PROGRAMACIÓN

Unidad 6. Tipo de Datos Abstractos

1. TIPO DE DATO ABSTRACTO

Definimos las cosas en base a su comportamiento, como se comporta lo que contiene, son muy importante las operaciones (modelamos según comportamiento).
Comportamiento → interfaces
Independiente del Estado u Objeto que almacene, nos lo da los genéricos.
Listas → de acceso aleatorio(ArrayList), acceso secuencial(LinkedList), FIFO(colas), LIFO(pilas)
Conjuntos
Mapas

1.1. COLECCIONES

Deben funcionar igual independientemente de su estado. Se modelan abstractamente y son una lista, pero con el mismo comportamiento.

- **Iterable**: nos fija como recorrer colecciones
- **Collection**: nos fija un comportamiento común a todas las operaciones
- **Lista**: es una interfaz que nos permite trabajar con lista de objetos, secuencia de nodos que almacenan información. Ventajas: es dinámica, crece y decrece según objetos.
    - **ArrayList**: lista de elementos, dinámica de objetos a los cuales se pueden manipular en base a un índice.
      ArrayList lista = new ArrayList();
      Lista.---
      Crear una matriz:

      ```
      ArrayList matriz = new ArrayList(3);
      For (int I = 0; i < matriz.size(); i++){
          matriz.add(I, new ArrayList());
      }
      ```

    - **LinkedList:** ventaja: le doy el índice y crece el vector. Diferencia, ideales cuando vamos a recorrer un for completo el tiempo va a ser menor. También tenemos Lista y Cola.
    - **Vector:** no se utiliza porque es la versión antigua del ArrayList.
    - **Stack:** es una pila, con vector renombrado, por lo tanto, es antiguo.
    - **ArrayDeque:** no acceso secuencial.
    - **Colas**: se conoce como lista FIFO, insertamos al final y sacamos del principio, pero da igual el lado.
    - **Pila**: LIFO, primero en entrar primero en salir, siempre por el mismo lado.
- **Conjuntos**: es una colección que no permite elementos repetidos.
    - **HashSet:** todos los conjuntos están entre comillas, y no permiten elementos duplicados. Ordena en base al hashCode
    - **LinkedHashSet**
    - **TreeSet**: un conjunto que automáticamente esta ordenado, en base al compareTo que le pongas. Se van a ordenar en base a árboles binarios, ordena por un criterio indicado.

- **Mapas**: es un objeto que enlaza una clave a un valor.
  Ventajas: Puedes buscar de forma rápida, no hay objetos con la misma clave.
  - HashMap: ordena en base a los índices.
    Var mapa = new HashMap<integer, Personas>();
    For (int I = 0; i<10; i++){
    Mapa.put(I, new Persona());
    }

    Mapa.put(-1, new Persona().setId(-1).setNombre("Pepe"));

    System.out.println("Mapa de claves");

  - TreeMap: ordena en base a como le indiquemos.

2. GENERICOS

Nos sirven para fijar el tipo de colecciones abstractas por lo que nos aseguramos todos los objetos del mismo tipo.

Interfaz diamante: <Indicamos todos los tipos, al que pertenecen por ejemplo Persona>//IntelliJ es muy listo y nos lo dirá. En interfaz le podemos indicar que sea de tipo <T>, por lo tanto, utiliza el tipo que yo utilice en la interfaz diamante.

Esto en la Interfaz: ICola<T>

Y pongo la T delante de cada variable para generalizar

En la clase: public Class Cola<T> extends<T> ArrayList<T> implements ICola<T>

3. ORDENACIÓN Y BUSQUEDAS

Ordenación
Tenemos dos interfaces:

- Comparable → implementar método compareTo, este método nos va a dar la ordenación por defecto.
  Necesitamos Equals y hashCode + compareTo
  En el main tendremos que llamarlo como Collections.sort(lista);
  Clases POJO: deben tener normalmente Equals, hashCode, compareTo.

- Comparator → ordenar de otra manera que no sea la estándar.
  El comparator no se define dentro de la clase.
  Tenemos que crear una nueva clase que implementa el comparator. Creamos las clases implementando a la clase el comparator.
  En el main llamamos al comparator y le indicamos método de comparación

  //Más fácil de hacer:

  lista.sort(new PersonaIdComparator());
  lista.sort((Persona p1, Persona p2)→ p1.getId() – p2.getId());

Busquedas:
Hay 2 métodos:

- Contains devuelve verdadero o falso si está o no.
  Persona b = new Persona(11, 23, "Pepe");
  System.out.println("Existe?" + lista.contains(b));
  Persona b = new Persona(11, 23, "Pepe");
  System.out.println("Existe?" + lista.contains(c));

- indexOf: me puede decir el índice de donde está o me puede decir verdadero o falso.
  System.out.println("Existe?" + lista.indexOf(b));
  System.out.println("Existe?" + lista.indexOf(c));
  System.out.println("Existe?" + Collectiosn.binarySearch(lista, b, c));

Ejercicio:
Implementar un Hashset implementando un ArrayList, antes del add o set comprobar que no existe ese objeto, si es hashset tiene que estar ordenado por el hashcode y si es TreeSet tiene que estar ordenado por el compareTo.

Dos objetos pueden ser iguales en orden, pero también pueden ser iguales en estado.
Ejemplo: dos personas pueden ser iguales en altura, pero no son iguales como persona.
Comparable // Equals()

4. JOIN Y SPLIT

Join → junta el contenido que hay dentro de un array, con el separador que le indiquemos.
[3, 4, 5] "3 → 4 → 5"
["a","b","c"] "a,b,c"
Var sal = String join("", lista)

Split → busca una cadena con el delimitador y nos devuelve un array.

```
Var cad = "Hola, que tal estas, por aquí, en clase de DAM";
Var lista = cad.split(",");
For (var s: lista){
        System.out.println(s.trim)
}
```

5. EXPRESIONES REGULARES

La expresión es un elemento que nos sirve para identificar patrones, tenemos que definir en base a una plantilla y un lenguaje, el patrón que vamos a utilizar.
Búscame en el texto un intervalo de números: [0-9]
\d : todo lo que sean dígitos
\D : todo lo que no sea dígitos
{8} : 8 primeros valores
[a-z]: texto en minúsculas de la **a** la **z**.
[]: intervalo a detectar
[^a-z] : que no se encuentre este entre la **a** y la **z**.
. : cualquier carácter
| : función or
\s : espacio en blanco
\S : no haya espacio en blanco
\d : todo lo que sea un digito
\D : todo lo que no sea un digito

\w : todas las letras de caracter
\W : todo lo que no sea letras de caracter
a? : que aparezca 0 o 1 vez
a* : aparezca de cero a muchas veces
a+ : aparezca de una a muchas veces
a{3} : tres a's seguidas
a{3,6} : entre tres y seis a's seguidas

```
var texto = "Hola que tal estas, aquí en clase de DAM, 12345678v 12345678ª";
var regex = \\d{8}[A-Z]*;
var lista = texto.split(regex);
// sale false, porque matches busca que la cadena completa sea esa expresión regular
var existe = texto.matches(regex);

final Pattern pattern = Pattern.compile(regex, Pattern.MULTILINE);
final Matcher matcher = pattern.matcher(texto);
while (matcher.find()) {
        System.out.println(matcher.group());
}
```

6. DOBLE BUFFER

Solo cuando estamos leyendo y escribiendo a la vez.
Tenemos dos matrices en una de ella leemos y en la otra escribimos, una vez que hemos escrito, intercambiamos las matrices.

```
//Inicio
tempMatrix = matrix.clone();
matrix = tempMatrix.clone();
```