

Classes

Introducción

- Es un tema muy potente
- Va más allá de la sintaxis
- Una de las características más deseadas de **ES6**
- Un pilar fundamental de...

Introducción

Programación Orientada a **Objetos**

El problema

- Programación Estructurada
- Spaghetti!
- El “qué” se pierde en el “cómo”
- Poca modularización
- Difícil de mantener y de reutilizar

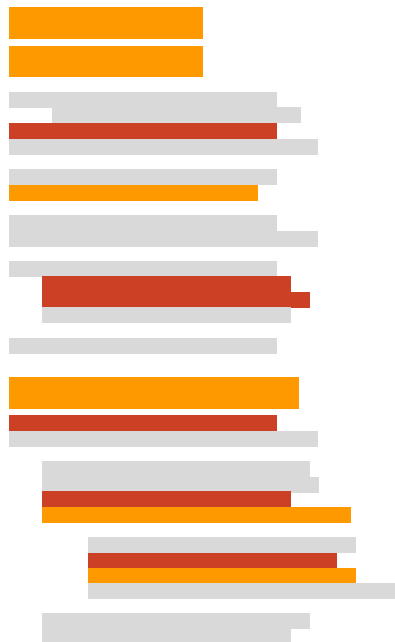
El problema

[Redacted text block containing multiple lines of obscured content]

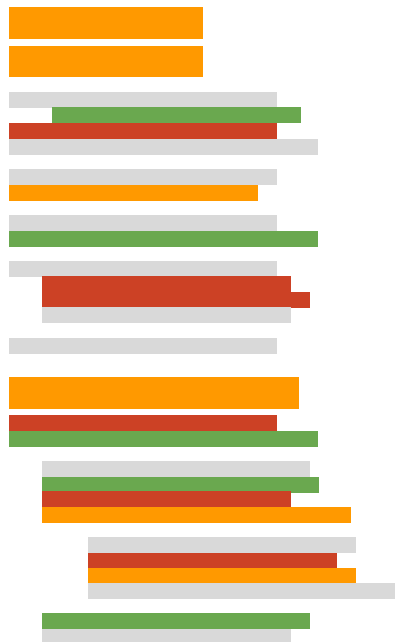
El problema



El problema



El problema



El problema



El problema



El problema



El problema



El problema

- El enfoque no escala a programas grandes
 - El **propósito** de un fragmento de código no es evidente
 - El **estado** se modifica y se consulta por todas partes
 - No es evidente qué partes del código utilizan qué variables
 - Es *muy difícil* encontrar y resolver errores
 - Es *muy difícil* meter a un nuevo programador

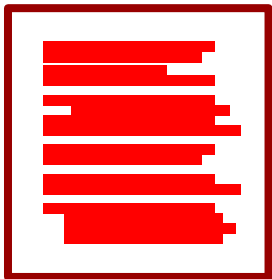
La visión



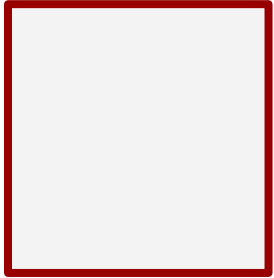
La visión



La visión



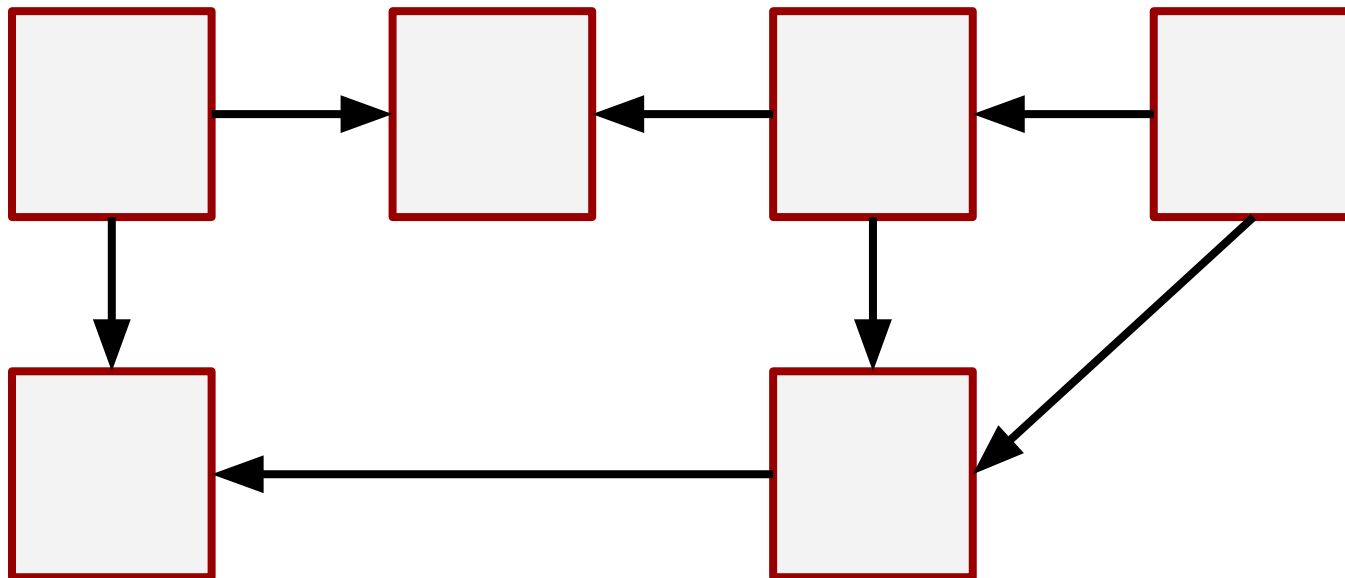
La visión



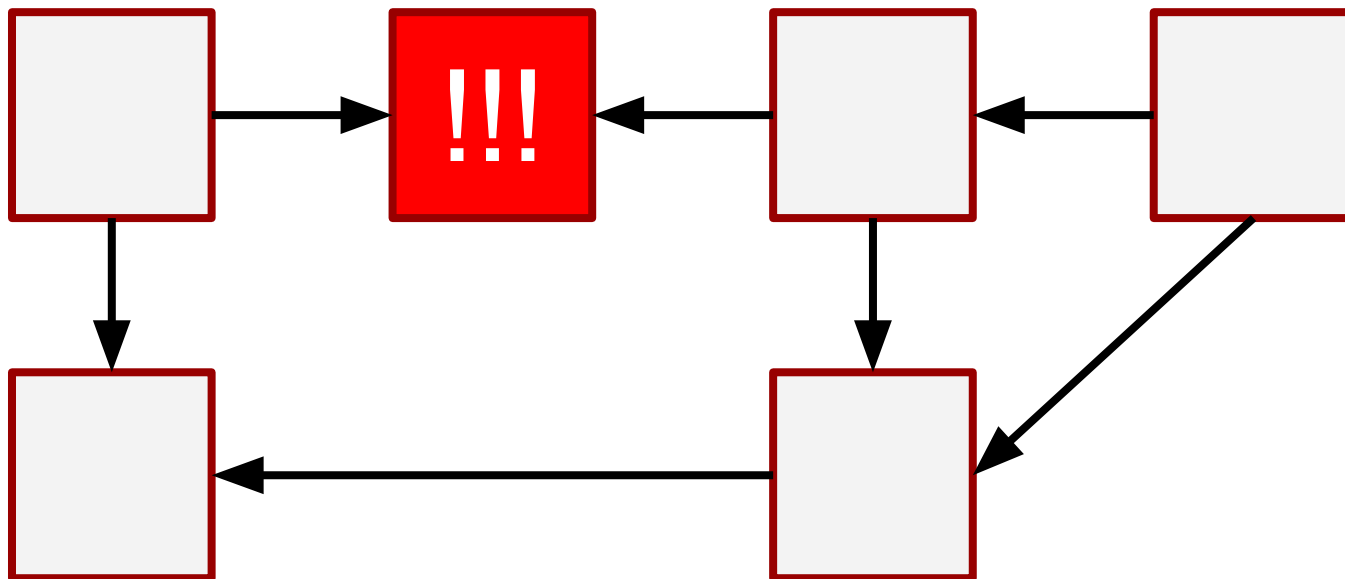
La visión



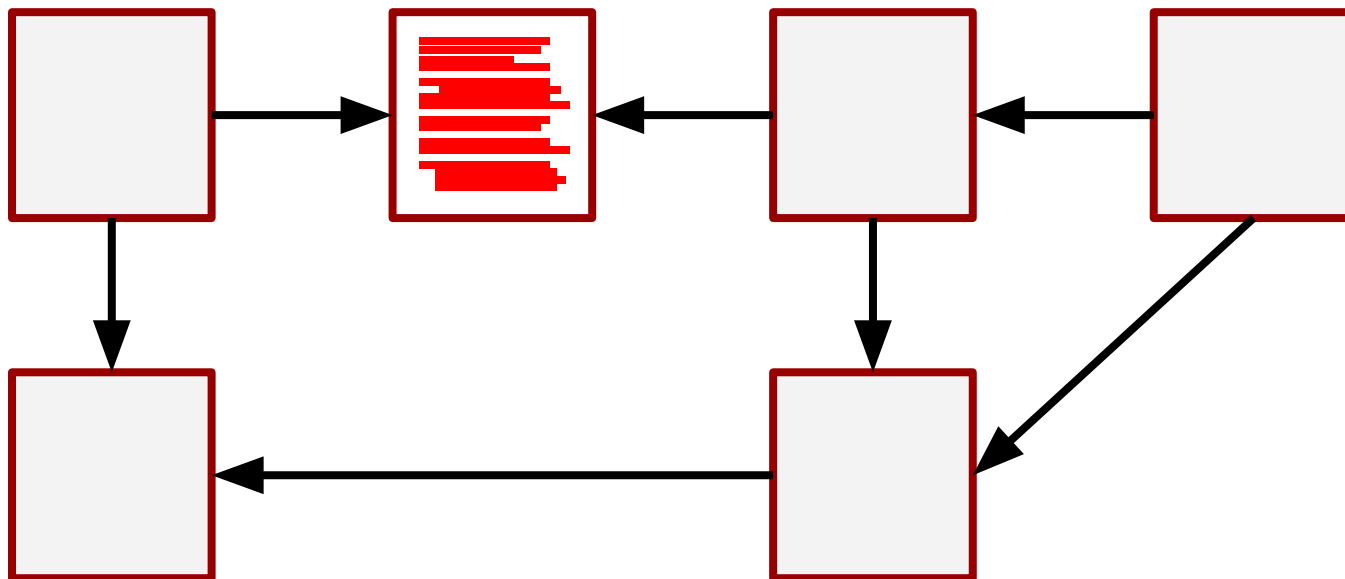
La visión



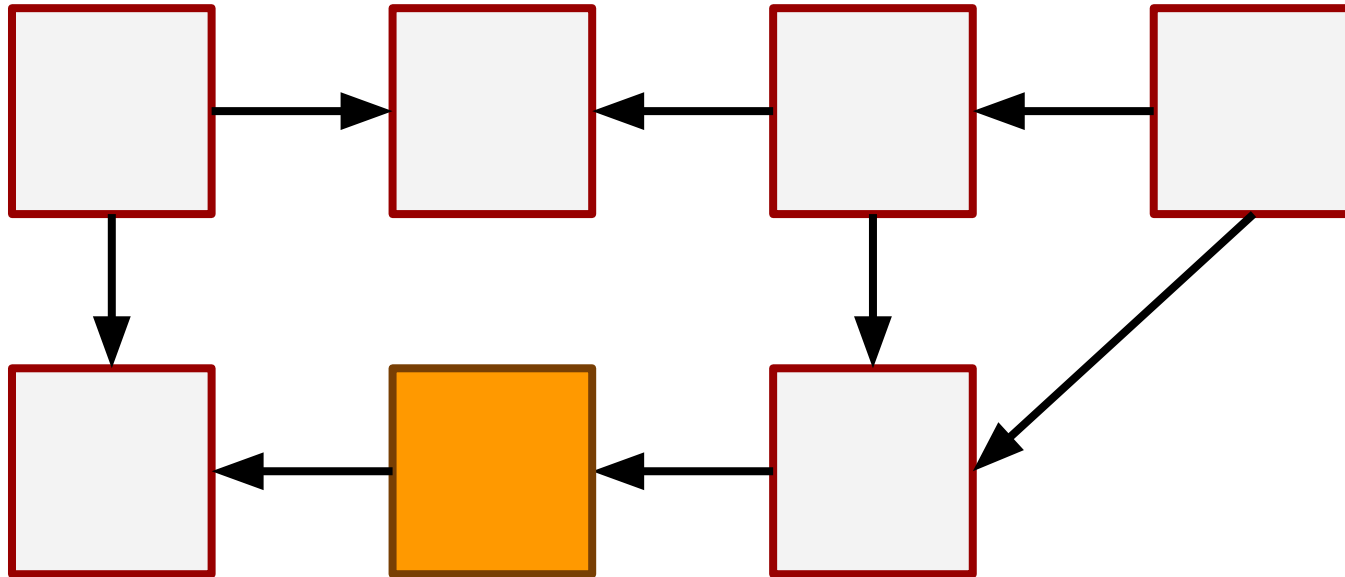
La visión



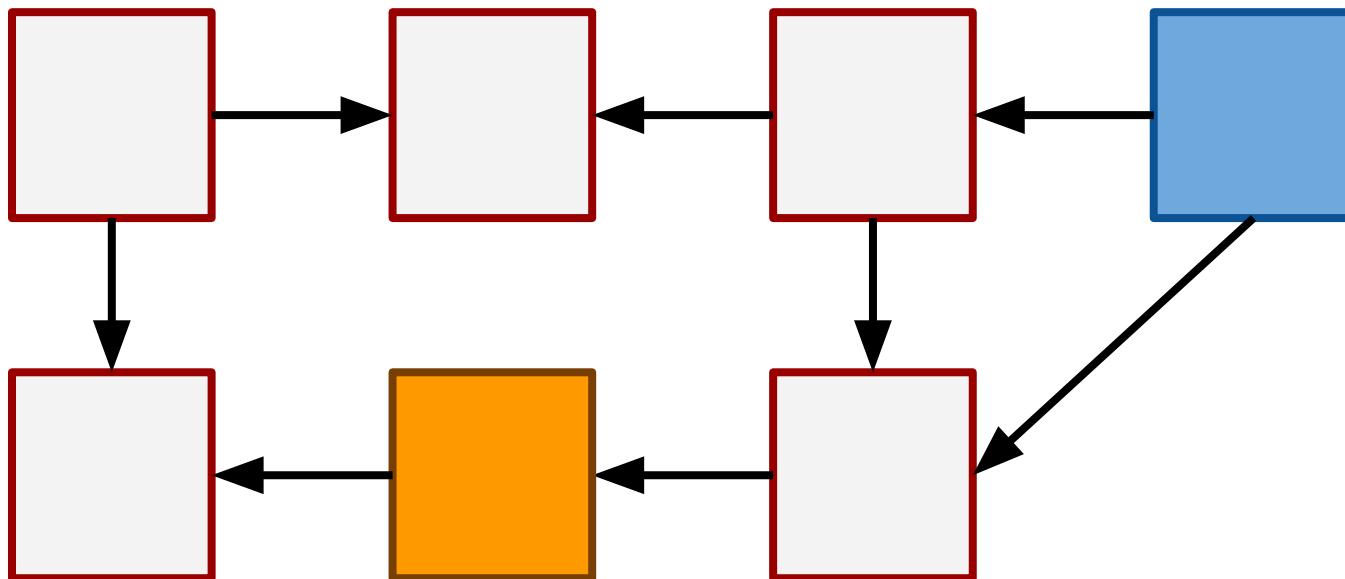
La visión



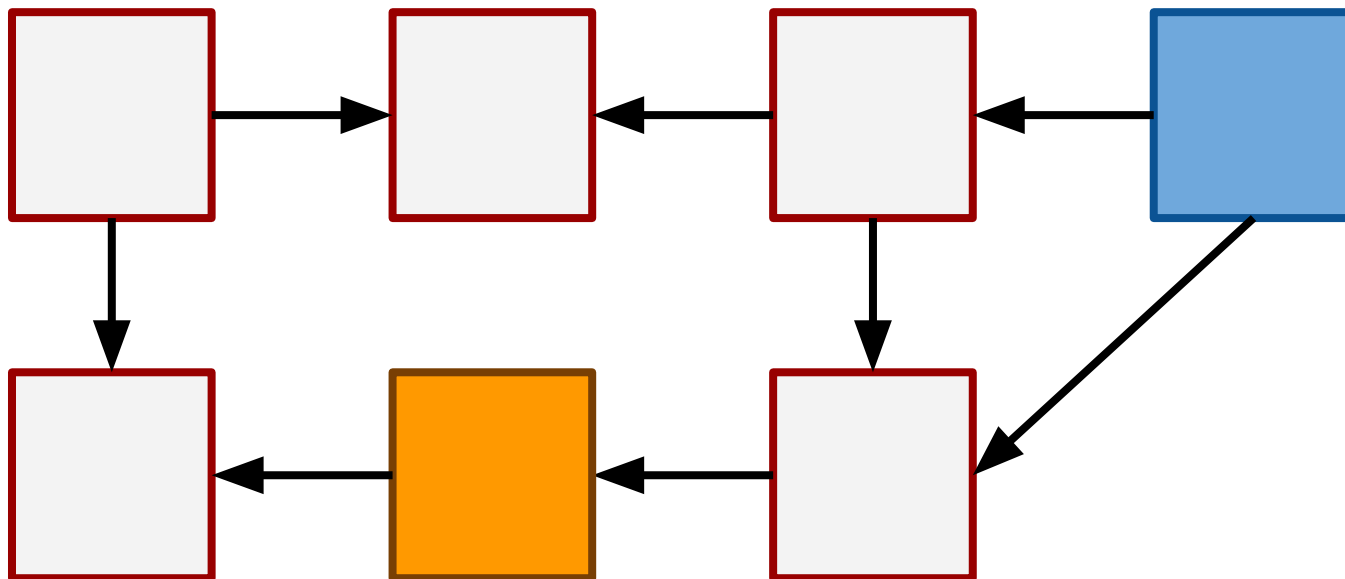
La visión



La visión



La visión



La visión

- Dividir un programa grande en **sub-programas**
- Cada uno con una **única responsabilidad**
- Cada uno con su propio **estado interno**
- **Independientes**
- Que se **comuniquen** según un protocolo bien definido

Programación Orientada a Objetos

- Es, sobre todo, una **herramienta conceptual**
- Que nos permite **modelar** nuestros programas mejor
- Y **razonar** sobre nuestro código con más claridad

Programación Orientada a Objetos

- Alan Kay y Smalltalk-72
- Tres requisitos para POO:
 - Encapsular estado local
 - Envío de mensajes
 - *Late binding*

Programación Orientada a Objetos

- Encapsular estado local:
 - **Propiedades!**
 - Ya hemos hablado de este tema

Programación Orientada a Objetos

- Envío de mensajes:
 - **Métodos**
 - Un objeto **expone** un conjunto de **operaciones**
 - Nos **comunicamos** con él mediante invocaciones...
 - ...que pueden recibir parámetros...
 - .. pero que **NO son llamadas a una función**

Programación Orientada a Objetos

¿No son llamadas a una función?

Programación Orientada a Objetos

- *Late Binding*
 - El **valor** se decide **en en momento de la invocación**

Programación Orientada a Objetos

- *Late Binding*
 - El **valor** se decide **en en momento de la invocación**
 - La referencia **al receptor del mensaje**
 - La **implementación** del método

Programación Orientada a Objetos

```
const obj = { counter: 0 };  
obj.counter++;  
  
console.log(obj.counter);
```

Programación Orientada a Objetos

```
const obj = { counter: 0 };

obj.increment = function () {
  // closure
  obj.counter++;
}

/* this is OK */

console.log(obj.counter) // -> 0
obj.increment()
console.log(obj.counter) // -> 1
```

Programación Orientada a Objetos

```
const obj = { counter: 0 };
```

```
obj.increment = function () {  
  // closure  
  obj.counter++;  
}
```

```
/* this is OK */
```

```
console.log(obj.counter) // -> 0  
obj.increment()  
console.log(obj.counter) // -> 1
```

```
/* this is NOT OK */
```

```
const obj2 = { counter: 0 };  
obj2.increment = obj.increment;
```

```
console.log(obj2.counter) // -> 0  
obj2.increment()  
console.log(obj2.counter) // -> 0
```

```
/* and... */
```

```
console.log(obj2.counter) // -> 2
```

Programación Orientada a Objetos

```
const obj = { counter: 0 };

obj.increment = function () {
  // closure
  obj.counter++;
}

/* this is OK */

console.log(obj.counter) // -> 0
obj.increment()
console.log(obj.counter) // -> 1
```

/* this is NOT OK */

```
const obj2 = { counter: 0 };
obj2.increment = obj.increment;
```

```
console.log(obj2.counter) // -> 0
obj2.increment()
console.log(obj2.counter) // -> 0
```

/* and... */

```
console.log(obj2.counter) // -> 2
```

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que no tenga binding léxico
 - Que apunte al objeto “adecuado”

Programación Orientada a Objetos

¿Cuál es el objeto adecuado?

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **está a la izquierda del punto**

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **recibe el mensaje**

Programación Orientada a Objetos

- Necesitamos...
 - Una referencia que **no tenga binding léxico**
 - Que apunte al objeto que **recibe el mensaje**
 - Y se **vincule** en el momento de la **invocación**

Programación Orientada a Objetos

this

Programación Orientada a Objetos

- Invocar a un método **no** es llamar a una función
 - Necesitamos más info que en una llamada a función:
 - **el objeto recibe el mensaje**
 - El lenguaje tiene que poner en marcha maquinaria adicional
 - **vincular `this`**
 - **seleccionar la implementación del método adecuada**

Programación Orientada a Objetos

- Estas dos *features* de javascript por sí solas:
 - propiedades
 - `this`
- Lo convierten en un lenguaje capaz de POO

Programación Orientada a Objetos

- Estas dos *features* de javascript por sí solas:
 - propiedades
 - `this`
- Lo convierten en un lenguaje capaz de POO
- Pero hay un tercer concepto que lo hace mucho más flexible... ¿os suena?

Clases

- JS nos permite modelar con objetos
- Pero crear cada objeto uno a uno es...
 - Muy laborioso
 - Muy poco escalable

Clases

- Las **clases** automatizan la generación de objetos
- Una clase define un ***arquetipo***
 - inicialización de estado
 - comportamiento (mensajes)

```
class Dog {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  bark() {  
    console.log("wof, wof...");  
  }  
  
  sit() {  
    console.log(`* ${this.name} sits and looks at you.`);  
  }  
  
}
```


Clases

- Generar y configurar un objeto
- *Instanciar* la clase
- instrucción **new**

```
class Dog {  
  constructor(name) {  
    this.name = name;  
  }  
  
  bark() {  
    console.log("wof, wof...");  
  }  
  
  sit() {  
    console.log(`* ${this.name} sits and looks at you.`);  
  }  
}  
  
const toby = new Dog('Toby');  
toby.sit();  
  
const spot = new Dog('Spot');  
spot.bark();
```

Clases

¿Qué valor nos devuelve esta expresión?

`typeof Dog`

Clases

¿¿Por qué??

Clases

Las clases de ES6 son **azúcar sintáctico**

Clases

En el fondo, las “clases” de Javascript están implementadas a base de...

Clases

Herencia de prototipos

Clases

(pausa dramática)


```
class Dog {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  bark() {  
    console.log("wof, wof...");  
  }  
  
  sit() {  
    console.log(`* ${this.name} sits and looks at you.`);  
  }  
  
}
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

Clases Constructores

¿Verdadero o falso?

```
Dog.prototype === Dog.__proto__
```

Clases Constructores

¿Para qué sirve la propiedad prototype?

Clases Constructores

La respuesta está en **new**

Clases Constructores

- Una función se ejecuta como constructor cuando la llamada está precedida por **new**
- Antes de ejecutar un constructor suceden **tres cosas**:

Clases Constructores

1. Se crea **un nuevo objeto** vacío
2. Se le asigna como **prototipo** el **valor de la propiedad **prototype**** del constructor
3. **this** dentro del constructor se vincula a este nuevo objeto

Clases Constructores

- Por último, se ejecuta el código del constructor
- El valor de la expresión **new Constructor()** será:
 - El nuevo objeto...
 - ...a no ser que el constructor devuelva otro valor con **return**

```
function Dog(name) {  
  this.name = name;  
}  
  
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}  
  
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}  
  
const toby = new Dog("Toby");  
toby.sit();
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");  
toby.sit();
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");  
toby.sit();
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");  
toby.sit();
```

Clases Constructores

¿Verdadero o falso?

```
toby.hasOwnProperty("name")
```

Clases Constructores

¿Verdadero o falso?

```
toby.hasOwnProperty("sit")
```



```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.bark = function() {  
  console.log("wof, wof...");  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");  
toby.sit();
```

Clases Constructores

- Cada instancia guarda su propio estado
- Pero comparten la implementación de los métodos a través de su prototipo

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} does not understand.`);  
}
```

```
const spot = new Dog("Spot");  
spot.sit();
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");
```

```
Dog.prototype.sit = function() {  
  console.log(`* ${this.name} does not understand.`);  
}
```

```
const spot = new Dog("Spot");  
spot.sit();  
toby.sit();
```

```
function Dog(name) {  
  this.name = name;  
}
```

```
Dog.prototype.sit = () => {  
  console.log(`* ${this.name} sits and looks at you.`);  
}
```

```
const toby = new Dog("Toby");  
toby.sit();
```

Herencia

- Muy incómoda con constructores
- Bastante rota...

```
function Mammal(name) {  
  this.name = name;  
}  
  
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');}  
  
function Dog(name) {  
}  
  
Dog.prototype = Mammal;  
  
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}  
  
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');}  
  
function Dog(name) {  
  
  Dog.prototype = Mammal;  
  
  const toby = new Dog('Toby');  
  toby.breathe();  
}
```



```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
}  
  
function Dog(name) {  
}
```

```
Dog.prototype = new Mammal('???');
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
function Dog(name) {  
  // ???  
}
```

```
Dog.prototype = new Mammal('');
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
function Dog(name) {  
  // WRONG!  
  Mammal(name);  
}
```

```
Dog.prototype = new Mammal('');
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
function Dog(name) {  
  // STILL WRONG!  
  new Mammal(name);  
}
```

```
Dog.prototype = new Mammal('');
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
function Dog(name) {  
  Mammal.call(this, name);  
}
```

```
Dog.prototype = new Mammal('');
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
}  
  
function Dog(name) {  
  Mammal.call(this, name);  
}
```

```
Dog.prototype = new Mammal();
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}  
  
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you. ');  
}  
  
function Dog(name) {  
  Mammal.call(this, name);  
}  
  
Dog.prototype = new Mammal('');  
  
Dog.prototype.breathe = function() {  
  console.log('* The dog looks at you puzzled. ');  
}  
  
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}  
  
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you. ');  
}  
  
function Dog(name) {  
  Mammal.call(this, name);  
}  
  
Dog.prototype = new Mammal('');  
  
Dog.prototype.breathe = function() {  
  console.log('* The dog looks at you puzzled. ');  
}  
  
const toby = new Dog('Toby');  
toby.breathe();
```



```
function Mammal(name) {  
  this.name = name;  
}
```

```
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you.');
```

```
}  
  
function Dog(name) {  
  Mammal.call(this, name);  
}
```

```
Dog.prototype = new Mammal('');
```

```
Dog.prototype.breathe = function() {  
  console.log('* The dog looks at you puzzled.');
```

```
// how can I call the parent implementation from here??  
}  
  
const toby = new Dog('Toby');  
toby.breathe();
```

```
function Mammal(name) {
  this.name = name;
}

Mammal.prototype.breathe = function() {
  console.log('* A deep breath sound reaches you. ');
}

function Dog(name) {
  Mammal.call(this, name);
}

Dog.prototype = new Mammal('');
```

```
Dog.prototype.breathe = function() {
  console.log('* The dog looks at you puzzled. ');
  Mammal.prototype.breathe.call(this);
}
```

```
const toby = new Dog('Toby');
toby.breathe();
```

```
function Mammal(name) {  
  this.name = name;  
}  
  
Mammal.prototype.breathe = function() {  
  console.log('* A deep breath sound reaches you. ');  
}  
  
function Dog(name) {  
  Mammal.call(this, name);  
}  
  
Dog.prototype = new Mammal('');  
  
Dog.prototype.breathe = function() {  
  console.log('* The dog looks at you puzzled. ');  
  Mammal.prototype.breathe.call(this);  
}  
  
const toby = new Dog('Toby');  
toby.breathe();
```

Clases

Vamos a repetir el mismo ejemplo, pero con ES6 **class**

```
class Mammal {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    breathe() {  
        console.log('* A deep breath sound reaches you.');    }  
  
}  
  
class Dog extends Mammal {  
  
}  
  
const toby = new Dog('Toby');  
toby.breathe();
```

```
class Mammal {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
  
}
```

```
class Dog extends Mammal {  
}
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
class Mammal {  
  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
  
}
```

```
class Dog extends Mammal {  
}
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
class Mammal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
  constructor(name) {  
    // ???  
  }  
}
```

```
const toby = new Dog('Toby');  
toby.breathe();
```



```
class Mammal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
  constructor(name) {  
    super(name);  
  }  
}
```

```
const toby = new Dog('Toby');  
toby.breathe();
```

```
class Mammal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
  constructor(name) {  
    super(name);  
  }  
  breathe() {  
    console.log('* A deep breath sound reaches you.');    // ???  
  }  
}
```

```
class Mammal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
  constructor(name) {  
    super(name);  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');    super.breathe();  
  }  
}
```

```
class Mammal {  
  constructor(name) {  
    this.name = name;  
  }  
  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
  constructor(name) {  
    super(name);  
  }  
  breathe() {  
    console.log('* A deep breath sound reaches you.');    super.breathe();  
  }  
}
```

Herencia

- Herencia => polimorfismo
 - uno de los mecanismos más potentes del diseño con clases
 - Fundamental en POO
 - La implementación en JS es sencilla (y elegante!)

```
class Mammal {  
  breathe() {  
    console.log('* A deep breath sound reaches you.');  }  
}
```

```
class Dog extends Mammal {  
}
```

```
class Cat extends Mammal {  
}
```

```
[new Cat(), new Dog()].map(m => m.breathe)
```

Polimorfismo

- Invocar a un método vs. llamar a una función:
 - Tienen sintaxis muy similar...
 - Pero su semántica es muy diferente
- Una gran diferencia: *binding*

Polimorfismo

- Cuando llamo a una función...
 - Sé **exactamente** qué código voy a ejecutar
 - Puedo **buscarlo** y **leerlo**
 - Puedo “saber” **el valor de todas las variables** a las que hace referencia

Polimorfismo

- Cuando invoco a un método...

```
function keepAlive(mammal) {  
  // where is "breathe" defined?  
  mammal.breathe();  
}
```

Polimorfismo

- Cuando invoco a un método...

```
function keepAlive(mammal) {  
  // where is "breathe" defined?  
  mammal.breathe();  
}
```

Polimorfismo

- Cuando invoco a un método...
 - **No sé qué código se va a ejecutar**
 - **Delego** en el objeto (por eso se llama “*paso de mensajes*”, para subrayar que la responsabilidad no es del llamante!)
 - Javascript no sabe qué código va a ejecutar hasta el **último momento** (*late binding*)

Polimorfismo

¿Por qué no sabe Javascript qué código se va a ejecutar hasta el último momento?

Polimorfismo

¿Qué mecanismo usa para encontrar el método adecuado?

Polimorfismo

¿Por qué podemos utilizar el código de un método del padre en una instancia del hijo?

Polimorfismo

Cadena de prototipos + **this**

=

late binding y dynamic dispatch por la cara!

Polimorfismo

- Expresión para saber si instancia deriva de constructor

```
class Mammal {  
    breathe() {  
        console.log('* A deep breath sound reaches you.');    }  
}
```

```
class Dog extends Mammal {}
```

```
console.log(new Dog instanceof Dog);
```


Polimorfismo

¿Verdadero o falso?

```
console.log(new Dog instanceof Dog);
```

Polimorfismo

¿Verdadero o falso?

```
console.log(new Dog instanceof Mammal);
```

Polimorfismo

¿Verdadero o falso?

```
console.log(new Dog instanceof Cat);
```

Polimorfismo

¿Verdadero o falso?

```
console.log(new Dog instanceof Object);
```

Clases Anónimas

- Se puede utilizar **class** como expresión
- Permite crear clases dinámicas y/o anónimas

Clases Anónimas

```
const Mammal = class {  
  constructor(name) {  
    this.name = name;  
  }  
}
```

```
const buddy = new Mammal('Buddy');  
console.log(buddy.name)
```

Ejercicio

- Escribe la función **withCounter** del siguiente ejemplo
- De **dos maneras**:
 - Usando clases anónimas
 - Usando constructores

Ejercicio

```
class Mammal {  
  breathe() { console.log('* you hear some air flowing'); }  
}
```

```
const CountedMammal = withCounter(Mammal);
```

```
const toby = new CountedMammal();  
toby.breath();
```

```
// attention!  
CountedMammal.getInstanceCount(); // -> 1
```

```
const spot = new CountedMammal();  
const willy = new CountedMammal();  
const toi = new CountedMammal();
```

```
CountedMammal.getInstanceCount(); // -> 4
```


Propiedades estáticas

¿Quién es el *receptor* del mensaje?

```
CountedMammal.getInstanceCount();
```

Propiedades estáticas

En la implementación de **getInstanceCount**,

¿A dónde apuntará **this**?

```
CountedMammal.getInstanceCount();
```

Propiedades estáticas

¿Cuál es el *prototipo de* **CountedMammal**?

```
CountedMammal.getInstanceCount();
```

Ejercicio

```
class Mammal {  
  breathe() { console.log('* you hear some air flowing' ); }  
}  
  
const CountedMammal = withCounter(Mammal);  
  
class Dog extends CountedMammal {  
  constructor(name) {  
    super()  
    this.name = name;  
  }  
}  
  
const toby = new Dog();
```

Propiedades estáticas

¿Qué valor me devuelve...?

```
toby.getInstanceCount();
```

Propiedades estáticas

¿Qué valor me devuelve...?

```
CountedMammal.getInstanceCount();
```

Propiedades estáticas

¿Qué valor me devuelve...?

```
Dog.getInstanceCount();
```

Propiedades estáticas

- Las **clases** (y, por tanto, los **constructores**) son también objetos
- Pueden almacenar propiedades
- Que no tienen nada que ver con las propiedades de sus **instancias**

Propiedades de clase

```
Dog.legs = 4;
```

```
Dog.walk = function(dog) {  
  console.log(`Come on, ${dog.name}, let's go outside!`);  
};
```

Propiedades de clase

```
class Dog extends CountedMammal {  
  constructor(name) {  
    super();  
    this.name = name;  
  }  
  static walk (dog) {  
    console.log(`Come on, ${dog.name}, let's go outside!`);  
  }  
}  
  
const spot = new Dog('Spot');  
Dog.walk(spot);
```

Propiedades de clase

```
class Dog extends CountedMammal {  
  constructor(name) {  
    super();  
    this.name = name;  
  }  
  static walk (dog) {  
    console.log(`Come on, ${dog.name}, let's go outside!`);  
  }  
}
```

```
const spot = new Dog('Spot');  
Dog.walk(spot);
```

Propiedades estáticas

```
spot.walk();
```

Propiedades estáticas

¿Cómo podría definir el valor de **Dog.legs** usando **static**?