# Apuntes Java: Estructuras de Datos, Lambdas e Interfaces Funcionales

En este documento se explica cómo utilizar diferentes estructuras de datos en Java como Mapas, Listas, Sets y LinkedLists,
junto con el uso de lambdas y interfaces funcionales para manipular y procesar los datos. Se incluye una serie de ejemplos
y explicaciones detalladas para ayudarte a entender cómo aplicar estos conceptos en tus programas.



## 1. Interfaces Funcionales en Java

Una interfaz funcional en Java es una interfaz que tiene un solo método abstracto. Es especialmente útil para trabajar
con lambdas. Al utilizar estas interfaces, se puede escribir código más compacto y limpio.
Por ejemplo, la interfaz `Filtro<T>` tiene un solo método `test()` que se puede implementar con una lambda.

Ejemplo de interfaz funcional `Filtro<T>`:

```

@FunctionalInterface

interface Filtro<T> {

    boolean test(T t);  // Un solo método abstracto

}

```

¿Por qué usar interfaces funcionales?

- Permiten utilizar lambdas para implementar el método de manera más eficiente.

- Son útiles cuando necesitas aplicar un criterio de filtrado o comparación de manera flexible.

## 2. Expresiones Lambda

Las expresiones lambda son una forma más compacta y eficiente de escribir implementaciones de métodos en Java. Se utilizan

principalmente con interfaces funcionales para hacer que el código sea más conciso y fácil de leer.

Sintaxis básica de una lambda:

(parametros) -> { cuerpo }

Ejemplo de lambda:

```

Filtro<Producto> filtro = p -> p.precio > 20;  // Filtra productos cuyo precio es mayor a 20

```

¿Cuándo usar lambdas?

- Para implementar de forma compacta y eficiente métodos de interfaces funcionales.

- Cuando necesitas realizar operaciones como filtrado, transformación o comparación de datos de

forma concisa.

## 3. Mapas (Maps) en Java

Un Map es una estructura de datos que almacena pares clave-valor, donde cada clave es única. En Java, los tipos más
comunes de Map son `HashMap` y `TreeMap`. Los Mapas permiten almacenar y recuperar datos de forma eficiente.

Ejemplo: Mapa de productos con nombre como clave y precio como valor:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<String, Double> productosMap = new HashMap<>();
        productosMap.put("Camiseta", 15.99);
        productosMap.put("Pantalón", 45.50);
        productosMap.put("Zapatos", 55.00);

        // Mostrar todos los productos del mapa
        for (Map.Entry<String, Double> entry : productosMap.entrySet()) {
            System.out.println(entry.getKey() + " - $" + entry.getValue());
        }
    }
}
```

Filtrar productos en Map con lambda:

```
productosMap.entrySet().stream()
    .filter(entry -> entry.getValue() > 20)
    .forEach(entry -> System.out.println(entry.getKey() + " - $" + entry.getValue()));
```

## 4. Listas en Java: Ordenación y Filtrado

Las listas son estructuras de datos que permiten almacenar elementos de manera ordenada. En Java, puedes usar `ArrayList`
y otras clases de lista para almacenar y manipular datos. Las listas son muy útiles cuando necesitas realizar operaciones
como la ordenación y el filtrado de elementos.

Ejemplo: Ordenar una lista de productos por nombre y precio:

```
List<Producto> productos = new ArrayList<>();
productos.add(new Producto("Camiseta", 15.99));
productos.add(new Producto("Pantalón", 45.50));
productos.add(new Producto("Zapatos", 55.00));
productos.add(new Producto("Camiseta", 10.00));

productos.sort(Comparator.comparing(Producto::toString).thenComparing(Producto::toString));
productos.forEach(System.out::println);
```

```

Filtrar con Lambdas:

```
List<Producto> productosFiltrados = new ArrayList<>();
Filtro<Producto> filtro = p -> p.precio > 20;

for (Producto p : productos) {
    if (filtro.test(p)) {
        productosFiltrados.add(p);
    }
}
```

Transformar una Lista de Productos:

```
List<String> nombresMayusculas = new ArrayList<>();
for (Producto p : productosFiltrados) {
    nombresMayusculas.add(p.nombre.toUpperCase());  // Transforma el nombre a mayúsculas
}
```

## 5. Sets en Java

Un Set es una colección que no permite elementos duplicados. En Java, los tipos más comunes son `HashSet` y `TreeSet`.

El `HashSet` no garantiza el orden, mientras que el `TreeSet` ordena los elementos
```

alfabéticamente.

Ejemplo: Uso de un Set con productos:

```
Set<String> productosSet = new HashSet<>();
productosSet.add("Camiseta");
productosSet.add("Pantalón");
productosSet.add("Zapatos");
productosSet.add("Camiseta");  // Duplicado, no se agregará

productosSet.forEach(System.out::println);
```

Ordenar los productos de forma alfabética:

```
Set<String> productosOrdenados = new TreeSet<>(productosSet);
productosOrdenados.forEach(System.out::println);
```

## 6. Mapa con computeIfAbsent()

El método `computeIfAbsent()` en un Map permite agregar un valor solo si la clave no existe aún. Esto es útil para agregar
elementos a un mapa solo si es necesario.

Ejemplo: Uso de computeIfAbsent() en un Map:

```

Map<String, List<Producto>> categorias = new HashMap<>();

categorias.computeIfAbsent("Ropa", k -> new ArrayList<>()).add(new Producto("Camiseta", 15.99));

categorias.computeIfAbsent("Ropa", k -> new ArrayList<>()).add(new Producto("Pantalón", 45.50));


categorias.get("Ropa").forEach(System.out::println);
```

## 7. Uso de LinkedList

LinkedList es una implementación de lista en Java que permite inserciones y eliminaciones rápidas en cualquier parte
de la lista, lo que la hace ideal para ciertos casos de uso.

Ejemplo: Usar LinkedList para una lista que necesita inserciones rápidas:

```

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

productos.add("Camiseta");

productos.add("Pantalón");

productos.add("Zapatos");


productos.add(1, "Sombrero");  // Inserta en la segunda posición

productos.removeFirst();       // Elimina el primer producto

productos.forEach(System.out::println);
```