Parcial CAP

Curs 2016-17 (28/XI/2016) Duració: 2 hores

1.- (2 punts) Defineix *introspecció* i *intercessió*. Dóna exemples d'introspecció i intercessió en Smalltalk i Java (un de cada a cada llenguatge).

Les definicions són a les transparències.
- Introspecció:
  Smalltalk: Poder *veure* la pila d'execució amb **`thisContext`**.
  Java: Poder saber quins mètodes i atributs té una classe a Java (amb **`getDeclaredMethods`** i **`getDeclaredFields`** de **`Class`**).
- Intercessió:
  Smalltalk: Poder *canviar* la pila d'execució, gràcies a **`thisContext`**.
  Java: Poder interceptar les invocacions a mètode amb l'ús de la classe **`java.lang.reflect.Proxy`**.

2.- (2 punts) Si us fixeu, una instància de la fàbrica d'objectes (l'exemple **`DogFactory`** de les transparències) o bé sempre crea objectes *proxificats* o bé sempre crea objectes *normals* (depén del booleà **`trace`** al constructor). Expliqueu quines modificacions caldria fer a la fàbrica d'objectes per, donada una instància de la fàbrica d'objectes modificada, poder triar si l'objecte creat és *proxificat* o *normal*.

Només cal eliminar l'atribut booleà de la classe i el paràmetre booleà del constructor i afegir un paràmetre booleà a **`newInstance`**. Així podrem crear objectes *proxificats* o *normals* depenent d'aquest paràmetre booleà.

3.- (1 punt) Escriviu un programa que, passant el nom d'una classe com a paràmetre de la línia de comandes, digui si aquesta classe implementa dos o més interficies.

```
import java.lang.reflect.*;

public class UnaInterficie {
    public static void main(String args[]) {
            if (una_o_mes(Class.forName(args[0]))) {
                  System.out.println("SI, implementa dos o mes interficies");
            } else {
                  System.out.println("NO, implementa un o cap interficie");
            }
    }
    private static boolean una_o_mes (Class c) {
      // suposem que (c != null)
      return (c.getInterfaces().length > 1);
    }
}
```

4.- (2.5 punts) Recordeu l'exemple de la classe **`BlockWithExit`** que vam veure a classe:

```
| theLoop coll |
Transcript open.
coll := OrderedCollection new.
1000 timesRepeat: [ coll add: 1000 atRandom ].
theLoop := [coll do: [:each  | Transcript show: each asString; cr.
                               (each < 100) ifTrue: [theLoop exit]]] withExit.
theLoop value.
```

Ara volem prescindir de la variable **`theLoop`**. Això vol dir que, d'entrada, NO sabem a qui hem d'enviar el missatge **`exit`** per sortir del bucle. Malgrat tot, si fem servir l'*inspector* i altres eines de l'entorn de Pharo, podem investigar i averiguar-ho. Així doncs, omple l'espai buit per a que aquest fragment de codi faci *exactament* el mateix que feia el de l'exemple:

```
| coll |
coll := OrderedCollection new.
1000 timesRepeat: [ coll add: 1000 atRandom ].
([coll do: [:each | Transcript show: each asString; cr.
  (each < 100) ifTrue: [(_____???????_____) exit ]]] withExit) value.
```

Només cal averiguar on para la instància de **`BlockWithExit`** a qui hem enviat el missatge **`value`**, i saber què fan els mètodes **`sender`** i **`receiver`** de la classe **`MethodContext`**.

```
| coll |
coll := OrderedCollection new.
1000 timesRepeat: [ coll add: 1000 atRandom ].
([coll do: [:each | Transcript show: each asString; cr.
  (each < 100)
    ifTrue: [(thisContext sender sender sender receiver) exit ]]] withExit) value.
```

5.- (2.5 punts) Si us heu llegit el capítol 14 de *Deep into Pharo* (com us he demanat repetides vegades), no tindreu problema a respondre aquesta pregunta. Canvieu la implementació del mètode **`#value`** de **`BlockWithExit`**, originalment definit com:

```
value
     exitBlock := [^ nil].
     ^ block value.
```

per a que utilitzi *continuacions* en lloc de guardar el bloc **`[^ nil]`**.

Aquest problema és senzill. Només cal capturar el retorn del mètode **`#value`**, així:

```
value
     ^ Continuation callcc: [:cc | exitBlock := [ cc value: nil].
                                   block value]
```