JS

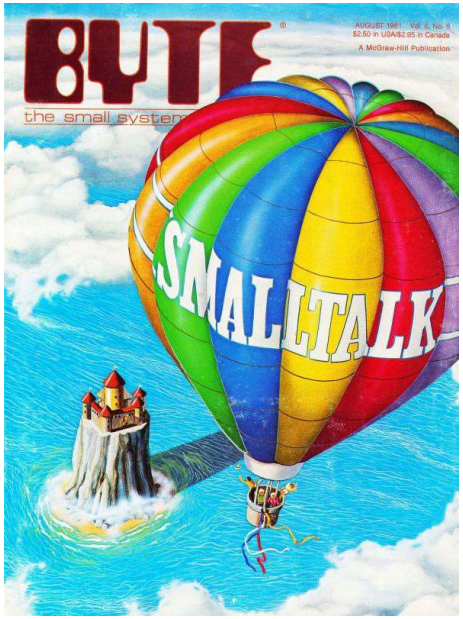
JavaScript

JavaScript (JS) is a lightweight interpreted or JIT-compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles. Read more about JavaScript.

ES

ECMAScript

The standard for JavaScript is ECMAScript. As of 2012, all modern browsers fully support ECMAScript 5.1. Older browsers support at least ECMAScript 3. On June 17, 2015, ECMA International published the sixth major version of ECMAScript, which is officially called ECMAScript 2015, and was initially referred to as ECMAScript 6 or ES6. Since then, ECMAScript standards are on yearly release cycles. This documentation refers to the latest draft version, which is currently ECMAScript 2018.



JavaScript y Smalltalk

Dos lenguajes OO bien particulares

Objetivos

- Presentar los aspectos mas particulares de dos lenguajes que se diferencian del resto
- JavaScript (ECMAScript)
  - Su naturaleza basada en prototipos
- Smalltalk
  - OO de pies a cabeza (escrito en Smalltalk)
  - Su ambiente que invita a un enfoque exploratorio de desarrollo
  - Fuente de muchas de las ideas que hoy vemos en otros lenguajes y ambientes

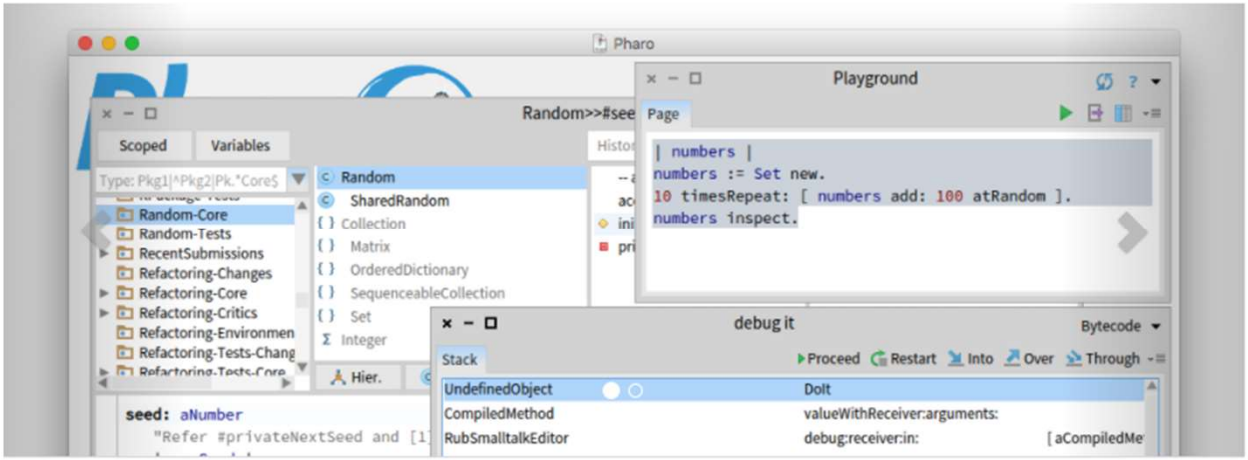
Smalltalk

- Lenguaje OO puro – todo es un objeto (¡ incluso las clases !)
- Tipado dinámicamente
- Propone una estrategia exploratoria (construccionista) al desarrollo de software
- El ambiente es tan importante como el lenguaje
  - Está implementado en Smalltalk
  - Ricas librerías de clases (fuentes de inspiración y ejemplos)
  - Todo su código fuente disponible y modificable
  - Tiene su propio compilador, debugger, editor, inspector, perfilador, etc.
  - Es extensible
- Sintaxis minimalista (con sustento en su foco educativo)
- Fuente de inspiración de casi todo lo que vino después (en OO)

Pharo

The immersive programming experience

Pharo is a pure object-oriented programming language *and* a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).

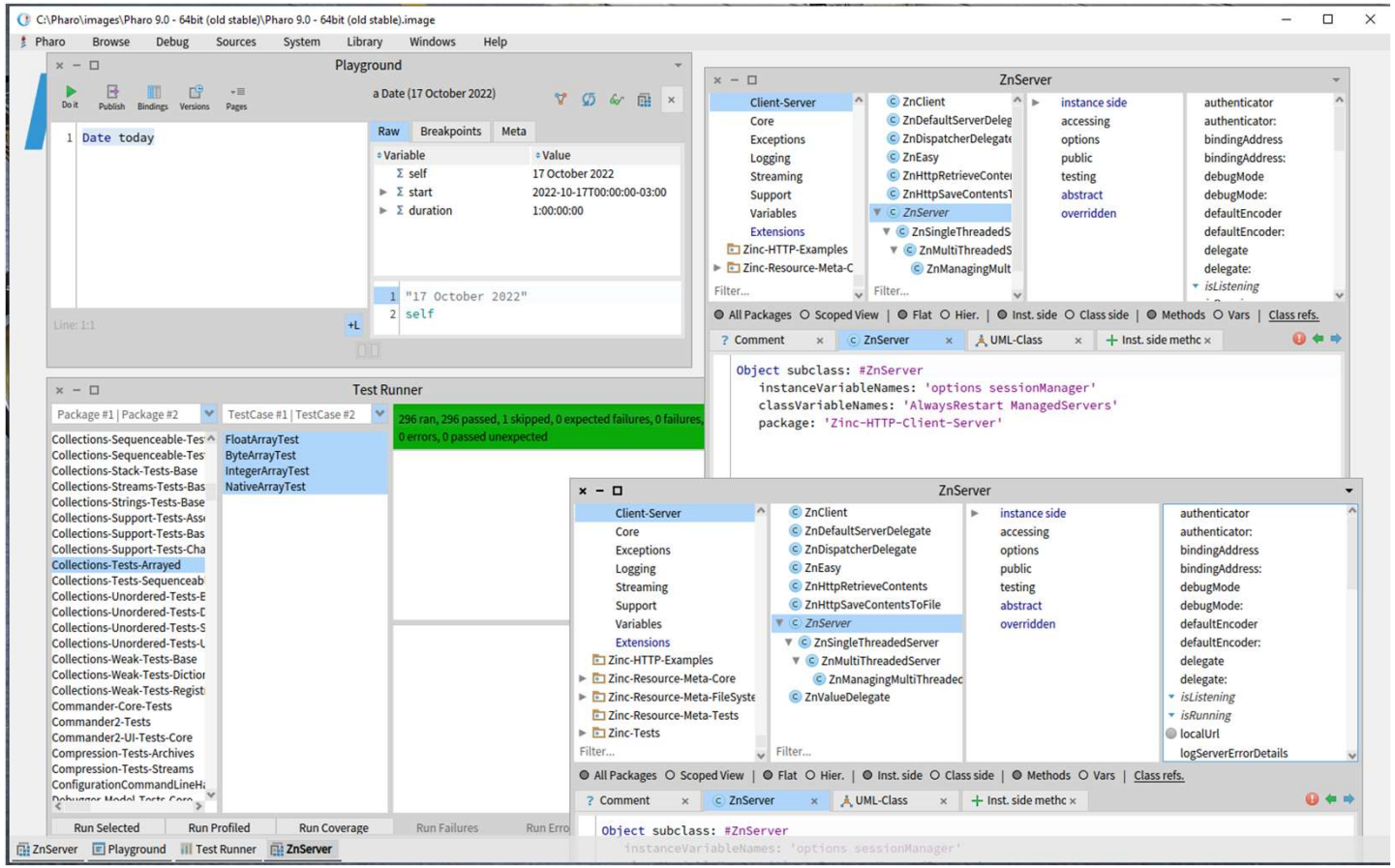


Discover | Download | Learn

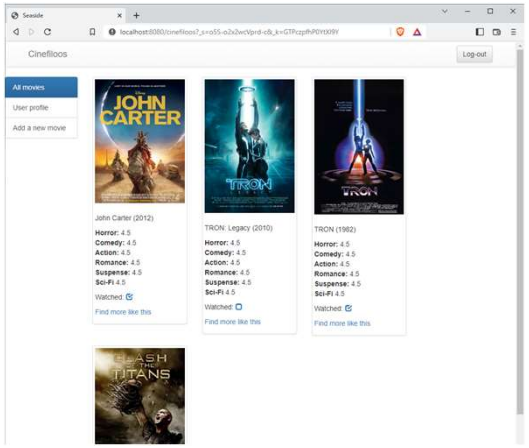
```
"Asignación y terminación"
dificil := false .

"Mensajes unarios; solo el objeto receptor, sin parámetros"
3 squared .
'Hola' reversed .

"Mensajes binarios; objeto receptor y un parámetro"

'Hola', ' Manola' .
1 @ 10 .

"Mensajes de palabra clave; objeto receptor y n parámetros"

'Manola' includesSubstring: 'ola' .
'Hola' copyWithoutAll: 'ol'.
3 between: 1 and: 2 .
```

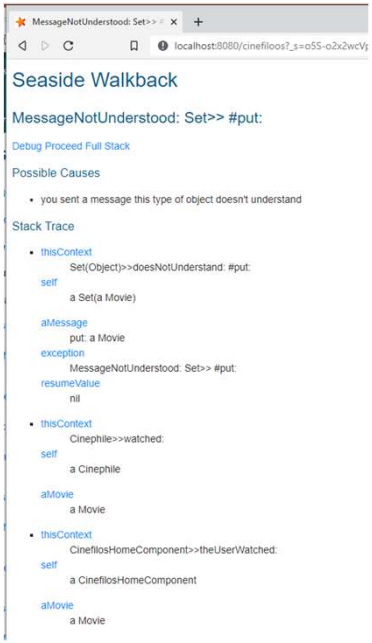
```
"Definición de clases e instanciación"

Object subclass: #Persona
  instanceVariableNames: 'nombre apellido edad'
  classVariableNames: ''
  package: 'OO1' .

Persona compile: 'initialize
    edad := 0.' classified: 'initialization' .

Persona compile: 'nombre: elNombre apellido: elApellido
  nombre := elNombre.
  apellido := elApellido.' classified: 'accessing' .

Persona compile: 'getNombreCompleto
  ^ nombre , '' '' , apellido.' classified: 'accessing' .

alguien := Persona new.
alguien nombre: 'Juan Carlos' apellido: 'Batman' .
```

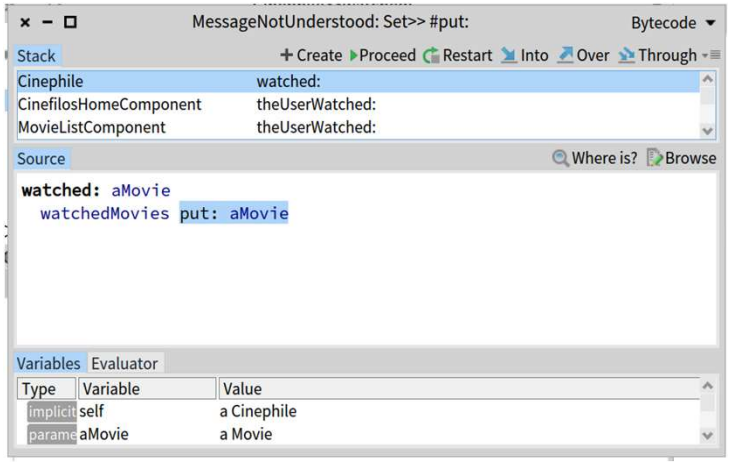
Smalltalk: el ambiente



Hot repair … (objetos vivos, siempre)







Cinefiloos

- Para explorar la implementación de referencia de Cinefiloos, evalúen la siguiente expresión en un playground de Pharo

```
Metacello new
    baseline: 'Cinefiloos';
    repository: 'bitbucket://lifia-oop/practicas-objetos-1’;
    onConflictUseLoaded;
    load.
```

Clausuras (Closures)

```
nombre := 'Juan Gomez'.
aBlockClosure :=  [ Transcript show: 'Hola ', nombre; cr ].
aBlockClosure value.

button :=   PluggableButtonMorph  new .
button label: 'Click me'.
button position: 400@10.
button actionBlock: aBlockClosure.
button openInWorld .
```

Qué devuelven ...

```
[ 1 < 10 ] value.
```

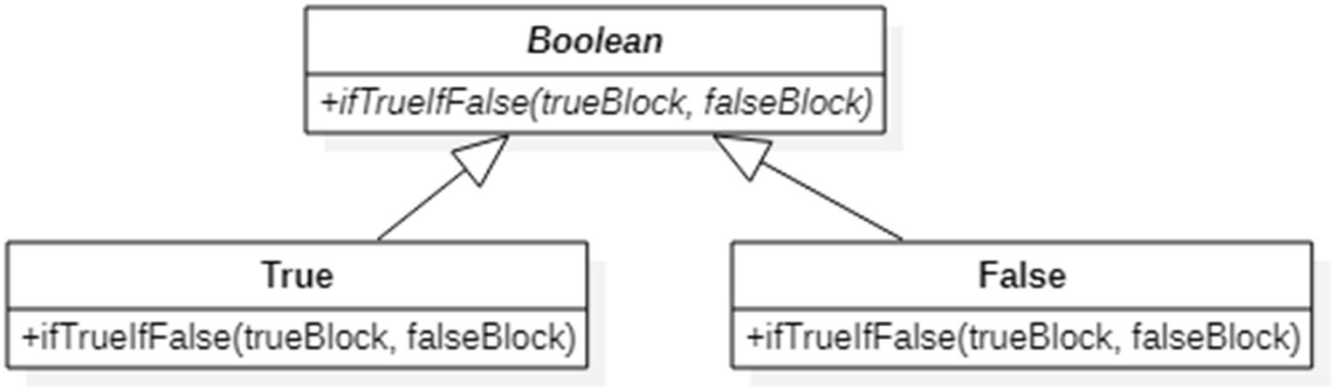
```
aBlockClosure := [ | temp |
                   temp := OrderedCollection new.
                   temp add: (Random new next).
                   temp  ].
aBlockClosure value.
aBlockClosure value.
```

Con parámetros

```
aBlockClosure := [ :algo | algo size ].
aBlockClosure value: 'hola'.
aBlockClosure value: Set new.


aBlockClosure := [ :a :b | a < b ].
aBlockClosure value: 1 value: 2.
```

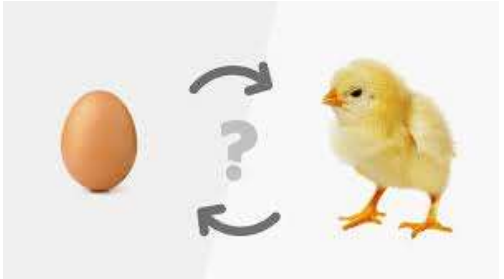
IF con objetos (puro polimorfismo)

```
(a < 100)
      ifTrue:  [Transcript shown: 'a es MENOR a 100']
      ifFalse: [Transcript shown: 'a es MAYOR a 100'  ]
```



Smalltalk – las clases son objetos ...

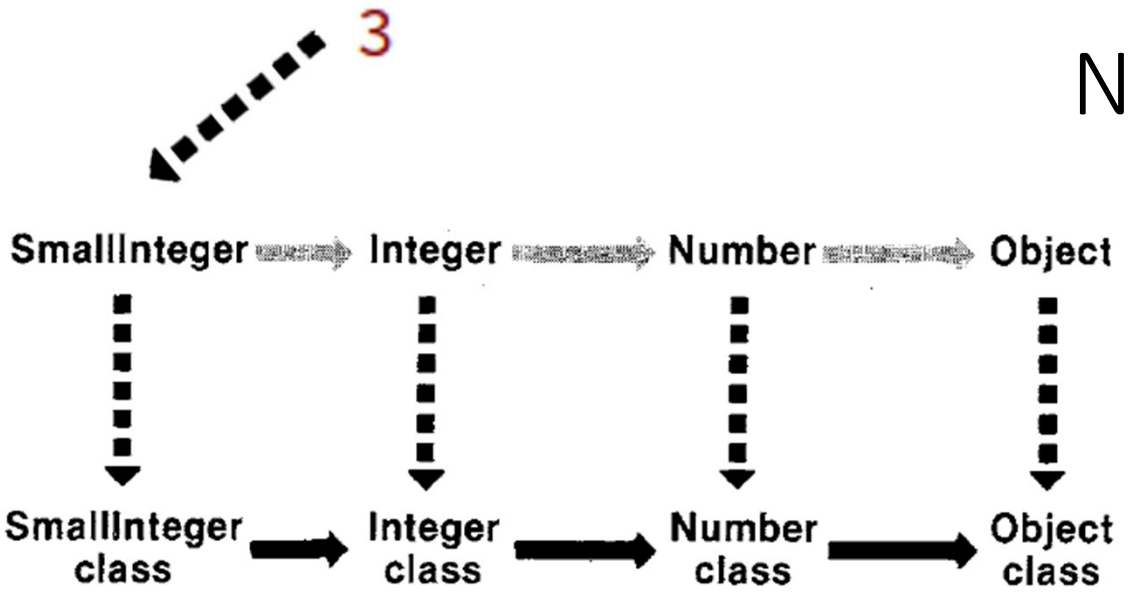
- Smalltalk hay dos tipos de objetos: los que pueden crear instancias (de si mismos), y los que no.
  - A los primeros les llamamos clases.
- Si las clases entienden mensajes, tienen su propio conocimiento y comportamiento
  - ¿Dónde se especifica su estructura y comportamiento? ¿En otra clase?
- Esto (El metamodelo de Smalltak) en uno de sus aspectos mas interesantes y desafiantes



Lo básico

- Hay objetos capaces de crear instancias y describir su estructura comportamiento: las clases (p.e., SmallInteger).
- Todo objeto es instancia de una clase (p.e., 1 de SmallInteger)
- Las clases son instancias de una clase también (su metaclase).
  - Por cada clase hay una metaclase (se crean juntas).
  - SmallInteger es instancia de “SmallInteger class”
- Las metaclases son instancias de la clase Metaclass
  - “SmallInteger class” es instancia de Metaclass
- …

Nivel básico



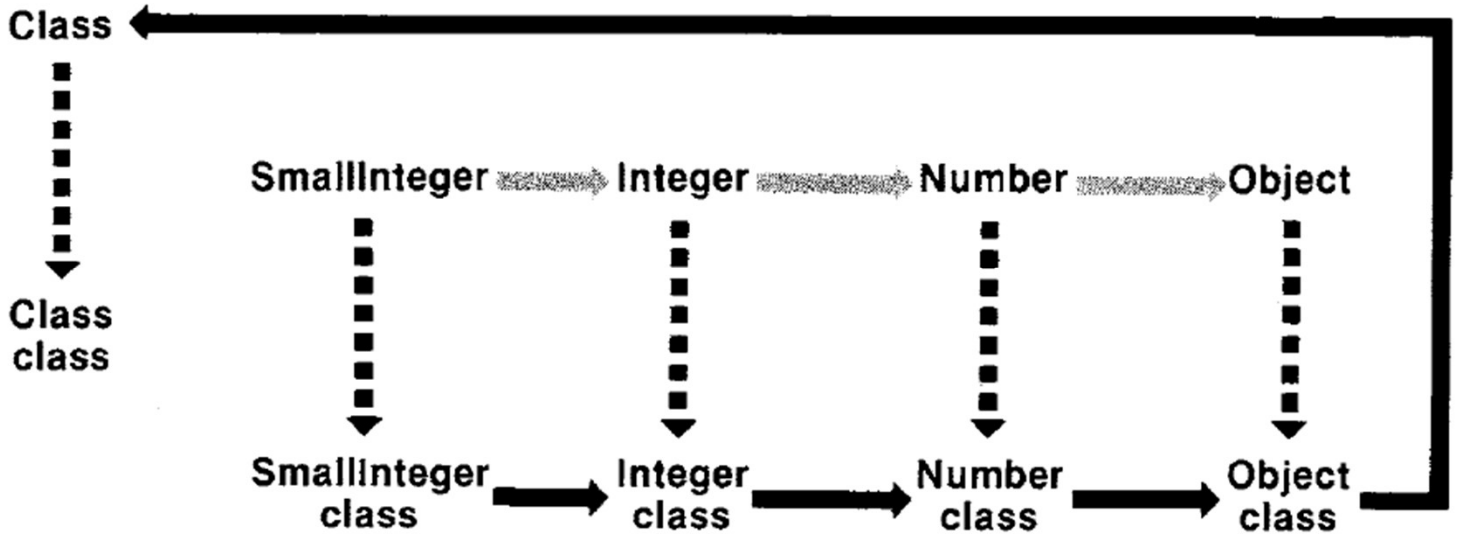

```
3 factorial .
```

Mensajes de “instancia”

```
SmallInteger maxVal .
```

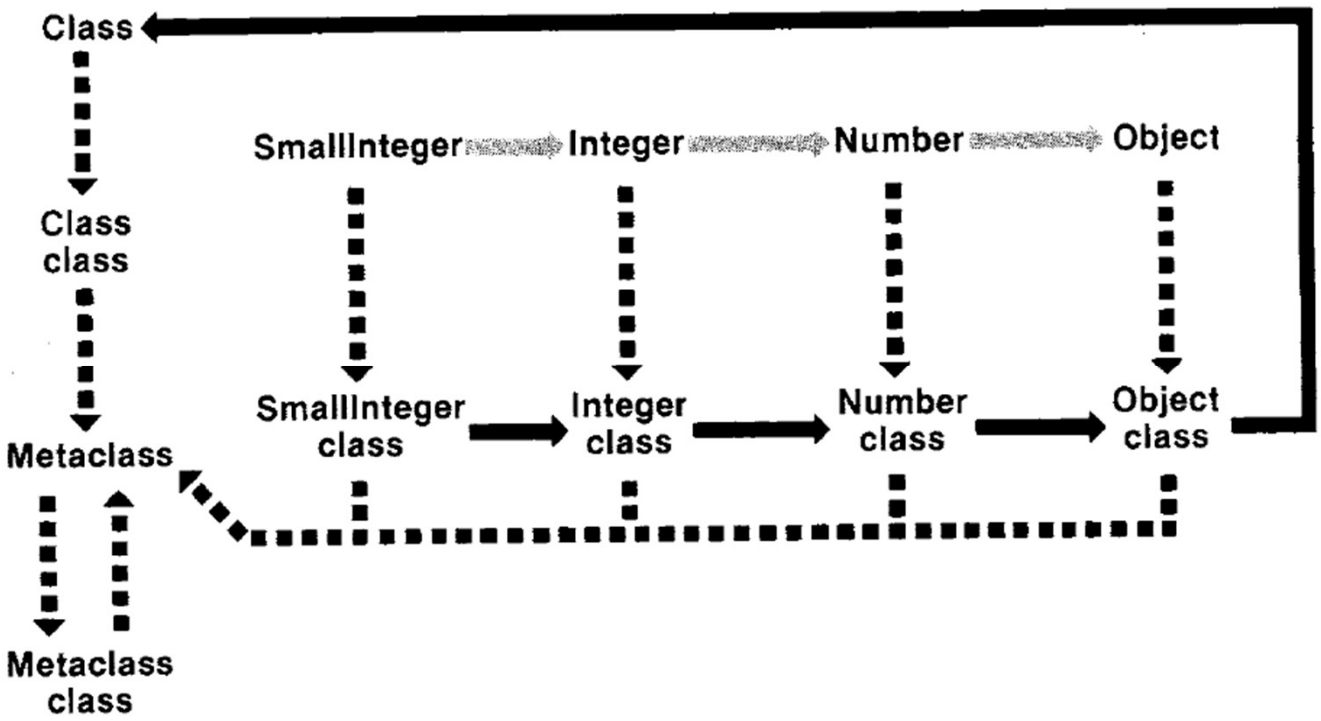
Mensajes de “clase”

Nivel avanzado

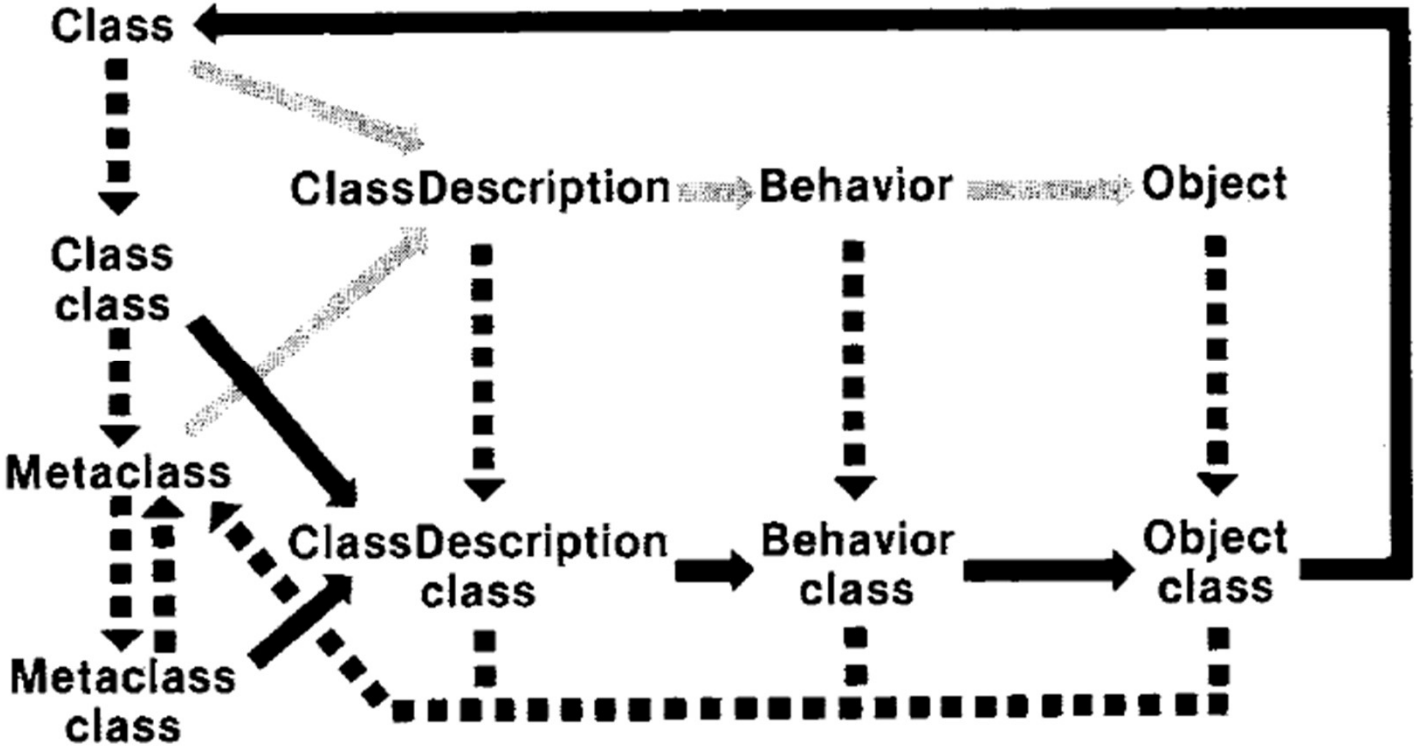



```
Integer subclass: #SmallInteger
```

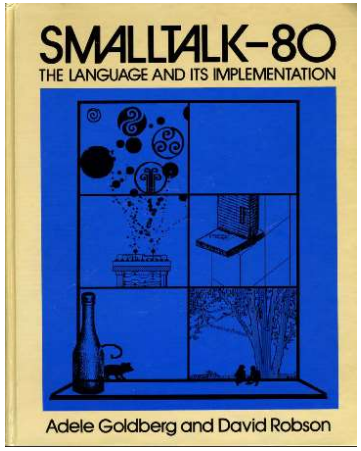
Nivel experto

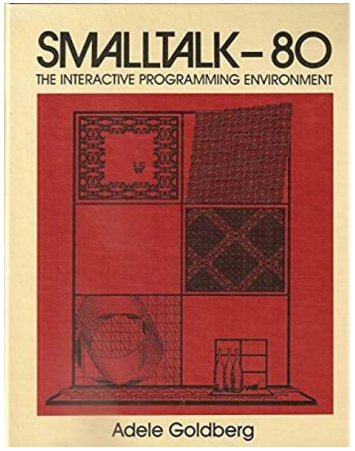


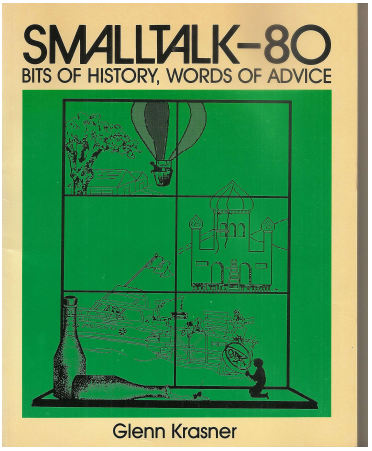
Nivel super mega requete experto

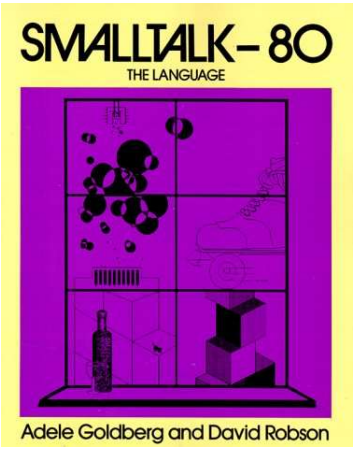


Blue, Green, Orange y Purple books







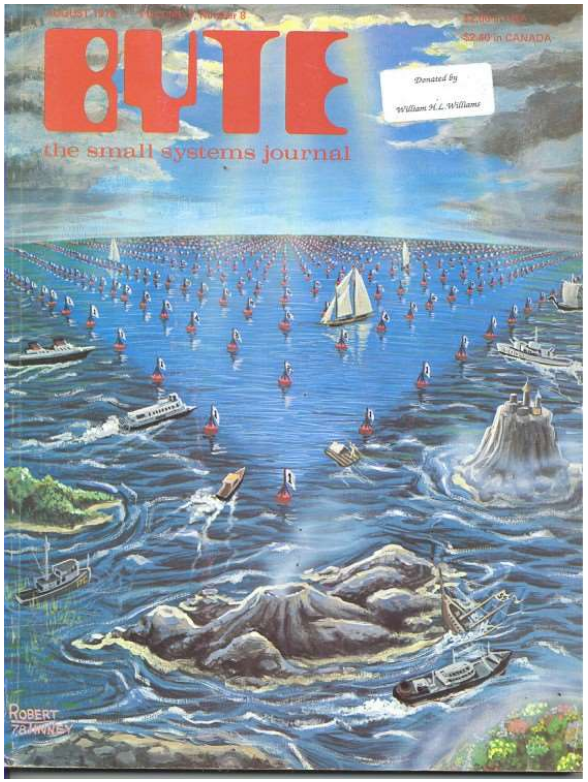


Unas anécdotas ...

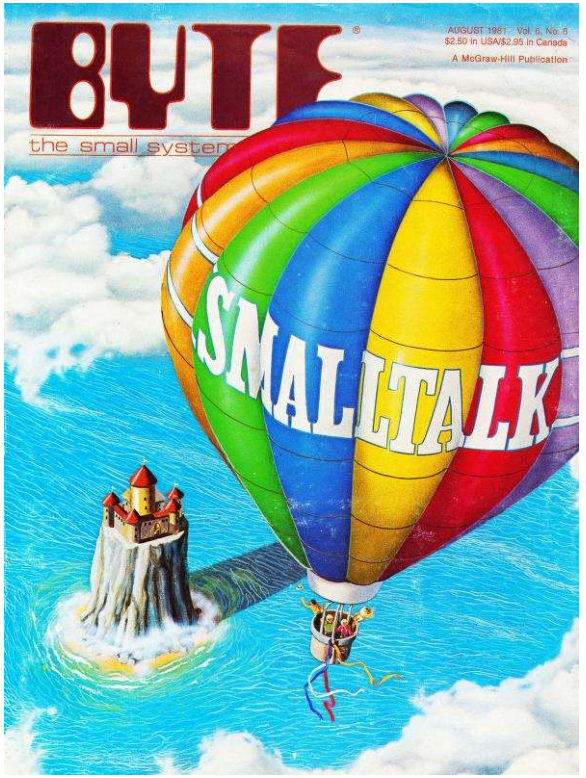
Piratas del Silicon Valley

https://www.youtube.com/watch?v=2u70CgBr-OI

Agosto 1978



Agosto 1981



https://www.tech-insider.org/star/research/acrobat/8108.pdf

JavaScript (ECMAScript)

- Lenguaje de propósito general
- Dinámico
- Basado en objetos (con base en prototipos en lugar de clases)
- Multiparadigma
- Se adapta a una amplia variedad de estilos de programación
- Pensado originalmente para scripting de páginas web
- Con una fuerte adopción en el lado del servidor (NodeJS)

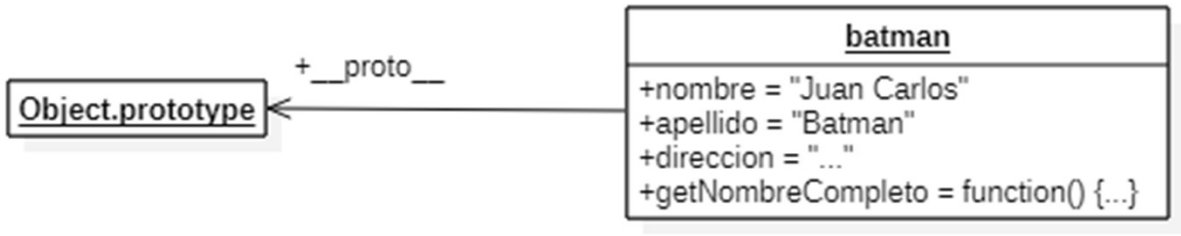
Un mínimo de sintaxis

```
sumar = function(a,b) {return a + b}

sumar(1,2)

function restar(a, b) {return a-b}

batman = { nombre: 'Juan Carlos', apellido: 'Batman', edad: 53 };

batman.edad = 44

batman.direccion = "Baticueva 14, entre Gallos y Medianoche"

batman.getNombreCompleto = function()
            {return this.nombre + " " + this.apellido}

batman.getNombreCompleto()
```

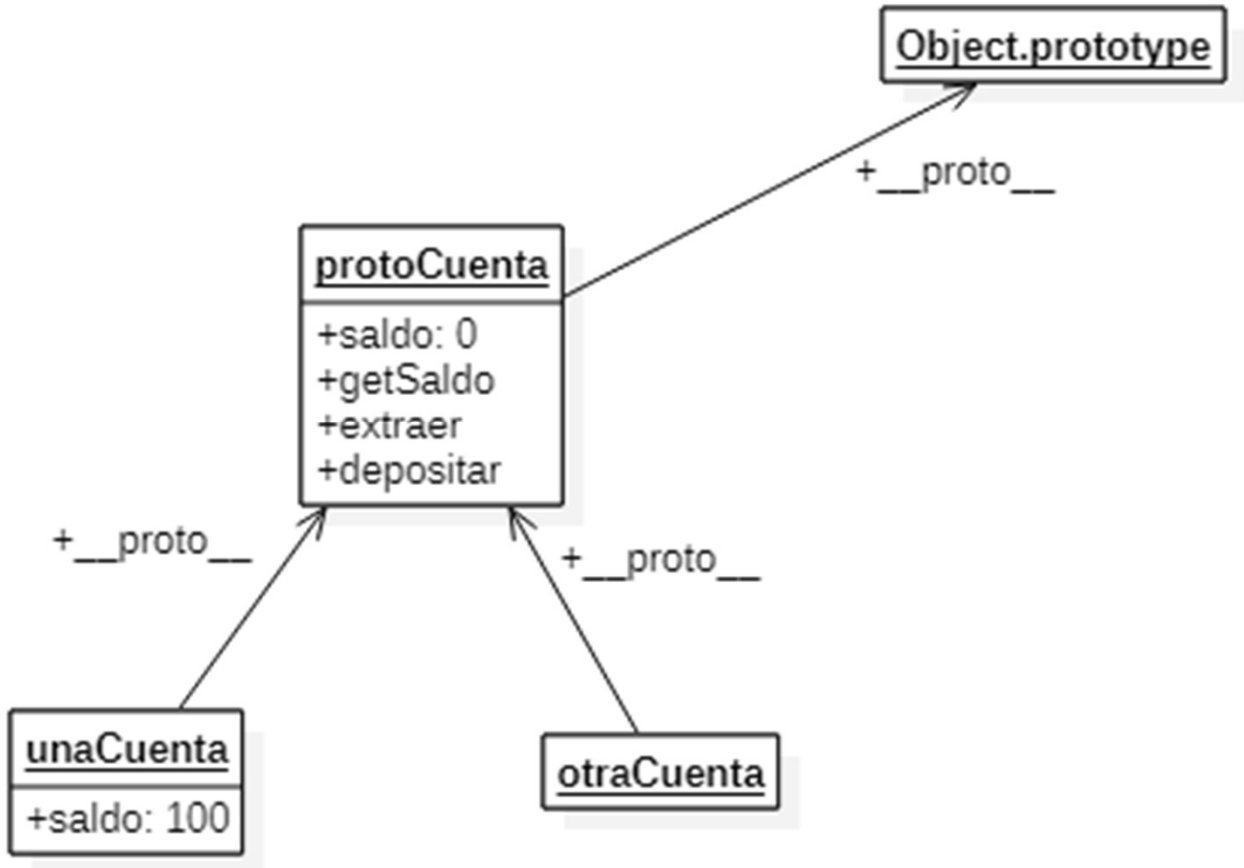
Prototipos

- En Javascript no tengo clases
- La forma mas simple de crear un objeto es mediante la notación literal (estilo JSON)
- Cada objeto puede tener su propio comportamiento (métodos)
- Los objetos heredan comportamiento y estado de otros (sus prototipos)
- Cualquier objeto puede servir como prototipo de otro
- Puedo cambiar el prototipo de un objeto (y así su comportamiento y estado)
- Termino armando cadenas de delegación

__proto__

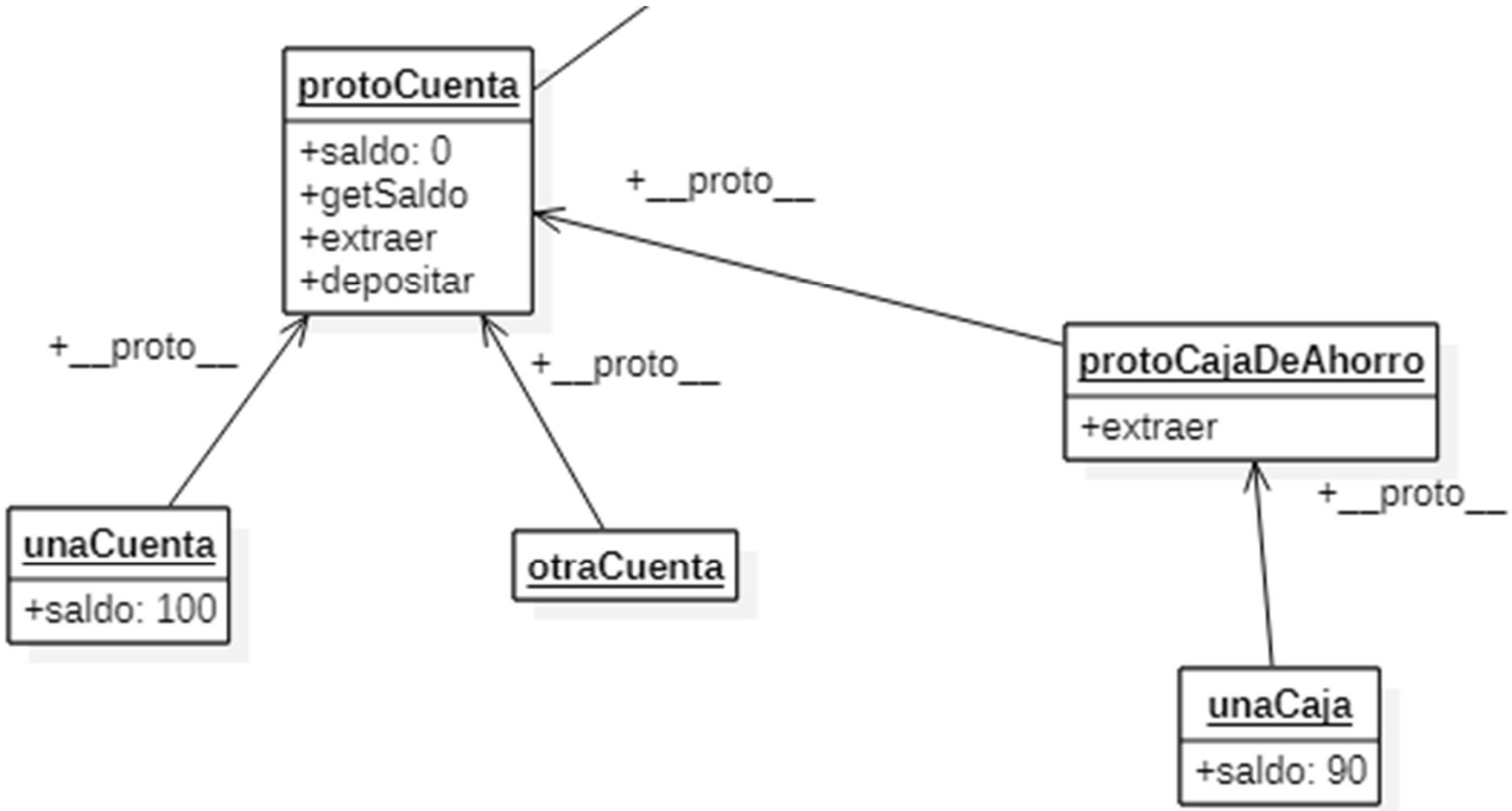



```
batman = { nombre: 'Juan Carlos', apellido: 'Batman', edad: 53 };

batman.getNombreCompleto = function()
            {return this.nombre + " " + this.apellido}

robin = {}


robin.__proto__ = batman;
```


Object.prototype
+__proto__
protoCuenta
+saldo: 0
+getSaldo
+extraer
+depositar
+__proto__
+__proto__
unaCuenta
+saldo: 100
otraCuenta

```
> protoCuenta = { saldo: 0}
◂ ▶ {saldo: 0}
> protoCuenta.getSaldo = function() { return this.saldo }
◂ ƒ () { return this.saldo }
> protoCuenta.depositar = function(monto) { this.saldo = this.saldo + monto }
◂ ƒ (monto) { this.saldo = this.saldo + monto }
> protoCuenta.extraer = function(monto) { this.saldo = this.saldo - monto }
◂ ƒ (monto) { this.saldo = this.saldo - monto }
> unaCuenta = Object.create(protoCuenta)
◂ ▶ {}
> unaCuenta.getSaldo()
◂ 0
> unaCuenta.depositar(100)
◂ undefined
> unaCuenta.getSaldo()
◂ 100
```

Prototipos


protoCuenta
+saldo: 0
+getSaldo
+extraer
+depositar
+__proto__
+__proto__
+__proto__
protoCajaDeAhorro
+extraer
+__proto__
unaCuenta
+saldo: 100
otraCuenta
unaCaja
+saldo: 90

```
> protoCajaDeAhorro = Object.create(protoCuenta)
<· ▶ {}
> protoCajaDeAhorro.extraer = function(monto) {
      if (monto < this.saldo) {
          this.saldo = this.saldo - monto;
      }
  }
<· ƒ (monto) {
      if (monto < this.saldo) {
          this.saldo = this.saldo - monto;
      }
  }
```

Prototipos y “herencia”

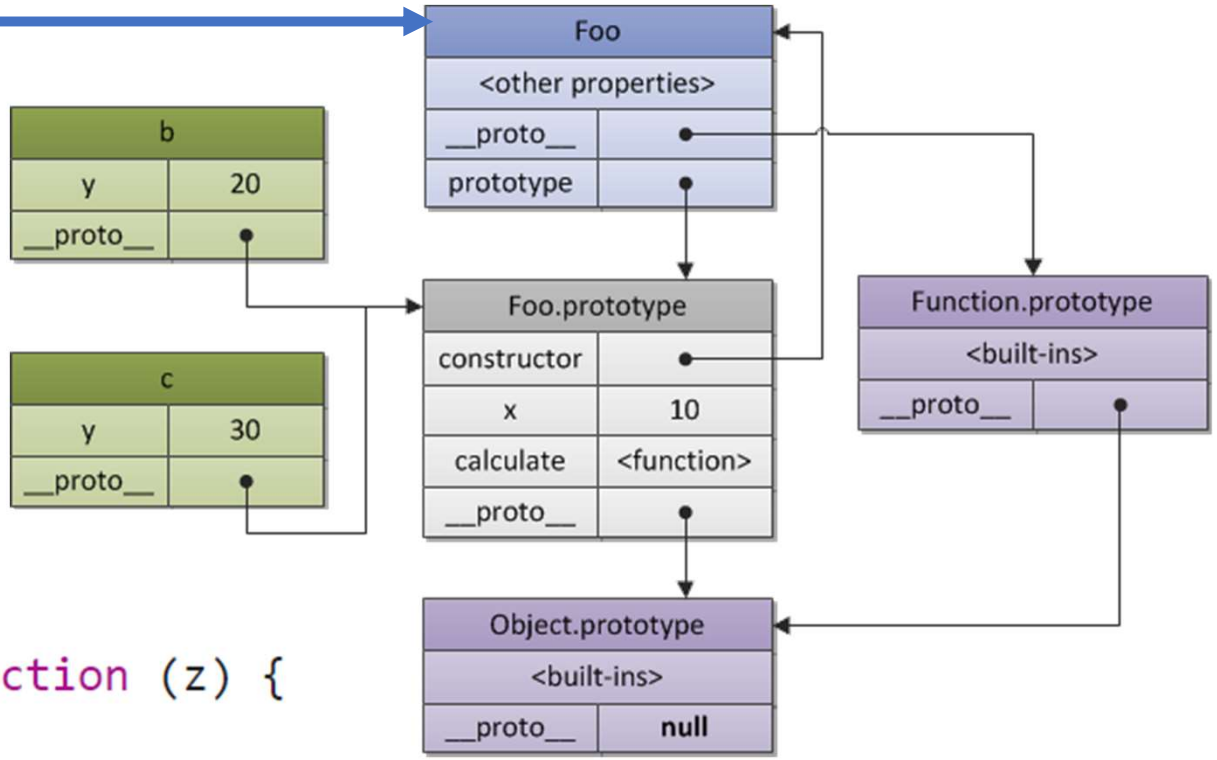
```
> unaCajaDeAhorro = Object.create(protoCajaDeAhorro)
<· ▶ {}
> unaCajaDeAhorro.getSaldo()
<· 0
> unaCajaDeAhorro.extraer(10)
<· undefined
> unaCajaDeAhorro.getSaldo()
<· 0
> unaCajaDeAhorro.depositar(100)
<· undefined
> unaCajaDeAhorro.extraer(10)
<· undefined
> unaCajaDeAhorro.getSaldo()
<· 90
```

Funciones que son objetos y constructores

```
function Persona(nombre, apellido) {
    this.nombre = nombre,
    this.apellido = apellido
}

juan = new Persona("juan","gomez")
```

¿Que prototipo tienen esas personas?
¿Cómo lo uso para agregar datos comportamiento que hereden todas?

```
function Foo(y) {
  this.y = y;
}

Foo.prototype.x = 10;

Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

var b = new Foo(20);
var c = new Foo(30);
```



ES6 – Clases como azúcar sintáctico

```
class Cuenta {

    constructor() {
        this.saldo = 0;
    }

    getSaldo() {
        return this.saldo;
    };

    depositar(monto) {
        this.saldo = this.saldo + monto;
    };

    extraer(monto) {
        this.saldo = this.saldo - monto;
    };

};
```

```
class CajaDeAhorro extends Cuenta {

    extraer(monto) {
        if (monto < this.saldo) {
            super.extraer(monto);
        }
    }
};
```