Final CAP

Curs 2015-16 (13/I/2016) Duració: 3 hores

1.- [3 punts] Doneu expressions per definir els següents *pointcuts*:

a) Qualsevol crida a un mètode definit per una subclasse de la classe A, però que NO està definit a la classe A.

```
call(* A+.*(..)) && !call(* A.*(..))
```

b) L'execució d'un mètode definit al codi font d'una classe que és subclasse de la classe A, però que NO està definit a la classe A.

```
execution(* *(..)) && within(A+) && !within(A)
```

c) Qualsevol crida a (un mètode de) una instància d'una subclasse de la classe A, que no és instància *directa* de la classe A (escric "directa" perque ja sabem que tota instància d'una classe és també instància de les corresponents superclasses).

```
call(* *(..)) && target(af) && !if(af.getClass() == A.class)
```

2.- [2 punts] Comenteu aquest aspecte. Investigueu la seva correctesa i els possibles problemes que poden aparèixer en utilitzar-lo. Com els arreglarieu?

```
aspect A {
    before(): call(* *(..)) { System.out.println("before"); }
    after():  call(* *(..)) { System.out.println("after"); }
}
```

La crida a println forma part de call (* *(..)), per tant podem tenir problemes de recursivitat infinita. Ho podem resoldre fent:

```
aspect A {
  before(): call(* *(..)) && !within(A) { System.out.println("before"); }
  after() returning:  call(* *(..)) && !within(A) { System.out.println("after");
}
```

3.- [1 punt] Quina diferència hi ha entre aquestes signatures:

a) `* * *.*(..)* *(..)`

b) `* * *.*(*,..)`

La signatura a) i b) capturen qualsevols mètodes de qualsevol tipus de retorn, qualsevol classe i qualsevol nom, excepte que a b) ha d'haver al menys un paràmetre.

4.- [2 punts] Ja sabeu que a Javascript l'abast (*scope*) de les variables és un abast de funció (*hoisting*). El nou estàndar ECMAScript 6 introdueix la possibilitat de declarar variables amb abast de bloc (utilitzant '**`let`**' en lloc de '**`var`**'), que són essencialment les que ja coneixeu i utilitzeu a Java o a C++. Expliqueu-ne la diferència i il·lustreu-ho amb un petit exemple de codi (us l'heu d'inventar).

Aquest codi us servirà per entendre la diferència:

```
function f1 () {
    let l = "let";
    console.log(l);
    console.log(v);
    var v = "var";
}
```

```
function f2 () {
   var v = "var";
   console.log(l);
   console.log(v);
   let l = "let";
}
```

Si executem f1 el resultat és

```
let
undefined
```

En canvi si executem f2 obtenim un error:

```
/home/n00ne/CAP/Tema4-Prototipus/resposta.js:10
    console.log(l);
                ^
ReferenceError: l is not defined
    at f2 (/home/n00ne/CAP/Tema4-Prototipus/resposta.js:10:17)
    at ...
```

5.- [2 punts] Suposem que tenim tres funcions constructores A, B i C. Volem que els objectes construïts per la funció C puguin utilitzar les funcionalitats que proporcionen les funcions constructores A i B (en un món OO amb classes i herència simple, diriem que C és una subclasse de B i que B és una subclasse d'A). Per exemple, si els objectes creats amb A tenen una propietat anomenada propA (de contingut inicial "a"), els objectes creats per B tenen una propietat anomenada propB (de contingut inicial "b") i els objectes creats amb C tenen una propietat anomenada propC (de contingut inicial "c"), el resultat d'executar:

```
var c = new C();
console.log(c.propA);
console.log(c.propB);
console.log(c.propC);
```

seria

```
a
b
c
```

Una possible solució seria (utilitzant la funció 'inherit' que vam veure a classe):

```
function A() {
    // . . . el que sigui
}
A.prototype.propA = "a";

function B() {
    // . . .  el que sigui
}

B.prototype = inherit(A.prototype);
B.prototype.constructor = B;
B.prototype.propB = "b";

function C() {
    // . . . el que sigui
}

C.prototype = inherit(B.prototype);
C.prototype.constructor = C;
C.prototype.propC = "c";
```