

FACULTAT D'INFORMÀTICA (UPC)



CONCEPTES AVANÇATS DE PROGRAMACIÓ

UPC-FIB

---

# Pràctica Smalltalk i Javascript

---

Marc MÉNDEZ ROCA  
Jordi ESTAPÉ CANAL

# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introducció</b>                                      | <b>2</b> |
| <b>2</b> | <b>Implementació continuation de Javascript a Pharo</b> | <b>3</b> |
| 2.1      | Implementació . . . . .                                 | 3        |
| 2.2      | Exemples i resultats . . . . .                          | 3        |
| <b>3</b> | <b>JavaScript implementació de Calcc</b>                | <b>5</b> |
| 3.1      | Implementació . . . . .                                 | 5        |
| 3.2      | Exemples i resultats . . . . .                          | 5        |
| <b>4</b> | <b>Javascript cooperative multi-threading</b>           | <b>8</b> |
| 4.1      | Introducció a cooperative multi-threading . . . . .     | 8        |
| 4.2      | Consideracions de disseny . . . . .                     | 10       |
| 4.3      | Definició de les funcionalitats demanades . . . . .     | 10       |
| 4.4      | Disseny final . . . . .                                 | 11       |
| 4.5      | Implementació . . . . .                                 | 11       |
| 4.5.1    | Continuations . . . . .                                 | 11       |
| 4.5.2    | Thread . . . . .  | 12       |
| 4.5.3    | CooperativePool . . . . .                               | 12       |
| 4.5.4    | Funcio make thread system . . . . .                     | 16       |
| 4.6      | Test . . . . .  | 16       |

# 1 Introducció

Aquest és un treball en el qual es tractarà el temari après a l'assignatura 'Conceptes avançats de programació'. Tractarem principalment el tema de les Continuacions en Javascript i Smalltalk. Per tal de fer-ho se'ns proposen dos exercicis diferents.

Un primer exercici basat en implementar estructures del tipus Continuations tant en Smalltalk com en Javascript que simulin el comportament de funcionalitats del llenguatge oposat. La proposta són aquestes dues implementacions:

- Implementar en Pharo un mètode unari continuation tal que funcioni exactament igual que el Continuation() de Javascript.
- Implementar en Javascript una funció calcc(f) que funcioni com l'estructura calcc explicada a Pharo.

El segon exercici proposat és la implementació en Javascript mitjançant prototipus i continuacions d'un sistema de multi-threading cooperatiu. La proposta d'exercici porta inclòs dos exemples per a la comprovació de funcionament (testing).

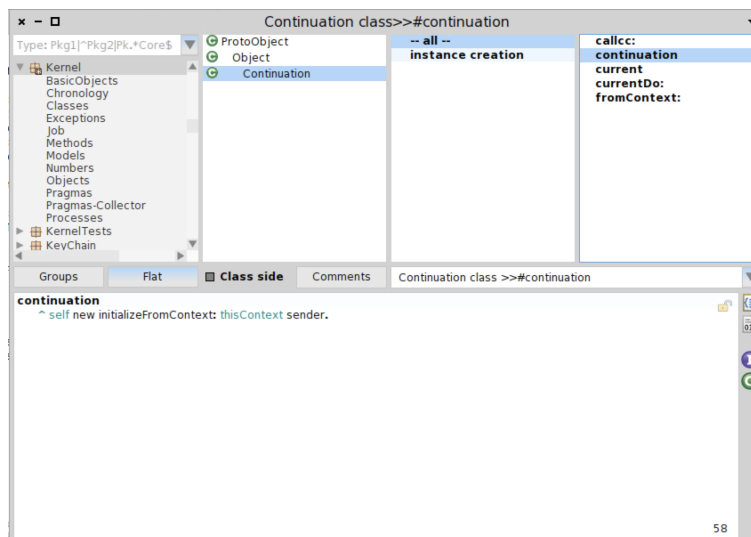
## 2 Implementació continuation de Javascript a Pharo

### 2.1 Implementació

En el primer apartat havíem d'implementar el mètode Continuation() de Javascript en SmallTalk. Per fer-ho hem de saber, tant el funcionament de Continuation() en Javascript, com els mètodes de la classe Continuation de SmallTalk. Analitzant els mètodes observem que initializeFromContext el que fa és guardar valors a la pila associats al context que se li passa per argument. Així doncs podríem dir que ens va perfecte ja que la continuació de Javascript es basa en això.

### 2.2 Exemples i resultats

El mètode seria el següent:



En el cas de no haver de crear un nou mètode que complís unes característiques concretes podríem utilitzar el ja existent "Continuation class >> current". El seu funcionament seria en l'àmbit pràctic el mateix.

S'ha comprovat mitjançant un exemple molt semblant als que es mencionaven en les continuacions de Javascript.

En l'exemple superior es crea una funció test i amb una simple comprovació sobre el tipus de classe determinem quan és una Continuació i quan deixa de ser-ho.

Els resultats que surten són els següents, i també els esperats segons els exemples de les transparències.

```
contPoint value: a Continuation
x is a continuation
contPoint value: 1
x is not a continuation, is a SmallInteger
```

Per tant podríem dir que imita el funcionament de Continuation() de Javascript

## 3 JavaScript implementació de Calcc

### 3.1 Implementació

En aquest apartat es demanava una implementació del mètode calcc existent en Smalltalk. El primer pas era analitzar el funcionament de calcc, ja que s'havia de trobar la forma d'imitar-lo, aquest pas ja s'havia realitzat a classe i per tant només va ser fer memòria del que ja havíem après. A continuació fer el mateix pel mètode de Continuation de JavaScript, que també el coneixíem de classe.

```
Continuation.callcc = function(f) {  
    return f(new Continuation());  
}
```

Un cop ja sabíem el funcionament de les dues funcions vàrem implementar un mètode de Continuation anomenat calcc, que és el mostrat just a sobre. Aleshores en cridar a la funció que hem anomenat com a "f", es torna al "punt de retorn", és a dir el lloc on s'ha creat la continuation.

### 3.2 Exemples i resultats

A continuació mostrarem un parell d'exemples per ensenyar el seu funcionament i quin ús se li podria donar.

```
Continuation.callcc = function(f) {  
    return f(new Continuation());  
}  
  
aux = 0;  
  
var cont = Continuation.callcc(function (cc) {  
    return cc;  
});  
  
if (aux < 10) {  
    print("aux = " + aux);  
    ++aux;  
    cont(cont);  
} else {  
    print ("Counter finished!");  
}
```

En el primer exemple podriem veure un contador que va de 0 a 9, però en lloc d'un while o un for utilitzem calcc. Al passar-li com a paràmetre de la continuació la mateixa, el que estem conseguint és fer un bucle. Que vindrà determinat pel valor de "aux".

La sortida d'aquest codi seria la següent:

```
aux = 0
aux = 1
aux = 2
aux = 3
aux = 4
aux = 5
aux = 6
aux = 7
aux = 8
aux = 9
Counter finished!
```

Com podem comprovar la nostra funció es manté en el bucle fins que "aux >= 10". Per tant podriem dir que seria el comportament esperat, però una prova no és suficient per determinar si funciona correctament o no.

Com a segon exemple tindriem un codi simple per comprovar els llocs on passa i pels que no.

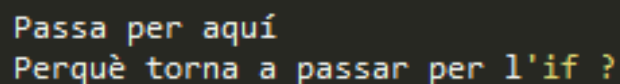
```
Continuation.callcc = function(f) {
  return f(new Continuation());
}

aux = 0;

var x = Continuation.callcc(function (cc) {
  cont = cc;
  return cc(true);
});

if (x) {
  print("Passa per aquí");
  cont(false);
  print("No passa per aquí");
} else {
  print ("Perquè torna a passar per l'if ?");
}
```

Com es veu en el codi superior tenim un if amb 3 prints per comprovar en quines parts passa i en quines no. Com ens podem imagina pel "Passa per aquí" arriba, en canvi pel "No passa per aquí" no. Això és degut a la crida "cont(false)" que retorna l'execució abans d'entrar a l'if. Això provoca que el valor de "x" sigui fals i per tant vagi a l'else i escrigui la frase corresponent.



```
Passa per aquí
Perquè torna a passar per l'if ?
```

Aquest seria el resultat que mostraria el nostre codi i que correspon amb el que havíem pensat, sabent com funciona calcc.



## 4 Javascript cooperative multi-threading

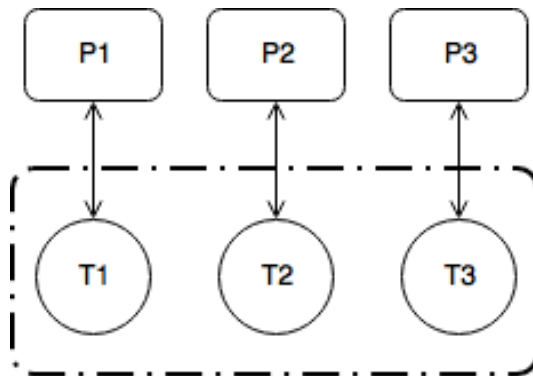
Per acabar, en aquesta pràctica se'ns proposa trobar una manera de portar a terme el concepte de 'multi-threading' cooperatiu en 'Javascript' mitjançant els coneixements adquirits durant el transcurs del curs.

### 4.1 Introducció a cooperative multi-threading

Per començar a fer-ho el primer pas que hem hagut de seguir és entendre què es 'multi-threading' cooperatiu per tal de comprendre quin és el resultat que s'espera del codi a implementar. Per tal de fer-ho s'ha consultat tant pàgines web com els exemples donats per a la comprovació del projecte.

Per tal de clarificar i per posteriorment facilitar-ne la implementació i el disseny, hem realitzat la següent descripció i esquemes del que representa el 'multi-threading cooperatiu'.

El 'multi-threading cooperatiu' es basa en què: donada una 'pool' de 'threads' ('T') que han de realitzar uns processos ('P'), aquests 'threads' ('T') s'auto-gestionaran el temps d'execució passant-se entre ells el relleu de l'execució quan escaigui. L'ordre dels 'threads' a executar serà definit per l'ordre d'entrada del vincle entre els 'threads' i els processos ('T' i 'P'). Per a fer-ho més comprensible hem graficat el procés:



Suposem doncs que tenim una 'pool' de 3 'threads' (T1, T2, T3) que tenen vinculats respectivament els processos (P1, P2, P3) i que han entrat al sistema ordre numèric ascendent. És a dir que han entrat:

- $T1 \implies P1$
- $T2 \implies P2$
- $T3 \implies P3$

Suposem doncs ara que, P1 està dividit en dues parts, i que entre les dues parts el procés P1 para l'execució de T1, l'introdueix de nou a la cua i dona relleu al següent 'thread' de la cua per executar. Quedant ara la cua així:

- $T2 \implies P2$
- $T3 \implies P3$
- $T1 \implies P1$

En aquest punt, T2 restaura el seu punt d'execució i executarà P2 normalment fins que de nou pugui donar relleu al següent element de la llista. Passant doncs ara a:

- $T3 \implies P3$
- $T1 \implies P1$
- $T2 \implies P2$

Veiem doncs que el que realment hem de reproduir és una cua de 'threads' que executin els seus processos quan tinguin el poder d'execució de la 'pool' i que siguin capaços de recuperar el seu punt d'execució per tal de poder restablir-lo. Aquesta gestió de poder d'execució serà donada pels mateixos threads.

## 4.2 Consideracions de disseny

Un cop estudiat el que representa el 'multi-threading cooperatiu' hem estat capaços de realitzar un conjunt de consideracions de disseny del codi que esperem permetin resoldre més apropiadament el problema.

La primera consideració que hem pogut determinar és que per a poder mantenir el context d'execució dels 'threads' haurem d'usar mètodes relacionats amb Continuacions.

La segona consideració que hem pogut extreure és que existeixen entitats clarament identificables en el problema proposat. Aquestes són:

- **Threads:** entitat mínima de feina que va vinculada a un procés. Poden estar en execució o en espera de rebre el permís d'execució i han de mantenir el context d'execució.
- **Pool:** conjunt de 'threads'. En aquest cas tan sols un 'thread' d'una mateixa 'pool' pot estar executant-se a la vegada.
- **Processos:** conjunt de línies de codi que ha de realitzar un 'thread' per tal de complir la seva execució.

Hem identificat també l'existència d'un ordre basat en un tipus d'estructura cua. Aquest mètode haurà de ser implementat en el nostre codi per tal de gestionar l'ordre d'execució.

## 4.3 Definició de les funcionalitats demanades

Una altra ajuda que se'ns dona és la definició de les funcionalitats a implementar per a la gestió d'un sistema cooperatiu. Definim doncs les següents:

- **Spawn:** Aquesta funcionalitat hauria de permetre afegir un thread a la pool.
- **Start threads:** Aquesta funcionalitat ha de donar pas a iniciar l'execució dels diferents processos acumulats a la pool.
- **Relinquish:** Aquesta funcionalitat ha de permetre frenar l'execució d'un thread per a reafegir-lo a la cua i donar pas al següent thread de la cua.
- **Quit:** Aquesta funcionalitat ha de permetre tancar l'execució d'un thread i fer-lo fora de la llista.

## 4.4 Disseny final

Finalment i donades totes les consideracions explicades prèviament s'ha decidit implementar un sistema que consti de les entitats que a continuació seran explicades.

- **Thread:** Per a nosaltres un 'thread' contindrà un bloc de codi a executar ("aBlock"), un boolea que indiqui si ha d'executar-se o no ("canExecute") i una continuació que ens permeti mantenir el estat d'execució per a recuperar-lo (state). És la unitat mínima d'execució.
- **CooperativePool:** Entenem una CooperativePool com un conjunt de n 'threads' que mantenen el seu ordre d'execució, per tant la idea està a implementar una 'queue' que ens permeti gestionar-ho. ("queue"). Aquest prototip contindrà les funcionalitats de gestió que s'han comentat prèviament.
- **Continuations:** Farem servir la classe 'Continuation' ja existent a 'JavaScript' per a implementar 'callcc' i mantenir l'estat dels diferents 'threads' i poder-ne assegurar la correcta execució dels processos.

## 4.5 Implementació

### 4.5.1 Continuations

S'usarà una implementació per a guardar l'estat prèviament explicada en aquest document, és la següent:

```
/* Implementation of callcc(SmallTalk)*/  
  
Continuation.callcc = function(f) {  
  return f(new Continuation());  
}
```

### 4.5.2 Thread

La implementació final de 'thread' que s'ha realitzat és al següent:

```
/* Thread entity */

function Thread(aBlock) {
  this.aBlock = aBlock;
  this.canExecute = false;
}

Thread.prototype.stop = function() {
  this.canExecute = false;
  this.state = Continuation.callcc(function(cc) { return cc; });
}

Thread.prototype.run = function() {
  this.canExecute = true;
  this.state(this.state);
}
```

Podem veure en aquesta implementació que tot 'thread' tindrà les següents dades:

- **aBlock:** Codi a executar
- **canExecute:** És troba en estat d'execució a l'espera?
- **State:** Estat d'execució del 'thread'.

A més oferirà dos mètodes que permetran posar en activitat o aturar l'execució del 'thread', essent aquestes run() i stop().

### 4.5.3 CooperativePool

Aquesta entitat es basarà a gestionar el control d'ordre d'execució. Per tal de fer-ho s'ha decidit usar un 'array' normal pel fet que JavaScript ofereix mètodes que permeten tractar-lo com una cua simple (shift i push). La implementació serà doncs la següent:

```

/* ---Cooperative Pool Entity--- */

function CooperativePool() {
    this.queue = [];
}

CooperativePool.prototype.spawn = function(aBlock) {
    var thread = new Thread(aBlock);
    this.queue.push(thread);
    thread.state = Continuation.callcc(function(cc) { return cc; });
    if (thread.canExecute) thread.aBlock();
};

CooperativePool.prototype.quit = function() {
    this.queue.shift();
    if (this.queue.length > 0) this.queue[0].run();
    else this.halt();
};

CooperativePool.prototype.relinquish = function() {
    this.queue[0].stop();
    if (!this.queue[0].canExecute) {
        this.queue.push(this.queue[0]);
        this.quit();
    }
};

CooperativePool.prototype.start_threads = function() {
    this.halt = Continuation.callcc(function(cc) { return cc; });
    if (this.queue.length > 0) this.queue[0].run();
};

/* ----- */

```

Comencem per explicar que és CooperativePool. Entenem que cooperative pool és una 'pool' de 'threads' que gestionarà l'ordre d'execució d'aquests. Per tal de fer-ho ofereix dues variables:

- **Queue:** Aquesta variable manté els 'threads' que s'han afegit a la 'pool' i el seu ordre d'execució. La necessitat d'aquesta cua ha estat explicada abans amb més detall. Val la pena recordar que aquesta 'queue' mantindrà referències a objectes Thread.
- **Halt():** Podem veure la creació d'aquesta variable a start-threads. Aquesta variable manté l'estat d'execució a l'inici de l'execució de la 'pool' i s'usarà per a restablir-lo un cop acabada l'execució de la 'pool'.

Passem ara a explicar les diferents funcions que es donen a l'objecte CooperativePool. Recordem que la idea de cada una de les funcions ha estat explicada en el punt (Definició de les funcionalitats demanades) i per tant no tornarà a ser explicat sinó que ens basarem a explicar la implementació una per una de les diferents funcionalitats.

**Spawn** (Aquesta funcionalitat hauria de permetre afegir un thread a la pool)

```
CooperativePool.prototype.spawn = function(aBlock) {  
  var thread = new Thread(aBlock)  
  this.queue.push(thread);  
  thread.state = Continuation.callcc(function(cc) { return cc; });  
  if (thread.canExecute) thread.aBlock();  
};
```

Veiem que per a la implementació del mètode el primer que fem és instanciar un 'thread' nou amb el bloc de codi que s'hagi assignat com a paràmetre. Posteriorment afegim aquest 'thread' a la cua d'execució i un cop realitzat li assignem l'estat d'execució en el que s'inicialitzarà. Podríem preguntar-nos per què, no assignem l'estat en la creadora de Thread i així ajuntem totes les inicialitzacions. El fet és que en tal cas, quan el 'thread' s'actives (run()) i es poses en execució, passaria per "this.queue.push(thread)" i s'afegiria a la llista de nou, donant un resultat erroni.

Posteriorment s'afegeix una línia de codi que permeti que, en cas que un 'thread' acabat de crear recuperi el seu estat d'execució (state) guardat just al codi explicat anteriorment (línia de sobre), en cas de tenir el permís d'execució, executi el bloc de codi assignat.

**Quit** (Aquesta funcionalitat ha de permetre tancar l'execució d'un thread i fer-lo fora de la llista)

```
CooperativePool.prototype.quit = function() {  
  this.queue.shift();  
  if (this.queue.length > 0) this.queue[0].run();  
  else this.halt();  
};
```

En aquest cas el que es realitza en la implementació és, primerament treure el primer element de la cua per eliminar el 'thread' que està executant actualment i posteriorment s'avalua si a la cua hi ha més 'threads' esperant per seguir amb el seu procés. En cas afirmatiu, s'activa el següent 'thread'. En cas contrari, es torna a l'estat d'execució del moment de creació de la Pool. (Recordem que això quedarà més clar un cop explicat start-thread).

**Relinquish** (Aquesta funcionalitat ha de permetre frenar l'execució d'un thread per a reafegir-lo a la cua i donar pas al següent thread de la cua)

```
CooperativePool.prototype.relinquish = function() {  
  this.queue[0].stop();  
  if (!this.queue[0].canExecute) {  
    this.queue.push(this.queue[0]);  
    this.quit();  
  }  
};
```

Primerament en aquesta implementació, parem l'execució del 'thread' que s'està executant (primer element de la cua). Posteriorment un cop parat, l'afegim de nou a la cua generant una còpia d'aquest, és a dir, que en aquest moment tenim el mateix 'thread' dos cops a la cua un en posició 0 i un en posició queue.length()-1. Un cop fet això realitzem un quit sobre la CooperativePool per tal de treure el 'thread' en posició 0 i executar el següent 'thread' a la cua.

Podem veure però que existeix també un if que comprova que el 'thread' estigui en actiu. A primera vista pot semblar què és un error, ja que prèviament realitzem stop() al 'thread' cosa que assegura que no estarà actiu però té una explicació i és totalment necessari. L'explicació és la següent, sabem que el thread que estem aturant i posant al final de la cua, guardarà un estat d'execució que en ser activat de nou el tornarà a fer passar per la funció relinquish, en aquest cas la continuació el tornarà a "this.queue[0].stop()" fet que farà que passi pel if, i a l'estar actiu no hi entrarà, seguint així amb la seva correcta execució.



**Start threads** (Aquesta funcionalitat ha de donar pas a iniciar l'execució dels diferents processos acumulats a la pool)

```
CooperativePool.prototype.start_threads = function() {  
  this.halt = Continuation.callcc(function(cc) { return cc; });  
  if (this.queue.length > 0) this.queue[0].run();  
};
```

La implementació d'aquest mètode és prou simple i tan sols consta de dues línies. En la primera podem veure que es guarda una continuació del punt on s'inicia l'execució de start-threads en la variable halt. Aquest fet serà el que posteriorment un cop acabada l'execució de la 'pool' ens donarà accés a sortir de l'execució tot restaurant l'estat d'entrada a l'execució.

La segona línia comprova que existeixin elements a la cua i en cas afirmatiu, dóna permís d'execució al primer element de la cua. Aquest if pot semblar que no és rellevant però recordem que quan es cridi halt, s'haurà de passar per la funció start-threads i per tant si no el poséssim, s'intentaria executar el queue[0] provocant un error, ja que, no n'hi hauria cap 'thread' a la cua.

#### 4.5.4 Funcio make thread system

Un cop generat tot el codi anteriorment explicitat, tan sols fa falta el següent codi:

```
function make_thread_system() {  
  return new CooperativePool();  
}
```

Veiem que tan sols faria falta retornar una nova instància de CooperativePool donat que, els mètodes es troben al prototipus o per tant també serien delegats a CooperativePool.

## 4.6 Test

Per a la realització del test de la implementació s'han usat els dos codis facilitats per a la realització de la pràctica. S'ajunta el codi dels dos com a recordatori. S'ha afegit format per a una millor visualització per consola dels resultats.

```

1
2  /* First example */
3
4  print("\n" + "=== First Example ===" + "\n");
5
6  var counter = 10;
7  function make_thread_thunk(name, thread_system) {
8
9      function loop() {
10         if (counter < 0) {
11             thread_system.quit();
12         }
13         print("in thread", name, "; counter =", counter);
14         counter--;
15         thread_system.relinquish();
16         loop();
17     };
18     return loop;
19 }
20
21 var example1_thread_sys = make_thread_system();
22 example1_thread_sys.spawn(make_thread_thunk('a',
23     example1_thread_sys));
24 example1_thread_sys.spawn(make_thread_thunk('b',
25     example1_thread_sys));
26 example1_thread_sys.spawn(make_thread_thunk('c',
27     example1_thread_sys));
28 example1_thread_sys.start_threads();
29
30 /* Second Example */
31
32 print("\n" + "=== Second Example ===" + "\n");
33
34 function make_thread_thunk2(name, thread_system) {
35     function loop() {
36         for(let i=0; i < 5; i++) {
37             print('in thread', name, '; i =', i);
38             thread_system.relinquish();
39         }
40         thread_system.quit();
41     };
42     return loop;
43 }
44 var example2_thread_sys = make_thread_system();

```

```
42 example2_thread_sys.spawn(make_thread_thunk2('a',  
    example2_thread_sys));  
43 example2_thread_sys.spawn(make_thread_thunk2('b',  
    example2_thread_sys));  
44 example2_thread_sys.spawn(make_thread_thunk2('c',  
    example2_thread_sys));  
45 example2_thread_sys.start_threads();  
46  
47 print("\n");
```

Listing 1: Tests

Com que hem vist que funcionava pels dos casos, hem decidit acceptar la implementació donada.