Parcial CAP

Curs 2018-19 (15/XI/2018) **Duració: 2 hores**

1.- (1 punt) Defineix *introspecció*, *intercessió*, digues què significa *reificar* i explica per quina raó no podem fer intercessió de la pila d'execució en Java.

Solució: Les definicions són a les transparències, i la raó per la que no podem fer intercessió de la pila d'execució en Java és per què no hi ha connexió causal entre la representació de la pila d'execució (array d'**`StackTraceElement`**) i la pila d'execució real.

2.- (1.5 punts) Si seleccionem aquest programa al Workspace i fem **`Ctrl-p`**...

```
| collection |
collection := OrderedCollection new.
#(1 2 3) do: [ :index |
      | temp |
      temp := index.
      collection add: [ temp ] ].
collection collect: [ :each | each value ]
```

el resultat és: **`an OrderedCollection(1 2 3)`**. En canvi, si seleccionem aquest programa al Workspace i fem **`Ctrl-p`**...

```
| collection temp |
collection := OrderedCollection new.
#(1 2 3) do: [ :index |
      temp := index.
      collection add: [ temp ] ].
collection collect: [ :each | each value ]
```

el resultat és: **`an OrderedCollection(3 3 3).`**

Explica i justifica la diferència en el resultat.

Solució: En el primer cas temp és una variable local al bloc, i per tant diferent cada cop que s'avalua el block, en el segon cas temp és una variable lliure en el bloc i per tant és capturada des de l'exterior del bloc (blocs = *closures*). En aquest cas és la mateixa variable cada cop que s'avalua el bloc.

3.- (1.5 punts) A la plana 318 del capítol 14 del llibre *Deep into Pharo*, anomenat *Blocks: a Detailed Analysis*, quan fa referència a l'ús del retorn (recordeu, **`^ expressió`**) dins d'un bloc (és a dir, quelcom similar a **`[... ^ expressió ]`**), diu:

The evaluation of the block returns to the ***block home context sender*** *(i.e., the context that invoked the method creating the block)*

I teniu un exemple bastant aclaridor del que això significa. A la plana 320 teniu explicats els riscos d'utilitzar retorns dins de blocs.

Vull que escriviu codi que il·lustri el cas en que l'ús del retorn dins d'un bloc surt malament, és a dir, que genera un error (i que no sigui, literalment, l'exemple que hi ha en el llibre).

Solució: Aquí les solucions poden ser molt diverses. N'hi ha prou amb que el bloc amb el retorn s'avaluï un cop hagi desaparegut el context que correspón a la crida al mètode on s'ha creat el bloc. Per exemple:

```
Classe1 >> creaBlocIRetorna
      ^ [ 'Es crea el bloc' traceCr. ^ 0 ]
Classe2 >> avaluaBloc: unBloc
      unBloc value
```

I executem al Workspace:

```
| c |
c := Classe1 new creaBlocIRetorna.
Classe2 new avaluaBloc: c.
```

Apareix un error **BlockCannotReturn**.

4.- (1.5 punts) Explica i justifica la relació entre aquesta invocació a **#callcc:**

```
Continuation callcc: [ :k | expressió ]
```

i aquesta invocació a **#callcc:**

```
Continuation callcc: [ :k | k value: expressió ]
```

Per simplificar suposarem que **no** s'invoca **k** dins d'***expressió***

Solució: Produeixen el mateix resultat:

Continuation callcc: [:k | *expressió*] avalua el bloc **[:k | *expressió*]** passant com a paràmetre la continuació corresponent i retorna el resultat d'avaluar ***expressió***.

Continuation callcc: [:k | k value: *expressió*] avalua el bloc **[:k | k value: *expressió*]** passant com a paràmetre la continuació corresponent i passa com a paràmetre per a **k** el resultat d'avaluar ***expressió***. Aquesta és retornada tot just en el mateix lloc on s'ha realitzat la invocació a **#callcc:**

5.- (1.5 punts) Escriviu un programa en Java que, passant el nom d'una classe com a paràmetre de la línia de comandes, digui si aquesta classe és abstracta.

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

public class EsAbstracta {
    public static void main(String args[]) {
            if (resposta(Class.forName(args[0]))) {
                  System.out.println("SI, és abstracta");
            } else {
                  System.out.println("NO, no és abstracta");
            }
    }
    private static boolean resposta (Class c) {
      if (c == null) {
            return false;
      }
      return Modifier.isAbstract(c.getModifiers());
    }
}
```

6.- (3 punts) Definirem dues versions del **`#whileTrue:`** amb continuacions que ja vam veure a classe:

a)
```
BlockClosure >> whileTrueCCa: aBlock
    | cont tmp |
    tmp := 0.                "Aquí està l'única diferència"
    cont := Continuation callcc: [ :cc | cc ].
    self value ifTrue: [ aBlock value.
                Transcript show: ('inside whileTrueCC: ', tmp asString);cr.
                tmp := tmp + 1.
                cont value: cont]
            ifFalse: [^ nil].
```

b)
```
BlockClosure >> whileTrueCCb: aBlock
    | cont tmp |
    [ tmp := 0 ] value.      "Aquí està l'única diferència"
    cont := Continuation callcc: [ :cc | cc ].
    self value ifTrue: [ aBlock value.
                Transcript show: ('inside whileTrueCC: ', tmp asString);cr.
                tmp := tmp + 1.
                cont value: cont]
            ifFalse: [^ nil].
```

Si ara avaluem al Workspace:

```
| n |
n := 4.
[ n > 0 ] whileTrueCCa: [ n := n-1 ]
```

el resultat és:

```
inside whileTrueCC: 0
inside whileTrueCC: 0
inside whileTrueCC: 0
inside whileTrueCC: 0
```

Si ara faig el mateix al Workspace, però amb **`BlockClosure >> #whileTrueCCb:`** :

```
| n |
n := 4.
[ n > 0 ] whileTrueCCb: [ n := n-1 ]
```

el resultat és:

```
inside whileTrueCC: 0
inside whileTrueCC: 1
inside whileTrueCC: 2
inside whileTrueCC: 3
```

Mireu d'entendre què passa i especuleu sobre les possibles raons d'aquest comportament. És una pregunta oberta, ja que la resposta no s'ha explicat a classe, però amb el que sabeu haurieu de poder deduir-ne una resposta aproximada.

Solució: La variable **`tmp`** en un cas és capturada per un bloc i en l'altre... també! Vull dir, en tots dos casos **`tmp`** es fa servir dins el bloc del **`ifTrue:`**, així que *en teoria* hauria d'haver estat capturada per aquest bloc (com a *closure* que és) en tots dos casos i s'hauria d'haver guardat en un *array* al *heap*, fora dels contextos de la pila d'execució. Així doncs, *en teoria*, hauríem d'haver observat el comportament de **`#whileTrueCCb:`** en tots dos casos.

El problema és que per questions d'eficiència, els blocs de l'**`ifTrue:`** (i de l'**`ifFalse:`** i d'altres missatges importants i utilitzats freqüentment)... *no són blocs de veritat!!* Aquests blocs es transformen *inline* i no són *closures* com cal. Per això, el bloc de l'**`ifTrue:`** a **`#whileTrueCCa:`** en realitat no captura la variable **`tmp`**, i per això observem la diferència en comportament. A **`#whileTrueCCb:`** la variable **`tmp`** *sí que és capturada* per un bloc **`[ tmp := 0 ] value`**, explícitament.