Clases

Introducción

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

redradix

Introducción

Programación Orientada a **Objetos**

redradix

El problema

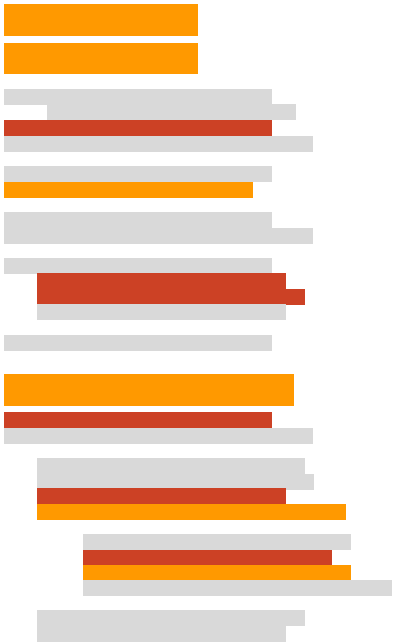
- Programación Estructurada
- Spaghetti!
- El “qué” se pierde en el “cómo”
- Poca modularización
- Difícil de mantener y de reutilizar

El problema

El problema

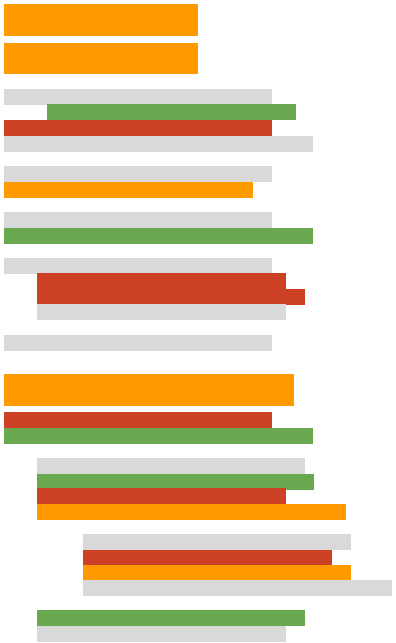
redradix

El problema



redradix

El problema



redradix

El problema

redradix

El problema

redradix

El problema

redradix

El problema

redradix

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

redradix

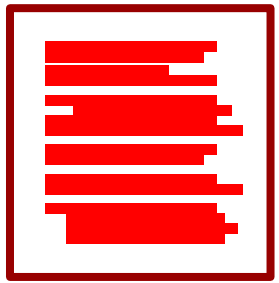
La visión

redradix

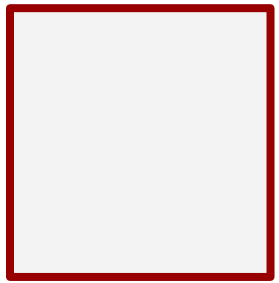
La visión

redradix

La visión

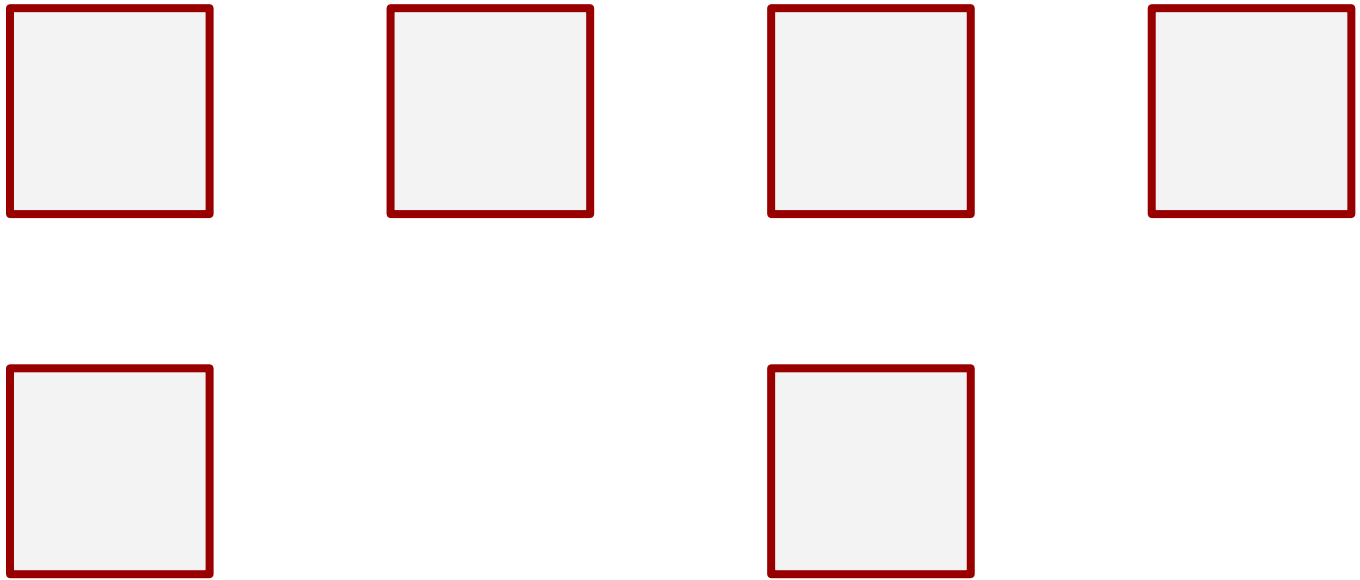


La visión



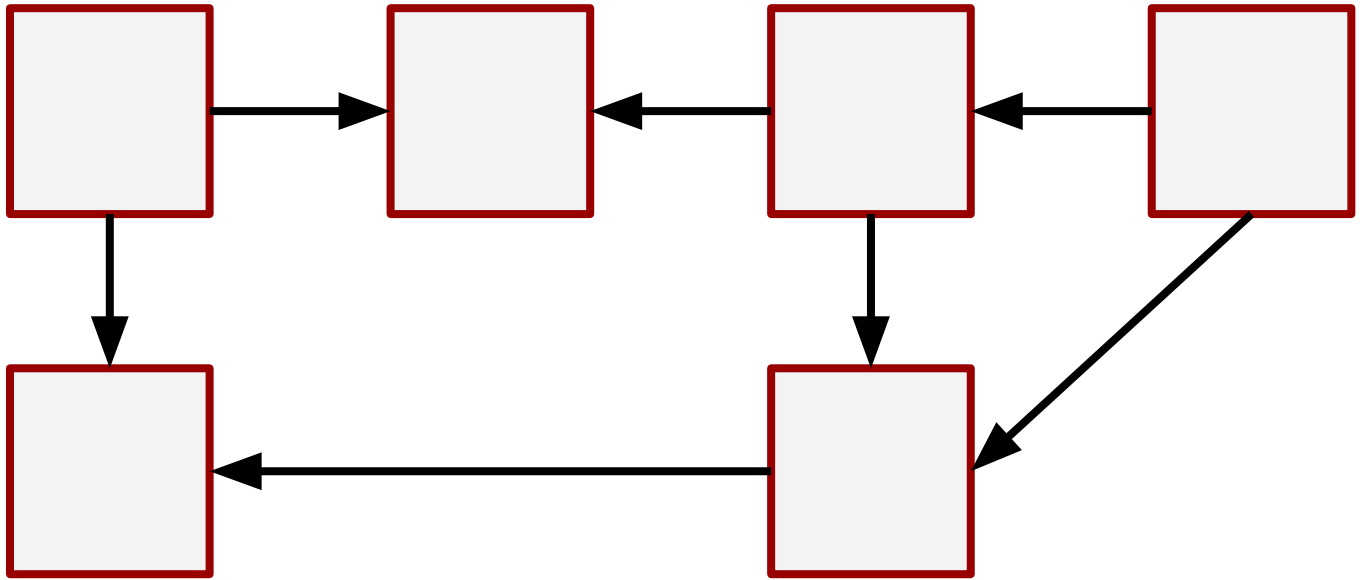
redradix

La visión



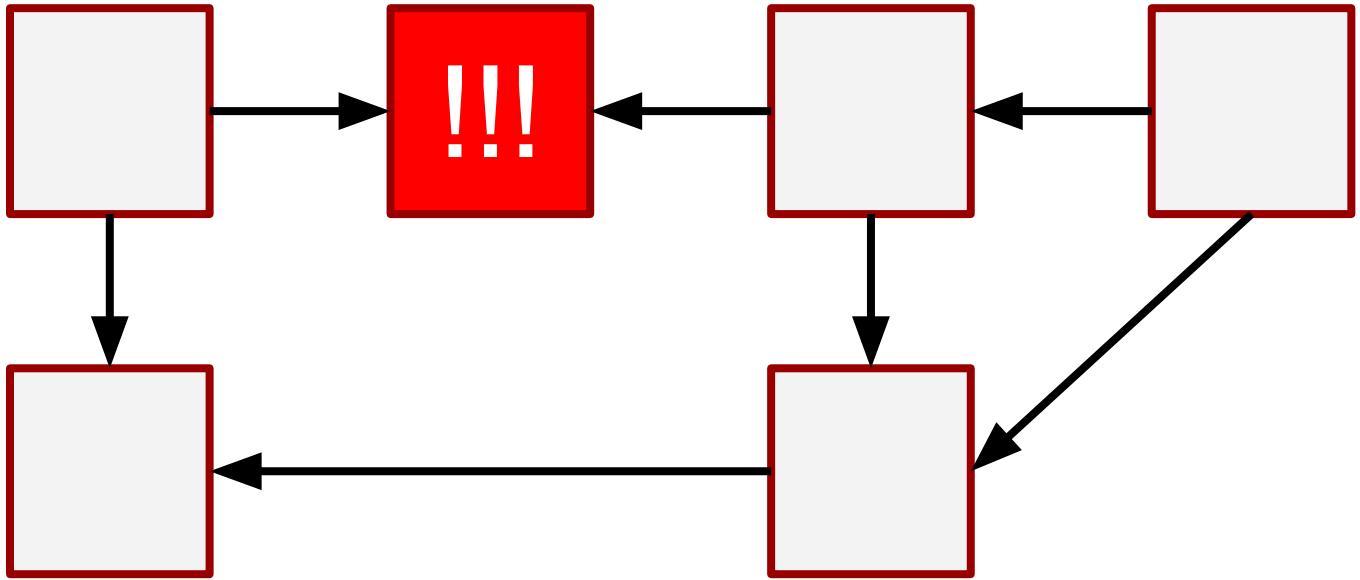
redradix

La visión



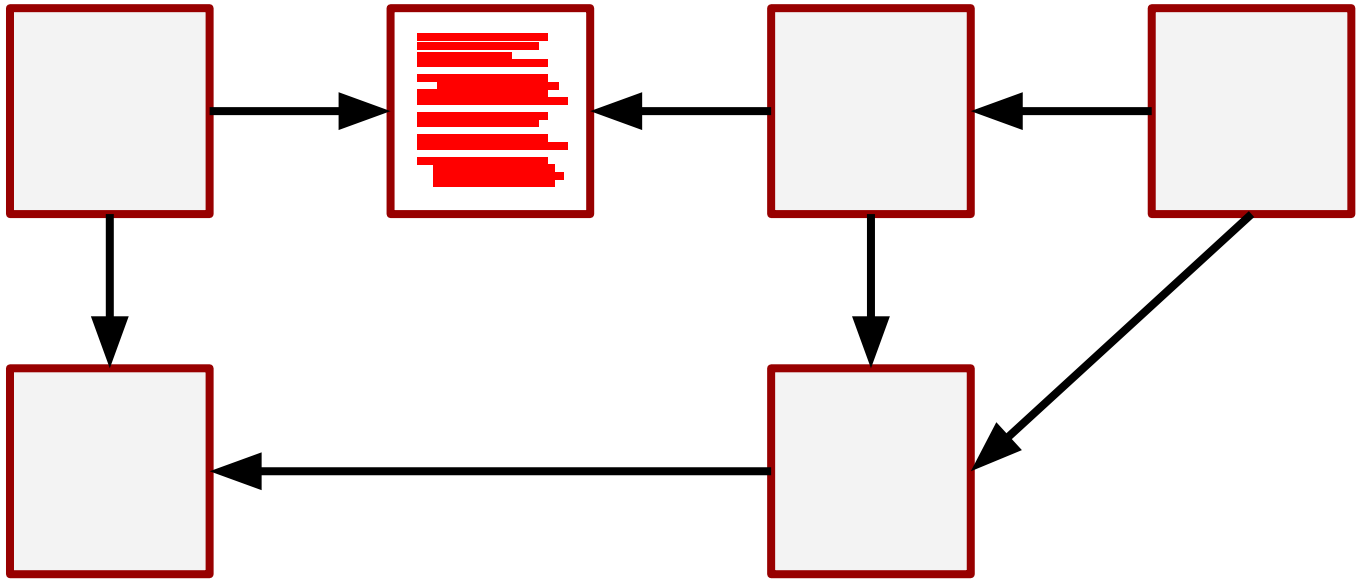
redradix

La visión



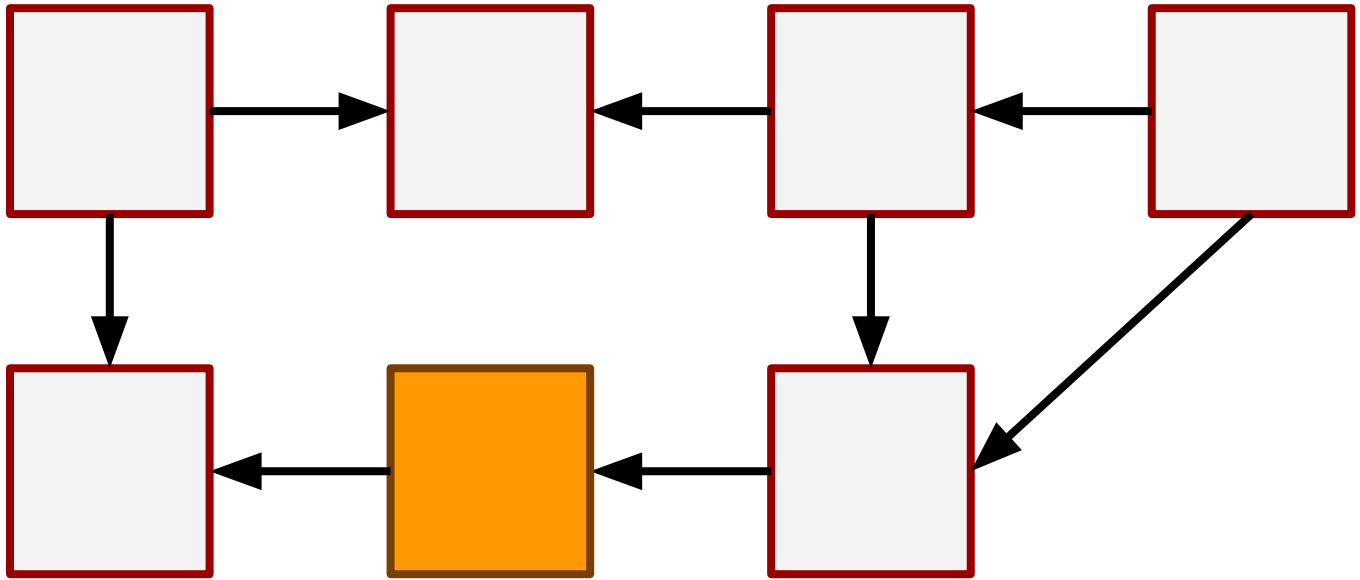
redradix

La visión



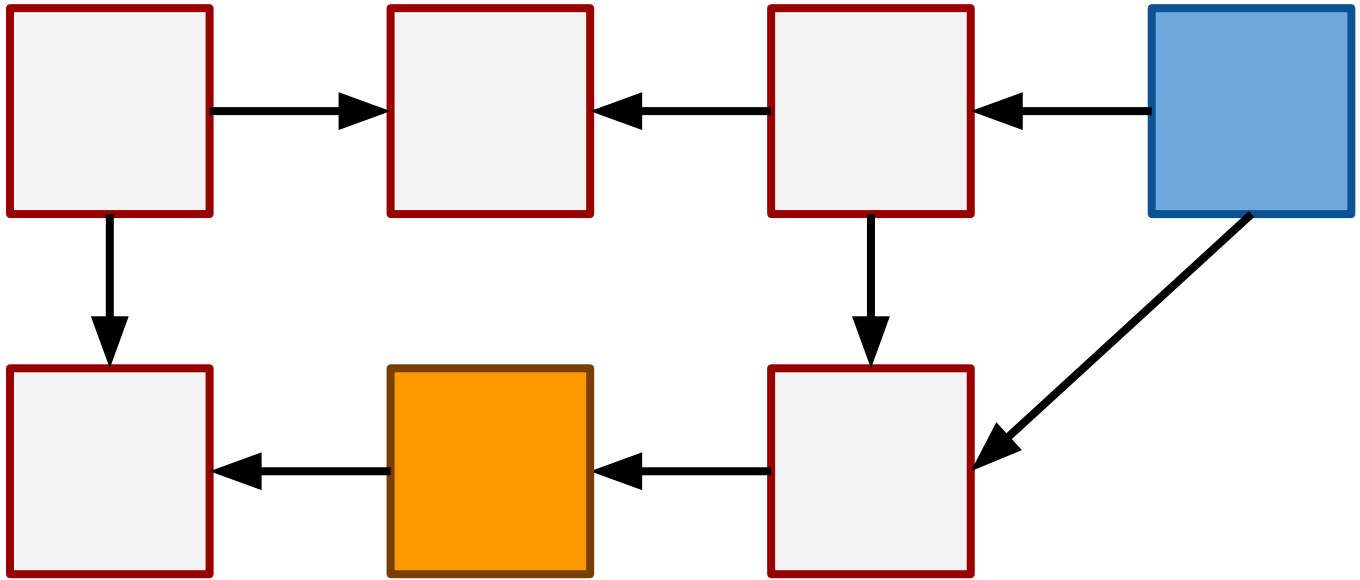
redradix

La visión



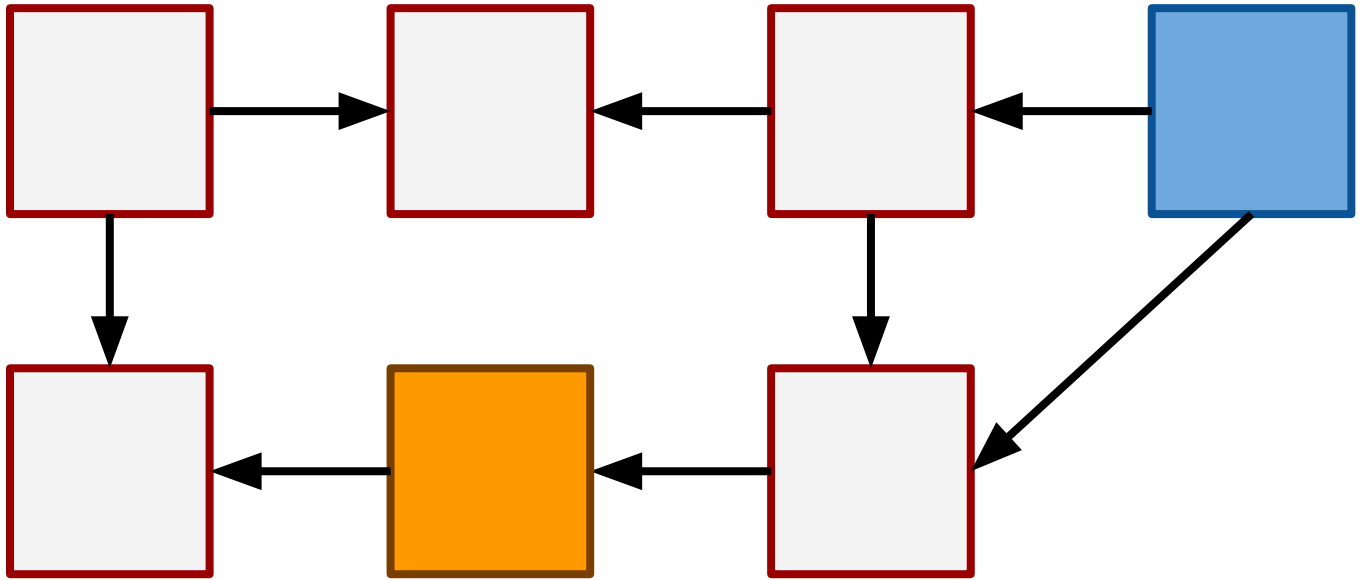
redradix

La visión



redradix

La visión



redradix

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

redradix

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

redradix

Programación Orientada a Objetos

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

redradix

Programación Orientada a Objetos

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

redradix

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?

redradix

Programación Orientada a Objetos

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

redradix

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

redradix

Programación Orientada a Objetos

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

console.log(obj.counter);
```

redradix

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
```

redradix

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(obj.counter) // -> 2
```

redradix

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(obj.counter) // -> 2
```

redradix

Programación Orientada a Objetos

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

redradix

Programación Orientada a Objetos

¿Cuál es el objeto adecuado?

redradix

Programación Orientada a Objetos

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

redradix

Programación Orientada a Objetos

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

redradix

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**

redradix

Programación Orientada a Objetos

`this`

redradix

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**

redradix

Programación Orientada a Objetos

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

redradix

Programación Orientada a Objetos

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: () => {
    console.log(`Hola, ${obj.nombre}`)
  }
};
```

¿Qué significa esto?

```
obj.nombre;
```

redradix

Programación Orientada a Objetos

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: () => {
    console.log(`Hola, ${obj.nombre}`)
  }
};
```

¿Y esto?

```
obj.saludo;
```

redradix

Programación Orientada a Objetos

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: () => {
    console.log(`Hola, ${obj.nombre}`)
  }
};
```

¿Y esto otro?

```
obj.saludo();
```

redradix

Programación Orientada a Objetos

Teniendo:

```
const obj = {
  nombre: 'Homer',
  saludo: () => {
    console.log(`Hola, ${obj.nombre}`)
  }
};
```

¿Es lo mismo?

```
const saludo = obj.saludo;
saludo();
```

redradix

NO

redradix

Programación Orientada a Objetos

```
obj.saludo();
```

1. ***Envía el mensaje*** "*saludo*" a obj
2. Si existe, ***obj se encarga de ejecutar*** la función adecuada
3. obj es el ***receptor***

```
const saludo = obj.saludo;
saludo();
```

1. ***Accede al valor de la propiedad*** "*saludo*" de obj
2. Supongo que es una función y ***la invoco***
3. **NO hay receptor**

redradix

Programación Orientada a Objetos

Cuatro maneras de invocar a una función:

1. **Invocación directa**

redradix

Programación Orientada a Objetos

Cuatro maneras de invocar a una función:

1. Invocación directa
2. **Enviando un mensaje a un objeto (método)**

redradix

Programación Orientada a Objetos

```
const counter = {
  count: 0,
  increment: function() { this.count++; }
};

$('#button').on('click', counter.increment);
```

redradix

Programación Orientada a Objetos

```
global.nombre = 'Fry';

const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};

const fn = obj.saludo;
fn();
```

redradix

Programación Orientada a Objetos

```
global.nombre = 'Fry';

const obj = {
  nombre: 'Homer',
  saludo: function() {
    console.log(`Hola, ${this.nombre}`)
  }
};

obj.saludo();
```

redradix

Programación Orientada a Objetos

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout(function() {
      console.log(`Hola, ${this.nombre}`)
    }, 100);
  }
}

obj.saludo()
```

redradix

Programación Orientada a Objetos

```
function saludo() {
  console.log(`Hola, ${this.nombre}`);
}
```

redradix

Programación Orientada a Objetos

```
function saludo() {
  console.log(`Hola, ${this.nombre}`);
}

const obj1 = {
  nombre: 'Homer'
};

const obj2 = {
  nombre: 'Fry'
};
```

redradix

Programación Orientada a Objetos

```
obj1.saludo = saludo;
obj1.saludo();

obj2.saludo = saludo;
obj2.saludo();
```

redradix

Programación Orientada a Objetos

Cuatro maneras de invocar a una función:

1. Invocación directa
2. Enviando un mensaje a un objeto (método)
3. **Function.prototype**

redradix

Programación Orientada a Objetos

```
fn.call(context, arg1, arg2, …)

fn.apply(context, [arg1, arg2, …])
```

- Ejecutamos la función **fn**
- Especificando el **valor de `this` explícitamente**

redradix

Programación Orientada a Objetos

```
function saludo() {
  console.log(`Hola, ${this.nombre}`);
}

const obj1 = {
  nombre: 'Homer'
};


saludo.call(obj1);
```

redradix

Programación Orientada a Objetos

```
function saludo() {
  console.log(`Hola, ${this.nombre}`);
}

const obj1 = {
  nombre: 'Homer'
};


saludo.apply(obj1);
```

redradix

Programación Orientada a Objetos

```
const a = [1, 2, 3];

[].slice.call(a, 1, 2); // [2]
[].slice.apply(a, [1, 2]); // [2]
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const algo = misterio();

typeof algo; // ???
typeof algo(); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const algo = misterio({}, function() {
  return this;
});

typeof algo(); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = {};
const algo = misterio(obj, function() {
  return this;
});

obj === algo(); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = {};
const algo = misterio({}, function() {
  return this;
});

obj === algo(); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = { nombre: 'Homer' };
const algo = misterio({}, function() {
  return this.nombre;
});

algo(); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const obj = { nombre: 'Homer' };
const algo = misterio({}, function(saludo) {
  return `${saludo}, ${this.nombre}`;
});

algo('Hola'); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const homer = { nombre: 'Homer' };
const fry = { nombre: 'Fry' };

const algo = misterio(homer, function(saludo) {
  return `${saludo}, ${this.nombre}`;
});

algo.call(fry, 'Hola'); // ???
```

redradix

¿Qué hace esta función?

```
function misterio(ctx, fn) {
  return function(...args) {
    return fn.apply(ctx, args);
  }
}

const algo = misterio({}, function() {
  return this;
});

typeof algo(); // ???
```

redradix

Programación Orientada a Objetos

```
function bind(ctx, fn) {
  return function() {
    return fn.apply(ctx, arguments);
  }
}
```

redradix

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout(bind(this, function() {
      console.log(`Hola, ${this.nombre}`)
    }), 100);
  }
}

obj.saludo()
```

redradix

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout((function() {
      console.log(`Hola, ${this.nombre}`)
    }).bind(this), 100);
  }
}

obj.saludo()
```

redradix

Programación Orientada a Objetos

¿Por qué no hemos usado ***arrow functions*** en las últimas diapositivas?

redradix

Programación Orientada a Objetos

- Arrow functions fijan **this** al **contexto de definición**
  - **NO** al contexto de invocación
  - **NO** depende de cómo se invoque la función
  - **NO** es dinámico!

redradix

```
const obj = {
  nombre: 'Homer',
  saludo: function() {
    setTimeout(() => {
      console.log(`Hola, ${this.nombre}`)
    }, 100);
  }
}

obj.saludo()
```

redradix

```
global.nombre = 'Fry';

const obj = {
  nombre: 'Homer',
  saludo: () => {
    console.log(`Hola, ${this.nombre}`);
  }
}

obj.saludo(); // ???
```

redradix

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?

redradix

Clases

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

redradix

Clases

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

redradix

```
class Dog {

  constructor(name) {
      this.name = name;
  }

  bark() {
      console.log("wof, wof...");
  }

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

}
```

redradix

Clases

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

redradix

```
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();
```

redradix

Clases

¿Qué valor nos devuelve esta expresión?

```
typeof Dog
```

redradix

Clases

¿¿Por qué??

Clases

Las clases de ES6 son **azúcar sintáctico**

redradix

Clases

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

redradix

Clases

Herencia de prototipos

redradix

Clases

(pausa dramática)

redradix

```
class Dog {

  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log("wof, wof...");
  }

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

}
```

redradix

```
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.`);
}
```

redradix

```
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.`);
}
```

redradix

```
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.`);
}
```

redradix

~~Clases~~ Constructores

¿Verdadero o falso?

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

redradix

~~Clases~~ Constructores

¿Para qué sirve la propiedad `prototype`?

redradix

~~Clases~~ Constructores

La respuesta está en **new**

redradix

Programación Orientada a Objetos

Cuatro maneras de invocar a una función:

1. Invocación directa
2. Enviando un mensaje a un objeto (método)
3. Function.prototype
4. **new**

redradix

~~Clases~~ Constructores

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

redradix

~~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

redradix

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`**

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

~~Clases~~ Constructores

¿Verdadero o falso?

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

redradix

~~Clases~~ Constructores

¿Verdadero o falso?

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

redradix

```
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();
```

redradix

~~Clases~~ Constructores

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

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

Clases

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

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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();
```

redradix

```
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.');
    // ???
  }
}
```

redradix

```
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();
  }
}
```

redradix

```
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();
  }
}
```

redradix

Herencia

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

redradix

```
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())
```

redradix

Polimorfismo

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

redradix

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

redradix

Polimorfismo

- Cuando invoco a un método...

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

redradix

Polimorfismo

- Cuando invoco a un método...

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

redradix

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)*

redradix

Polimorfismo

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

redradix

Polimorfismo

¿Qué mecanismo usa para encontrar el método adecuado?

redradix

Polimorfismo

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

redradix

Polimorfismo

Cadena de prototipos + **this**

=

late binding y *dynamic dispatch* gratis!

redradix

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);
```

redradix

Polimorfismo

¿Verdadero o falso?

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

redradix

Polimorfismo

¿Verdadero o falso?

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

redradix

Polimorfismo

¿Verdadero o falso?

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

redradix

Polimorfismo

¿Verdadero o falso?

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

redradix

Clases Anónimas

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

redradix

Clases Anónimas

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

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

redradix

Ejercicio

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

redradix

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
```

redradix

Propiedades estáticas

¿Quién es el *receptor* del mensaje?

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

redradix

Propiedades estáticas

En la implementación de **`getInstanceCount`**,

¿A dónde apuntará **`this`**?

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

redradix

Propiedades estáticas

¿Cuál es el *prototipo de* **`CountedMamal`**?

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

redradix

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();
```

redradix

Propiedades estáticas

¿Qué valor me devuelve...?

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

redradix

Propiedades estáticas

¿Qué valor me devuelve...?

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

redradix

Propiedades estáticas

¿Qué valor me devuelve...?

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

redradix

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**

redradix

Propiedades de clase

```
Dog.legs = 4;

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

redradix

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);
```

redradix

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);
```

redradix

Propiedades estáticas

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

redradix

Propiedades estáticas

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

redradix

Decoradores

- Funciones que alteran el comportamiento de métodos
- Habitualmente añaden funcionalidad *alrededor* del método:
  - hacen algo *antes* de la ejecución
  - hacen algo *después* de la ejecución

redradix

Decoradores

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

  @educated
  bark () {
    console.log('Wof, wof!');
  }
}
```

redradix

Decoradores

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

  @educated
  bark () {
    console.log('Wof, wof!');
  }
}
```

redradix

Decoradores

```
function educated(target, key, descriptor) {
  descriptor.value = function() {
    console.log(`* ${this.name} controls itself.`);
  }
  return descriptor;
}
```

redradix

Decoradores

```
function educated(target, key, descriptor) {
  descriptor.value = function() {
    console.log(`* ${this.name} controls itself.`);
  }
  return descriptor;
}
```

redradix

Decoradores

```
function educated(target, key, descriptor) {
  descriptor.value = function() {
    console.log(`* ${this.name} controls itself.`);
  }
  return descriptor;
}
```

redradix

Decoradores

```
function educated(target, key, descriptor) {
  descriptor.value = function() {
    console.log(`* ${this.name} controls itself.`);
  }
  return descriptor;
}
```

redradix

Decoradores

```
{
  value: [Function],
  writable: true,
  enumerable: false,
  configurable: true
}
```

redradix

Decoradores

¿Cómo podríamos escribir un decorador que proteja un método contra sobreescritura?

Decoradores

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

  @readonly
  bark() {
    console.log('Wof, wof!');
  }
}

const spot = new Dog('Spot');
spot.bark = () => console.log('Guau..?');

spot.bark() // -> Wof, wof!
```

redradix

Decoradores

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

  @readonly
  bark() {
    console.log('Wof, wof!');
  }
}

const spot = new Dog('Spot');
spot.bark = () => console.log('Guau..?');

spot.bark() // -> Wof, wof!
```

redradix

Decoradores

```
function readonly(target, key, descriptor) {
  descriptor.writable = false;
  return descriptor;
}
```

redradix

Decoradores

- Los decoradores son **funciones** que **alteran el comportamiento** de un método
- Reciben tres parámetros:
  - la *clase* en la que está el método
  - el *nombre* del método
  - el *descriptor* de la propiedad

redradix

Decoradores

- Un decorador también puede ser una **función** que **devuelva una función**
- Nos permite hacer decoradores más potentes

redradix

Decoradores

```
function prevent(msg) {
  return (target, key, descriptor) => {
    descriptor.value = () => console.log(msg);
    return descriptor;
  }
}
```

redradix

Decoradores

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

  @prevent('Do not disturb the dog!')
  bark() {
    console.log('Wof, wof!');
  }
}

const spot = new Dog('Spot');
spot.bark() // -> Do not disturb the dog!
```

redradix

Decoradores

- Los decoradores *no son parte oficial de ES6* todavía
- Están en *stage-2*
- Son una herramienta muy conveniente!

redradix

Ejercicios

- Escribe un decorador **`@deprecated`**
  - Que muestre un aviso de obsolescencia cuando se invoca al método
  - Modifícalo para que reciba, opcionalmente, un mensaje extra
    - **`@deprecated('do not use this')`**

redradix

Ejercicios

- Escribe un decorador **`@trace`**
  - Muestra por la consola info cada vez que se invoca al método
  - Parámetros
  - Valor de retorno

redradix

Ejercicios

- Escribe un decorador **`@debug`**
  - Ejecuta el método si **`window.debug`** es true
  - En caso contrario, omite la ejecución

redradix

Ejercicios

- Escribe un decorador **`@catch`**
  - Captura excepciones que lance el método
  - Las loguea por la consola
  - si **`window.debug`** es true, relanza la excepción
  - si **`window.debug`** es false, continúa la ejecución

redradix

Ejercicios

- Escribe un decorador **`@benchmark`**
  - Loguea el tiempo que tarda cada ejecución del método

redradix

Decoradores

- Hay *muchos* usos interesantes para los decoradores
- Se pueden apilar!

redradix

Decoradores

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

  @trace
  @debug
  @benchmark
  bark() {
    console.log('Wof, wof!');
  }
}
```

redradix

Decoradores

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

  @trace
  @debug
  @benchmark
  bark() {
    console.log('Wof, wof!');
  }
}
```

redradix

Decoradores

- También se puede decorar **la clase completa**

```
@vaccinate
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log('Wof, wof!');
  }
}
```

redradix

Decoradores

- También se puede decorar **la clase completa**

```
function vaccinate(Constructor) {
  Constructor.prototype.vaccined = true;
  return Constructor;
}
```

redradix

Ejercicios

- Escribe un decorador de clase **`@withCount`**
  - Con el mismo comportamiento que la versión que hicimos unas cuantas diapositivas atrás

redradix

Ejercicios

- Escribe un decorador de clase **`@bind('metodo', 'metodo2')`**
  - Fije el contexto de los métodos que recibe como parámetros

redradix

Ejercicios

```
@bind('bark')
class Dog {
  constructor(name) {
    this.name = name;
  }

  bark() {
    console.log(this.name, 'says: wof, wof!');
  }
}

const spot = new Dog('Spot');
const toi = new Dog('Toi');
toi.bark.call(spot) // -> Toi says: wof, wof!
```

redradix

Ejercicios

- Escribe un decorador de método **`@bindMethod`**
  - Que fije el contexto del método al que se aplica

redradix

Ejercicios

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

  @bindMethod
  bark() {
    console.log(this.name, 'says: wof, wof!');
  }
}

const spot = new Dog('Spot');
const toi = new Dog('Toi');
toi.bark.call(spot) // -> Toi says: wof, wof!
```

redradix