gracias a `TreeMap`).
3. Obtener la venta de una fecha en particular.
4. Obtener la **primera** y la **última** fecha registrada.
5. Obtener todas las ventas entre dos fechas dadas (submap).
6. Eliminar un registro por fecha.

```java
public static void TreeMap() {

    // 1. Crear el TreeMap
    // LocalDate se ordena naturalmente (compareTo)
    TreeMap<LocalDate, Integer> ventas = new TreeMap<>();

    // 2. Cargar datos (más típico en ejercicios)
    ventas.put(LocalDate.of(2025, 1, 10), 120);
    ventas.put(LocalDate.of(2025, 1, 8), 80);
    ventas.put(LocalDate.of(2025, 1, 12), 150);
    ventas.put(LocalDate.of(2025, 1, 9), 60);

    // 3. Mostrar ventas ordenadas automáticamente
    System.out.println("Ventas ordenadas por fecha:");
    for (Map.Entry<LocalDate, Integer> entry : ventas.entrySet()) {
        System.out.println(entry.getKey() + " → " + entry.getValue());
    }

    // 4. Obtener la venta de una fecha específica
    LocalDate fechaConsulta = LocalDate.of(2025, 1, 9);
    System.out.println("\nVenta del " + fechaConsulta + ": " + ventas.get(fechaConsulta));
```

```
    // 5. Obtener primera y última fecha
    System.out.println("\nPrimera fecha registrada: " + ventas.firstKey());
    System.out.println("Última fecha registrada:   " + ventas.lastKey());

    // 6. Obtener rango de fechas: subMap(desde, hastaIncluida)
    System.out.println("\nVentas entre el 09/01 y el 11/01:");
    Map<LocalDate, Integer> rango = ventas.subMap(
            LocalDate.of(2025, 1, 9),
            true,
            LocalDate.of(2025, 1, 11),
            true
    );

    for (Map.Entry<LocalDate, Integer> e : rango.entrySet()) {
        System.out.println(e.getKey() + " → " + e.getValue());
    }

    // 7. Eliminar un registro
    ventas.remove(LocalDate.of(2025, 1, 10));
    System.out.println("\nDespués de eliminar 10/01:");
    System.out.println(ventas);
}
```

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

```java
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);
    }
}
```

En **Java no existe una API estándar para grafos**, pero hay **dos librerías muy buenas y muy usadas:**

🥇 1. JGraphT (la mejor y más completa)

Recomendada para casi todos los usos.

- ✅ Muy activa y mantenida
- ✅ Soporta grafos dirigidos, no dirigidos, ponderados
- ✅ Algoritmos: BFS, DFS, Dijkstra, A*, Bellman-Ford, Kruskal, Prim, flujo máximo, etc.
- ✅ Muy fácil de usar
- ✅ Permite grafos personalizados, etiquetas, pesos, etc.

⭐ Ejemplo típico con JGraphT

Problema típico de grafos

Dado un conjunto de ciudades y distancias entre ellas, modelar el grafo y:

1. Agregar ciudades como vértices.
2. Agregar caminos como aristas con pesos.
3. Mostrar todos los caminos.
4. Calcular el camino más corto entre dos ciudades.

```java
/**
 * La libreria se descarga de: https://mvnrepository.com/artifact/org.jgrapht/jgrapht-core
 */
public static void Graph() {

    // 1. Crear grafo ponderado no dirigido
    Graph<String, DefaultWeightedEdge> grafo = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);

    // 2. Agregar vértices (ciudades)
    grafo.addVertex("Buenos Aires");
    grafo.addVertex("Rosario");
    grafo.addVertex("Cordoba");
    grafo.addVertex("Mendoza");

    // 3. Agregar aristas con pesos (distancias)
    grafo.setEdgeWeight(grafo.addEdge("Buenos Aires", "Rosario"), 300);
    grafo.setEdgeWeight(grafo.addEdge("Rosario", "Cordoba"), 400);
    grafo.setEdgeWeight(grafo.addEdge("Cordoba", "Mendoza"), 650);
    grafo.setEdgeWeight(grafo.addEdge("Buenos Aires", "Cordoba"), 700);

    // 4. Mostrar el grafo
    System.out.println("Aristas del grafo:");
    for (DefaultWeightedEdge e : grafo.edgeSet()) {
        System.out.println(e + " = " + grafo.getEdgeWeight(e));
    }
```

```java
    // 5. Camino más corto usando Dijkstra
    DijkstraShortestPath<String, DefaultWeightedEdge> dijkstra =
            new DijkstraShortestPath<>(grafo);

    var path = dijkstra.getPath("Buenos Aires", "Mendoza");

    System.out.println("\nCamino más corto Buenos Aires → Mendoza:");
    System.out.println(path.getVertexList());
    System.out.println("Distancia total: " + path.getWeight());
}
```

Aquí tenés **un ejemplo completo y práctico**, ideal para alumnos:

- Modela un **grafo ponderado** con **20 capitales provinciales argentinas**.
- Usa sus **geocoordenadas reales** (lat/lon).
- Calcula **camino mínimo** con **Dijkstra (JGraphT)**.
- Y además genera **un link real de Google Maps** para ver el recorrido entre las 2 provincias elegidas.

🌎 20 provincias con sus capitales y coordenadas

Provincia	Capital	Lat	Lon
Buenos Aires	La Plata	-34.9214	-57.9544
Catamarca	San Fernando del Valle	-28.4696	-65.7852
Chaco	Resistencia	-27.4514	-58.9866
Chubut	Rawson	-43.3002	-65.1023
Córdoba	Córdoba	-31.4201	-64.1888
Corrientes	Corrientes	-27.4712	-58.8396
Entre Ríos	Paraná	-31.7319	-60.5238
Formosa	Formosa	-26.1830	-58.1750
Jujuy	San Salvador	-24.1858	-65.2995
La Pampa	Santa Rosa	-36.6200	-64.2900
La Rioja	La Rioja	-29.4131	-66.8558
Mendoza	Mendoza	-32.8895	-68.8458
Misiones	Posadas	-27.3671	-55.8961
Neuquén	Neuquén	-38.9516	-68.0591

Río Negro	Viedma	-40.8135	-62.9967
Salta	Salta	-24.7821	-65.4232
San Juan	San Juan	-31.5375	-68.5364
San Luis	San Luis	-33.2950	-66.3356
Santa Cruz	Río Gallegos	-51.6230	-69.2168
Santa Fe	Santa Fe	-31.6333	-60.7000

```java
17
18    public static void main(String[] args) {
19
20        // 1. Mapa de provincias -> (lat, lon)
21        Map<String, Punto> coords = new HashMap<>();
22        coords.put("La Plata", new Punto(-34.9214, -57.9544));
23        coords.put("San Fernando del Valle", new Punto(-28.4696, -65.7852));
24        coords.put("Resistencia", new Punto(-27.4514, -58.9866));
25        coords.put("Rawson", new Punto(-43.3002, -65.1023));
26        coords.put("Córdoba", new Punto(-31.4201, -64.1888));
27        coords.put("Corrientes", new Punto(-27.4712, -58.8396));
28        coords.put("Paraná", new Punto(-31.7319, -60.5238));
29        coords.put("Formosa", new Punto(-26.1830, -58.1750));
30        coords.put("San Salvador de Jujuy", new Punto(-24.1858, -65.2995));
31        coords.put("Santa Rosa", new Punto(-36.6200, -64.2900));
32        coords.put("La Rioja", new Punto(-29.4131, -66.8558));
33        coords.put("Mendoza", new Punto(-32.8895, -68.8458));
34        coords.put("Posadas", new Punto(-27.3671, -55.8961));
35        coords.put("Neuquén", new Punto(-38.9516, -68.0591));
36        coords.put("Viedma", new Punto(-40.8135, -62.9967));
37        coords.put("Salta", new Punto(-24.7821, -65.4232));
38        coords.put("San Juan", new Punto(-31.5375, -68.5364));
39        coords.put("San Luis", new Punto(-33.2950, -66.3356));
40        coords.put("Río Gallegos", new Punto(-51.6230, -69.2168));
```

Problems | Javadoc | Declaration | Search | Console | History

<terminated> ProvinciasGrafo [Java Application] C:\na\app\eclipses\eclipse-java-2024-06-R-win32-x86_64\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_21.0.3.v202

```
Camino mínimo: [La Plata, Mendoza]
Distancia total (km aprox): 1029.636143633745

Abrir en Google Maps:
https://www.google.com/maps/dir/-34.9214,-57.9544/-32.8895,-68.8458/
```

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

```java
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

```java
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

```java
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

```java
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

```java
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.get
```

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

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

Fin

Encuesta de Mejora:

https://docs.google.com/forms/d/e/1FAIpQLSctc7kNdLuwz1iVQzjLtx1qmhFs25sWzNnzMBH4RnITHB00kg/viewform?usp=dialog