**M1S7: Objetos. Clases.**

**¡Hola! Espero te encuentres muy bien, hasta este momento hemos visto: HTML, CSS y algunos conceptos relacionados con JavaScript, incluyendo sus sintaxis y sentencias, esta semana estaremos viendo lo relacionado con objetos, clases e iniciaremos con conceptos básicos de la programación orientada a objetos (POO).**

**Recuerda que, si presentas cualquier duda puedes consultar con tus coaches. Sigamos... 😄**

**ÍNDICE**

* [Objetos](https://github.com/U-Camp/BOOT-M1-SEM7#objetos)
* [Clases](https://github.com/U-Camp/BOOT-M1-SEM7#clases)
* [Programación Orientada a Objetos (POO): Principios Básicos](https://github.com/U-Camp/BOOT-M1-SEM7" \l "programaci%C3%B3n-orientada-a-objetos-poo-principios-b%C3%A1sicos)

**Objetos**

Los objetos son un tipo de dato no primitivo.

Pueden contener múltiples datos, bajo la estructura de propiedad - valor (key-value).

Los valores establecidos en las propiedades pueden ser otros tipos de datos o funciones (en este caso, se conocerán como métodos).

Bajo este concepto, hay dos formas de crear un objeto:

1. **Objetos literales**, también conocidos como Object literals. En este sentido, necesitamos asignar a través de llaves {} el objeto a una variable.

Dentro, establecemos una propiedad y establecemos el valor de esa propiedad. La sintaxis sería de esta forma:

let nombreDelObjeto = {

propiedad1: "valor1",

propiedad2: "valor2",

propiedad3: "valor3"

}

Veamos un ejemplo, con dos propiedades y un método:

let country = {

name: "México",

language: "Español",

getPopulation: function(){

return `En ${this.name}, se habita 127.6 millones`

}

}

// console.log(country)

// console.log(country.name)

// console.log(country.language)

// console.log(country.getPopulation())

Observaciones ante este objeto:

* Contamos con las propiedades name y language.
* Vemos que existe un método llamado getPopulation. Este permite involucrar una función como valor, el cual puede llamarse desde el objeto a través de notación de punto (dot notation).
* Ejecuta los diferentes console.log()para observar su comportamiento.

1. **Constructores**, conocidos como Object constructors. Usaremos la palabra new. Puedes agregar propiedades y métodos a través de la técnica de notación de punto (dot notation).

let nombreObjeto = new Object()

nombreObjeto.nombre = "Joe"

nombreObjeto.apellido = "Doe"

nombreObjeto.edad = 35

nombreObjeto.saludar = function(){

return `Hola. Un gusto.`

}

Toma en cuenta que puedes modificar una propiedad utilizando corchetes [ ] (*brackets*).

Por ejemplo:

nombreObjeto["apellido"] = "Smith"

// console.log(nombreObjeto.apellido)

Ahora bien, profundicemos en los accesos de propiedades.

Como hemos visto anteriormente, puedes acceder a través de dot notation o bracket notation.

let user = {

nickname: "johndoe",

email: "johndoe@gmail.com"

}

console.log(user.nickname) // DOT NOTATION

console.log(user["email"]) // BRACKET NOTATION

Recuerda colocar en bracket notation las comillas para que puedas acceder sin problema a la propiedad.

**Clases**

**JavaScript**, es un lenguaje de programación orientado a prototipos, por lo que se puede decir que todo en este lenguaje es un prototipo, incluyendo las variables/constantes, funciones y cualquier otra estructura que permita manejar datos.

Antes de ECMAScript 2015, no existían las clases como tal en JavaScript, pero a partir de ese estándar, se introdujeron mejoras sintácticas sobre la herencia basada en prototipos de Javascript, por lo que se agregó la palabra reservada class para definir una clase, con lo que se provee una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

Internamente para JavaScript, las clases son "funciones especiales", y así como se pueden definir funciones nombradas y funciones anónimas, de la misma manera las clases se pueden definir como **declaraciones de clases** (nombradas y la forma más usada para representarlas) o como **expresiones de clases** (anónimas asignadas a una variable, no es muy usada esta sintaxis).

Para definir una clase se usa la palabra reservada class seguida del nombre que se le quiera dar (si ésta es nombrada).

El contenido de una clase es la parte que se encuentra entre las llaves {}. Este es el lugar donde se definen las propiedades de la clase, así como el constructor y los métodos de la clase.

El constructor es un método especial para crear e inicializar un objeto creado con una clase. Sólo puede haber un solo método con el nombre constructor en una clase. Si ésta contiene más de un método constructor, se lanzará un error. Así mismo, un constructor puede usar la palabra reservada super, con la cual, si la clase actual hereda de una clase padre, se manda a llamar el constructor del padre.

Así mismo, la palabra reservada extends es usada en declaraciones de clase o expresiones de clase para crear una clase hija que extiende o hereda de una clase padre.

// Declaración de Clase - Clase Nombrada\*

class Rectangulo {

constructor(alto, ancho) {

// Con la palabra `this` se hace referencia al ámbito de la clase

// Las propiedades `alto` y `ancho` representan el estado de la clase

this.alto = alto;

this.ancho = ancho;

}

// Getter (encapsulamiento)

get area() {

return this.calcularArea();

}

// Método, comportamiento de la clase

calcularArea () {

return this.alto \* this.ancho;

}

};

// Expresión de clase - Clase Anónima

const RectanguloAnonimo = class {

constructor(alto, ancho) {

this.alto = alto;

this.ancho = ancho;

}

};

// Instanciamos la clase...

const cuadrado = new Rectangulo(10, 10);

// Imprime: `Area: 100`

console.log(`Area: ${cuadrado.area}`);

// Clase padre

class Animal {

constructor(nombre) {

this.nombre = nombre;

}

hablar() {

console.log(this.nombre + ' hace un ruido.');

}

}

const animal = new Animal('Jirafa');

// Imprime: `Animal dice: Jirafa hace un ruido`

console.log(`Animal dice: ${animal.hablar()}`);

// Clase hija, hereda de la clase `Animal`

class Perro extends Animal {

constructor(nombre) {

super(nombre);

}

// Sobrecarga del método `hablar`, polimorfismo

hablar() {

console.log(this.nombre + ' ladra.');

}

}

const perro = new Perro('Firulais');

// Imprime: `Perro dice: Firulais ladra`

console.log(`Perro dice: ${perro.hablar()}`);

*Ejemplo de clases en JavaScript*

**En caso de que desees consultar y descargar una infografía sobre clases, puedes [presionar aquí.](https://github.com/U-Camp/BOOT-M1-SEM7/blob/main/infografias/M1_S7_Infografia%2001.pdf)**

**También te comento que tu salud es muy importante, por lo que te recomiendo que hagas pausas activas, vayas al baño y comas algo, el cuerpo necesita mucha energía para poder estudiar y aprender. 🍝**

**Si lograste hacer tus pausas activas, seguimos...**

**Programación Orientada a Objetos (POO): Principios básicos**

La Programación Orientada a Objetos da características que otorgan a los objetos propiedades que guardan datos así como de métodos que realizan acciones y procesan esos datos, pero para poder hacer uso de las mismas, es necesario definir esas propiedades y comportamientos en un concepto llamado Clase, básicamente una clase es una plantilla de código que sirve para instanciar objetos, esto es crear objetos en memoria, la cual debe seguir los principios básicos de la Programación Orientada a Objetos, los cuales son los siguientes:

* **Encapsulamiento**: Este principio indica que hay que mantener el estado de un objeto privado, esto quiere decir que sólo el objeto pueda modificar características internas, pongamos de ejemplo la historia de un gato y su usuario; el usuario puede alimentar al gato, pero no puede decidir qué tan hambriento se siente; así mismo, en la POO un objeto no puede comunicarse o alcanzar métodos o propiedades privados de otro objeto.
* **Abstracción**: Al aplicar el encapsulamiento, sólo unos cuantos métodos quedan accesibles externamente, lo que genera una interfaz de comunicación, esto quiere decir que un objeto no tiene porqué saber cómo está construido otro objeto o cómo funciona por dentro, sólo tiene que saber cómo comunicarse con él; tal cual como se interactúa con los objetos reales, muchas veces no se sabe cómo están construidos o cómo funcionan, el objeto con el que se interactúa expone una interfaz con la que se pueden realizar acciones en él. Si un usuario acaricia un gato y el gato como reacción empieza a ronronear, el usuario no tiene porqué saber cómo es que el gato ronronea, sólo interactúa a través del pelo y la piel del gato.
* **Herencia**: Muchas veces los objetos son bastantes similares, pueden compartir una lógica común, pero no siempre es exactamente lo mismo; la herencia permite el re-uso del código al crear una clase hija con base en una clase padre, heredando las propiedades y métodos de la clase padre, haciendo posible que se escriban en la hija nuevas propiedades y métodos específicos de la clase hija. Por ejemplo, el gato es un animal y comparte algunas propiedades y acciones de otros animales, como el alimentarse, las cuales se heredan de su clase "animal", pero tiene otros atributos y comportamientos específicos de los gatos, como el maullar.
* **Poliformismo**: En el ejemplo anterior, la acción de maullar también podría tomarse como una herencia de la clase "felino", ya que la mayoría de los felinos realizan exclamaciones con la boca, pero los gatos específicamente maúllan, ese pequeño cambio en como un objeto realiza una acción determinada es el polimorfismo y normalmente se logra sobrescribiendo o sobrecargando (overriding) los métodos de la clase padre.

// La clase define las propiedades y comportamiento del objeto

class Persona {

nombre = '';

correo = '';

// El constructor instancia la clase

// es lo que hace que el objeto que se está creando

// tenga una identidad y un estado

constructor(nombre, correo) {

// La palabra reservada `this` hace referencia al ámbito `scope` del objeto

// en este caso hace referencia a sus propiedades internas.

this.nombre = nombre;

this.correo = correo;

}

// Los métodos definen el comportamiento que tendrá el objeto

hablar(mensaje) {

console.log(`Persona ${this.nombre} dice: ${mensaje}`);

}

decirCorreo() {

this.hablar(`Mi correo es: ${this.correo}`);

}

}

// La clase `Instructor` hereda las propiedades y métodos de la clase `Persona`

// También se puede decir que `Instructor` es un tipo de `Persona`

class Instructor extends Persona {

// Se pueden crear nuevas propiedades para esta clase hija

materia = null;

// Así como crear nuevos métodos para esta clase hija

asignarMateria(materia) {

this.materia = materia;

}

explicar(tema) {

// Validamos si tiene materia asignada

if (!this.materia) {

console.log(`${this.nombre} no tiene materia asignada.`);

} else {

console.log(`Hoy, en clase de ${this.materia}, veremos el tema: ${tema}`);

}

}

// O se pueden sobrescribir métodos

hablar(mensaje) {

console.log(`${this.nombre} dice: ${mensaje}`); }

}

// Creamos o instanciamos un nuevo objeto de tipo `Persona`

// Al instanciarlo se le asigna una identidad.

const fulano = new Persona('Fulano', 'fulanito@de.tal');

// Se pueden realizar acciones por medio de su interfaz (abstracción)

// Imprime `Mi correo es: fulanito@de.tal`

fulano.decirCorreo();

// Imprime `Persona Fulano dice: Hola, qué tal!`

fulano.hablar('Hola, que tal!');

// Así mismo se puede crear un nuevo objeto de tipo `Instructor`

// El cual recibe una nueva identidad.

const instructorZutano = new Instructor('Instructor Zutano', 'instructor.zutano@de.tal');

// Al ser el objeto de tipo `Instructor` se tiene acceso a los nuevos métodos definidos en su clase

// Ya que aún no asignamos una materia

// imprime: `Instructor Zutano no tiene materia asignada`

instructorZutano.explicar('Programación orientada a objetos');

// Los métodos de las clases son los únicos que modifican las propiedades internas

// Por lo que las propiedades están encapsuladas

instructorZutano.asignarMateria('Desarrollo de Software');

// Ya que el Instructor tiene materia asignada

// imprime: `Hoy, en la clase de Desarrollo de Software, veremos el tema Programación orientada a objetos` 22

instructorZutano.explicar('Programación orientada a objetos');

// Se puede acceder a los métodos de la clase padre `Persona`

// Imprime `Mi correo es: instructor.zutano@de.tal`

instructorZutano.decirCorreo();

// O se pueden usar los métodos sobrescritos de la clase hija (polimorfismo)

// Imprime: `Zutano Instructor dice: Hola desde Instructor`

instructorZutano.hablar('Hola desde Instructor');

**¿Qué tal te pareció el contenido visto? Te dejo por aquí una infografía por si deseas consultarla y descargarla, es sobre Programación Orientada a Objetos, [presiona aquí.](https://github.com/U-Camp/BOOT-M1-SEM7/blob/main/infografias/M1_S7_Infografia%2002.pdf)**

**¡Muy bien! Hasta aquí llega el contenido de esta semana, por favor, repasa todo lo visto hasta el momento, mientras más practiques mucho mejor.**

**Nos vemos la próxima semana. 😃**