21. Colecciones (II)

20.1 Equals y hashcode.

Hemos visto la importancia de definir equals cuando utilicemos una clase en Colecciones. Cuando dos objetos son considerados iguales deben de tener el mismo hash, una ristra binaria que identifica unívocamente a un objeto. Es importante este hecho para utilizar arrays asociativos.

Podemos utilizar el método Objects.hash() para crear un hash con los atributos que consideramos para la igualdad.

Afortunadamente los IDEs nos generan los métodos equals y hashCode() automáticamente, pudiendo indicar los atributos a considerar para la igualdad. En la siguiente figura se pueden ver equals y hashCode generados por Eclipse IDE:

```
public class TrianguloEquilatero extends Poligono{


    private double base;
    private double altura;

    @Override
    public int hashCode() {
        return Objects.hash(altura, base);
    }


    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        TrianguloEquilatero other = (TrianguloEquilatero) obj;
        return Double.doubleToLongBits(altura) == Double.doubleToLongBits(other.altura)
                && Double.doubleToLongBits(base) == Double.doubleToLongBits(other.base);
    }
```

20.2 LinkedList. Métodos adicionales.

En este caso se implementa la lista mediante una lista doblemente enlazada. Por lo tanto, el coste temporal de las operaciones será el de este tipo de listas. Cuando realicemos inserciones, borrados o lecturas en los extremos inicial o final de la lista el tiempo será constante, mientras que para cualquier operación en la que necesitemos localizar un determinado índice dentro de la lista deberemos recorrer la lista de inicio a fin, por lo que el coste será lineal con el tamaño de la lista

• addFirst(), addLast(), getFirst(), getLast(), removeFirst() y removeLast(), que no están definidos en ningún interfaz o clase base y que permiten utilizar la Lista Enlazada como una
Pila y una Cola.
• getFirst(), getLast(), removeFirst() y removeLast() lanzan NoSuchElementException si la lista está vacía.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una pila o de una cola. En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

20.3 Comparable

Para poder realizar ordenaciones o estructuras de datos ordenadas es necesario que los objetos implementen Comparable. El método compareTo devolverá un valor negativo cuando ***this*** sea menor al objeto que se pasa como parámetro, cero cuando sea igual y un número positivo cuando sea mayor:

```java
public class TrianguloEquilatero extends Poligono implements Comparable<TrianguloEquilatero>{


    private double base;
    private double altura;



    @Override
    public int compareTo(TrianguloEquilatero te) {

        int comparador;
        if (this.area()==te.area()){
            comparador = 0;
        } else if (this.area()<te.area()) {
            comparador = -1;
        } else {
            comparador = 1;
        }
        return comparador;
    }
```

Ejercicio 21.1

Implementa equals y hashcode en Alumno teniendo en cuenta solo el número de expediente. Usa contains y hash() para comprobar que funciona correctamente.

Ejercicio 21.2

Mejora el ejercicio 20.6, de forma que el Grupo tenga un método ordenarAlumnos(), que ordene a los alumnos por número de expediente.

Ejercicio 21.3

Crea una clase ColaAlumnos, que contenga una cola de Alumnos, el método llegarAlInstituto(Alumno) lo añadirá a la cola, el método Alumno entrarEnClase() quitará al primer alumno de la cola, devolviéndolo, lanzando una excepción ColaVacia si intentamos que entre un alumno en clase cuando la cola está vacía. El método mostrarCola() nos mostrará a los alumnos en la cola.

Ejercicio 21.4

Utilizando una pila, crea una función que a partir de un String nos diga si una expresión con (){}<> es correcta.

Ejemplo:

“((<>))” - Correcto

“<({{}})>” - Correcto

“<{(){}}>” -Correcto

“(<(>)>)” - Incorrecto

Ejercicio 21.5

Mejora el MiniRPG de forma que:

- Además del guerrero existirán tres personajes más, Arquera, Maga y Nigromante, cada uno de ellos tendrá un método atacar distinto.

- Todos los personajes pueden portar un **Arma**, el Arma tendrá métodos int minDanho() e int maxDanho(), que proporcionarán un daño extra al atacar. Existen tres tipos de Arma: Espada, Arco, y Varita. Sólo las Magas y los Nigromantes podrán portar varitas, además del resto de Armas.

Las Armas tendrán atributos destrezaMinima() y fuerzaMinima(), los Personajes no podrán **portar** un Arma si no cumplen en sus atributos.

Las Magas pueden lanza **Hechizos**, los Hechizos los tienen que **aprender**, las magas usan su método **hechizar(Personaje, Hechizo h)** para hechizar a sus oponentes, quitándole un daño extra, siempre y cuando el Hechizo haya sido aprendido. Los hechizos tendrán un método **hechizar(Personaje)**, que infringirá el daño mágico.

Los Nigromantes tienen un método **sanar(Personaje p)**, que recupera la vida otro Personaje al máximo.

Una vez acometidos los requerimientos básicos, se puede mejorar libremente la jerarquía de **Armas** y **Hechizos**, además de clasificar los Hechizos e implementar resistencias a ellos.