ES6 y POO (Parte I)

Introducción a la Programación Orientada a Objetos

Competencias

- Comprender el paradigma de la Programación Orientada a Objetos para implementar los objetos, sus métodos y propiedades.
- Utilizar el operador “this” para poder acceder a las variables del entorno de un objeto.

Introducción

La Programación Orientada a Objetos (POO) es una forma distinta de pensar las soluciones que busca facilitar el reuso de código y reducir la complejidad, o sea la cantidad de código que interactúa con otro código distinto. Aprender a trabajar con objetos y crear los nuestros, nos da acceso a un mundo completo de oportunidades.

Algunas de sus principales ventajas son:

- Proporciona una estructura clara para los programas.
- Ayuda a mantener el código bajo el concepto DRY "Don't repeat yourself - No se repita", y hace que el código sea más fácil de mantener, modificar y depurar.
- Hace posible crear aplicaciones reutilizables con menos código y menor tiempo de desarrollo.

Programación Orientada a Objetos

En la Programación Orientada a Objetos (POO) se busca describir algo del mundo real de manera abstracta (es decir, en el código). Todo es un objeto, y cualquier acción que necesitemos realizar en los datos (lógica, modificaciones, entre otros) se escriben como métodos de un objeto. En otras palabras, la Programación Orientada a Objetos puede mejorar la capacidad del desarrollador para crear prototipos de software rápidamente, ampliar la funcionalidad existente, refactorizar el código y mantenerlo a medida que se desarrolla.

Para comprender mejor en qué consiste este paradigma y cómo se relaciona con JavaScript, a lo largo de la lectura veremos algunas terminologías básicas, como lo son: atributos, métodos, abstracción, encapsulamiento, entre otros.

Objetos y propiedades

Un objeto es una representación de algo en el mundo real. Puede ser un auto, una casa, una bicicleta, un conjunto de números, puede ser una página web o incluso un carro de compras. Un objeto puede estar compuesto de más objetos. Así mismo, los objetos también tienen propiedades o datos, también conocidos como atributos, como por ejemplo, color, tamaño, nombre, edad, valores o cualquier otra cosa que imagines. Mientras que las funcionalidades, también conocidas como métodos tienden a modificar y/o utilizar estos estados o atributos, ya que son una capacidad del objeto en sí, como caminar. Por ejemplo cuando un auto se mueve, éste cambia su posición.

Para ilustrar un uso básico de objetos, el siguiente ejercicio solicita crear un objeto denominado “carro”, el cual debe tener las propiedades de marca, modelo y fecha con los valores “Toyota, Corolla, 2020” respectivamente. Por lo tanto, lo primero que debemos hacer en la consola del navegador, es crear el objeto mediante la instrucción `new Object()`, con el nombre de carro. Luego asignar las propiedades de marca, modelo y fecha con sus respectivos valores.

```
var carro = new Object();
carro.marca = 'Toyota';
carro.modelo = 'Corolla';
carro.fecha = 2020;
```

La instrucción *new* es la que nos permite crear un nuevo objeto, a esto le llamaremos instanciar (simplemente es crear un nuevo objeto partiendo de otro). Luego sobre este objeto que acabamos de instanciar definiremos diversas propiedades, por ejemplo, el modelo y el año.

Podemos observar el objeto creado utilizando `console.log(carro)` y observando los valores dentro del inspector de elementos.

```
Object { marca: "Toyota", modelo: "Corolla", fecha: 2020 }
```

Ejercicio guiado: Creando objetos

Crear un rectángulo con las propiedades de largo y ancho, dando la libertad para escoger los valores a asignar en cada propiedad.

- ***Paso 1:*** En la consola del navegador debemos primeramente crear el nuevo objeto denominado rectángulo:

```
var rectángulo = new Object();
```

- ***Paso 2:*** En la misma consola del navegador web, vamos a crear las propiedades de ancho y de largo para el objeto "rectangulo":

```
rectangulo.ancho = 10;
rectangulo.largo = 5;
```

- ***Paso 3:*** Finalmente, podemos observar el valor utilizando `console.log(rectangulo);` arrojando como resultado sobre la misma consola del navegador web lo siguiente:

```
Object { ancho: 10, largo: 5 }
```

Así como existe la metodología para crear objetos, vista anteriormente, también existe otra forma de crearlos, la cual se logra implementando la siguiente sintaxis:

```
cuadrado = {ancho: 10, largo: 5}
```

Utilizando la instrucción "console.log" podemos observar que obtenemos el mismo resultado. Por consiguiente, esta notación recibe el nombre de notación literal.

Por otra parte, también podemos mostrar o modificar el valor de estas propiedades de la siguiente forma:

```
console.log(cuadrado.ancho);
cuadrado.ancho = 7;
cuadrado['ancho'] = 8;
```

Finalmente, se puede implementar cualquiera de las dos notaciones independiente de la sintaxis que utilicemos para crear el objeto. Pero en la práctica, para la Programación Orientada a Objetos, es común instanciar objetos mediante el operador new, por lo tanto, es el más utilizado. Ahora bien, para saber cómo funciona el operador new en JavaScript, puedes revisar el **Material Apoyo Lectura - ¿Cómo funciona new?**, nombre con el cual lo encontrarás en "Material Complementario". Así mismo, puedes revisar el **Material Apoyo Lectura - Copia profunda de objetos**, ubicado "Material Complementario", para conocer cómo puedes realizar una copia de un objeto sin perder información del objeto creado.

Ejercicio propuesto

1. Traspasa el siguiente objeto a su notación literal y utiliza console.log para verificar el resultado.

```
var notebook = new Object();
myCar.make = 'Dell';
myCar.processor = 'I7';
```

2. Utilizando la sintaxis objeto.propiedad, modifica el procesador del notebook por "I9".

3. Utilizando la sintaxis objeto['propiedad'], modifica la marca del computador por "HP".

4. Utilizando la sintaxis new Object, crea el objeto estudiante con las propiedades nombre y edad, siendo los valores "Juan" y "35" respectivamente para cada propiedad.

Ejercicio guiado: Creando objetos con métodos

Los métodos sirven para que los objetos realicen acciones. Veamos un ejercicio sencillo de cómo crear un método y utilizarlo. En este caso, se debe crear un objeto denominado “vaca” y luego crear un método “sonido” que permita mostrar por la consola del navegador web el sonido que emite la vaca. Para esto, sigamos los siguientes pasos:

- ***Paso 1:*** En la consola de tu navegador web, crea el objeto con el nombre de “vaca” como se muestra a continuación:

```
var vaca = new Object();
```

- ***Paso 2:*** Agregar el método al objeto vaca creado anteriormente, el método debe llevar el nombre del sonido y deberá mostrar por la consola del navegador web el sonido que hace una vaca “moo”.

```
vaca.sonido = function () {
      console.log("moo");
}
```

- ***Paso 3:*** Utilizar el método realizando el llamado al objeto, indicando el método que deseamos implementar, en este caso, “sonido”. Recuerda que esto se puede lograr mediante la convención del punto, es decir, agregar el objeto y seguidamente el punto con el nombre del método que deseamos invocar. Finalmente, después de ejecutar el método requerido, podremos observar el texto "moo" en la consola del navegador web.

```
vaca.sonido();
```

Ahora bien, también es posible agregar un método a un objeto utilizando la notación literal.

```
perro = {
   hablar: function () {
      console.log('guau');
   }
};
perro.hablar();
```

Como se puede observar en el ejemplo anterior, mediante la notación literal se puede agregar cualquier método a un objeto, y posteriormente utilizarlo realizando el llamado convencional mediante el punto, para indicar el método al cual queremos acceder.

Ejercicio propuesto (5)

Traspasa el siguiente objeto a su notación literal.

```
var gato = new Object();

gato.sonido = function (){
  console.log("miau");
};
gato.sonido();
```

Ejercicio propuesto (6)

Utilizando la sintaxis new Object crea el objeto con el nombre de estudiante, agregando el método hablar que muestre en pantalla "Estoy aprendiendo objetos con JS".

Ejercicio guiado: Métodos que utilizan propiedades (operador this)

Es muy frecuente cuando trabajamos con objetos, que un método deba modificar o utilizar uno de los estados o atributos de un objeto, para lograr esto tendremos que utilizar la instrucción *this*. Para ilustrar esto, en el siguiente ejercicio se solicita crear un objeto con el nombre de "persona" en la consola del navegador web, luego, agregar la propiedad o atributo "nombre" con el valor de "Camila".

- ***Paso 1:*** Crear la variable con el nombre de "persona" y luego se debe asignar un nuevo objeto mediante la instrucción "`new Object()`", el cual permitirá inicializar un nuevo objeto en la variable indicada, como se muestra a continuación:

```
var persona = new Object();
persona.nombre = "Camila";
```

- ***Paso 2:*** Agregar un método al objeto creado anteriormente, en este caso se debe crear el método “saludar”, y agrega una función que permita mostrar un saludo por la consola del navegador web, indicando el valor asignado a la propiedad “nombre”, aquí es donde se debe implementar el operador this:

```
persona.saludar = function(){
  console.log("Hola soy " + this.nombre);
};
```

- ***Paso 3:*** Realizar el llamado al método “saludar” y observa el resultado en la consola del navegador web:

```
persona.saludar();
```

El resultado en la consola del navegador web será:

```
Hola soy Camila
```

Todo bien hasta el momento, pero qué sucedería si en el código anterior no utilizamos la palabra reservada *this*, y ejecutamos el código nuevamente en la consola del navegador web ¿Qué pasaría?

```
Uncaught ReferenceError: nombre is not defined
```

¿Qué fue lo que sucedió? Si observamos el error que obtuvimos veremos que nos muestra el mensaje "nombre is not defined", esto se debe a que si no especificamos con **this** que nombre es una propiedad, dentro del objeto buscará una variable llamada nombre. Por consiguiente, para referirnos a una propiedad del objeto, tenemos que utilizar **this**. Para saber cómo funciona el operador **this** en JavaScript, puedes revisar el **Material Apoyo Lectura - ¿Por qué utilizar this?**, nombre con el cual lo encontrarás en "Material Complementario”.

Ejercicio propuesto (7)

Partiendo del código explicado anteriormente, crea una segunda propiedad llamada edad, asigna un valor a ésta y que al llamar al método saludar se muestre en la consola el siguiente mensaje: "Hola soy Camila y tengo 30 años".

Ejercicio propuesto (8)

El siguiente código posee un error, encuéntralo y realiza las modificaciones pertinentes para que funcione a la perfección.

```
var notebook = new Object();
notebook.marca = "Dell";
notebook.obtener_informacion = function(){
     console.log("Computador marca " + marca);
}
notebook.obtener_informacion();
```

Programación Orientada a Objetos en ES5

Competencias

- Crear objetos a partir de una función constructora para definir las propiedades de ese objeto.
- Agregar métodos a una función constructora mediante la propiedad “prototype” para ampliar la capacidad de un objeto.

Introducción

En ES5 igualmente se trabaja la Programación Orientada a Objetos (POO), pero mediante la estructura base en la cual se basa JavaScript, como es el caso de los Prototipos. Por ende, en este capítulo se trabajará con la POO mediante la creación de funciones constructoras, la asignación de valores y métodos basados en los prototipos, aprendiendo a implementar algunos de los conceptos fundamentales de la POO como atributos y métodos desde la sintaxis te ES5.

La función constructora

Existen mejores formas para crear objetos, una muy utilizada es la función constructora, la cual, es una función normal y corriente que se utiliza para definir una especie de plantilla para nuestros objetos personalizados, la ventaja de esta forma es que nos permite crear múltiples objetos del mismo tipo.

Ejercicio guiado: Creación de objetos a partir de una función constructora

Implementar una función constructora para crear un objeto denominado "Estudiante" donde, por convención, la primera letra del nombre del objeto debe ir en mayúscula.

Esta función recibe uno o varios parámetros, los cuales serán los nombres de las propiedades que se inicializan dentro de la función. Por ejemplo, para este ejercicio el atributo o propiedad debe ser el nombre del estudiante.

- ***Paso 1:*** Crear en la consola del navegador web una función constructora denominada "Estudiante", utiliza el parámetro "nombre" y asigna el parámetro a una propiedad que llevará el mismo indicativo, pero recuerda que se debe implementar la palabra reservada "this".

```
function Estudiante(nombre) { // Función constructora
  this.nombre = nombre;
};
```

- ***Paso 2:*** Instanciar el objeto con nuevos valores, en este caso con distintos nombres, para ello se debe implementar la instrucción "new" seguidamente del nombre de la función constructora, y como argumento el valor que deseamos que almacene el objeto. Quedando el código de la siguiente manera:

```
var c1 = new Estudiante('Javiera');
var c2 = new Estudiante('Francisco');
var c3 = new Estudiante('Diana');
```

- ***Paso 3:*** Mostrar en la consola del navegador web el valor del nombre de cada uno de los objetos creados:

```
console.log(c1);
console.log(c2);
console.log(c3);
```

Si observamos ahora en el inspector de elementos veremos lo siguiente:

```
Object { nombre: "Javiera" }
Object { nombre: "Francisco" }
Object { nombre: "Diana" }
```

Esta forma de crear objetos nos será útil cuando queremos construir múltiples objetos.

Ahora bien, para agregar un método a un objeto en específico ocuparemos la misma fórmula de antes, es decir, a la instancia creada se le asigna el nombre del método que deseamos crear separados por un punto, seguidamente se le agrega una función anónima con el código correspondiente que necesite el método en sí. Por ejemplo, si a la función constructora creada anteriormente y a la instancia denominada "c1", le creamos un método denominado "saludar", el cual, muestre por consola el siguiente mensaje: "Hola soy Javiera". Se debería entonces implementar el siguiente código:

```
c1.saludar = function () {
  console.log("Hola soy " + this.nombre);
};
```

Luego, podemos hacer el llamado al método saludar, agregando los dos paréntesis al final del nombre del método para que pueda ejecutar la función, como se muestra a continuación:

```
c1.saludar();
```

Al ejecutar todo el código anterior en nuestra consola del navegador web, el resultado será:

```
Hola soy Javiera
```

Ahora, implementemos el método saludar con la misma funcionalidad realizada anteriormente, esta vez para la instancia creada en la variable "c2". Por consiguiente, utilizando la variable "c2", luego con el punto más el nombre del método y la función que este almacenará, se muestra por consola el siguiente mensaje: "Hola soy" más el valor de la variable denominada "nombre". Quedando el código:

```
c2.saludar = function(){
  console.log("Hola soy " + this.nombre);
}
```

Posteriormente, realizamos el llamado a esa función dentro del objeto y automáticamente podremos observar el resultado en la consola del navegador:

```
c2.saludar();
// resultado
Hola soy Francisco
```

Un detalle importante que siempre es bueno recordar, por convención los nombres que utilizaremos para las funciones constructoras serán con la primera letra mayúscula, de esta forma las diferenciaremos de otros tipos de funciones.

Ejercicio propuesto (9)

Reescribe el siguiente código utilizando una función constructora.

```
var persona = new Object();
persona.nombre = "Camila";
persona.saludar = function(){
      console.log("Hola soy " + this.nombre);
}
persona.saludar();
```

Prototipos

Los prototipos o prototype nos permiten agregar una propiedad o diversas funcionalidades a múltiples objetos. Por consiguiente, los prototipos son un mecanismo mediante el cual los objetos en JavaScript heredan características entre sí, es decir, un objeto puede tener diversas características pertenecientes a otro objeto, en otras palabras, los objetos en JavaScript se construyen en base a prototipos. Por ende, cuando se pide a un objeto una propiedad que no tiene, automáticamente el objeto la busca en su prototipo.

Ahora bien, los métodos y propiedades para un objeto son definidos en la propiedad prototype, la cual reside en la función constructora del objeto, no en la instancia misma del objeto cuando se crea o inicializa, por ello, cada instancia tiene un prototipo, en este caso fue el objeto con el que se creó, es decir, hablamos de la función constructora utilizada.

Ejercicio guiado: Función constructora y prototipos

Crear una función constructora con el nombre de "Usuario", la cual recibe como parámetro un nombre y será parte del atributo o propiedad de la función. Para luego, mediante la propiedad prototype, agregar un nuevo método a la función constructora que permita indicar el nombre del usuario en la consola del navegador web. Sigamos los siguientes pasos:

- ***Paso 1:*** Crear la función constructora en la consola del navegador web con el nombre de "Usuario", pasando el parámetro denominado "nombre" e inicializando la propiedad con ese parámetro:

```
function Usuario(nombre) {
      this.nombre = nombre;
}
```

- ***Paso 2***: Agregar un nuevo método a la función constructora creada, para ello se debe invocar la función por su nombre, más la propiedad "prototype" y el nombre del método que deseamos agregar, recuerda que todo esto se separa por un punto. Finalmente, se asigna mediante el operador de asignación la función con las instrucciones del caso, en este preciso momento solo será la instrucción de console.log para mostrar un mensaje de saludo con el nombre del usuario. Quedando el código de la siguiente forma:

```
Usuario.prototype.saludar = function(){
  console.log("Hola soy el usuario " + this.nombre);
}
```

- ***Paso 3:*** Instanciar la cantidad de objetos que deseamos, en este caso vamos a instanciar solo dos objetos, uno para el nombre "Juan" y otro para el nombre "Jocelyn".

```
var usuario1 = new Usuario('Juan');
var usuario2 = new Usuario('Jocelyn');
```

- ***Paso 4:*** Una vez instanciado los objetos, procedemos a invocar el método creado con el nombre de saludar. Por lo que debemos posteriormente ejecutar el código en la consola del navegador web para obtener el resultado.

```
usuario1.saludar();
usuario2.saludar();
// resultado
Hola soy el usuario Juan
Hola soy el usuario Jocelyn
```

También podemos hacer lo mismo con las propiedades o atributos, es decir, crear y asignar atributos globales para todos los objetos directamente a la función constructora, mediante el uso de la propiedad "prototype". Por ejemplo, asignemos una nueva propiedad o atributo a la función constructora denominada "Usuario" creada anteriormente, con el nombre de "edad" e inicializada con el valor numérico "30", quedando el código así:

```
Usuario.prototype.edad = 30;
```

Al igual que si crearamos la propiedad directamente sobre la función constructora, cuando asignamos un nuevo atributo con la propiedad "prototype", este atributo quedará disponible para todos los objetos instanciados en base a la función constructora. Para comprobar si esto es realmente cierto, invoquemos el atributo "edad" para cada una de las instancias creadas anteriormente, es decir, "usuario1" y "usuario2" y mostremos el resultado en la consola del navegador web ¿Cuál crees que será el valor mostrado?

```
console.log("La edad del usuario1 es " + usuario1.edad);
console.log("La edad del usuario2 es " + usuario2.edad);
```

Al ejecutar el código anterior en la consola del navegador web, el resultado encontrado será:

```
La edad del usuario1 es 30
La edad del usuario1 es 30
```

Como se puede apreciar en el resultado anterior, la edad es igual para ambos usuarios, ya que el atributo quedará compartido para todos los objetos instanciados sobre la función constructora.

Nota: Tenemos que tener mucho cuidado de modificar masivamente las propiedades de todos los objetos, porque es muy probable que terminamos modificando un objeto que no queremos modificar. Por ejemplo, si estuviéramos creando un juego de mesa y queremos darle puntos a un jugador utilizando prototype le daríamos puntajes a todos.

Ejercicio propuesto (10)

Reescribe el siguiente código utilizando prototipos para evitar definir múltiples veces el método agregarPuntos.

```
function Jugador(nombre){
  this.nombre = nombre;
  this.puntos = 0;
}

var jugador1 = new Jugador('Francisca');
jugador1.agregarPuntos = function(){
  this.puntos = this.puntos + 100;
}

var jugador2 = new Jugador('Camilo');
jugador2.agregarPuntos = function(){
  this.puntos = this.puntos + 100;
}

jugador1.agregarPuntos();
jugador2.agregarPuntos();
console.log(jugador1);
console.log(jugador2);
```

Trabajando con objetos

Competencias

- Comprender el principio de abstracción y encapsulamiento para la Programación Orientada a Objetos.
- Crear getters y setters para ampliar la funcionalidad de un objeto.

Introducción

Ahora que ya conocemos la sintaxis básica para crear objetos, ya podemos empezar a utilizarlos para satisfacer diversos tipos de requerimientos, en este capítulo abordaremos los conceptos básicos para poder lograrlo. Así mismo, implementaremos algunos conceptos básicos de Programación Orientada a Objetos como abstracción y encapsulamiento mediante el uso de objetos con ES5. Permitiendo crear, acceder y proteger objetos mediante métodos como la buena práctica lo indica.

El principio de abstracción

La Programación Orientada a Objetos tiene un fuerte foco en la reutilización. Una vez que los métodos de un objeto fueron programados no necesitamos conocer el detalle de estos para utilizarlos, al foco de pensar en el que, en vez de pensar en el cómo, se le conoce como principio de **abstracción**. Por ejemplo, cuando describes un objeto por primera vez, hablas en términos más abstractos, como el caso de un "Vehículo que se puede mover", aquí no dices cómo se moverá ese vehículo, es decir, no explicas si se moverá usando neumáticos, si volará o si navegará, sólo indicas que se mueve, nada más. A esto llamamos Abstracción, ya que estamos hablando de una de las cosas más esenciales de ese objeto, y es que está en movimiento, en vez de centrarse en detalles sobre cómo se mueve.

Sin duda la abstracción es un tema muy importante para la Programación Orientada a Objetos, de ahí que te invitamos a continuar con esta lectura en el **Material Apoyo Lectura - Principio de abstracción**, nombre con el cual lo encontrarás en "Material Complementario".

Ejercicio guiado: Construyendo un objeto a partir de un requerimiento

Vamos a partir de la premisa que estamos creando un programa de arquitectura que permite crear cuadrados para dibujar las bases de una casa, por ende, necesitaríamos al menos saber el valor de uno de los lados del cuadrado, también vamos a suponer que este programa que estamos construyendo debería entregar la información del área y del perímetro del cuadrado que dibujemos. Lo cual, sería muy útil si queremos saber cuánto espacio ocuparía nuestra casa y cuanto material necesitamos para construirla. Por ende, aplicando el principio de abstracción simplemente necesitamos entonces calcular un área y un perímetro de la figura geométrica.

Ahora bien, partiendo de los requerimientos anteriores, vamos a diseñar un programa que permita conocer el área y el perímetro de los cuadrados basados en el valor del lado, pero el programa debe ser creado en base a objetos, por lo tanto, sabemos que necesitamos construir un objeto “Cuadrado” que tenga un atributo llamado “lado” y a su vez métodos que permitan calcular el área y el perímetro de la figura geométrica, a los cuales llamaremos “calcularArea” y “calcularPerimetro” respectivamente**.** El área de un cuadrado se logra al multiplicar el lado por el mismo, mientras que el perímetro se consigue multiplicando el lado por cuatro (4). Sigamos los siguientes pasos:

- ***Paso 1:*** En la consola del navegador web crea una función constructora con el nombre de “Cuadrado”, la cual, recibe como parámetro “lado”:

```
function Cuadrado(lado) {
  this.lado = lado;
}
```

- ***Paso 2:*** Crear el método "calcularArea" mediante la propiedad "prototype", este método recibirá una función, la cual retornará un valor, para ello se implementa la palabra reservada "return", por consiguiente, el valor a retornar debe ser el cálculo del área, por lo que se debe multiplicar el lado por el mismo para obtener el área según la fórmula matemática para el cálculo del área de la figura geométrica.

```
Cuadrado.prototype.calcularArea = function() {
  return this.lado * this.lado;
}
```

- ***Paso 3:*** Crear el método "calcularPerimetro" que nos permita calcular el perímetro de la figura geométrica, implementado la propiedad "prototype", y se debe retornar el valor del perímetro, el cual se consigue al multiplicar el lado por 4.

```
Cuadrado.prototype.calcularPerimetro = function() {
  return this.lado * 4;
}
```

- ***Paso 4:*** Creada la función constructora y los métodos, ahora se debe proceder a instanciar el objeto, en este caso lo instanciamos sobre una variable con el nombre de "c1", y pasaremos un valor numérico, por ejemplo el número 2. Finalmente mostraremos por la consola del navegador el llamado a los métodos creados por individualmente:

```
var c1 = new Cuadrado(2);

console.log(c1.calcularArea());
console.log(c1.calcularPerimetro());
```

- ***Paso 5:*** El resultado en la consola del navegador deberá ser un área = 4 y un perímetro = 8.

```
4
8
```

Ejercicio propuesto (11)

Modifica el siguiente código utilizando prototipos para evitar tener que definir múltiples veces la misma función.

```
function Rectangulo(ancho, alto) { // Función constructora
  this.ancho = ancho; // Asignamos ancho inicial
  this.alto = alto; // Asignamos alto inicial
}

r1 = new Rectangulo(3, 5); // Creamos una instancia
r1.calcularArea = function() { // Agregamos el método
  return this.ancho * this.alto;
}

r2 = new Rectangulo(4, 8); // Creamos una instancia
r2.calcularArea = function() {
  return this.ancho * this.alto;
}

console.log(r1.calcularArea() + r2.calcularArea()); // La suma del área
de los rectángulos es:
```

Ejercicio propuesto (12)

Partiendo del ejercicio anterior, agrega un nuevo método denominado “calcularPerimetro” utilizando “prototype”, luego muestra en pantalla la suma de ambos perímetros (rectángulo r1 y r2) como un solo resultado. Tips: Perímetro de un rectángulo: 2*(ancho+alto).

El principio de Encapsulación

El principio de encapsulación nos enseña que debemos proteger los estados internos, es decir, las propiedades de los objetos. En donde su único propósito es ocultar el trabajo interno de los objetos del mundo exterior. En términos prácticos, este principio nos dice que debemos tener métodos para obtener y modificar cada estado, y que no deberíamos modificar los estados directamente. Este aislamiento hace que los datos (propiedades) del objeto sólo pueden gestionarse con las operaciones (métodos) definidos en ese objeto.

Por consiguiente, para lograr la protección de las propiedades de un objeto, existen diversas formas de aplicar el Encapsulamiento, como lo son: **Object.defineProperty(), Object.freeze(), WeakMap, Closures y/o IIFE**. Los dos últimos los trataremos con mucho detalle en lecturas posteriores para implementar Encapsulamiento.

Ejercicio guiado: Encapsulamiento en una función constructora

Crear una función constructora denominada "Estudiante", la cual recibe el nombre del estudiante como parámetro y luego modifica directamente el atributo creado después de instanciar el objeto.

- ***Paso 1:*** Crear la función constructora en la consola del navegador web con el nombre de "Estudiante" y agregar el parámetro que será inicializado como un atributo para el objeto.

```
function Estudiante(nombre){
  this.nombre = nombre;
}
```

- ***Paso 2:*** Sobre una variable denominada "estudiante1", se crea la instancia pasando el nombre de "Juan" y seguidamente mostramos por la consola del navegador la propiedad del nombre, obteniendo como resultado el valor pasado en la instancia.

```
var estudiante1 = new Estudiante('Juan');
console.log(estudiante1.nombre); // Juan
```

- ***Paso 3:*** Si deseamos modificar el atributo o propiedad del objeto dentro de la función constructora, se debe utilizar la instancia creada y hacer referencia a la propiedad "nombre", pasando el nuevo valor que se desee asignar, posteriormente mostramos por consola nuevamente el valor de la propiedad, como se muestra a continuación:

```
estudiante1.nombre = "Javier";
console.log(estudiante1.nombre); // Javier
```

Por consiguiente, el resultado que aparecerá en la consola del navegador web será el nuevo nombre "Javier". Todo funciona bien hasta el momento, pero, el código realizado anteriormente está sobreescribiendo la propiedad sin estar encapsulada desde fuera del objeto sin utilizar métodos, lo cual, **es una mala práctica y no se debe hacer, para ello se deben implementar getters y setters, tema que trataremos después de este inciso**.

Por el contrario, **la buena práctica** indica que debemos crear maneras de resguardar nuestras propiedades dentro de las funciones constructoras, quedando así las propiedades protegidas de cualquier cambio directo. Para esto, trabajamos con el ejercicio creado anteriormente, pero ahora si aplicamos las buenas prácticas. El objetivo del ejercicio sigue siendo el mismo, evitar la modificación de la propiedad "nombre" creada en la función constructora, pero esta vez mediante el método estático Object.defineProperty(). Para esto, sigamos los siguientes pasos:

- ***Paso 1:*** Crear la función constructora en la consola del navegador web, primeramente pasamos el parámetro e inicializamos la propiedad desde una variable denominada "getNombre" directamente en el constructor sin utilizar la propiedad this, evitando así el acceso fuera de la función constructora a la propiedad.

```
function Estudiante(nombre){
  var getNombre = nombre;
}
```

- ***Paso 2:*** Ahora se debe implementar uno de los métodos propios de **Object**, como el `Object.defineProperty,` el cual nos permite definir el objeto que queremos "proteger", asignando "this" como el objeto sobre el cual se define la propiedad, "getNombre", como el nombre de la propiedad a ser definida y finalmente el valor asociado a la propiedad. Esto nos permitirá obtener el valor de la propiedad pero no modificarla. Como se muestra a continuación:

```
function Estudiante(nombre){
  var getNombre = nombre;
  Object.defineProperty(this, "getNombre", {value: getNombre});
}
```

- ***Paso 3:*** Crear una nueva instancia con el nombre de "Estudiante", pasando un argumento, por ejemplo "Juan".

```
var estudiante1 = new Estudiante('Juan');
```

- ***Paso 4:*** Seguidamente, intentemos modificar la propiedad creada con el nombre de "getNombre", y luego obtengamos el valor de esta misma propiedad sobre la instancia creada:

```
estudiante1.getNombre = "Jocelyn";
console.log(estudiante1.getNombre);
```

- ***Paso 5:*** Al ejecutar el código anterior en al consola del navegador web, el resultado será el mismo valor que se asignó en la instancia, es decir:

```
Juan
```

Finalizando el ejercicio anterior, se puede apreciar que el principio de Encapsulamiento en la Programación Orientada a Objetos, nos indica que debemos "proteger" los atributos de un objeto. En consecuencia, puedes seguir aprendiendo y ampliando tu conocimiento sobre Encapsulamiento en el **Material Apoyo Lectura - Encapsulamiento**, nombre con el cual lo encontrarás en "Material Complementario". Así mismo, en lecturas posteriores trabajaremos con Closures e IIFE para aplicar de una mejor manera el encapsulamiento de propiedades.

Ejercicio propuesto (13)

Crear una función constructora denominada "Usuario", la cual reciba como parámetro el tipo de usuario que ingresa a un sistema, es decir, si es Administrador, Invitado o Cliente; esta propiedad debe llevar el nombre de "tipo". Luego protege la propiedad mediante la creación de una nueva variable local e implementando el método `Object.defineProperty` para evitar cambios externos en las propiedades existentes en la función constructora.

Getters y Setters

Hemos revisado hasta ahora la importancia de encapsular nuestros datos. En el paradigma orientado a objetos, los getters y setters nos permiten tener accesos controlados a la información que se encuentra en los objetos. Los métodos que permiten obtener el valor de un atributo se denominan getters (del inglés "get", obtener), por lo tanto, los getters son un método que devuelve el valor de una propiedad, dependiendo de lo que necesitemos podemos devolver el mismo valor, o uno modificado. Mientras que los métodos que modifican el valor de un atributo se denominan setter (del inglés "set", fijar) y dependiendo de lo que necesitemos puede modificarla directamente o través de una fórmula.

Ahora bien, como aún estamos trabajando con ES5, seguiremos implementando los prototipos y las funciones constructoras, así como el método `Object.defineProperty`, creando métodos que permitan obtener y modificar las propiedades de un objeto. Agregando como inicio el nombre al método para obtener el valor de la propiedad, inicializando con el término "get", mientras que al método que permita modificar la propiedad, lo iniciaremos con el término "set", agregando a ambos seguidamente el nombre de la propiedad que deseamos trabajar. Por ejemplo "getNombre" o "setNombre".

Ejercicio guiado: Getters y Setters

Generar un programa que permita crear un objeto para un vehículo con la propiedad de marca, luego se debe mostrar la marca y, posteriormente, modificarla mediante el uso de los getters y setters.

- ***Paso 1:*** En la consola del navegador web crearemos una función constructora denominada "Vehículos". Esta función recibe el parámetro "marca", pero ahora dentro de la función constructora, vamos a crear una variable para asignar el valor del parámetro, permitiendo aislarlo del exterior, pero esta vez con la notación de guión bajo "_", quedando la propiedad: "_marca". Quedando el código:

```
function Vehiculos(marca) {
      var _marca = marca;
}
```

- ***Paso 2:*** Ahora se debe implementar uno de los métodos propios de **Object**, como el `Object.defineProperty`, como se realizó en el inciso de encapsulamiento, asignando "this" como el objeto sobre el cual se define la propiedad, seguidamente del nombre de la propiedad a ser definida, pero esta vez utilizaremos los descriptores de acceso, con las claves opcionales de get (una función cuyo valor retornado será el que se use como valor de la propiedad) y el set (una función que recibe como único argumento el nuevo valor que se desea asignar a la propiedad y que devuelve el valor que se almacenará finalmente en el objeto). Los cuales, permitirían mostrar el valor de la propiedad interna de la función constructora o modificarla desde el exterior pero no directamente sobre ella, aplicando encapsulamiento, getters y setters en ES5. Como se muestra a continuación:

```
function Vehiculos(marca) {
      var _marca = marca;

      Object.defineProperty(this, "_getMarca", {
        get: function () {
            return _marca
        }
      });

      Object.defineProperty(this, "_setMarca", {
        set: function (marca) {
            _marca = marca
        }
      });
}
```

- ***Paso 3:*** Crear el método para el getter denominado “getMarca” que nos permita obtener el valor del atributo. Este método se crea con la propiedad “prototype” y se le asigna una función anónima que retorne el valor del atributo, siendo el código:

```
Vehiculos.prototype.getMarca = function(){
      return this._getMarca;
};
```

- ***Paso 4:*** Crear el método setter que permita modificar el valor del atributo desde el exterior, para ello dentro de la función asignaremos al acceso a la propiedad definida en el constructor mediante el nombre “_setMarca”, un nuevo valor de la marca del vehículo. Esto permite modificar la propiedad pero no directamente, ya que sin la definición del set dentro del `Object.defineProperty`, no se pudiera realizar ninguna modificación.

```
Vehiculos.prototype.setMarca = function(marca){
      this._setMarca = marca;
};
```

- ***Paso 5:*** Queda por ahora instanciar el nuevo objeto y realizar el llamado del método getter para ver el valor actual que pasaremos en la instancia del objeto. Por ejemplo, instancia el objeto sobre una variable con el nombre "v1" y pasa el valor como argumento "Ford". Luego hacemos el llamado al método getter para que nos muestre ese valor. Obteniendo como resultado la marca instanciada:

```
var v1 = new Vehiculos("Ford");
console.log(v1.getMarca()); // "Ford"
```

- ***Paso 6:*** Para realizar la modificación de ese atributo, implementamos el método setter denominado "setMarca", pasando como parámetro el nuevo valor que deseemos agregar al atributo, como por ejemplo "Kia". Luego se invoca nuevamente el método "getMarca" para ver si el cambio surtió efecto. Obteniendo como resultado la nueva marca agregada:

```
v1.setMarca("Kia");
console.log(v1.getMarca()); // "Kia"
```

Ejercicio propuesto (14)

A partir del siguiente código, crea un método getter y uno setter para las propiedades de número y pinta, que permitan visualizar y modificar cada uno de ellos.

```
function Carta(numero, pinta){
  this.numero = numero;
  this.pinta = pinta;
}

c1 = new Carta(2, 'Corazón');
c2 = new Carta(4, 'Espadas');
```

Trabajando con múltiples objetos

Competencias

- Identificar la cantidad de objetos asociados a otros para relacionarlos entre sí mediante el uso de la cardinalidad.
- Diferenciar los tipos primitivos de objetos para comprender su comportamiento con respecto a otros tipos de datos.

Introducción

Cuando se trabaja con objetos es muy frecuente trabajar con más de uno, por ejemplo, una persona puede tener una mascota, o un arreglo puede contener otros objetos. Los objetos pueden asociarse de distintas formas siendo una de las más frecuentes las relaciones de pertenencia, o sea un objeto o un conjunto de objetos le pertenece a otro. Asimismo, el comportamiento de estos objetos dependerá de su tipo, por lo que se puede identificar mediante las instrucciones adecuadas. En consecuencia, en este capítulo abordaremos este tipo de asociaciones y aprenderemos sobre la cardinalidad de estas, y como identificar a un objeto en JavaScript.

UML

UML (Unified Modeling Language - Lenguaje Unificado de Modelado), es un lenguaje común con el que se modelan varios procesos en el mundo de la informática, que permite que no sólo los programadores, sino también los usuarios de negocio puedan compartir información común. Asimismo, existen dos tipos de diagramas UML, de comportamiento y de estructura:

- **Diagramas de Comportamiento:** Muestra cómo se comporta el sistema de forma dinámica, es decir, los cambios que experimenta en su ejecución y sus interacciones con otros sistemas. Existen siete (7) subtipos, de acuerdo a lo siguiente: Diagrama de actividad, de casos de uso, de máquina de estados y de interacción, que a su vez se subdivide en diagrama de secuencia, global de interacción, de comunicación y de tiempos.

- **Diagramas de Estructura:** Muestran la estructura estática de un sistema y sus partes en diferentes niveles de abstracción. En esta clasificación existen seis (6) subtipos: Diagrama de clases, de componentes, de objetos, de estructura compuesta, de despliegue y de paquetes.

De estos últimos, el más comúnmente usado y de gran utilidad para el POO, es el diagrama de clases, que representa las clases dentro del sistema, atributos y operaciones, y la relación entre clases.

Cardinalidad

La cardinalidad nos especifica la cantidad de objetos que pueden estar asociados a otro objeto, por ejemplo, podemos decir que una persona puede tener solo un carnet de identidad o podríamos especificar que en un proyecto pueden trabajar hasta 50 personas. Este tipo de reglas a veces nos las entregan específicamente, a veces tenemos que entrevistar gente para conseguirlas o incluso en algunas ocasiones tendremos que proponerlas. Por ahora, supondremos que son entregadas.

Cardinalidad en el diagrama UML

En las asociaciones de un diagrama UML se debe indicar la cardinalidad. En el diagrama mostrado a continuación, se puede observar cómo una Persona puede tener cero (0) o una (1) mascota, la dirección de la flecha indica la navegación posible, o sea este diagrama indica que de una persona se puede llegar a una mascota pero de una mascota no se puede llegar a una persona.

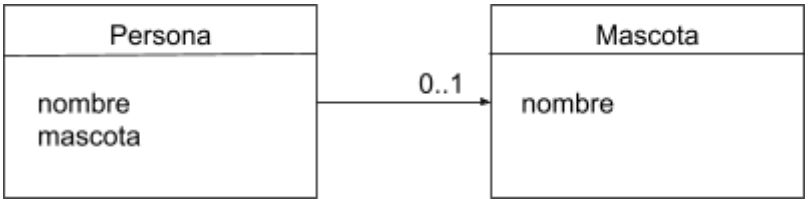

Imagen 1. Diagrama UML.
Fuente: Desafío Latam

Ejercicio guiado: Construir un código en JavaScript a partir de un diagrama UML

Crear dos objetos mediante funciones constructoras, uno para "Mascota" y otro para "Persona", luego se debe instanciar cada uno de ellos, pero en la instancia del objeto "Persona" se debe pasar como argumento el objeto instanciado en "Mascota", todo esto se debe realizar según las indicaciones mostradas en el diagrama UML en la imagen anterior. Por lo tanto, sigamos los siguientes pasos:

- **Paso 1:** Crear una función constructora que implemente dos propiedades o atributos (nombre y mascota) en la consola de nuestro navegador web, esta función constructora según el diagrama UML, pertenece al objeto Persona, la cual debe tener dos atributos (nombre y mascota):

```
function Persona(nombre, mascota){
  this.nombre = nombre;
  this.mascota = mascota;
}
```

- ***Paso 2:*** Repetir el paso anterior, pero esta vez para crear la función constructora para Mascota, la cual, sólo recibe como parámetro el valor para un solo atributo como se puede observar en el diagrama.

```
function Mascota(nombre){
  this.nombre = nombre;
}
```

- ***Paso 3:*** Disponibles las dos funciones constructoras, se debe proceder a instanciar los objetos en sus respectivas variables por separado. El primero que debemos instanciar en este caso será el de Mascota, el cual, debe enviar como argumento el nombre de la mascota. Luego, se debe instanciar el objeto Persona, pasando como argumento el nombre de la persona más el objeto instanciado en la variable "m1". Esto se debe a que en el diagrama UML nos indica según la cardinalidad que una persona puede llegar a una mascota, pero desde una mascota no se puede llegar a una persona.

```
var m1 = new Mascota('Snowball');
var p1 = new Persona('Julián', m1);
```

- ***Paso 4:*** Mostrar ambos objetos en la consola del navegador y analicemos el resultado obtenido:

```
console.log(p1);
console.log(m1);
```

El resultado obtenido en la consola del navegador web es:

```
Object { nombre: "Julián", mascota: {...} }
Object { nombre: "Snowball" }
```

Por lo tanto, en el código anterior vemos que desde la persona podemos obtener la mascota, pero desde las mascota no podemos obtener a la persona, eso no quiere decir que no podamos ocupar la variable que definimos, solo nos limita a no poder obtener la persona desde el objeto mascota.

Ejercicio propuesto (15)

Al ejercicio de Persona y Mascota creado anteriormente, agrega getters y setters para obtener el nombre de la mascota y el nombre de la persona.

Ejercicio propuesto (16)

A partir del siguiente diagrama UML, crea la función constructora para cada uno de los objetos, así mismo, crea una instancia de cada objeto.

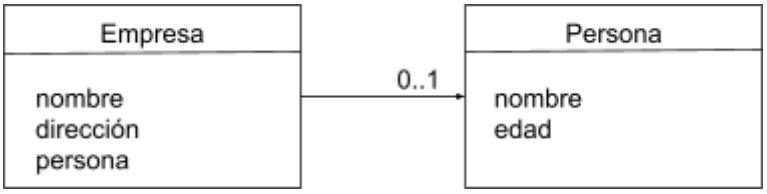

Imagen 2. Diagrama UML.
Fuente: Desafío Latam

Cardinalidad N

Existen diversas formas con las que podemos asociar múltiples objetos, dos formas sencillas de lograrlo son con arrays y hashes (objetos literales), por ahora lo realizaremos con arrays.

Ejercicio guiado: Cardinalidad N

Solicitan almacenar información de múltiples personas y cada una de estas personas puede tener múltiples redes sociales, además, cada persona tendrá un número de identificación "id" distinto. Por lo que se requiere almacenar esa información para cada persona y luego mostrarla. Para detallar mejor este ejercicio, partiremos del diagrama UML mostrado a continuación:

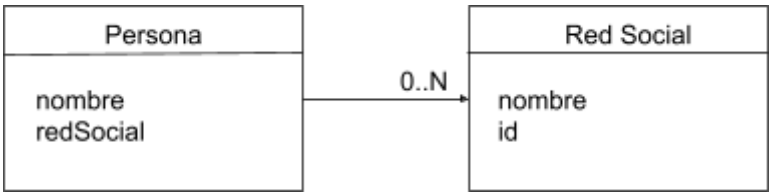

Imagen 3. Diagrama UML.
Fuente: Desafío Latam

Como se puede observar en la imagen anterior perteneciente al diagrama UML, cuando la cardinalidad es N, quiere decir que puede tener cero (0), uno (1), dos (2) o "n" cantidad de valores relacionados, en estos caso, se puede utilizar un arreglo, es decir, en lugar de simplemente asignar el atributo, se agrega al arreglo implementando el método "push" para ir agregando cada valor ingresado al arreglo sin borrar o dañar el anterior.

- ***Paso 1:*** Crear la función constructora en la consola del navegador web, la cual se denominará "persona" y tendrá dos parámetros "nombre y redes sociales". Aquí es importante destacar que el atributo para las redes sociales se debe iniciar con el valor ingresado por el parámetro o como un arreglo vacío, utilizando el operador lógico "o", por consiguiente, como vamos a guardar los datos en un arreglo, tenemos que asegurarnos que ese arreglo exista aunque esté vacío. De esta forma, si se crea una persona sin ninguna red social, posteriormente podremos añadirla sin problema alguno. Como se muestra en el siguiente código:

```
function Persona(nombre, redes_sociales){
  this.nombre = nombre;
  this.redes_sociales = redes_sociales || [];
}
```

- ***Paso 2:*** Crear el método que nos permita agregar "n" cantidad de redes sociales por usuario. En este método será donde agregaremos al atributo los valores ingresados para las redes sociales mediante un "push".

```
Persona.prototype.agregar_redsocial = function(red_social){
  this.redes_sociales.push(red_social);
}
```

- ***Paso 3:*** Crear la función constructora para las redes sociales, con el nombre de "RedSocial", recibiendo como parámetros dos valores, tal cual como se indica en el diagrama UML.

```
function RedSocial(nombre, id){
  this.nombre = nombre;
  this.id = id;
}
```

- ***Paso 4:*** Instanciar primeramente dos objetos para las redes sociales en dos variables separadas, en este caso "r1" y "r2", pasando como argumento los valores de: "'Twitter, @juanduran85" y "GitHub, JuanDuran85" respectivamente. Como se muestra a continuación:

```
r1 = new RedSocial('Twitter', '@juanduran85);
r2 = new RedSocial('GitHub', 'JuanDuran85');
```

- ***Paso 5:*** Instanciar ahora un objeto para "Persona", al cual llamaremos "p1". Este objeto y siguiendo las indicaciones del diagrama UML del ejercicio, debe recibir el nombre de la persona y el arreglo con los objetos de las redes sociales, quedando el código así:

```
p1 = new Persona('Juan', [r1,r2]);
```

- ***Paso 6:*** Mostrar el objeto creado para Persona por la consola del navegador web con la instrucción: "`console.log(p1);`" y veamos el resultado obtenido:

```
Object { nombre: "Juan", redes_sociales: (2) [...] }
```

En el resultado anterior, se puede observar como el objeto tiene el atributo nombre con `Juan` y el atributo `redes_sociales` con dos valores dentro del arreglo, si desglosamos este resultado para ver el arreglo, podríamos observar:

```
redes_sociales: Array [ {...}, {...} ]
0: Object { nombre: "Twitter", id: "@juanduran85" }
1: Object { nombre: "GitHub", id: "JuanDuran85" }
```

Como se puede apreciar en el resultado anterior, el atributo redes sociales tiene un arreglo con dos objetos dentro de él, esos objetos tienen las propiedades "nombre" y "id", cada uno con los valores pasados al momento de instanciar cada objeto. Agreguemos nuevos datos::

- ***Paso 7:*** Volver a instanciar un nuevo objeto para las redes sociales, en esta oportunidad denominado "r3", pasando los valores de: "Facebook, JuanCDuranR".

```
r3 = new RedSocial('Facebook', 'JuanCDuranR');
```

- ***Paso 8:*** Para pasar este nuevo objeto a la persona, utilizaremos el método creado denominado "`agregar_redsocial`", pasando como argumento el nuevo objeto creado para la red social "r3"

```
p1.agregar_redsocial(r3);
```

- ***Paso 9:*** Mostrar nuevamente el objeto "p1" en la consola del navegador web para apreciar si el nuevo valor creado en red social se cargó correctamente en el arreglo.

```
console.log(p1);
```

Al ejecutar el código anterior, obtendremos:

```
nombre: "Juan"
redes_sociales: (3) [...]
0: Object { nombre: "Twitter", id: "@juanduran85" }
1: Object { nombre: "GitHub", id: "JuanDuran85" }
2: Object { nombre: "Facebook", id: "JuanCDuranR" }
```

En el resultado anterior, se puede observar como los nuevos datos para la red social se agregaron sin problema, mediante el método "`agregar_redsocial`". Esto no es estrictamente obligatorio, pero debemos tener en mente el principio de encapsulación y respetar las buenas prácticas, por lo que se debe evitar manipular las propiedades directamente de un objeto.

Ejercicio propuesto (17)

Dado el siguiente diagrama UML, crea las funciones constructoras para cada una de las entidades.

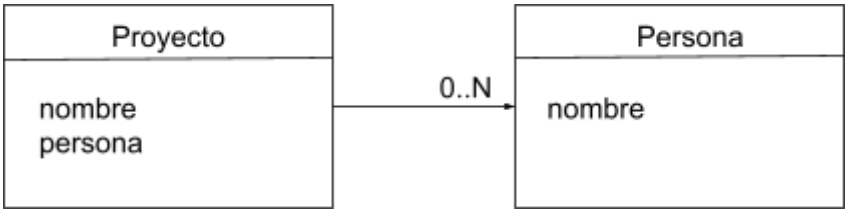

Imagen 4. Diagrama UML.
Fuente: Desafío Latam

Identificando Objetos En JavaScript

Partiendo del hecho que JavaScript es un lenguaje débilmente tipado y dinámico, lo que significa que no es necesario declarar el tipo de variable antes de usarla. Por ejemplo:

```
var una_variable = "Hola"; // una_variable es de tipo String
var una_variable = 22; // una_variable es ahora de tipo number
var una_variable = false; // una_variable es ahora de tipo boolean
```

Como podemos observar, el tipo de dato es asignado por su contexto. De este modo, en JavaScript podemos identificar dos grupos básicos de tipos de datos: primitivos y objetos. En la siguiente imagen, podemos revisar en detalle los distintos tipos de datos que existen en JavaScript:

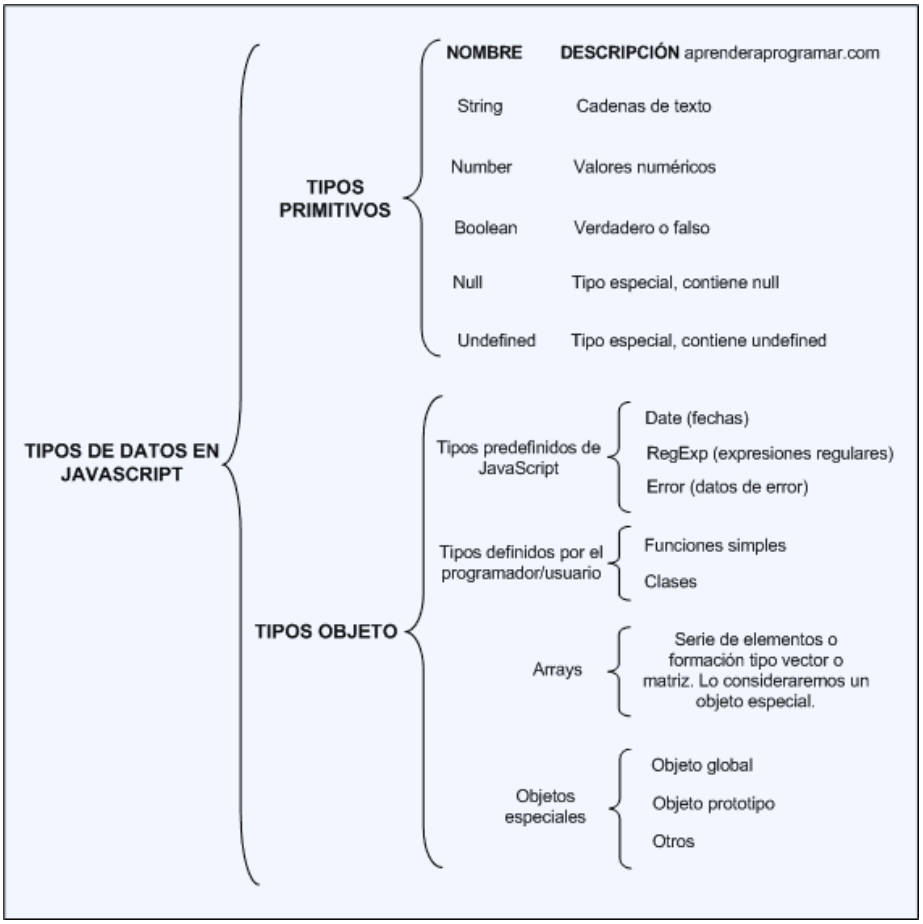

Imagen 5. Tipo de datos JavaScript
Fuente: aprenderaprogramar.com

En la imagen anterior se pueden observar siete (7) tipos de datos primitivos. Estos no poseen métodos ni propiedades a las que podamos acceder. Mientras que los objetos, por otra parte, son estructuras de datos complejas, que pueden ser referenciadas por medio de un identificador. Así mismo, para saber si una variable contiene un tipo primitivo o un objeto, existe un operador llamado `typeof,` que ya utilizamos en capítulos anteriores. Pero ahora, vamos a identificar los distintos tipos de datos, si son primitivos u objetos. Para ello:

Como primer paso, utilizando el comando "console.log" directamente en la consola del navegador web, se probarán distintos tipos de datos con el typeof, los datos a probar serán: `true, 2, '', function () {},Symbol(),undefined.`

El siguiente paso consistirá en escribir en la consola del navegador web mediante las instrucciones console.log y typeof los valores que deseamos probar, como se muestra a continuación:

```
console.log(typeof true);
console.log(typeof 2);
console.log(typeof '');
console.log(typeof function () {});
console.log(typeof {});
console.log(typeof Symbol());
console.log(typeof undefined);
```

Finalmente al ejecutar o presionar enter en la consola del navegador después de escribir todo el código anterior, el resultado sería:

```
boolean
number
string
function
object
symbol
undefined
```

Con este operador (typeof) puedes saber el tipo de una variable, cuando la variable contiene un tipo primitivo y/o cuando contiene un objeto. Pero typeof no es suficiente para determinar su tipo, porque siempre dice que la variable es de tipo object. En cambio, con instanceof validamos si el prototipo es instancia de otro objeto en particular. Este concepto lo abordaremos en profundidad más adelante.

Ejercicio propuesto (18)

Para los siguientes datos, identifica el tipo de dato utilizando typeof.

```
3.1416 , 'Maria' , NaN , "José" , null, false , [1,2,3,4,5,6,7,8]
```

Resumen

A lo largo de esta lectura cubrimos:

- Comprender el paradigma de la Programación Orientada a Objetos para implementar los objetos, sus métodos y propiedades.
- Utilizar el operador “this” para poder acceder a las variables del entorno de un objeto.
- Crear objetos a partir de una función constructora para definir las propiedades de ese objeto.
- Agregar métodos a una función constructora mediante la propiedad “prototype” para ampliar la capacidad de un objeto.
- Implementar el principio de abstracción y encapsulamiento para comprender el uso de la Programación Orientada a Objetos.
- Crear getters y setters para ampliar la funcionalidad de un objeto.
- Identificar la cantidad de objetos asociados a otros para relacionarlos entre sí mediante el uso de la cardinalidad.
- Diferenciar los tipos primitivos de objetos para comprender su comportamiento con respecto a otros tipos de datos.

Solución de los ejercicios propuestos

1. Traspasa el siguiente objeto a su notación literal y utiliza console.log para verificar el resultado.

```
var notebook = {make: 'Dell', processor: 'I7'};
console.log(notebook);
```

2. Utilizando la sintaxis objeto.propiedad, modifica el procesador del notebook por “I9”.

```
notebook.processor = 'I9';
```

3. Utilizando la sintaxis objeto['propiedad'], modifica la marca del computador por “HP”.

```
notebook['make'] = 'HP';
console.log(notebook);
```

4. Utilizando la sintaxis new Object, crea el objeto estudiante con las propiedades nombre y edad, siendo los valores “Juan” y “35” respectivamente para cada propiedad.

```
var estudiante = new Object();
estudiante.nombre = "Juan";
estudiante.edad = 35;
console.log(estudiante);
```

5. Traspasa el siguiente objeto a su notación literal.

```
var gato = {
      sonido: function (){
         console.log("miau");
      }
};
gato.sonido();
```

6. Utilizando la sintaxis new Object crea el objeto con el nombre de estudiante, agregando el método hablar que muestre en pantalla "Estoy aprendiendo objetos con JS".

```
var estudiante = new Object();

estudiante.hablar = function (){
  console.log("Estoy aprendiendo objetos con JS");
};

estudiante.hablar();
```

7. Partiendo del código explicado anteriormente, crea una segunda propiedad llamada edad, asigna un valor a ésta y que al llamar al método saludar se muestra en la consola el siguiente mensaje: "Hola soy Camila y tengo 30 años".

```
var persona = new Object();
persona.nombre = "Camila";
persona.edad = 30;
persona.saludar = function(){
      console.log("Hola soy " + this.nombre + " y tengo " + this.edad);
};
persona.saludar();
```

8. El siguiente código posee un error, encuéntralo y realiza las modificaciones pertinentes para que funcione a la perfección.

```
var notebook = new Object();
notebook.marca = "Dell";
notebook.obtener_informacion = function(){
      console.log("Computador marca " + this.marca);
}
notebook.obtener_informacion();
```

9. Reescribe el siguiente código utilizando una función constructora.

```
function Persona (nombre) {
      this.nombre = "Camila";
}

var persona1 = new Persona('Camila');

persona1.saludar = function () {
      console.log("Hola soy " + this.nombre);
}

persona1.saludar();
```

10. Reescribe el siguiente código utilizando prototipos para evitar definir múltiples veces el método agregarPuntos.

```
function Jugador(nombre){
      this.nombre = nombre;
      this.puntos = 0;
}

Jugador.prototype.agregarPuntos = function () {
      this.puntos = this.puntos + 100;
}

var jugador1 = new Jugador('Francisca');
var jugador2 = new Jugador('Camilo');
jugador1.agregarPuntos();
jugador2.agregarPuntos();
console.log(jugador1);
console.log(jugador2);
```

11. Modifica el siguiente código utilizando prototipos para evitar tener que definir múltiples veces la misma función.

```
function Rectangulo(ancho, alto) {
      this.ancho = ancho;
      this.alto = alto;
}

Rectangulo.prototype.calcularArea = function () {
      return this.ancho * this.alto;
}

r1 = new Rectangulo(3, 5);
r2 = new Rectangulo(4, 8);

console.log(r1.calcularArea() + r2.calcularArea());
```

12. Partiendo del ejercicio anterior, agrega un nuevo método denominado "calcularPerimetro" utilizando "prototype", luego muestra en pantalla la suma de ambos perímetros (rectángulo r1 y r2) como un solo resultado. Tips: Perímetro de un rectángulo: 2*(ancho+alto).

```
function Rectangulo(ancho, alto) {
      this.ancho = ancho;
      this.alto = alto;
}

Rectangulo.prototype.calcularArea = function () {
      return this.ancho * this.alto;
}

Rectangulo.prototype.calcularPerimetro = function () {
      return 2*(this.ancho + this.alto);
}

r1 = new Rectangulo(3, 5);
r2 = new Rectangulo(4, 8);

console.log(r1.calcularArea() + r2.calcularArea());
console.log(r1.calcularPerimetro() + r2.calcularPerimetro());
```

13. Crear una función constructora denominada “Usuario”, la cual reciba como parámetro el tipo de usuario que ingresa a un sistema, es decir, si es Administrador, Invitado o Cliente; esta propiedad debe llevar el nombre de “tipo”. Luego protege la propiedad mediante la creación de una nueva variable local e implementando el método `Object.defineProperty` para evitar cambios externos en las propiedades existentes en la función constructora.

```
function Usuario(tipo){
      let getTipo = tipo;
      Object.defineProperty(this, "getTipo", {value: getTipo});
}

var usuario1 = new Usuario('Administrador');
usuario1.getTipo = "Invitado";
console.log(usuario1.getTipo);
```

14. A partir del siguiente código, crea un método getter y uno setter para las propiedades de número y pinta, que permitan visualizar y modificar cada uno de ellos.

```
function Carta(numero, pinta){
      var _numero = numero;
      var _pinta = pinta;

      Object.defineProperty(this, "_getNumero", {
         get: function () {
            return _numero;
         }
      });

      Object.defineProperty(this, "_setNumero", {
         set: function (numero) {
            _numero = numero;
         }
      });

      Object.defineProperty(this, "_getPinta", {
         get: function () {
            return _pinta
         }
      });
```

```
      Object.defineProperty(this, "_setPinta", {
         set: function (pinta) {
            _pinta = pinta
         }
      });
};

Carta.prototype.getNumero = function(){
      return this._getNumero;
};

Carta.prototype.setNumero = function(numero){
      this._setNumero = numero;
};

Carta.prototype.getPinta = function(){
      return this._getPinta;
};

Carta.prototype.setPinta = function(pinta){
      this._setPinta = pinta;
};

c1 = new Carta(2, 'Corazón');
c2 = new Carta(4, 'Espadas');

console.log(c1.getNumero());
console.log(c2.getNumero());
console.log(c1.getPinta());
console.log(c2.getPinta());
c1.setNumero(5);
c2.setNumero(7);
c1.setPinta('Tréboles');
c2.setPinta('Rombos');
console.log(c1.getNumero());
console.log(c2.getNumero());
console.log(c1.getPinta());
console.log(c2.getPinta());
```

15. Al ejercicio de Persona y Mascota creado anteriormente, agrega getters y setters para obtener el nombre de la mascota y el nombre de la persona.

```
function Persona(nombre, mascota){
      var _nombre =  nombre;
      var _mascota = mascota;

      Object.defineProperty(this,"_nombre", {
      get: function(){
            return _nombre;
      },
      set: function(nuevoNombre){
            _nombre = nuevoNombre;
      }
      });
      Object.defineProperty(this,"_mascota", {
      get: function(){
            return _mascota;
      }
      });
};

function Mascota(nombre){
      var _nombre = nombre;

      Object.defineProperty(this,"_nombre", {
      get: function(){
            return _nombre;
      },
      set: function(nuevoNombre){
            _nombre = nuevoNombre;
      }
      });
};

Persona.prototype.getNombrePersona = function () {
      return this._nombre;
}
Mascota.prototype.getNombreMascota = function () {
      return this._nombre;
}

Persona.prototype.setNombrePersona = function (nuevoNombre) {
      this._nombre = nuevoNombre;
```

```
}
Mascota.prototype.setNombreMascota = function (nuevoNombre) {
      this._nombre = nuevoNombre;
}

var m1 = new Mascota('Snowball');
var p1 = new Persona('Julián', m1);
console.log(p1);
console.log(m1);
console.log(m1.getNombreMascota());
console.log(p1.getNombrePersona());
m1.setNombreMascota('Taty');
console.log(m1.getNombreMascota());
```

16. A partir del siguiente diagrama UML, crea la función constructora para cada uno de los objetos, así mismo, crea una instancia de cada objeto.

```
function Empresa(nombre, direccion, persona) {
      this.nombre = nombre;
      this.direccion = direccion;
      this.persona = persona;
}

function Persona(nombre, edad) {
      this.nombre = nombre;
      this.edad = edad;
}

var persona1 = new Persona('Jocelyn',30);
var empresa1 = new Empresa('Desafío Latam','Santiago',persona1);
console.log(persona1);
console.log(empresa1);
```

17. Dado el siguiente diagrama UML, crea las funciones constructoras para cada una de las entidades.

```
function Proyecto(nombre, persona){
      this.nombre = nombre;
      this.persona = persona || [];
}

function Persona(nombre){
      this.nombre = nombre;
}

Proyecto.prototype.setAgregarPersona = function(persona_nueva){
      this.persona.push(persona_nueva);
}

var persona1 = new Persona('Juan');
var persona2 = new Persona('Jocelyn');
var proyecto1 = new Proyecto('JS',[persona1,persona2]);
console.log(proyecto1);
var persona3 = new Persona('Yecenia');
proyecto1.setAgregarPersona(persona3);
console.log(proyecto1);
console.log(proyecto1.nombre);
```

18. Para los siguientes datos, identifica el tipo de dato utilizando typeof.

```
number
string
number
string
object
boolean
object
```