Segon Parcial d'IIP - ETSInf
Data: 12 de gener de 2015. Duració: 2:30 hores.

1. 6.5 punts Es disposa de la classe `Bloc` (que permet representar blocs apilables en torres d'un joc), ja coneguda i de la que es mostra a continuació un resum de la seua documentació:

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	**`BLAU`** Constant que indica que el Bloc es de color blau
`static int`	**`ROIG`** Constant que indica que el Bloc es de color roig

Constructor Summary

Constructors

Constructor and Description
`Bloc``()` Crea un Bloc de color blau que no es un comodi i la dimensio del qual es un enter aleatori dins del rang [1,50].
`Bloc``(int color, int dimensio, boolean comodi)` Crea un Bloc amb valors de color, dimensio i comodi donats.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	**`equals`**`(java.lang.Object o)` Comprova si el Bloc en curs es igual a un altre donat; es a dir, si coincideixen en el color i la dimensio i els dos son o no comodins.
`int`	**`getColor`**`()` Torna el color del Bloc en curs.
`boolean`	**`getComodi`**`()` Comprova si el Bloc en curs es comodi.
`int`	**`getDimensio`**`()` Torna la dimensio del Bloc en curs.
`boolean`	**`potEstarDamuntDe`**`(Bloc b)` Comprova si el Bloc en curs pot estar damunt d'un Bloc donat; un Bloc a pot estar damunt d'un Bloc b si i solament si la dimensio del Bloc a es menor o igual que la del Bloc b i, be a es un comodi, o be els colors de a i b son diferents.
`java.lang.String`	**`toString`**`()` Torna un String amb la informacio del Bloc en curs en un format com el mostrat en els seguents exemples: "(Color: roig, dimensio: 22 i SI es comodi)", "(Color: blau, dimensio: 15 i NO es comodi)".

Per a formar una torre de blocs s'han de respectar les següents regles:

- Els blocs apilats en una torre han de seguir colors alterns (damunt d'un bloc blau solament pot haver-hi un bloc roig i viceversa).
- Damunt d'un bloc de dimensió x solament pot haver-hi un bloc de dimensió y, on $y \leq x$ (la torre s'estreny cap a la punta, és a dir, s'eixampla cap a la base).
- Un bloc pot ser un **comodí**, en aquest cas pot anar damunt de qualsevol altre bloc, independentment dels seus colors. Ara bé, un bloc comodí, com qualsevol altre, ha de respectar la regla de la dimensió.

Es demana: implementar la classe `TorreBlocs` que representa una torre de blocs mitjançant els components (atributs i mètodes) que s'indiquen a continuació.

Recorda que les constants de la classe `Bloc` i de la classe `TorreBlocs` s'han d'utilitzar sempre que es requerisca.

a) (0.5 punts) Atributs:

- `MAX_BLOCS`, una constant de classe (o estàtica) que representa el número màxim de blocs d'una torre: 10.
- `numBlocs`, un enter en l'interval `[0..MAX_BLOCS]` que representa el número de blocs que té la torre en cada moment.
- `torre`, un array de tipus base `Bloc`, de capacitat `MAX_BLOCS`. Els components d'aquest array s'emmagatzemen seqüencialment seguint les regles del joc, en posicions consecutives des de la `0` fins la `numBlocs-1`, de manera que la base de la torre serà `torre[0]` i la punta serà `torre[numBlocs-1]`.
- `numBlocsComodi`, que representa el número de blocs de la torre que són comodí.

b) (0.5 punts) Un constructor per defecte (sense paràmetres) que crea una torre buida, amb 0 blocs.

c) (1 punt) Un mètode amb perfil:

```
private int posicioDe(Bloc b)
```

que, donat un `Bloc b`, torna la posició de la primera aparició del bloc en la torre des de la base, o -1 si no està.

d) (1 punt) Un mètode amb perfil:

```
public boolean apilar(Bloc b)
```

que torna `true` després d'apilar el `Bloc b` donat a la torre. Si `b` no cap o no pot estar damunt del darrer bloc apilat, el mètode torna `false` per tal d'advertir que no s'ha pogut apilar. S'ha d'actualitzar l'atribut `numBlocsComodi` si procedeix.

e) (1 punt) Un mètode amb perfil:

```
public Bloc primerMesGranQue(Bloc b)
```

que torna el primer `Bloc` de la torre, des de la base, que és de dimensió més gran que el `Bloc b` donat, o `null` si no hi ha cap.

f) (1 punt) Un mètode amb perfil:

```
public Bloc[] filtrarBlocsComodi()
```

que torna un array de `Bloc` amb els blocs que són comodí que formen la torre. La longitud d'aquest array serà igual al número de blocs comodí, o 0 si no hi ha cap.

g) (1.5 punts) Un mètode amb perfil:

```
public String toString()
```

que torna "`Torre buida`" si la torre està buida o, en cas contrari, torna un `String` amb la informació dels blocs que formen la torre en un format com el que es mostra en el següent exemple per a una torre amb 5 blocs:

- `torre[0]`: bloc de color roig, dimensió 15 i no és comodí.
- `torre[1]`: bloc de color blau, dimensió 10 i no és comodí.
- `torre[2]`: bloc de color blau, dimensió 7 i sí és comodí.
- `torre[3]`: bloc de color roig, dimensió 4 i no és comodí.
- `torre[4]`: bloc de color blau, dimensió 2 i no és comodí.

El `String` resultant serà:

```
BB
RRRR
CCCCCCC
BBBBBBBBB
RRRRRRRRRRRRRR
```

on "`C`" indica que el bloc és un comodí, "`B`" que és de color blau i "`R`" que és de color roig.

Cal notar que hi ha `numBlocs` línies i en cada línia apareixen tants caràcters indicant el color/comodí com la dimensió del bloc representat.

Solució:

```
public class TorreBlocs {
    public static final int MAX_BLOCS = 10;
    private Bloc[] torre;
    private int numBlocs, numBlocsComodi;

    public TorreBlocs() {
       torre = new Bloc[MAX_BLOCS];
       numBlocs = 0;
       numBlocsComodi = 0;
    }

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

    public boolean apilar(Bloc b) {
        boolean res = false;
        if (numBlocs != MAX_BLOCS
            && (numBlocs == 0 || b.potEstarDamuntDe(torre[numBlocs - 1]))) {
                torre[numBlocs++] = b;
                if (torre[numBlocs - 1].getComodi()) numBlocsComodi++;
                res = true;
        }
        return res;
    }

    public Bloc primerMesGranQue(Bloc b) {
        return (numBlocs != 0 && torre[0].getDimensio() > b.getDimensio()) ? torre[0] : null;
    }

    public Bloc[] filtrarBlocsComodi() {
        Bloc[] aux = new Bloc[numBlocsComodi];
        for (int i = 0, k = 0; k < numBlocsComodi; i++)
            if (torre[i].getComodi()) {
                aux[k] = torre[i];
                k++;
            }
        return aux;
    }

    public String toString() {
        String res = "";
        for (int i = numBlocs - 1; i >= 0; i--) {
            String color = "R";
            if (torre[i].getComodi()) color = "C";
            else if (torre[i].getColor() == Bloc.BLAU) color = "B";
            for (int j = 1; j <= torre[i].getDimensio(); j++) res += color;
            res += "\n";
        }
        return (numBlocs == 0 ? "Torre buida" : res);
    }
}
```

2. [1.75 punts] Es diu que un número enter positiu és perfecte si és igual a la suma de tots els seus divisors (excepte ell mateix). **Es demana:** Implementar un mètode de classe (o estàtic) que comprove si un enter `n`, `n > 0`, és un número perfecte. Per exemple, si `n` és `28`, el mètode ha de tornar `true` donat que els seus divisors són `1`, `2`, `4`, `7`, `14`, la suma dels quals val `28`.

Solució:

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

3. [1.75 punts] **Es demana:** Implementar un mètode de classe (o estàtic) que reba com paràmetres un array d'enters `a` (`a.length` $>$ `0`) i un enter `p` que representa una posició vàlida dins l'array (`0` $\leq$ `p` $<$ `a.length`). El mètode ha de tornar el valor màxim de les sumes dels elements de l'array en les posicions prèvies i posteriors a la posició donada, sense incloure l'element que ocupa aquesta posició en els càlculs. Per exemple, donat l'array `{1, 7, -2, 3, 4, 8, 1, -4}` i la posició `2`, tornarà el màxim entre `1 + 7 = 8` i `3 + 4 + 8 + 1 - 4 = 12`, és a dir, `12`.

Solució:

```
/** a.length > 0 i 0 <= p < a.length */
public static int maximSumaParticio(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;
}
```