

Conceptes Avançats de Programació

# Programació basada en Prototipus

***Prototype-Based  
Programming***

*J.Noble, A.Taivalsaari &  
I.Moore (eds.)  
Springer 1999*

***JavaScript: The  
Definitive Guide  
(7th edition)***

*David Flanagan  
O'Reilly 2020*



# CAP: PBP, conceptes bàsics



*Origen:*

Lieberman, H.

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

OOPSLA'86



# CAP: PBP, conceptes bàsics



Diferents llenguatges apareixen durant els  
'80 i '90

Self  
Object Lisp  
Garnet  
Amulet  
Agora  
Moosttrap

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*)



# CAP: PBP, conceptes bàsics



## *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*.



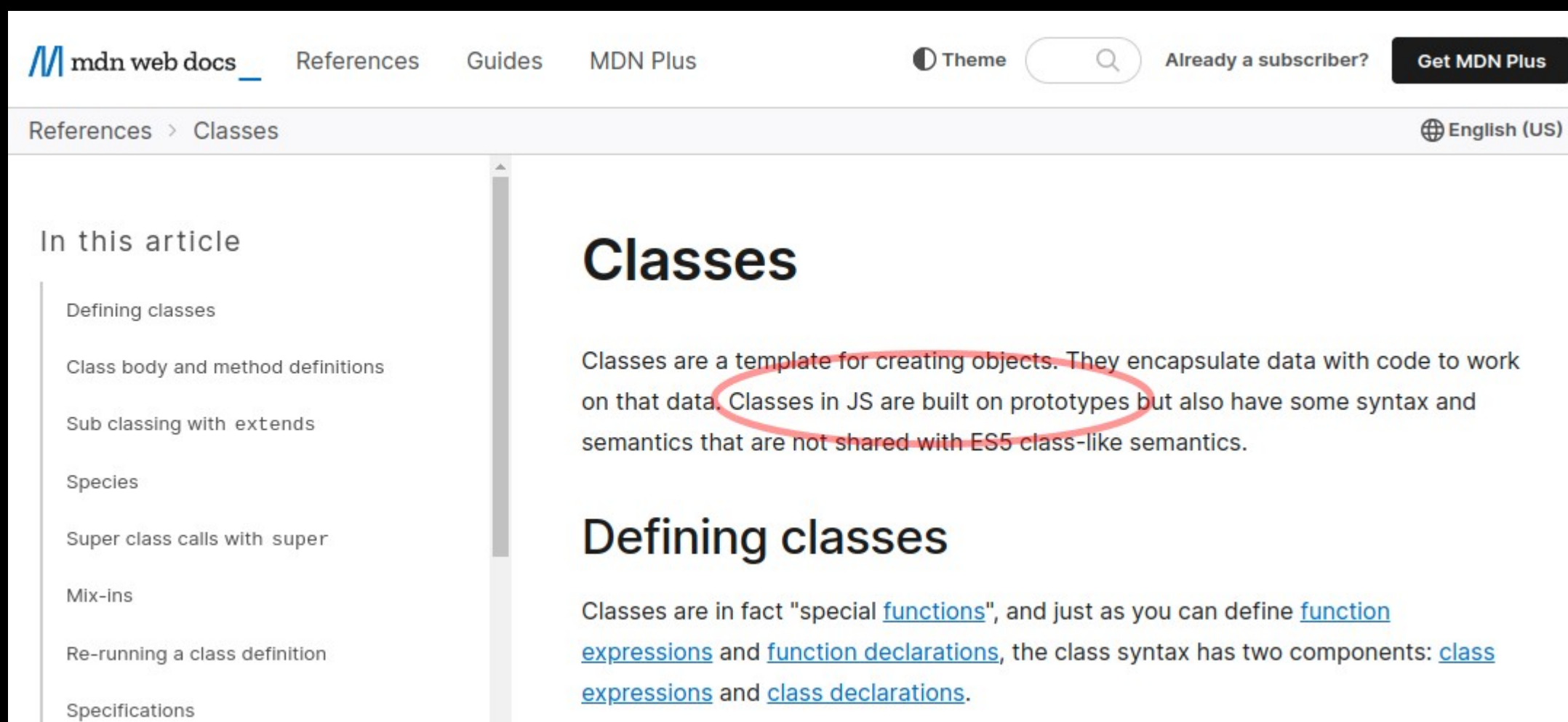
# CAP: PBP: *Javascript*



***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 i/o PROP) són implementats en Javascript (*que quedi clar, doncs, que no volem ensenyar Javascript per sí mateix*)

*“Però si Javascript sí que té classes!!”*



The screenshot shows the MDN Web Docs page for 'Classes'. The top navigation bar includes 'mdn web docs', 'References', 'Guides', 'MDN Plus', a 'Theme' toggle, a search bar, and a 'Get MDN Plus' button. The breadcrumb trail is 'References > Classes'. The left sidebar, titled 'In this article', lists sections: 'Defining classes', 'Class body and method definitions', 'Sub classing with extends', 'Species', 'Super class calls with super', 'Mix-ins', 'Re-running a class definition', and 'Specifications'. The main content area has a heading 'Classes' followed by a paragraph: 'Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are not shared with ES5 class-like semantics.' The word 'template' is circled in red. Below this is a section 'Defining classes' with a paragraph: 'Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.'

*Les ignorarem!*

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>





## *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?



# CAP: PBP: *Objectes i Prototipus*



**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 ':').

```
let 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";
```



# CAP: PBP: *Creació d'Objectes*



**Creació d'Objectes:** Objectes literals.

Modismes per proporcionar valors per defecte:

```
let 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:

```
for (let 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;  
}  
  
let 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(let x = Math.ceil(this.from); x <= this.to; x++) {  
    f(x);  
  }  
};
```

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

Cal tenir en compte:

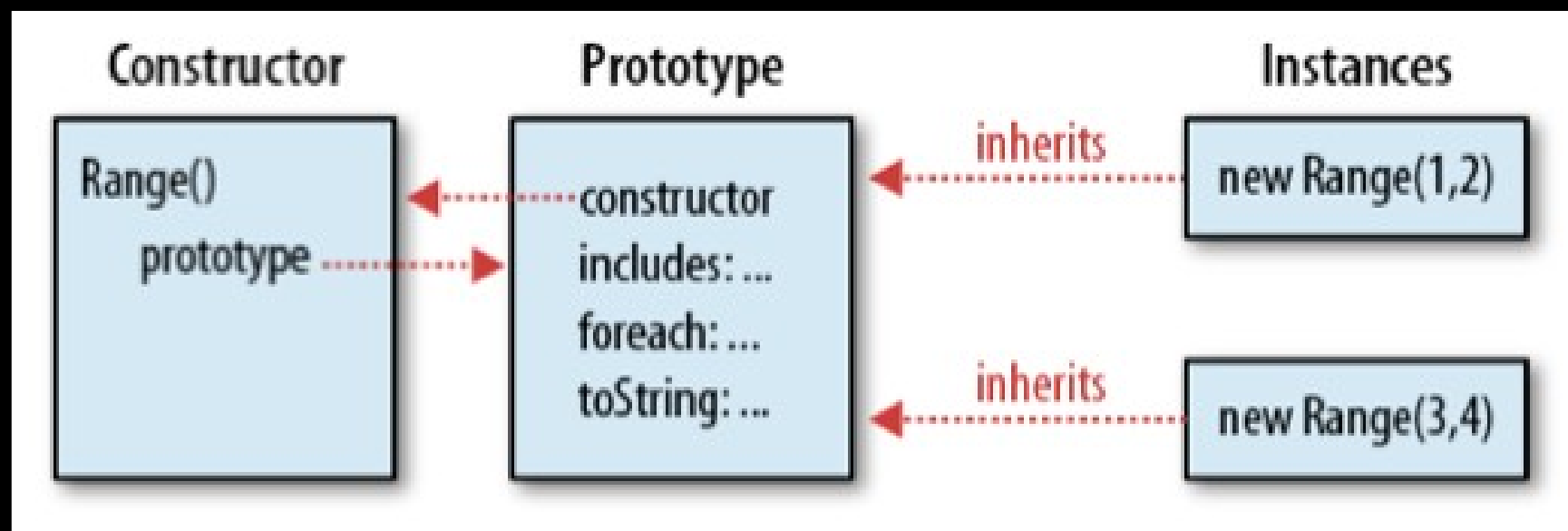
- El prototipus de tota funció és **Function.prototype**
- Tota funció a Javascript té una propietat **prototype**
- L'objecte referenciat per aquesta propietat de les funcions té una propietat, **constructor**, que referencia la funció.

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

```
let 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*  
(7th. ed.),  
David Flanagan,  
2020,  
p. 228

```
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(let 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, 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 (a partir d'ECMAScript 5).

Podem aprofitar aquest codi 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  
}
```



# CAP: PBP: *Objectes i Prototipus*



## 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*.



# CAP: PBP: *Objectes i Prototipus*



## Prototipus (altre cop):

Així, l'operador **instanceof** no mira si un objecte ha estat creat amb un determinat constructor, mira 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



# CAP: PBP: *Objectes i Prototipus*



**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 = ...
```



# CAP: PBP: *Objectes i Prototipus*



**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**.

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



## Prototipus (seguim):

El prototipus d'un objecte és un *atribut*, no una propietat. A partir d'ECMAScript 5 podem demanar pel prototipus d'un objecte amb

**`Object.getPrototypeOf(...)`:**

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



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

**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  
let bar = function (...) { ... }    // expressió  
let baz = function baz(...) { ... } // expressió amb nom
```

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

Sota l'*strict mode* hi ha restriccions en la manera de definir funcions.

**Funcions**: Les funcions a JavaScript són *objectes*.

L'abast de les variables declarades amb **let** o **const** és un **abast de bloc**, o **abast lèxic** (amb el que probablement ja esteu familiaritzats)

```
let x = 1;

if (x === 1) {
  let x = 2;
  console.log(x);
  // expected output: 2
}

console.log(x);
// expected output: 1
```

Al tanto amb la *temporal dead zone*!

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal\\_dead\\_zone\\_tdz](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz)

**Funcions:** Les funcions a JavaScript són *objectes*.

Al tanto amb la *temporal dead zone*!

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal\\_dead\\_zone\\_tdz](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz)

```
#include <iostream>
using namespace std;
```

```
int x = 1;
```

```
void prova_tdz() {
    cout << x << endl;
    int x = 2;
    cout << x << endl;
}
```

```
int main() {
    prova_tdz();
}
```

```
$ g++ -o tdz.exe tdz.cpp
```

```
$ ./tdz.exe
```

```
1
```

```
2
```

***Funcions***: Les funcions a JavaScript són *objectes*.

Al tanto amb la *temporal dead zone*!

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal\\_dead\\_zone\\_tdz](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz)

```
#include <iostream>
using namespace std;
```

```
int x = 1;
```

```
void prova_tdz() {
    cout << x << endl;
    int x = 2;
    cout << x << endl;
}
```

```
int main() {
    prova_tdz();
}
```

```
$ g++ -o tdz.exe tdz.cpp
$ ./tdz.exe
1
2
```

```
let x = 1;
```

```
function prova_tdz() {
    console.log(x)
    let x = 2
    console.log(x)
}
```

```
prova_tdz()
```

```
$ node tdz.js
~/tdz.js:4
    console.log(x)
                ^
```

```
ReferenceError: Cannot access 'x'
before initialization
```

**Funcions**: Les funcions a JavaScript són *objectes*.

L'abast de les variables declarades amb **var**, o bé és **global**, o bé, igual que els noms de funcions locals, és un **abast de funció** (*function scope*): **hoisting**.

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

```
function hoistMe() {  
  console.log(typeof foo); // ???  
  console.log(typeof bar); // ???  
  console.log(foo());      // ???  
  console.log(bar());      // ???  
  function foo() { return 'local foo' };  
  var bar = function() { return 'local bar' };  
}
```

```
hoistMe();
```

(alguns diuen que l'ús de **var** està *deprecated*)



**Funcions:** Les funcions a JavaScript són *objectes*.

En realitat les funcions són **Closures**:

## Funció + context lèxic

*(en el moment de la creació de la funció)*

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

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

**Funcions:** Les funcions a JavaScript són *objectes*.

En realitat les funcions són **Closures**

```
> counter.increment(8)
undefined
> counter.getValue()
8
> counter.increment(-4)
undefined
> counter.getValue()
4
> counter.increment(2345)
undefined
> counter.getValue()
2349
> counter.increment(-2349)
undefined
> counter.getValue()
0
>
```

**Funcions:** Les funcions a JavaScript són *objectes*.

En realitat les funcions són **Closures**:

## Funció + context lèxic

*(en el moment de la creació de la funció)*

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

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

**Funcions:** Les funcions a JavaScript són *objectes*.

En realitat les funcions són **Closures**

```
> counter1 = make_counter()
{ increment: [Function: increment], getValue: [Function: getValue] }
> counter1.increment(7)
undefined
> counter1.increment(15)
undefined
> counter1.getValue()
22
> counter2 = make_counter()
{ increment: [Function: increment], getValue: [Function: getValue] }
> counter2.getValue()
0
> counter2.increment(42)
undefined
> counter2.increment(-21)
undefined
> counter2.getValue()
21
> counter1.getValue()
22
>
```

**Closures:** Funcions amb el seu context lèxic

Exemple: La pila  
(no vàlid en *strict mode*)

```
let Pila = function() {  
  let index = 0;  
  let 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);  
    }  
  };  
};
```

**Closures**: Funcions amb el seu context lèxic

Exemple: La pila  
(no vàlid en *strict mode*)

```
> let pila1 = Pila();  
undefined  
> pila1.push(1);  
undefined  
> pila1.push('hola');  
undefined  
> pila1.push(1.5);  
undefined  
> pila1.push('adeu');  
undefined  
> console.log(pila1.top());  
adeu  
undefined  
> pila1.pop();  
undefined  
> console.log(pila1.top());  
1.5  
undefined
```

```
> pila1.pop();  
undefined  
> console.log(pila1.top());  
hola  
undefined  
> pila1.pop();  
undefined  
> console.log(pila1.top());  
1  
undefined  
> pila1.pop();  
undefined  
> console.log(pila1.empty());  
true  
undefined
```



# CAP: PBP: *Invocació de Funcions*



**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.

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





# CAP: PBP: *Invocació de Funcions*



***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



# CAP: PBP: *Invocació de Funcions*



***Funcions***: La invocació com a Funció

Exemple:

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



***Funcions***: La invocació com a Funció

Exemple:

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

## *Funcions*: La invocació com a Funció

Exercici:

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

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



# CAP: PBP: *Invocació de Funcions*



**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 ja hem vist.

**Funcions:** La invocació com a '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**.

```
let Obj = function(n) { this.value = n; };  
Obj.prototype.double = function () {  
    this.value = 2 * this.value;  
};  
let foo = new Obj(7);  
foo.double(); // ==> foo és { value: 14 }  
  
let bar = { value : 7 };  
Obj.prototype.double.apply(bar); // ==> bar és { value: 14 }
```



# CAP: PBP: *Invocació de Funcions*



**Funcions**: La invocació com a '**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**.

**Function.prototype.call():**

**<funcio>.call(objecte que serà this, arg1, arg2, ...)**

**Function.prototype.apply():**

**<funcio>.apply(objecte que serà this, [arg1, arg2, ...])**



## *Funcions*: Els arguments

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*: Els arguments

En el Javascript més modern tenim la possibilitat de fer servir la *rest parameter syntax*, que consisteix en fer explícit un paràmetre que reculli tots els arguments que no han estat considerats a la definició de la funció.

Per exemple:

```
function sum(...theArgs) {  
    let total = 0;  
    for (const arg of theArgs) {  
        total += arg;  
    }  
    return total;  
}
```

Aquest paràmetre sí és un veritable array



## *Funcions*: Els arguments

Hi ha tres diferències principals entre els *rest parameters* i l'objecte **arguments**:

- els *rest parameters* són només aquells als que no s'ha donat un nom separat (és a dir, definits formalment a la definició de la funció), mentre que l'objecte **arguments** conté tots els arguments passats a la funció
- l'objecte **arguments** no és un **array** real, mentre que els *rest parameters* sí que són instàncies d'**array**, és a dir, mètodes com **sort**, **map**, **forEach** o **pop** es poden aplicar directament
- l'objecte **arguments** té funcionalitats addicionals específiques, com la propietat **callee**.



# CAP: PBP: *Funcions*



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

**Funcions:** Exercici: **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: *Funcions*



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



# CAP: PBP: *Patrons*



## *Patrons de disseny*

Veurem el detall dels patrons:

*Singleton* (creational)

*Factory* (creational)

*Decorator* (structural)

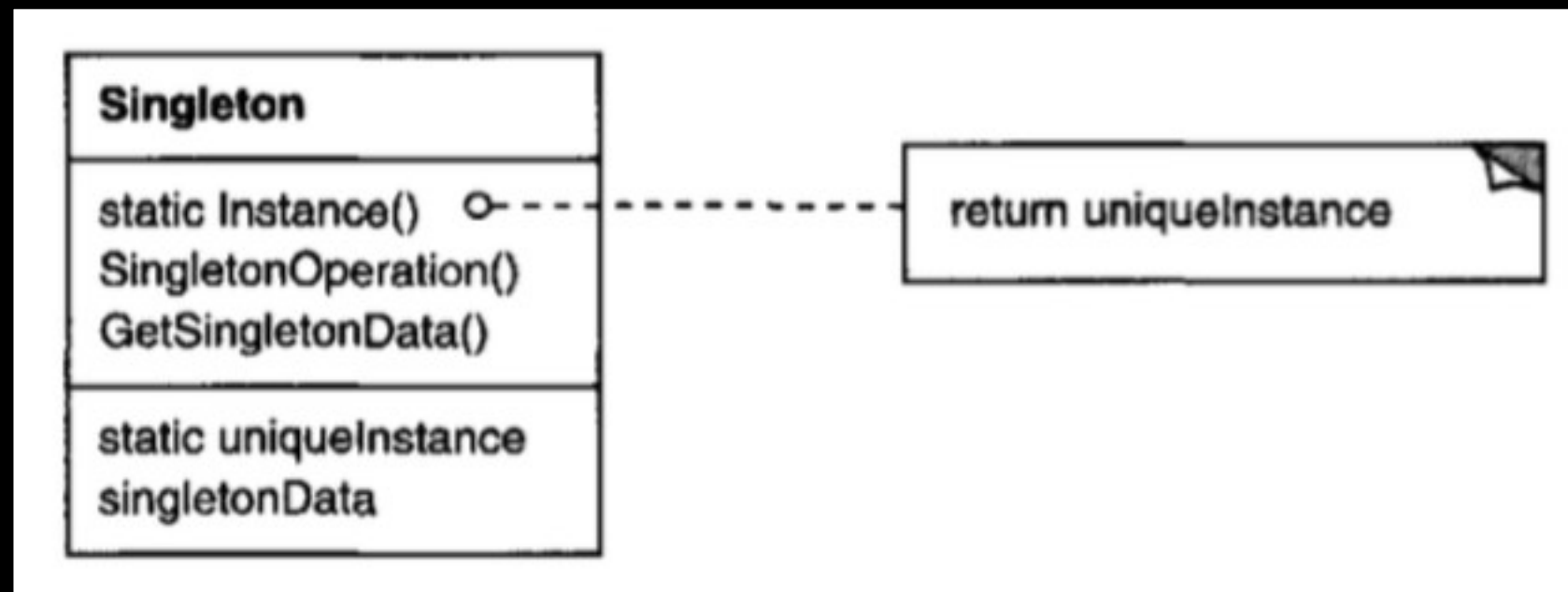
*Observer* (behavioral)

***Design Patterns: Elements of  
Reusable Object-Oriented Software***  
Erich Gamma, Richard Helm, Ralph  
Johnson, and John Vlissides  
Addison Wesley 1994

***JavaScript Patterns***  
Stoyan Stefanov  
O'Reilly 2010

## *Patrons de disseny*

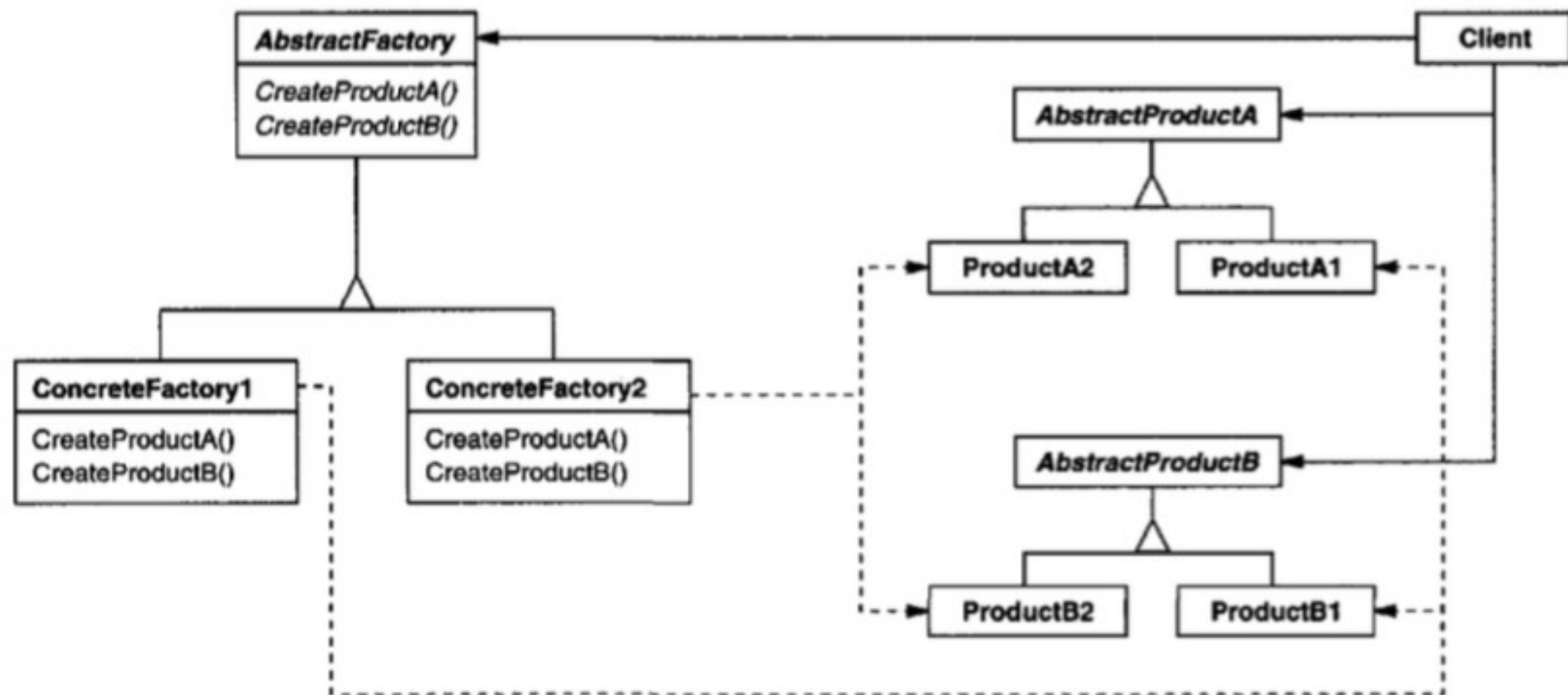
### *Singleton*





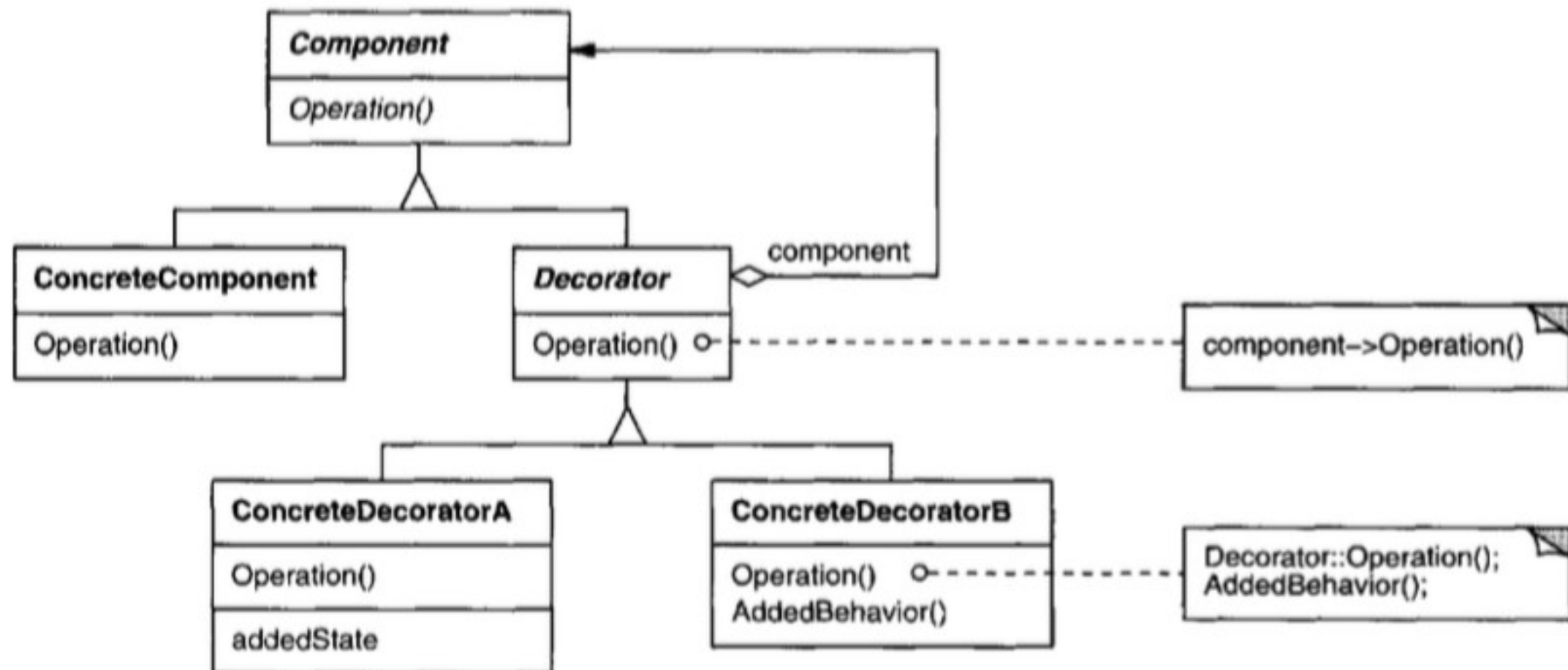
## *Patrons de disseny*

### *Factory*



## *Patrons de disseny*

### *Decorator*



## *Patrons de disseny*

### *Observer*

