Programación con Java Script I

Programación orientada a objetos

Introducción

Recordando el concepto de objetos, según los define Kantor (2022), son la estructura de datos fundamental de JavaScript. Es importante retomar este concepto porque existe un estilo de programación denominado Programación Orientada a Objetos (POO), en donde se utilizan estos objetos para emular la vida real. Una de las partes más complejas del desarrollo con objetos es definir qué estructura de datos será la más adecuada para cada situación, pues a medida que se tiene la necesidad de programar cosas más complejas, estas se pueden tornar un poquito difíciles.

Para poder organizar cada uno de los elementos que se van agregando a nuestro código (variables, funciones, constantes, etc.), estos se agrupan en una estructura denominada Clase, que, dicho de otra manera, es una plantilla sobre la cual se crearán los objetos. Estas estructuras están compuestas por atributos y operaciones. Los atributos, propiedades o características son todas aquellas variables o constantes, mientras que las funciones son todos los métodos que se utilizan para realizar una actividad.

Explicación

Programación orientada a objetos

Con relación a la programación orientada a objetos, Haverbeke (2018) expone que la idea central es dividir a los programas en piezas más pequeñas y lograr que cada una de estas sea responsable de gestionar su propio estado. Así, se logra conocer el funcionamiento de una parte del programa de manera local, lo que permite a los desarrolladores ocuparse solamente de la pieza que les corresponde sin tener la necesidad de interesarse por otras partes del programa.

Lo anterior permite que el mantenimiento y crecimiento sea modular. La intercomunicación de estas pequeñas partes se logra a través de interfaces, las cuales son un conjunto limitado de funciones y vinculaciones que proporcionan funcionalidades en un nivel más abstracto.

Cuando hablamos de abstracción en programación, nos referimos a la extracción de las propiedades esenciales de un concepto u objeto que existe en la realidad y que deseamos representarlo en nuestras aplicaciones. La abstracción nos permite no enfocarnos en los detalles que no son útiles respecto a lo que estamos realizando.

Al igual que otros lenguajes orientados a objetos, JavaScript proporciona soporte para la implementación de herencia, es decir, la reutilización de código o datos mediante un mecanismo de delegación dinámico. A diferencia de muchos lenguajes convencionales, el mecanismo de herencia de JavaScript se basa en prototipos en lugar de clases. Para muchos programadores, JavaScript es el primer lenguaje orientado a objetos que encuentran sin clases.

En muchos lenguajes, cada objeto es una instancia de una clase asociada, o sea, un objeto es una copia de su clase o plantilla asociada que proporciona código compartido entre todas sus instancias. JavaScript, por el contrario, no tiene una noción incorporada de clases, sino que los objetos heredan de otros objetos. Cada objeto está asociado con otro objeto, conocido como su prototipo. Trabajar con prototipos puede ser diferente de las clases, aunque muchos conceptos de los lenguajes tradicionales orientados a objetos todavía se conservan.

Explicación

Estas piezas de las que estamos hablando se modelan utilizando objetos y estas interfaces consisten en un conjunto de métodos y propiedades. Las interfaces pueden ser de dos tipos: las públicas, que son parte de la interfaz, y las privadas, que son las que solo deben ser consideradas adentro del objeto. Muchos lenguajes de programación proveen una forma de distinguir entre propiedades públicas y privadas, pero JavaScript no lo hace, por lo menos hasta este momento. Como desarrollador de JavaScript, esto no te va a provocar mayor problema si utilizas el "estándar" que utilizan otros programadores, por ejemplo, colocar un guion bajo (_) al comienzo de las propiedades para indicar que son privadas, entre otros.

Figura 1. Métodos.

Profundizando un poco en los métodos, podemos decir que son el conjunto de operaciones que puede realizar un objeto de acuerdo con su estado actual. Cada método contiene funciones, que son las que especifican una acción y siempre van acompañadas de paréntesis (), de tal manera que definirán su comportamiento. Cuando son invocados, realizan la acción y opcionalmente regresan un valor. Por ejemplo:

```
let persona = {};
persona.hablar = function(linea) {
 console.log(`la persona dice '${linea}'`);
};

persona.hablar("Estoy vivo.");
// → La persona dice 'Estoy vivo.'
```

Figura 2. Declaración de métodos.

Cuando ejecutamos el ejemplo anterior, se está ejecutando el método diseñado hablar() del objeto persona. Es importante recordar que cuando escribes tus propias funciones, lo que en realidad se hace es añadir métodos al objeto window, que es el padre de todos los demás objetos de JavaScript.

Explicación

En los casos en que una función es invocada como método, del tipo objeto.metodo(), la vinculación llamada “this” (este) en su cuerpo apunta al objeto en la que fue llamada. Observa el siguiente ejemplo para mejor referencia:

```
function hablar(linea) {
  console.log(`La persona ${this.tipo} dice '${linea}'`);
 }
 let personaFeliz = {tipo: "feliz", hablar};
 let personaHambrienta = {tipo: "hambrienta", hablar};

 personaFeliz.hablar("La vida es bella, " +
          "Hoy es un buen día para empezar!");
 // → La persona feliz dice 'La vida es bella, Hoy es un buen
día para empezar'
 personaHambrienta.hablar("Podría comerme un león ahora
mismo.");
 // → La persona hambrienta dice 'Podría comerme un león
ahora mismo.'
```

Figura 3. Uso de this.

Posiblemente hayas notado algo un poco extraño en el método de este objeto y quizás te preguntes qué es this. Esta palabra se refiere a la función con la que se está trabajando y la clave en su uso es tener claro el lugar desde donde se invoca para saber qué valor va a tomar. Por tanto, regresando a nuestro caso, this es equivalente a la *persona*.

Entonces te preguntarás, ¿por qué no se escribió la palabra persona en lugar de this? La respuesta es porque this nos ayuda a asegurarnos de que se usen los valores correctos cuando cambie el contexto de un miembro. Por ejemplo, dos instancias de objetos personas *(personaFeliz es una instancia y personaHambrienta es otra instancia)* pueden tener diferentes nombres, pero querrás usar su atributo (hambrienta y feliz respectivamente) para identificar quién está hablando. Esto tomará relevancia y utilidad cuando se generan objetos dinámicamente, por ejemplo, usando **constructores.**

Ahora, profundizando un poco en las propiedades, una propiedad se puede definir como las diferentes variables y/o constantes que representan las características de un objeto, las cuales tienen un nombre y un valor. El nombre de una propiedad puede ser cualquiera (o incluso ser una cadena vacía), pero un objeto no puede tener dos propiedades con el mismo nombre. El valor puede ser cualquiera permitido por JavaScript o puede ser una función obtenedora (getter) o asignadora (setter).

Explicación

Es importante ser capaz de distinguir entre las propiedades definidas directamente en el objeto de aquellas que son heredadas del objeto prototipo.

Además del nombre y valor, cada propiedad tiene tres atributos:

- El atributo escritura especifica que se puede establecer el valor de la propiedad.
- El atributo enumerable especifica si el nombre de la propiedad es devuelto por el bucle *for/in*.
- El atributo configurable especifica si la propiedad puede ser borrada y si sus atributos pueden ser alterados.

Observa el siguiente código:

```
let objeto = {};
console.log(objeto.toString);
// → function toString(){…}
console.log(objeto.toString());
// → [object Object]
Figura 4. Objeto vacío.
```

Figura 4. Objeto vacío.

Como se puede apreciar, se está declarando un objeto vacío, sin embargo, ese objeto tiene propiedades que son adheridas al conjunto de propiedades propias del objeto a través de un **prototipo**, pero si este objeto está vacío, ¿quién es su prototipo? En estos casos, este tipo de objetos tienen como prototipo el gran prototipo ancestral, la entidad detrás de casi todos los objetos *Object.prototype*. Cuando deseas crear un objeto con un prototipo específico, deberás usar *Object.create:*

```
let personaPrototipo = {
  hablar(palabras) {
   console.log(`La persona ${this.tipo} dice '${palabras}'`);
  }
 };
 let personaSonriente = Object.create(personaPrototipo);
 personaSonriente.tipo = "sonriente";
 personaSonriente.hablar("Jajajajaja!");
 // → La persona sonriente dice 'Jajajajaja!'
```

Figura 5. Prototipos.

Explicación

Una propiedad como hablar(palabras) en un objeto es la forma corta para declarar un método.

La persona “prototipo” actúa como un contenedor para las propiedades que son compartidas por todas las personas. Un objeto persona individual, como la persona sonriente, contiene propiedades que aplican solo a sí mismo y deriva propiedades compartidas desde su prototipo.

Los prototipos nos sirven para definir métodos y propiedades en un objeto y estas serán heredadas por todas las instancias del objeto. No obstante, estas instancias no siempre deberán ser exactamente iguales, todas las instancias tienen las propiedades y métodos de su prototipo, pero podrían tener propiedades y métodos más que sus instancias “hermanas”. Todas aquellas propiedades que difieren de instancia en instancia, como lo es la propiedad tipo de nuestras personas, es necesario que se almacenen directamente en los objetos mismos.

Para instanciar una clase, debes crear un objeto que derive del prototipo adecuado, pero también es necesario que el objeto tenga las propiedades que las instancias de esta clase deberían tener, como se muestra en la figura 6.

```
function crearPersona(tipo) {
 let persona = Object.create(personaPrototipo);
 persona.tipo = tipo;
 return persona;
}
```

Figura 6. Función constructora.

JavaScript nos provee de una manera muy fácil de hacer la definición de una función constructora, solo debes colocar la palabra reservada new delante de una llamada de función y eso permitirá que la función sea tratada como un constructor. Esto se traduce en que un objeto con el prototipo adecuado es creado automáticamente y vinculado a this en la función.

El objeto prototipo utilizado al construir objetos se encuentra al tomar la propiedad prototype de la función constructora.

```
function Persona(tipo) {
 this.tipo = tipo;
}
Persona.prototype.hablar = function(linea) {
 console.log(`La persona ${this.tipo} dice '${linea}'`);
};

let personaRara = new Persona("rara");
```

Explicación

Por convención, los nombres de los constructores tienen la primera letra en mayúscula, esto para que se puedan distinguir de las funciones.

Entonces, resumiendo, las clases en JavaScript son funciones constructoras con una propiedad prototipo, por tanto, el código anterior se podría implementar de la siguiente manera:

```
class Persona {
 constructor(tipo) {
   this.tipo = tipo;
 }
 hablar(linea) {
   console.log(`La persona ${this.tipo} dice '${linea}'`);
 }
}

let personaRara = new Persona("rara");
```

Figura 8. Uso de constructores.

La palabra class nos permite definir un constructor y un conjunto de métodos. El método llamado constructor es tratado de manera especial, los otros métodos estarán empacados en el prototipo de ese constructor.

Este sistema de clases y prototipos de JavaScript hace posible crear una nueva clase parecida a la clase anterior, pero con nuevas definiciones para algunas de sus propiedades. Tal como lo explica Haverbeke (2018), el prototipo de la nueva clase deriva del antiguo prototipo, pero agrega una nueva definición. A esto se le conoce como herencia en programación orientada a objetos y se dice que una nueva clase hereda propiedades y comportamientos de la clase padre.

En JavaScript se puede utilizar la palabra extends para indicar que una clase es hija de otra, por lo que deberá heredar sus características y métodos. Por ejemplo:

Explicación

```
// Clase padre
class Figura {
  constructor() {
    console.log("Soy una figura geométrica.");
  }
}
// Clase hija
class Triangulo extends Figura {
  constructor() {
  super();
  console.log("Soy un triángulo.");
  }
}
```

Figura 9. Uso de extends.

Cuando se implementa la clase hija, se puede observar la llamada a la función super(), esta lo que realiza es llamar al constructor de la clase padre, por tanto, antes de continuar, ejecuta las instrucciones del constructor de la clase padre y luego continúa con las instrucciones propias de la clase hija.

Las clases tienen la capacidad de realizar una misma acción de diferentes formas, a esta capacidad se le conoce como **polimorfismo**, es decir, una clase puede sobrescribir un método de una clase de la que hereda para emplear la misma funcionalidad, pero con alguna variante. Tomando el ejemplo anterior, si implementamos una nueva clase **Circulo**, se observará que realiza las acciones del padre, pero adicionalmente realiza algo más:

Explicación

```
// Clase padre
class Figura {
    constructor(x,y) {
        this.xPosition = x;
        this.yPosition = y;
      console.log("Soy una forma geométrica.");
    }
    obtenerArea(){ }
  }
// Clase hija
class Circulo extends Figura{
    constructor(x,y,radio){
        super(x,y,radio);
        this.radio=radio
    }
    obtenerArea(){ }
 }

let circulo = new Circulo(1,2,3)
```

Figura 10. Polimorfismo.

Otro concepto muy importante en la programación orientada a objetos es la encapsulación, que se podría definir como la capacidad para ocultar detalles que no son relevantes para el exterior. Dicho de otra manera, lo que hace es agrupar propiedades y métodos que actúan sobre esas propiedades, de manera que el acceso a estos datos está restringido fuera de la clase, función o paquete.

En otras palabras, con la encapsulación hacemos que las propiedades sean:

- Públicas: pueden ser observadas desde cualquier parte.
- Privadas: solo pueden ser observadas desde el interior del objeto, lo que se implementa utilizando los métodos **get** y **set**.

En JavaScript se utiliza un _ antes del nombre de la propiedad para indicar que la propiedad es privada, pero esto solamente es una convención entre programadores para indicar que no se debe cambiar esa variable de forma directa, sino que se debe utilizar un set.

Explicación

Otro concepto muy importante en la programación orientada a objetos es la encapsulación, que se podría definir como la capacidad para ocultar detalles que no son relevantes para el exterior. Dicho de otra manera, lo que hace es agrupar propiedades y métodos que actúan sobre esas propiedades, de manera que el acceso a estos datos está restringido fuera de la clase, función o paquete.

En otras palabras, con la encapsulación hacemos que las propiedades sean:

- Públicas: pueden ser observadas desde cualquier parte.
- Privadas: solo pueden ser observadas desde el interior del objeto, lo que se implementa utilizando los métodos **get** y **set**.

En JavaScript se utiliza un _ antes del nombre de la propiedad para indicar que la propiedad es privada, pero esto solamente es una convención entre programadores para indicar que no se debe cambiar esa variable de forma directa, sino que se debe utilizar un set.

```
Class Cafetera {
_cantidadAgua = 0;

set cantidadAgua(value) {
    if (value < 0) {
  value = 0;
    }
    this._cantidadAgua = value;
}

get cantidadAgua() {
    return this._cantidadAgua;
}

constructor(power) {
    this._power = power;
}
```

Figura 11. Encapsulación.

Tecmilenio no guarda relación alguna con las marcas mencionadas como ejemplo. Las marcas son propiedad de sus titulares conforme a la legislación aplicable, se utilizan con fines académicos y didácticos, por lo que no existen fines de lucro, relación publicitaria o de patrocinio.

Cierre

En este tema pudiste aprender los conceptos básicos de la programación orientada a objetos y que JavaScript no es un lenguaje orientado a objetos, sin embargo, a través del uso de prototipos, da la bienvenida a todos aquellos desarrolladores de POO. También pudiste aprender que JavaScript tiene maneras sencillas para crear y manipular prototipos y constructores, así como los diferentes prototipos que puedes crear y todos sus elementos.

También se revisó el concepto de herencia y cómo este facilita la comprensión de crear objetos más complejos sin la necesidad de reescribir muchas cosas que JavaScript ya hereda por defecto.

Referencias bibliográficas

- Haverbeke, M. (2018). *ELOQUENT JAVASCRIPT* (3ª ed.). Estados Unidos: No Starch Press.
- Kantor, I. (s.f.). *El lenguaje JavaScript*. Recuperado de https://es.javascript.info/js

Para saber más

Lecturas

- LenguajeJS. (s.f.). ¿Qué es la orientación a objetos? Recuperado de https://lenguajejs.com/javascript/oop/que-es/
- MDN. (s.f.). Conceptos básicos de los objetos JavaScript. Recuperado de https://developer.mozilla.org/es/docs/Learn/JavaScript/Objects/Basics
- Sulbaran, C. (2021). Programación Orientada a Objetos en JavaScript – Explicado con ejemplos. Recuperado de https://www.freecodecamp.org/espanol/news/programacion-orientada-a-objectos-en-javascript-explicado-con-ejemplos/

Videos

- Jonmircha. (2020, 9 de marzo). *Curso JavaScript: 22. Prototipos - #jonmircha* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=nS5FmAx4u_0
- Jonmircha. (2020, 10 de marzo). *Curso JavaScript: 23. Herencia Prototípica - #jonmircha* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=1-m7xtwvH1E
- Jonmircha. (2020, 11 de marzo). *Curso JavaScript: 24. Clases y Herencia - #jonmircha* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=C86JvqET61A
- Jonmircha. (2020, 12 de marzo). *Curso JavaScript: 25. Métodos estáticos, getters y setters - #jonmircha* [Archivo de video]. Recuperado de https://www.youtube.com/watch?v=TEzu31q9MVA

Checkpoints

Asegúrate de:

- Comprender los conceptos básicos de programación orientada a objetos en JavaScript.
- Construir clases e instanciarlas para crear objetos.
- Aplicar el uso de las clases y funciones constructoras dentro de JavaScript para que no te quede duda de por qué la teoría indica que JavaScript es orientado a objetos, pero no orientado a clases como los demás lenguajes.

Requerimientos técnicos

- Computadora con acceso a Internet.
- Editor de texto.
- Permisos de administrador y/o Git instalado previamente.

Prework

- Haber realizado la actividad guiada del tema 4, de preferencia durante la sesión.
- Leer detenidamente y comprender el material explicado en el tema.
- Practicar todos los ejemplos que se describen previamente en el tema.
- Revisar cada uno de los recursos adicionales que se proponen en este tema.