

Conceptes Avançats de Programació

Programació basada en Prototipus

Prototype-Based Programming
J.Noble, A.Taivalsaari & I.Moore (eds.)
Springer 1999

JavaScript: The Good Parts
Douglas Crockford
O'Reilly 2008

Origen:

Lieberman, H.

Using prototypical objects to implement shared behavior in object-oriented systems.

OOPSLA'86

Diferents llenguatges apareixen durant els '80 i '90

Self

Object Lisp

Garnet

Amulet

Agora

Moostrop

NewtonScript

Kevo

Omega

Obliq

Yafool

JavaScript

Idea principal: No hi ha classes

*Més enllà d'això, els llenguatges BP són
ben bé com els llenguatges OO.*

Es pretén:

- Simplificar la descripció dels objectes
- Incrementar l'adaptabilitat
- Simplificar el model de programació

Idea general:

El model de programació BP consisteix, *a grans trets*, en:

- Objectes amb atributs i mètodes
- Tres maneres primitives de crear objectes (*ex nihilo, clonatge i extensió*)
- Un mecanisme primitiu de computació (*enviament de missatges*)

Idea important:

Dins del model de programació BP és central la noció de *delegació*.

La idea és que si un missatge enviat a un objecte no és entès per l'objecte, aquest pot *delegar* la resposta del missatge a un altre objecte. Així es dóna suport a allò essencial de l'herència: La modificació incremental. L'objecte al que es delega se l'anomena *pare*.

JavaScript: Parlarem del llenguatge Javascript com a exemple de llenguatge basat en prototipus.

Per tant, el que segueix pretén explicar precisament els objectes a Javascript, com funciona concretament la relació *ser-prototipus-de*, i com això es reflecteix en la manera en que determinats patrons coneguts (p.ex. a IES) són implementats en Javascript (*que quedi clar, doncs, que no volem ensenyar Javascript per sí mateix*)

JavaScript: Sobre les classes a ECMAScript 6...

The screenshot shows the MDN web docs page for JavaScript Classes. The page has a header with the MDN logo, a search bar, and navigation links. The main heading is 'Classes'. Below it, there are links to 'Jump to:' sections like 'Defining classes', 'Class body and method definitions', 'Sub classing with extends', and 'Species'. A sidebar on the left shows the breadcrumb trail: 'Web technology for developers > JavaScript > JavaScript reference > Classes'. The main content area contains a paragraph about JavaScript classes.

MDN web docs
moz://a

Search

Technologies ▾ References & Guides ▾ Feedback ▾ Sign in

Classes

Languages Edit

Jump to: Defining classes Class body and method definitions Sub classing with extends Species

Super class calls with super Mix-ins Specifications Browser compatibility Running in Scratchpad See also

Web technology for developers >
JavaScript >
JavaScript reference > Classes

JavaScript classes, introduced in ECMAScript 2015, are primarily syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax *does not* introduce a new object-oriented inheritance model to JavaScript.

Prototipus

En primera aproximació, podem dir que tot objecte Javascript té associat un segon objecte (o **null**, però això és poc freqüent), anomenat *prototipus*. El primer objecte hereta propietats del seu prototipus. El lligam entre els dos està *ocult*.

Però... Què és un objecte en Javascript?
 Què és una propietat?

Objectes: Col.leccions de parelles nom/valor

Si un objecte conté la parella 'x/1' es diu que té la *propietat x de valor 1*, i usualment es denota 'x : 1'

El nom d'una propietat pot ser qualsevol *string*, i no pot haver dues propietats amb el mateix nom dins del mateix objecte.

Les propietats són *dinàmiques*, poden ser afegides i/ o esborrades d'un objecte en temps d'execució. Diem que els objectes són *mutables*, i es manipulen per *referència* i no per valor.

Creació d'Objectes: Objectes literals.

Llista de parelles 'nom : valor' separades per comes (el separador del 'nom' i del 'valor' és el ':').

```
var flight = {  
  airline: "Oceanic",  
  number: 815,  
  departure: {  
    IATA: "SYD",  
    time: "2004-09-22 14:55",  
    city: "Sydney"  
  },  
  arrival: {  
    IATA: "LAX",  
    time: "2004-09-23 10:42",  
    city: "Los Angeles"  
  }  
}
```

Tot objecte té un enllaç *ocult* cap al seu prototipus, que, en aquest cas, és l'objecte anomenat **Object.prototype**

Creació d'Objectes: Objectes literals.

A les propietats s'hi accedeix amb la notació del `.` o bé com si fos una taula associativa. Aquest accés es el mateix tant per a la consulta com per a la modificació.

```
flight.airline      // --> "Oceanic"  
flight["airline"]   // --> "Oceanic"  
  
flight.departure["city"] // --> "Sydney"  
  
flight["arrival"].city = "Unknown";  
flight.arrival.city = "Unknown";
```

Creació d'Objectes: Objectes literals.

Modismes per proporcionar valors per defecte:

```
var status = flight.status || "unknown";
```

hasOwnProperty per diferenciar propietats pròpies de les heretades:

```
flight.hasOwnProperty('number') // => true  
flight.hasOwnProperty('constructor') // => false
```

Creació d'Objectes: Objectes literals.

Enumeració de propietats:

```
var name;
for (name in object) {
    if (object.hasOwnProperty(name)) { ... }
    if (typeof object[name] !== 'function') { ... }
}
```

L'operador `delete`: esborra la propietat pròpia de l'objecte, si en té una (no toca els prototipus):

```
delete flight.number
```

Creació d'Objectes: Crear objectes amb **new**

L'operador **new** crea i inicialitza un objecte. La paraula clau **new** s'ha d'aplicar a una invocació de *funció*. A aquesta funció l'anomenarem *constructor* i serveix per inicialitzar l'objecte creat.

```
function Range(from,to) {  
    this.from = from;  
    this.to = to;  
}
```

```
var r = new Range(1,10);
```

Creació d'Objectes: Crear objectes amb `new`

Quan invoquem el constructor amb `new`, el següent passa dins una funció:

- Es crea un objecte buit que és referenciat per la variable `this`, *heretant l'objecte referenciat per la propietat `prototype` de la funció.*
- Propietats i mètodes s'afegeixen a l'objecte referenciat per `this`.
- L'objecte nou creat i referenciat per `this` es retorna implícitament (si és que no es retorna cap altre objecte explícitament).

Creació d'Objectes: Crear objectes amb **new**

Per tant, si volem que tots els objectes creats amb un constructor determinat disposin d'uns mètodes determinats, cal afegir-los al objecte referenciat per la propietat **prototype** del constructor:

```
function Range(from,to) {  
    this.from = from;  
    this.to = to;  
}  
  
Range.prototype.includes = function(x) {  
    return this.from <= x && x <= this.to;  
};  
  
Range.prototype.foreach = function(f) {  
    for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);  
};
```

Creació d'Objectes: Crear objectes amb `new`

Cal tenir en compte:

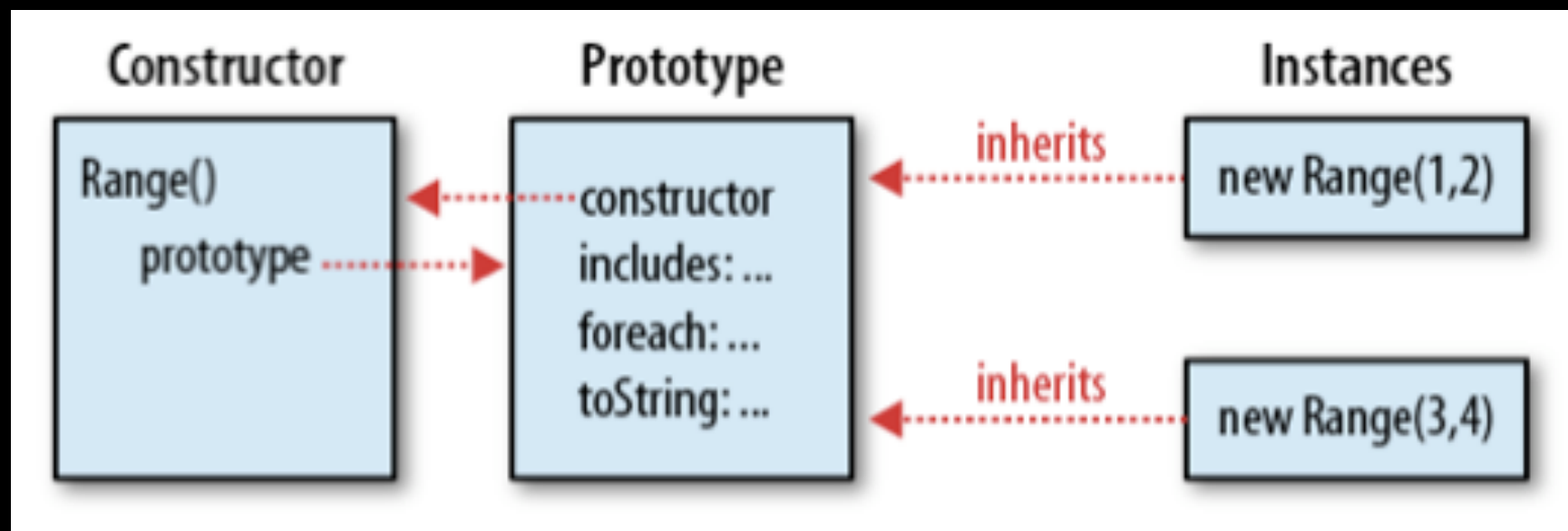
- El prototipus de tota funció és `Function.prototype`
- Tota funció a Javascript té una propietat `prototype`
- L'objecte referenciat per aquesta propietat de les funcions té una propietat, `constructor`, que referencia la funció.

```
var F = function() {}; // Això és un objecte funció.  
var p = F.prototype;   // Aquest és l'objecte prototipus associat.  
var c = p.constructor; // Funció associada al prototipus.  
c === F // => true: F.prototype.constructor===F per a qualsevol funció
```

```
var o = new F();        // Crea un objecte de "classe" F  
o.constructor === F    // => true
```

Creació d'Objectes: Crear objectes amb **new**

Tornant a l'exemple de **Range**, tindriem:



JavaScript, The Definitive
Guide (6th. ed.),
David Flanagan, 2011,
p. 204

```
Range.prototype = {  
  constructor: Range, // Dóna valor a la propietat 'constructor' explícitament  
  includes: function(x) { return this.from <= x && x <= this.to; },  
  foreach: function(f) {  
    for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);  
  }  
};  
// atenció que això no fa exactament el mateix que el que ja s'ha  
// vist (t.32), ja que si ja hi ha objectes creats amb Range() no es  
// veuran afectats per aquesta assignació
```

Creació d'Objectes: `Object.create`

`Object.create(p)` crea un objecte amb `p` com a prototipus (només a ECMAScript 5).

Podem aprofitar això per crear objectes especificant directament l'objecte que volem com a prototipus, tant si estem a ECMAScript 5 com si no.

```
function inherit(p) {
    if (p == null) throw TypeError(); // p no ha de ser null

    if (Object.create)                // Object.create (ECMAScript 5)
        return Object.create(p);      // potser ja existeix
                                      // fer-lo servir si és així

    var t = typeof p;
    if (t !== "object" && t !== "function") throw TypeError();
    function f() {};                  // fer servir una funció "de mentida"
    f.prototype = p;                  // per assignar-hi el prototipus
    return new f();                    // i forçar-ne l'herència
}
```

Prototipus (altre cop): L'objecte prototipus d'un objecte és fonamental per a la *identitat d'un objecte*. Com no hi ha classes, la manera d'identificar dos objectes que pertanyen a la mateixa categoria (*instàncies de la mateixa classe?*) és mirar si hereten del mateix objecte prototipus. El constructor no és important.

Així, dos constructors diferents construeixen *instàncies de la mateixa classe* si les seves propietats **prototype** referencien *el mateix objecte*.

Prototipus (altre cop):

Així, l'operador `instanceof` no mira si un objecte ha estat creat amb un determinat constructor, si no si hi ha una relació d'herència amb el prototipus:

`r instanceof R`

serà cert si `r` hereta del prototipus de `R`

Tot i així, el nom del constructor s'acostuma a fer servir com a nom de la classe

Exemple: Simular una classe clàssica amb JavaScript

```
// constructor
```

```
function Complex(real,imaginary) {  
    if (isNaN(real) || isNaN(imaginary)) throw new TypeError();  
    this.r = real;  
    this.i = imaginary;  
}
```

```
// mètodes es defineixen al prototipus del constructor
```

```
Complex.prototype.add = function (that) {  
    return new Complex(this.r + that.r, this.i + that.i);  
};
```

```
// ...                // mètodes i variables de classe es defineixen com a propietats  
                        // del constructor
```

```
Complex.ZERO = new Complex(0,0);  
Complex.ONE = new Complex(1,0);
```

```
Complex.parse = function(s) { . . . }
```

```
// mètodes i variables privats... per convenció  
Complex._format = ...
```

Exemple: Simular una classe *clàssica* amb JavaScript

I si volguéssim fer subclasses? Altre cop, la clau està en els prototipus. Suposem la classe **B** vol ser subclasse (*extends*) de la classe **A**.

Després del que hem vist, només caldria forçar l'herència entre prototipus, de manera que el prototipus dels objectes instància de **B** heretés del prototipus dels objectes instància d'**A** (i el `call`).

```
B.prototype = inherit(A.prototype);  
B.prototype.constructor = B;
```


Prototipus (seguim): El prototipus d'un objecte és un *atribut*, no una propietat. A ECMAScript 5 podem demanar pel prototipus d'un objecte amb `Object.getPrototypeOf(...)`:

```
var p = {};  
Object.getPrototypeOf(p)    // ==> Object.prototype  
  
var o = Object.create(p);  
Object.getPrototypeOf(o)    // ==> p
```

Alguns navegadors permeten accedir al prototipus via una propietat anomenada `__proto__`, però això no és portable.

1.- Tenim el següent mètode, digues què fa:

```
Function.prototype.method = function (name, func) {  
    this.prototype[name] = func;  
    return this;  
}
```

2.- Donada una funció Foo, preguntem el següent:

```
Foo.prototype === Object.getPrototypeOf(Foo)    false  
Foo.prototype === Object.getPrototypeOf(new Foo()) true  
Function.prototype === Object.getPrototypeOf(Foo) true  
Object.prototype === Object.getPrototypeOf(Function.prototype) true  
Function.prototype === Object.getPrototypeOf(Function) true  
Object.prototype === Object.getPrototypeOf(Foo.prototype) true  
Function.prototype === Object.getPrototypeOf(Object) true  
null === Object.getPrototypeOf(Object.prototype) true
```

Funcions: Les funcions a JavaScript són *objectes*.
La diferència respecte dels altres objectes és que les funcions són objectes *invocables*.

- Poden ser creades dinàmicament, en temps d'execució
- Poden ser assignades a variables
- Poden ser arguments d'altres funcions i poden ser retornades per altres funcions
- Poden tenir propietats i mètodes (com qualsevol altre objecte)

Funcions: Les funcions a JavaScript són *objectes*.

Les funcions poden ser *declarades* o podem especificar-les en *expressions*

```
function foo(...) { ... }           // declaració
var bar = function (...) { ... }     // expressió
var baz = function baz(...) { ... } // expressió amb nom

foo.name // "foo"
bar.name // ""
baz.name // "baz"
```

La propietat `name` no és estàndar

Funcions: Les funcions a JavaScript són *objectes*.

L'abast de les variables a JavaScript és un abast de funció (*function scope*).

```
function foo() { return 'global foo' };
function bar() { return 'global bar' };
```

```
function hoistMe() {
  console.log(typeof foo); // ??? function
  console.log(typeof bar); // ??? undefined

  console.log(foo()); // ??? local foo
  console.log(bar()); // ??? TypeError{ bar is not a function. (In 'bar()', 'bar' is undefined) }

  function foo() { return 'local foo' };
  var bar = function() { return 'local bar' };
}
hoistMe();
```

Funcions: Les funcions a JavaScript són *objectes*.

En realitat les funcions són *Closures*: Funcions + el seu context lèxic *en el moment de la creació de la funció*. Són fonamentals a JavaScript

```
var myObject = function () {  
    var value = 0;  
  
    return {  
        increment: function (inc) {  
            value += typeof inc === 'number' ? inc : 1;  
        },  
        getValue: function () {  
            return value;  
        }  
    };  
}();
```

```
> myObject  
Object {  
  getValue: f()  
  increment: f(inc)  
}
```

Closures: Funcions amb el seu context lèxic

Exemple: La pila

```
var stackCreator = function () {
    var index = 0;
    var arr = [];

    return {
        push : function(val) {
            arr[index] = val;
            index++;
        },

        pop : function() {
            index--;
        },

        top : function() {
            return arr[index-1];
        },

        size : function() {
            return index;
        },

        empty : function() {
            return (index == 0);
        }
    };
};
```

```
var myStack1 = stackCreator();
var myStack2 = stackCreator();

myStack1.push(1);
myStack1.push('hola');
myStack1.push(1.5);
myStack1.push('adeu');

console.log(myStack1.top());
myStack1.pop();
console.log(myStack1.top());
myStack1.pop();
console.log(myStack1.top());
myStack1.pop();
console.log(myStack1.top());
myStack1.pop();
console.log(myStack1.empty());

myStack2.push('a');
myStack2.push('b');
myStack2.push('c');
myStack2.push('d');

console.log(myStack2.top());
myStack2.pop();
console.log(myStack2.top());
myStack2.pop();
console.log(myStack2.top());
myStack2.pop();
console.log(myStack2.top());
myStack2.pop();
console.log(myStack2.empty());
```

Funcions: Les funcions a JavaScript són *objectes*.

Hi ha quatre maneres d'invocar les funcions:

- La invocació com a Mètode
- La invocació com a funció
- La invocació com a constructor
- La invocació 'apply/call'

En què es diferencien? En el lligam de `this` dins la funció invocada (`this` és una paraula clau, no una variable ni una propietat).

Funcions: La invocació com a Mètode

Una funció pot ser emmagatzemada com a propietat d'un objecte, en aquest cas l'anomenem *mètode*. Quan invoquem un mètode, **this** queda associat a l'objecte del que la funció és una propietat.

```
var myObject = {  
    value : 0;  
  
    increment: function (inc) {  
        this.value += typeof inc === 'number'  
            ? inc : 1;  
    }  
}
```

Funcions: La invocació com a Funció

Una funció també pot existir sense ser la propietat de cap objecte (recordem que la funció *sí* és un objecte).

En aquest cas la invocació vincula **this** a l'objecte global.

Això pot ser un problema amb les funcions internes a mètodes

Funcions: La invocació com a Funció

Exemple:

```
myObject.double = function () {  
    var helper = function () {  
        this.value = 2 * this.value; // Objecte global!!!!  
    };  
    helper();  
}
```

Funcions: La invocació com a Funció

Exemple:

```
myObject.double = function () {  
    var that = this;  
  
    var helper = function () {  
        that.value = 2 * that.value; // Objecte  
    };  
  
    helper();  
}
```

Funcions: La invocació com a Funció

Exercici:

```
var o = {
  m: function() {
    var self = this;
    console.log(this === o);      // ??? true
    f();

    function f() {
      console.log(this === o);    // ??? false
      console.log(self === o);    // ??? true
    }
  }
}

o.m();
```

Funcions: La invocació com a Constructor

Una funció sempre pot utilitzar-se com a constructor (totes les funcions tenen la propietat **prototype**).

En aquest cas, invocada amb l'operador **new**, el lligam de **this** es fa amb l'objecte tot just creat, tal i com hem vist.

Funcions: La invocació 'apply/call'

Una funció és un objecte, i per tant pot tenir mètodes. Una mostra en són `apply` i `call`, que ens permeten l'aplicació de la funció indirectament, controlant l'associació a `this`.

```
var Obj = function(n) { this.value = n; };  
Obj.prototype.double = function () {  
    this.value = 2 * this.value;  
};  
var foo = new Obj(7);  
console.log(foo.double());
```

mirar examen 14-15, Pregunta 4
sobre atributs "heredats"

```
var bar = { value : 7 };  
console.log(Obj.prototype.double.apply(bar));
```

call -> has de posar els arguments que necessita
apply -> has de passar-li un array amb els arguments

Funcions: Els arguments

`f.call (obj, ...lst)`

`f.apply (obj, lst)`

sent `lst` un array, te'l desplega (spread syntax)
és equivalent

Quan una funció s'invoca amb menys arguments que paràmetres declarats, els paràmetres que no han rebut cap valor són **undefined**. Així, podem fer funcions amb paràmetres opcionals, però els hem de posar al final de la declaració.

Si s'invoca amb més arguments que paràmetres declarats hem d'accedir als arguments *sobrants* amb l'objecte **arguments**, que és *com* un **array** amb tots els paràmetres passats a la funció

Funcions

Exemple: `factorial`. Veurem un factorial que aprofita que les funcions tenen propietats per fer un *auto-cache*:

```
function factorial(n) {  
    if (isFinite(n) && n>0 && n==Math.round(n)) {  
        if (!(n in factorial))  
            factorial[n] = n * factorial(n-1);  
        return factorial[n];  
    }  
    else return NaN;  
}  
factorial[1] = 1;
```



CAP: PBP: *JavaScript*



Funcions: Exercici: `memoize`. Donada una funció retornar una funció que sigui capaç de recordar valors ja calculats.

Funcions: Exercici: `memoize`. Donada una funció retornar una funció que sigui capaç de recordar valors ja calculats.

```
function memoize(f) {  
    var cache = {}; // Value cache stored in the closure.  
  
    return function() {  
        // Create a string version of the  
        // arguments to use as a cache key.  
        var key = arguments.length +  
                    Array.prototype.join.call(arguments, ",");  
        if (key in cache) return cache[key];  
        else return cache[key] = f.apply(this, arguments);  
    };  
}
```

Patrons de disseny

Veure els exemples:

Singleton

Factory

Decorator

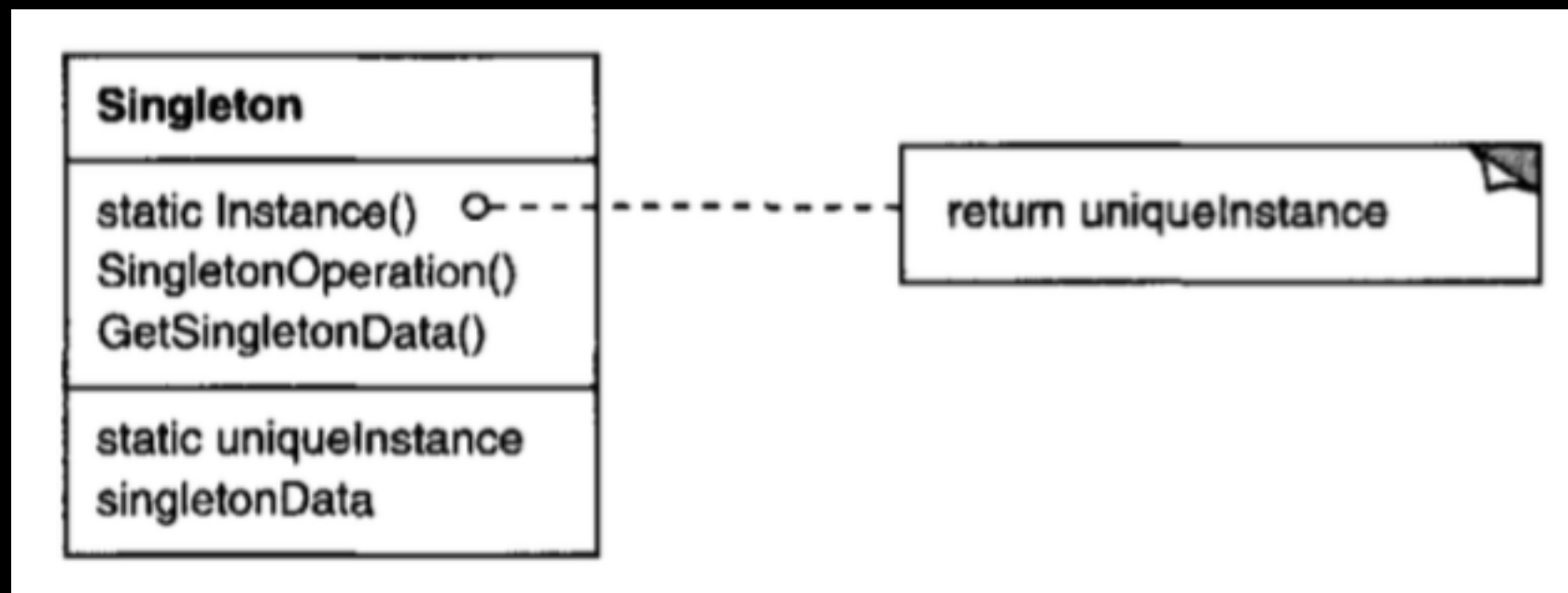
JavaScript Patterns

Stoyan Stefanov

O'Reilly 2010

CAP: PBP: *JavaScript*

Patrons de disseny *Singleton*



```

function Universe()
{
    if (typeof Universe.instance === "object") return Universe.instance;
    this.start_time = 0;
    this.bang = "Big";
    Universe.instance = this;
}
Universe.prototype.nothing = true
...
    
```

BÉ

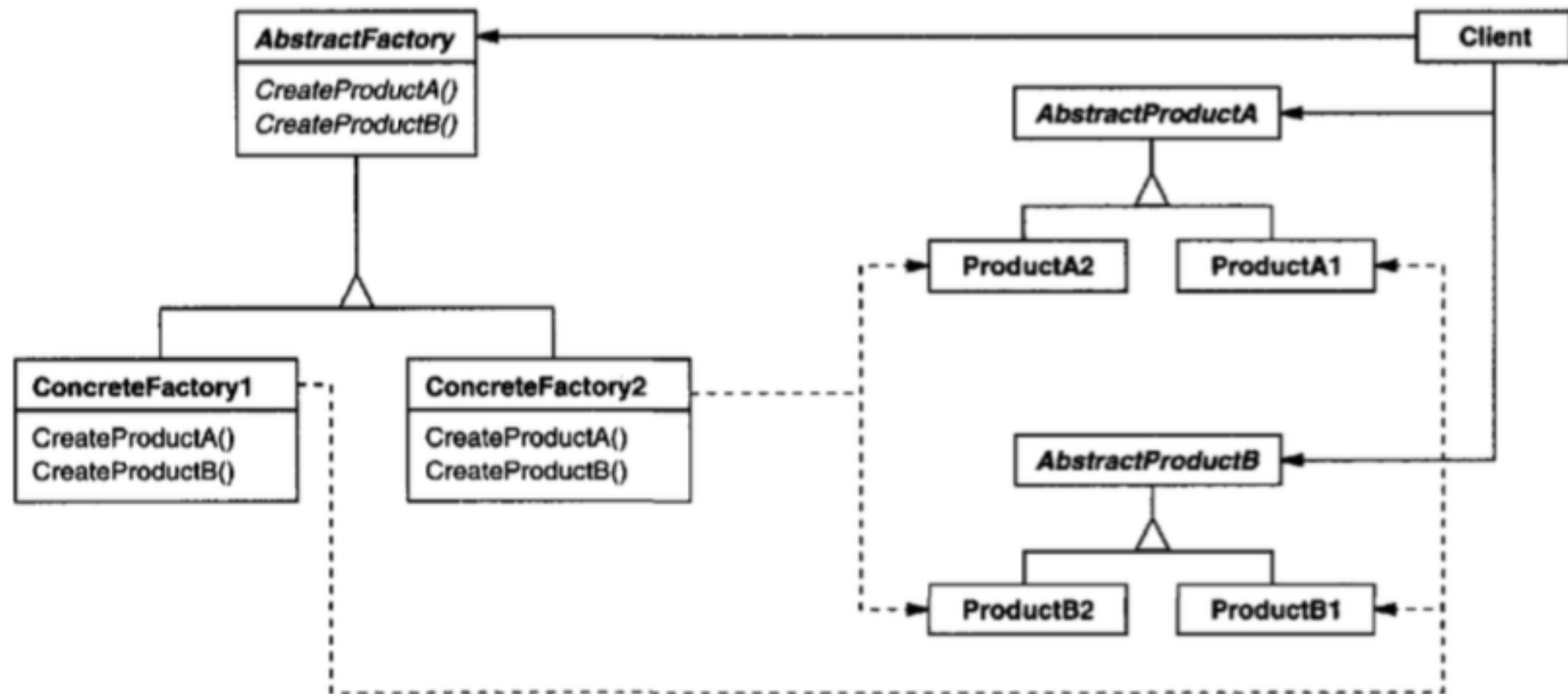
```

function Universe()
{
    var instance = this;
    this.start_time = 0;
    this.bang = "Big";
    Universe = function Universe() { return instance }
}
var uni = new Universe();
Universe.prototype.nothing = true
...
    
```

MALAMENT
nothing undefined

CAP: PBP: *JavaScript*

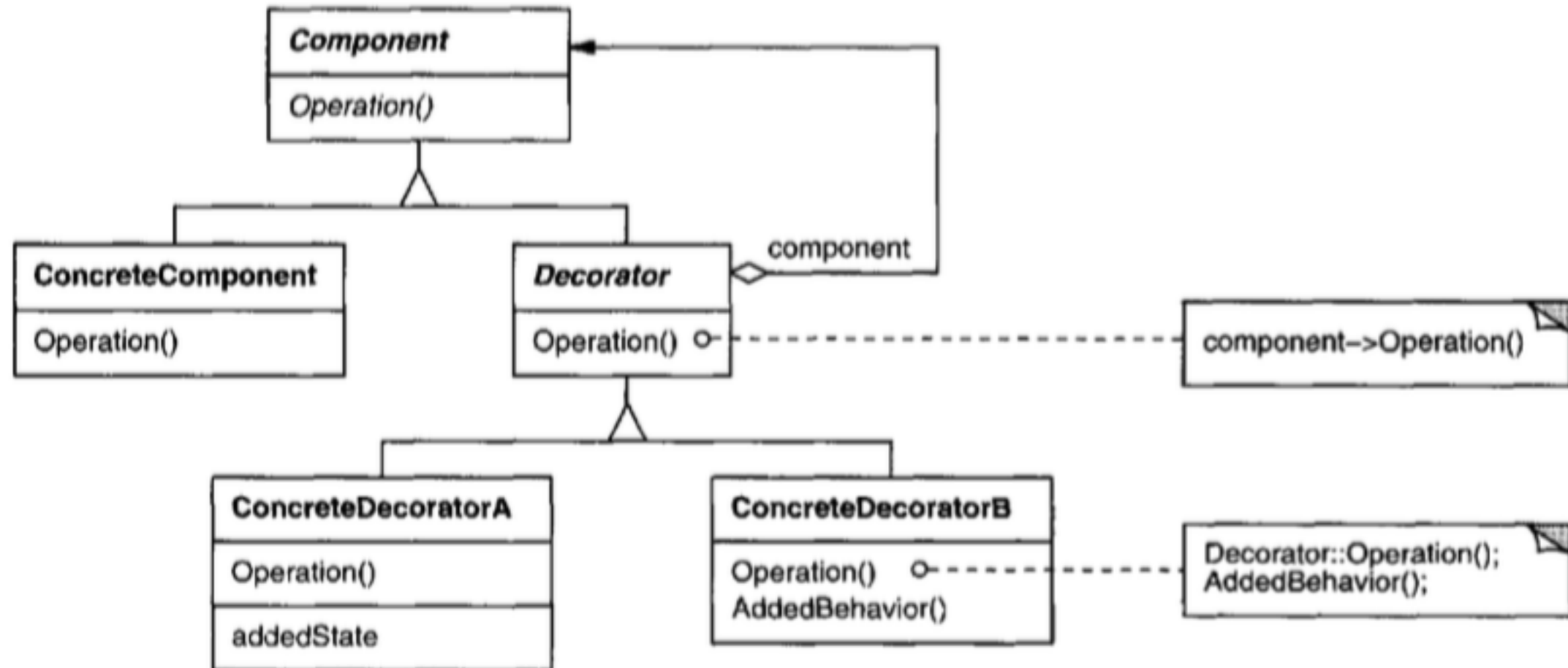
Patrons de disseny Factory



CAP: PBP: *JavaScript*

Patrons de disseny

Decorator



CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Tail Position

A function call is in tail position if the following criteria are met:

- The calling function is in **strict** mode.***
- The calling function is either a normal function or an arrow function.***
- The calling function is **not a generator** function.***
- The return value of the called function is returned by the calling function.***

<https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/>

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

*When a function call is in tail position, ECMAScript 6 mandates that such a call must reuse the stack space of its own frame instead of pushing another frame onto the call stack. To emphasize, **ECMAScript 6 requires that a call in tail position will reuse the caller's stack space**. The calling function's frame is called a tail deleted frame as it is no longer on the stack once it makes a tail call.*

<https://webkit.org/blog/6240/ecmascript-6-proper-tail-calls-in-webkit/>

CAP: PBP: *JavaScript*

Continuation-Passing Style *(CPS)*

Proper Tail Calls ***(PTC)***

Les *Proper Tail Calls* permeten optimitzar les crides a funcions que estan en *tail position* no creant un nou *stack frame*.

La manera en que s'invoca una funció NO importa (recordem que n'hi ha 4 maneres en Javascript). Només importa si està en *tail position*.

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

```
function factorial(x) {  
  if (x <= 1) {  
    return 1;  
  }  
  return x * factorial(x-1);  
}
```

crida: factorial(x,1)

```
function factorial (x, acc) {  
  if (x <= 1) {  
    return acc;  
  }  
  return factorial(x-1, x*acc);  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

```
function foo() {  
    bar();  
}
```

és equivalent a:

```
function foo() {  
    bar();  
    return undefined;  
}
```

```
function foo() {  
    return bar();  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

```
function forEach(arr, body, start) {  
  if (0 <= start && start < arr.length) {  
    body(arr[start], start, arr);  
    return forEach(arr, body, start+1);  
  }  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

És imprescindible tenir PTC (o tail call optimization, que ve a ser el mateix) per poder programar en CPS

PTC a node.js:

`node --harmony --use_strict`

versió de JS 6!!!

PTC a Rhino:

```
java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar  
org.mozilla.javascript.tools.shell.Main -opt -2
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

```
function forEach(arr, body, start) {  
  if (0 <= start && start < arr.length) {  
    body(arr[start], start, arr);  
    forEach(arr, body, start+1);  
  }  
}
```

```
$ node --harmony --use_strict  
> forEach([...Array(100000).keys()],function (elem, i) { console.log(i, elem ) } )  
.  
.  
.  
10981 10981  
10982 10982  
RangeError: Maximum call stack size exceeded (!!)
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Proper Tail Calls (PTC)

```
function forEach(arr, body, start) {
  if (0 <= start && start < arr.length) {
    body(arr[start], start, arr);
    return forEach(arr, body, start+1);
  }
}
```

```
$ node --harmony --use_strict
> forEach([...Array(100000).keys()],function (elem, i) { console.log(i, elem ) } )
. . .
99997 99997
99998 99998
99999 99999
```


CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

- **Prohibit utilitzar cap expressió en un *return*: Es retorna sempre el resultat d'una crida a funció (en *tail position*), o una constant.**
- **El darrer paràmetre d'una funció és sempre la seva continuació.**
- **Cada funció ha d'acabar cridant la seva continuació amb el resultat del seu càlcul.**

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció identitat:

```
function id(x) {  
    return x;  
}
```



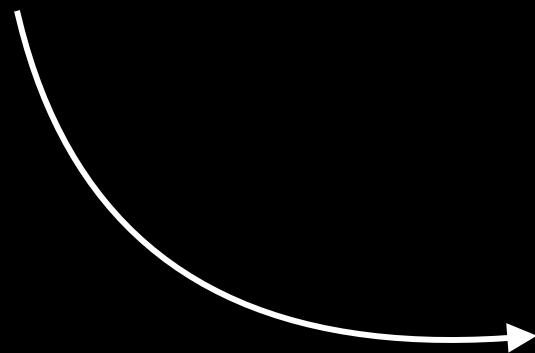
```
function id_cps(x, ret) {  
    return ret(x);  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció factorial:

```
function fact(n) {  
  if (n <= 1) { return 1; }  
  else { return n * fact(n-1) }  
}
```



```
function fact_cps(n, ret)  
{  
  if (n <= 1) return ret(1);  
  else  
  {  
    return fact_cps  
    (  
      n-1,  
      function(v) { return ret(v*n) }  
    )  
  }  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció coeficient binomial:

```
function binomial_coef (n,k)
{
  return fact(n) / (fact(k) * fact(n-k)) ;
}
```

```
function binomial_coef_cps (n,k,ret)
{
  return fact_cps
  (
    n,
    function (factn)
    {
      return fact_cps
      (
        n-k,
        function (factnk)
        {
          return fact_cps
          (
            k,
            function (factk)
            {
              return ret (factn / (factnk * factk))
            }
          )
        }
      )
    }
  )
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció n-èssim nombre de Fibonacci:

```
function fib(n) {
  if (n < 2) { return 1 }
  else { return fib(n-1) + fib(n-2) }
}
```

igual d'ineficient



```
function fib_cps(n, ret) {
  if (n < 2) { return ret(1); }
  else { return fib_cps(n-1, function(fibn1) {
    return fib_cps(n-2, function(fibn2) {
      return ret(fibn1 + fibn2)}))}}}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció eliminar un element *e* d'un *array*:

```
function remove(arr,e) {
  if (arr.length == 0) {
    return [];
  } else {
    var [car,...cdr] = arr;
    var rem = remove(cdr,e)
    if (car != e) {
      rem.unshift(car)
    }
    return rem;
  }
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció eliminar un element *e* d'un *array*:

```
function remove_cps(arr,e,ret)
{
  if (arr.length === 0)
  {
    return ret([]);
  }
  else
  {
    var [car,...cdr] = arr;
    return remove_cps
    (
      cdr,
      e,
      function (rcdr)
      {
        if (car !== e) rcdr.unshift(car);
        return ret(rcdr);
      }
    )
  }
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció escriure els elements d'un *array*:

```
function escriuArray(arr) {  
  for(var i=0; i < arr.length; i++) {  
    console.log(arr[i]);  
  }  
  console.log("Done");  
}
```


CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

Exemple: La funció escriure els elements d'un *array*:

```
function escriuArray(arr) {  
    return forEachCps(arr,  
        function (elem, index, next) {  
            console.log(elem);  
            return next();  
        },  
        function () {  
            console.log("Done");  
        });  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

on tenim la funció auxiliar:

```
function forEachCps(arr, visitor, done) {  
    return forEachCpsRec(0, arr, visitor, done)  
}
```

```
function forEachCpsRec(index, arr, visitor, done) {  
    if (index < arr.length) {  
        return visitor(arr[index],  
                        index,  
                        function () {  
                            return forEachCpsRec(index+1, arr, visitor, done);  
                        });  
    } else {  
        return done();  
    }  
}
```

CAP: PBP: *JavaScript*

Continuation-Passing Style (CPS)

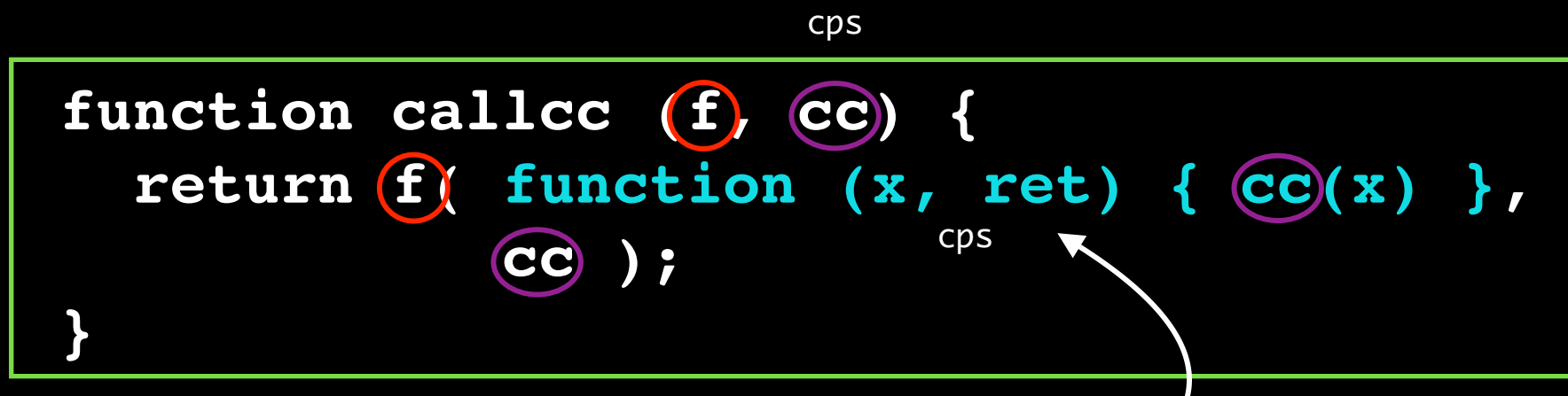
Imaginem que ***tot*** un programa està escrit en estil CPS.

Aleshores `callcc(f,c)` és molt fàcil d'implementar, ja que tenim la continuació explícitament en tot moment.

cps

```
function callcc (f, cc) {  
  return f( function (x, ret) { cc(x) },  
            cc );  
}
```

cps



ret és ignorat en favor de la
continuació que existia en el
moment de crear-la

CAP: PBP: *JavaScript*

Continuation-Passing Style

(CPS)

```
function callcc (f, cc)
{
  return f( function (x, ret) { cc(x) }, cc)
}
```

```
callcc(function(cc,rt){...},ret)
↓
kont = function(x,ret){ret(x)}
return ret(1);
```

Exemple d'ús de **callcc(f,c)** en CPS:

```
function fact_cps(n, ret) {
  if (n <= 1) {
    return callcc( function (cc, rt) {
      kont = cc;= function(x,ret){cc(x)}
      return rt(1);
    }, ret);
  }
  else {
    return fact_cps (n-1, function (v) {
      return ret(v*n)
    })
  }
}
```

ret passa a ser rt → retorna
ret(1) que és factorial de 1

(on utilitzem la variable global **kont** i una funció **function id(x) { console.log(x); return x; }** com a continuació inicial)

Continuations

(Continuation() a Rhino)

A Rhino podem capturar la continuació d'una (crida a) funció amb:

new Continuation()

Retorna *la continuació de la crida a la funció dins de la que s'ha invocat new Continuation()*.

Continuations (*Continuation()* a Rhino)

Per exemple, si executem el següent fragment de codi a l'interpret Rhino:

```
function someFunction() {  
    var kont = new Continuation();  
    print("captured: " + kont);  
    return kont;  
}  
  
var k = someFunction();  
if (k instanceof Continuation) {  
    print("k is a continuation");  
    k(200);  
} else {  
    print("k is now a " + typeof(k));  
}  
print(k);
```

```
(fent: java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar  
        org.mozilla.javascript.tools.shell.Main -opt -2 )
```

CAP: PBP: *JavaScript*

Continuations (*Continuation()* a Rhino)

El resultat serà:


```
captured: [object Continuation]
k is a continuation
k is now a number
200
```

```
(fent: java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar
      org.mozilla.javascript.tools.shell.Main -opt -2 )
```

Continuations (Continuation() a Rhino)

Exemple: No Determinisme (l'operador **amb**)

```
function amb(choices) {  
  let cc = current_continuation();  
  if (choices && choices.length > 0) {  
    let choice = choices.shift();  
    fail_stack.push(cc);  
    return choice;  
  } else {  
    return fail();  
  }  
}
```



```
function current_continuation()  
{  
  return new Continuation();  
}
```


Continuations

(Continuation() a Rhino)

Exemple: No Determinisme (l'operador `amb`)

```
function fail() {  
  if (fail_stack.length > 0) {  
    let back_track_point = fail_stack.pop();  
    back_track_point(back_track_point);  
  } else {  
    throw 'back-tracking stack exhausted!';  
  }  
}  
  
function assert(condition) {  
  if (condition) {  
    return true;  
  } else {  
    fail();  
  }  
}
```

Continuations

(Continuation() a Rhino)

Exemple: No Determinisme (l'operador **amb**)

```
function current_continuation() {  
  return new Continuation();  
}  
  
var { amb_reset, fail, amb, assert } =  
  ( function () {  
  
    let fail_stack = [];  
  
    function amb_reset() { fail_stack = []; }  
  
    function fail() { ... }  
  
    function amb(choices) { ... }  
  
    function assert(condition) { ... }  
  
    return { amb_reset: amb_reset, fail: fail, amb: amb, assert: assert }  
  })();
```

Continuations (Continuation() a Rhino)

Exemple senzill:

```
var a = amb([1,2,3,4,5,6,7]);
var b = amb([1,2,3,4,5,6,7]);
var c = amb([1,2,3,4,5,6,7]);

assert( ((c*c) == (a*a + b*b)) );

print(a, ' -- ', b, ' -- ', c);

assert( (b < a) );

print(a, ' -- ', b, ' -- ', c);
```

```
$ java -cp rhino1.7.7.2/lib/rhino-1.7.7.2.jar
  org.mozilla.javascript.tools.shell.Main -opt -2 amb.js
3  --  4  --  5
4  --  3  --  5
4  --  3  --  5
```

Continuations (*Continuation()* a Rhino)

Exemple: La tribu des Kalotan:

Els Kalotan són una tribu desconeguda amb una característica peculiar: Els mascles sempre diuen la veritat. Les femelles no fan mai dues sentències vertaderes consecutives, ni dues sentències falses consecutives.

Un antropòleg, anomenem-lo Worf, ha començat a estudiar els Kalotan, que parlen el llenguatge Kalotan. Un dia, es troba una parella (heterosexual) i el seu fill/filla Kibi. Worf pregunta en Kibi: "Ets un noi?" i Kibi respon en Kalotan, que l'antropòleg no entén.

Worf pregunta els pares (que entenen el català) que què ha dit en Kibi. Un dels pares respon: "Kibi ha dit: 'sóc un noi'". L'altre afegeix: "Kibi és noia. Kibi ha mentit"

Resol el sexe de Kibi i els seus pares.

Continuations (Continuation() a Rhino)

Exemple: La tribu des Kalotan:

```
var progenitor1    = amb(['m', 'f']);
var progenitor2    = amb(['m', 'f']);
var kibi           = amb(['m', 'f']);
var kibi_va_dir    = amb(['m', 'f']);
var kibi_va_mentir = amb([true, false]);
```

```
// els pares han de ser de sexe diferent
assert((progenitor1 != progenitor2));
```

```
// kibi és mascle => no va mentir
assert((      (kibi == 'm') ? (!kibi_va_mentir) : (true)));
```

Continuations (Continuation() a Rhino)

Exemple: La tribu des Kalotan:

```
// el primer progenitor és mascle => kibi va dir que era mascle i
// i com el segon progenitor és femella va mentir en una sentència i va dir
// la veritat en l'altre
assert(((progenitor1 == 'm') ? (kibi_va_dir == 'm' &&
                                XOR((kibi == 'f' && !kibi_va_mentir),
                                     (kibi == 'm' && kibi_va_mentir))) : (true))));

// el primer progenitor és femella => no sabem si el que va dir
// és cert o fals, però sabem que el segon progenitor és mascle i no va mentir.
assert(((progenitor1 == 'f') ? (kibi == 'f' && kibi_va_mentir) : (true))));
```

Continuations (Continuation() a Rhino)

Exemple: La tribu des Kalotan:

```
// kibi no va mentir => o bé va dir que era mascle i ho és, o va dir que era femella i ho és.
assert((    (!kibi_va_mentir) ? (XOR((kibi_va_dir === 'm' && kibi === 'm'),
                                     (kibi_va_dir === 'f' && kibi === 'f')))) : (true)));
```

```
// kibi va mentir => o bé va dir que era mascle i és femella o a l'inrevés
assert((    (kibi_va_mentir) ? (XOR((kibi_va_dir === 'm' && kibi === 'f'),
                                     (kibi_va_dir === 'f' && kibi === 'm')))) : (true)));
```

Finalment, només cal fer:

```
print('Progenitor1 -> ', progenitor1,
      ' Progenitor2 -> ', progenitor2,
      ' Kibi -> ', kibi);
```


Continuations

(Continuation() a Rhino)

Exemple: No Determinisme (l'operador **amb**)

Aquesta implementació no és 100% satisfactòria, ja que l'**amb**, tal i com l'hem implementat, no satisfà algunes propietats que hauria de tenir. Veure el capítol 14 de *Teach Yourself Scheme in Fixnum Days*, (de Dorai Sitaram), cap. 14:

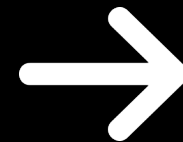
https://ds26gte.github.io/tyscheme/index-Z-H-16.html#node_chap_14

Continuations (Continuation() a Rhino)

Podem serialitzar i des-serialitzar continuacions:

```
function capture(filename) {  
    var k = new Continuation();  
    serialize(k, filename);  
    java.lang.System.exit(0);  
}
```

```
function foo(level) {  
    var now = new java.util.Date();  
    if(level > 5) {  
        print("run the file foo.ser");  
        capture("foo.ser");  
    } else {  
        print("next level");  
        foo(level + 1);  
    }  
    print("restarted("+level+"): " + now)  
}
```



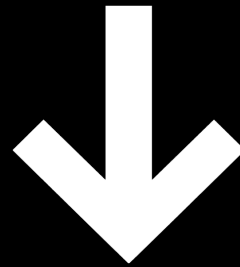
```
next level  
next level  
next level  
next level  
next level  
run the file foo.ser
```

```
foo(1);
```

Continuations (`Continuation()` a Rhino)

Podem serialitzar i des-serialitzar continuacions:

```
var k = deserialize("foo.ser");  
k();
```



```
restarted(6): Wed Dec 06 12:05:23 CET 2017  
restarted(5): Wed Dec 06 12:05:23 CET 2017  
restarted(4): Wed Dec 06 12:05:23 CET 2017  
restarted(3): Wed Dec 06 12:05:23 CET 2017  
restarted(2): Wed Dec 06 12:05:23 CET 2017  
restarted(1): Wed Dec 06 12:05:23 CET 2017
```