PRÁCTICA 3. Colecciones | ***2 H (1 Ses)***

Objetivos:

- Afianzar y practicar el concepto de colecciones: List y Set

Contenido:

Ejercicio 0: Interfaces Comparable y Comparator

Java integra dos interfaces **Comparable** y **Comparator** para establecer la comparación entre objetos.

Interfaz Comparable

Tiene un único método que devuelve un valor menor, igual o mayor que cero si el objeto actual es menor, igual o mayor que el objeto que se le pasa como parámetro, es decir, devuelve

- un valor negativo (-1) si nuestra instancia es menor que la instancia pasada por parámetro
- cero (0) si ambas instancias son iguales
- un valor positivo mayor que cero (1) si nuestra instancia es mayor que la pasada por parámetro.
- Se lanza una excepción si la instancia obtenida por parámetro no es una instancia de la clase implementada.

```
public interface Comparable{
        int compareTo(Object o) throws ClassCastException;
}
```

De este modo, la implementación de la interfaz Comparable en la clase Persona es la forma más limpia, elegante y reusable para poder aplicar posteriormente una comparación entre objetos Persona. Debemos realizar la implementación del método compareTo.

Al implementar este método a la clase Persona, es posible ordenar y comparar objetos Persona que se encuentren en contenedores como arrays, Vector, Stack, List, TreeSet, etc

//**IMPORTANTE**: Se muestra dos implementaciones de compareTo para Persona. Una donde el DNI es tratado como int y otra donde el DNI es una cadena.

```
public class Persona implements Comparable, implements Relaciones{
  protected int dni;
          .....
//Si DNI esta definido como entero .
  public int compareTo (Object otro){
     Persona otroP = (Persona)otro;
     if (dni >otroP.dni) return 1;
           else if (dni==otroP.dni) return 0
                 else return -1;
  }
//Si DNI esta definido como String .
   public int compareTo(Object o) {
      Persona persona = (Persona)o;
      return this.apellidos.compareTo(persona.apellidos);
    }
}
```

Interfaz Comparator

El interfaz Comparable se utiliza para establecer una ordenación. ¿Qué sucede si en una clase se pueden establecer más de una ordenación? Ordenar por nombre, por edad y por DNI.

Para resolver este problema se utiliza el interfaz **Comparator** para crear clases específicas para ordenar. **Se debe crear una nueva clase que** implementará **Comparator** para fijar los otros tipos de ordenaciones.

Cuidado: este interfaz obliga la implementación de un método denominado **compare** que recibe los dos objetos por parámetros (pues esta fuera de la clase).

public int compare(Object obj) throws ClassCastException;

```
public class NacimientoPersonaComparator implements Comparator {

   public int compare(Object o1, Object o2) {
      Persona persona1 = (Persona)o1;          //Upcasting
      Persona persona2 = (Persona)o2;          //Upcasting
      return persona1.getFechaNacimiento().
            compareTo(persona2.getFechaNacimiento());

   }
}
```

La principal ventaja de esta solución es que esta interfaz puede ser usada para que las distintas colecciones establecidas en Java (ArrayList, LinkedList, TreeSet, ...) lo utilicen para realizar la ordenación por si solas.

```
public class AlphabeticSorting {
  public static void main(String[] args) {

  Persona [] lista=new Persona[6]; //Vector de Personas
  //Añadimos Personas al vector

    System.out.println("---------- SIN ORDENAR ----------");
    pintaLista(Arrays.asList(lista));

    System.out.println( "ORDEN NATURAL DEFINIDO en compareTo por Apellidos----------");
    Arrays.sort(lista);
    pintaLista(Arrays.asList(lista));

   System.out.println("---------- POR FECHAS DE NACIMIENTO ----------");
    Arrays.sort(sa, new AlphabeticComparator());
    System.out.println("After sorting: " + Arrays.asList(sa));
  }
} ///:~
```

Ejercicio 1.

Se pide continuar con el ejercicio de la sesión anterior. Los cambios introducidos son:

- Se debe implementar la clase **Proveedor** como clase derivada de Persona y establecer un **LinkedList** de Proveedores en desguace.
- Desguace esta formado por el **conjunto** de todas las piezas que tiene en Stock (posteriormente se asignarán a los vehículos). Se debe implementar mediante un treeset de modo que es necesario implementar el **interface comparable** en Pieza.
- En lugar de tener un array de vehiculos se tienen un ArrayList de Vehiculos, de modo que se debe modificar los constructores así como el método getVehiculo() y addVehiculo()

Ejercicio 2: Modificación de Desguace

Se pide añadir las siguientes funcionalidades al desguace:

- public boolean addPiezaDesguace(Pieza p): La operación de añadir una pieza al desguace
- public Pieza getPiezaDesguace(String id): La operación de buscar una pieza en el Desguace
- public boolean addProveedor(Proveedor p): La operación de añadir un proveedor al desguace
- public Proveedor getProveedor(String dni): Devuelve un proveedor del desguace
- public Pieza getPiezaVehiculo(String idP, Integer bastidor): Recupera la pieza identificada por idP del coche con dicho bastidor

- Modificar el metodo addPiezaVehiculo() cambia de
  - public boolean addPiezaVehiculo(Pieza p, Integer bastidor)
  - public boolean addPiezaVehiculo(String id, Integer bastidor) porque se debe comprobar si la pieza que se quiere añadir se encuentra en el almacén de piezas. Si se encuentra se decrementa en uno el número de piezas del almacén. Si se encuentra en el vehiculo se incrementa en uno y sino se añade.

Ejercicio 3: Implementación de Programa Principal

Se pide implementar un main que:
- Inicialice el Desguace
- Añada 10 piezas
  - El identificador será aleatorio [0-100]
  - El nombre será aleatorio
  - El stock será aleatorio [0-10]
- Mostrar las piezas ordenadas por identificador
- Mostrar las piezas ordenadas por el número en Stock