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()))`**.

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>)`.

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?

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.

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?

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.