Pràctica CAP Q1 2019-2020

Coroutines

Conceptes Avançats de Programació (CAP)

Ismael de la Gràcia

Júlia Gasull

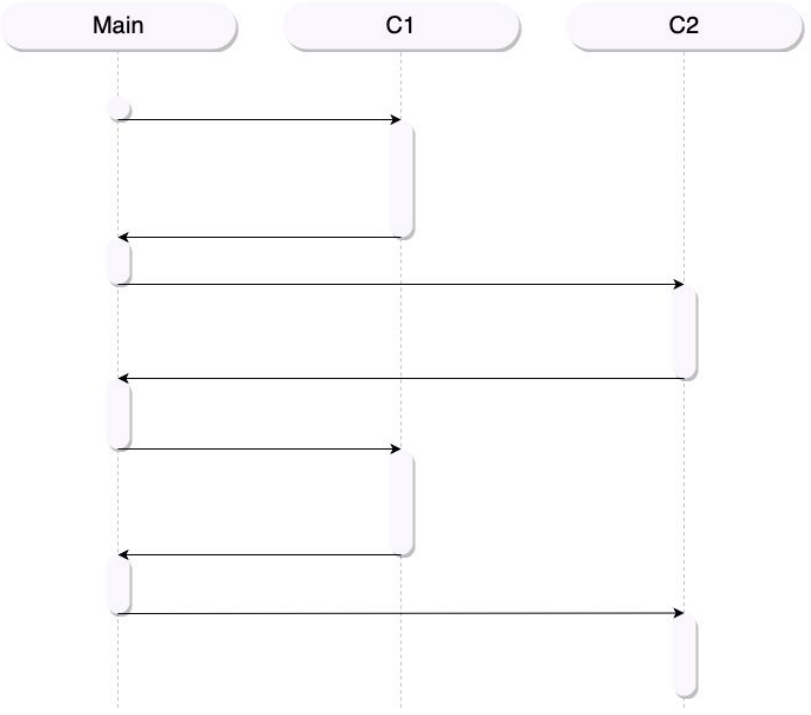
Índex

Introducció

Les coroutines són un tipus especial d'objecte que permeten suspendre i reprendre l'execució d'elles mateixes sempre i quan siguin cridades. Són adequades per implemen- tar coses com ara tasques, excepcions, bucles d'esdeveniments, iteradors i llistes infinites.

Com diu l'enunciat:

> *La idea de l'estructura de control anomenada corutina és la següent: imaginem una funció (o procediment, o subrutina) tal i com ja les coneixem de C o C++. Les funcions s'invoquen, executen el seu cos, i quan acaben retornen. Si les tornem a invocar, torna a executar-se tot el cos de la funció, i quan acaba retorna.*
>
> *Les corutines funcionen diferent: Quan una corutina C1 invoca una altra corutina C2, s'atura i espera que se la torni a invocar. Si això passa, l'execució de C1 es reprén just en el moment en que va invocar C2; si ara C1 torna a invocar C2, l'execució de C2 es reprén en el moment que va decidir invocar a un altre corutina.*



Hem creat una classe en `Smalltalk` anomenada `Coroutine` per poder-la implementar.

Classe `Coroutine`

La classe `Coroutine` té els mètodes necessaris com per poder fer una nova corutina i cridar-ne d'altres. Una corutina té la capacitat de cridar a una altra corutina, passant-li com a paràmetre el bloc de codi que aquesta tindrà.

```
Object      subclass: #Coroutine
            instanceVariableNames: 'dream kick'
            classVariableNames: ''
            package: 'Practica'
```

Mètodes de classe

`maker:`

`maker` és una funció que crea instàncies de `Coroutine`. Caldrà passar un bloc com a paràmetre, que és on posarem el codi de la corutina. Aquest bloc serà un bloc de dos paràmetres:

<instancia corutina> ← `Coroutine maker: [:resume :value | codiCorutina]`

El paràmetre `:resume` serveix per invocar altres corutines, i ho fa de la següent manera:

```
resume value: <nomCorutina> value: <valorPassatACorutinaInvocada>
```

`resume` serà, doncs, un bloc amb dos paràmetres:

1. referència a una altra corutina
2. valor que se li passarà a aquesta corutina com a valor de retorn de la crida a corutina que va fer el darrer cop que es va executar.

`maker` tan sols crearà una nova instància de Coroutine i li donarà per valor el bloc passat com a paràmare:

```
maker: aBlock
      ^ Coroutine new initializeWith: aBlock.
```

Variables d'instància

Les varables d'instància de `Coroutine` són les següents:

- `dream`
  - l'utilitzarem com a tipus `BlockClosute`
  - conté el codi de la corutina
- `kick`
  - l'utilitzarem com a tipus `MethodContext`
  - guarda el context d'execució en de la corutina
  - inicialment és `nil` donat que encara no s'ha executat

Es pot veure que els noms són una mica extranys. Ens hem inspirat en la pel·lícula *Inception*. `dream` és el somni en el que s'està en un moment en concret, és a dir, el codi que s'ha d'executar. `kick` és la "patada" que et permet sortir d'aquell somni.

Mètodes d'instància

`initializeWith:`

Aquesta funció es crida quan es crea una instància de `Coroutine`. Inicialitza les seves variables d'instància. `dream` amb el bloc que li passem com a paràmetre i `kick` a `nil`, donat que el bloc encara no s'ha executat.

```
initializeWith: aBlock
	dream := aBlock.
	kick := nil.
```

`reset`

`reset` permet que, si una corutina s'acaba d'executar, la següent vegada que sigui cridada, s'executi des de l'inici.

```
reset
	kick := nil.
```

`value:`

`value:` és la funció més important de tota la classe. Com hem explicat abans, és la funció que fa començar o continuar una corutina. Començarà si no ha estat cridada abans (o la seva execució havia acabat) i continuarà si estava a mitja execució.

En el primer cas, quan es crida per primer cop, es començarà a executar directament el codi de la corutina. Els paràmetres que li passem seran, com sempre, dos: `resume` (bloc), i guardarem el context d'execució en la variable d'instància `kick` i cridarem la funció `value` de l'altra corutina; i el segon paràmetre serà el que li passem a aquesta última funció.

En els següents casos, la corutina ja estarà a mitja execució, i l'haurem de fer continuar allà on estava en "stand by". El que em aquí és guardar-nos el context d'execució en el que estava, posar l'actual a `nil`. Llavors, es torna a començar l'execució de `dream`.

Aquí tenim la implementació:

```
value: aValue
      ^ (kick isNil)
            ifTrue:
            [
                  dream
                  value:
                  [
                        :coroutine :value |
                              Continuation callcc:
                              [
                                    :executionFlow |
                                          kick := executionFlow.
                                          coroutine value: value.
                              ]
                  ]
                  value: aValue.
            ]
            ifFalse:
            [
                  | kickTmp |
                  kickTmp := kick.
                  kick := nil.
                  kickTmp value: aValue.
            ]
```

Classe `CoroutineTest`

La classe `CoroutineTest` permet validar que la classe `Coroutine` funcioni correctament.

```
TestCase  subclass: #CoroutineTest
          instanceVariableNames:
          'sumOneToNTestResult1 sumOneToNTestResult2
          sumOneToNTestResult3 enunciatTestResult
          philosophyTestResult primeTestResult
          greaseTestResult'
          classVariableNames: ''
          package: 'Practica'
```

testEnunciat

És el test que hi ha a l'enunciat de la pràctica.

testPhilosophy

Aquest test és una mica filosòfic. Donat un conjunt de paraules: `#('life' 'death' 'dying' 'all of us')`, arribem a la següent frase:

life** leads to **death
death** leads to **dying
dying** leads to **all of us

testSum

Aquest test fa el sumatori de tots els naturals fins al número passat a la corrutina sum. Sum simula un comportament recursiu per realitzar aquesta tasca, amb un cas base i una crida recursiva.

En concret, aquest test busca el resultat de sumar tots els naturals fins el 200, el 5 i el 0.

El resultat d'aquest test és:

```
20100, 15, 0
```

Cal destacar que al principi intentavem utilitzar el paràmetre value per emmagatzemar els números que queden per sumar, però això donava el error "Cannot store into ->value".

El motiu d'aquest error pensem que es que, a diferencia de la recursivitat habitual, aquesta utilitza corutines. Això implica que no es fa una nova crida a sum, si no que es resumeix l'anterior, i gràcies al whileTrue que simula el cas recursiu aquest "resum" funciona.

Per tant, el que a primera vista pot semblar una solució recursiva és en realitat una iterativa. Això implica, doncs, que la variable value no es pot modificar entre resumes, doncs es un paràmetre comú a totes les crides o a tots els resums.

No són mètodes diferents invocats recursivament, és el mateix mètode amb les mateixes variables assignades a les mateixes posicions de memòria.

testRecursive

Aquest test es va intentar fer per tal de simular d'una forma més propera la recursivitat abans mencionada, però com no vam aconseguir fer que funcionés, no l'hem inclós al fitxer de la pràctica. De totes formes, hem decidit comentar-lo perqu pensem que és interessant i que no estem massa lluny de fer-lo funcionar.

La idea del test es utilitzar les corrutines per fer una funció recursiva. Per això, i aprenent del test anterior que no es pot fer un *resume* d'una corrutina com si es tractés d'un mètode (esperant que fos una nova invocació independent), vam pensar en fer una copia de la mateixa i utilitzar el mètode *reset* sobre aquesta copia, i fer un *resume* d'aquesta.

Per fer desar aquesta copia vam utilitzar una variable local de la propia corrutina i el mètode deepCopy, ja present a Pharo.

El codi, és el següent:

```
testRecursive
  "Factorial n"
  "IMPORTAT NO EXECUTAR AQUEST CODI,
   LLEGIR EXPLICACIÓ A BAIX"
  | main recursive ret |

  ret := OrderedCollection new.

  recursive := Coroutine maker:
  [
    :resume :value |
      | aux |
      (value <= 1)
        ifTrue: [ value. ]
        ifFalse:
        [
          aux := recursive deepCopy.
          aux reset.
          value * (resume value: aux value: (value-1)).
        ].
  ].
```

```
main := Coroutine maker:
  [
    :resume :value |
      ret
        add: 'Starting recursive strategy...';
        add: 'N is :';
        add: value;
        add: 'n! is: '
      .
      ret add: (resume value: recursive value: value)
  ].

  main value: 4.
  ret := ret joinUsing: Character cr asString.
  Transcript show: ret.
```

Quan executem aquest codi (i després d'esperar una estona), la màquina virtual fa un *crash*. Fent proves vam trobar que és el missatge *deepCopy* el que dona aquest problema. En una d'aquestes proves, Pharo ens va donar un error que deia que no es podia fer una *deepCopy* d'un objecte de context del sistema. Aquestes proves també es van fer amb un missatge semblant, anomenat *veryDeepCopy*.

testPrimes

Aquest test prova el correcte funcionament de les corrutines i la seva versatilitat. Donat un número natural qualsevol, busca tots els nombres primers que hi ha fins aquest número i els retorna.

En concret, aquest test busca els números enters fins al 25.

Per fer això utilitza l'algorisme per trobar si un numero es primer i dues corrutines. Una fa la funció de programa principal i l'altre de generador. La segona es la que genera tots els números primers amb ajuda de l'algoritme.

El resultat és el següent:

```
2, 3, 5, 7, 11, 13, 17, 19, 23
```

testGrease

Aquest test demostra el correcte funcionament de el mètode *reset*. Consta de dues corrutines, una que representa a Danny, el cantant principal de la canço Grease Lighting, i l'altre representa el coro.

Quan en Danny acaba la seva part, el coro canta la seva. Com el coro canta la mateixa frase tota l'estona, el reset s'utilitza per a que torni a cantarla.

Per demostrar que la corrutina pot seguir el seu funcionament correcte després del reset, s'ha afegit una altra crida que no pertany a la cançó.

El resultat del test es el següent:

```
- Go, grease lightnin', you're burnin' up the quarter mile
    - Grease lightnin', go grease lightnin'
- Go, grease lightnin', you're coastin' through the heat
lap trials
    - Grease lightnin', go grease lightnin'
.
.
.
Can we go home now?
```

Classe `StableMarriage`

Aquesta classe es base en l'article adjuntat amb la pràctica, que parla del problema dels "matrimonis estables".

El problema és el següent:

> *Donat un conjunt de n homes, un conjunt de n dones i una llista de preferències de parella matrimonial per a cada home i cada dona, trobi un matrimoni estable per a tot el grup. El matrimoni és inestable si hi ha un home i una dona que no estan casats entre si, però que tots dos es prefereixen a les seves parelles reals.*
>
> *El problema es va donar com una variació alegre de el problema d'admissió a la universitat. Gale i Shapley van demostrar que sempre hi ha a l'almenys una i possiblement moltes solucions i van donar un algoritme per resoldre el problema.*

I l'algoritme que van aconseguir per resoldre'l era:

> *Cada home proposa les seves preferències en ordre fins que sigui acceptat. Si després és rebutjat, continua amb la seva pròxima elecció. Cada dona accepta al seu pretendent més preferit; ella accepta la primera proposta però després pot rebutjar al seu pretendent si obté una millor oferta.*
>
> *La seqüència d'esdeveniments comença amb el primer home proposant la seva primera opció i sent acceptat. Cada home subsegüent li proposa una dona no compromesa que l'accepta o una dona ja compromesa.*
>
> *En l'últim cas, ella pot rebutjar-lo, si ho prefereix al seu marit actual, canviar de lleialtat. En qualsevol cas, l'home rebutjat proposa la seva pròxima elecció.*

La classe que hem fet per implementar-lo té la següent estructura:

```
Object      subclass: #StableMarriage
            instanceVariableNames:
                  'solver trace maxn men women menRanking
                  womenRanking menChoice womenChoice
                  menEngagement womenEngagement
                  womenRankingOrder'
            classVariableNames: ''
            package: 'Practica'
```

Variables d'instància

`solver`

És la variable sobre la qual s'acaba treballant. Si es fa solver solve, es retorna un array d'enters que representa els emparellaments dels homes. Per exemple, si l'array és `#(2 1 3 4)`, significa que l'home 1 s'ha emparellat amb la dona 2, l'home 2 amb la dona 1, l'home 3 amb la dona 3 i l'home 4 amb la dona 4.

`trace`

És un booleà que serveix per imprimir al `Transcript` si és necessari.

`maxn`

Nombre màxim d'homes i dones, sempre el mateix.

`men`

Un array amb els noms dels homes.

`women`

Un array amb els noms de les dones.

`menRanking`

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 dona corresponent.

`womenRanking`

Array amb les preferències de les dones. Implementat igual que el dels homes.

others

Les variables:

- `menChoice`
- `womenChoice`
- `menEngagement`
- `womenEngagement`
- `womenRankingOrder`

són variables que anirem necessitant a mida que es busqui una solució. Per exemple, `womenRankingOrder` l'utilitzem per mirar quin ordre prefereixen les dones en cas que hagin de rebutjar alguna petició (i no és el mateix que `womenRanking`, donat que és diferent per cada petició).

Preparació i execució

Per tal d'inicialitzar un problema de StableMarriage, necesstem els següents paràmatres: `men`, `women`, `menRanking` i `womanRanking`.

Aquí deixem un exemple d'un dels tests que explicarem més endavant (en cas que al lector li agradi la sèrie, recomanem que es miri el codi, donat que hi ha comentaris referents als personatges):

```
testFriends
	"stable marriage problem with the Friends characters"
	| solver res boys girls boysRanking girlsRanking |

	"names"
	boys :=    #( 'Joey' 'Chandler' 'Ross' 'Mike' ).
	girls :=   #( 'Rachel' 'Monica' 'Phoebe' 'Julie' ).

	"preferences"
	boysRanking :=
	#(
		"Joey"
		#( 1 "Rachel" 3 "Phoebe" 2 "Monica" 4 "Julie" )
		"Chandler"
		#( 2 "Monica" 1 "Rachel" 4 "Julie" 3 "Phoebe" )
		"Ross"
		#( 1 "Rachel" 4 "Julie" 3 "Phoebe" 2 "Monica" )
		"Mike"
		#( 3 "Phoebe" 1 "Rachel" 2 "Monica" 4 "Julie" )
	).
	girlsRanking :=
	#(
		"Rachel"
		#( 3 "Ross" 1 "Joey" 4 "Mike" 2 "Chandler" )
		"Monica"
		#( 2 "Chandler" 1 "Joey" 4 "Mike" 3 "Ross" )
		"Phoebe"
		#( 1 "Joey" 4 "Mike" 3 "Ross" 2 "Chandler" )
		"Julie"
		#( 3 "Ross" 4 "Mike" 1 "Joey" 2 "Chandler" )
	).

	"init solver"
	solver := StableMarriage new.
	solver men: boys.
	solver women: girls.
	solver menRanking: boysRanking .
	solver womenRanking: girlsRanking .
	solver trace: true.
```

```
res := solver solve.
self assert: friendsTestResult = res.
```

Com podem veure, al fer els passos de la secció “init solver”, s’executa l’algoritme explicat anteriorment i ens dona el resultat esperat.

Classe `StableMarriageTest`

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

La seva estructura és la següent:

```
TestCase  subclass: #StableMarriageTest
          instanceVariableNames:
               'statementTestResult friendsTestResult
               spidermanTestResult'
          classVariableNames: ''
          package: 'Practica'
```

testArticle

És el test que hi ha a l'article adjunt amb la pràctica.

testSpiderman

Aquest test serveix per comprovar que l'algorisme funciona correctament si només hi ha una parella possible.

Els personatges son:

- Homes: Peter Parker
- Dones: Mary Jane Watson

I les preferences son:

Peter Parker	Mary Jane Watson

Mary Jane Watson	Peter Parker

El resultat és el següent matrimoni estable:

Peter Parker	Mary Jane Watson

testFriends

Aquest test toba els matimonis estables dels personatges de Friends. N'hem explicat el codi en l'exemple de preparació i execució de la classe `StableMarriage`.

Donats els personatges:

- Homes: Joey, Chandler, Ross i Mike
- Dones: Rachel, Monica, Phoebe i Julie

I les preferències:

Joey	Rachel, Phoebe, Monica, Julie
Chandler	Monica, Rachel, Julie, Phoebe
Ross	Rachel, Julie, Phoebe, Monica
Mike	Phoebe, Rachel, Monica, Julie

Rachel	Ross, Joey, Mike, Chandler
Monica	Chandler, Joey, Mike, Ross
Phoebe	Joey, Mike, Ross, Chandler
Julie	Ross, Mike, Joey, Chandler

Acabem amb els següents matrimonis estables:

Joey	Phoebe
Chandler	Monica
Ross	Rachel
Mike	Julie

[imatge explicatòria en la següent pàgina]

L’esquema queda de la següent manera: