PRACTICA 1

Creación de TDA arreglo (Arreglo) para su posterior uso en aplicaciones comunes.

Descripción breve

se implementó nuevos métodos a la clase arreglo y en sus respectivas interfaces y se trató de probar que cada método funcionara bien

José Francisco Hurtado Muro
Josefrancisco45rh@outlook.com

Contenido

Introducción

"Existe diversidad de usos que se les puede dar a los arreglos. Desde este punto de vista, los arreglos propician que muchos planteamientos puedan tener una solución sencilla si se llevan a cabo con la ayuda de ellos"

Aldonso Becerra Sánchez

Desarrollo

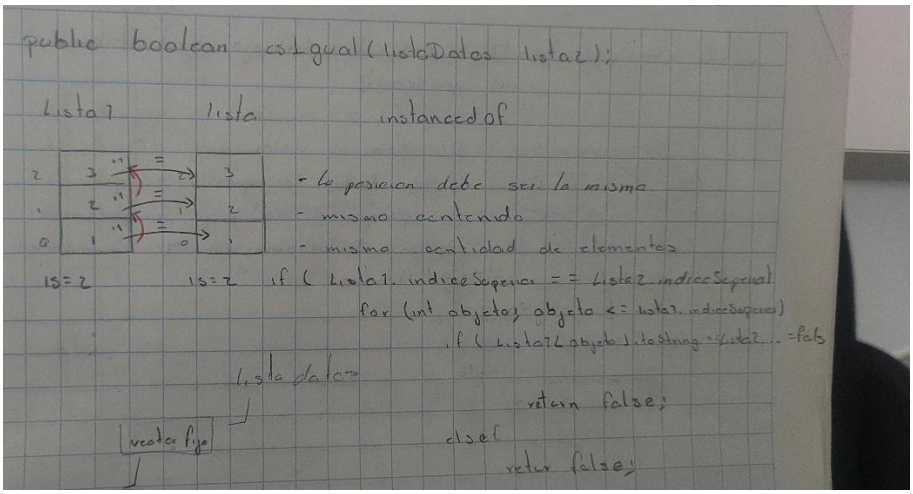
Complete el TDA llamado “Arreglo”, el cual debe tener atributos y métodos necesarios para manipularlo en inserción, búsqueda, modificación y eliminación de forma desordenada.

Para completar muchos de estos métodos necesitará crear los métodos int capacidad (), que regresará el tamaño máximo del arreglo, o el método int cantidad (), que regresa la cantidad de elementos existentes en el arreglo. Estos métodos deben ir desde la interfaz VectorFijo (por ser memoria estática).

Además, agregue los métodos de:

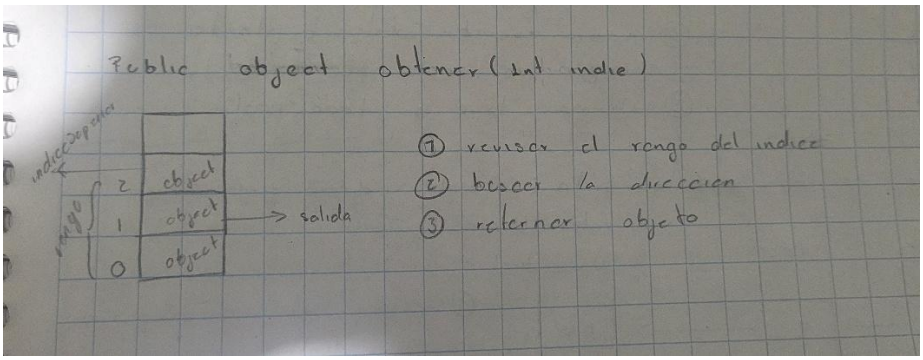
- [declarado en interface ListaDatos] public boolean esIgual(ListaDatos lista2)**:** indica si la lista actual es igual a la lista2. La igualdad se determina por el contenido de cada elemento en la posición correspondiente al mismo índice. Debe validar que lista2 sea una Arreglo.

  Análisis:

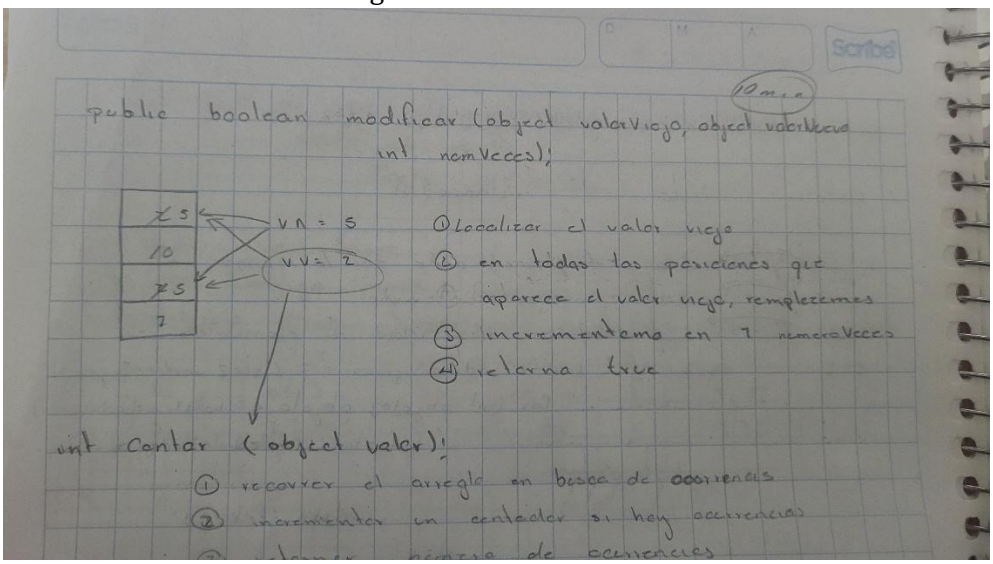
  En este se comparó posición con posición hasta dar si es igual o no donde los puntos más relevantes son si ambas tienen la misma cantidad de elementos

- [declarado en interface VectorFijo] public Object obtener (int indice): regresa el objeto de la posición “indice”. Recuerde validar que el índice que se pasa como argumento debe estar en un rango válido, en caso contrario regrese null.



Análisis:
aquí lo mas importante era ubicar la posición del objeto dentro del arreglo y poder retornarlo

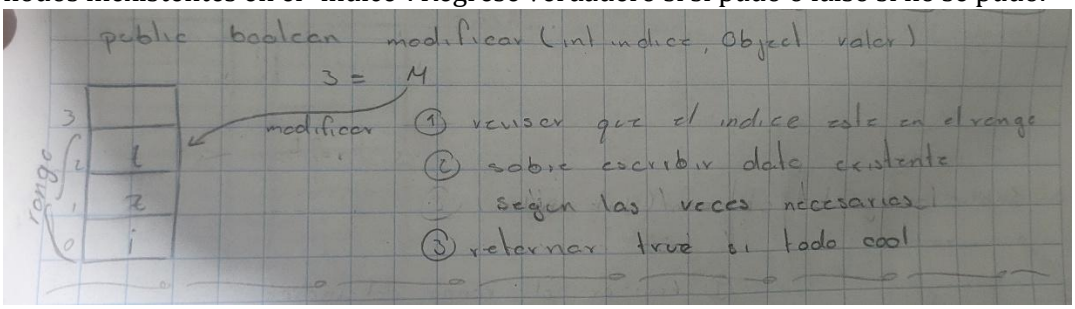
- **[declarado en interface ListaDatos] public boolean modificar (Object valorViejo, Object valorNuevo, int numVeces):** modifica el elemento viejo por el elemento nuevo haciendo una búsqueda y modificación del número de veces de ocurrencias del elemento encontrado indicado por numVeces. Regrese verdadero si hizo alguna modificación. No olvide las validaciones de rangos de índices.



```
10
10
10
3
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025> ^C
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025>
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025>  c:; cd 'c:\U
11-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '-cp' 'C:\Users\Josef\On
10
10
3
3
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025> ^C
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025>
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre\estructuraDatos\ED_4B_2025>  c:; cd 'c:\U
```

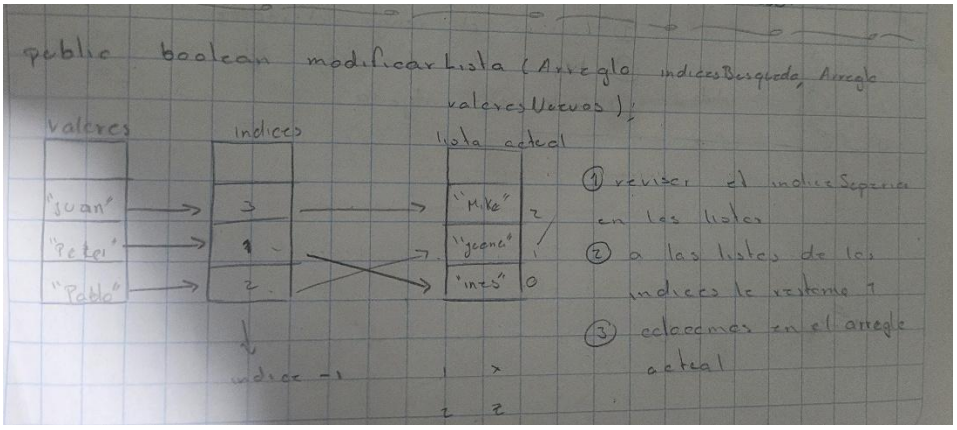
Análisis:
En este la solución fue usando como ayuda a otro método posterior para saber cuantos datos iguales tenemos dentro de un arreglo, y con eso pudimos ubicar un poco mas y poner restricciones cuando el usuario exceda al querer modificar

- [declarado en interface VectorFijo] public boolean modificar (int indice, Object valor): modifica el elemento de la posición indicada por “indice”. “valor” es el nuevo elemento a colocar. Realice el proceso de validación para que no permita cambiar nodos inexistentes en el “índice”. Regrese verdadero si sí pudo o falso si no se pudo.



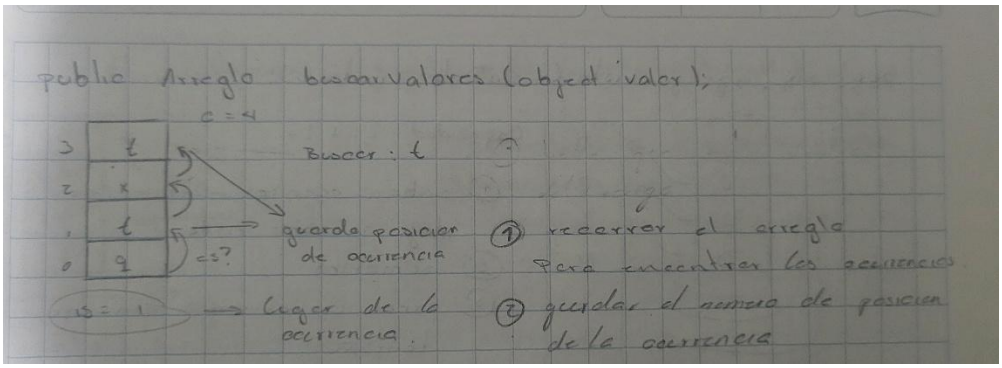
Análisis:
En este caso se nos dio un índice y un valor que será el sustituto del datoViejo, entonces aquí lo que hicimos fue ubicar ese dato y remplazarlo con el nuevo dato

- [declarado en interface VectorFijo] public boolean modificarLista(Arreglo indicesBusqueda, Arreglo valoresNuevos): modifica el elemento del arreglo actual en las posiciones de “indicesBusqueda” por el contenido de “valoresNuevos”. De esta forma se tendrán nuevos valores en cada una de las posiciones del arreglo viejo por cada una de las posiciones del arreglo nuevo tomando como base los “indicesBusqueda”. Regrese verdadero si sí pudo o falso si no se pudo debido a las dimensiones o índices inválidos, por ejemplo.



Análisis:
Aquí estuvo un poco confuso el hecho de cómo íbamos a modificar la lista, pero al ultimo se opto hacer unas listas paralelas entre la lista de los valores y la lista de índices, y lo que se estuvo haciendo era iterar y mandar el valor a la lista actual en el índice que se nos proporcionó en la lista de índices

- [declarado en interface ListaDatos] public Arreglo buscarValores(Object valor): busca dentro de un arreglo los elementos indicados por valor, si existen muchos elementos o en su caso uno; debe guardar en un arreglo (que devolverá) las posiciones de todas y cada una de las ocurrencias del elemento buscado.

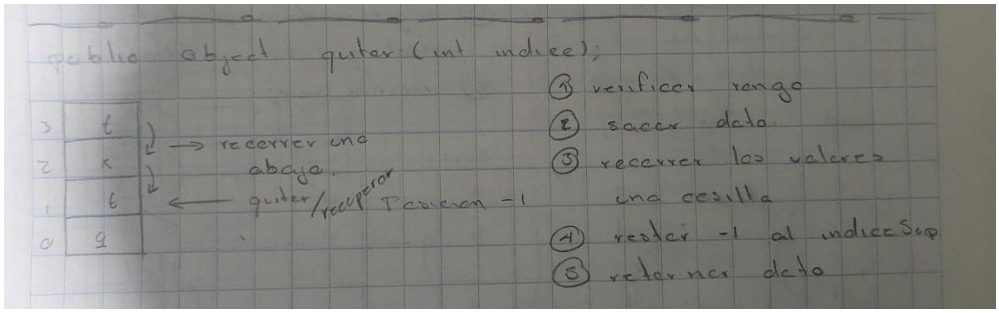


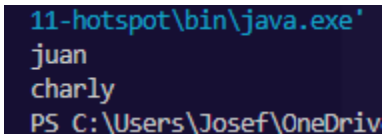

```
2
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
11-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInE
0
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
11-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInE
1
PS C:\Users\Josef\OneDrive\Documentos\4toSemestre
```

Análisis:

En este caso lo que se hizo era buscar cierto valor en el arreglo y obtener su posición y guardarla en un arreglo distinto, en el cual solo tuvimos que iterar por el arreglo principal e ir localizando los objetos

- [declarado en interface VectorFijo] public Object quitar(int indice): elimina un elemento del arreglo en una posición específica, regresando el elemento eliminado.

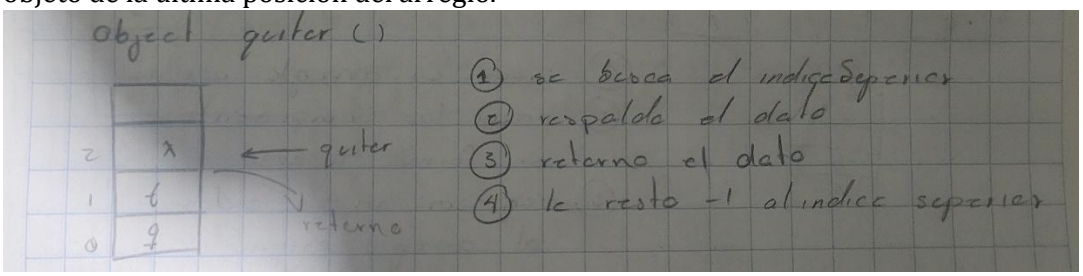




Análisis:

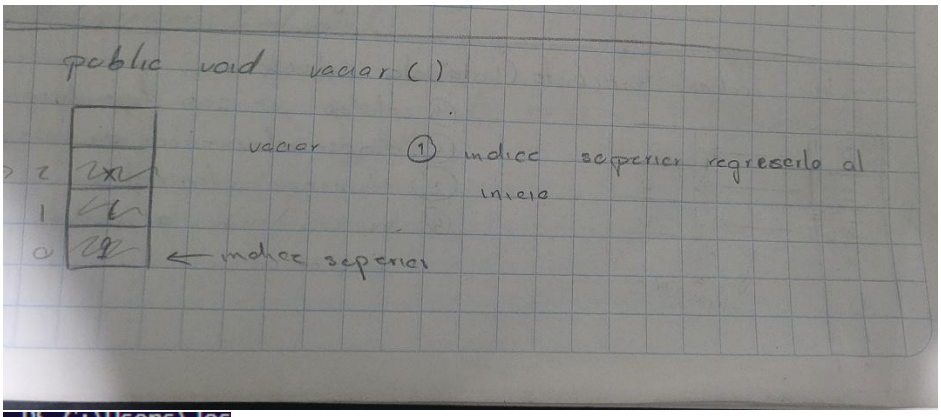
Aquí se nos da el índice del elemento a eliminar, entonces lo que se hizo fue buscar en ese índice el objeto y eliminarlo y recorrer los demás elementos del arreglo

- [declarado en interface ListaDatos] public Object quitar(): regresa y elimina el objeto de la última posición del arreglo.



Análisis:
Aquí solo se tenia que eliminar el ultimo objeto que había entrado, así que la idea fue remover ese dato y al índice superior fue regresarlo un espacio atrás para que en la siguiente escritura sobrescriba ese espacio

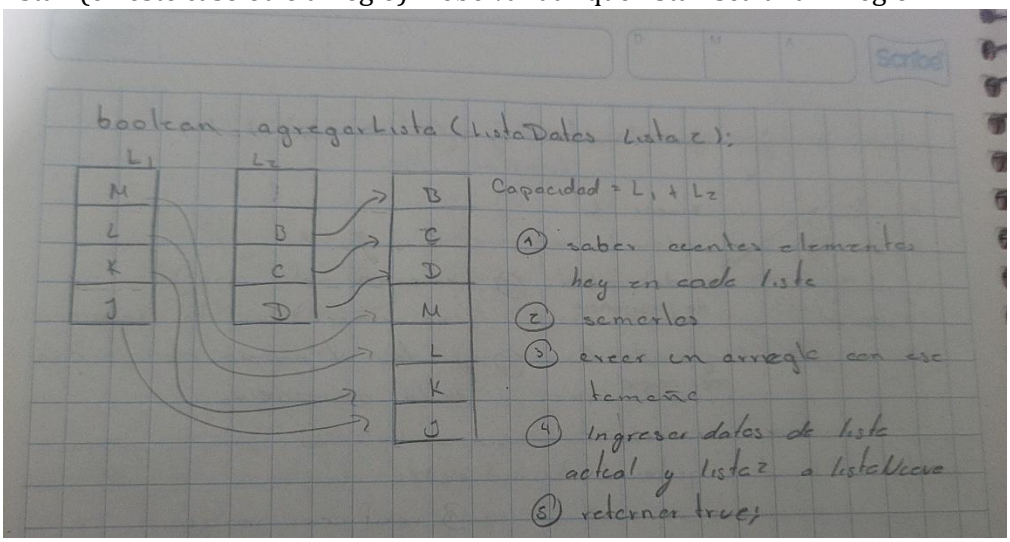
- [declarado en interface ListaDatos] public void vaciar(): vacía el contenido de un arreglo.




```
PS C:\Users\Jos
11-hotspot\bin\
juan
mencho
charly
se limpio
PS C:\Users\Jos
```

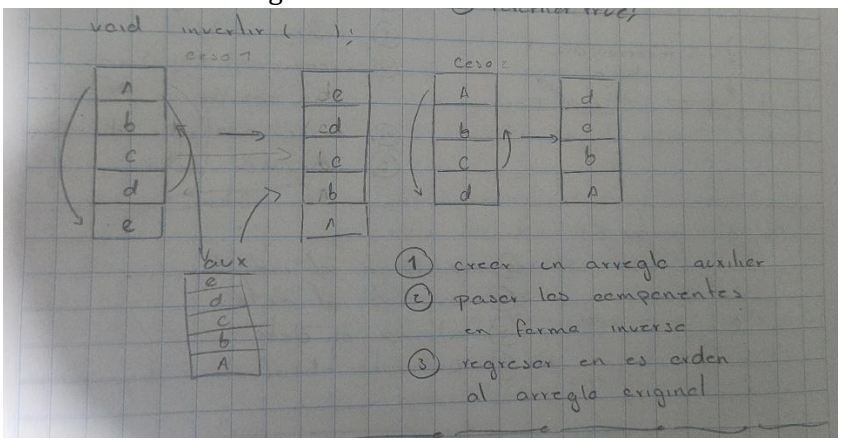
Análisis:
En este lo único que se hizo fue recorrer el índice superior hasta el inicio para poder sobrescribir en futuras inserciones

- [declarado en interface ListaDatos] public boolean agregarLista(ListaDatos lista2): agrega al final de la lista actual (en este caso el TDA Arreglo) el contenido de la lista2 (en este caso otro arreglo). Debe validar que lista2 sea una Arreglo



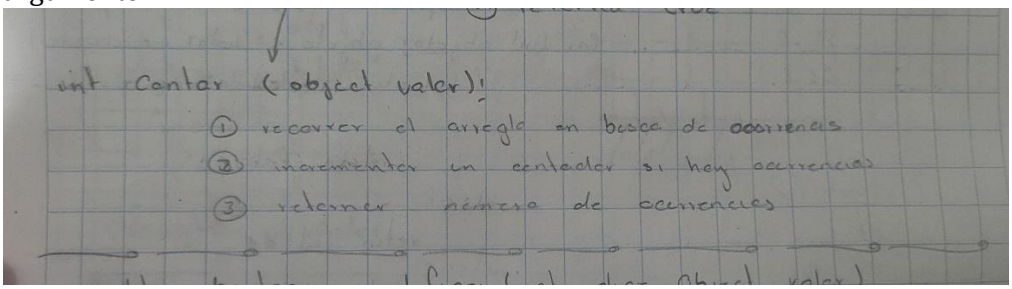
Análisis:
Lo que se hizo aquí fue unir dos arreglos tras la suma de ambos así como que el arreglo final fue un producto de la suma de campos de ambas listas

- [declarado en interface ListaDatos] public void invertir(): invierte el orden de los elementos de un arreglo.



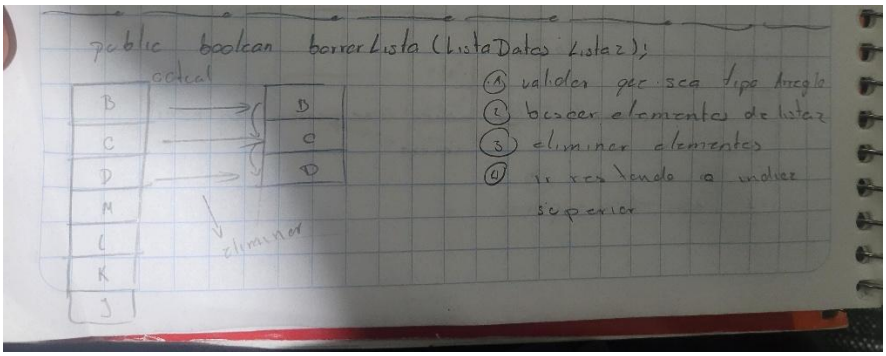
Análisis:
Lo que se hizo aquí es invertir el orden del arreglo original, con ayuda de un arreglo auxiliar para poder pasar todo a este de forma inverso y así poder regresarlo al arreglo principal de nuevo pero con el orden invertido al original

- [declarado en interface ListaDatos] public int contar(Object valor): contar cuántos elementos hay en el arreglo con el contenido igual a valor especificado como argumento.



Análisis:
En este caso fu el método ya antes utilizado que por medio de un filtrado va llevando la cuenta de cierto tipo de datos que le digamos que busque

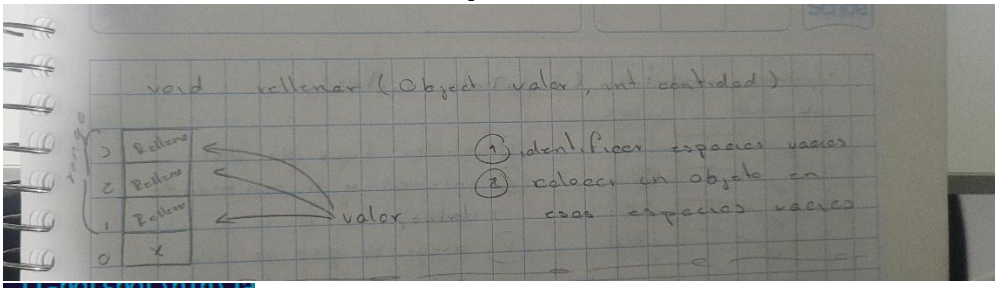
- [declarado en interface ListaDatos] public boolean borrarLista(ListaDatos lista2): elimina cada elemento de lista2 que se encuentre en la lista actual (en este caso el TDA Arreglo). Debe validar que lista2 sea una Arreglo.

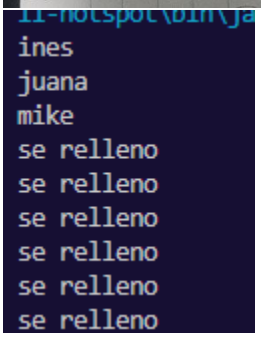


Análisis:

En este caso se nos pedía eliminar una lista u objetos que pertenezcan a otra lista, por lo cual solo hacia falta hacer un filtrado de la lista original y si pertenecía a la segunda lista, este era eliminado dejando solo los datos que pertenezcan a la lista original

- [declarado en interface ListaDatos] public void rellenar(Object valor, int cantidad): rellena todos los elementos indicados por “cantidad” del arreglo con el “valor” indicado. Validar que cantidad no se pase del tamaño de la estructura, en caso que sea estática; sino sólo rellenar hasta la capacidad máxima.

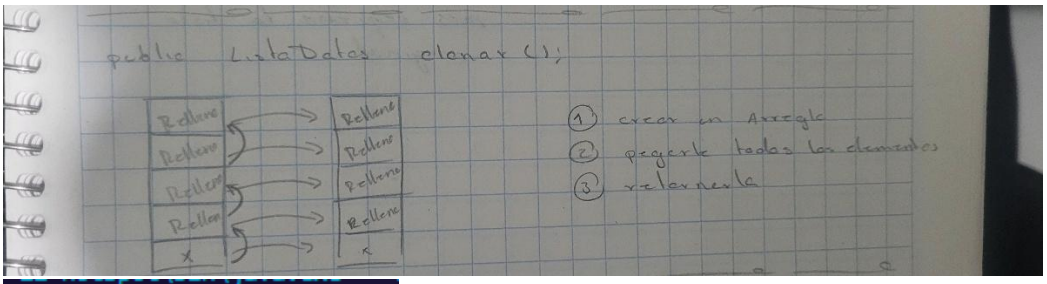




Análisis:

Aquí lo que se tenia que hacer es que dependiendo de la capacidad y de que tan lleno estaba el arreglo, teníamos que rellenar esos espacios en blanco que faltaban para estar lleno con un tipo de dato las veces que quisiéramos

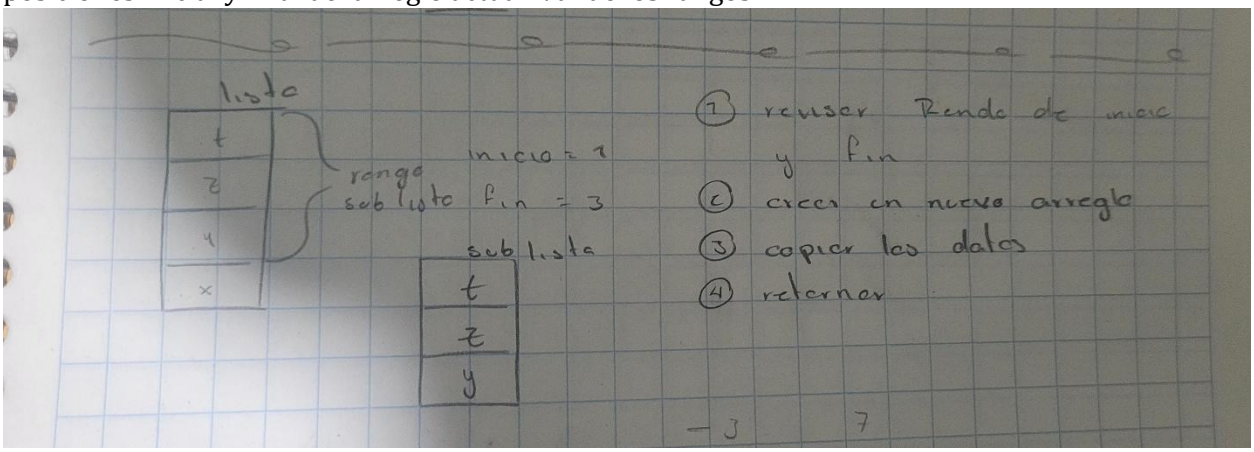
- [declarado en interface ListaDatos] public ListaDatos clonar(): regresa una copia de la lista actual

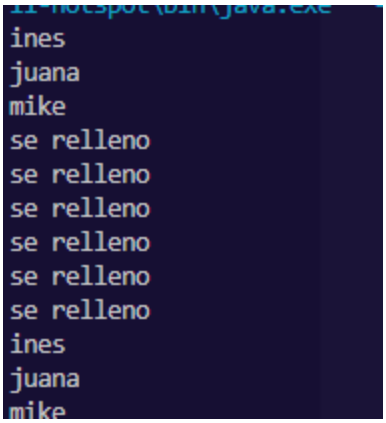



```
ines
juana
mike
se relleno
se relleno
se relleno
se relleno
se relleno
se relleno
ines
juana
mike
se relleno
se relleno
se relleno
se relleno
PS C:\Users\Josef\OneDrive\
PS C:\Users\Josef\OneDrive\
```

Análisis:
Aquí lo único que se hizo fue copiar dato por dato a otro arreglo para tener un arreglo clon de uno original

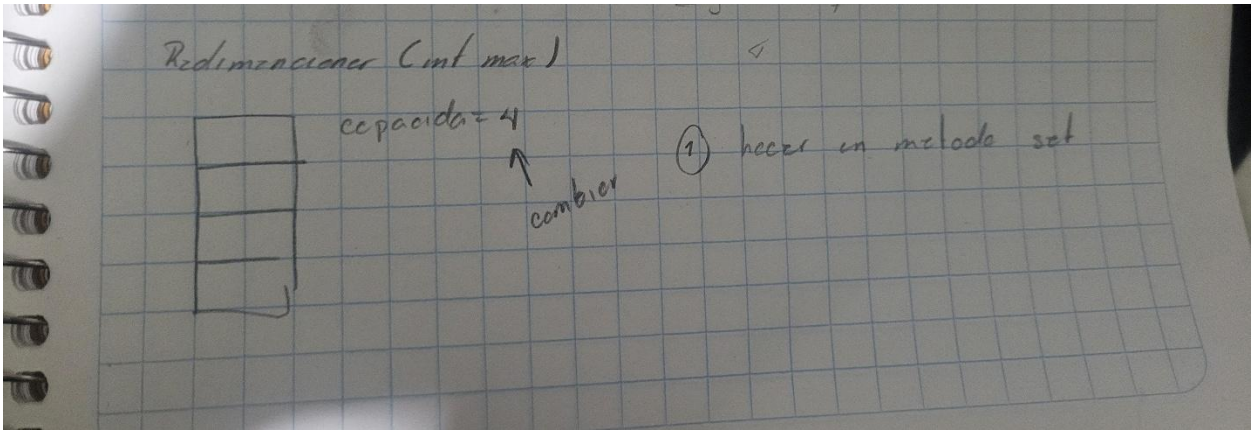
- [declarado en interface ListaDatos] public ListaDatos subLista(int indiceInicial, int indiceFinal): regresa una lista con los elementos indicados con las posiciones inicial y final del arreglo actual. Valide los rangos.





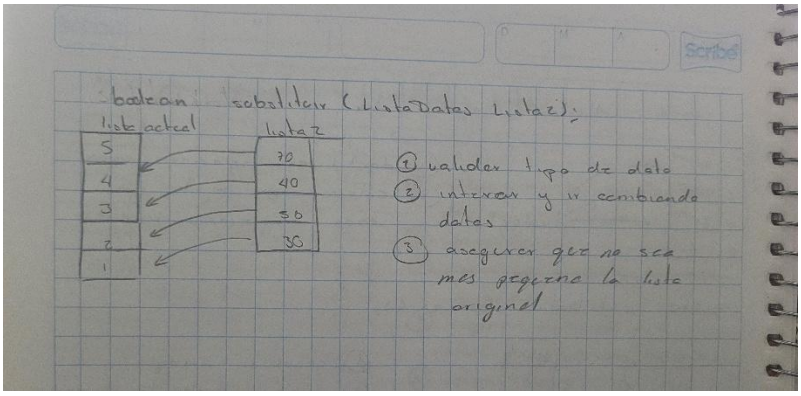
Análisis:
Aquí lo que mas se tomo en cuenta fue el rango de donde a donde seria la sablista, entonces se tuvieron que validar los índices para que todo estuviera acuerdo a lo que se pedía

- [declarado en interface VectorFijo] public boolean redimensionar(int maximo): **redimensiona el tamaño del arreglo al nuevo tamaño indicado por máximo. Si el** tamaño es menor, los elementos sobrantes deben ser eliminados. Si el tamaño es mayor, los datos anteriores deben conservarse y los nuevos espacios deben estar vacíos.



Análisis:
En este caso lo que se hizo fue hacer algo al estilo seter ya que solo teníamos que modificar el dato de la longitud del arreglo

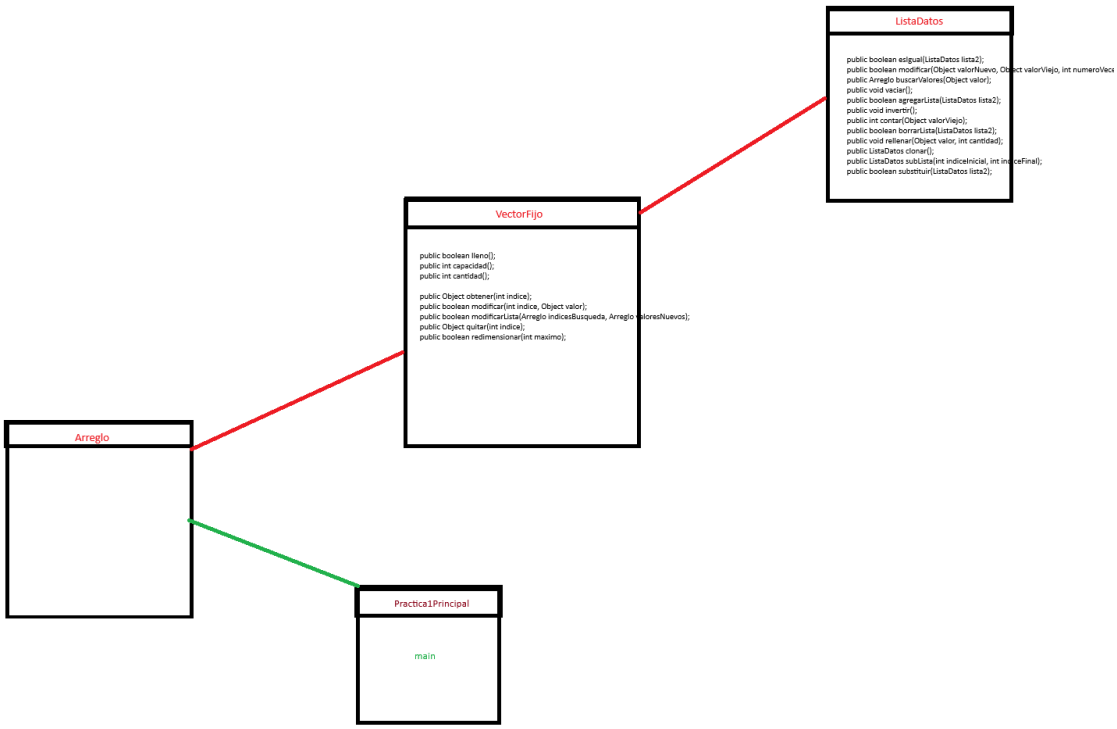
- [declarado en interface ListaDatos] public bolean substituir(ListaDatos lista2): modifica la lista actual por toda la lista2, tengan o no el mismo tamaño.



Análisis:

Lo que se hizo aquí fue que de una lista a la lista actual se pudieran pasar solo los datos que pudieran caber en la lista actual, sin importar si una tuviera mas o menos campos lo cual se hizo restringiendo los datos después de cierto límite del arreglo

Diagrama UML



Preevaluación

1. Cumple con la funcionalidad solicitada. si

2. Dispone de código auto-documentado. si

3. Dispone de código documentado a nivel de clase y método. si

4. Dispone de indentación correcta. si

5. Cumple la POO. si

6. Dispone de una forma fácil de utilizar el programa para el usuario. no

7. Dispone de un reporte con formato IDC. si

8. La información del reporte está libre de errores de ortografía. no

9. Se entregó en tiempo y forma la práctica. no

10. Incluye el código agregado en formato UML.si

11. Incluye las capturas de pantalla del programa funcionando. si

12. La práctica estä totalmente realizada (especifique el porcentaje completado). 85%

Conclusiones

Se vio la importancia d aplicar de manera correcta la POO, además de la importancia del uso de los análisis y diseños para antes de desarrollar, ya que ahorra tiempo y ayuda a visualizar el problema de manera más fácil además de tartar de comprenderlo de varias maneras