Taller 3 - Implementación de Lista Doblemente Enlazada

Conceptos clave:

- **`Secuencia<T>`** es una interfaz que dice qué métodos debe tener una colección tipo lista (agregar, eliminar, obtener, etc).
- **`Iterador<T>`** es una interfaz que dice qué métodos debe tener un objeto que recorre la colección.
- Tu clase **`ListaEnlazada<T>` implementa la interfaz `Secuencia<T>`**, es decir: dice “yo soy una Secuencia, y acá están mis métodos”.
- Para proveer la funcionalidad de recorrido, **`ListaEnlazada<T>` crea una clase interna `ListaIterador` que implementa `Iterador<T>`**.
-

Sobre herencia vs implementación

- **Una clase implementa interfaces** (no hereda de ellas).

Herencia es cuando una clase extiende otra clase (por ejemplo, `class MiLista extends OtraLista`).

¿Qué es una interfaz en Java?

Una interfaz es una lista de métodos que una clase se compromete a implementar. No tiene lógica interna (es decir, no tiene código que se ejecute), pero obliga a las clases que la implementan a cumplir con su contrato.

Por ejemplo:

```
interface Secuencia<T> {
    int longitud();
    void agregarAtras(T elem);
}
```

Esto solo declara qué métodos deben existir, pero no dice cómo funcionan. La clase que implemente esta interfaz debe definir esa lógica.

¿Por qué influye en la compilación?

Aunque la interfaz no tenga código ejecutable, el compilador la usa para verificar si una clase que dice implementarla realmente cumple con el contrato. Si te olvidás de implementar uno de los métodos, o lo hacés mal (por ejemplo, cambiando el nombre o el tipo de retorno), el código no compila.

```
public class ListaEnlazada<T> implements Secuencia<T> {
    ...
}
```

Acá el compilador va a revisar si `ListaEnlazada` tiene todos los métodos que `Secuencia` declara. Si falta alguno, error de compilación.

Operaciones de la interfaz `Iterador<T>`

1. **`boolean haySiguiente()`**
   — Devuelve `true` si hay un elemento siguiente para recorrer (es decir, si no estás al final de la colección).

2. **`boolean hayAnterior()`**
   — Devuelve `true` si hay un elemento anterior para recorrer (si no estás al principio).

3. **`T siguiente()`**
   — Devuelve el elemento *actual* en la posición del iterador y luego avanza el iterador al siguiente elemento.

4. **`T anterior()`**
   — Retrocede el iterador al elemento anterior y devuelve ese elemento.

Explicación detallada línea por línea — ListaEnlazada.java

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

- Importa todas las clases del paquete `java.util`. En este código no se ve uso directo, pero puede servir si usás estructuras o utilidades de Java.

```
public class ListaEnlazada<T> implements Secuencia<T> {
```

- Definís una clase pública llamada `ListaEnlazada` con tipo genérico `T` (puede ser `Integer`, `String`, o cualquier objeto).
- Dice que **implementa la interfaz `Secuencia<T>`**, lo que significa que debe definir todos los métodos que esa interfaz declara (como agregar, eliminar, etc.).

```
private Nodo primero;
private int size;
```

- **`primero`** es una referencia al primer nodo de la lista (puede ser `null` si la lista está vacía).
- **`size`** guarda la cantidad de elementos que hay en la lista.

```
private class Nodo {
    T valor;
    Nodo antecesor;
    Nodo sucesor;

    Nodo(T _valor){
        this.valor = _valor;
    }
}
```

- Definís una clase interna privada `Nodo`. Cada nodo tiene:

    - Un valor (`valor`) del tipo genérico `T`.
    - Una referencia al nodo anterior (`antecesor`).
    - Una referencia al nodo siguiente (`sucesor`).

- El constructor recibe un valor y lo guarda en `valor`.

  Esto define una **lista doblemente enlazada** porque cada nodo conoce quién está antes y después, lo que permite recorrer la lista en ambos sentidos.

```
public ListaEnlazada() {
   primero = null;
   size = 0;
}
```

- Constructor público sin parámetros.
- Inicializa la lista vacía, donde `primero` apunta a `null` y el tamaño es 0.

```
public int longitud() {
   return size;
}
```

- Devuelve la cantidad de elementos en la lista.
- Simplemente devuelve la variable `size`.

```
public void agregarAdelante(T elem) {
   Nodo nuevo = new Nodo(elem);
   nuevo.antecesor = null;
   nuevo.sucesor = primero;
   primero = nuevo;
   size++;
}
```

- Crea un nuevo nodo con el valor `elem`.
- Como va adelante, su `antecesor` es `null`.

- Su `sucesor` apunta al nodo que antes era el primero.
- Actualiza `primero` para que apunte al nuevo nodo.
- Incrementa el tamaño `size`.

```
public void agregarAtras(T elem) {
    Nodo nuevo = new Nodo(elem);
    if (primero == null) {
        primero = nuevo;
    }
    else {
        Nodo actual = primero;
        while (actual.sucesor != null) {
            actual = actual.sucesor;
        }
        actual.sucesor = nuevo;
        nuevo.antecesor = actual;
    }
    size++;
}
```

- Crea un nuevo nodo con `elem`.
- Si la lista está vacía (`primero == null`), el nuevo nodo es el primero.
- Si no, recorre la lista hasta encontrar el último nodo (donde `sucesor` es `null`).
- El último nodo apunta al nuevo nodo (`sucesor`).
- El nuevo nodo apunta hacia atrás (`antecesor`) al último nodo anterior.
- Incrementa el tamaño.

```
public T obtener(int i) {
    int j = 0;
    Nodo actual = primero;
    while (j < i){
        actual = actual.sucesor;
        j++;
    }
    return actual.valor;
}
```

- Recorre la lista desde el inicio hasta la posición `i`.
- Retorna el valor del nodo en la posición `i`.
- como la lista no tiene acceso aleatorio (no es arreglo), para obtener el elemento en la posición `i` hay que recorrerla secuencialmente.

```
public void eliminar(int i) {
    Nodo actual = primero;
    Nodo prev = primero;

    for (int j = 0; j < i; j++) {
        prev = actual;
        actual = actual.sucesor;
    }

    if (i == 0) {
        primero = actual.sucesor;
    }
    else {
        prev.sucesor = actual.sucesor;
    }

    size--;
}
```

- Elimina el nodo en la posición `i`.
- Recorre hasta el nodo a eliminar (`actual`) y su previo (`prev`).
- Si es el primero (`i == 0`), actualiza `primero`.
- Si no, el nodo previo apunta al nodo siguiente, “saltando” el nodo eliminado.
- Decrementa el tamaño.

```
public void modificarPosicion(int indice, T elem) {
    int j = 0;
    Nodo actual = primero;
    while (j < indice)
```

```
    {
        actual = actual.sucesor;
        j++;
    }
    actual.valor = elem;
}
```

- Recorre la lista hasta la posición `indice`.
- Cambia el valor del nodo en esa posición por `elem`.

```
public ListaEnlazada<T> copiar() {
    ListaEnlazada<T> nuevaLista = new ListaEnlazada<>(null);
    Nodo actual = primero;
    while (actual != null){
        nuevaLista.agregarAtras(actual.valor);
        actual = actual.sucesor;
    }
    return nuevaLista;
}
```

- Crea una copia profunda de la lista actual.
- Recorre todos los nodos y agrega sus valores al final de la nueva lista.
- Retorna la lista copiada.

```
public ListaEnlazada(ListaEnlazada<T> lista) {
    if (lista != null) {
        ListaEnlazada<T> copiaLista = lista.copiar();
        this.primero = copiaLista.primero;
        this.size = lista.size;
    }
    else {
        this.primero = null;
        this.size = 0;
    }
}
```

- Constructor que crea una lista a partir de otra dada.

- Si la lista pasada no es `null`, crea una copia y asigna su primer nodo y tamaño.
- Si es `null`, inicializa vacía.

```
@Override
public String toString() {
   Nodo actual = primero;
   StringBuffer sbuff = new StringBuffer();
   sbuff.append("[");
   while (actual != null){
      sbuff.append(actual.valor.toString());
      if (actual.sucesor != null)
        sbuff.append(", ");
      actual = actual.sucesor;
   }
   sbuff.append("]");
   return sbuff.toString();
}
```

- Sobrescribe el método `toString` para representar la lista como una cadena de texto.
- Recorre todos los nodos, los concatena en formato `[valor1, valor2, valor3]`.
- Muy útil para imprimir la lista fácilmente.

```
private class ListaIterador implements Iterador<T> {

   int indice;

   ListaIterador() {
      indice = 0;
   }

   public boolean haySiguiente() {
      return indice < size;
   }

   public boolean hayAnterior() {
      return indice > 0;
   }

   public T siguiente() {
```

```
        T res = null;
        if (haySiguiente()){
            res = obtener(indice);
            indice++;
        }
        return res;
    }

    public T anterior() {
        T res = null;
        if (hayAnterior()){
            indice--;
            res = obtener(indice);
        }
        return res;
    }
}
```

- Clase interna que implementa la interfaz `Iterador<T>`.
- Tiene un índice que indica la posición actual del iterador.
- Métodos:
    - `haySiguiente()` y `hayAnterior()` indican si se puede avanzar o retroceder.
    - `siguiente()` devuelve el elemento actual y avanza.
    - `anterior()` retrocede y devuelve el elemento previo.

```
public Iterador<T> iterador() {
    return new ListaIterador();
}
```

- Método público que devuelve una nueva instancia del iterador para recorrer la lista.

Clase interna `ListaIterador` que implementa `Iterador<T>`

```
private class ListaIterador implements Iterador<T> {

    int indice;

    ListaIterador() {
        indice = 0;
    }

    public boolean haySiguiente() {
        return indice < size;
    }

    public boolean hayAnterior() {
        return indice > 0;
    }

    public T siguiente() {
        T res = null;
        if (haySiguiente()){
            res = obtener(indice);
            indice++;
        }
        return res;
    }

    public T anterior() {
        T res = null;
        if (hayAnterior()){
            indice--;
            res = obtener(indice);
        }
        return res;
    }
}
```

Línea por línea y conceptos del iterador

- `private class ListaIterador implements Iterador<T>`
  Define una clase interna privada que implementa la interfaz `Iterador<T>`.
  Esto significa que debe implementar los métodos que están en `Iterador`: `haySiguiente()`, `hayAnterior()`, `siguiente()`, `anterior()`.

- `int indice;`
  Un contador para saber en qué posición de la lista está el iterador.

- `ListaIterador() { indice = 0; }`
  Constructor que inicializa el iterador en la posición 0 (al principio).

- `public boolean haySiguiente()`
  Retorna `true` si el índice actual es menor que el tamaño de la lista (`size`), o sea que aún queda un elemento para recorrer hacia adelante.

- `public boolean hayAnterior()`
  Retorna `true` si el índice es mayor que cero, o sea que se puede retroceder.

- `public T siguiente()`
  Si hay siguiente, devuelve el elemento en la posición actual y luego avanza el índice para que apunte al próximo.

- `public T anterior()`
  Si hay anterior, retrocede el índice y devuelve el elemento en esa posición.

Conceptos clave en el código

- **Lista doblemente enlazada:** cada nodo sabe quién está antes y después.

- **Recorrido secuencial:** para acceder a cualquier elemento, tenés que empezar desde el primero y seguir el `sucesor`.

- **Encapsulamiento:** el usuario de la clase no manipula los nodos directamente, sino que usa métodos como `agregarAdelante`, `eliminar`, etc.

- **Iterador:** te permite recorrer la lista sin exponer su estructura interna.

¿Qué es StringBuffer?

Es una clase de Java que sirve para **armar cadenas de texto de manera eficiente** cuando vas a hacer **muchas modificaciones o agregados**.

¿Por qué no simplemente usar `String`?

En Java, los `String` son **inmutables**:
Cada vez que hacés algo como:

java
CopiarEditar
```
texto = texto + "más texto";
```

Java crea **una nueva cadena** y copia todo lo anterior más lo nuevo. Eso puede ser lento si lo hacés muchas veces.

¿Qué hace `StringBuffer`?

- Es **mutable**: podés ir agregando cosas sin crear nuevos objetos.
- Tiene métodos como:
    - `.append("texto")` → para sumar al final
    - `.toString()` → para convertirlo a `String` final

Existe también `StringBuilder`, que es casi igual a `StringBuffer`, pero **no es seguro para múltiples hilos (threads)**. En la mayoría de los casos actuales, se prefiere `StringBuilder` porque es más rápido si no estás trabajando con concurrencia.