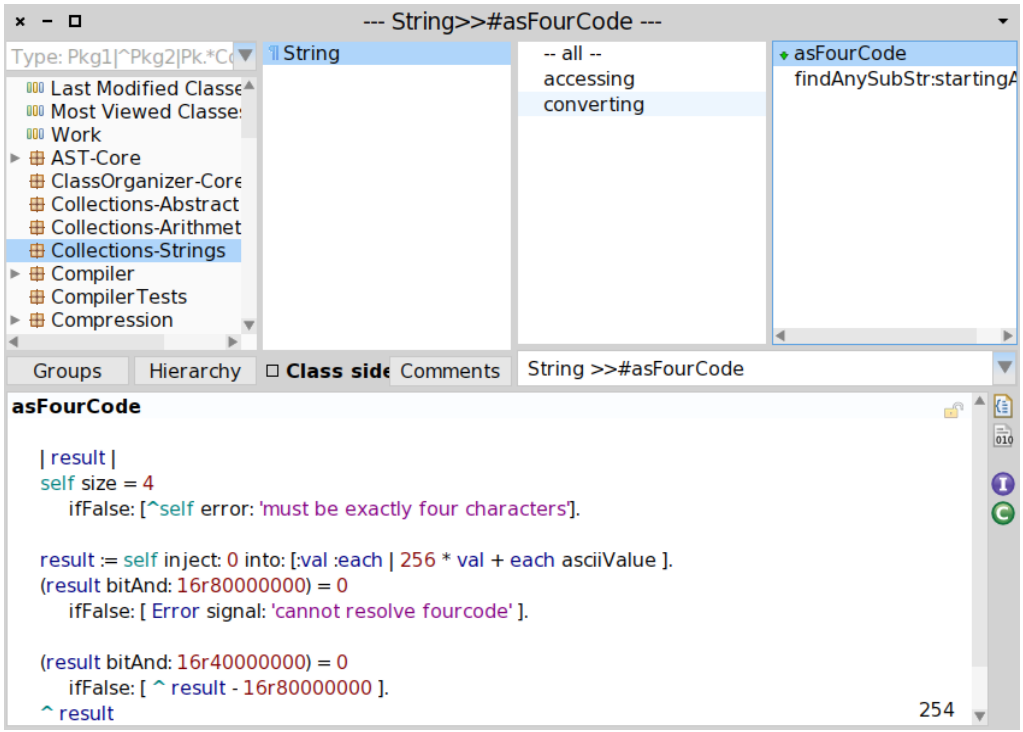
FIB

CONCEPTES AVANÇATS DE PROGRAMACIÓ
Tema 4
Col·lecció de Problemes

Recull per Jordi Delgado
(Dept. CS, UPC)

Grau d'EI, 2023-24
FIB (UPC)

1. Fes un programa (per ser executat al *Playground*) que, donat un símbol que representarà el selector d'un mètode, generi un *browser* amb tots els mètodes que l'utilitzen dins el seu codi font. Per exemple, si el símbol fos **`#inject:into:`** el resultat hauria de ser:



   És a dir, un *browser* només amb els mètodes (i classes i paquets corresponents) que fan servir **`#inject:into:`**.

2. Fes petits bocinets de Smalltalk (per ser executats al *Playground*) per respondre les següents qüestions:

- Per a cada classe de la imatge, escriure al **`Transcript`** el nom de la classe i la quantitat de mètodes que té (en un format com, p.ex. **`'OrderedCollection -- 61'`**).
- Obtenir una col·lecció amb tots els mètodes que fan servir en el seu codi **`#callcc:`**
- Trobar quantes classes implementen el mètode de selector **`#name`**
- Trobar quantes classes tenen al menys un mètode sobreescrit (*overriden*) en alguna subclasse.

   Nota: *Recordeu que totes les classes hereten els getters per a les variables d'instància de la classe* **`Behavior`**, *superclasse de totes les metaclasses. També us pot interessar mirar si la classe* **`CompiledMethod`** *té algún mètode que us pugui ser útil.*

3. Per quina raó s'avalua a **`true`** la següent expressió?
   `Class class class class == Class class class class class class`

4. Els mètodes **`#new`** i **`#new:`** són mètodes d'instància de la classe **`Behavior`**, tot i que és habitual que se'ls redefineixi en altres classes. Malgrat són mètodes

d'instància de **`Behavior`**, usualment es redefineixen com a mètodes de classe. Per exemple, utilitzem **`Array new: 5`** per crear instàncies de la classe **`Array`**, i el missatge **`#new:`** l'estem enviant a la classe **`Array`**. Les redefinicions, doncs, s'acostumen a fer en el *Class side*. Això aparentment viola la "*regla*" que diu que en l'herència els mètodes d'instància s'hereten en l'*instance side* i els mètodes de classe s'hereten en el *Class side*. Expliqueu per quina raó no hi ha res d'incorrecte en el fet de redefinir **`#new`** i **`#new:`** en el *Class side*.

5. Un bloc en Smalltalk té variables *locals* (paràmetres, variables declarades dins el bloc), i variables *lliures*, que pertanyen al context en el que s'ha creat el bloc, però no són declarades *dins* el bloc. Sabem que els blocs en Smalltalk són realment el que hem anomenat *closures*. Aleshores:

a) Què significa que un bloc d'Smalltalk sigui en realitat una *closure*?

b) La pseudo-variable **`self`** es pot fer servir dins un bloc, si aquest es crea dins un mètode. Aquesta pseudo-variable és capturada per un bloc/*closure* que la faci servir? Dona un exemple que justifiqui la teva resposta.

6. 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).

7. Expliqueu com un bloc pot cridar-se a ell mateix (aconseguint així blocs sense nom, anònims, però recursius). *No podeu suposar que el bloc ha estat assignat a una variable*. Feu-ho servir per fer un bloc que calculi el factorial.

8. Escriviu un mètode **`find: aString`** que, *enviat a una classe*, retorni una col·lecció de selectors tal que els mètodes corresponents contenen **`aString`** dins del seu codi font.

Exemple: El resultat d'executar al *playground* **`Object find: 'reflect'`** hauria de ser una col·lecció amb els selectors de tots aquells mètodes d'**`Object`** tals que la *string* '*reflect*' apareix en el seu codi font. En aquest cas obtindriem
`#(#perform:withArguments:inSuperclass: #perform:with: #perform:withArguments: #perform:with:with: #perform: #perform:with:with:with:)`

A quina classe cal posar aquest mètode per a que *qualsevol* classe o metaclasse del sistema sigui capaç d'executar-lo? I on el posem, a l'*instance side* o al *Class side*?

9. Escriviu un fragment de codi (per ser executat en el *playground*) que ens digui quants mètodes tenen la **`String`** '*this*' en el seu codi font.

10. A classe vam veure un mètode anomenat **`#haltIf: aSymbol`** que permetia fer un *halt* només quan a la cadena de crides hi havia un mètode amb **`aSymbol`** com a nom. Feu ara un mètode **`#haltIfTrue: aBlock`** on **`aBlock`** és un bloc sense paràmetres. Aleshores, **`self haltIfTrue: [...]`** s'aturarà només si el resultat d'avaluar **`[...]`** és **`true`**. A quina classe ha d'anar aquest mètode?

11. a) Què fa aquest codi?

```
#aa bindTo: 1 in: [
      Transcript show: #aa binding asString.
      #aa bindTo: 2 in: [
            Transcript show: #aa binding asString. ].
      Transcript show: #aa binding asString , 'again'. ].
```

b) I aquest codi?

```
#aa bindTo: '1' in: [
 #bb bindTo: 'a' in: [
  Transcript show: #aa binding  , '-' , #bb binding.
  #aa bindTo: '2' in: [
   #bb bindTo: 'b' in: [
      Transcript show: #aa binding  , '-' , #bbbinding.].
   Transcript show: #aa binding  , '-' , #bb binding. ].
  #bb bindTo: 'b' in: [
     Transcript show: #aa binding  , '-' , #bb binding ]]]
```

12. Executeu aquest codi al *Playground*:

```
| globalVariable bloc b |
globalVariable := 'Primera assignació'.
bloc := [ :s |  [ (globalVariable , ' ', s) traceCr ] ].
b := bloc value: '1'.
b value.
globalVariable := 'Segona assignació'.
b value.
b := bloc value: '2'.
b value.
```

Què surt? Explica amb precisió, però també breument, per què observeu el que observeu.

13. Si seleccionem aquest programa al **`Playground`** i fem **`Ctrl-d`**...

```
| comptador reset incrementa decrementa |
comptador := [ | valor |
```

```
      { [ valor := 0. valor traceCr. ] .
        [ :n | valor := valor + n. valor traceCr. ] .
        [ :n | valor := valor - n. valor traceCr. ] } ] value.
reset := comptador at: 1.
incrementa := comptador at: 2.
decrementa := comptador at: 3.
reset value.
incrementa value: 2.
incrementa value: 2.
decrementa value: 1.
```

apareixen al **`Transcript`** els números **`0 2 4 3`** (en aquest ordre).
En canvi, si seleccionem aquest programa al *Playground* i fem **`Ctrl-d`**...

```
| comptador reset incrementa decrementa |
comptador := [ {
         [ | valor | valor := 0. valor traceCr. ] .
        [ :n | | valor | valor := valor + n. valor traceCr. ] .
        [ :n | | valor | valor := valor - n. valor traceCr. ] } ] value.
reset := comptador at: 1.
incrementa := comptador at: 2.
decrementa := comptador at: 3.
reset value.
incrementa value: 2.
incrementa value: 2.
decrementa value: 1.
```

el programa no funciona. Apareix un error (**`#+ was sent to nil`**).
Explica i justifica la diferència en el resultat.

14. Afegiu un mètode a la classe **`#BlockClosure`** anomenat **`#whileWithBreak:`** que funcioni de manera similar al mètode **`#whileTrue:`** que ja coneixem: El receptor del missatge és un bloc sense paràmetres que s'ha d'avaluar a un booleà, i l'argument és un bloc sense paràmetres que representa el cos del bucle, que es repetirà mentre el receptor sigui **`true`**.

La diferència és que a **`#whileWithBreak:`** dins el cos del bucle (l'argument) podem utilitzar la següent expressió:

`(#breakWhile binding) value: <expressió>` ⇨

> Atura el bucle **`whileWithBreak:`** i retorna el valor resultant d'avaluar **`<expressió>`**

Fixeu-vos que, al contrari que **`#whileTrue:`**, de **`#whileWithBreak:`** s'espera que retorni un resultat. També, fixeu-vos que necessitareu utilitzar variables dinàmiques amb la classe **`Binding`** que ja vam veure a classe.

Mireu aquest exemple, similar a l'exemple que vam veure de **`BlockWithExit`** (classe que **no** podeu fer servir en aquesta resposta), excepte que **`#whileWithBreak:`** aquí retorna **`true`** si troba un nombre menor que 100:

```
| index found coll |
coll := Array new: 1000.
1 to: 1000 do: [ :i | coll at: i put: 1000 atRandom ].
index := 1.
found := [ index < 1000 ] whileWithBreak: [
```

```
| each |
each := coll at: index.
each traceCr.
(each < 100) ifTrue: [ (#breakWhile binding) value: true ].
index := index + 1 ].
```

Aquest exemple escriu al **Transcript** els nombres triats a l'atzar que hi ha a **coll** (que aquí és un **Array**) fins que troba un nombre menor que 100, moment en que atura el bucle retornant **true**, que serà el nou valor de la variable **found**.

15. Imaginem una expressió:

Continuation callcc: [:k | ... <cos del bloc>...]

on a **<cos del bloc>** s'ignora el paràmetre **k** del bloc (és a dir, no fem *absolutament res* amb la continuació). Sustitueix l'expressió anterior per una expressió *equivalent* que no utilitzi la classe **Continuation**.

16. Implementeu els bucles de tipus

```
repeat
     <body>
until <boolean expression>
```

en Smalltalk: **BlockClosure >> #repeatUntil: aBlock**, utilitzant continuacions. El paràmetre **aBlock** ha de ser un bloc sense paràmetres que, en ser avaluat, retorni un booleà.

17. Executeu aquest codi al *Playground*:

```
| b cont |
b := Continuation callcc: [ :cc | cont := cc. false ].
#(1 2 3 4 5) do: [ :each |
                    (b ifFalse: [ 'NO' ]
                       ifTrue: [ each asString ]) traceCr].
b ifFalse: [ cont value: true ].
```

Digueu què us surt al **Transcript**. Per què surt el que heu vist que surt?

18. Utilitzar dos cops **callcc:** en una expressió fa que el codi resultant sigui força complicat, a part de tenir una certa importancia teòrica (que no podem discutir ni en aquest examen ni a l'assignatura). Veiem-ne un exemple. Primer definim **twicecc:**

```
| twicecc |
twicecc := [ :coll |
             Continuation callcc: [ :f |
                 [ :n | f value: { n . (coll at: 2) } ] value:
                       (Continuation callcc: [ :q |
                            f value: { (coll at: 1) . q } ] ) ] ].

[ :k | [ :arr | (arr at: 2) value: ((arr at: 1) + 1) ]
                  value: (twicecc value: { 0 . k } ) ] value: [ :x | x ]
```

i després el fem servir en una expressió. Si avalueu aquest codi en el *Playground* el resultat és **2**.

19. Definirem dues versions del **`#whileTrueCC:`** 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.
                         ('inside whileTrueCC: ', tmp asString) traceCr.
                         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.
                         ('inside whileTrueCC: ', tmp asString) traceCr.
                         tmp := tmp + 1.
                         cont value: cont]
               ifFalse: [^ nil].
```

Si ara avaluem al *Playground*:

```
| 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 *Playground*, 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.