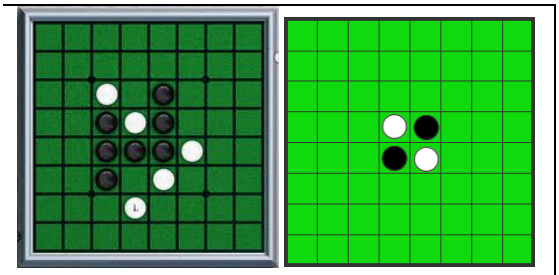
ETSIT-UPM. PROGRAMACIÓN. CONVOCATORIA EXTRAORDINARIA. JULIO 2013.
APELLIDOS:
NOMBRE:
EMAIL:

Problema 1. (2 puntos) Se dispone de la clase *Otelo* para jugar a este juego, en el que se usa un tablero de 8 filas por 8 columnas y 64 fichas idénticas, redondas, blancas por una cara y negras por la otra. Juegan dos jugadores, uno lleva las fichas blancas y el otro las negras. Al inicio se colocan cuatro fichas tal como se ve en la figura de la derecha, dos fichas blancas y dos negras en diagonal.



Cada jugador en su turno, coloca una ficha con la cara de su color hacia arriba en una casilla libre del tablero, de manera que deje una o varias fichas del color contrario entre ella y otra de su mismo color ya colocada, formando una línea continua horizontal, vertical o diagonal. A todas las fichas contrarias que han quedado entre medio se les da la vuelta para que muestren el color de la otra cara.

La clase *Otelo* dispone de un atributo *tablero* que consiste en un array de enteros bidimensional cuyos valores sólo pueden ser *0* (casilla vacía), *1* (casilla con ficha blanca) o *-1* (casilla con ficha negra). También dispone de un método que comprueba si la posición elegida por un jugador en su turno es válida –esto es, que hay fichas a las que se les cambia el color. Sobre el tablero de la izquierda, la ficha blanca marcada con "L" (la de más abajo) no es una jugada válida al no encerrar fichas negras contiguas hasta otra blanca.

public boolean jugadaValida(int fila, int columna, int color)

Pregunta 1. (2 puntos) Escriba el método *movimientoJugadorFila* con tres parámetros: la fila y columna donde el jugador ha colocado la ficha y el color de la ficha. Este método cambia de color todas las fichas del jugador contrario que se encuentren entre la ficha recién jugada y que se indica con los parámetros y otra ficha de su color previamente colocada que esté en la misma fila. En la misma fila puede cambiar fichas a la izquierda y a la derecha de la ficha nueva.

El método lanzará una Excepción si la posición de la nueva ficha no se corresponde con una casilla válida y vacía del tablero, si el color del jugador no es blanco o negro y si la posición elegida para jugar no es válida. Para el tablero de la izquierda cuya esquina superior izquierda corresponde a la casilla 0,0, al llamar a este método con parámetros (4,1,1), convertirá en blancas las fichas en las casillas [4,2]; [4,3]; y [4,4].

```
public void movimientoJugadorFila (int fila, int columna, int color) throws Exception{
        if ((color != 1) && (color != -1) ) throw new Exception("bad color");
        if (fila < 0 || fila >= tablero.length || columna < 0 || columna >= tablero[0].length)  throw new Exception("Bad position");
        if (tablero[fila][columna] != 0) throw new Exception();
        if (!jugadaValida(fila,columna,color)) throw new Exception();

        tablero[fila][columna]=color;

        // compruebo cuantas fichas a la drcha de la jugada hay que cambiar
        int fichasACambiar = 0;
        boolean cambia = false;
        for (int i = columna +1; i < tablero[fila].length; i++) {
                if (tablero[fila][i] == 0) break;
                else if (tablero[fila][i] == (color*(-1))) fichasACambiar++;
                else if (tablero[fila][i] == color){
                        cambia=true;
                        break;
                }
        }
        if (cambia) for (int i = columna+1; i<=columna+fichasACambiar; i++) tablero[fila][i] *= -1;

        // compruebo cuantas fichas a la izq de la jugada hay que cambiar
        fichasACambiar = 0;
        cambia = false;
        for (int i = columna -1; i >= 0; i--) {
                if (tablero[fila][i] == 0) break;
                else if (tablero[fila][i] == (color*(-1))) fichasACambiar++;
                else if (tablero[fila][i] == color){
                        cambia=true;
                        break;
                }
        }
        if (cambia) for (int i = columna-1; i>=columna-fichasACambiar; i--) tablero[fila][i] *= -1;
}
}
```

Problema 2. (2 puntos). La clase *Arrays* de la biblioteca de java incluye el método *public static sort(Object[] a, Comparator c)* que permite ordenar un array *a*, con un comparador *c*. El comparador *Comparator* aparece a la derecha.

```
interface Comparator {

  int compare(Object x,Object y);

}
```

El método *sort* ordena el array haciendo llamadas al comparador, que compara pares de objetos del array. Con el comparador, *sort* determina cuándo un objeto *x* debe ir antes o después de otro objeto *y* en el array. Si *x* debe ir antes, el comparador devuelve algo menor que 0; si *x* debe ir después, devuelve un positivo mayor que 0; y si no hay precedencia devuelve 0.

Pregunta 1. (0.5 puntos) Escriba la clase *ComparaString* que implementa *Comparator* y compara dos String *x*, *y*, devolviendo negativo cuando la longitud de *x* es mayor que *y*, positivo cuando la longitud de *y* es mayor y 0 cuando tienen igual longitud.

```
public class ComparaString implements Comparator {

        public int compare(Object x, Object y) {
                return ((String) y).length() - ((String) x).length();
        }
}
```

Pregunta 2. (1 punto) Utilizando la clase *ComparaString* y el método *sort* implemente un método para ordenar por longitud el siguiente array de *String:*

String[] arrayAOrdenar = {"Esteban","Luis","Pedro","Juan","Fernando"};

(se puede suponer que está implementada la clase *ComparaString*, aunque no se haya contestado la pregunta anterior). Su cabecera es la siguiente:

public static void pruebaSort(String[] unArray)

```
public static void pruebaSort(String[] unArray) {
        Arrays.sort(unArray,new ComparaString());
}
```

Pregunta 3. (0.5 puntos) Si se cambia la cabecera de pruebaSort por:

public static void pruebaSort(String[][] unArray)

Indique si servirá exactamente la misma implementación anterior para ordenar el siguiente array

String[][] arrayAOrdenar = {{"Esteban","Luis"},{"Pedro","Juan","Fernando"}};

La implementación:

```
public static void pruebaSort(String[][] unArray) {
        Arrays.sort(unArray,new ComparaString());
}
```

produce errores de ejecución porque ComparaString compara String y no arrays de String que es lo que pasamos como parámetro

Problema 3. (4 puntos) Una empresa distribuye componentes de sistemas que pueden clasificarse en tres categorías: *Hardware*, *Software* y *Soporte*. Partiremos de la clase Componente de la derecha (suponga que existen métodos accesores para todos los atributos).

Pregunta 1. (0.5 puntos) Se pide crear un tipo enumerado (*TipoComponente*) que permita caracterizar cada componente en función de su categoría.

```
public class Componente {
  private String identificador;
  private String descripcion;
  private TipoComponente tipo;
  public Componente (String id, String desc, TipoComponente t) {
     this. identificador = id;
     this.descripcion = desc;
     this.tipo = t;
  }
  public boolean equals(Componente c) {
    return this.identificador.equals(c.getIdentificador());
  }
}
```

```
public enum TipoComponente {HARDWARE, SOFTWARE, SOPORTE};
```

Pregunta 2. (1 punto) La clase *Componente* contiene los atributos necesarios para identificar cualquier componente de los que vende la empresa, salvo el precio. Escriba una clase *ComponenteEnVenta* que extienda a la clase *Componente* y permita introducir el precio (en euros) de cada componente. Debe crear: constructor, accesor y modificador (*getter* y *setter*) para el nuevo atributo.

```
public class ComponenteEnVenta extends Componente {
        private double precio;
        public ComponenteEnVenta (String id, String desc, TipoComponente tipo, double p) {
                super(id,desc, tipo);
                this.precio = p;
        }
        public void setPrecio (double nuevo) {
                this.precio = nuevo;
        }
        public double getPrecio () {
                return precio;
        }
}
```

Pregunta 3. (1 punto) La empresa necesita crear una lista de precios con todos los componentes que vende. Escriba la clase *ListaDePrecios* para almacenar el conjunto de todos los componentes a la venta por la empresa, donde se incluyan todos los atributos de cada uno, incluido el precio. Escriba el atributo y el constructor. Se pretende que la lista de precios no contenga componentes repetidos.

```
public class ListaDePrecios {

private Set<ComponenteEnVenta> componentes;

  public ListaDePrecios() {
    componentes = new HashSet <ComponenteEnVenta> ();
  }
}
```

Pregunta 4. (0.5 puntos) En la clase anterior, defina un método *void insertaComponente (ComponenteEnVenta c)* que añada un componente a la lista de precios anterior. Si el componente a añadir ya se encuentra en la lista de precios, el método no debe hacer nada.

```
public void insertaComponente (ComponenteEnVenta c) {
        if (!componentes.contains(c)) {
                componentes.add(c);
        }
}
```

Pregunta 5. (1.0 punto) Para la clase anterior (*ListaDePrecios*), defina un método *double getPrecio (Componente c)* que devuelva el precio, en euros, del componente que se le pasa como parámetro. Si el componente no está en la lista de precios, se devuelve -1.0.

```
public double getPrecio (Componente c) {
        for (ComponenteEnVenta comp: componentes) {
                if (comp.equals(c)) {
                  return comp.getPrecio();
                }
        }
        return -1.0;
}
```

Problema 4. (2 puntos) Suponga que existe la clase *TelemetriaEstado* que contiene información recibida de un satélite; uno de sus atributos es un número real que mide la temperatura del satélite; la clase tiene un método accesor (*getTemperatura*) para este atributo. Por otra parte, la clase *Satelite* contiene un array (*telemetrias*) de referencias a objetos *TelemetriaEstado*.

Pregunta 1. (2 puntos) Escriba un método de la clase *Satelite* para calcular la diferencia entre las temperaturas máxima y mínima que hay en el array de *TelemetriaEstado*. El método no lanza excepciones. Debe cuidar de no considerar las referencias nulas en el array. Puede suponer que siempre hay al menos dos referencias diferentes de *null* en el array.

```
public double calcularMayorDiferenciaTemperatura(){
        int i=0;
        while(telemetrias[i]==null) i++;
        double maxima = telemetrias[i].getTemperatura();
        double minima = telemetrias[i].getTemperatura();
        //Busca la temperatura maxima y la minima
        for(i=0; i<telemetrias.length; i++){
                TelemetriaEstado t = telemetrias[i];
                if(t!=null){
                        if(maxima < t.getTemperatura())
                        maxima = t.getTemperatura();
                        if(minima > t.getTemperatura())
                        minima = t.getTemperatura();
                }
        }
        return maxima-minima;
}
```

```
public double calcularMayorDiferenciaTemperatura(){
        double máxima =0.0;
        double minima = 0.0;;
        boolean esperandoPrimerDatoValido = true;
        for(int i=0; i<telemetrias.length; i++){
                if(telemetrias[i]!=null){
                double temp=telemetrías[i].getTemperatura();
                if (esperandoPrimerDatoValido){
                        maxima = temp;
                        minima = temp;
                        esperandoPrimerDatoValido=false;
                }
                if(maxima < temp) maxima = temp;
                if(minima > temp) minima = temp;
        }
        }
        return maxima-minima;
}
```