ES6+ y POO

Trabajando con objetos

{desafío} latam_

Utilizar los conceptos fundamentales de la programación orientada a objetos acorde al lenguaje Javascript para resolver un problema simple.

- Unidad 1:
  ES6+ y POO
- Unidad 2:
  Herencia
- Unidad 2:
  Callbacks y APIs

Te encuentras aquí

{desafío}
latam_

¿Qué aprenderás en esta sesión?

- *Reconoce los pilares de la programación orientada a objetos para el desarrollo de una pieza de software.*

¿Sabes cuál es el principio fundamental de la POO que se enfoca en la reutilización de métodos y funciones?

/* El principio de abstracción */

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.

Ejercicio guiado: Construyendo un objeto a partir de un requerimiento

{desafío} latam_

Ejercicio guiado

Construyendo un objeto a partir de un requerimiento

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;
}
```

Ejercicio guiado

Construyendo un objeto a partir de un requerimiento

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;
}
```

Ejercicio guiado

Construyendo un objeto a partir de un requerimiento

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;
}
```

Ejercicio guiado

Construyendo un objeto a partir de un requerimiento

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 individualmente:

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

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

Ejercicio guiado

Construyendo un objeto a partir de un requerimiento

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

```
4
8
```

/* El principio de Encapsulación */

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.**

Ejercicio guiado: Encapsulamiento en una función constructora

{desafío} latam_

Ejercicio guiado

Encapsulamiento en una función constructora

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;
}
```

Ejercicio guiado

Encapsulamiento en una función constructora

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
```

Ejercicio guiado

Encapsulamiento en una función constructora

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
```

Ejercicio guiado

Encapsulamiento en una función constructora

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.

/* Getters y Setters */

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.

Getters y Setters

Ahora bien, como aún estamos trabajando con ES6, 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

{desafío} latam_

Ejercicio guiado

Ejercicio guiado: 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 guion bajo “_”, quedando la propiedad: “_marca”. Quedando el código:

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

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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 ES6. Como se muestra a continuación:

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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

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

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

{desafío} latam_

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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;
};
```

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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;
};
```

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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"
```

Ejercicio guiado

Ejercicio guiado: Getters y Setters

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"
```

/* UML */

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.

UML

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 */

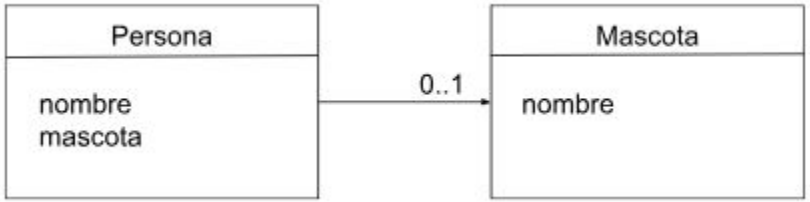
Cardinalidad

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 */

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.



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

{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:

Ejercicio guiado

Construir un código en JavaScript a partir de un diagrama UML

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;
}
```

Ejercicio guiado

Construir un código en JavaScript a partir de un diagrama UML

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;
}
```

Ejercicio guiado

Construir un código en JavaScript a partir de un diagrama UML

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);
```

Ejercicio guiado

Construir un código en JavaScript a partir de un diagrama UML

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

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

Ejercicio guiado

Construir un código en JavaScript a partir de un diagrama UML

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.

Resumiendo

El trabajo con múltiples objetos en programación orientada a objetos es fundamental para poder crear sistemas complejos que puedan manejar grandes cantidades de información. La cardinalidad es una herramienta muy útil para poder especificar la relación entre objetos y poder modelar de manera adecuada el sistema. Además, el uso de diagramas UML es una práctica común en la industria de la programación que permite una mejor comunicación y comprensión entre los desarrolladores.

La identificación de objetos en JavaScript es una tarea importante para poder trabajar con ellos de manera adecuada. La creación de funciones constructoras es una forma común de crear objetos en JavaScript y el uso de arreglos y hashes es muy útil para poder trabajar con múltiples objetos.

Comenta con tus palabras:

¿Qué aprendiste la clase de hoy?

Próxima sesión...

- *Desafío opcional - Creando y modificando objetos*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales