Objetos Literales, constructores y clases

1. Objetos Literales

Los objetos literales en JavaScript son una forma concisa y directa de crear objetos, usando una lista de pares clavevalor dentro de llaves “{}”. Cada clave es una propiedad del objeto, y el valor puede ser cualquier tipo de dato, incluyendo funciones (lo que convierte a esas propiedades en métodos).

Características:

- Simplicidad: Los objetos literales son fáciles de crear y leer. Son útiles para representar entidades únicas o configuraciones sencillas.
- Flexibilidad: Puedes agregar o modificar propiedades después de que el objeto ha sido creado.
- No necesitan “new”: A diferencia de los objetos creados por funciones constructoras o clases, los objetos literales no requieren el uso del operador “new”.

Ejemplo:

```
const persona = {
    nombre: "Gabriel",
    edad: 30,
    saludar: function() {
      console.log("Hola, soy " + this.nombre);
    }
  };

  // Acceso y modificación de propiedades
  persona.nombre = "Maria";
  persona.saludar(); // "Hola, soy Maria"
```

2. Prototypes de Atributos

El prototipo (“prototype”) en JavaScript es un mecanismo que permite a los objetos heredar propiedades y métodos de otros objetos. Cada objeto en JavaScript tiene un prototipo, que es otro objeto del cual hereda métodos y propiedades. Esto permite la reutilización de código y es una base para la herencia en JavaScript.

Características:

Compartición de métodos: Los métodos definidos en el prototipo son compartidos por todas las instancias del objeto, lo que ahorra memoria.

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

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

Persona.prototype.email = ""

const persona1 = new Persona("Gabriel", 30);
persona1.saludar(); // "Hola, soy Gabriel"
persona1.email = "pepe@hotmail.com"
```

Herencia: Cuando accedes a una propiedad o método de un objeto, JavaScript primero busca en el objeto mismo. Si no la encuentra, busca en el prototipo.

3. La Clase Object

“Object” es la clase base de la que todos los objetos en JavaScript heredan. Esto significa que cualquier objeto creado en JavaScript, ya sea a través de objetos literales, funciones constructoras o clases, tiene “Object” como su prototipo en algún punto de la cadena de prototipos.

Características:

Métodos heredados: Todos los objetos heredan métodos de “Object.prototype”, como “.toString()”, “.valueOf()”, y “.hasOwnProperty()”.

Personalización de métodos: Puedes sobrescribir los métodos heredados en tus propios objetos para personalizar su comportamiento.

Funciones utilitarias: “Object” también proporciona funciones estáticas útiles, como “Object.keys()”, “Object.values()”, “Object.assign()”, y “Object.create()”.

```
const persona = {
    nombre: "Gabriel",
    edad: 50
};

// Métodos heredados de Object.prototype
console.log(persona.toString()); // "[object Object]"
console.log(persona.hasOwnProperty("nombre")); // true

// Función utilitaria de Object
const copia = Object.assign({}, persona);
console.log(copia); // {nombre: "Gabriel", edad: 30}
```

Objetos constructores

Los objetos constructores en JavaScript son una manera de crear múltiples instancias de un objeto con las mismas propiedades y métodos, usando una función especial llamada función constructora. Este patrón permite la creación de objetos con una estructura común, al mismo tiempo que se proporciona flexibilidad para asignar valores diferentes a las propiedades de cada instancia.

1. Funciones Constructoras

Una función constructora es simplemente una función que, por convención, se define con una letra inicial en mayúscula y se utiliza con el operador "new". Cuando llamas a una función con "new", se crean los siguientes pasos automáticamente:

1. Crea un nuevo objeto vacío.
2. Asigna ese objeto al contexto ("this") dentro de la función.
3. Ejecuta el código dentro de la función.
4. Devuelve el nuevo objeto automáticamente (a menos que se devuelva explícitamente otro objeto).

```
function Producto(nombre, precio) {
    this.nombre = nombre;  // 'this' se refiere al nuevo objeto
    this.precio = precio;
}

const producto1 = new Producto("Televisor", 300);
const producto2 = new Producto("Lavadora", 400);

console.log(producto1.nombre); // "Televisor"
console.log(producto2.nombre); // "Lavadora"
```

2. Prototipos y Métodos Compartidos

Cuando usas una función constructora, es común que quieras que todas las instancias compartan ciertos métodos, en lugar de definir esos métodos dentro de la función constructora misma, lo cual sería ineficiente y desperdiciaría memoria. Para lograr esto, los métodos se definen en el "prototype" de la función constructora.

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

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

const persona1 = new Persona("Gabriel", 30);
const persona2 = new Persona("Maria", 25);

persona1.saludar(); // "Hola, soy Gabriel"
persona2.saludar(); // "Hola, soy Maria"
```

En este ejemplo, el método “saludar” es compartido por todas las instancias de “Persona”. Esto significa que “persona1” y “persona2” no tienen copias individuales del método, sino que acceden al mismo método a través del prototipo, lo que es más eficiente en términos de memoria.

3. Encapsulamiento y Privacidad

Las funciones constructoras en JavaScript no ofrecen encapsulamiento o privacidad de manera nativa como en otros lenguajes de programación. Sin embargo, se pueden utilizar variables locales dentro de la función constructora que no estén adjuntas al objeto “this” para simular la privacidad.

```javascript
function CuentaBancaria(titular, saldoInicial) {
    this.titular = titular;

    let saldo = saldoInicial; // Variable privada

    this.depositar = function(cantidad) {
        saldo += cantidad;
        console.log(`Saldo después del depósito: ${saldo}`);
    };

    this.verSaldo = function() {
        console.log(`Saldo actual: ${saldo}`);
    };
}

const cuenta = new CuentaBancaria("Gabriel", 1000);
cuenta.verSaldo(); // Saldo actual: 1000
cuenta.depositar(500); // Saldo después del depósito: 1500
cuenta.verSaldo(); // Saldo actual: 1500
```

En este ejemplo, la variable “saldo” es privada porque no está expuesta a través de “this”, por lo que no se puede acceder directamente desde fuera de la función.

4. Comparación con Clases

Las funciones constructoras son el equivalente más antiguo de las clases en JavaScript. Aunque las clases fueron introducidas en ECMAScript 2015 (ES6) para proporcionar una sintaxis más limpia y cercana a la de otros lenguajes orientados a objetos, internamente siguen usando el mismo mecanismo basado en prototipos que las funciones constructoras.

```
class Producto {
    constructor(nombre, precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    mostrar() {
        console.log(`Producto: ${this.nombre},
            Precio: ${this.precio}`);
    }
}

const producto1 = new Producto("Televisor", 300);
producto1.mostrar(); // "Producto: Televisor, Precio: 300"
```

Clases

Las clases en JavaScript son una forma más reciente de trabajar con la orientación a objetos, introducidas en ECMAScript 2015 (ES6). Aunque son una abstracción sintáctica sobre el sistema de prototipos existente, ofrecen una sintaxis más familiar para los desarrolladores que vienen de otros lenguajes orientados a objetos. Vamos a profundizar en los aspectos clave de las clases, incluyendo atributos privados, y el uso de métodos “get” y “set”.

1. Definición Básica de Clases

Una clase en JavaScript se define usando la palabra clave “class”. Dentro de una clase, se pueden definir un constructor (para inicializar objetos) y métodos que operan sobre esos objetos.

```
class Producto {
    constructor(nombre, precio) {
        this.nombre = nombre;
        this.precio = precio;
    }

    mostrar() {
        console.log(`Producto: ${this.nombre},
            Precio: ${this.precio}`);
    }
}

const producto1 = new Producto("Televisor", 300);
producto1.mostrar(); // "Producto: Televisor, Precio: 300"
```

2. Atributos Privados

Tradicionalmente, JavaScript no tenía soporte nativo para atributos privados, lo que significaba que todas las propiedades de un objeto eran públicas por defecto. Sin embargo, con la introducción de ECMAScript 2020, se introdujeron los campos privados en las clases, que se definen usando un “” antes del nombre de la propiedad.

```
class CuentaBancaria {
    #saldo; // Atributo privado

    constructor(titular, saldoInicial) {
        this.titular = titular;
        this.#saldo = saldoInicial;
    }

    depositar(cantidad) {
        this.#saldo += cantidad;
    }

    verSaldo() {
        console.log(`Saldo actual: ${this.#saldo}`);
    }
}

const cuenta = new CuentaBancaria("Gabriel", 1000);
cuenta.depositar(500);
cuenta.verSaldo(); // "Saldo actual: 1500"

// Error: no se puede acceder a un campo privado desde fuera de la clase
console.log(cuenta.#saldo);    No se puede acceder a la propiedad "#saldo" f
```

En este ejemplo, la propiedad “saldo” es completamente privada y no se puede acceder directamente fuera de la clase. Solo los métodos definidos dentro de la clase pueden interactuar con esta propiedad.

3. Métodos “get” y “set”

Los métodos “get” y “set” permiten controlar cómo se accede y modifica el valor de una propiedad. Son especialmente útiles para encapsular la lógica de acceso a propiedades y para validar o transformar los valores antes de asignarlos o devolverlos.

```
class Persona {
    #nombre; // Atributo privado
    constructor(nombre, edad) {
        this.#nombre = nombre;
        this.edad = edad;
    }

    // Método getter para nombre
    get nombre() {
        return this.#nombre;
    }

    // Método setter para nombre
    set nombre(nuevoNombre) {
        if(nuevoNombre.length > 0) {
            this.#nombre = nuevoNombre;
        } else {
            console.error("El nombre no puede estar vacío");
        }
    }
}

const persona1 = new Persona("Gabriel", 30);
console.log(persona1.nombre); // "Gabriel"

persona1.nombre = "Maria"; // Usa el setter para cambiar el nombre
console.log(persona1.nombre); // "Maria"

persona1.nombre = ""; // Intento de asignar un nombre vacío
// Error: El nombre no puede estar vacío
```

En este ejemplo, el método “get nombre” permite acceder al atributo privado “nombre”, mientras que el método “set nombre” permite modificarlo, con la adición de una validación que impide que se asigne un nombre vacío.

4. Ventajas de Atributos Privados y Métodos “get”/”set”

- Encapsulamiento y Seguridad: Los atributos privados permiten ocultar los detalles internos de una clase, evitando que se modifiquen o accedan directamente desde fuera de la clase.
- Control sobre las Propiedades: Los métodos “get” y “set” proporcionan un control adicional sobre cómo se acceden y modifican las propiedades, permitiendo validaciones o transformaciones de datos.
- Facilidad de Uso: Para el usuario de la clase, los métodos “get” y “set” se usan como propiedades regulares, lo que mejora la claridad y simplicidad del código.