Introducció a les Continuacions a Smalltalk

A Pharo 3.0 tenim la classe `Continuation`, les instàncies de la qual serveixen per guardar la pila[1] d'execució en un moment donat. Per a això caldra que tingui una variable d'instància, `values`, per guardar aquesta pila:

```
Object subclass: #Continuation
     instanceVariableNames: 'values'
     classVariableNames: ''
     category: 'Kernel-Methods'
```

Les instàncies d'aquesta classe caldrà inicialitzar-les amb un mètode `initializeFromContext: aContext` que el que farà és guardar, a `values`, la pila associada al context que es passa com a argument. Si examineu els mètodes `Continuation class>>#current` i `Continuation class>>#currentDo:`, veureu el paper que hi juga `#initializeFromContext:`. Us ho deixo com a exercici.

```
initializeFromContext: aContext
     | valueStream context |
(1)  valueStream := WriteStream on: (Array new: 20).
(2)  context := aContext.
(3)  [context notNil] whileTrue: [
(4)       valueStream nextPut: context.
(5)       1 to: context class instSize do:
(6)            [:i | valueStream nextPut: (context instVarAt: i)].
(7)       1 to: context size do:
(8)            [:i | valueStream nextPut: (context at: i)].
(9)       context := context sender].
(10) values := valueStream contents
```

Això, però, no és tan senzill, perque el que volem fer és guardar una llista d'instàncies de `MethodContext`, i no qualsevol llista de qualsevol objecte. Caldrà recòrrer la llista d'instàncies de `MethodContext` (començant per la referenciada per l'argument `aContext`) i anar guardant cada context que trobem (línia 4). Això *no* és suficient. També caldrà guardar, per a cada context, les seves variables d'instància i les seves variables indexades[2]. Podem fer-ho en un bucle, per a cada context: guarda el context (línia 4), guarda les variables d'instància (línies 5 i 6), guarda les variables indexades (línies 7 i 8), proper context (línia 9). I per a què volem guardar la pila d'execució? La volem guardar *per poder recuperar-la més tard*, és a dir, reinstaurar aquesta pila d'execució en algun moment de l'execució del programa.

Fixeu-vos que farem servir un *stream* per guardar tot allò que cal, del que després només aprofitarem el contingut (que és una col·lecció) que serà guardat a `values`. Això ho

1 Tota l'estona estem parlant de *pila d'execució*, però és un abús de llenguatge. De fet, ja veieu que aquesta pila està implementada dins d'Smalltalk com una llista enllaçada, i nosaltres voldrem fer operacions que van més enllà de les operacions que caracteritzen les piles.

2 A l'**Annex** teniu una guia per investigar per quina raó hem de guardar aquesta informació afegida.

podeu veure a la línia 1, on s'instancia la classe `WriteStream`, a les línies 4, 6 i 8, on hi afegim contingut amb el mètode `Stream>>#nextPut:` i a la línia 10 on s'extrau el contingut amb `Stream>>#contents` per finalment assignar-ho a `values`.

Hi ha un detall, però, amb el que cal anar amb compte. Nosaltres hem guardat tot el context d'execució en un moment donat. *Aquest context espera rebre un valor de l'expressió que s'ha fet càrrec de guardar el context, senzillament perque tota expressió Smalltalk retorna un valor i per tant el que s'espera de tot enviament de missatge és un valor*. Quan nosaltres reinstaurem el context guardat haurem de passar un valor a aquest context que hem guardat *congelat en el temps* (i que ara estarem *descongelant*), senzillament perque així ho espera.

Per tant, la classe `Continuation` té un mètode anomenat `value: anObject`[3] que, donat un objecte qualsevol,

a) recupera el context que teniem guardat,
b) el converteix en el context actual
c) i li retorna l'objecte `anObject`, que ha rebut com a argument, per poder continuar l'execució del context tot just restaurat.

Per entendre bé quin és el paper que ha de tenir aquest mètode `#value:` fixeu-vos en aquest fragment de codi. Si l'avaluéssim amb `PrintIt` el resultat seria `$a`.

```
| s |
s := Continuation new initializeFromContext: thisContext.
s isCharacter ifFalse: [s value: $a].
s ⟹ $a
```

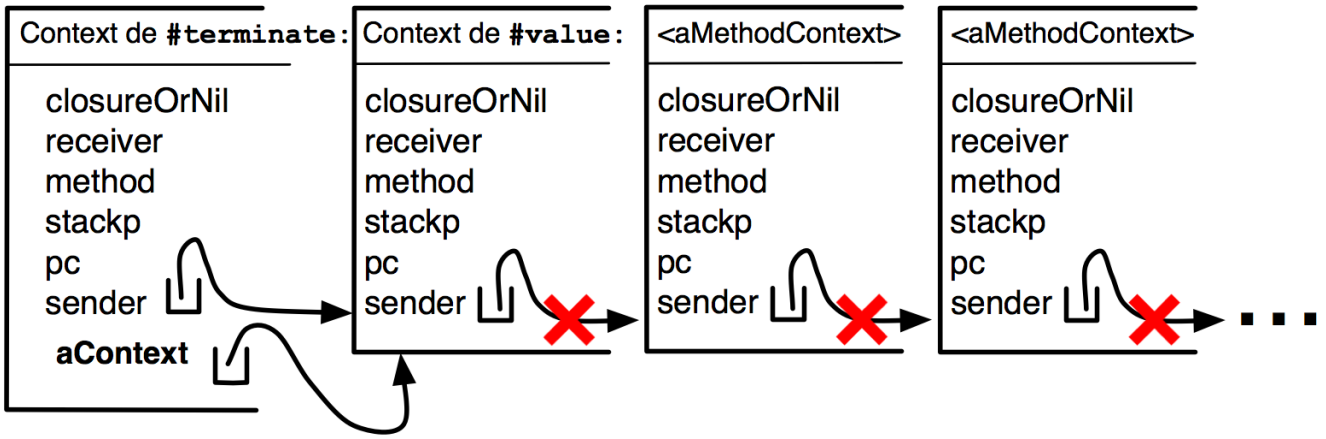
Per què? perque el context que hem guardat en crear la instància de `Continuation` és un context que correspón a assignar un valor a `s` i continuar (aquest és el context al que fa referència `thisContext` en el moment de ser avaluat). Si considerem el lloc on el context espera un valor com un *forat*, podem representar la instància de `Continuation` que conté `s` després de l'assignació com:

```
s := ❑ ← forat
s isCharacter ifFalse: [s value: $a].
s
```

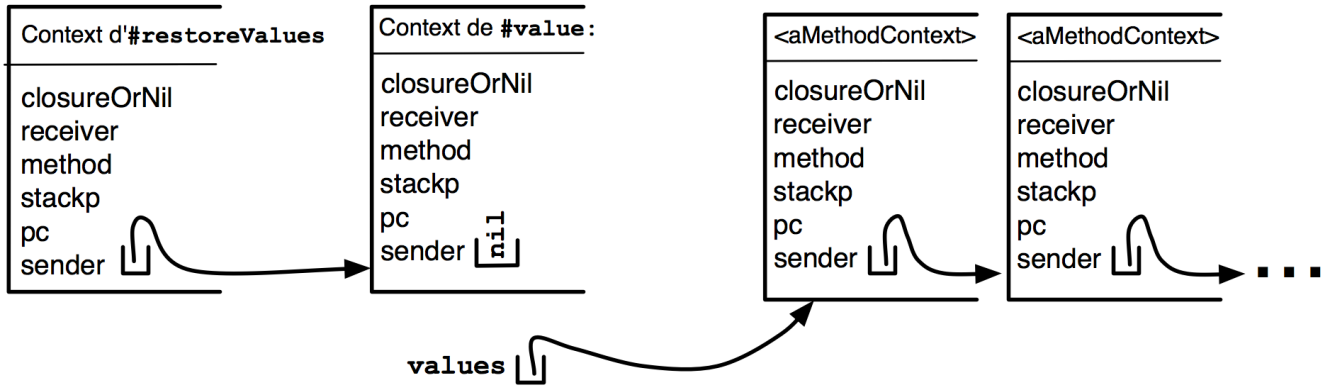
La Implementació de `Continuation>>#value:`, que tot seguit estudiarem, es fa en quatre passos amb l'ajut de dos mètodes auxiliars.

3 S'anomena `value:` i no d'una altra manera perque vull invocar aquest mètode tant en instàncies de `BlockClosure` com en instàncies de `Continuation`, sense fer distincions. És una qüestió de polimorfisme.

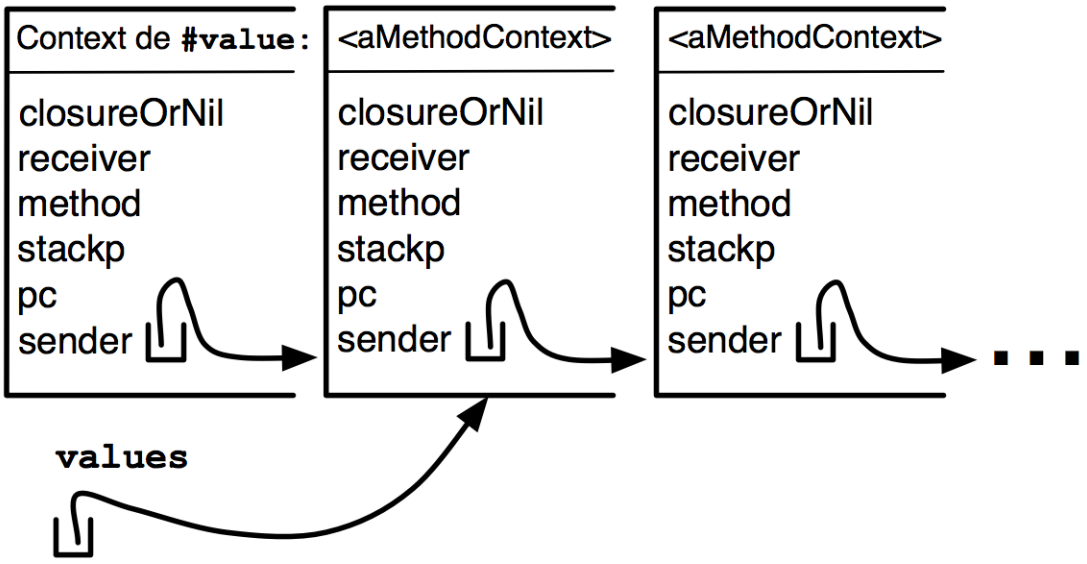
• mètode `Continuation>>#terminate:`. Aquest mètode el que fa és destruir el context d'execució actual. El cridarem amb `self terminate: thisContext` dins de `#value:`. Posarem `nil` a l'atribut `sender` per *alliberar* els contextos (veieu el dibuix). Fixem-nos que hi ha enllaços que *no* hem de trencar. Resulta imprescindible entendre bé el mètode anomenat `ContextPart>>#swapSender:` (on `MethodContext` és subclasse de `ContextPart`)



• mètode `Continuation>>#restoreValues`. Aquest mètode el crida `#value:` després de `#terminate:`. És un mètode que en certa manera fa la feina inversa a la feina que fa `initializeFromContext:`. És a dir, els contextos i la resta d'informació guardada a `values` els reconstruirem ara i aquí. Al final, hauria de quedar com està representat al dibuix següent:



• Finalment unim el context de `#value:`, és a dir el context actual, amb el resultat de la reconstrucció a partir de `values`. Això és tant senzill com fer: `thisContext swapSender: (values first)` i el resultat és:



• Ara ja només queda retornar el valor que han passat com a argument a `#value:`. Penseu-ho bé... on retornarà `anObject` quan faig `^ anObject`?

Així doncs, us acabo de donar la implementació de `Continuation>>#value:`

```
self terminate: thisContext.
self restoreValues.
thisContext swapSender: values first.
^ anObject
```

El mètode `Continuation>>#value` només fa `self value: nil` (serà útil més endavant).

Ara caldria tenir un mètode per poder aplicar d'una manera *controlada* això de guardar el context. No volem que s'utilitzi fent senzillament quelcom com
`Continuation new initializeFromContext: <qualsevol cosa>.`

Aquest mètode l'hauriem de poder cridar des de qualsevol lloc, en qualsevol moment, ja que el que volem és guardar i/o restaurar contextos d'execució, no cal que estigui vinculat a cap objecte en especial. La solució en aquests casos ja sabeu quina és, fem d'aquest mètode un mètode de classe, en aquest cas de la classe `Continuation`. Anomenem-lo `#callcc:`. Aquest mètode tindrà un argument, que serà un bloc amb un paràmetre: `[:k |...]`[4].

4 Tenim un `#callcc:` a la classe `ContinuationTest`, per fer els tests amb les continuacions. Nosaltres, però, el volem tenir disponible a tot arreu. El que fem senzillament és repetir-lo a `Continuation class`

Com funciona `#callcc: aBlock`? La idea és capturar el context actual en una instància de `Continuation` i passar-lo com a paràmetre a `aBlock` en avaluar-lo.
Exemple: El valor d'`x` en avaluar aquesta expressió sería `true`.

```
x := Continuation callcc: [:k | k value: true].
x ⟹ true
```

El que ha passat aquí és que al bloc `[:k | k value: true]` se li ha passat una instància de `Continuation` tal que el context guardat és precisament aquell on el valor que s'espera serà assignat a `x`. Podriem representar un context com un fragment de codi amb un *forat* on hauria d'anar el valor que s'espera. En aquest cas seria quelcom així:

```
x := ❑
x
```

Si heu examinat i entés, com us he demanat al principi del document, `Continuation class>>#currentDo:`, entendreu de seguida la implementació de `Continuation class>>#callcc:`

```
callcc: aBlock
     ^ self currentDo: aBlock
```

Finalment, encara no us heu adonat, però un llenguatge que disposi de `#callcc:` és un llenguatge *molt* expressiu, on puc construir qualsevol estructura de control en termes de `#callcc:`. Com a exemple, la següent pregunta:

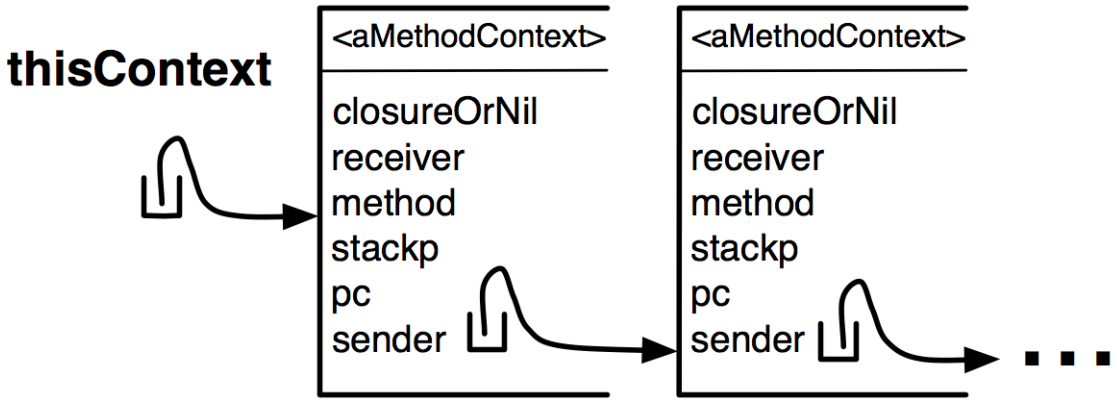
Implementeu `BlockClosure>>#mentreCert:` *(un mètode que faci el mateix que* `#whileTrue:`*) sense utilitzar cap iteració, només utilitzant* `#callcc:`

```
mentreCert: aBlock
     "versió de #whileTrue: implementada amb #callcc:"
     | cont |
     cont := Continuation callcc: [ :k | k ].
     self value
               ifTrue: [ aBlock value.
                         cont value: cont]
               ifFalse: [^ nil].
```

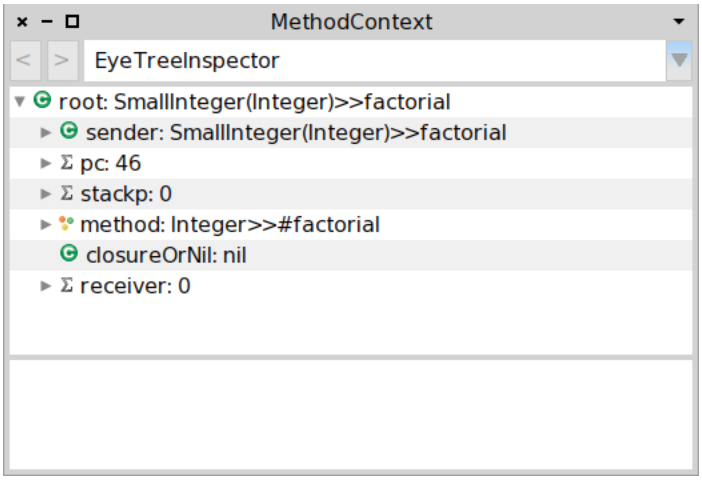
Igual que implementem construccions iteratives, amb `#callcc:` puc implementar excepcions, corutines, multi-threading, backtracking, etc.

Annex:

Sabeu, perque ho vam explicar a classe, que qualsevol moment de l'execució d'un programa Smalltalk té associat una pila d'execució formada per una llista simplement encadenada d'instàncies de la classe `MethodContext`[5]. També vam veure la pseudo-variable d'Smalltalk `thisContext`, que en qualsevol moment de l'execució d'un programa té com a valor una referència a la instància de `MethodContext` que correspon al context d'allò que s'està executant en el moment de demanar pel valor de `thisContext`.

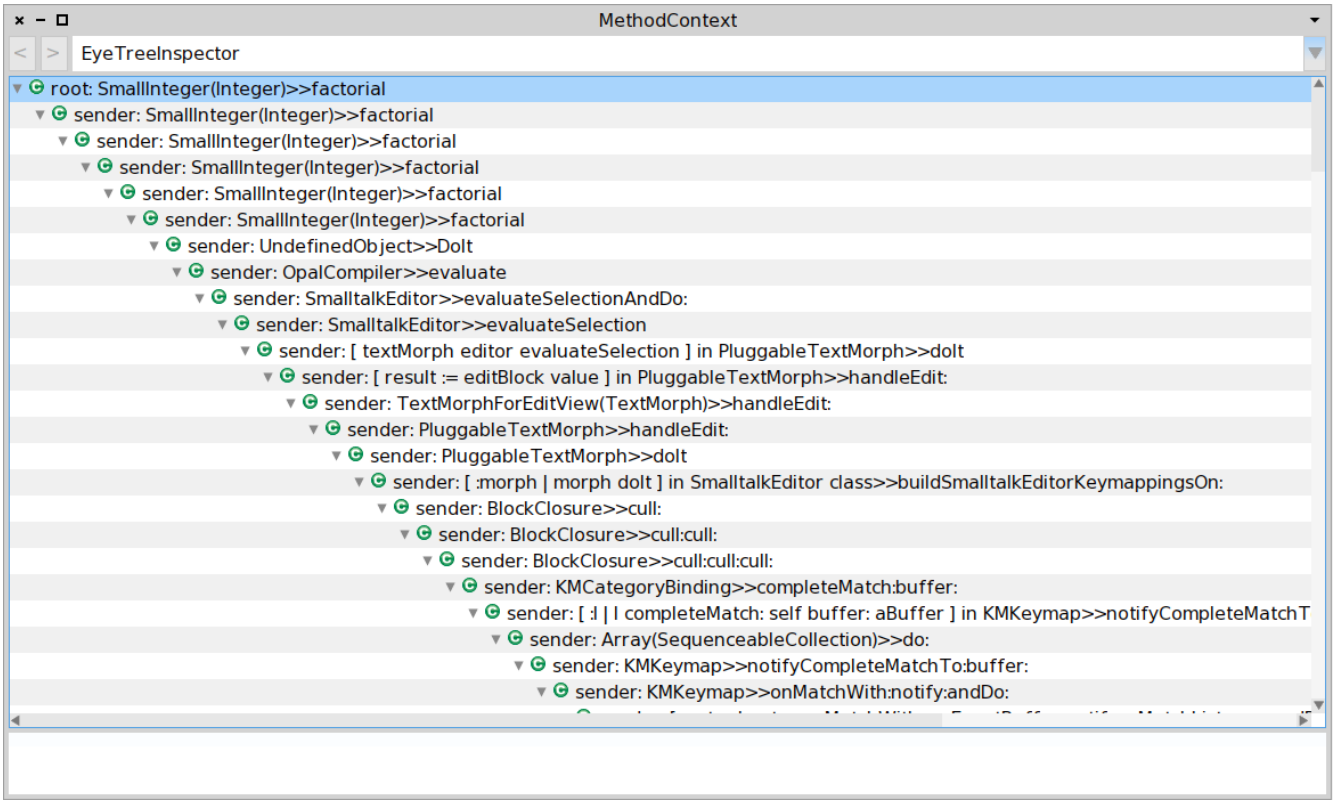


Feu el següent: modifiqueu el mètode `Integer>>#factorial` afegint `thisContext explore. self halt.` al cas base de la recursivitat (just abans del `^1`), i investigueu el que us ensenya l'inspector en avaluar, per exemple, `5 factorial`. S'obriran dues finestres, l'inspector del `thisContext` i la finestra del `Halt`. La finestra del `Halt` la ignorarem, *però no la tanqueu*! La finestra del inspector exposa la instància de `MethodContext` referenciada per `thisContext`. Haurieu de veure les variables d'instància de `MethodContext`: `sender`, `pc`, `stackp`, `method`, `closureOrNil` i `receiver` (vegeu la figura). A més, si hi ha arguments o variables locals, també els veureu com a variables indexades.



5 En altres versions d'Smalltalk, com *Squeak* o *VisualWorks*, la pila pot estar formada per instàncies de `MethodContext` o de `BlockContext`. A Pharo han unificat els dos conceptes per fer més uniforme el tractament de la pila d'execució.

Podeu analitzar la pila d'execució tot seguint l'enllaç de la llista, que és a `sender`. Si el desplegueu del tot podreu veure quelcom de semblant a:



Aquesta és precisament la pila d'execució. Ara, si teniu la pila desplegada torneu-la a plegar (prement la icona triangular al costat del primer `sender`) i tanqueu la finestra del `Halt`. Torneu a l'explorador i proveu de tornar a desplegar la pila d'execució.

Fixeu-vos en `sender` i `pc`. *Què ha passat amb el context d'execució? Per quina raó?* Proveu de repetir l'experiment: avaluació de `5 factorial` i desplegar tota la pila d'execució. Ara tanqueu la finestra del `Halt` i exploreu individualment algunes de les instàncies de `MethodContext` que hi ha a la pila (sel·leccionant-les i triant `inspect` al menú contextual). *Què observeu?*

El que ha passat és que en tancar la finestra del `Halt` hem acabat l'execució del mètode, amb la qual cosa el context associat (penseu que executar aquest mètode `factorial` era tot el que en realitat calia fer) ha desaparegut.