Módulo 11 - Clases y this - Teoría

La necesidad

Hasta ahora estamos acostumbrados a escribir variables y funciones en un fichero.

Imaginemos que tenemos un centro comercial y queremos calcular el total de las ventas del área de perfumería (aplicamos un 10% de descuento al subtotal de la compra).

```
const subtotal = 30;
const descuento = 10;

const calculaTotal = (subtotal, descuento) => {
  return subtotal - (descuento * subtotal) / 100;
};

console.log(calculaTotal(subtotal, descuento));
```

Codepen

¿Qué pasa si ahora quiero calcular el subtotal de otros productos? Por ejemplo, del área de supermercado (hacemos un 20 % de descuento en este caso), podríamos escribir algo así como:

```
const subtotal = 50;
const descuento = 20;

const calculaTotal = (subtotal, descuento) => {
  return subtotal - (descuento * subtotal) / 100;
};

console.log(calculaTotal(subtotal, descuento));
```

Codepen

Todo bien, vamos a unir el código en un sólo fichero:

```
const subtotal = 30;
const descuento = 10;

const calculaTotal = (subtotal, descuento) => {
  return subtotal - (descuento * subtotal) / 100;
};

console.log(calculaTotal(subtotal, descuento));

const subtotal = 50;
const descuento = 20;

const calculaTotal = (subtotal, descuento) => {
  return subtotal - (descuento * subtotal) / 100;
};

console.log(calculaTotal(subtotal, descuento));
```

Code pen

¿Qué pasa ahora? Vaya, tenemos conflictos de nombres, por un lado las variables *subtotal* y *descuento* se repiten, y por otro la función *calculaDescuento* también.

"Uncaught SyntaxError: Identifier 'subtotal' has already been declared"

¿Qué podríamos hacer? Podemos tomar varias aproximaciones:

- Una sería: renombrar las variables y métodos, y añadirles un sufijo por departamento.
- Otra sería: romper en ficheros distintos (módulos) y vía *imports* importarnos las funciones que toquen.

Y no sería interesante definir un comportamiento y poder tener "copias" separadas del mismo? Es decir algo así como:

- **Clase:** Definimos las reglas de lo que queremos implementar.
- **Instancia**: Podemos tener múltiples islas en las que tenemos nuestros datos, pero nos guíamos por las reglas y estructuras definidas en las clases.

Nuestra primera clase

¿Cómo podríamos pasar el ejemplo anterior a clases e instancias? Primero vamos a ver qué queremos encapsular en una clase.

A nivel de datos:

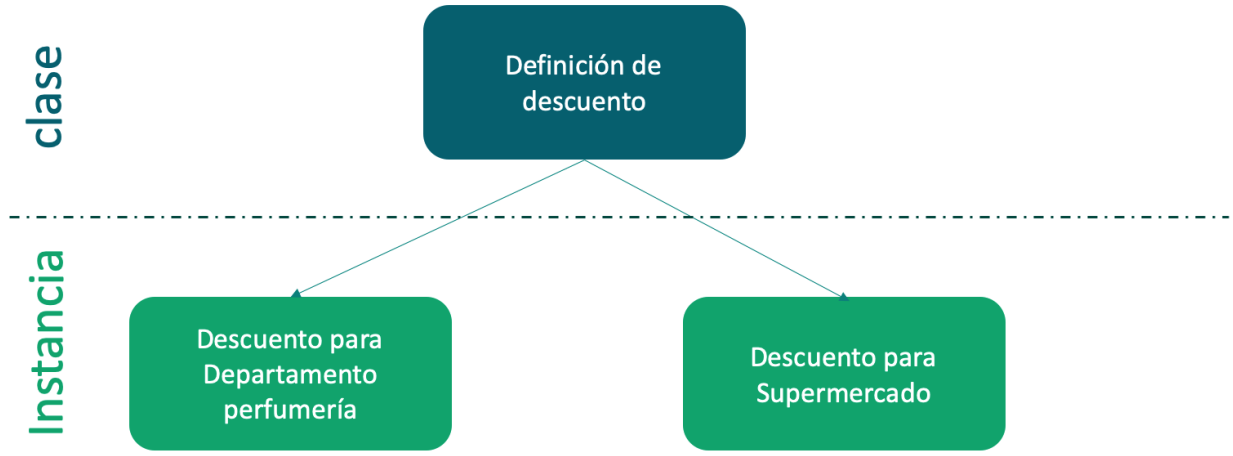
- El valor de subtotal puede ser un buen candidato.
- El valor de descuento podríamos también meterlo en esta clase.
- El valor total podría ser interesante guardarlo en la clase también.

A nivel de métodos (funciones que va a exponer la clases):

- Podríamos añadir un método para calcular el descuento.
- Podríamos añadir un método para definir el subtotal.

Para ello podemos definir una clase.

> Una clase en un tipo de función que en vez de usar la palabra clave *function*, utiliza la palabra clave *class*, y las propiedades (variables) se asignan dentro de un método *constructor*.



Constructor

Empezamos por los *datos*

```
class TotalesDepartamento {
  constructor() {
    this.descuento = 10;
    this.subtotal = 0;
    this.total = 0;
  }
}
```

A destacar, cuando tenemos propiedades las *prefijamos* con la palabra reservada *this ¿Esto por qué?* Porque nos estamos refiriendo a la propiedad de la instancia con la que estemos trabajando (perfumería o supermercado).

*Interesante... aquí tenemos estos datos como **propiedades**, pero hemos harcodeado valores. ¿Cómo podría definir valores dependiendo de la instancia que esté levantando (perfumería o supermercado)? Esto lo podemos hacer añadiendo parámetros al _constructor*:

```
class TotalesDepartamento {
-  constructor() {
    this.descuento = 10;
+  constructor(descuento) {
+    this.descuento = descuento;
    this.subtotal = 0;
    this.total = 0;
  }
}
```

Métodos de clase

Vamos ahora añadir el método que calcula el total de la compra:

```
class TotalesDepartamento {
  constructor(descuento) {
    this.descuento = descuento;
    this.subtotal = 0;
    this.total = 0;
  }
}

+ calculaTotal() {
+  this.total =  this.subtotal - (this.descuento * this.subtotal) / 100;
+ };
}
```

Fijate, un método es como una función, sólo que está dentro de una clase, y puede acceder a las propiedades y métodos de las clases (usando el prefijo **this**).

El código que nos queda:

```
class TotalesDepartamento {
  constructor(descuento) {
    this.descuento = descuento;
    this.subtotal = 0;
    this.total = 0;
  }

  calculaTotal() {
    this.total = this.subtotal - (this.descuento * this.subtotal) / 100;
  }
}
```

Codepen

> Un método puede aceptar parámetros de entrada y también puede devolver valores (como en una función).

Instanciando la clase

Ahora que tenemos lo básico (más adelante añadiremos mejoras). Vamos a crear una instancia de clase para manejar los descuento de perfumería, y otra para aplicar los descuento de supermercado:

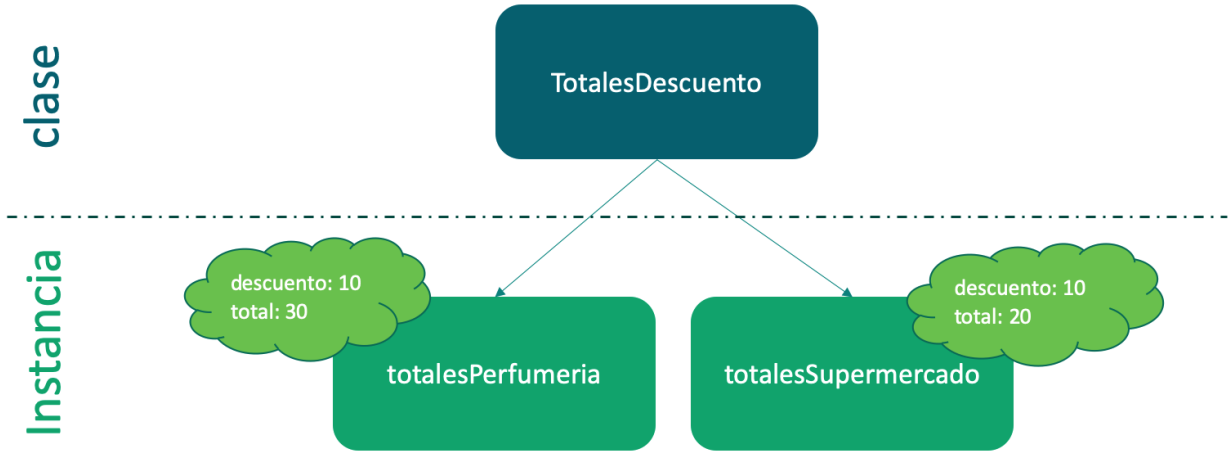
```
const totalesPerfumeria = new TotalesDepartamento(10);
const totalesSupermercado = new TotalesDepartamento(20);
```

Ahora podemos introducir el total y calcular el descuento:

```
totalesPerfumeria.subtotal = 30;
totalesPerfumeria.calculaTotal();
console.log("Total Perfumería: ", totalesPerfumeria.total);

totalesSupermercado.subtotal = 50;
totalesSupermercado.calculaTotal();
console.log("Total Supermercado: ", totalesSupermercado.total);
```

Fijate aquí en cómo diferenciamos los valores de perfumería y los de supermercado. Son instancias diferentes.



El ejemplo completo:

```
class TotalesDepartamento {
  constructor(descuento) {
    this.descuento = descuento;
    this.subtotal = 0;
    this.total = 0;
  }

  calculaTotal() {
    this.total = this.subtotal - (this.descuento * this.subtotal) / 100;
  }
}

const totalesPerfumeria = new TotalesDepartamento(10);
const totalesSupermercado = new TotalesDepartamento(20);

totalesPerfumeria.subtotal = 30;
totalesPerfumeria.calculaTotal();
console.log("Total Perfumería: ", totalesPerfumeria.total);

totalesSupermercado.subtotal = 50;
totalesSupermercado.calculaTotal();
console.log("Total Supermercado: ", totalesSupermercado.total);
```

Codepen

Getters y Setters

Vamos a mejorar este código, ahora mismo cada vez que establecemos un subtotal, tenemos que manualmente llamar a *calculaTotal* para que calcule e introduzca el valor de `total` en la propiedad *total*, ... peroooo *si esta propiedad total se tiene que calcular cada vez que cambie un total ¿No podríamos calcularla al vuelo?* ¡Bingo! podemos usar campos calculados.

Vamos a reemplazar la propiedad *subtotal* por un *getter* es decir de cara al usuario, está pidiendo una propiedad, de cara a implementación de la clase lo que tenemos es un método que calcula el valor cada vez que lo pedimos:

```
class TotalesDepartamento {
  constructor(descuento) {
```

```
    this.descuento = descuento;
    this.subtotal = 0;
-    this.total = 0;
  }

+  get total() {
+    return this.subtotal - (this.descuento * this.subtotal) / 100;
+  }

-  calculaTotal() {
-   this.total =  this.subtotal - (this.descuento * this.subtotal) / 100;
-  };
}

+ const totalesPerfumeria = new TotalesDepartamento(10);
+ totalesPerfumeria.subtotal = 30;
+ console.log("Total Perfumería: ", totalesPerfumeria.total);
```

Codepen

Esta opción es interesante, pero *¿No podría pasar que si llamamos cada dos por tres al getter total estamos gastando ciclos de cpu calculando una y otra vez lo mismo?* Vamos a darle la vuelta a la tortilla, ya que el elemento que dispara el calculo del *total* es la propiedad *subtotal ¿Por qué no hacer ése calculo cuando se setea el valor de total?* Vamos a crear un *setter*.

```
class TotalesDepartamento {
  constructor(descuento) {
    this.descuento = descuento;
-    this.subtotal = 0;
+    this._subtotal = 0;
+    this.total = 0;
  }

+ set subtotal(nuevoValor) {
+   this._subtotal = nuevoValor;
+   this.total = this._subtotal - (this.descuento * this._subtotal) / 100;
+ }


-  get total() {
-    return this.subtotal - (this.descuento * this.subtotal) / 100;
-  }
}

const totalesPerfumeria = new TotalesDepartamento(10);
totalesPerfumeria.subtotal = 30;
console.log("Total Perfumería: ", totalesPerfumeria.total);
```

Codepen

> Fijate que en hemos puesto un `_` a la propiedad subtotal, es una manera *amistosa* de decir que esa propiedad es privada. En las clases de ES6, podemos acceder desde el exterior a a todas las propiedades y métodos de una clase, recientemente se añadió el prefijo # para hacer métodos privados https://www.sitepoint.com/javascript-private-class-fields/

Ojo con el this

Una de las principales pegas del manejo de clases en Javascript es el uso de *this*.

Vamos a simular que hacemos una llamada asíncrona utilizando set timeout: en este caso, tenemos una propiedad miembro `descuento`. Al precio de la ficha de producto que nos "viene de servidor" (estamos simulando) le aplicamos el descuento de la propiedad miembro:

```
class PreciosAPI {
  constructor() {
    this.descuento = 0.8;
  }
```

```
  cargaPrecioDeServidor() {
    setTimeout(function() {
      const precio = 2; // <- simulando precio desde el servidor

      console.log(precio * this.descuento);
    });
  }
}

const preciosAPI = new PreciosAPI();
preciosAPI.cargaPrecioDeServidor();
```

CodePen

Si ejecutamos esto vemos que por consola nos sale 'NaN', si investigamos un poco más (depurando el código), podemos ver que *this* vale *undefined* :-@ ¿Por qué? En un lenguaje normal, *this* está enlazado a la instancia del objeto donde se creo, en Javascript se define por el contexto de ejecución: en este caso es *setTimeout* que pertenece al objeto *Window* global el que está realizando la llamada, por lo tanto *this* apunta a *Window*, *un momento... El this apunta a undefined no al objeto Window ¿Qué está pasando?* Eso viene porque estamos trabajando en modo estricto.

¿Cómo podemos arreglar esto? Podemos usar varias aproximaciones, dos de ellas:

Explícitamente especificándole que toma el `this` de la clase (bind this)

```
class PreciosAPI {
  constructor() {
    this.descuento = 0.80;
  }

  cargaPrecioDeServidor() {
    setTimeout(function() {
      const precio = 2;

      console.log(precio * this.descuento);
+    }.bind(this))
-    })
  }
}

const preciosAPI = new PreciosAPI();
preciosAPI.cargaPrecioDeServidor();
```

Codepen

Definiendo la función como una fat arrow o función flecha (aquí toma el `this` en tiempo de interpretación, no de ejecución):

```
class PreciosAPI {
  constructor() {
    this.descuento = 0.80;
  }

  cargaPrecioDeServidor() {
-    setTimeout(function() {
+    setTimeout(() => {
      const precio = 2;

      console.log(precio * this.descuento);
    })
  }
}

const preciosAPI = new PreciosAPI();
preciosAPI.cargaPrecioDeServidor();
```

Codepen

Por simple curiosidad y para que os suene cuando lo veáis, a veces veremos código legacy que asigna el valor de `this` dentro de una variable por fuera de la función interna para poder usarla.

```
class PreciosAPI {
  constructor() {
    this.descuento = 0.8;
  }

  cargaPrecioDeServidor() {
    const self = this;

    setTimeout(function() {
      const precio = 2;

      console.log(precio * self.descuento);
    });
  }
}

const preciosAPI = new PreciosAPI();
preciosAPI.cargaPrecioDeServidor();
```

Codepen

Heredando una clase

Un tema muy interesante que ofrecen las clases es la herencia. *¿Esto qué es?* Imagínate que tienes un clase y una parte de la misma podrías aprovecharla en otras clases, es decir, tienes una base común de código.

¿Qué podrías hacer para evitar tener que ir copiando y pegando código de un sitio a otro?

- Una opción que tienes es la composición, es decir sacas a funciones comunes lo que tengas hecho, pero esto puede ser un poco lío si quieres tener propiedades comunes.
- La segunda opción es la herencia: creamos una clase base con lo común, y creamos clases que heredan de la padre en el que implementamos el comportamiento específico.

Vamos a ver como crear clases bases y heredar de ellas, para la teoría usaremos un ejemplo básico, en la práctica lo aplicaremos a un caso real.

Vamos a crear nuestra clase Animal:

```
class Animal {
  constructor(nombre, piernas, ruido) {
    this.tipo = "animal";
    this.nombre = nombre;
    this.piernas = piernas;
    this.ruido = ruido;
  }

  habla() {
    console.log(`${this.nombre} dice ${this.ruido}`);
  }

  anda() {
    console.log(`${this.name} camina con ${this.piernas} piernas`);
  }

  set cazacomida(comida) {
    this.comida = comida;
  }

  get come() {
    return `${this.nombre} se come ${this.comida || "nada grouaun"}`;
  }
}
```

Vamos a probar a instanciar esta clase:

```
const laika = new Animal("Laika", 4, "woff");
laika.cazacomida = "huesos";
console.log(laika.come);
```

Codepen

Vamos a meter un nuevo animal en nuestra casa, en esta caso un Loro. El loro tiene dos patas, come pipas, pero ¡ojo! cuando le pides que hable, además de emitir un ruido, te dice después tu nombre. ¿Qué tenemos?

- Un loro cumple con casi todo el comportamiento de nuestra clase animal.
- Tiene sólo una diferencia, y es que en el método "habla" además de ejecutar el código base, añade otra línea en la que dice el nombre del amo.

¿Cómo podemos hacer para evitar tener que copiar y pegar todo el código que ya tenemos común? ¡Herencia al rescate!

- Por un lado creamos una clase *Loro* que hereda de *Animal*.
- En el constructor de la clase ponemos los parámetros que necesitemos (varios de la clase padres los inferimos nosotros el tipo de animal es un loro tiene 2 piernas y hace *curruuukak*,), y llamamos al constructor de la clase utilizando *super*.
- Por otro lado sobreescribimos el método *habla*.
- En este método *habla* ejecutamos el *habla* del padre usando *super*.
- Después lo ejecutamos.

Añadimos este código al ya existente

```
class Loro extends Animal {
  constructor(nombre) {
    // Llamamos al constructor del padre usando super
    super(nombre, 2, "Curuuukaaaak !!!");
    this.tipo = "loro";
  }

  // Sobreescribimos habla
  habla(amo) {
    super.habla();
    console.log(`Dame chocolatinas ${amo}`);
  }
}
```

Vamos a probarlo:

```
console.log("*** Vamos a por el loro ****");
const paco = new Loro("Paco");
paco.cazacomida = "pipas";
console.log(paco.come);
paco.habla("Lissette");
```

Más material

- Tutorial clases básicos y campos privados: https://www.sitepoint.com/javascript-private-class-fields/
- Método estáticos (no está cubierto en esta introducción): https://medium.com/@yyang0903/static-objects-static-methods-in-es6-1c026dbb8bb1