Final CAP

Curs 2016-17 (12/I/2017) Duració: 3 hores

1.- Explica el concepte de "*join point* basat en el control de flux" i fes servir aquesta explicació per diferenciar **`cflow(<Pointcut>)`** i **`cflowbelow(<Pointcut>)`**. Això ho fareu en tres parts:

1.1.- (1 punt) Utilitzeu l'aspecte (que ja teniu perquè us el vaig passar, però per si de cas us el llisto):

```
public aspect JoinPointTraceAspect {
    private int callDepth;
    pointcut traced() : !within(JoinPointTraceAspect);
    before() : traced() {
      print("Before", thisJoinPoint);
      callDepth++;
    }
    after() : traced() {
      callDepth--;
      print("After", thisJoinPoint);
    }
    private void print(String prefix, Object message) {
      for (int i = 0; i < callDepth; i++) {
          System.out.print(" ");
      }
      System.out.println(prefix + ": " + message);
    }
}
```

per fer una traça de tots els *join points* que hi ha a l'execució de la classe:

```
public class FinalEx {
    public static void main (String[] args) {
      int x = f();
      System.out.println("==> "+x);
    }
    static int f() {
      int x = 1;
      return g(x);
    }
    static int g(int x) {
      int y = 2*x;
      return h(y);
    }
    static int h(int x) {
      return x-1;
    }
}
```

Modifiqueu l'aspecte per a que us mostri només els *join points* que corresponen a **`cflow(call(* *.f()))`** i modifiqueu l'aspecte un altre cop pels *join points* de **`cflowbelow(call(* *.f()))`**.

Una possibilitat (no l'única) és fer: **`traced() && cflow(call(* *.f()))`** després del **`before`** i l'**`after`**. Així veiem que en un cas tenim:

```
Before: call(int FinalEx.f())
 Before: execution(int FinalEx.f())
  Before: call(int FinalEx.g(int))
   Before: execution(int FinalEx.g(int))
    Before: call(int FinalEx.h(int))
     Before: execution(int FinalEx.h(int))
     After: execution(int FinalEx.h(int))
    After: call(int FinalEx.h(int))
   After: execution(int FinalEx.g(int))
  After: call(int FinalEx.g(int))
 After: execution(int FinalEx.f())
After: call(int FinalEx.f())
==> 1
```

i en l'altre cas tot és igual excepte que les línies vermelles no hi són.

1.2.- (2 punts) Si voleu podeu jugar i investigar una mica amb l'exemple de l'apartat 1.1, però un cop penseu que ho teniu clar, expliqueu, *en general* (no només per a l'exemple anterior), quines són les diferències entre **`cflow(<Pointcut>)`** i **`cflowbelow(<Pointcut>)`**.

La idea és que cada *join point* correspón a un lloc determinat dins del control de flux, proporcionant un marcador al qual li podem assignar un cert abast (*scope*). El *join point* inicial (corresponent a la definicio del *pointcut* argument de **`cflow`** o **`cflowbelow`**) defineix aquest abast, incloent-hi qualsevol joint point que hi trobem dins. La diferència entre **`cflow(<Pointcut>)`** i **`cflowbelow(<Pointcut>)`** és que aquest *join point* inicial no és inclós en el cas del **`cflowbelow(<Pointcut>)`**.

1.3.- (1 punt) Ara modifiqueu l'aspecte de l'apartat 1.1 per a que mostri els *join points* que corresponen a **`cflowbelow(execution(* *.f()))`**. El que es mostra és coherent amb la teva explicació de l'apartat 1.2?

```
Before: call(int FinalEx.g(int))
 Before: execution(int FinalEx.g(int))
  Before: call(int FinalEx.h(int))
   Before: execution(int FinalEx.h(int))
   After: execution(int FinalEx.h(int))
  After: call(int FinalEx.h(int))
 After: execution(int FinalEx.g(int))
After: call(int FinalEx.g(int))
==> 1
```

Si, perfectament coherent. Aquest exemple no afegeix cap "sorpresa" respecte del que hem dit abans.

2.- (2 punts) Defineix un *pointcut* que triï els *join points* corresponents a l'execució *de primer nivell* d'una funció recursiva. És a dir, per a cada vegada que cridem la funció, només ens interessa la crida que *no* és recursiva, les crides que la funció fa recursivament *no* les volem considerar.

```
pointcut cridesPrimerNivell() : execution(* *.f(..)) &&
                                !cflowbelow(execution(* *.f(..)));
```

3.- (2 punts) Sabem que (quasi) tot objecte en Javascript té un prototipus (un altre objecte al que fa referència). I sabem que tot objecte-funció (objectes invocables) conté una propietat anomenada **`prototype`**. Aleshores, respón a aquestes qüestions:

a) En general, el prototipus d'un objecte-funció i el **`prototype`** d'aquest objecte-funció són el mateix objecte?

b) Hi ha cap excepció a la regla general?

c) Per a què serveix el **`prototype`** d'un objecte-funció?

a) En general són objectes diferents

b) Però hi ha el cas excepcional de l'objecte **`Function`**, que té **`Function.prototype`** com a prototipus.

c) Serveix per determinar qui serà el prototipus dels objectes nous creats quan s'invoqui l'objecte-funció amb la paraula clau **`new`**.

4.- (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
```

Dóna una possible definició per a A, B i C per a que es verifiquin les condicions de l'enunciat.

Una solució seria (amb 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";
```