# Final CAP

## Curs 2019-20 (14/I/2020) Durada: 2 hores

**1.- (1 punt)** En què es diferencien les quatre maneres diferents d'invocar funcions en Javascript? Explica-ho resumidament, només emfatitzant els aspectes més importants.

Solució:
Està explicat a les transparències del tema 3, planes 32-39 (d'aquest curs).

**2.- (1 punt)** Implementeu en Javascript (Rhino) una funció **`callcc(f)`** que funcioni com l'estructura de control que ja coneixeu de Pharo, fent servir, naturalment, la funció **`Continuation()`** de Javascript/Rhino.

Solució:
Recordem com funciona **`callcc`**: Aquesta funció s'invoca amb una funció com a paràmetre, que s'invoca amb la continuació corresponent a la crida a **`callcc`** com a paràmetre:

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

**3.- (1.5 punts)** Sabem que (quasi) tot objecte en Javascript té un prototipus (un altre objecte al que fa referència). I sabem que tot objecte-funció (objectes invocables) conté una propietat anomenada **`prototype`**. Aleshores, respón a aquestes qüestions:

a) En general, el prototipus d'un objecte-funció i el **`prototype`** d'aquest objecte-funció són el mateix objecte?
b) Hi ha cap excepció a la regla general?
c) Per a què serveix el **`prototype`** d'un objecte-funció?

Solució:
a) En general són objectes diferents
b) Però hi ha el cas excepcional de l'objecte **`Function`**, que té **`Function.prototype`** com a prototipus.
c) Serveix per determinar qui serà el prototipus dels objectes nous creats quan s'invoqui l'objecte-funció amb la paraula clau **`new`**.

**4.- (2 punts)** Tenim una funció recursiva final per invertir un *array*. Si **`arr`** és l'*array* que volem invertir, s'ha de cridar **`reverse(arr, [])`**:

```
function reverse (arr, res) {
    if (arr.length === 0) {
        return res
    } else {
        let [car, ...cdr] = arr;
        res.unshift(car);
        return reverse(cdr,res);
```

```
    }
}
```

Aleshores: **`reverse ([1,2,3,4,5], []) => [5,4,3,2,1]`**

Sabem que si fem servir Node.js, que *no* fa TCO tindrem problemes amb la pila. Si fem **`reverse([...Array(10000).keys()],[])`** obtindrem un error **`RangeError: Maximum call stack size exceeded`**. Apliqueu la técnica del *trampolining* per obtenir una versió de **`reverse`** que no tingui problemes amb la mida de la pila.

Solució:
Si tenim

```
function trampoline (fun) {
    while (typeof fun == 'function') {
            fun = fun();
    }
    return fun;
};
```

podem fer, seguint l'esquema general que vam veure a classe:

```
const reverse_tramp = (function () {
    function __f(a, r) {
        if (a.length > 0) {
            return function () {
                let [car, ...cdr] = a;
                r.unshift(car);
                return __f(cdr,r);
            };
        }
        return r;
    };

    return function (arr, res) {
        return trampoline(__f(arr,res));
    }
})();
```

**5.- (2 punts)** Utilitzant la resposta a la pregunta 2 d'aquest examen (la definició de **`callcc`** en Javascript/Rhino), digueu amb detall què fa aquesta funció.

```
function arg_fc() {
    return callcc(function(k) {
        k( function(x) {
            k( function(y) {
                return x;  })})});
};
```

Per exemple, mireu d'executar això (però podeu fer més proves per mirar d'esbrinar què fa aquest programa):

```
let f = arg_fc();
f(true);
print(f(false));
```

Solució:
Aquest programa, en ser executat per primer cop, **`arg_fc()`**, retorna la funció:

```
function(x) {
  k( function(y) {
       return x;
     })
}
```

que és assignada a **`f`**. En fer **`f(true)`** s'executa aquesta funció, que torna al punt on vam retornar una funció que va ser assignada a **`f`**. Aquest cop, la funció assignada a **`f`** és:

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

que, com la primera invocació ha estat amb paràmetre **`true`**, **`x`** té com a valor **`true`**. Aquesta funció sempre retornarà **`true`**, no importa quin sigui el seu argument.

Així doncs, bastant obvi a partir d'aquest experiment, hem aconseguit una funció que, a partir de la segona invocació, sempre retornarà l'argument de la primera invocació.

**6.- (2.5 punts)** Recordeu que el patró **Decorador** és útil per resoldre el problema d'un objecte que necessita ser modificat, afegint-li *propietats* (no confondre amb les "propietats" de Javascript) en temps d'execució. Ja vam veure un exemple de patró Decorador a classe, l'exemple amb la funció **`Sale`**. Ara, com que segur que sentiu nostàlgia de l'assignatura de PROP, us demano que utilitzeu el patró Decorador per implementar el famós exemple del Café, és a dir, el programa per a que el següent codi funcioni correctament:

```
var c = new Cafe();
c = c.decorate('llet');
c = c.decorate('xocolata');
c = c.decorate('crema');
console.log(c.getPrice());
console.log(c.getIngredients());
```

que té com a sortida:

```
2.75
Cafe, llet, xocolata, crema
```

*Pista*: hauríeu de prendre com a punt de partida l'exemple de **`Sale`** que vam veure a classe, tenint en compte que la funció **`Cafe.prototype.decorate`** ha de ser *exactament* la mateixa que **`Sale.prototype.decorate`** (sense cap canvi).

Solució:

```
function Cafe() {
    this.price = 1;
    this.ingredients = "Cafe";
}

Cafe.prototype.getPrice = function () {
    return this.price;
};

Cafe.prototype.getIngredients = function () {
    return this.ingredients;
};
```

`Cafe.prototype.decorate =` (la ***MATEIXA*** funció de l'exemple que vam veure a classe amb l'exemple de **`Sale`**, sense ***CAP*** canvi)

```
// Els objecte decoradors s'implementaran com a propietats d'una
// propietat del constructor

Cafe.decorators = {};

Cafe.decorators.llet = {
    getPrice: function () {
            var price = this.uber.getPrice();
            price += 0.5; // suposem que afegir llet val 0.5 euros
            return price;
    },

    getIngredients: function () {
        var ing = this.uber.getIngredients();
        return ing + ", llet"
    }
};

Cafe.decorators.xocolata = {
    getPrice: function () {
            var price = this.uber.getPrice();
            price += 0.75; // suposem que afegir xocolata val 0.75 euros
            return price;
    },

    getIngredients: function () {
        var ing = this.uber.getIngredients();
        return ing + ", xocolata"
    }
};

Cafe.decorators.crema = {
```

```
    getPrice: function () {
            var price = this.uber.getPrice();
            price += 0.5; // suposem que afegir crema val 0.5 euros
            return price;
    },

    getIngredients: function () {
        var ing = this.uber.getIngredients();
        return ing + ", crema"
    }
};
```

Sembla molt de codi, però els decoradors són essencialment iguals, i la resta és com la de l'exemple de **`Sale`** que vam veure a classe.