Pràctica CAP Q1 2016-17
Corutines

Conceptes Avançats de Programació

Grau en Enginyeria en Informàtica

Albert Lobo Cusidó

Gener, 2017

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - BarcelonaTech

1 Introducció

Les corutines són un tipus especial de funcions que, a diferència de les funcions tradicionals, permeten aturar i continuar la seva execució mitjançant punts d'entrada/sortida en diferents parts del seu codi. Tal com explica l'encunciat de la pràctica:

> *Quan una corutina C_1 invoca una altra corutina C_2, s'atura i espera que se la torni a invocar. Si això passa, l'execució de C_1 es reprén just en el moment en que va invocar C_2; si ara C_1 torna a invocar C_2, l'execució de C_2 es reprén en el moment que va decidir invocar a un altre corutina.*

Per aquesta pràctica, implementarem una estructura de control en *Smalltalk* anomenada `Coroutine`.

2 Classe `Coroutine`

La classe `Coroutine` proporciona mètodes per crear i cridar corutines. En aquesta implementació, les corutines poden cridar altres corutines i passar-les-hi un paràmetre addicional. La seva definició és:

```
Object subclass: #Coroutine
       instanceVariableNames: 'block cont'
       classVariableNames: ''
       category: 'Coroutines-CAP-2017'
```

2.1 Mètodes de classe

2.1.1 `maker`

La funció `maker` permet crear instàncies de `Coroutine`. Cal passar-li un paràmetre `aBlock` que contindrà el codi de la corutina (el que vulguem). Aquest bloc ha de tenir la següent estructura:

```
[ :resume :value | codiDeLaCorutina ]
```

El paràmetre `resume` és un bloc que proporciona el punt d'entrada/sortida de la corutina, i que permet invocar altres corutines. `value` conté el valor que passem a la corutina el primer cop que la invoquem.

Per cridar altres corutines mitjançant el bloc `resume`, cal que li passem dos paràmetres. El primer paràmetre és una `Coroutine` C, i el segon és el paràmetre que se li passarà a C quan comenci/continui l'execució:

```
resume value: aCorutina value: aValorPassatALaCorutina
```

La funció `maker` simplement crea una nova instància de `Coroutine`, i la inicialitza amb el bloc donat:

```
maker: aBlock
    "create an instance"
    ^ self new initializeOn: aBlock.
```

2.2 Variables d'instància

La classe `Coroutine` té les següents variables d'instància:

`block`: `BlockClosure` que conté el codi de la corutina.

`cont`: `MethodContext` que desarà el context d'execució de la corutina. Inicialment el seu valor és `nil`.

2.3 Mètodes d'instància

2.3.1 `initializeOn`

És necessari cridar aquesta funció per inicialitzar una instància de `Coroutine`. Rep un paràmetre que és el bloc amb el codi de la corutina:

```
initializeOn: aBlock
    "aBlock is the coroutine"
    block := aBlock.
    cont := nil.
```

2.3.2 `reset`

Reinicia la corutina per que la següent vegada que es cridi s'executi `block` des del començament:

```
reset
    "reset the coroutine so next time it starts at the
    begining of the block"
    cont := nil.
```

2.3.3 `value`

La funció `value` arrenca o continua l'execució de la corutina, segons si és la primera vegada que es crida o no.

Quan la funció es crida per primer cop, s'ha d'arrencar l'execució del bloc amb el codi de la corutina. Així, cal que li passem dos paràmetres. El primer és el bloc `resume`, que desa el context d'execució de la corutina en la variable `cont`, i crida la funció `value` de l'altra corutina. El segón és el mateix paràmetre de la funció.

Les seguents vegades que es cridi aquesta funció, l'execució de la corutina ha de continuar allà on s'havia aturat (a causa d'haver executat previament el bloc `resume`). Això s'aconsegueix simplement intercanviant el `sender` de la funció amb `cont`, on haviem desat el context d'execució abans de passar a una altra corutina.

Si la funció es crida després d'acabar la corutina, torna a començar l'execució de `block` sense necessitat de fer `reset`. Per això s'ha de tenir cura d'establir `cont := nil` abans d'intercanviar contextes d'execució. Ho podem veure tot a la següent implementació:

```
value: aValue
    "execute coroutine"
    ^ (cont isNil)
        ifTrue: [ block
            value: [ :coroutine :value |
                cont := thisContext sender.
                coroutine value: value.
            ]
            value: aValue
        ]
        ifFalse: [
            | contTmp |
            contTmp := cont.
            cont := nil.
            thisContext swapSender: contTmp.
            aValue.
        ]
```

3 Classe `CoroutineTest`

Aquesta classe conté els tests que validen el correcte funcionament de la classe `Coroutine`.

3.1 testStatement

Aquest test és el donat a l'enunciat de la pràctica.

3.2 testTheForce

Unim una llista de paraules (*fear*, *anger*, *hate*, i *suffering*) amb una conjunció (*leads to*) per aconseguir la coneguda cita de *Yoda*:

> fear leads to anger
> anger leads to hate
> hate leads to suffering

3.3 testAlphabet

Una corutina genera la següent lletra cada cop que se la invoca, per aconseguir l'abecedari sencer.

3.4 testSumOneToN

Una corutina fa el sumatori de 1 fins el valor especificat com a paràmetre de la corutina.

3.5 testReset

Una corutina crida una altra corutina dues vegades, però entre mig invoca la funció `reset` de l'altra corutina.

4 Classe `StableMarriage`

La classe `StableMarriage` permet solucionar el problema de l'*stable marriage* fent servir corutines.

```
Object subclass: #StableMarriage
    instanceVariableNames: 'n guys gals guyPrefs galPrefs
    solver guyCors galCors guyEng galEng galPrefsOrd trace'
    classVariableNames: ''
    category: 'Coroutines-CAP-2017'
```

4.1 Funcions d'inicialització

Per establir les dades del problema, la classe `StableMarriage` proporciona les següents funcions:

4.1.1 `guys: anArray`

Un array amb els noms dels homes.

4.1.2 `gals: anArray`

Un array amb els noms de les dones.

4.1.3 `guyPrefs: anArray`

Array amb les preferències dels homes. Cada element representa les preferències, en ordre, d'un home. Aquestes preferències es representen amb un array d'enters: l'index de la dóna corresponent.

4.1.4 `galPrefs: anArray`

Array amb les preferències de les dones. Es defineixen igual que les dels homes.

4.2 Exemple d'inicialització

És imprescindible que les dades del problema estiguin ben definides per que StableMarriage funcioni bé (mateix número d'homes que de dones, noms i preferències en el mateix ordre). El següent codi inicialitza una instància d'StableMarriage amb l'exemple de problema del PDF mencionat a l'enunciat:

```
| solver res guys gals guyPrefs galPrefs |
"names"
guys := #( '1' '2' '3' ).
gals := #( 'A' 'B' 'C' ).
"preferences"
guyPrefs := #(
    "1" #(
        1 "A"
        2 "B"
        3 "C"
    )
    "2" #(
        1 "A"
        3 "C"
        2 "B"
    )
    "3" #(
        2 "B"
        3 "C"
        1 "A"
    )
).
galPrefs := #(
    "A" #( 2 "2"  1 "1"  3 "3" )
    "B" #( 2 "2"  3 "3"  1 "1" )
    "C" #( 1 "1"  2 "2"  3 "3" )
).
"init solver"
solver := StableMarriage new.
solver guys: guys.
solver gals: gals.
solver guyPrefs: guyPrefs.
solver galPrefs: galPrefs.
```

4.3 Execució

Un cop entrades les dades, podem obtenir una solució del problema cridant la funció `solve`. Aquesta retorna un array d'enters que representa els emparellaments dels homes —el 1er element és l'índex de la dóna que es casa amb el 1er home; el 2^{on} és l'índex de la dóna que es casa amb el 2^{on} home, etc.

Per imprimir els passos de l'algoritme i el resultat final al `Transcript`, podem cridar la funció `trace:true`. Veiem-ho al següent codi, que completa l'exemple d'inicialització:

```
solver trace: true.
res := solver solve. "#( 3 1 2 )"
```

5 Classe `StableMarriageTest`

Aquesta classe conté els tests que validen el correcte funcionament de la classe `StableMarriageTest`.

5.1 testStatement

Aquest test prova l'exemple del PDF mencionat a l'enunciat de la pràctica.

5.2 testOne

Resol el problema per una sola parella.

5.3 testPeanuts

Troba els *matrimonis estables* del personatges de Peanuts. Podem veure les preferències a la figura 1.

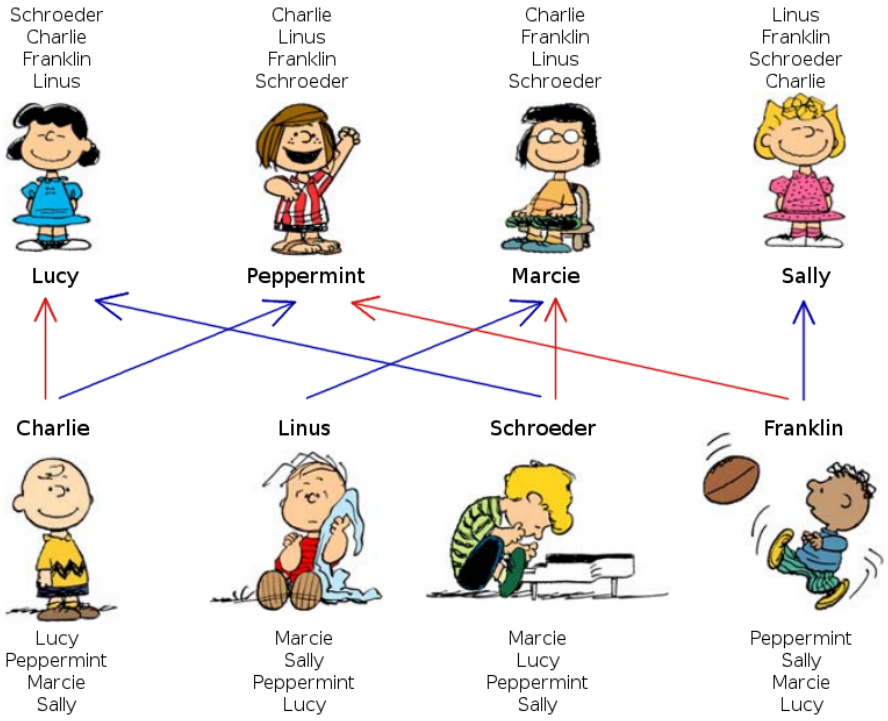

Figura 1: Test de l'*stable marriage* amb els personatges de Peanuts

6 *SmalltalkHub*

Aquest projecte està compartit a *SmalltalkHub.com*. La URL del repositori és:

```
http://www.smalltalkhub.com/#!/~Llop/Coroutines-CAP-2017
```

Per registrar-lo amb el *Monticello*:

```
MCHttpRepository
    location: 'http://www.smalltalkhub.com/mc/Llop/Coroutines-
CAP-2017/main'
    user: ''
    password: ''
```

7 Annex: Corutines amb `swapSender` o `Continuation`

`Coroutine` fa servir la funció `swapSender`, de la classe `ContextPart`, per intercanviar els contextes d'execució de les diferents corutines.

La gràcia és que l'objecte `sender` de `thisContext` conté l'estat dinàmic associat amb l'execució del bloc de la corutina. Quan una corutina fa `swapSender`, s'està fent un salt dins la pila d'execució cap a l'última instrucció que s'havia executat en el context de la corutina.

També és possible implementar corutines amb `Continuation`s. D'aquesta manera, no ens movem sempre per la mateixa pila d'execució, sino que la canviem sencera cada cop que cridem una continuació. En el repositori de *SmalltalkHub.com* podem trobar la classe `CoroutineCont` com exemple del descrit.

La principal diferència pràctica entre les dues estratègies es fa patent quan una corutina que no és la principal acaba la seva execució. El següent codi ho demostra:

```
1  | arr a b |
2  a := b := nil.
3  arr := OrderedCollection new.
4  a := Coroutine maker: [ :resume :value |
5      arr add: 'Hello from A'.
6      arr add: ('Came from ', (resume value: b value: 'A')).
7      arr add: 'Back in A'.
8      arr add: ('Again from ', (resume value: b value: 'A')).
9  ].
10 b := Coroutine maker: [ :resume :value |
11     arr add: '    Hello from B'.
12     arr add: ('    Came from ', (resume value: a value: 'B')).
13     arr add: '    Back in B'.
14 ].
15 a value: nil.
16 Transcript show: (arr joinUsing: Character cr asString); cr.
```

Els tres punts clau del codi són: **1)** línia 6, **2)** línia 8, i **3)** línia 12. Si executem aquest codi fent servir `swapContext`, quan la corutina `b` acaba l'execució, es torna a **2)** perquè des d'allà s'havia fet l'últim `resume` de `b`.

En canvi, si fem servir continuacions amb `CoroutineCont`, el que passa quan `b` acaba és completament diferent, i potser no tan intuïtiu. L'execució no continua a **2)**, sino al punt **1)** —el lloc des on s'havia cridat `b` immediatament abans de desar la continuació (en **3)**).

En la següent taula podem veure les diferències entre les sortides:

`Coroutine`

```
Hello from A
    Hello from B
Came from B
Back in A
    Came from A
    Back in B
Again from      Back in B
```

`CoroutineCont`

```
Hello from A
    Hello from B
Came from B
Back in A
    Came from A
    Back in B
Came from      Back in B
Back in A
    Hello from B
Again from B
```