EJERCICIO DE COLECCIONES.

1. Usando las colecciones basadas en claves realizar un programa que realice la gestión de una lista de contactos usando el dni (String) como clave y como valor el nombre (String) del usuario. Usar **HashMap<String,String>.**

El menú será:

1. Añadir contacto. Solicita DNI y nombre de usuario y añade un nuevo contacto compuesto por el par key-value : dni – nombre.
2. Eliminar contacto. Elimina un contacto (par key-value) mediante el DNI que introduce el usuario. En caso de no existir dicho contacto mostrará un mensaje de error.
3. Mostrar contactos. Se muestra todos los contactos con información de DNI y nombre.
4. Salir. Termina y sale del programa. Pues el menú estará constantemente mostrándose hasta que el usuario desee "Salir".

El código debe ser lo más modular posible, asignando al menos a cada acción un funciona diferente.

1.1. Introduce una mejora/cambio en el ejercicio anterior ahora en el campo valor no registraremos únicamente un String con el nombre sino un objeto de la clase Contacto que contendrá los siguientes:

- Atributos: DNI, nombre, email, teléfono.
- Constructor: Un único constructor con todos los atributos.
- Métodos: Getters&Setters&toString

Al igual que en el ejercicio anterior, se acompañará el ejercicio con el menú:

1. Añadir contacto. Solicita DNI, nombre, email y teléfono del usuario.
2. Eliminar contacto. Elimina todo el objeto por DNI indicado, en caso de existir dicho contacto.
3. Mostrar contactos. Se muestran todos los contactos con toda la información utilizar toString.
4. Salir.

2.- Programa que nos permite registrar una serie de precios dados por consola ofreciendo la posibilidad de obtener unos datos estadísticos de ellos. Usar **ArrayList <Double>:**

El programa ofrecerá un menú que será el siguiente:

1. Nuevo precio. Se añade una nueva cantidad a la colección.
2. Mostrar todos los precios. Se lista las cantidades y el índice que ocupan en la colección.

3. Eliminar precio. Permite eliminar un precio mediante el índice que ocupa en la colección.
4. Precio medio. Muestra por pantalla la media de todos los precios introducidos en la colección.
5. Precio máximo. Proporciona el precio máximo de la colección.
6. Precio mínimo. Proporciona el precio mínimo de la colección.
7. Salir. Termina y sale del programa. Pues el menú estará constantemente mostrándose hasta que el usuario desee "Salir".

3.Contexto. **Las pilas y las colas** son TADs (tipos abstractos de datos) con innumerables aplicaciones. Se usan en multitud de aplicaciones y procesos de nuestro ordenador:

- Pila de llamadas a métodos
- Gestión del historial de acciones
- Navegación Sistemas Operativos (round robin)

LIFO (PILA): Last In First Out: El último elemento apilado (push) es el primer elemento en ser desapilado(pop).

FIFO (COLA): First In First Out: El primer elemento apilado (push) es el primer elemento en ser desapilado(pop).

Métodos comunes:

- **push**. Añade un elemento nuevo a la pila.
- **pop**. Retira un elemento de la pila.
- **peek**. Devuelve el elemento que eliminaría pop, pero no en esta ocasión solo retorna el contenido no lo elimina.
- **size**. Cantidad de elementos en la pila.
- **isEmpty**. Nos indica si la pila está vacía.
- **isFull**. Nos indica si la pila ha alcanzado el número máximo de elementos de tenerlo.

Para implementar se proporciona los siguientes recursos:

Interfaz:

```
public interface TAD {

    public void push(String input) throws TADLleno,TADLLenado;
    public String pop() throws TADVacio, TADVaciado;
    public String peek() throws TADVacio;
    public int size();
    public boolean estaVacia();
    public boolean estaLlena();
    public void listar();

}
```

Excepciones:

- **TADLlenado()** : Enviará el mensaje: “TAD se ha llenado tras la última inserción". Extiende la clase RuntimeException.
- **TADVaciado()** : Enviará el mensaje: “TAD se ha vaciado tras la última retirada". Extiende la clase RuntimeException.
- **TADLleno()** : Enviará el mensaje: “TAD lleno. No es posible insertar más elementos". Extiende la clase Exception.
- **TADVacio()** : Enviará el mensaje: “TAD vacio. No es posible retirar más elementos". Extiende la clase Exception.

Uso de la interfaz:

- **push(String input) : void**. Insertamos un nuevo elemento a la colección (ArrayList) puede suceder las siguientes situaciones:
    - No es posible insertar porque la colección esta llena: lanzamos la excepción TADLleno.
    - Es posible inserta un nuevo elemento, tras lo cual si la cola se ha llenado y no se podrá insertar más elementos: lanzamos la excepción TADLlenado.
- **pop() : String.** Retiramos un elemento del ArrayList función del tipo de TAD. Puede suceder:
    - La pila esta completamente vacía por lo que no hay ningún elemento que extraer: lanzamos la excepción TADVacio.
    - Extraemos un elemento y no retornamos, pero también detectamos que si tras esta acción la pila va a quedar completamente vacía: lanzamos TADVaciado.
- **peek() : String.** Devuelve un elemento del ArrayList en función del tipo de TAD pero no retiramos/quitamos de la colección. Puede suceder:
    - La pila está completamente vacía por lo que no hay ningún elemento que extraer: lanzamos la excepción TADVacio
- **size() : int**. Devuelve el número de elementos dentro de la colección actualmente.
- **estaVacia() : boolean**. Retorna true si la colección está vacía sino false.
- **estaLlena() : boolean**. Retorna true si la colección está completamente llena sino false.
- **listar(): void**. Saca por pantalla los elementos dentro de la colección en ese instante.

3.1. Crea una clase llamada “FIFO” que implementa la interfaz TAD conforme a la naturaleza propia de una cola FIFO trabajando con String como elemento a insertar y retirar de la cola, utilizamos un ArrayList (`private ArrayList<String>)`

A parte de los métodos a implementar introducidos por la interfaz, debemos codificar un constructor donde le pasamos el número máximo de elementos permitidos en la cola.

3.2. Crea una clase llamada “LIFO” que implementa la interfaz TAD conforme a la naturaleza propia de una pila LIFO trabajando con String como elemento a insertar y retirar de la cola. utilizamos un ArrayList (`private ArrayList<String>)`

A parte de los métodos a implementar introducidos por la interfaz, debemos codificar un constructor donde le pasamos el número máximo de elementos permitidos en la pila.

3.3. Crea una clase de pruebas "MainTAD" donde mediante polimorfismo de la clase "TAD" crearemos dos objetos uno que represente una cola (clase "FIFO") de 10 elementos y una pila (clase "LIFO") de 15 elementos.

Probar el correcto funcionamiento de cada método y en caso de producir las diversas excepciones mostrar un aviso notificando de dicha excepción mediante el terminal.

3.4. Serialización de los objetos de las clases LIFO y FIFO.

En "MainTAD" crearemos las siguientes funciones:

- **guardar(TAD obj, String path) : void**. Le pasamos un objeto creado a partir de la interfaz TAD, ya sea FIFO o LIFO y guarda el objeto dentro de un fichero con extensión .dat en función de la ruta proporcionada.
  Será un fichero diferente para el objeto FIFO y el objeto LIFO.
- **cargar(String path): TAD**. Le pasamos una ruta y vuelve el objeto registrado en dicho fichero .dat . En caso de no existir retorna null.

Para implementar estas funciones en el main:

1. Al inicio de la aplicación cargamos los dos objetos desde el fichero .dat, pero en caso de que nos exista dicho fichero y recibamos un null pues utilizamos el constructor por defecto igual que en 3.3.
2. Ante cualquier cambio en la pila o en la cola se guardará el objeto actual dentro de un fichero llamado lifo.dat o fifo.dat respectivamente.