Decoradores

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

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!');
  }
}
```

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;
}
```

Decoradores

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

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!
```

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!
```

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

Decoradores

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

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!
```

Decoradores

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

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')`**

Ejercicios

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

Ejercicios

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

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

Ejercicios

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

Decoradores

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

Decoradores

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

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

Decoradores

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

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

Decoradores

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

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

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

Decoradores

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

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

Ejercicios

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

Ejercicios

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

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!
```

Ejercicios

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

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!
```