INTRODUCCIÓN A LA ORIENTADA A OBJETOS EN JS

La programación orientada a objetos es un paradigma de la programación en el que se crean objetos para la manipulacón de datos y donde, por lo general, cada objeto ofrece una funcionalidad especial.

Con la salida de ES6, se agregó el concepto de clases en JavaScript, ¿esto quiere decir que antes de ES6 no existían clases?. Básicamente no, no existían clases antes de ES6, al menos no semánticamente hablando, y para explicar esto necesitamos analizar un poco el concepto de Clases dentro de la POO.

Según wikipedia, una clase es "*es una plantilla para la creación de objetos de datos según un modelo predefinido.*". Si tomamos en cuenta esta definición, esto era posible de lograr antes de ES6, a través de funciones usadas como constructores, y luego ejecutadas con el operador *new*, que crea una nueva instancia tomando como plantilla el constructor. Por ejemplo:

```
function User( name, lastName ){
    this.name = name;
    this.lastName = lastName;
}
let user1 = new User( 'Jose', 'Paredes' );
let user2 = new User( 'Pablo', 'Pinto' );
```

Entonces, de esta manera, cumplimos con la definición de *clase*, sin tener nada semánticamente hablando que nos diga que se comporta como una clase. Y es aquí donde vienen los cambios de ES6, haciendo que semántica y sintácticamente tengan un poco mas sentido las clases.

Clases

Teniendo en cuenta que era posible crear clases antes de ES5, queda la incógnita de cuales fueron esos cambios que ayudaron a la semántica y a la sintaxis al momento de querer crear clases en JavaScrpit
En ES6, la manera de crear una clase, es la siguiente:

```
class User {
}
let user = new User()
```

A simple vista se nota que va mucho más acorde a la creación de clases de otros lenguajes de programación, haciendo así que sea mucho mas semántico a la hora de definir clases.

Hasta este punto podemos notar lo siguiente:

- Se introduce la primera palabra reservada de ES6 *class* para hacer la declaración de la clase.
- Por convención, el nombre de las clases se escribe en PascalCase
- Si la clase es creada con *class*, esta no puede ser llamada como una función: *User()*.
- Las clases no obedecen al *hoisting*, por lo tanto, no puede ser usada antes de su declaración.
- De manera implícita las clases se comportan como constantes, es decir, no puede redeclararse bajo un mismo ámbito.
- La forma de instanciar una clase es a través de *new*.

¿Que se puede usar dentro de una clase?

Utilizar las clases con ES6, no solo te ayudará con la parte sintáctica y semántica, sino también, te aportará otras herramientas para poder crear una clase con todo lo necesario.

Constructor

En el ejemplo de la sintaxis de ES6 para clases, creamos una clase vacía. Pero ¿que sucede si necesitamos recibir por parámetros los datos necesarios para instanciar una clase? Es aquí donde entra el *constructor*, que es el encargado de inicializar las instancias de una clase.
Evolucionando un poco el primer ejemplo, resulta lo siguiente:

```
class User {
    Constructor( name, lastName ) {
        this.name = name;
        this.lastName = lastName;
    }
}
let user1 = new User( 'Jose', 'Paredes' );
let user2 = new User( 'Pablo', 'Pinto' );
```

Setters y Getters

Otra de las herramientas atribuidas a las con ES6 son los setters y getters, que son de alguna manera, variaciones de una función, que te permiten cambiar valores y obtener valores.

La forma de definirlos es la siguiente:

```
class Car {
  Constructor( model='No tiene modelo' ) {
    this.model = model;
  }
  get model() {
     console.log( 'GETTER' )
     return this.model;
  }
  set model( value ) {
    console.log( 'SETTER' )
    this.model = value;
  }
}
let car = new Car  // Esto mostrará un montón de 'SETTER' y al final dirá esto:
                   // RangeError: Maximum call stack size exceeded
```

La forma de entrar a las funciones setters y getters es a través de la asignación o consulta del atributo del cual existen los setter y getters, en este caso, ese atributo es llamado model. Es decir, para que salga el *console.log* de la linea 12, que pertenece al setter de model, solo basta con escribir lo siguiente: *car.model = 'Fortuner'*, siendo *value* la nueva asignación dada a model ('Fortuner'). Por otro lado, para que salga el *console.log* de la linea 7, que pertene al getter de model, solo hay que consultar ese atributo, es decir: *car.model.*

El error de RangeError es dado porque al hacer *new Car,* entra a su respectivo constructor que en la linea 3 tiene *this.model = model;* eso hace que entre al setter *set model* que en la linea 13 tiene *this.model = value;* y por ser asignación, vuelve a entrar al setter *set model*, haciendo así que entre en un loop infinito.

Una de las formas para solucionar este problema es la siguiente:

```
class Car {
    Constructor( model = 'No tiene modelo' ) {
        this._model = model;
    }
    get model() {
        console.log( 'GETTEEEEEER' )
        return this._model;
    }
    set model( value ) {
        console.log( 'SETTEEEEEER' )
        this._model = value;
    }
}
let car = new Car;
car.model
// GETTEEEEEER
car.model = 'Fortuner'
// SETTEEEEEER
```

Cambiando el atributo *model* a *_model*, y manteniendo el nombre en las funciones de los getters y setters, al asignar (*car.model = 'Fortuner'*) entra al setter de model y lo que hace es asignar el nuevo valor al atributo *_model*, haciendo así que no entre en un loop infinito porque el atributo *_model* no tiene un setter.

Un ejemplo completo usando la clase User del inicio, sería el siguiente:

```
class User {
    Constructor( name, lastName ) {
        this._name = name;
        this._lastName = lastName;
    }
    get name() {
        return this._name
    }
    set name(value) {
        this._name = value;
    }
    get lastName() {
        return this._lastName
    }
```

```
    set lastName(value) {
        this._lastName = value;
    }
    get fullName() {
        return `${this.name} ${this.lastName}`;
    }
}
let user1 = new User( 'Jose', 'Paredes' );
user1.fullName
// Jose Paredes
let user2 = new User( 'Pablo', 'Pinto' );
user2.fullName
// Pablo Pinto
```

Métodos estáticos

Los métodos estáticos nos son mas que aquellos métodos que solo pueden ser ejecutados desde la clase y no desde una instancia. Por lo general se usar para cosas que tengan un aporte a la clase como tal y que no varia según la instancia.

```
class StaticMethodCall {
  static staticMethod() {
    return 'Haciendo algo desde un método estático';
  }
}

StaticMethodCall.staticMethod()
// Haciendo algo desde un método estático

let staticMethodCall = new StaticMethodCall
staticMethodCall.staticMethod()
// TypeError: staticMethodCall.staticMethod is not a function
```

Métodos públicos

Los métodos públicos son todos aquellos métodos que se usan para hacer procedimientos y que no necesitas que devuelvan algo en específico. Estos métodos son llamados desde las instancias.

```
class PublicMethodCall {
  publicMethod() {
    console.log( 'empezando algún procedimiento desde un método público' );
    console.log( 'terminando algún procedimiento desde un método público' );
  }
}

let publicMethodCall = new PublicethodCall
publicMethodCall.publicMethod()
// empezando algún procedimiento desde un método público
// terminando algún procedimiento desde un método público
```

Los métodos públicos son escritos con la sintaxis de funciones comunes en JavaScript *function publicMethod() {}* pero sin la palabra reservada *function* y para la ejecución de la misma se llama como un atributo de la instancia con sus respectivos paréntesis al final para su ejecución: *instance.publicMethod()*