

***RECUERDA PONER A GRABAR LA
CLASE***



CODER HOUSE



Clase 2. Programación Backend

Principios básicos de Javascript



OBJETIVOS DE LA CLASE

- Comprender las estructuras y conceptos fundamentales al programar utilizando Javascript
- Conocer las ventajas y el uso de los nuevos elementos de lenguaje aportados por ES6

CRONOGRAMA DEL CURSO

Clase 1



**Principios de
programación
Backend**

Clase 2



**Principios básicos
de Javascript**

Clase 3



**Programación
sincrónica y
asincrónica**

REPASANDO...

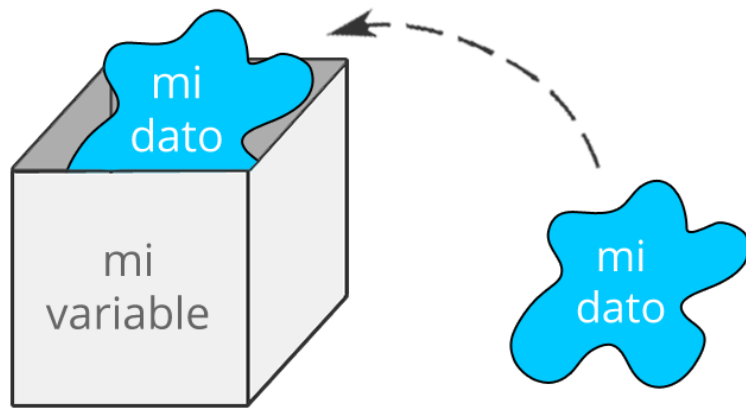
CODER HOUSE

Tipos de datos en Javascript

Variables y tipos de datos

Variable: es un espacio reservado para almacenar un dato que puede ser usado o modificado tantas veces como se desee.

Tipo de dato: es el atributo que especifica la clase de dato que almacena la variable.



Tipos de datos

- **Tipo Primitivos:** Incluyen a las cadenas de texto (String), variables booleanas cuyo valor puede ser true o false (Boolean) y números (Number). Además hay dos tipos primitivos especiales que son Null y Undefined. La copia es por valor.
- **Tipo Objeto:** Incluyen a los objetos (Object), a los arrays (Array) y funciones. La copia es por referencia.

TIPOS DE DATOS EN JAVASCRIPT		NOMBRE	DESCRIPCIÓN
	TIPOS PRIMITIVOS	String	Cadenas de texto
		Number	Valores numéricos
		Boolean	true ó false
		Null	Tipo especial, contiene null
		Undefined	Tipo especial, contiene undefined
	TIPOS OBJETO	Tipos predefinidos de JavaScript	Date (fechas) RegExp (expresiones regulares) Error (datos de error)
		Tipos definidos por el programador / usuario	Funciones simples Clases
		Arrays	Serie de elementos o formación tipo vector o matriz. Lo consideraremos un objeto especial que carece de métodos.
		Objetos especiales	Objeto global
			Objeto prototipo
			Otros

Javascript y ES6

EcmaScript 6

ES6 o EcmaScript 2015, fue una enorme revisión que surgió en el año 2015 y trajo -dentro de varias polémicas- enormes avances en el mundo de la programación JavaScript. Entre sus mayores innovaciones se encuentra la declaración de variables con `let` y `const`, la introducción de **clases** al lenguaje, y los *template strings*.

Variables en Javascript

Recordemos...

- Una variable es un **contenedor dinámico** que nos permite **almacenar valores**.
- Los valores pueden ser diversos **tipos de datos**, según la variable.
- Tal como lo indica su nombre, el **valor de la variable puede cambiar**, permitiéndonos crear programas que funcionen independientemente del valor de la variable.

Distintas maneras de crear variables en Javascript

Let y const

`let` y `const` son dos formas de declarar variables en JavaScript introducidas en ES6 que **limitan el ámbito de la variable** al **bloque** en que fue declarada (antes de ES6 esto **no** era así).

Es posible que se encuentren con ejemplos y código en internet utilizando la palabra reservada “var” para crear variables. Esta es la manera en que se hacía antes de ES6, y no se recomienda su uso!



Let

Un **bloque** en JavaScript se puede entender como “**lo que queda entre dos llaves**”, ya sean definiciones de funciones o bloques if, while, for y loops similares. Si una variable es declarada con let en el ámbito global o en el de una función, la variable pertenece al ámbito global o al ámbito de la función respectivamente.


```
let i = 0;
function foo() {
  i = 1;
  let j = 2;
  if(true) {
    console.log(i); // 1
    console.log(j); // 2
  }
}
foo();
```

```
function foo() {
  let i = 0;
  if(true) {
    // Sería otra variable i
    // sólo para el bloque if
    let i = 1;
    console.log(i); // 1
  }
  console.log(i); // 0
}
foo();
```

Ejemplo Let

Aquí la variable `i` es global y la variable `j` es local.

Pero si declaramos una variable con **let dentro un bloque**, que **a su vez está dentro de una función**, la variable pertenece solo a ese bloque.

```
function foo() {  
  if(true) {  
    let i = 1;  
  }  
  // ReferenceError:  
  // i is not defined  
  console.log(i);  
}  
foo();
```

Ejemplo Let

Fuera del bloque donde se declara con `let`, la **variable no** está **definida**.

Const

Al igual que con `let`, el **ámbito** (scope) para una **variable** declarada con `const` es el **bloque**.

Sin embargo, `const` además **prohíbe la reasignación de valores** (const viene de *constant*).

```
const i = 0;  
  
// TypeError:  
// Assignment to constant variable  
i = 1;
```

Ejemplo Const

Si se intenta reasignar una constante se obtendrá un **error**.

Mutabilidad y const

- Mientras que con `let` una variable puede ser reasignada, con `const` no es posible.
- Si se intenta reasignar una constante se obtendrá un error tipo `TypeError`.
- Pero que no se puedan reasignar **no significa que sean inmutables**.
- Si el valor de una constante es algo "mutable", como un array o un objeto, **se pueden cambiar los valores internos** de sus elementos.

NO REASIGNABLE \neq INMUTABLE

Ejemplo Mutabilidad

Por ejemplo, una constante se puede asignar a un objeto con determinadas propiedades. Aunque la constante no se pueda asignar a un nuevo valor, **sí se puede cambiar el valor de sus propiedades.**

```
const user = { name: 'Juan' };  
user.name = 'Manolo';  
console.log(user.name); // Manolo
```

Esto sería **posible**

```
const user = 'Juan';  
//TypeError: Assignment to constant  
user = 'Manolo';
```

Esto **NO** sería posible

Funciones en Javascript

Declaración

Declaración de una función

```
function nombre([param[,param[, ...param]]) {  
    instrucciones  
}
```

- **nombre:** Es el nombre de la función. **Se puede omitir**, en ese caso la función se conoce como **función anónima**.
- **param:** Es el nombre de un argumento que se pasará a la función. Una función puede tener hasta 255 argumentos.
- **instrucciones:** Son las instrucciones que forman el cuerpo de la función

Funciones Anónimas

```
function([param[,param[, ...param]]) {  
    instrucciones  
}
```

- Cuando una función se define **sin un nombre**, se conoce como una función anónima
- La función se almacena en la memoria, pero el tiempo de ejecución no crea automáticamente una referencia a la misma
- Hay varios escenarios donde las funciones anónimas son muy convenientes.

Usos de una función anónima

- Asignando una función anónima a una variable

```
var foo = function( ){ /*...*/ };
```

- Devolviendo una función anónima desde otra función

```
function foo( ) { return function( ){ /*...*/  
} };
```

- Invocando inmediatamente una función anónima

```
(function( ){ var foo = ' '; })( )
```

Funciones IIFE

Las expresiones de función ejecutadas inmediatamente (“IIFE”: *Immediately Invoked Function Expressions*), son funciones que **se ejecutan tan pronto como se definen**.

```
(function () {  
    statements  
})();
```

Se componen por dos partes

- función anónima con alcance léxico encerrado por el *Operador de Agrupación* `()`
- expresión de función cuya ejecución es inmediata `()`

Scope

Scope

- Indica el **ámbito o alcance actual de ejecución**.
- En él los **valores** y las **expresiones** son "**visibles**" o pueden ser referenciados.
- Una **función** sirve como un cierre en JavaScript y, por lo tanto, **crea un ámbito**
- Los Scope también **se pueden superponer** en una jerarquía, de modo que los Scope secundarios tengan acceso a los ámbitos primarios, pero no al revés.

Ejemplo Scope no válido

```
function exampleFunction() {  
  // x solo se puede utilizar en exampleFunction  
  const x = 'declarada en el scope local'  
  console.log(x)  
}  
  
console.log(x) // ReferenceError: x is not defined
```

Si la **variable** está **definida** exclusivamente **dentro de la función**, **no será accesible** desde **fuera** de la misma o desde **otras** funciones.

Ejemplo Scope válido

```
const x = 'declarada en el scope global'

function exampleFunction() {
  console.log(x) // x existe acá adentro
}

exampleFunction() // esto no lanza error

console.log(x) // x existe acá afuera también
```

El siguiente código es válido debido a que **la variable se declara fuera de la función**, lo que la hace **global**.

Closure

Closure

- Una clausura o closure es una función que **guarda referencias del estado adyacente** (ámbito léxico).
- En otras palabras, una clausura permite acceder al ámbito de una función exterior desde una función interior.
- En JavaScript, las clausuras **se crean cada vez que una función es creada.**

Ejemplo Closure

Un closure es un tipo especial de objeto que **combina** dos cosas: **una función, y el entorno** en que se creó la misma.

```
function crearGritarNombre(nombre) {  
  const signosDeExclamacion = '!!!'  
  return function () {  
    console.log(`${nombre}${signosDeExclamacion}`)  
  }  
}  
  
const gritarCH = crearGritarNombre('coderhouse')  
  
gritarCH() // muestra por pantalla: coderhouse!!!
```

El entorno está formado por las variables locales que estaban dentro del alcance en el momento que se creó el closure. En este caso, **gritarCH** es un closure que incorpora la función anónima, junto con el parámetro nombre y el string "!!!", que existían cuando se creó.

Template String

Template String

```
`texto de cadena de caracteres`  
  
`línea 1 de la cadena de caracteres  
línea 2 de la cadena de caracteres`  
  
`texto de cadena de caracteres ${expresión} texto adicional`
```

Las plantillas de texto (o Template Strings) **son cadenas literales de texto incrustadas en el código fuente** que permiten su interpolación mediante expresiones.

Características

- Los template string utilizan las **comillas invertidas** ``` (*grave accent o backtick*) para delimitar las cadenas, en lugar de las comillas sencillas o dobles.
- Si se utiliza **`${ }`** dentro de su expresión se habilita la **interpolación**, sustituyendo el fragmento por el valor al que apunta. Pueden ejecutar código en su interior.
- Soportan **texto multilínea**, manteniendo el formato introducido, incluyéndose los saltos de línea y las tabulaciones.



Funciones y Closures

Tiempo aproximado: 10 minutos

Funciones y Closures



- 1) Definir la función `mostrarLista` que reciba una lista de datos y muestre su contenido, si no está vacía, o de lo contrario muestre el mensaje: “lista vacía”. Luego, invocarla con datos de prueba para verificar que funciona bien en ambos casos.
- 2) Definir una función anónima que haga lo mismo que la del punto 1, e invocarla inmediatamente, pasando una lista con 3 números como argumento.
- 3) Definir la función `crearMultiplicador` que reciba un número y devuelva una función anónima que reciba segundo número y dé como resultado el producto de ambos.



BREAK

**¿Sabías que premiamos a nuestros
estudiantes por su dedicación
durante la cursada?**

Conocé los beneficios del TOP10

¡5/10 MINUTOS Y VOLVEMOS!

CODER HOUSE

Clases

Declaración de clases

```
class Cliente {  
    constructor (nombre, fecha, direccion) {  
        this.nombre = nombre;  
        this.fechaNacimiento = fecha;  
        this.direccion = direccion;  
    }  
}
```

Características

- El contenido de una clase es la parte que se encuentra **entre las llaves { }**. En ella se declaran los atributos y los métodos, tanto de instancia como de clase.
- Poseen un método *constructor* donde se declaran los atributos usando la palabra reservada *this*.
- Un constructor puede usar la palabra reservada *super* para llamar al constructor de una superclase.
- Las clases son sólo azúcar sintáctica, es decir, no son una nueva funcionalidad, solo una nueva manera de escribir lo que antes ya se podía pero de otra manera menos convencional.

Ejemplo: Clase Persona

```
class Persona {  
  constructor(nombre, edad) {  
    this.nombre = nombre  
    this.edad = apellido  
  }  
  
  static saludoCorto = 'hola'  
  
  saludoCompleto() {  
    console.log(`buenaaass, soy `${this.nombre}``)  
  }  
  
  saludoEstatico() {  
    console.log(Persona.saludoCorto)  
  }  
}
```

Operador new

Funcionamiento

El operador `new` permite **crear una instancia de un tipo de objeto definido por el usuario**. Se utiliza sobre una clase.

Realiza básicamente 3 tareas en la construcción

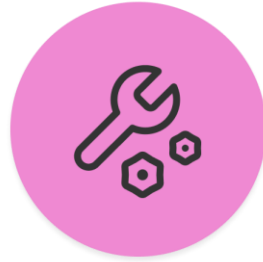
1. **Crea** un objeto vacío
2. **Ejecuta** el constructor de la clase en el contexto del objeto creado
3. **Retorna** el objeto

Ejemplo Operador new con class

```
const p = new Persona('pepe', 5)

console.log(p)

// muestra por pantalla:
// Persona { nombre: 'pepe', edad: 5 }
```



Clases

Tiempo aproximado: 15 minutos

Clases



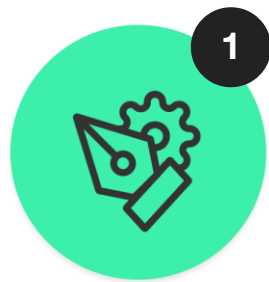
En este ejercicio construiremos una herramienta que permita que diferentes personas puedan llevar cuentas individuales sobre algo que deseen contabilizar, al mismo tiempo que nos brinde una contabilidad general del total contado. Para ello:

- 1) Definir la clase Contador.
- 2) Cada instancia de contador debe ser identificada con el nombre de la persona responsable de ese conteo.
- 3) Cada instancia inicia su cuenta individual en cero.
- 4) La clase en sí misma posee un valor estático con el que lleva la cuenta de todo lo contado por sus instancias, el cual también inicia en cero.

Clases



- 4) Definir un método `obtenerResponsable` que devuelva el nombre del responsable de la instancia.
- 5) Definir un método `obtenerCuentaIndividual` que devuelva la cantidad contada por la instancia.
- 6) Definir un método `obtenerCuentaGlobal` que devuelva la cantidad contada por todos los contadores creados hasta el momento.
- 7) Definir el método `contar` que incremente en uno tanto la cuenta individual como la cuenta general

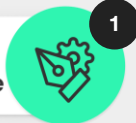


Clases

CLASES

Formato: Un documento de texto con nombre de archivo “ApellidoNombre” con que cumpla la siguiente consigna.

Desafío
entregable



>> Consigna:

1) Declarar una clase Usuario

2) Hacer que Usuario cuente con los siguientes atributos:

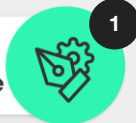
- *nombre*: String
- *apellido*: String
- *libros*: Object[]
- *mascotas*: String[]

Los valores de los atributos se deberán cargar a través del constructor, al momento de crear las instancias.

CLASES

Formato: Un documento de texto con nombre de archivo “ApellidoNombre” con que cumpla la siguiente consigna.

Desafío
entregable



3) Hacer que Usuario cuente con los siguientes métodos:

- `getFullName(): String`. Retorna el completo del usuario. *Utilizar template strings.*
- `addMascota(String): void`. Recibe un nombre de mascota y lo agrega al array de mascotas.
- `countMascotas(): Number`. Retorna la cantidad de mascotas que tiene el usuario.
- `addBook(String, String): void`. Recibe un string 'nombre' y un string 'autor' y debe agregar un objeto: { nombre: String, autor: String } al array de libros.
- `getBookNames(): String[]`. Retorna un array con sólo los nombres del array de libros del usuario.

4) Crear un objeto llamado usuario con valores arbitrarios e invocar todos sus métodos.

CLASES

>> Ejemplos:

- *countMascotas*: Suponiendo que el usuario tiene estas mascotas: ['perro', 'gato']
`usuario.countMascotas()` debería devolver 2.
- *getBooks*: Suponiendo que el usuario tiene estos libros: [{nombre: 'El señor de las moscas', autor: 'William Golding'}, {nombre: 'Fundacion', autor: 'Isaac Asimov'}]
`usuario.getBooks()` debería devolver ['El señor de las moscas', 'Fundacion'].
- *getFullName*: Suponiendo que el usuario tiene: nombre: 'Elon' y apellido: 'Musk'
`usuario.getFullName()` debería devolver 'Elon Musk'

¿PREGUNTAS?



PARA LA PRÓXIMA CLASE

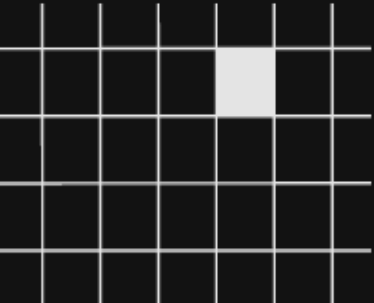
Descargar VSCode
(o editor de código de tu preferencia)

Instalar la última versión de NodeJS
(en este momento es la versión 16.x.x)



¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Conceptos de programación en Javascript
 - Novedades de ES6
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDOLAEDUCACIÓN