# Segundo Parcial de IIP - ETSInf
## Fecha: 15 de enero de 2016. Duración: 2:30 horas.

1. 6.5 puntos Se dispone de la clase `Termostato` que representa el controlador de temperatura de un dispositivo térmico instalado en alguna zona de un espacio determinado (una vivienda, espacio de oficinas, etc.). Cada termostato se define en base a cuatro datos: su identificador (nombre de la zona donde se sitúa), su modo (`FRIO` para refrigeración, `CALOR` para calefacción), la temperatura actual de la zona y la temperatura de confort que desea el usuario.

   Esta clase ya es conocida y se muestra a continuación un resumen de su documentación:

**Field Summary**

| | |
|---|---|
| static int | CALOR<br>Constante que representa el modo calefaccion de un termostato |
| static int | FRIO<br>Constante que representa el modo refrigeracion de un termostato |
| static int | T_IDEAL_CALOR<br>Constante que representa la temperatura ideal en modo CALOR de un termostato |
| static int | T_IDEAL_FRIO<br>Constante que representa la temperatura ideal en modo FRIO de un termostato |

**Constructor Summary**

| |
|---|
| Termostato()<br>Crea un termostato estandar, i.e. en modo FRIO, de nombre "zona de estar", cuya temperatura de confort es T_IDEAL_FRIO y cuya temperatura actual es un valor aleatorio en el intervalo [20.0, 40.0[ |
| Termostato(int m, java.lang.String n, int tC, double tAct)<br>Crea un termostato en modo m, de nombre n, con temperatura de confort tC y temperatura actual tAct |

**Method Summary**

| | |
|---|---|
| int | diferenciaConIdeal()<br>Devuelve un entero que debe ser:<br>- Cero si la temperatura de confort es adecuada al modo, i.e. si es mayor o igual que la ideal en modo FRIO o menor o igual en modo CALOR<br>- La diferencia en valor absoluto entre las temperaturas de confort e ideal en cualquier otro caso |
| java.lang.String | getNombre()<br>Devuelve el nombre de un termostato |

   **Se pide:** implementar la clase `GestorTermostatos` que representa los termostatos existentes en un espacio mediante las componentes (atributos y métodos) que se indican a continuación.

   Recuerda que las constantes de la clase `Termostato` y de la clase `GestorTermostatos` tienen que utilizarse siempre que se requiera.

   a) (0.5 puntos) Atributos:

   - `MAX_TERMS`, una constante de clase (o estática) que representa el numero máximo de termostatos que pueden haber en un espacio y que vale 15.
   - `numTerms`, un entero en el intervalo `[0..MAX_TERMS]` que representa el número de termostatos del espacio en cada momento.
   - `terms`, un array de tipo base `Termostato`, de capacidad `MAX_TERMS`, para almacenar los termostatos que hay en el espacio en cada momento, dispuestos secuencialmente en posiciones consecutivas del array, desde la `0` hasta la `numTerms - 1` inclusive.

- `noEficientes`, entero que representa el número de termostatos que hay en el espacio en cada momento que no cumplen con las normas de eficiencia, es decir, aquellos cuya temperatura de confort no es la adecuada para su modo de trabajo (tal como lo comprueba el método `diferenciaConIdeal` de la clase `Termostato`).

b) (0.5 puntos) Un constructor por defecto (sin parámetros) que crea una gestor de termostatos vacío, con 0 termostatos.

c) (1 punto) Un método con perfil:

```
private int termostatoEnZona(String nomZona)
```

que, dado el nombre `String nomZona` de una zona del espacio, devuelve la posición del array donde se encuentra el termostato cuyo nombre corresponde a dicha zona o -1 si no está.

d) (1.5 puntos) Un método con perfil:

```
public boolean instalar(Termostato t)
```

que si no hay ningún termostato con el mismo nombre que `t`, lo añade al gestor; si existe un termostato con el mismo nombre que `t` lo remplaza por `t`. El método devuelve `true` si se ha instalado o actualizado con éxito, o `false` en caso de que no se puedan gestionar más termostatos. Nótese que se debe actualizar, si procede, el atributo `noEficientes`, y que se debe usar el método privado `termostatoEnZona` para buscar el termostato `t` por su nombre.

e) (1.5 puntos) Un método con perfil:

```
public Termostato diferenciaMayor()
```

que devuelve el `Termostato` con mayor diferencia en valor absoluto entre las temperaturas de confort e ideal, o `null` si no hay ningún `Termostato`.

f) (1.5 puntos) Un método con perfil:

```
public Termostato[] termsNoEficientes()
```

que devuelve un array de `Termostato` con los termostatos no eficientes. La longitud de este array será igual al número de termostatos no eficientes, o 0 si no hay ninguno.

**Solución:**

```
/**
 * Class GestorTermostatos: representa un dispositivo que gestiona
 * todos los termostatos de un espacio.
 *
 * @author Examen IIP
 * @version Segundo Parcial - Curs 2015-2016
 */
public class GestorTermostatos {
    /** Constante que representa el numero maximo de
     *  termostatos que puede haber en un espacio */
    public static final int MAX_TERMS = 15;
    // entero en [0..MAX_TERMS] que representa el numero de
    // termostatos que tiene el espacio en cada momento
    private int numTerms;
    // array de objetos Termostato, de capacidad MAX_TERMS en
    // el que los termostatos se almacenan secuencialmente, en
    // posiciones consecutivas desde la 0 hasta la numTerms - 1
    private Termostato[] terms;
    // entero que representa el numero de termostatos que no cumplen
    // las recomendaciones de eficiencia energetica
    private int noEficientes;
```

```
    /** Constructor por defecto que crea el gestor vacio, sin ningun termostato */
    public GestorTermostatos() {
        terms = new Termostato[MAX_TERMS];
        numTerms = 0;
        noEficientes = 0;
    }

    /** Devuelve la posicion del array en la que se encuentra el Termostato
     *  cuyo nombre es nomZona, o -1 si no esta.
     *  @param z String, nombre de la zona de la casa.
     *  @return int, posicion del array terms o -1 si no esta.
     */
    private int termostatoEnZona(String z) {
        int i = 0;
        while (i < numTerms && !terms[i].getNombre().equals(z)) { i++; }
        if (i < numTerms) { return i; }
        else { return -1; }
    }

    /** Si no hay ningun Termostato con el mismo nombre que t, lo añade al
     *  gestor; si existe un Termostato con el mismo nombre que t lo
     *  reemplaza por t. Devuelve true si se ha instalado/actualizado con
     *  exito o false en caso de que no se puedan gestionar mas termostatos.
     *  Se debe actualizar, si procede, el atributo noEficientes.
     *  @param t Termostato, el termostato a instalar/actualizar.
     *  @return boolean.
     */
    public boolean instalar(Termostato t) {
        boolean cabe = true;
        int pos = termostatoEnZona(t.getNombre());
        if (pos != -1) {
            if (terms[pos].diferenciaConIdeal() != 0) { noEficientes--; }
            terms[pos] = t;
        }
        else if (numTerms < MAX_TERMS) { terms[numTerms++] = t; }
        else { cabe = false; }
        if (cabe && t.diferenciaConIdeal() != 0) { noEficientes++; }
        return cabe;
    }

    /** Devuelve el Termostato con mayor diferencia en valor absoluto
     *  entre las temperaturas de confort e ideal o null si no hay
     *  ninguno.
     *  @return Termostato.
     */
    public Termostato diferenciaMayor() {
        Termostato tMax = null;
        if (numTerms != 0) {
            tMax = terms[0];
            int difMax = tMax.diferenciaConIdeal();
            for (int i = 1; i < numTerms; i++) {
                int dif = terms[i].diferenciaConIdeal();
                if (dif > difMax) { tMax = terms[i]; difMax = dif; }
            }
        }
        return tMax;
    }

    /** Devuelve un array de Termostato con los termostatos de la casa
     *  no eficientes. La longitud de este array sera igual al numero
     *  de termostatos no eficientes, o 0 si no hay ninguno.
     *  @return Termostato[].
     */
    public Termostato[] termsNoEficientes() {
        Termostato[] aux = new Termostato[noEficientes];
        int k = 0;
        for (int i = 0; i < numTerms && k < noEficientes; i++) {
            if (terms[i].diferenciaConIdeal() != 0) {
                aux[k++] = terms[i];
            }
        }
        return aux;
    }
}
```

2. 1.75 puntos Escribir un método Java estático que, dado un entero n $\geq$ 2, escriba en la salida una figura de n líneas, con dos diagonales que se junten en la última línea, sobre un fondo rectangular de `'-'`, de manera que:
   - En cada línea se deben escribir dos `'V'` separadas por un número de `'-'` cada vez menor,
   - en la primera línea la primera `'V'` aparece pegada al margen izquierdo, y la segunda en el extremo derecho de la figura.

   Ejemplo para n=5:

```
V--------V
-V------V-
--V----V--
---V--V---
----VV----
```

**Solución:**

```
/** Escribe en la salida estandar una figura de n lineas (n >=2),
 *  formada por dos diagonales de 'V' que se juntan en la ultima
 *  linea, sobre un fondo rectangular de '-'.
 */
public static void escribeV(int n) {
    // g1 es el numero de guiones que hay que escribir
    // en cada linea delante de la primera 'V' y detras de la segunda;
    // g2 es el numero de guiones que hay que escribir
    // en cada linea entre las dos 'V';
    int g1 = 0, g2 = 2 * n - 2;
    while (g1 < n) {
        for (int i = 1; i <= g1; i++) { System.out.print('-'); }
        System.out.print('V');
        for (int i = 1; i <= g2; i++) { System.out.print('-'); }
        System.out.print('V');
        for (int i = 1; i <= g1; i++) { System.out.print('-'); }
        System.out.println();
        g1++; g2 = g2 - 2;
    }
}
```

La siguiente solución alternativa tiene en cuenta que, para la línea `i`, el número de guiones a escribir en los extremos y en el centro se pueden calcular como `i` y `2*(n-i)-2` respectivamente.

```
public static void escribeV(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) { System.out.print('-'); }
        System.out.print('V');
        for (int j = 0; j < (2 * (n - i) - 2); j++) { System.out.print('-'); }
        System.out.print('V');
        for (int j = 0; j < i; j++) { System.out.print('-'); }
        System.out.println();
    }
}
```

3. 1.75 puntos Se pide escribir un método Java estático que, dado un array de `double`, compruebe si las componentes de índice par aparecen ordenadas ascendentemente. Ejemplos:

Para

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **1.5** | 0.0 | **3.0** | -1.0 | **3.5** | 2.0 |

y

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **3.0** | 0.0 | **4.5** | -1.0 | **6.5** | 2.0 | **8.5** |

debe dar `true`.

Para

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| **1.5** | 0.0 | **3.0** | -1.0 | **1.5** | 2.0 |

y

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **3.0** | 0.0 | **1.0** | -1.0 | **6.5** | 2.0 | **8.5** |

debe dar `false`.

**Solución:**

```
/** Comprueba si las componentes de indice par de a
 *  estan ordenadas ascendentemente.
 */
public static boolean enOrdenPares(double[] a) {
    int i = 0;
    while (i < a.length - 2 && a[i] <= a[i + 2]) { i = i + 2; }
    return (i >= a.length - 2);
}
```