«Talento Tech»

Back-End

Node JS

Clase 04

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 4 - Objetos, clases y operadores avanzados

Temario:

1. **Objetos:**
   - Definición
   - Características de los objetos
   - Tipos de objetos
     - Literales
     - Funcionales
2. **Clases:**
   - Sugar Syntax
   - Similitud con objetos funcionales
   - Método constructor
3. **Operadores avanzados**:
   - Destructuring operator
   - Spread operator

Objetivos de la Clase

En esta clase, los estudiantes comprenderán los objetos en JavaScript y aprenderán a crear y manipular tanto objetos literales como funcionales. Se introducirá la sintaxis de las clases en JavaScript, junto con el uso del operador destructuring para extraer valores de objetos y arrays de manera eficiente. Además, se explorará el operador spread como herramienta para clonar, combinar o manipular objetos y arrays. Finalmente, se identificarán escenarios en los que los operadores avanzados y las clases simplifican y mejoran la legibilidad del código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Objetos

Definición

En Javascript, existe un tipo de dato llamado `Object` del cual se desprenden los objetos.

Estos podríamos considerarlos como una variable especial que puede contener más variables en su interior. Se trata de una estructura que permite crear colecciones de datos que tienen sentido en conjunto pero que a diferencia de los arrays que se ordenan mediante un índice, en los objetos, cada **propiedad** o elemento es un par **clave/valor** separado de otro a través de una coma.

```
const hero = {
  alias: 'Cody',
  universe: 'TechLab',
  powers: ['fly', 'nightvision', 'stregth'],
  hit: 128,
  vitality: 100,
  decreaseVitality: function(damage) {
    return this.vitality - damage
  }
}
```

Los objetos son una excelente alternativa para crear colecciones con información relacionada y que precise ser identificada mediante una clave específica.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Características de los objetos

Pares clave/valor

Cada dato en un objeto está asociado a una clave única a la que se la conoce como **propiedad**. Normalmente se dice que un objeto está compuesto por propiedades.

```
const student1 = {
  name: 'Juan',
  age: 30,
  active: true
};
```

Dinamismo

Los objetos pueden expandirse o modificarse en tiempo de ejecución, añadiendo, eliminando o actualizando propiedades.

Esto representa un cambio de paradigma respecto de las variables, ya que no podemos crear variables en tiempo de ejecución pero si podemos lograr que un objeto guarde una propiedad que antes no existía.

```
const student1 = {
  name: 'Juan',
  age: 30,
  active: true
};

student1.address = 'Calle Falsa 123';
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Métodos

Además de almacenar datos, un objeto puede incluir funciones como valor de sus propiedades. A estas funciones se las llama **métodos**.

```
const calculator = {
        addition: (a, b) => a + b,
        subtract: (a, b) => a - b
    };
    console.log(calculator.addition(5, 3)); // 8
```

Prototipos

Todos los objetos en JavaScript heredan propiedades y métodos de un prototipo, lo que facilita la reutilización de lógica.

Objetos Literales

Los literales de los objetos en Javascript son las llaves {}. Declarar un objeto asignando a una variable un par de llaves que contenga propiedades es la manera más sencilla y tradicional de hacerlo.

Las propiedades son cada clave del objeto a las que se le asigna un valor. Ese valor puede ser de cualquier tipo de dato, incluso una función.

A continuación vemos un ejemplo de objeto con propiedades y valores:

```
    const user = {
      name: 'Jhon',
      lastName: 'Doe',
      age: 27,
      address: 'Fake street 123',
      isMarried: false,
      sayHi: () => console.log('Hi there, buddy')
    };
```

Nuestro objeto representa un usuario con sus propiedades y valores.

Podemos acceder al valor de cada propiedad utilizando el `.` (punto) seguido del nombre de la propiedad, como en el siguiente ejemplo:

```
console.log(user.name); // Jhon
console.log(user.age); // 27
console.log(user.sayHi()); // Hi there, buddy
```

Otra forma, es acceder colocando el **nombre de la propiedad** como un `string` dentro de corchetes, de la siguiente manera:

```
console.log(user['age']); // 27
console.log(user['isMarried']); // false
```

Por otra parte, podemos agregar propiedades adicionales a un objeto, luego de que este haya sido definido. Para ello simplemente accedemos a una propiedad nueva (que no exista en el objeto) y con el operador de asignación (`=`) le damos un valor.

```
const user = {
  name: 'Jhon',
  lastName: 'Doe',
  age: 27,
  address: 'Fake street 123',
  isMarried: false,
  sayHi: () => console.log('Hi there, buddy')
};

user.gender = 'Male';

console.log(user);
```

```
/**
 {
   name: 'Jhon',
   lastName: 'Doe',
   age: 27,
   address: 'Fake street 123',
   isMarried: false,
   sayHi: () => console.log('Hi there, buddy'),
   gender: 'Male'
 }
*/
```

Objetos Funcionales

A diferencia de los Literales, estos objetos se declaran como una función de javascript tradicional.

Para ello encontramos diversas maneras, el primer enfoque tiene un condimento más similar a un objeto de clase, haciendo uso de la palabra reservada `this` y creando variables accesibles desde fuera del objeto como si fuera un objeto literal.

Un objeto funcional en JavaScript es una forma de crear objetos utilizando una función constructora. Este patrón, común antes de la introducción de las clases en **ES6**, permite definir un "molde" para crear múltiples objetos con la misma estructura y comportamiento.

La función constructora actúa como una plantilla que puede inicializar las propiedades y métodos de cada instancia del objeto.

```
function Person(name, age) {
  this.name = name;
  this.age = age;

  this.sayHi = function () {
    console.log(`Hola, me llamo ${this.nombre} y
     tengo ${this.age} años.
   `);
  };
};

// Creación de instancias
const person1 = new Person("María", 30);
const person2 = new Person("Juan", 25);

person1.sayHi(); // Hola, me llamo María y tengo 30 años.
person2.sayHi(); // Hola, me llamo Juan y tengo 25 años.
```

Ahora el molde **Person** creado a través de la función denominada “función constructora” nos permite crear **objetos** basados en las **propiedades** y **métodos** previamente definidos. Para ello, podemos crear tantas variables como sean necesarias y asignarles mediante la palabra reservada **new**, seguida de la invocación de la función constructora, pasándole los argumentos requeridos, la creación de un nuevo objeto.

Ahora sí tenemos nuestros objetos **person1** y **person2** que pertenecen al tipo de objeto **Person**. Lo que hicimos fue crear un molde con el fin de reimplementar esta “fórmula” cada vez que necesitemos un nuevo objeto con las características de **Person**.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

La palabra reservada this

En el ejemplo anterior, se utiliza una palabra nueva llamada **this**, esta palabra tiene diferentes usos y significados dependiendo el contexto donde es utilizada.

En JavaScript, **this** es una referencia que apunta al contexto actual de ejecución. Su valor depende de cómo se invoque la función:

1. En una función constructora (objeto funcional) o clase:
   this se refiere al objeto recién creado por el uso de new.

```
function Person(name, age) {
  /*
  * this hace referencia a las variables
  * declaradas dentro de la función constructora
  */
  this.name = name;
  this.age = age;
  this.sayHi = function () {
    /*
    * aquí también invoca las variables definidas
dentro del
    * objeto para ser utilizadas en un método
interno.
    */
    console.log(`Hola, me llamo ${this.nombre} y
      tengo ${this.age} años.
    `);
  };
};
```

2. En un método de objeto:
   `this` hace referencia al objeto al que pertenece el método.

```
/**
 * Aquí this se utiliza dentro del
 * método (o función) de un objeto
 * literal para hacer referencia a
 * una propiedad dentro del
 * mismo objeto
 */
const hero = {
  alias: 'Cody',
  universe: 'TechLab',
  powers: ['fly', 'nightvision', 'stregth'],
  hit: 128,
  vitality: 100,
  decreaseVitality: function(damage) {
    return this.vitality - damage
  }
}
```

3. En funciones normales:
   El valor de `this` es `undefined` en entornos fuera del navegador o apunta al objeto global `window` en navegadores.
   - Si usamos Javascript fuera del navegador(como en Node) `this` tomará el valor de undefined
   - Si lo usamos en un entorno web, `this` hará referencia al objeto global window que contiene propiedades del navegador como el DOM (document object model).

```
function sayHi() {
  console.log(`Hi there i'm ${this}`);
}
```

4. En funciones flecha (=>):
   `this` conserva el valor del contexto en el que se definió la función.

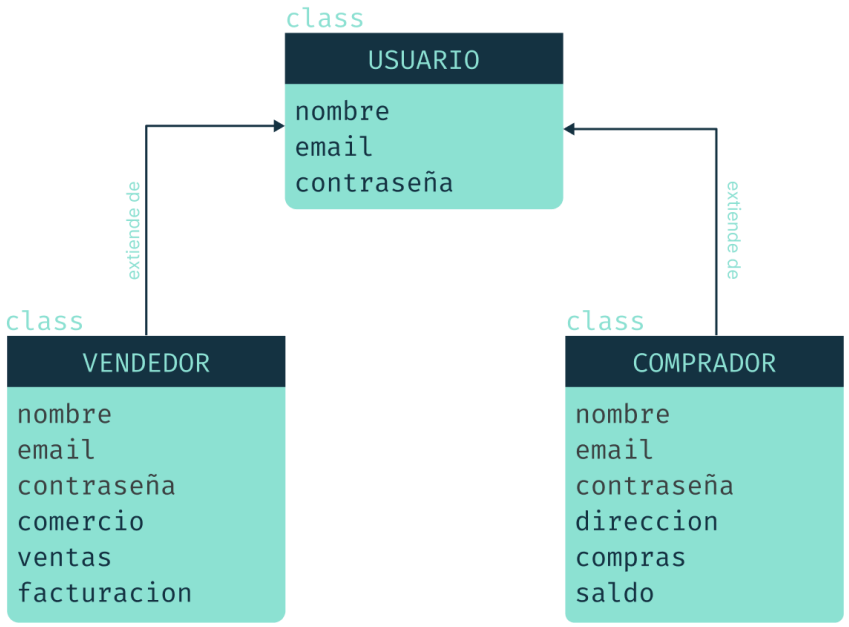
```
/* Aquí this devuelve undefined en un
* entorno de servidor o error en un
* navegador ya que window.vitality
* no existe.
*/

const hero = {
  alias: 'Cody',
  universe: 'TechLab',
  powers: ['fly', 'nightvision', 'stregth'],
  hit: 128,
  vitality: 100,
  decreaseVitality: (damage) => this.vitality - damage
}
```

Para evitar esto podemos cambiar `this.vitality` por `hero.vitality` o simplemente considerar cambiar el **arrow function** o función de flecha por una función expresada tradicional `function(){...}` donde el `this` si toma en cuenta el contexto como en el ejemplo del punto **2.**

Clases

Javascript es un lenguaje multiparadigma, lo que significa que soporta múltiples metodologías de programación, como la programación imperativa, la funcional o la programación orientada a objetos muy común en lenguajes como **Java** o **C#**.

La programación orientada a objetos o **POO** se basa en la creación de **Clases** que definen propiedades y comportamientos (métodos) de subclases o clases hijas que heredan estas definiciones, de hecho, los objetos funcionales vistos en la sección anterior, son una representación básica de lo que las clases pueden lograr.



La manera de crear clases en lenguajes basados enteramente en el uso de **POO**, es a través de la palabra reservada `class` y con la intención de atraer a programadores de estos lenguajes hace algunos años se agregó al estándar de EcmaScript la **"sugar syntax"** necesaria para contar con esta opción

> *Nota: La **sugar syntax** es una forma simplificada y más legible de escribir código, que enmascara una implementación más compleja sin cambiar su funcionalidad subyacente.

.

Gracias a esto, hoy en día Javascript permite utilizar el paradigma de orientación a objetos mediante el uso de una sintaxis mucho más parecida a la tradicional.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Veamos algunos ejemplos de cómo declarar y trabajar con clases:

En primer lugar creamos la `class User`

```
class User {
   constructor(name, email, password) {
     this.name = name;
     this.email = email;
     this.password = password;
   }

   login(inputEmail, password) {
     return this.email === inputEmail && this.password === password
       ? "Login successful!"
       : "Invalid email or password.";
   }
 }
```

Como se puede observar, la sintaxis es bastante similar a la de las **funciones constructoras**, solo que en lugar de la palabra `function` se utiliza la palabra `class`, se quitan los `()` y se utiliza lo que se conoce como el "método constructor" `constructor()` donde se definen los parámetros necesarios.

Además, agregamos un **método** `login` que compara el valor de `email` y `password`, devolviendo un mensaje según el resultado.

Lo interesante de las clases es que pueden **extender** su estructura a otras clases de la siguiente manera:

```
class Seller extends User {
  constructor(name, email, password, store, sells, incomes) {
    super(name, email, password);
    this.store = store;
    this.sells = sells;
    this.incomes = incomes;
  }
```

```
    addNewSale(amount) {
      this.sells += 1; // Incrementa el contador de ventas
      this.incomes += amount; // Agrega el monto al total de ingresos
      return `Sale added! Total sales: ${this.sells},
        Total incomes: ${this.incomes}`;
    }
  }
```

Utilizando la palabra reservada **extends** le indicamos a nuestra nueva clase **Seller** que "herede" todos los métodos y propiedades de **User** con la posibilidad de agregar nuevos métodos y propiedades únicos para esta clase. Del mismo modo, podemos crear una nueva clase **Buyer** que siga el mismo procedimiento:

```
class Buyer extends User {
  constructor(name, email, password, address, purchases, balance) {
    super(name, email, password);
    this.address = address;
    this.purchases = purchases;
    this.balance = balance;
  }

  makePurchase(amount) {
    if (this.balance >= amount) {
      this.balance -= amount; // Resta el monto del balance
      this.purchases += 1; // Incrementa el contador de compras
      return `Purchase successful! Remaining balance:
           ${this.balance}`;
    } else {
      return "Insufficient balance.";
    }
  }
}
```

Una vez que tenemos nuestras clases, podremos crear instancias de cualquiera de ellas:

```
const user1 = new User('John', 'j@correo.com', 'jhon123');
const seller1 = new Seller('Juan', 'j@correo.com', 'juan123', 'centro', 0, 0);
const buyer1 = new Buyer('Caro', 'c@correo.com', 'caro123', 'Calle 123', 0, 0);
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Lo curioso es que tanto `Seller` como `Buyer` pueden utilizar el método `login` perteneciente a `User`:

```
seller1.login('j@correo.com', 'juan123');
buyer1.login('c@correo.com', 'caro123');
```

Si bien esta sintaxis puede entenderse lejana de las anteriores el resultado final siguen siendo **objetos** de Javascript a los que se puede acceder a sus propiedades mediante el uso del `.` o del `['propiedad']` tal y como sucede con los objetos literales y los funcionales.

Cabe destacar que el **paradigma de programación orientada a objetos** es un concepto mucho más amplio del abordado en esta sección y posee aún más cosas por conocer.

Operadores Avanzados

Destructuring operator

El **destructuring** permite extraer valores de arrays o propiedades de objetos y asignarlos a variables de manera rápida y sencilla. Esto evita la necesidad de acceder a los valores mediante índices o claves repetidamente.

Destructuring en Arrays

Se utiliza para desempaquetar valores basados en su posición en el array:

```
const numeros = [1, 2, 3];

const [primero, segundo, tercero] = numeros;
console.log(primero); // 1
console.log(segundo); // 2
console.log(tercero); // 3
```

En este caso, optamos por nombre aleatorios (primero, segundo y tercero) que en función de cómo fueron ordenados, reclamarán el elemento del array en esa posición.

Destructuring en objetos

Se utiliza para extraer propiedades de un objeto y asignarlas a variables:

```
const persona = {
  nombre: "Ana",
  edad: 25,
};

const { nombre, edad } = persona;

console.log(nombre); // Ana
console.log(edad); // 25
```

A diferencia de los arrays, los nombres que asignamos a las variables deben ser idénticos a los de las propiedades del objeto y no de forma aleatoria. Además, estos pasan a ser nombres de variables en el scope donde son declarados, por lo que no podríamos crear otras variables con el mismo nombre.

Otras funciones disponibles mediante el destructuring, son las siguientes:

```
/* Utilizar valores predeterminados
en caso que el objeto no posea esa propiedad */

const { apellido = "No especifcado" } = persona;
console.log(apellido); // No especificado


/* Renombrar propiedades, en caso que ya exista
```

```
una variable con ese nombre */

const { nombre: nombrePersona } = persona;
console.log(nombrePersona); // Ana
```

```
/* Destructuring anidado, para cuando tenemos
objetos de más de 1 nivel de profundidad */

const usuario = {
  info: {
    nombre: "Luis",
    edad: 30
  }
};

const { info: { nombre, edad } } = usuario;
console.log(nombre); // Luis
```

Spread operator

El spread operator (`...`) permite expandir arrays u objetos en lugares donde se esperan múltiples elementos, como en argumentos de funciones, creación de nuevos objetos o arrays, y más.

Uso en Arrays

```
/* Combinar 2 o más arrays */

const a = [1, 2];
const b = [3, 4];
const combinado = [...a, ...b];
```

```
console.log(combinado); // [1, 2, 3, 4]

/* Clonar o copiar un array */

const original = [1, 2, 3];
const copia = [...original];
console.log(copia); // [1, 2, 3]
```

Utilizamos los `...` para “extraer” el contenido completo de un array y combinarlo con otro o realizar simplemente una copia sin modificar el array original.

Uso en Objetos

La metodología es bastante similar a los arrays y los resultados también:

```
/* Combinar 2 o más objetos */

const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const combinado = { ...obj1, ...obj2 };
console.log(combinado); // { a: 1, b: 2, c: 3, d: 4 }

/* Clonar o copiar un objeto */

const original = { a: 1, b: 2 };
const copia = { ...original };
console.log(copia); // { a: 1, b: 2 }
```

***Nota: Las propiedades duplicadas se sobrescriben con el valor más reciente.**

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Diferencias entre ambos

- **Destructuring** se usa para extraer valores de arrays u objetos en nuevas variables.
- **Spread Operator** se usa para expandir arrays u objetos en elementos individuales o para combinarlos.

Ejercicio Práctico.

Ejercicio 1 - La prueba de la flota:

Matías y Sabrina han preparado un nuevo reto para evaluar tus habilidades.

“Imagina que estás organizando información sobre una flota de vehículos”, dice Sabrina. “Queremos ver cómo manejas datos más estructurados”.

Misión 1:

1. Crea un array con 10 objetos, donde cada objeto represente un automóvil con la siguiente información:
    - Marca
    - Modelo
    - Año
    - Color

2. Usa un método de array para recorrer la lista e imprime por consola todos los datos de los automóviles cuyo **año sea mayor a 2018**.

Matías añade: “Queremos que veas esto como un ejercicio para prepararte para trabajar con datos reales en el futuro”.

Ejercicio 2 - Análisis de Colores:

Impresionados con tu avance, Matías y Sabrina suben un poco la dificultad. Sabrina te plantea:

“Queremos saber si puedes analizar la información de forma específica. Aquí tienes tu próximo desafío”.

Misión 2:

1. Crea una función que recorra el array de automóviles.
2. Usa **destructuring** dentro de la función para obtener el color de cada automóvil.
3. La función debe aceptar un color como parámetro y devolver por consola cuántos automóviles tienen ese color.

“Este tipo de habilidad es esencial para manejar sistemas dinámicos”, explica Matías.

¿Estás listo para superar esta prueba y demostrar tu capacidad para manipular datos con precisión? ¡Adelante!

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Materiales y Recursos Adicionales:

Documentación oficial de MDN: Objetos en JavaScript

Guía sobre funciones constructoras: JavaScript Function Constructor

Documentación oficial de MDN: Clases en JavaScript

Preguntas para Reflexionar:

1. ¿Cuándo es más útil utilizar objetos literales frente a clases o funciones constructoras?
2. ¿Qué ventajas ofrece el destructuring en la simplificación de código?
3. ¿En qué situaciones puede ser útil combinar objetos y arrays con el spread operator?

Próximos Pasos:

Módulos y librerías: Aprenderemos todo lo necesario para trabajar con código modular y de terceros.

Manejo de Promesas: Exploraremos cómo manejar procesos asíncronos utilizando promesas y `async/await` en JavaScript.

Servidores Web: Aprenderemos sobre la comunicación web y cómo funcionan los servidores que dan vida a internet.

Buenos Aires Ciudad