📂 Actividad: Construcción progresiva de una ListaArray propia en Java

Esta actividad nos invita a ponernos en el lugar de un desarrollador backend que se enfrenta a la necesidad de implementar una estructura de datos personalizada basada en un array, con comportamiento similar a una lista.

En el marco de un proyecto real, no contamos con las colecciones de Java preexistentes o necesitamos una estructura más liviana o particular. Nuestra misión será construir una implementación propia de una lista, y evolucionarla paso a paso, incorporando conceptos de **encapsulamiento**, **herencia**, **polimorfismo**, **interfaces** y **clases internas**.

Nos guiaremos por un conjunto de tests unitarios preexistentes, que iremos desbloqueando y haciendo pasar conforme avancemos.

🔄 Paso 1: ListaArray básica

Partimos de una clase `ListaArray` que tiene:

- Un array interno de objetos (`Object[] elementos`).
- Un método `add(Object elemento)`.
- Un método `size()`.
- Un `toString()` que devuelve el contenido de la lista.

Nuestro objetivo en este paso es entender cómo encapsulamos la estructura interna, y cómo empezamos a construir una colección básica.

Debemos:

- Revisar el código base.
- Comprobar que funciona con los primeros tests de creación y agregado.
- Comprender las limitaciones del uso de `Object[]` sin tipado genérico.
- **Agregar el método `get(int index)`** que devuelva el elemento correspondiente dentro del array, respetando los límites válidos (debe lanzar excepción si el índice está fuera de rango).

Este método será clave para probar comparaciones y acceder a los elementos individualmente, incluso en futuras etapas.

👥 Paso 2: Agregar un iterador propio (no estándar)

Necesitamos ahora recorrer la lista sin usar `for` con índices desde afuera. Vamos a simular un iterador manual con los siguientes métodos directamente en la clase:

- `reiniciarIterador()` — reinicia el actual interno de recorrido al inicio.
- `haySiguiente()` — devuelve `true` si quedan elementos por recorrer.
- `siguiente()` — devuelve el siguiente elemento en la lista y avanza el actual.

🧪 Prueba sugerida en `main`

Podemos probar la implementación con este flujo básico:

```
ListaArray lista = new ListaArray();
lista.add("uno");
lista.add("dos");
lista.add("tres");

lista.reiniciarIterador();
while (lista.haySiguiente()) {
    Object elem = lista.siguiente();
    System.out.println(elem);
}
```

📌 Detalle de implementación esperada

- Debemos mantener un atributo interno `int actual` en la clase.
- `reiniciarIterador()` debe asignar `0` a ese actual.
- `haySiguiente()` debe verificar si `actual < size()`.
- `siguiente()` debe devolver `elementos[actual++]`.
- Si `siguiente()` se llama sin que haya siguiente, lanzar `NoSuchElementException`.

Este paso nos hace pensar en el estado interno del recorrido y su manejo seguro.

📜 Paso 3: Implementar Iterable

En java se espera que una Lista se pueda iterar directamente usando los mecanismos de la API de colecciones de java, esto es implementar la Interfaz iterable en nuestra lista.

Esta acción va a implicar algunos elementos extra más adelante pero el objetivo es que se pueda transformar el flujo de uso anterior en el siguiente:

```
Iterator it = miLista.iterator();
while (it.hasNext()) {
Object elem = it.next();
System.out.println(elem);
}
```

Y eso va a permitir también el uso del ciclo foreach, como sigue:

```
for (Object elem : miLista) {
    ...
}
```

Para eso debemos:

- Hacer que `ListaArray` implemente `Iterable`.
- Crear una **clase interna** (privada o pública interna) que implemente `Iterator`.
- Hacer que el método `iterator()` retorne una nueva instancia del iterador.

📌 Detalle de implementación de la versión Java

- La clase interna `ListaArrayIterator` debe tener un `int pos = 0`.
- `hasNext()` devuelve `pos < size()`.
- `next()` devuelve `elementos[pos++]`.
- Si no hay siguiente, debe lanzar `NoSuchElementException`.

Este paso nos conecta con el sistema estándar de colecciones de Java y nos permite reutilizar estructuras externas como `Collections`, o usarlas con expresiones lambda y streams más adelante.

🤖 Paso 4: Extender `AbstractList`

Primer contacto con el framework de colecciones de Java.

Como paso final, vamos a hacer que nuestra `ListaArray` herede de `AbstractList`, una clase base parcial que nos exige implementar ciertos métodos clave para convertirnos en una verdadera lista Java:

- `get(int index)` — ya lo tenemos, podemos adaptarlo.
- `size()` — debe devolver la cantidad real de elementos agregados.
- (opcionalmente `set`, `remove`, etc.)

📌 Detalle de implementación final

- Debemos extender `AbstractList` e implementar sus métodos abstractos mínimos.
- Reutilizar los métodos ya desarrollados (`get`, `size`).
- Podemos agregar `add(Object o)` si deseamos soportar más funciones del contrato `List`.

Esto nos permite compatibilidad con:

- APIs que esperan listas (`Collections.sort`, etc.).
- Polimorfismo sobre `List`.
- Validaciones con `instanceof List`.

✅ Tests y validación

El proyecto contiene una clase de test unitario que debemos ejecutar paso a paso. Cada test fallido nos indica qué funcionalidad debemos agregar o corregir.

Estrategia sugerida:

- Ejecutar los tests descomentando uno a uno.
- Ver cuál falla y por qué.
- Implementar lo necesario para pasarlo.

- Volver a ejecutar.

🚀 Objetivo

Al finalizar esta actividad:

- Habrás implementado una colección propia desde cero.
- Habrás aplicado herencia, interfaces, clases internas y polimorfismo.
- Comprenderás mejor cómo funcionan las colecciones reales de Java.

Podremos reutilizar esta experiencia para implementar otras estructuras (como pilas, colas o sets) con lógica similar.

¡Manos al código!

🐾 Próximos pasos

Una vez que completemos esta actividad, podremos continuar explorando algunos caminos clave para profundizar en el diseño de estructuras de datos:

- **Incorporar Genéricos (`<T>`)**: Generalizar el tipo de los elementos que maneja nuestra lista para evitar casteos y mejorar la seguridad en tiempo de compilación. Esta refactorización nos permitirá aplicar los mismos conceptos a listas de cualquier tipo de objeto.
- **Revisar el framework de colecciones de Java**: Estudiar en profundidad las interfaces `List`, `Set`, `Queue` y sus principales implementaciones (`ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `PriorityQueue`, etc.). Identificaremos similitudes con nuestra implementación y analizaremos ventajas y desventajas.
- **Comparar con estructuras reales**: Evaluar en qué contextos puede tener sentido una implementación propia y cuándo es más recomendable utilizar las colecciones de la biblioteca estándar.

Estos pasos nos permitirán dar el salto desde una implementación artesanal hacia un conocimiento más profundo del ecosistema de colecciones en Java y su aplicación en proyectos reales.