# Segundo Parcial de IIP - ETSInf
## Fecha: 12 de enero de 2015. Duración: 2:30 horas.

1. 6.5 puntos Se dispone de la clase `Bloque` (que permite representar bloques apilables en torres de un juego), ya conocida y de la que se muestra a continuación un resumen de su documentación:

### Field Summary

**Fields**

| Modifier and Type | Field and Description |
|---|---|
| `static int` | **`AZUL`**<br>Constante que indica que el Bloque es de color azul. |
| `static int` | **`ROJO`**<br>Constante que indica que el Bloque es de color rojo. |

### Constructor Summary

**Constructors**

| Constructor and Description |
|---|
| **`Bloque`**`()`<br>Crea un Bloque de color azul que no es un comodin y cuya dimension es un entero aleatorio dentro del rango [1,50]. |
| **`Bloque`**`(int color, int dimension, boolean comodin)`<br>Crea un Bloque con valores de color, dimension y comodin dados. |

### Method Summary

**Methods**

| Modifier and Type | Method and Description |
|---|---|
| `boolean` | **`equals`**`(java.lang.Object o)`<br>Comprueba si el Bloque en curso es igual a otro dado; es decir, si coinciden en el color y la dimension y los dos son o no comodines. |
| `int` | **`getColor`**`()`<br>Devuelve el color del Bloque en curso. |
| `boolean` | **`getComodin`**`()`<br>Comprueba si el Bloque en curso es comodin. |
| `int` | **`getDimension`**`()`<br>Devuelve la dimension del Bloque en curso. |
| `boolean` | **`puedeEstarEncimaDe`**`(Bloque b)`<br>Comprueba si el Bloque en curso puede estar encima de un Bloque dado; un Bloque a puede estar encima de un Bloque b si y solo si la dimension del bloque a es menor o igual que la del bloque b y, o bien a es un comodín, o bien los colores de a y b son distintos. |
| `java.lang.String` | **`toString`**`()`<br>Devuelve un String con la informacion del Bloque en curso en un formato como el mostrado en los siguientes ejemplos: "(Color: rojo, dimension: 22 y SI es comodin)", "(Color: azul, dimension: 15 y NO es comodin)". |

Para formar una torre de bloques hay que respetar las siguientes reglas:

- Los bloques apilados en una torre deben seguir colores alternos (encima de un bloque azul solo puede haber un bloque rojo y viceversa).
- Encima de un bloque de dimensión $x$ solo puede haber un bloque de dimensión $y$, donde $y \leq x$ (la torre se estrecha hacia la punta, es decir, se ensancha hacia la base).
- Un bloque puede ser un **comodín**, en cuyo caso puede ir encima de cualquier otro bloque independientemente de su color. Ahora bien, un bloque comodín debe de respetar, como cualquier otro, la regla de la dimensión.

**Se pide:** implementar la clase `TorreBloques` que representa una torre de bloques mediante las componentes (atributos y métodos) que se indican a continuación.

Recuerda que las constantes de la clase `Bloque` y de la clase `TorreBloques` tienen que utilizarse siempre que se requiera.

a) (0.5 puntos) Atributos:

- `MAX_BLOQUES`, una constante de clase (o estática) que representa el número máximo de bloques de una torre: 10.
- `numBloques`, un entero en el intervalo `[0..MAX_BLOQUES]` que representa el número de bloques que tiene la torre en cada momento.
- `torre`, un array de tipo base `Bloque`, de capacidad `MAX_BLOQUES`. Las componentes de este array se guardan secuencialmente siguiendo las reglas del juego, en posiciones consecutivas desde la `0` hasta la `numBloques-1`, de manera que la base de la torre será `torre[0]` y la punta será `torre[numBloques-1]`.
- `numBloquesComodin`, que representa el número de bloques de la torre que son comodín.

b) (0.5 puntos) Un constructor por defecto (sin parámetros) que crea una torre vacía, con 0 bloques.

c) (1 punto) Un método con perfil:

```
private int posicionDe(Bloque b)
```

que, dado un `Bloque b`, devuelve la posición de la primera aparición del bloque en la torre desde la base, o -1 si no está.

d) (1 punto) Un método con perfil:

```
public boolean apilar(Bloque b)
```

que devuelve `true` después de apilar el `Bloque b` dado en la torre. Si `b` no cabe o no puede estar encima del último bloque apilado, el método devuelve `false` para advertir que no se ha podido apilar. Tiene que actualizarse el atributo `numBloquesComodin` si procede.

e) (1 punto) Un método con perfil:

```
public Bloque primeroMasGrandeQue(Bloque b)
```

que devuelve el primer `Bloque` de la torre, desde la base, cuya dimensión es más grande que la del `Bloque b` dado, o `null` si no hay ninguno.

f) (1 punto) Un método con perfil:

```
public Bloque[] filtrarBloquesComodin()
```

que devuelve un array de `Bloque` con los bloques que son comodín que forman la torre. La longitud de este array será igual al número de bloques comodín, o 0 si no hay ninguno.

g) (1.5 punts) Un método con perfil:

```
public String toString()
```

que devuelve "`Torre vacía`" si la torre está vacía o, en caso contrario, devuelve un `String` con la información de los bloques que forman la torre en un formato como el que se muestra en el siguiente ejemplo para una torre con 5 bloques:

- `torre[0]`: bloque de color rojo, dimensión 15 y no es comodín.
- `torre[1]`: bloque de color azul, dimensión 10 y no es comodín.
- `torre[2]`: bloque de color azul, dimensión 7 y sí es comodín.
- `torre[3]`: bloque de color rojo, dimensión 4 y no es comodín.
- `torre[4]`: bloque de color azul, dimensión 2 y no es comodín.

El `String` resultante será:

```
AA
RRRR
CCCCCCC
AAAAAAAAAA
RRRRRRRRRRRRRR
```

donde "`C`" indica que el bloque es un comodín, "`A`" que es de color azul y "`R`" que es de color rojo.

Nótese que hay `numBloques` líneas y en cada línea aparecen tantos caracteres indicando el color/comodín como la dimensión del bloque representado.

**Solución:**

```
public class TorreBloques {
    public static final int MAX_BLOQUES = 10;
    private Bloque[] torre;
    private int numBloques, numBloquesComodin;

    public TorreBloques() {
       torre = new Bloque[MAX_BLOQUES];
       numBloques = 0;
       numBloquesComodin = 0;
    }

    private int posicionDe(Bloque b) {
        int i = 0;
        while (i < numBloques && !torre[i].equals(b)) i++;
        if (i < numBloques) return i;
        else return -1;
    }

    public boolean apilar(Bloque b) {
        boolean res = false;
        if (numBloques != MAX_BLOQUES
            && (numBloques == 0 || b.puedeEstarEncimaDe(torre[numBloques - 1]))) {
                torre[numBloques++] = b;
                if (torre[numBloques - 1].getComodin()) numBloquesComodin++;
                res = true;
        }
        return res;
    }

    public Bloque primeroMasGrandeQue(Bloque b) {
        return (numBloques != 0 && torre[0].getDimension() > b.getDimension()) ? torre[0] : null;
    }

    public Bloque[] filtrarBloquesComodin() {
        Bloque[] aux = new Bloque[numBloquesComodin];
        for (int i = 0, k = 0; k < numBloquesComodin; i++)
            if (torre[i].getComodin()) {
                aux[k] = torre[i];
                k++;
            }
        return aux;
    }

    public String toString() {
        String res = "";
        for (int i = numBloques - 1; i >= 0; i--) {
            String color = "R";
            if (torre[i].getComodin()) color = "C";
            else if (torre[i].getColor() == Bloque.AZUL) color = "A";
            for (int j = 1; j <= torre[i].getDimension(); j++)
                res += color;
            res += "\n";
        }
        return (numBloques == 0 ? "Torre vacía" : res);
    }
}
```

2. 1.75 puntos Se dice que un número entero positivo es perfecto si es igual a la suma de todos sus divisores (excepto él mismo). **Se pide:** Implementar un método de clase (o estático) que compruebe si un entero `n`, `n > 0`, es un número perfecto. Por ejemplo, si `n` es `28`, el método debe devolver `true` dado que sus divisores son `1`, `2`, `4`, `7`, `14`, cuya suma vale `28`.

**Solución:**

```
/** n > 0 */
public static boolean perfecto(int n) {
    int suma = 1, i = 2;
    while (i <= n / 2) {
        if (n % i == 0) suma += i;
        i++;
    }
    return suma == n;
}
```

3. 1.75 puntos **Se pide:** Implementar un método de clase (o estático) que tenga como parámetros un array de enteros `a` (`a.length > 0`) y un entero `p` que representa una posición válida del array (`0 ≤ p < a.length`). El método tiene que devolver el valor máximo de las sumas de los elementos del array en las posiciones previas y posteriores a la posición dada, sin incluir el elemento que ocupa dicha posición en los cálculos. Por ejemplo, dado el array `{1, 7, -2, 3, 4, 8, 1, -4}` y la posición `2`, devolverá el máximo entre `1 + 7 = 8` y `3 + 4 + 8 + 1 - 4 = 12`, es decir, `12`.

**Solución:**

```
/** a.length > 0 y 0 <= p < a.length */
public static int maximoSumaParticion(int[] a, int p) {
    int sum1 = 0, sum2 = 0;
    for (int i = 0; i < p; i++) sum1 += a[i];
    for (int i = p + 1; i < a.length; i++) sum2 += a[i];
    if (sum1 > sum2) return sum1;
    else return sum2;
}
```