Estructuras de Datos

Ing. En Sistemas Computacionales – 3er Semestre agosto – diciembre 2023

JOSE ARTURO BUSTAMANTE LAZCANO

Contenido

[Semana 1 – Unidad 1 - Clasificación de las estructuras de datos 1](#_Toc147732639)

[Introducción a las Estructuras de datos 1](#_Toc147732640)

[Semana 2 – Unidad 1 – Tipos de datos Abstractos 2](#_Toc147732641)

[Tipos de datos abstractos 2](#_Toc147732642)

[Manejo de memoria 8](#_Toc147732643)

[Semana 3 – Unidad 1 – Cierre de la unidad 10](#_Toc147732644)

[Ejemplos de estructuras 10](#_Toc147732645)

[Examen unidad 1 13](#_Toc147732646)

[Semana 4 – Unidad 2 Resumen de la unidad 1 y códigos para Unidad 2 14](#_Toc147732647)

[Tipos de datos 14](#_Toc147732648)

[Operadores aritméticos 15](#_Toc147732649)

[Concatenación de cadenas 17](#_Toc147732650)

[Métodos 18](#_Toc147732651)

[Objetos incorporados 20](#_Toc147732652)

[Instrucciones 21](#_Toc147732653)

[Revisar 23](#_Toc147732654)

[Versiones de JavaScript: ES6 y anteriores 28](#_Toc147732655)

[Semana 5 – Unidad 2 Recursividad - Definición 32](#_Toc147732656)

[Semana 6 – Unidad 2 Recursividad - Procedimientos recursivos. 38](#_Toc147732657)

[console.log() 39](#_Toc147732658)

[JavaScript 39](#_Toc147732659)

[Methods 39](#_Toc147732660)

[Built-in Objects 39](#_Toc147732661)

[Numbers 39](#_Toc147732662)

[String .length 40](#_Toc147732663)

[Data Instances 40](#_Toc147732664)

[Booleans 40](#_Toc147732665)

[Math.random() 40](#_Toc147732666)

[Math.floor() 40](#_Toc147732667)

[Single Line Comments 40](#_Toc147732668)

[Null 41](#_Toc147732669)

[Strings 41](#_Toc147732670)

[Arithmetic Operators 41](#_Toc147732671)

[Multi-line Comments 41](#_Toc147732672)

[Remainder / Modulo Operator 42](#_Toc147732673)

[Assignment Operators 42](#_Toc147732674)

[String Interpolation 43](#_Toc147732675)

[Variables 43](#_Toc147732676)

[Undefined 43](#_Toc147732677)

[Learn Javascript: Variables 43](#_Toc147732678)

[Declaring Variables 44](#_Toc147732679)

[Template Literals 44](#_Toc147732680)

[let Keyword 44](#_Toc147732681)

[const Keyword 45](#_Toc147732682)

[String Concatenation 45](#_Toc147732683)

[Control Flow 45](#_Toc147732684)

[Logical Operator || 45](#_Toc147732685)

[Ternary Operator 46](#_Toc147732686)

[else Statement 46](#_Toc147732687)

[Logical Operator && 46](#_Toc147732688)

[switch Statement 47](#_Toc147732689)

[if Statement 47](#_Toc147732690)

[Logical Operator ! 48](#_Toc147732691)

[Comparison Operators 48](#_Toc147732692)

[else if Clause 48](#_Toc147732693)

[Truthy and Falsy 49](#_Toc147732694)

[Arrow Functions (ES6) 49](#_Toc147732695)

[Functions 50](#_Toc147732696)

[Anonymous Functions 51](#_Toc147732697)

[Function Expressions 51](#_Toc147732698)

[Function Parameters 51](#_Toc147732699)

[return Keyword 52](#_Toc147732700)

[Function Declaration 52](#_Toc147732701)

[Calling Functions 52](#_Toc147732702)

[Scope 53](#_Toc147732703)

[Block Scoped Variables 53](#_Toc147732704)

[Global Variables 54](#_Toc147732705)

[Property .length 54](#_Toc147732706)

[Index 54](#_Toc147732707)

[Method .push() 55](#_Toc147732708)

[Method .pop() 55](#_Toc147732709)

[Mutable 55](#_Toc147732710)

[Arrays 56](#_Toc147732711)

[Reverse Loop 56](#_Toc147732712)

[Do…While Statement 56](#_Toc147732713)

[For Loop 57](#_Toc147732714)

[Looping Through Arrays 57](#_Toc147732715)

[Break Keyword 57](#_Toc147732716)

[Nested For Loop 58](#_Toc147732717)

[Loops 58](#_Toc147732718)

[While Loop 59](#_Toc147732719)

[Functions Assigned to Variables 59](#_Toc147732720)

[Callback Functions 60](#_Toc147732721)

[Higher-Order Functions 60](#_Toc147732722)

[JavaScript Functions: First-Class Objects 60](#_Toc147732723)

[The .reduce() Method 61](#_Toc147732724)

[The .forEach() Method 62](#_Toc147732725)

[The .filter() Method 62](#_Toc147732726)

[The .map() Method 62](#_Toc147732727)

[Restrictions in Naming Properties 63](#_Toc147732728)

[Dot Notation for Accessing Object Properties 63](#_Toc147732729)

[Objects 64](#_Toc147732730)

[Accessing non-existent JavaScript properties 64](#_Toc147732731)

[JavaScript Objects are Mutable 64](#_Toc147732732)

[JavaScript for...in loop 65](#_Toc147732733)

[Properties and values of a JavaScript object 65](#_Toc147732734)

[Delete operator 65](#_Toc147732735)

[javascript passing objects as arguments 66](#_Toc147732736)

[JavaScript Object Methods 67](#_Toc147732737)

[JavaScript destructuring assignment shorthand syntax 67](#_Toc147732738)

[*shorthand property name* syntax for object creation 68](#_Toc147732739)

[this Keyword 68](#_Toc147732740)

[javascript function this 69](#_Toc147732741)

[JavaScript Arrow Function this Scope 69](#_Toc147732742)

[getters and setters intercept property access 70](#_Toc147732743)

[javascript factory functions 70](#_Toc147732744)

[javascript getters and setters restricted 71](#_Toc147732745)

[Semana 7 – Unidad 3 Estructuras lineales - Pilas. 72](#_Toc147732746)

[Semana 8 – Unidad 3 Estructuras lineales - Representación en memoria y Operaciones básicas. 81](#_Toc147732747)

# Semana 1 – Unidad 1 - Clasificación de las estructuras de datos

21, 22 y 24 de agosto.

## Introducción a las Estructuras de datos

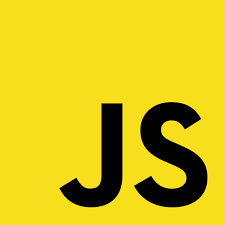
Las estructuras de datos son formas de organizar y almacenar datos en la memoria de una computadora de manera eficiente y accesible. Se pueden clasificar en varias categorías según su organización y funcionalidad.

Las estructuras de datos se pueden clasificar de varias maneras, pero una de las formas más comunes es por su organización. Las estructuras de datos lineales son aquellas en las que los datos están organizados en una secuencia, como una lista, una pila o una cola. Las estructuras de datos no lineales son aquellas en las que los datos no están organizados en una secuencia, como un árbol o un grafo.

Otra forma de clasificar las estructuras de datos es por su tamaño. Las estructuras de datos estáticas son aquellas en las que el tamaño de los datos es fijo, mientras que las estructuras de datos dinámicas son aquellas en las que el tamaño de los datos puede cambiar durante la ejecución del programa.

Es importante elegir la estructura de datos adecuada según los requisitos del problema y las operaciones que se realizan con los datos. Cada estructura de datos tiene sus propias ventajas y desventajas en términos de eficiencia y complejidad en las operaciones.

Por último, las estructuras de datos también se pueden clasificar por su tipo. Las estructuras de datos primitivas son aquellas que almacenan datos simples, como números, cadenas o booleanos. Las estructuras de datos complejas son aquellas que almacenan datos compuestos, como listas, árboles o grafos.



Conclusión: 🤔

# Semana 2 – Unidad 1 – Tipos de datos Abstractos

28, 29 y 31 de agosto.

## Tipos de datos abstractos

Los tipos de datos abstractos (ADT) son tipos de datos que definen un conjunto de operaciones que se pueden realizar sobre los datos, pero no especifican cómo se implementan los datos. Los ADT se utilizan para abstraer los detalles de la implementación de los datos de la lógica de la aplicación. Esto hace que el código sea más portable y mantenible, ya que los cambios en la implementación de los datos no requieren cambios en la lógica de la aplicación.

Los ADT se suelen implementar como clases en un lenguaje de programación orientado a objetos. Las clases definen los datos y las operaciones que se pueden realizar sobre ellos. Los objetos de las clases son instancias de los ADT.

Algunos ejemplos de ADT comunes son:

Listas: una lista es una colección de elementos ordenados. Las operaciones típicas que se pueden realizar sobre una lista son agregar, eliminar y buscar elementos.

Pilas: una pila es una colección de elementos ordenados en la que los elementos se agregan y eliminan al final de la pila.

Colas: una cola es una colección de elementos ordenados en la que los elementos se agregan al final de la cola y se eliminan del principio de la cola.

Diccionarios: un diccionario es una colección de pares de valores clave. Las operaciones típicas que se pueden realizar sobre un diccionario son agregar, eliminar y buscar elementos por clave.

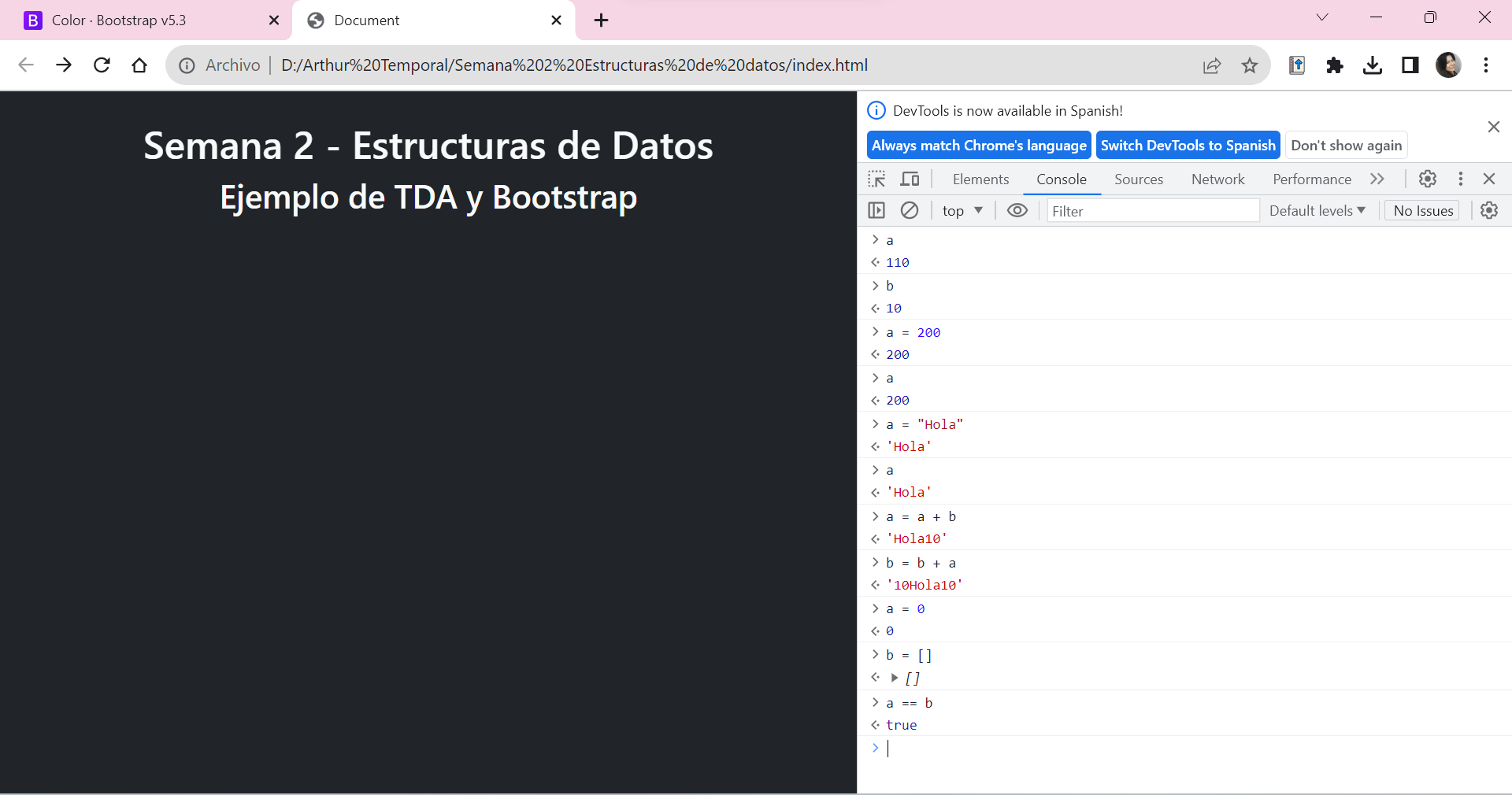
Los ADT se utilizan en una amplia gama de aplicaciones, incluyendo:

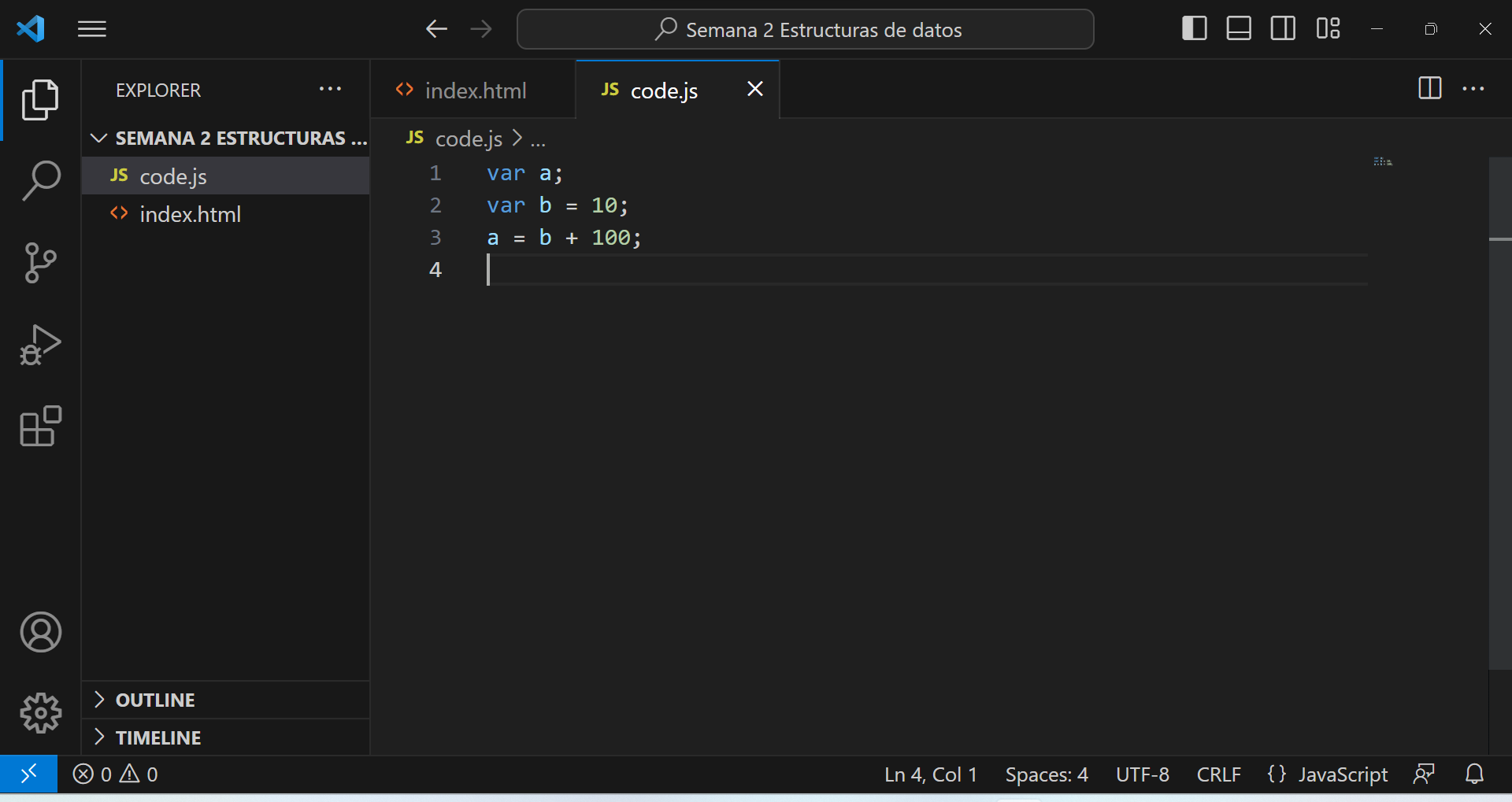
Programación de sistemas: los ADT se utilizan para representar datos abstractos en sistemas operativos, compiladores y otras aplicaciones de software de sistemas.

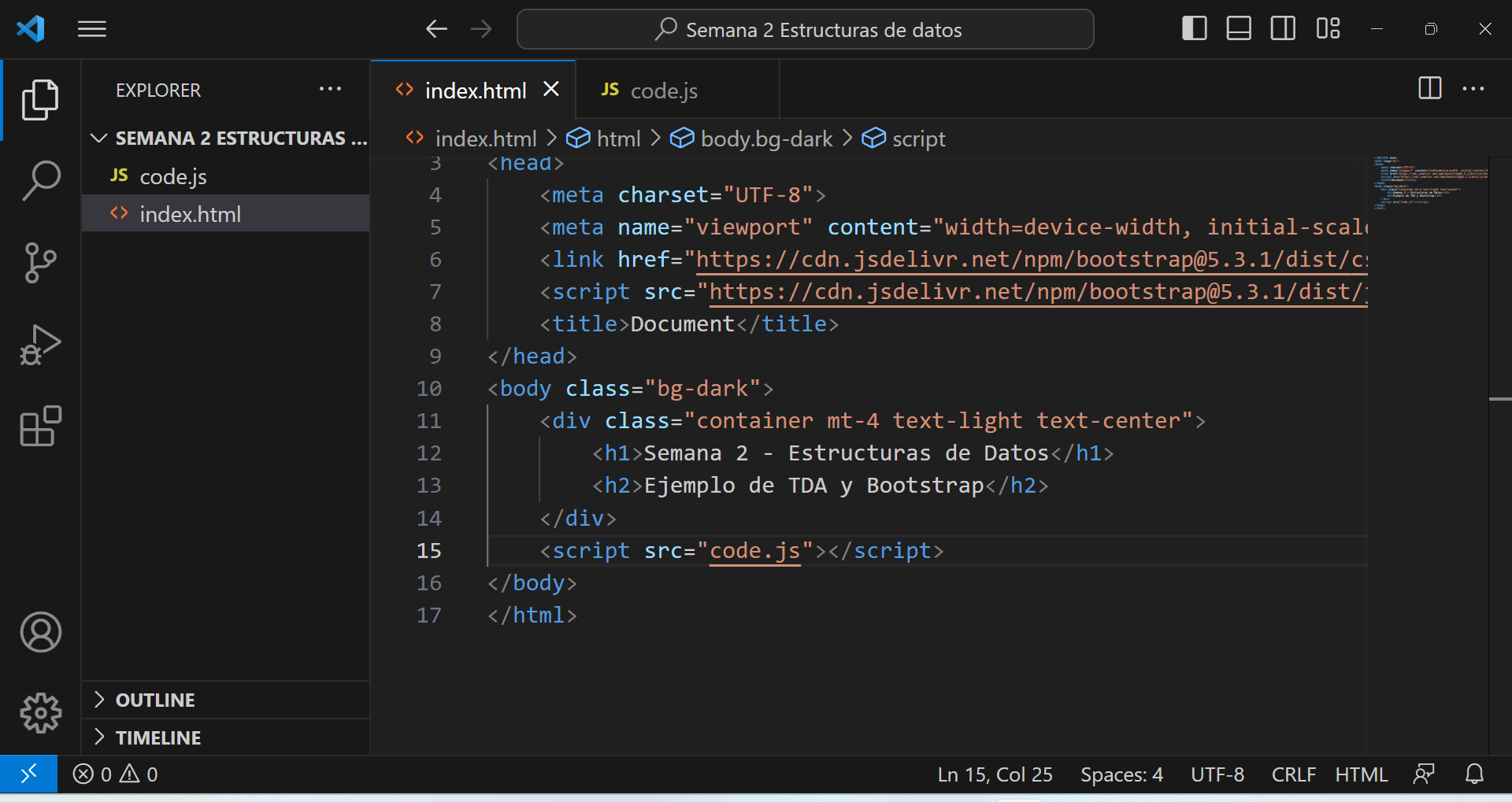
Programación de aplicaciones: los ADT se utilizan para representar datos abstractos en aplicaciones de usuario final, como procesadores de texto, hojas de cálculo y bases de datos.

Los ADT son una herramienta poderosa que puede ayudar a los programadores a escribir código más portable y mantenible.

Ejemplos de TDA’s







Notas:

Una variable de tipo numérico que es sumada con una variable de tipo carácter cambia su tipo de dato siempre a carácter

Las variables pueden cambiar el entorno de almacenamiento a: Variable, Arreglo y Objeto.

Las variables pueden cambiar de tipo de dato en cualquier momento.

Referencia: <https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array>

Los arreglos son listas ordenadas que van de la posición 0 a la n por lo tanto su tamaño se calcula como n - 1

Ejemplos en el navegador:

a

110

b

10

a = 200

200

a

200

a = "Hola"

'Hola'

a

'Hola'

a = a + b

'Hola10'

b = b + a

'10Hola10'

a = 0

0

b = []

[]

a == b

true

a = []

[]

b = 0

0

b

0

b = 0.7

0.7

b = "Hola"

'Hola'

b = 10

10

b = []

[]

b = ["Manzana","Pera","Durazno","Mango"]

(4) ['Manzana', 'Pera', 'Durazno', 'Mango']0: "Manzana"1: "Pera"2: "Durazno"3: "Mango"length: 4[[Prototype]]: Array(0)at: ƒ at()concat: ƒ concat()constructor: ƒ Array()copyWithin: ƒ copyWithin()entries: ƒ entries()every: ƒ every()fill: ƒ fill()filter: ƒ filter()find: ƒ find()findIndex: ƒ findIndex()findLast: ƒ findLast()findLastIndex: ƒ findLastIndex()flat: ƒ flat()flatMap: ƒ flatMap()forEach: ƒ forEach()includes: ƒ includes()indexOf: ƒ indexOf()join: ƒ join()keys: ƒ keys()lastIndexOf: ƒ lastIndexOf()length: 0map: ƒ map()pop: ƒ pop()push: ƒ push()reduce: ƒ reduce()reduceRight: ƒ reduceRight()reverse: ƒ reverse()shift: ƒ shift()slice: ƒ slice()some: ƒ some()sort: ƒ sort()splice: ƒ splice()toLocaleString: ƒ toLocaleString()toReversed: ƒ toReversed()toSorted: ƒ toSorted()toSpliced: ƒ toSpliced()toString: ƒ toString()unshift: ƒ unshift()values: ƒ values()with: ƒ with()Symbol(Symbol.iterator): ƒ values()Symbol(Symbol.unscopables): {at: true, copyWithin: true, entries: true, fill: true, find: true, …}[[Prototype]]: Object

b = ["Mandarina"]

['Mandarina']0: "Mandarina"length: 1[[Prototype]]: Array(0)

b = 4

Un ejemplo de una estructura de datos en JavaScript es un array. Un array es una colección de datos del mismo tipo, que están indexados. Los arrays se pueden utilizar para almacenar una variedad de datos, como números, cadenas, objetos y funciones.

Aquí hay un ejemplo de cómo crear un array en JavaScript:

JavaScript

const myArray = [1, 2, 3, 4, 5];

Usa el código con precaución. Más información

Este array contiene cinco números enteros.

Los arrays se pueden manipular mediante una variedad de métodos, como push(), pop(), shift() y unshift(). Estos métodos se utilizan para agregar, eliminar y mover elementos en un array.

Aquí hay un ejemplo de cómo utilizar algunos de estos métodos:

JavaScript

// Add a new element to the end of the array

myArray.push(6);

// Remove the last element from the array

myArray.pop();

// Remove the first element from the array

myArray.shift();

// Add a new element to the beginning of the array

myArray.unshift(0);

Usa el código con precaución. Más información

Este código agregará un nuevo número entero al final del array, eliminará el último elemento del array, eliminará el primer elemento del array y agregará un nuevo elemento entero al principio del array.

Los arrays son una estructura de datos muy versátil que se puede utilizar para una variedad de tareas. Son una parte esencial de la programación en JavaScript.

Aquí hay algunos otros ejemplos de estructuras de datos en JavaScript:

Objetos: Un objeto es una colección de datos de diferentes tipos, que están indexados por claves. Los objetos se utilizan para almacenar datos complejos, como información de contacto o configuraciones.

Matrices: Una matriz es una estructura de datos similar a un array, pero que puede almacenar datos de diferentes tipos. Las matrices se utilizan para almacenar datos de gran tamaño.

Listas enlazadas: Una lista enlazada es una estructura de datos que almacena datos en forma de nodos. Los nodos están conectados entre sí por enlaces. Las listas enlazadas se utilizan para almacenar datos de forma eficiente.

Arboles binarios: Un árbol binario es una estructura de datos que almacena datos en forma de árbol. Los árboles binarios se utilizan para almacenar datos jerárquicos.

Estas son solo algunas de las muchas estructuras de datos disponibles en JavaScript. El tipo de estructura de datos que se utiliza depende de la tarea que se quiera realizar.

## Manejo de memoria

Las estructuras de datos son una forma de organizar datos de manera eficiente para que puedan ser manipulados de forma rápida y fácil. El manejo de memoria es el proceso de asignar y liberar memoria para las estructuras de datos.

Existen dos tipos principales de manejo de memoria: estático y dinámico.

Manejo de memoria estático

El manejo de memoria estático asigna memoria para las estructuras de datos en tiempo de compilación. Esto significa que el tamaño de la memoria asignada no puede cambiar durante la ejecución del programa.

El manejo de memoria estático es simple y eficiente, pero puede ser inflexible. Por ejemplo, si se necesita más memoria para una estructura de datos, no se puede asignar más memoria en tiempo de ejecución.

Manejo de memoria dinámico

El manejo de memoria dinámico asigna memoria para las estructuras de datos en tiempo de ejecución. Esto significa que el tamaño de la memoria asignada puede cambiar durante la ejecución del programa.

El manejo de memoria dinámico es más flexible que el manejo de memoria estático, pero puede ser menos eficiente. Por ejemplo, asignar y liberar memoria en tiempo de ejecución puede requerir más operaciones que asignar memoria en tiempo de compilación.

Relaciones entre estructuras de datos y manejo de memoria

Las estructuras de datos y el manejo de memoria están estrechamente relacionados. El tipo de manejo de memoria que se utiliza puede afectar al rendimiento y la eficiencia de las estructuras de datos.

Por ejemplo, las estructuras de datos que requieren grandes cantidades de memoria pueden beneficiarse del manejo de memoria dinámico. Esto se debe a que el manejo de memoria dinámico permite asignar más memoria a las estructuras de datos en tiempo de ejecución, si es necesario.

Por otro lado, las estructuras de datos que se utilizan con frecuencia pueden beneficiarse del manejo de memoria estático. Esto se debe a que el manejo de memoria estático puede asignar memoria para las estructuras de datos en tiempo de compilación, lo que puede mejorar el rendimiento.

En conclusión

El manejo de memoria es un aspecto importante del diseño y la implementación de estructuras de datos. El tipo de manejo de memoria que se utiliza puede afectar al rendimiento y la eficiencia de las estructuras de datos.

a = []

[]

a.push(100)

1

a

[100]

a.push(200)

2

a

(2) [100, 200]

a.push(300)

3

# Semana 3 – Unidad 1 – Cierre de la unidad

04 de Septiembre

## Ejemplos de estructuras

Para mostrar una variable de JavaScript en un documento HTML, puedes utilizar la manipulación del DOM (Document Object Model) para actualizar el contenido de un elemento HTML con el valor de la variable JavaScript. Aquí hay un ejemplo sencillo de cómo hacerlo:

Supongamos que tienes una variable en JavaScript llamada miVariable y quieres mostrar su valor en un elemento HTML con el id "resultado". Aquí está el código HTML y JavaScript correspondiente:

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Mostrar Variable en HTML</title>

</head>

<body>

<!-- Un elemento donde mostrarás la variable -->

<p>El valor de miVariable es: <span id="resultado"></span></p>

<script>

// Tu variable JavaScript

var miVariable = "Hola, esto es un ejemplo.";

// Obtén una referencia al elemento donde deseas mostrar la variable

var elementoResultado = document.getElementById("resultado");

// Asigna el valor de la variable al contenido del elemento HTML

elementoResultado.textContent = miVariable;

</script>

</body>

</html>

En este ejemplo, hemos creado un elemento p con un span dentro de él con el id "resultado". Luego, en JavaScript, hemos obtenido una referencia a ese elemento utilizando getElementById y hemos asignado el valor de miVariable al contenido de ese elemento utilizando la propiedad textContent.

Cuando cargues este HTML en tu navegador, verás que el valor de miVariable se muestra en la página web. Puedes adaptar este ejemplo para mostrar cualquier variable en tu HTML según tus necesidades.

//Trabajando Variables

//Var de tipo entero

var entero = 10;

//Var de tipo Flotante

var real = 20.89;

//Var de tipo Cadena

var cadena = "Mi Saludo";

//Var de tipo Boleano

var banderaT = true;

var banderaF = false;

var elemento1 = document.getElementById("miEntero");

elemento1.textContent = entero;

var elemento2 = document.getElementById("miReal");

elemento2.textContent = real;

var elemento3 = document.getElementById("miCadena");

elemento3.textContent = cadena;

var elemento4 = document.getElementById("miBanderaT");

elemento4.textContent = banderaT;

var elemento5 = document.getElementById("miBanderaF");

elemento5.textContent = banderaF;

//Trabajando con arreglos

//Arreglo con datos constantes

var arregloTipo1 = [20,90,10,7,8,12];

//Arreglo con datos variados

var arregloTipo2 = ["uno",1,"dos",2,"tres",3];

function mostrarLista1(){

    var lista = document.getElementById("miLista1");

    for(var i=0; i<arregloTipo1.length; i++){

        var elementoLista = document.createElement("li");

        elementoLista.textContent = arregloTipo1[i];

        lista.appendChild(elementoLista);

    }

}

mostrarLista1();

function mostrarLista2(){

    var lista = document.getElementById("miLista2");

    for(var i=0; i<arregloTipo2.length; i++){

        var elementoLista = document.createElement("li");

        elementoLista.textContent = arregloTipo2[i];

        lista.appendChild(elementoLista);

    }

}

mostrarLista2();

# Examen unidad 1

Pegar el código HTML y JS

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<CODIGO INDIVIDUAL>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>



# Semana 4 – Unidad 2 Resumen de la unidad 1 y códigos para Unidad 2

11, 12 y 14

## Tipos de datos

Los tipos de datos son las clasificaciones que damos a los diferentes tipos de datos que utilizamos en la programación. En JavaScript, hay siete tipos de datos fundamentales:

Número : Cualquier número, incluidos los números con decimales: 4, 8, 1516, 23.42.

BigInt : cualquier número, mayor que 2 53 -1 o menor que -(2 53 -1), con n añadido al número: 1234567890123456n.

Cadena : cualquier agrupación de caracteres en su teclado (letras, números, espacios, símbolos, etc.) entre comillas simples: ' ... 'o comillas dobles " ... ", aunque preferimos las comillas simples. A algunas personas les gusta pensar en cuerda como una palabra elegante para referirse al texto.

Booleano : este tipo de datos solo tiene dos valores posibles: trueo false(sin comillas). Es útil pensar en los valores booleanos como interruptores de encendido y apagado o como respuestas a una pregunta de “sí” o “no”.

Nulo : este tipo de datos representa la ausencia intencional de un valor y está representado por la palabra clave null(sin comillas).

Indefinido : este tipo de datos se indica mediante la palabra clave undefined(sin comillas). También representa la ausencia de un valor aunque tiene un uso diferente al de null. undefinedsignifica que un valor dado no existe.

Símbolo : una característica más nueva del lenguaje, los símbolos son identificadores únicos, útiles en codificaciones más complejas. No hay necesidad de preocuparse por esto por ahora.

Objeto : Colecciones de datos relacionados.

Los primeros 6 de esos tipos se consideran tipos de datos primitivos . Son los tipos de datos más básicos del lenguaje. Los objetos son más complejos y aprenderá mucho más sobre ellos a medida que avance en JavaScript. Al principio, siete tipos pueden no parecer muchos, pero pronto observarás que el mundo se abre con posibilidades una vez que comiences a aprovechar cada uno de ellos. A medida que aprenda más sobre los objetos, podrá crear colecciones complejas de datos.

Pero antes de hacer eso, ¡vamos a familiarizarnos con las cadenas y los números!

console.log('Location of Codecademy headquarters: 575 Broadway, New York City');

console.log(40);

## Operadores aritméticos

La aritmética básica suele resultar útil a la hora de programar.

Un operador es un personaje que realiza una tarea en nuestro código. JavaScript tiene varios operadores aritméticos integrados , que nos permiten realizar cálculos matemáticos sobre números. Estos incluyen los siguientes operadores y sus símbolos correspondientes:

Agregar:+

Sustraer:-

Multiplicar:\*

Dividir:/

Resto:%

Los primeros cuatro funcionan como se puede adivinar:

console.log(3 + 4); // Prints 7

console.log(5 - 1); // Prints 4

console.log(4 \* 2); // Prints 8

console.log(9 / 3); // Prints 3

Tenga en cuenta que cuando console.log()la computadora evaluará la expresión dentro del paréntesis e imprimirá ese resultado en la consola. Si quisiéramos imprimir los caracteres 3 + 4, los envolveríamos entre comillas y los imprimiríamos como una cadena.

console.log(11 % 3); // Prints 2

console.log(12 % 3); // Prints 0

El operador restante, a veces llamado módulo , devuelve el número que queda después de que el número de la derecha se divide entre el número de la izquierda tantas veces como sea posible: es 11 % 3igual a 2 porque 3 cabe en 11 tres veces, dejando 2 como resto.

## Concatenación de cadenas

¡ Los operadores no son sólo para números! Cuando +se utiliza un operador en dos cadenas, agrega la cadena derecha a la cadena izquierda:

console.log('hi' + 'ya'); // Prints 'hiya'

console.log('wo' + 'ah'); // Prints 'woah'

console.log('I love to ' + 'code.')

// Prints 'I love to code.'

Este proceso de agregar una cadena a otra se llama concatenación . Observe que en el tercer ejemplo teníamos que asegurarnos de incluir un espacio al final de la primera cadena. La computadora unirá las cadenas exactamente, por lo que necesitábamos asegurarnos de incluir el espacio que queríamos entre las dos cadenas.

console.log('front ' + 'space');

// Prints 'front space'

console.log('back' + ' space');

// Prints 'back space'

console.log('no' + 'space');

// Prints 'nospace'

console.log('middle' + ' ' + 'space');

// Prints 'middle space'

Al igual que con las matemáticas normales, podemos combinar o encadenar nuestras operaciones para obtener un resultado final:

console.log('One' + ', ' + 'two' + ', ' + 'three!');

// Prints 'One, two, three!'

## Métodos

Recuerde que [los métodos](https://www.codecademy.com/resources/docs/javascript/methods?page_ref=catalog) son acciones que podemos realizar. Los tipos de datos tienen acceso a métodos específicos que nos permiten manejar instancias de ese tipo de datos. JavaScript proporciona varios métodos de cadena.

Llamamos o usamos estos métodos agregando una instancia con :

* un punto (el operador de punto)
* el nombre del método
* paréntesis de apertura y cierre

Por ejemplo 'example string'.methodName().

¿Esa sintaxis te parece un poco familiar? Cuando usamos console.log(), llamamos al .log()método del consoleobjeto. ¡ Veamos console.log()algunos métodos de cadenas reales en acción!

console.log('hello'.toUpperCase()); // Prints 'HELLO'  
console.log('Hey'.startsWith('H')); // Prints true

Veamos cada una de las líneas anteriores:

* En la primera línea, el [.toUpperCase()](https://www.codecademy.com/resources/docs/javascript/strings/toUpperCase)método se llama en la instancia de cadena 'hello'. El resultado se registra en la consola. Este método devuelve una cadena en letras mayúsculas: 'HELLO'.
* En la segunda línea, el [.startsWith()](https://www.codecademy.com/resources/docs/javascript/strings/startsWith)método se llama en la instancia de cadena 'Hey'. Este método también acepta el carácter 'H'como entrada o argumento entre paréntesis. Dado que la cadena 'Hey'comienza con la letra 'H', el método devuelve el valor booleano true.

Puede encontrar una lista de métodos de cadena integrados en la [documentación de JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/prototype) . Los desarrolladores utilizan la documentación como herramienta de referencia. Describe las palabras clave, los métodos y la sintaxis de JavaScript.

## Objetos incorporados

Además console, hay otros [objetos integrados en JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects) . Más adelante, creará sus propios [objetos](https://www.codecademy.com/resources/docs/javascript/objects) , pero por ahora estos objetos "integrados" están llenos de funciones útiles.

MathPor ejemplo, si desea realizar operaciones matemáticas más complejas que la aritmética, JavaScript tiene el objeto incorporado .

¡Lo mejor de los objetos es que tienen métodos! Llamemos al .random()método desde el objeto incorporado Math:

console.log(Math.random()); // Prints a random number between 0 and 1

En el ejemplo anterior, llamamos al .random()método agregando al nombre del objeto el operador de punto, el nombre del método y abriendo y cerrando paréntesis. Este método devuelve un número aleatorio entre 0 (inclusive) y 1 (exclusivo).

Para generar un número aleatorio entre 0 y 50, podríamos multiplicar este resultado por 50, así:

Math.random() \* 50;

El ejemplo anterior probablemente se evaluará como un decimal. Para asegurarnos de que la respuesta sea un número entero, podemos aprovechar otro Mathmétodo útil llamado Math.floor().

Math.floor()toma un número decimal y redondea hacia abajo al número entero más cercano. Puedes usar Math.floor()para redondear hacia abajo un número aleatorio como este:

Math.floor(Math.random() \* 50);

En este caso:

1. Math.random()genera un número aleatorio entre 0 y 1.
2. Luego multiplicamos ese número por 50, por lo que ahora tenemos un número entre 0 y 50.
3. Luego, Math.floor()redondea el número hacia abajo al número entero más cercano.

Si quisiera ver el número impreso en la terminal, aún necesitaría usar una console.log()declaración:

console.log(Math.floor(Math.random() \* 50)); // Prints a random whole number between 0 and 50

Para ver todas las propiedades y [métodos](https://www.codecademy.com/resources/docs/javascript/methods) del Mathobjeto, consulte [la documentación aquí](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) .

### Instrucciones

**1 .**

Dentro de a console.log(), crea un número aleatorio con Math.random()y luego multiplícalo por 100.

Punto de control 2 superado

¿Atascado? Obtén un consejo

**2 .**

Ahora, use Math.floor()para hacer que la salida sea un número entero.

Dentro del que console.log()escribiste en el último paso, coloca el Math.random() \* 100código existente dentro del paréntesis de Math.floor().

Punto de control 3 superado

¿Atascado? Obtén un consejo

**3 .**

Encuentre un método en el [objeto JavaScriptMath](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math) que devuelva el número entero más pequeño mayor o igual a un número decimal.

Utilice este método con el número 43.8. Registre la respuesta en la consola.

Punto de control 4 superado

Pista

Utilice el .ceil()método para calcular el número entero más pequeño mayor o igual a 43.8.

**4 .**

Utilice la [documentación de JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number) para encontrar un método en el objeto integrado Numberque verifique si un número es un número entero.

Coloque el número 2017entre paréntesis del método y utilícelo console.log()para imprimir el resultado.

Punto de control 5 superado

Pista

Utilice el .isInteger()método del Numberobjeto para comprobar si 2017es un número entero.

Vea un ejemplo de cómo usar .isInteger()y registrar el resultado:

console.log(Number.isInteger(34.2)); // Prints false

## Revisar

Echemos un vistazo más a los conceptos que acabamos de aprender:

* Los datos se imprimen o registran en la consola, un panel que muestra mensajes, con console.log().
* Podemos escribir [comentarios](https://www.codecademy.com/resources/docs/javascript/comments) de una sola línea y //comentarios de varias líneas entre /\*y \*/.
* Hay 7 [tipos de datos](https://www.codecademy.com/resources/docs/javascript/data-types) fundamentales en JavaScript: [cadenas](https://www.codecademy.com/resources/docs/javascript/strings) , números, booleanos, nulos, indefinidos, símbolos y objetos.
* Los números son cualquier número sin comillas:23.8879
* Las cadenas son caracteres entre comillas simples o dobles:'Sample String'
* [Los operadores](https://www.codecademy.com/resources/docs/javascript/operators) aritméticos integrados incluyen +, -, \*, /y %.
* [Los objetos](https://www.codecademy.com/resources/docs/javascript/objects) , incluidas instancias de tipos de datos, pueden tener propiedades e información almacenada. Las propiedades se indican con un .después del nombre del objeto, por ejemplo: 'Hello'.length.
* Los objetos, incluidas instancias de tipos de datos, pueden tener [métodos](https://www.codecademy.com/resources/docs/javascript/methods) que realicen acciones. Los métodos se llaman agregando al objeto o instancia un punto, el nombre del método y paréntesis. Por ejemplo: 'hello'.toUpperCase().
* Podemos acceder a propiedades y métodos utilizando el .operador de punto.
* Los objetos integrados, incluido Math, son colecciones de métodos y propiedades que proporciona JavaScript.

Aquí hay algunos recursos más para agregar a su kit de herramientas:

* [Documentos de Codecademy: JavaScript](https://www.codecademy.com/resources/docs/javascript)
* [Espacios de trabajo de Codecademy: JavaScript](https://www.codecademy.com/workspaces/new)

Practica en plataforma.

console.log("Hola Muchachos")

var tono = "Buenos Dias"

console.log(tono)

//Hola comentario

/\*

este es un

comentario

de varias

Lineas

\*/

//Tipos de dato

//Siete son los tipos de datos base de js

/\*

string

numbers

boolean

null

undefined

symbol

object

\*/

//Ejemplo de un metodo

console.log("hola".toUpperCase())

//Ejemplo de una propiedad

console.log("Hola".length)

///////////////////////////////////////////////////////////////77

let newVariable = 'Playing around with typeof.';

console.log(typeof newVariable);

newVariable = 1

console.log(typeof newVariable);

var a= "Marjan"

var b= 5

var c= true

console.log(typeof a)

console.log(typeof b)

console.log(typeof c)

var variableUno = 20

var variableDos = 30.5

var cadenaUno = "Hola"

var cadenaDos = "Como estas"

var banderaUno = true

var banderaDos = false

const constanteUno = 10

const constanteDos = "Que tal"

variableUno = variableDos + 20;

//constanteUno = 20;

console.log(variableUno)

console.log(variableDos)

console.log(cadenaUno)

console.log(cadenaDos)

console.log(banderaUno)

console.log(banderaDos)

console.log(constanteUno)

console.log(constanteDos)

//Concatenar es mezclar texto y datos

console.log(cadenaUno + " " + cadenaDos + " Este es un ejemplo de concatenar")

console.log("El dato de la variable variableUno es: " + variableUno + ", " + "el dato de la variable variableDos es " + variableDos + ", " + " el dato de la variable banderaUno es: " + banderaUno + ", el dato de la variable banderaDos es: " + banderaDos)

//Interpolar

console.log(`El valor de la variableUno es ${variableUno} y el de la variableDos es ${variableDos}`)

const x= 50;

var resultado

resultado= x\*5

console.log(`El resultado de la variable es:  ${resultado}` )

//Declarando constante kelvin con un valor de 293

const kelvin = 0

//Declarando variable celsius que vale kelvin - 273

var celsius = kelvin - 273

//Declaramos Fahrenheit usando el valor de celsius y una ecuación

var fahrenheit = celsius \* (9/5) + 32

//Convertimos a un numero redondo el calculo anterior esto con un objeto Math y un metodo floor que redondear hacia abajo

fahrenheit = Math.floor(fahrenheit)

console.log(`the temperature is ${fahrenheit} degress fahrenhei`)

// Convert to Newton

let newton = celsius \* (33 / 100);

// Round down

newton = Math.floor(newton);

console.log(`The temperature is ${newton} degrees Newton.`);

Jugar en <https://zty.pe/> el código de programación.



# Versiones de JavaScript: ES6 y anteriores

**¿Alguna vez has oído hablar del término “ES6” y te has preguntado de qué se trata? ¡Lea este artículo para leer y descubrirlo!**

Es posible que hayas visto el término “ES6” o “JavaScript ES6” y te hayas preguntado qué significa realmente. Bueno, no te preguntes más, porque vamos a profundizar en qué es ES6 y cómo se relaciona con JavaScript.

Primero, traigamos algo de historia. JavaScript fue introducido en 1995 por la empresa Netscape Communications como un lenguaje de programación para que los diseñadores y programadores web interactúen con las páginas web. El año siguiente, Netscape envió JavaScript a una organización de desarrollo de estándares llamada Ecma International para crear estándares para un lenguaje de secuencias de comandos (un tipo de lenguaje de programación). En 1997, Ecma International lanzó ECMA-262, que establece estándares para la primera versión de un lenguaje de programación llamado ECMAScript, abreviado como ES.

Estos nuevos estándares ECMAScript proporcionaron reglas para la arquitectura de las funciones de JavaScript. A medida que surgieron nuevos paradigmas de programación y los desarrolladores buscaron nuevas funciones, las versiones más nuevas de ECMAScript proporcionaron una base para la coherencia entre las versiones nuevas y antiguas de JavaScript.

Para distinguir completamente la diferencia entre JavaScript y ECMAScript: si desea crear una aplicación o programa, puede usar JavaScript; si desea crear un nuevo lenguaje de secuencias de comandos, puede seguir las pautas de ECMAScript. Entonces, cuando ve ES6 o JavaScript ES6, significa que esa versión de JavaScript sigue las especificaciones de la sexta edición de ECMAScript. Es posible que también vea ES2015 en lugar de ES6, pero ambas terminologías se refieren a la misma sexta edición de ECMAScript que se lanzó en 2015. Eche un vistazo a la línea de tiempo a continuación para ver cómo ha evolucionado JavaScript a lo largo de los años:

Ahora, quizás se pregunte, ¿qué hace que una actualización de 2015 siga siendo relevante hoy cuando hay actualizaciones más recientes como ES7 y ES8?

Bueno, a pesar del lanzamiento de versiones más nuevas, ES6 es en realidad la mayor actualización realizada a ECMAScript desde la primera edición lanzada en 1997. Algunos desarrolladores incluso se refieren a ES6 como "JavaScript moderno" debido a todas las adiciones importantes. Se agregaron muchas características excelentes para ayudar a los desarrolladores de JavaScript que incluyen:

* nuevas palabras clave como **let**y **const**para declarar variables
* nueva sintaxis de función usando funciones de flecha
* creación de clases
* parámetros con valores predeterminados
* promesas para acciones asincrónicas
* ¡y muchos más!

Los navegadores actualizados ahora admiten la mayoría de las funciones de ES6, lo que permite a los desarrolladores aprovechar estas nuevas incorporaciones. En última instancia, ES6 permite a los programadores ahorrar tiempo y escribir código más conciso. Tomemos, por ejemplo, la sintaxis anterior a ES6 para expresiones de funciones:

var greeting = function() {  
  console.log('Hello World!');    
};

Con las funciones de flecha de ES6, podemos transformar la expresión anterior en:

const greeting = () => console.log('Hello World');

Sin embargo, las funciones de flecha no son simplemente reescrituras sintácticas. Al igual que con otras características de ES6, existen otros beneficios y compensaciones subyacentes a considerar. No obstante, ha habido una fuerte adopción de ES6 en la comunidad de desarrollo. Beneficios como la nueva sintaxis de ES6 facilitan el uso de un paradigma de programación popular, la programación orientada a objetos (OOP). Con este cambio, los desarrolladores de otros lenguajes que están acostumbrados a la programación orientada a objetos tienen una transición más fluida hacia el aprendizaje y el uso de JavaScript. Otra razón de la popularidad de ES6 está relacionada con el uso de ES6 en marcos populares como React. Por lo tanto, si desea aprender las herramientas y marcos más nuevos, tendrá que adquirir ES6 en el camino.

Dicho esto, no debemos ignorar el código heredado, es decir, las versiones anteriores de JavaScript. De hecho, todavía hay muchos proyectos que se crean y mantienen con código heredado. Si desea tener la capacidad y la libertad de trabajar en cualquier tipo de proyecto de JavaScript, debe familiarizarse con la sintaxis de JavaScript anterior a ES6 y ES6. Pero no se preocupe, cubrimos tanto pre-ES6 como ES6 en nuestro curso de JavaScript. ¡Compruébalo para convertirte en una estrella de rock en los conceptos básicos de JavaScript y aprender habilidades fundamentales de programación!

# Semana 5 – Unidad 2 Recursividad - Definición

18, 19 y 21

La recursividad es un concepto fundamental en programación y matemáticas que se refiere a la capacidad de una función o proceso para llamarse a sí mismo. En otras palabras, es la técnica mediante la cual una función se define en términos de sí misma. Esto puede ser útil en muchas situaciones, especialmente cuando se trata de resolver problemas que pueden dividirse en casos más pequeños y similares.

var a = 100;

var b = 200;

var c = 80;

//Si a = b

if(a === b){

console.log("Son iguales")

}

if(a === b){

console.log("Son iguales")

}else{

console.log("Son diferentes")

}

if(a === b){

console.log("Son iguales")

}else if(a > b){

console.log("a es mayor")

}

if(a === b){

console.log("Son iguales")

}else if(a > b){

console.log("a es mayor")

}else{

console.log("b es mayor")

}

//Caalcular cual es el mayor a y b

if(a > b){

console.log("A es Mayor")

}else{

console.log("B es Mayor")

}

//Caalcular cual es el mayor a y b o si son iguales

if(a === b){

console.log("A y B son iguales")

}else if(a > b){

console.log("A es Mayor")

}else{

console.log("B es Mayor")

}

if(a>b){

if(a>c){

console.log("a es mayor")

}else{

console.log("c es mayor")

}

}else{

if(b>c){

console.log("b es mayor")

}else{

console.log("c es mayor")

}

}

En programación, la recursividad se utiliza principalmente para resolver problemas que se pueden dividir en subproblemas más pequeños y similares. La función recursiva continúa llamándose a sí misma con estos subproblemas hasta que se alcanza un caso base que se puede resolver de manera directa. A continuación, se muestra un ejemplo en pseudocódigo de una función recursiva para calcular el factorial de un número:

```pseudocodigo

Función factorial(n):

Si n es igual a 0:

Devolver 1

De lo contrario:

Devolver n \* factorial(n - 1)

```

En este ejemplo, la función `factorial` se llama a sí misma con un valor más pequeño en cada iteración hasta que `n` alcanza el caso base (0), momento en el que la función deja de llamarse a sí misma y comienza a calcular el resultado final.

var userName = 'Jacinto';

userName === '' ? console.log('Hello!') : console.log(`Hello, ${userName}`)

var userQuestion = '¿Mis alumnos aprenderan a programar?';

//Concatenado

console.log(userName + " pregunta: " + userQuestion);

var randomNumber = Math.floor(Math.random() \* 8)

var eightBall = '';

switch(randomNumber){

  case 0:

    eightBall = 'It is certain';

  break;

  case 1:

    eightBall = 'It is decidedly so';

  break;

  case 2:

    eightBall = 'Reply hazy try again';

  break;

  case 3:

    eightBall = 'Cannot predict now';

  break;

  case 4:

   eightBall = 'Do not count on it';

  break;

  case 5:

    eightBall = 'My sources say no';

  break;

  case 6:

   eightBall = 'Outlook not so good';

  break;

  case 7:

    eightBall = 'Signs point to yes';

  break;

}

if(randomNumber === 0){

  eightBall = 'It is certain';

}else if(randomNumber === 1){

  eightBall = 'It is decidedly so';

}else if(randomNumber === 2){

  eightBall = 'Reply hazy try again';

  }else if(randomNumber === 3){

    eightBall = 'Cannot predict now';

     }else if(randomNumber === 4){

eightBall = 'Do not count on it';

   }else if(randomNumber === 5){

   eightBall = 'My sources say no';

   }else if(randomNumber === 6){

   eightBall = 'Outlook not so good';

   }else if(randomNumber === 7){

   eightBall = 'Signs point to yes';

}

console.log(eightBall);

Es importante tener en cuenta que las funciones recursivas deben definir un caso base para evitar que se ejecuten indefinidamente y causen un desbordamiento de pila. También es fundamental asegurarse de que cada llamada recursiva se acerque al caso base para que el proceso de recursión finalice en algún momento.

La recursividad se utiliza en una variedad de algoritmos y problemas, como el cálculo de secuencias Fibonacci, la exploración de árboles y grafos, la resolución de problemas de búsqueda y muchas otras aplicaciones en programación y matemáticas.

function makeShoppingList(item1 = 'milk', item2='bread', item3='eggs'){

  console.log(`Remember to buy ${item1}`);

  console.log(`Remember to buy ${item2}`);

  console.log(`Remember to buy ${item3}`);

}

makeShoppingList('Galletas', 'Cerveza', 'Cacahuates');

function myFuncion(num1 = 'x001', num2 = 'x002'){

  if(num1 === 'x001' || num2 === 'x002'){

    console.log("Esta función trabaja con 2 argumentos, intenta de nuevo");

  }else{

    num1 += num2;

    console.log('La suman es: ' + num1);

  }

}

myFuncion(10,90);

# Semana 6 – Unidad 2 Recursividad - Procedimientos recursivos.

25, 26 y 28 septiembre

Los procedimientos recursivos son funciones o subrutinas en programación que se llaman a sí mismas para resolver un problema de manera iterativa. Estas funciones se utilizan para dividir un problema en problemas más pequeños y similares, resolviendo cada subproblema de manera recursiva hasta que se alcance un caso base y se pueda calcular la solución final. Aquí hay una descripción más detallada de cómo funcionan los procedimientos recursivos:

1. Caso base: Cada función recursiva debe tener un caso base que define una condición de parada. Cuando se cumple esta condición, la función recursiva deja de llamarse a sí misma y comienza a devolver resultados. El caso base evita que la recursión se vuelva infinita y garantiza que el procedimiento recursivo finalice en algún momento.

2. Llamadas recursivas: Dentro de la función recursiva, se realizan llamadas a sí misma para resolver subproblemas más pequeños y similares. Estas llamadas deben acercarse al caso base con cada iteración, de lo contrario, la recursión no terminará.

3. Combinación de resultados: A medida que las llamadas recursivas se ejecutan y se resuelven, los resultados se combinan de alguna manera para calcular la solución final del problema original. Esto suele implicar operaciones matemáticas o combinación de datos.

4. Flujo de control: El flujo de control en una función recursiva puede ser un poco diferente al de una función iterativa típica. Cada llamada recursiva crea una nueva instancia de la función en la pila de llamadas y se ejecuta de manera independiente. A medida que las llamadas recursivas se resuelven, las instancias anteriores de la función se desapilan.

Aquí hay un ejemplo en pseudocódigo de una función recursiva para calcular la suma de los primeros `n` números naturales:

```pseudocodigo

Función sumaNaturales(n):

Si n es igual a 0 (caso base):

Devolver 0

De lo contrario:

Devolver n + sumaNaturales(n - 1) (llamada recursiva)

```

En este ejemplo, el caso base es cuando `n` es igual a 0, y en ese caso, la función devuelve 0. En las llamadas recursivas, se suma `n` al resultado de la llamada recursiva con `n-1`. La recursión continuará hasta que `n` alcance el caso base.

Es importante tener en cuenta que en la programación recursiva, es esencial garantizar que las llamadas recursivas eventualmente lleguen al caso base para evitar que el programa se quede atascado en una recursión infinita.

### console.log()

The console.log() method is used to log or print messages to the console. It can also be used to print objects and other info.

console.log('Hi there!');

// Prints: Hi there!

### JavaScript

JavaScript is a programming language that powers the dynamic behavior on most websites. Alongside HTML and CSS, it is a core technology that makes the web run.

### Methods

Methods return information about an object, and are called by appending an instance with a period ., the method name, and parentheses.

// Returns a number between 0 and 1

Math.random();

### Built-in Objects

Built-in objects contain methods that can be called by appending the object name with a period ., the method name, and a set of parentheses.

Math.random();

// ☝️ Math is the built-in object

### Numbers

Numbers are a primitive data type. They include the set of all integers and floating point numbers.

let amount = 6;

let price = 4.99;

### String .length

The .length property of a string returns the number of characters that make up the string.

let message = 'good nite';

console.log(message.length);

// Prints: 9

console.log('howdy'.length);

// Prints: 5

### Data Instances

When a new piece of data is introduced into a JavaScript program, the program keeps track of it in an instance of that data type. An instance is an individual case of a data type.

### Booleans

Booleans are a primitive data type. They can be either true or false.

let lateToWork = true;

### Math.random()

The Math.random() method returns a floating-point, random number in the range from 0 (inclusive) up to but not including 1.

console.log(Math.random());

// Prints: 0 - 0.9999999999999999

### Math.floor()

The Math.floor() function returns the largest integer less than or equal to the given number.

console.log(Math.floor(5.95));

// Prints: 5

### Single Line Comments

In JavaScript, single-line comments are created with two consecutive forward slashes //.

// This line will denote a comment

### Null

Null is a primitive data type. It represents the intentional absence of value. In code, it is represented as null.

let x = null;

### Strings

Strings are a primitive data type. They are any grouping of characters (letters, spaces, numbers, or symbols) surrounded by single quotes ' or double quotes ".

let single = 'Wheres my bandit hat?';

let double = "Wheres my bandit hat?";

### Arithmetic Operators

JavaScript supports arithmetic operators for:

* + addition
* - subtraction
* \* multiplication
* / division
* % modulo

// Addition

5 + 5

// Subtraction

10 - 5

// Multiplication

5 \* 10

// Division

10 / 5

// Modulo

10 % 5

### Multi-line Comments

In JavaScript, multi-line comments are created by surrounding the lines with /\* at the beginning and \*/ at the end. Comments are good ways for a variety of reasons like explaining a code block or indicating some hints, etc.

/\*

The below configuration must be

changed before deployment.

\*/

let baseUrl = 'localhost/taxwebapp/country';

### Remainder / Modulo Operator

The remainder operator, sometimes called modulo, returns the number that remains after the right-hand number divides into the left-hand number as many times as it evenly can.

// calculates # of weeks in a year, rounds down to nearest integer

const weeksInYear = Math.floor(365/7);

// calcuates the number of days left over after 365 is divded by 7

const daysLeftOver = 365 % 7 ;

console.log("A year has " + weeksInYear + " weeks and " + daysLeftOver + " days");

### Assignment Operators

An assignment operator assigns a value to its left operand based on the value of its right operand. Here are some of them:

* += addition assignment
* -= subtraction assignment
* \*= multiplication assignment
* /= division assignment

let number = 100;

// Both statements will add 10

number = number + 10;

number += 10;

console.log(number);

// Prints: 120

### String Interpolation

String interpolation is the process of evaluating string literals containing one or more placeholders (expressions, variables, etc).

It can be performed using template literals: text ${expression} text.

let age = 7;

// String concatenation

'Tommy is ' + age + ' years old.';

// String interpolation

`Tommy is ${age} years old.`;

### Variables

Variables are used whenever there’s a need to store a piece of data. A variable contains data that can be used in the program elsewhere. Using variables also ensures code re-usability since it can be used to replace the same value in multiple places.

const currency = '$';

let userIncome = 85000;

console.log(currency + userIncome + ' is more than the average income.');

// Prints: $85000 is more than the average income.

### Undefined

undefined is a primitive JavaScript value that represents lack of defined value. Variables that are declared but not initialized to a value will have the value undefined.

var a;

console.log(a);

// Prints: undefined

### Learn Javascript: Variables

A variable is a container for data that is stored in computer memory. It is referenced by a descriptive name that a programmer can call to assign a specific value and retrieve it.

// Examples of variables

let name = "Tammy";

const found = false;

var age = 3;

console.log(name, found, age);

// Prints: Tammy false 3

### Declaring Variables

To declare a variable in JavaScript, any of these three keywords can be used along with a variable name:

* var is used in pre-ES6 versions of JavaScript.
* let is the preferred way to declare a variable when it can be reassigned.
* const is the preferred way to declare a variable with a constant value.

var age;

let weight;

const numberOfFingers = 20;

### Template Literals

Template literals are strings that allow embedded expressions, ${expression}. While regular strings use single ' or double " quotes, template literals use backticks instead.

let name = "Codecademy";

console.log(`Hello, ${name}`);

// Prints: Hello, Codecademy

console.log(`Billy is ${6+8} years old.`);

// Prints: Billy is 14 years old.

### let Keyword

let creates a local variable in JavaScript & can be re-assigned. Initialization during the declaration of a let variable is optional. A let variable will contain undefined if nothing is assigned to it.

let count;

console.log(count); // Prints: undefined

count = 10;

console.log(count); // Prints: 10

### const Keyword

A constant variable can be declared using the keyword const. It must have an assignment. Any attempt of re-assigning a const variable will result in JavaScript runtime error.

const numberOfColumns = 4;

numberOfColumns = 8;

// TypeError: Assignment to constant variable.

### String Concatenation

In JavaScript, multiple strings can be concatenated together using the + operator. In the example, multiple strings and variables containing string values have been concatenated. After execution of the code block, the displayText variable will contain the concatenated string.

let service = 'credit card';

let month = 'May 30th';

let displayText = 'Your ' + service + ' bill is due on ' + month + '.';

console.log(displayText);

// Prints: Your credit card bill is due on May 30th.

### Control Flow

Control flow is the order in which statements are executed in a program. The default control flow is for statements to be read and executed in order from left-to-right, top-to-bottom in a program file.

Control structures such as conditionals (if statements and the like) alter control flow by only executing blocks of code if certain conditions are met. These structures essentially allow a program to make decisions about which code is executed as the program runs.

### Logical Operator ||

The logical OR operator || checks two values and returns a boolean. If one or both values are truthy, it returns true. If both values are falsy, it returns false.

| **A** | **B** | **A || B** |
| --- | --- | --- |
| false | false | false |
| false | true | true |
| true | false | true |
| true | true | true |

true || false; // true

10 > 5 || 10 > 20; // true

false || false; // false

10 > 100 || 10 > 20; // false

### Ternary Operator

The ternary operator allows for a compact syntax in the case of binary (choosing between two choices) decisions. It accepts a condition followed by a ? operator, and then two expressions separated by a :. If the condition evaluates to truthy, the first expression is executed, otherwise, the second expression is executed.

let price = 10.5;

let day = "Monday";

day === "Monday" ? price -= 1.5 : price += 1.5;

### else Statement

An else block can be added to an if block or series of if-else if blocks. The else block will be executed only if the if condition fails.

const isTaskCompleted = false;

if (isTaskCompleted) {

console.log('Task completed');

} else {

console.log('Task incomplete');

}

### Logical Operator &&

The logical AND operator && checks two values and returns a boolean. If both values are truthy, then it returns true. If one, or both, of the values is falsy, then it returns false.

true && true; // true

1 > 2 && 2 > 1; // false

true && false; // false

4 === 4 && 3 > 1; // true

### switch Statement

The switch statements provide a means of checking an expression against multiple case clauses. If a case matches, the code inside that clause is executed.

The case clause should finish with a break keyword. If no case matches but a default clause is included, the code inside default will be executed.

**Note:** If break is omitted from the block of a case, the switch statement will continue to check against case values until a break is encountered or the flow is broken.

const food = 'salad';

switch (food) {

case 'oyster':

console.log('The taste of the sea 🦪');

break;

case 'pizza':

console.log('A delicious pie 🍕');

break;

default:

console.log('Enjoy your meal');

}

// Prints: Enjoy your meal

### if Statement

An if statement accepts an expression with a set of parentheses:

* If the expression evaluates to a truthy value, then the code within its code body executes.
* If the expression evaluates to a falsy value, its code body will not execute.

const isMailSent = true;

if (isMailSent) {

console.log('Mail sent to recipient');

}

### Logical Operator !

The logical NOT operator ! can be used to do one of the following:

* Invert a Boolean value.
* Invert the truthiness of non-Boolean values.

let lateToWork = true;

let oppositeValue = !lateToWork;

console.log(oppositeValue);

// Prints: false

### Comparison Operators

Comparison operators are used to comparing two values and return true or false depending on the validity of the comparison:

* === strict equal
* !== strict not equal
* > greater than
* >= greater than or equal
* < less than
* <= less than or equal

1 > 3 // false

3 > 1 // true

250 >= 250 // true

1 === 1 // true

1 === 2 // false

1 === '1' // false

### else if Clause

After an initial if block, else if blocks can each check an additional condition. An optional else block can be added after the else if block(s) to run by default if none of the conditionals evaluated to truthy.

const size = 10;

if (size > 100) {

console.log('Big');

} else if (size > 20) {

console.log('Medium');

} else if (size > 4) {

console.log('Small');

} else {

console.log('Tiny');

}

// Print: Small

### Truthy and Falsy

In JavaScript, values evaluate to true or false when evaluated as Booleans.

* Values that evaluate to true are known as truthy
* Values that evaluate to false are known as falsy

Falsy values include false, 0, empty strings, null undefined, and NaN. All other values are truthy.

### Arrow Functions (ES6)

Arrow function expressions were introduced in ES6. These expressions are clean and concise. The syntax for an arrow function expression does not require the function keyword and uses a fat arrow => to separate the parameter(s) from the body.

There are several variations of arrow functions:

* Arrow functions with a single parameter do not require () around the parameter list.
* Arrow functions with a single expression can use the concise function body which returns the result of the expression without the return keyword.

// Arrow function with two parameters

const sum = (firstParam, secondParam) => {

return firstParam + secondParam;

};

console.log(sum(2,5)); // Prints: 7

// Arrow function with no parameters

const printHello = () => {

console.log('hello');

};

printHello(); // Prints: hello

// Arrow functions with a single parameter

const checkWeight = weight => {

console.log(`Baggage weight : ${weight} kilograms.`);

};

checkWeight(25); // Prints: Baggage weight : 25 kilograms.

// Concise arrow functions

const multiply = (a, b) => a \* b;

console.log(multiply(2, 30)); // Prints: 60

### Functions

Functions are one of the fundamental building blocks in JavaScript. A function is a reusable set of statements to perform a task or calculate a value. Functions can be passed one or more values and can return a value at the end of their execution. In order to use a function, you must define it somewhere in the scope where you wish to call it.

The example code provided contains a function that takes in 2 values and returns the sum of those numbers.

// Defining the function:

function sum(num1, num2) {

return num1 + num2;

}

// Calling the function:

sum(3, 6); // 9

### Anonymous Functions

Anonymous functions in JavaScript do not have a name property. They can be defined using the function keyword, or as an arrow function. See the code example for the difference between a named function and an anonymous function.

// Named function

function rocketToMars() {

return 'BOOM!';

}

// Anonymous function

const rocketToMars = function() {

return 'BOOM!';

}

### Function Expressions

Function expressions create functions inside an expression instead of as a function declaration. They can be anonymous and/or assigned to a variable.

const dog = function() {

return 'Woof!';

}

### Function Parameters

Inputs to functions are known as parameters when a function is declared or defined. Parameters are used as variables inside the function body. When the function is called, these parameters will have the value of whatever is passed in as arguments. It is possible to define a function without parameters.

// The parameter is name

function sayHello(name) {

return `Hello, ${name}!`;

}

### return Keyword

Functions return (pass back) values using the return keyword. return ends function execution and returns the specified value to the location where it was called. A common mistake is to forget the return keyword, in which case the function will return undefined by default.

// With return

function sum(num1, num2) {

return num1 + num2;

}

// Without return, so the function doesn't output the sum

function sum(num1, num2) {

num1 + num2;

}

### Function Declaration

Function declarations are used to create named functions. These functions can be called using their declared name. Function declarations are built from:

* The function keyword.
* The function name.
* An optional list of parameters separated by commas enclosed by a set of parentheses ().
* A function body enclosed in a set of curly braces {}.

function add(num1, num2) {

return num1 + num2;

}

### Calling Functions

Functions can be called, or executed, elsewhere in code using parentheses following the function name. When a function is called, the code inside its function body runs. Arguments are values passed into a function when it is called.

// Defining the function

function sum(num1, num2) {

return num1 + num2;

}

// Calling the function

sum(2, 4); // 6

### Scope

Scope is a concept that refers to where values and functions can be accessed.

Various scopes include:

* Global scope (a value/function in the global scope can be used anywhere in the entire program)
* File or module scope (the value/function can only be accessed from within the file)
* Function scope (only visible within the function),
* Code block scope (only visible within a { ... } codeblock)

function myFunction() {

var pizzaName = "Volvo";

// Code here can use pizzaName

}

// Code here can't use pizzaName

### Block Scoped Variables

const and let are block scoped variables, meaning they are only accessible in their block or nested blocks. In the given code block, trying to print the statusMessage using the console.log() method will result in a ReferenceError. It is accessible only inside that if block.

const isLoggedIn = true;

if (isLoggedIn == true) {

const statusMessage = 'User is logged in.';

}

console.log(statusMessage);

// Uncaught ReferenceError: statusMessage is not defined

### Global Variables

JavaScript variables that are declared outside of blocks or functions can exist in the global scope, which means they are accessible throughout a program. Variables declared outside of smaller block or function scopes are accessible inside those smaller scopes.

**Note:** It is best practice to keep global variables to a minimum.

// Variable declared globally

const color = 'blue';

function printColor() {

console.log(color);

}

printColor(); // Prints: blue

### Property .length

The .length property of a JavaScript array indicates the number of elements the array contains.

const numbers = [1, 2, 3, 4];

numbers.length // 4

### Index

Array elements are arranged by index values, starting at 0 as the first element index. Elements can be accessed by their index using the array name, and the index surrounded by square brackets.

// Accessing an array element

const myArray = [100, 200, 300];

console.log(myArray[0]); // 100

console.log(myArray[1]); // 200

console.log(myArray[2]); // 300

### Method .push()

The .push() method of JavaScript arrays can be used to add one or more elements to the end of an array. .push() mutates the original array and returns the new length of the array.

// Adding a single element:

const cart = ['apple', 'orange'];

cart.push('pear');

// Adding multiple elements:

const numbers = [1, 2];

numbers.push(3, 4, 5);

### Method .pop()

The .pop() method removes the last element from an array and returns that element.

const ingredients = ['eggs', 'flour', 'chocolate'];

const poppedIngredient = ingredients.pop(); // 'chocolate'

console.log(ingredients); // ['eggs', 'flour']

### Mutable

JavaScript arrays are mutable, meaning that the values they contain can be changed.

Even if they are declared using const, the contents can be manipulated by reassigning internal values or using methods like .push() and .pop().

const names = ['Alice', 'Bob'];

names.push('Carl');

// ['Alice', 'Bob', 'Carl']

### Arrays

Arrays are lists of ordered, stored data. They can hold items that are of any data type. Arrays are created by using square brackets, with individual elements separated by commas.

// An array containing numbers

const numberArray = [0, 1, 2, 3];

// An array containing different data types

const mixedArray = [1, 'chicken', false];

### Reverse Loop

A for loop can iterate “in reverse” by initializing the loop variable to the starting value, testing for when the variable hits the ending value, and decrementing (subtracting from) the loop variable at each iteration.

const items = ['apricot', 'banana', 'cherry'];

for (let i = items.length - 1; i >= 0; i -= 1) {

console.log(`${i}. ${items[i]}`);

}

// Prints: 2. cherry

// Prints: 1. banana

// Prints: 0. apricot

### Do…While Statement

A do...while statement creates a loop that executes a block of code once, checks if a condition is true, and then repeats the loop as long as the condition is true. They are used when you want the code to always execute at least once. The loop ends when the condition evaluates to false.

x = 0

i = 0

do {

x = x + i;

console.log(x)

i++;

} while (i < 5);

// Prints: 0 1 3 6 10

### For Loop

A for loop declares looping instructions, with three important pieces of information separated by semicolons ;:

* The initialization defines where to begin the loop by declaring (or referencing) the iterator variable
* The stopping condition determines when to stop looping (when the expression evaluates to false)
* The iteration statement updates the iterator each time the loop is completed

for (let i = 0; i < 4; i += 1) {

console.log(i);

};

// Output: 0, 1, 2, 3

### Looping Through Arrays

An array’s length can be evaluated with the .length property. This is extremely helpful for looping through arrays, as the .length of the array can be used as the stopping condition in the loop.

for (let i = 0; i < array.length; i++){

console.log(array[i]);

}

// Output: Every item in the array

### Break Keyword

Within a loop, the break keyword may be used to exit the loop immediately, continuing execution after the loop body.

Here, the break keyword is used to exit the loop when i is greater than 5.

for (let i = 0; i < 99; i += 1) {

if (i > 5) {

break;

}

console.log(i)

}

// Output: 0 1 2 3 4 5

### Nested For Loop

A nested for loop is when a for loop runs inside another for loop.

The inner loop will run all its iterations for each iteration of the outer loop.

for (let outer = 0; outer < 2; outer += 1) {

for (let inner = 0; inner < 3; inner += 1) {

console.log(`${outer}-${inner}`);

}

}

/\*

Output:

0-0

0-1

0-2

1-0

1-1

1-2

\*/

### Loops

A loop is a programming tool that is used to repeat a set of instructions. Iterate is a generic term that means “to repeat” in the context of loops. A loop will continue to iterate until a specified condition, commonly known as a stopping condition, is met.

### While Loop

The while loop creates a loop that is executed as long as a specified condition evaluates to true. The loop will continue to run until the condition evaluates to false. The condition is specified before the loop, and usually, some variable is incremented or altered in the while loop body to determine when the loop should stop.

while (condition) {

// code block to be executed

}

let i = 0;

while (i < 5) {

console.log(i);

i++;

}

### Functions Assigned to Variables

In JavaScript, functions are a data type just as strings, numbers, and arrays are data types. Therefore, functions can be assigned as values to variables, but are different from all other data types because they can be invoked.

let plusFive = (number) => {

return number + 5;

};

// f is assigned the value of plusFive

let f = plusFive;

plusFive(3); // 8

// Since f has a function value, it can be invoked.

f(9); // 14

### Callback Functions

In JavaScript, a callback function is a function that is passed into another function as an argument. This function can then be invoked during the execution of that higher order function (that it is an argument of).

Since, in JavaScript, functions are objects, functions can be passed as arguments.

const isEven = (n) => {

return n % 2 == 0;

}

let printMsg = (evenFunc, num) => {

const isNumEven = evenFunc(num);

console.log(`The number ${num} is an even number: ${isNumEven}.`)

}

// Pass in isEven as the callback function

printMsg(isEven, 4);

// Prints: The number 4 is an even number: True.

### Higher-Order Functions

In Javascript, functions can be assigned to variables in the same way that strings or arrays can. They can be passed into other functions as parameters or returned from them as well.

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

### JavaScript Functions: First-Class Objects

JavaScript functions are first-class objects. Therefore:

* They have built-in properties and methods, such as the name property and the .toString() method.
* Properties and methods can be added to them.
* They can be passed as arguments and returned from other functions.
* They can be assigned to variables, array elements, and other objects.

//Assign a function to a variable originalFunc

const originalFunc = (num) => { return num + 2 };

//Re-assign the function to a new variable newFunc

const newFunc = originalFunc;

//Access the function's name property

newFunc.name; //'originalFunc'

//Return the function's body as a string

newFunc.toString(); //'(num) => { return num + 2 }'

//Add our own isMathFunction property to the function

newFunc.isMathFunction = true;

//Pass the function as an argument

const functionNameLength = (func) => { return func.name.length };

functionNameLength(originalFunc); //12

//Return the function

const returnFunc = () => { return newFunc };

returnFunc(); //[Function: originalFunc]

### The .reduce() Method

The .reduce() method iterates through an array and returns a single value.

In the above code example, the .reduce() method will sum up all the elements of the array. It takes a callback function with two parameters (accumulator, currentValue) as arguments. On each iteration, accumulator is the value returned by the last iteration, and the currentValue is the current element. Optionally, a second argument can be passed which acts as the initial value of the accumulator.

const arrayOfNumbers = [1, 2, 3, 4];

const sum = arrayOfNumbers.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

});

console.log(sum); // 10

### The .forEach() Method

The .forEach() method executes a callback function on each of the elements in an array in order.

In the above example code, the callback function containing a console.log() method will be executed 5 times, once for each element.

const numbers = [28, 77, 45, 99, 27];

numbers.forEach(number => {

console.log(number);

});

### The .filter() Method

The .filter() method executes a callback function on each element in an array. The callback function for each of the elements must return either true or false. The returned array is a new array with any elements for which the callback function returns true.

In the above code example, the array filteredArray will contain all the elements of randomNumbers but 4.

const randomNumbers = [4, 11, 42, 14, 39];

const filteredArray = randomNumbers.filter(n => {

return n > 5;

});

### The .map() Method

The .map() method executes a callback function on each element in an array. It returns a new array made up of the return values from the callback function.

The original array does not get altered, and the returned array may contain different elements than the original array.

In the example code above, the .map() method is used to add ' joined the contest.' string at the end of each element in the finalParticipants array.

const finalParticipants = ['Taylor', 'Donald', 'Don', 'Natasha', 'Bobby'];

// add string after each final participant

const announcements = finalParticipants.map(member => {

return member + ' joined the contest.';

})

console.log(announcements);

### Restrictions in Naming Properties

JavaScript object key names must adhere to some restrictions to be valid. Key names must either be strings or valid identifier or variable names (i.e. special characters such as - are not allowed in key names that are not strings).

// Example of invalid key names

const trainSchedule = {

platform num: 10, // Invalid because of the space between words.

40 - 10 + 2: 30, // Expressions cannot be keys.

+compartment: 'C' // The use of a + sign is invalid unless it is enclosed in quotations.

}

### Dot Notation for Accessing Object Properties

Properties of a JavaScript object can be accessed using the dot notation in this manner: object.propertyName. Nested properties of an object can be accessed by chaining key names in the correct order.

const apple = {

color: 'Green',

price: {

bulk: '$3/kg',

smallQty: '$4/kg'

}

};

console.log(apple.color); // 'Green'

console.log(apple.price.bulk); // '$3/kg'

### Objects

An object is a built-in data type for storing key-value pairs. Data inside objects are unordered, and the values can be of any type.

### Accessing non-existent JavaScript properties

When trying to access a JavaScript object property that has not been defined yet, the value of undefined will be returned by default.

const classElection = {

date: 'January 12'

};

console.log(classElection.place); // undefined

### JavaScript Objects are Mutable

JavaScript objects are mutable, meaning their contents can be changed, even when they are declared as const. New properties can be added, and existing property values can be changed or deleted.

It is the reference to the object, bound to the variable, that cannot be changed.

const student = {

name: 'Sheldon',

score: 100,

grade: 'A',

}

console.log(student)

// { name: 'Sheldon', score: 100, grade: 'A' }

delete student.score

student.grade = 'F'

console.log(student)

// { name: 'Sheldon', grade: 'F' }

student = {}

// TypeError: Assignment to constant variable.

### JavaScript for...in loop

The JavaScript for...in loop can be used to iterate over the keys of an object. In each iteration, one of the properties from the object is assigned to the variable of that loop.

let mobile = {

brand: 'Samsung',

model: 'Galaxy Note 9'

};

for (let key in mobile) {

console.log(`${key}: ${mobile[key]}`);

}

### Properties and values of a JavaScript object

A JavaScript object literal is enclosed with curly braces {}. Values are mapped to keys in the object with a colon (:), and the key-value pairs are separated by commas. All the keys are unique, but values are not.

Key-value pairs of an object are also referred to as properties.

const classOf2018 = {

students: 38,

year: 2018

}

### Delete operator

Once an object is created in JavaScript, it is possible to remove properties from the object using the delete operator. The delete keyword deletes both the value of the property and the property itself from the object. The delete operator only works on properties, not on variables or functions.

const person = {

firstName: "Matilda",

age: 27,

hobby: "knitting",

goal: "learning JavaScript"

};

delete person.hobby; // or delete person[hobby];

console.log(person);

/\*

{

firstName: "Matilda"

age: 27

goal: "learning JavaScript"

}

\*/

### javascript passing objects as arguments

When JavaScript objects are passed as arguments to functions or methods, they are passed by reference, not by value. This means that the object itself (not a copy) is accessible and mutable (can be changed) inside that function.

const origNum = 8;

const origObj = {color: 'blue'};

const changeItUp = (num, obj) => {

num = 7;

obj.color = 'red';

};

changeItUp(origNum, origObj);

// Will output 8 since integers are passed by value.

console.log(origNum);

// Will output 'red' since objects are passed

// by reference and are therefore mutable.

console.log(origObj.color);

### JavaScript Object Methods

JavaScript objects may have property values that are functions. These are referred to as object methods.

Methods may be defined using anonymous arrow function expressions, or with shorthand method syntax.

Object methods are invoked with the syntax: objectName.methodName(arguments).

const engine = {

// method shorthand, with one argument

start(adverb) {

console.log(`The engine starts up ${adverb}...`);

},

// anonymous arrow function expression with no arguments

sputter: () => {

console.log('The engine sputters...');

},

};

engine.start('noisily');

engine.sputter();

/\* Console output:

The engine starts up noisily...

The engine sputters...

\*/

### JavaScript destructuring assignment shorthand syntax

The JavaScript destructuring assignment is a shorthand syntax that allows object properties to be extracted into specific variable values.

It uses a pair of curly braces ({}) with property names on the left-hand side of an assignment to extract values from objects. The number of variables can be less than the total properties of an object.

const rubiksCubeFacts = {

possiblePermutations: '43,252,003,274,489,856,000',

invented: '1974',

largestCube: '17x17x17'

};

const {possiblePermutations, invented, largestCube} = rubiksCubeFacts;

console.log(possiblePermutations); // '43,252,003,274,489,856,000'

console.log(invented); // '1974'

console.log(largestCube); // '17x17x17'

### shorthand property name syntax for object creation

The shorthand property name syntax in JavaScript allows creating objects without explicitly specifying the property names (ie. explicitly declaring the value after the key). In this process, an object is created where the property names of that object match variables which already exist in that context. Shorthand property names populate an object with a key matching the identifier and a value matching the identifier’s value.

const activity = 'Surfing';

const beach = { activity };

console.log(beach); // { activity: 'Surfing' }

### this Keyword

The reserved keyword this refers to a method’s calling object, and it can be used to access properties belonging to that object.

Here, using the this keyword inside the object function to refer to the cat object and access its name property.

const cat = {

name: 'Pipey',

age: 8,

whatName() {

return this.name

}

};

console.log(cat.whatName());

// Output: Pipey

### javascript function this

Every JavaScript function or method has a this context. For a function defined inside of an object, this will refer to that object itself. For a function defined outside of an object, this will refer to the global object (window in a browser, global in Node.js).

const restaurant = {

numCustomers: 45,

seatCapacity: 100,

availableSeats() {

// this refers to the restaurant object

// and it's used to access its properties

return this.seatCapacity - this.numCustomers;

}

}

### JavaScript Arrow Function this Scope

JavaScript arrow functions do not have their own this context, but use the this of the surrounding lexical context. Thus, they are generally a poor choice for writing object methods.

Consider the example code:

loggerA is a property that uses arrow notation to define the function. Since data does not exist in the global context, accessing this.data returns undefined.

loggerB uses method syntax. Since this refers to the enclosing object, the value of the data property is accessed as expected, returning "abc".

const myObj = {

data: 'abc',

loggerA: () => { console.log(this.data); },

loggerB() { console.log(this.data); },

};

myObj.loggerA(); // undefined

myObj.loggerB(); // 'abc'

### getters and setters intercept property access

JavaScript getter and setter methods are helpful in part because they offer a way to intercept property access and assignment, and allow for additional actions to be performed before these changes go into effect.

const myCat = {

\_name: 'Snickers',

get name(){

return this.\_name

},

set name(newName){

//Verify that newName is a non-empty string before setting as name property

if (typeof newName === 'string' && newName.length > 0){

this.\_name = newName;

} else {

console.log("ERROR: name must be a non-empty string");

}

}

}

### javascript factory functions

A JavaScript function that returns an object is known as a factory function. Factory functions often accept parameters in order to customize the returned object.

// A factory function that accepts 'name',

// 'age', and 'breed' parameters to return

// a customized dog object.

const dogFactory = (name, age, breed) => {

return {

name: name,

age: age,

breed: breed,

bark() {

console.log('Woof!');

}

};

};

### javascript getters and setters restricted

JavaScript object properties are not private or protected. Since JavaScript objects are passed by reference, there is no way to fully prevent incorrect interactions with object properties.

One way to implement more restricted interactions with object properties is to use getter and setter methods.

Typically, the internal value is stored as a property with an identifier that matches the getter and setter method names, but begins with an underscore (\_).

const myCat = {

\_name: 'Dottie',

get name() {

return this.\_name;

},

set name(newName) {

this.\_name = newName;

}

};

// Reference invokes the getter

console.log(myCat.name);

// Assignment invokes the setter

myCat.name = 'Yankee';

## Evidencia Examen unidad 2

# Semana 7 – Unidad 3 Estructuras lineales - Pilas.

02, 03 y 05

MATRICES

Revisar matrices

¡Buen trabajo! En esta lección, aprendimos estos conceptos sobre matrices:

Las matrices son listas que almacenan datos en JavaScript.

Las matrices se crean entre paréntesis [].

Cada elemento dentro de una matriz está en una posición numerada, o índice, que comienza en 0.

Podemos acceder a un elemento de una matriz utilizando su índice, con una sintaxis como: myArray[0].

También podemos cambiar un elemento en una matriz usando su índice, con sintaxis como myArray[0] = 'new string';

Las matrices tienen una lengthpropiedad que le permite ver cuántos elementos hay en una matriz.

Las matrices tienen sus propios métodos , incluidos .push()y .pop(), que agregan y eliminan elementos de una matriz, respectivamente.

Las matrices tienen muchos métodos que realizan diferentes tareas, como .slice()y .shift(). Puede encontrar documentación en el sitio web de Mozilla Developer Network .

Algunos métodos integrados están mutando, lo que significa que el método cambiará la matriz, mientras que otros no están mutando. Siempre puedes consultar la documentación.

Las variables que contienen matrices se pueden declarar con leto const. Incluso cuando se declaran con const, las matrices siguen siendo mutables. Sin embargo, una variable declarada con constno se puede reasignar.

Las matrices mutadas dentro de una función mantendrán ese cambio incluso fuera de la función.

Las matrices se pueden anidar dentro de otras matrices.

Para acceder a elementos en índices de cadena de matrices anidadas utilizando notación entre corchetes.

¡Aprender a trabajar y manipular matrices te ayudará a trabajar con grandes cantidades de datos!

Instrucciones

¿Buscas más formas de practicar? Considere probar estos:

Utilice la .lengthpropiedad para encontrar el último elemento de una matriz.

Utilice los otros métodos en la documentación de MDN que no se mencionan en la lección.

Toma todos los elementos de una matriz y forma una cadena.

Encuentre el valor de retorno de llamar .push()a una matriz.

Anida una matriz dentro de una matriz.

Acceda a un elemento en la matriz anidada.

BUCLES

La palabra clave de descanso

Imagínese que estamos buscando adoptar un perro. Planeamos ir al refugio todos los días durante un año y luego rendirnos. Pero ¿y si conocemos a nuestro perro soñado el día 65? No queremos seguir yendo al refugio durante los próximos 300 días sólo porque nuestro plan original era ir durante un año entero. En nuestro código, cuando queremos detener la ejecución de un bucle aunque no se haya cumplido la condición de detención original que escribimos para nuestro bucle, podemos usar la palabra clave break.

La breakpalabra clave permite que los programas “salgan” del bucle desde dentro del bloque del bucle.

Veamos la sintaxis de una breakpalabra clave:

for (let i = 0; i < 99; i++) {

if (i > 2 ) {

break;

}

console.log('Banana.');

}

console.log('Orange you glad I broke out the loop!');

Este es el resultado del código anterior:

Banana.

Banana.

Banana.

Orange you glad I broke out the loop!

break ¡ Las declaraciones pueden ser especialmente útiles cuando recorremos grandes estructuras de datos! Con las pausas, podemos agregar condiciones de prueba además de la condición de parada y salir del ciclo cuando se cumplan.

Instrucciones

Punto de control 1 habilitado

1 .

Registre cada elemento rapperArrayen un forbucle con la variable iteradora i.

¿Atascado? Obtén un consejo

La instrucción del paso 2 del punto de control no está disponible hasta que se completen los pasos anteriores

2 .

Después del forciclo, registre la cadena "And if you don't know, now you know."en la consola. Nota: dado que hay un carácter de comilla simple, 'en nuestra cadena, podemos usar comillas dobles alrededor de la cadena para asegurarnos de que se imprima el carácter.

La instrucción del paso 3 del punto de control no está disponible hasta que se completen los pasos anteriores

3 .

Agregue un breakbloque dentro del bucle que salga del bucle si el elemento en el índice actual rapperArrayes 'Notorious B.I.G.'.

Registre el elemento antes de salir del bucle.

const rapperArray = ["Lil' Kim", "Jay-Z", "Notorious B.I.G.", "Tupac"];

let tipos\_mascotas = [

"Perro",

"Gato",

"Conejo",

"Hámster",

"Cobaya",

"Pez",

"Tortuga",

"Cerdo",

"Caballo",

"Ardilla",

"Chinchilla",

"Iguana",

"Serpiente",

"Loro",

"Canario",

"Hamster chino",

"Hamster ruso",

"Hamster dorado",

"Hamster roborowski",

"Hámster sirio",

"Cobaya de pelo corto",

"Cobaya de pelo largo",

"Cobaya de pelo semilargo",

"Pez Betta",

"Pez Guppy",

"Pez Molly",

"Pez Arcoíris",

"Pez Neón",

"Pez Espada",

"Pez Disco",

"Pez Gurami",

"Pez Ángel",

"Pez Carpa Koi",

"Tortuga Terrestre",

"Tortuga Acuática",

"Cerdo doméstico",

"Caballo doméstico",

"Ardilla Gris",

"Ardilla Roja",

"Chinchilla doméstica",

"Iguana Verde",

"Boa Constrictor",

"Pitón Birmana",

"Loro Gris Africano",

"Canario Malinois",

"Hamster enano Roborovski",

"Hámster enano Campbell",

"Hámster enano Winter White",

"Cobaya de pelo satinado",

"Pez Betta Gigante",

"Pez Guppy Albino",

"Pez Molly Albino",

"Pez Arcoíris Albino",

"Pez Neón Albino",

"Pez Espada Albino",

"Pez Disco Albino",

"Pez Gurami Albino",

"Pez Ángel Albino",

"Pez Carpa Koi Albino",

"Tortuga Mora",

"Tortuga de Agua dulce",

"Cerdo Vietnamita",

"Caballo Pura Sangre",

"Ardilla Voladora",

"Chinchilla Gigante",

"Iguana Terrestre",

"Boa Constrictor Albino",

"Pitón Birmana Albino",

"Loro Gris Africano Albino",

"Canario Rojo",

"Hamster enano Roborovski Albino",

"Hámster enano Campbell Albino",

"Hámster enano Winter White Albino",

"Cobaya Gigante",

"Pez Betta Halfmoon",

"Pez Guppy Halfmoon",

"Pez Molly Halfmoon",

"Pez Arcoíris Halfmoon",

"Pez Neón Halfmoon",

"Pez Espada Halfmoon",

"Pez Disco Halfmoon",

"Pez Gurami Halfmoon",

"Pez Ángel Halfmoon",

"Pez Carpa Koi Halfmoon",

"Tortuga Leopardo",

"Tortuga de Tierra",

"Cerdo Duroc",

"Caballo Árabe",

"Ardilla Gigante Gris",

"Chinchilla Peruana",

"Iguana Iguana",

"Boa Constrictor Concolor",

"Pitón Reticulada",

"Loro Amazona",

"Canario Amarillo",

"Hamster enano Roborovski Blanco",

"Hámster enano Campbell Blanco",

"Hámster enano Winter White Blanco",

"Cobaya de pelo largo Blanco",

"Pez Betta Halfmoon Albino",

"Pez Guppy Halfmoon Albino",

"Pez Molly Halfmoon Albino",

"Pez Arcoíris Halfmoon Albino",

"Pez Neón Halfmoon Albino",

"Pez Espada Halfmoon Albino",

"Pez Disco Halfmoon Albino",

"Pez Gurami Halfmoon Albino",

"Pez Ángel Halfmoon Albino",

"Pez Carpa Koi Halfmoon Albino",

"Tortuga Gigante de Galápagos",

"Tortuga de Mar",

"Cerdo Landrace",

"Caballo Clydesdale"];

console.log(tipos\_mascotas.length);

// Write your code below

for(let dia = 1; dia<=113; dia++){

if(dia === 60){

console.log("Felicidades tu mascota es el ", tipos\_mascotas[dia]);

break;

}

}

BUCLES

Revisar

¡Gran trabajo! En esta lección, aprendimos cómo escribir código más limpio con bucles . Ahora sabes:

Los bucles realizan acciones repetitivas para que no tengamos que codificar ese proceso manualmente cada vez.

Cómo escribir forbucles con una variable iteradora que aumenta o disminuye

Cómo usar un forbucle para iterar a través de una matriz

Un forbucle anidado es un bucle dentro de otro bucle.

whileLos bucles permiten diferentes tipos de condiciones de parada.

Las condiciones de parada son cruciales para evitar bucles infinitos.

do...whileLos bucles ejecutan el código al menos una vez; solo verifican la condición de detención después de la primera ejecución.

La breakpalabra clave permite que los programas dejen un bucle durante la ejecución de su bloque.

FUNCIONES DE ORDEN SUPERIOR

Introducción

A menudo no somos conscientes de la cantidad de suposiciones que hacemos cuando nos comunicamos en nuestras lenguas nativas. Si le dijéramos que "cuente hasta tres", esperaríamos que dijera los números "uno", "dos" y "tres". Supusimos que sabrías empezar con “uno” y terminar con “tres”. Con la programación, debemos ser más explícitos con nuestras instrucciones a la computadora. Así es como podríamos decirle a la computadora que "cuente hasta tres":

for (let i = 1; i <= 3; i++) {

console.log(i);

}

Cuando hablamos con otros humanos, compartimos un vocabulario que proporciona formas rápidas de comunicar conceptos complicados. Cuando decimos la palabra "hornear", nos recuerda una subrutina familiar: precalentar un horno, poner algo en el horno durante un tiempo determinado y finalmente retirarlo. Esto nos permite abstraer muchos de los detalles y comunicar conceptos clave de manera más concisa. En lugar de enumerar todos esos detalles, podemos decir: "Hicimos un pastel" y aún así transmitirle todo ese significado.

En esta lección, aprenderemos a utilizar la “abstracción” en la programación escribiendo funciones . Además de permitirnos reutilizar nuestro código, las funciones ayudan a crear programas claros y legibles. Si lo encuentra countToThree()en un programa, es posible que pueda adivinar rápidamente qué hace la función sin tener que detenerse y leer el cuerpo de la función.

También aprenderemos una forma de agregar otro nivel de abstracción a nuestra programación: funciones de orden superior . Las funciones de orden superior son funciones que aceptan otras funciones como argumentos y/o devuelven funciones como salida. Esto nos permite construir abstracciones sobre otras abstracciones, del mismo modo que "Organizamos una fiesta de cumpleaños" es una abstracción que puede basarse en la abstracción "Hicimos un pastel".

# Semana 8 – Unidad 3 Estructuras lineales - Representación en memoria y Operaciones básicas.

9, 10, 12 octubre.

FUNCIONES DE ORDEN SUPERIOR

Introducción

A menudo no somos conscientes de la cantidad de suposiciones que hacemos cuando nos comunicamos en nuestras lenguas nativas. Si le dijéramos que "cuente hasta tres", esperaríamos que dijera los números "uno", "dos" y "tres". Supusimos que sabrías empezar con “uno” y terminar con “tres”. Con la programación, debemos ser más explícitos con nuestras instrucciones a la computadora. Así es como podríamos decirle a la computadora que "cuente hasta tres":

for (let i = 1; i <= 3; i++) {

console.log(i);

}

Cuando hablamos con otros humanos, compartimos un vocabulario que proporciona formas rápidas de comunicar conceptos complicados. Cuando decimos la palabra "hornear", nos recuerda una subrutina familiar: precalentar un horno, poner algo en el horno durante un tiempo determinado y finalmente retirarlo. Esto nos permite abstraer muchos de los detalles y comunicar conceptos clave de manera más concisa. En lugar de enumerar todos esos detalles, podemos decir: "Hicimos un pastel" y aún así transmitirle todo ese significado.

En esta lección, aprenderemos a utilizar la “abstracción” en la programación escribiendo funciones . Además de permitirnos reutilizar nuestro código, las funciones ayudan a crear programas claros y legibles. Si lo encuentra countToThree()en un programa, es posible que pueda adivinar rápidamente qué hace la función sin tener que detenerse y leer el cuerpo de la función.

También aprenderemos una forma de agregar otro nivel de abstracción a nuestra programación: funciones de orden superior . Las funciones de orden superior son funciones que aceptan otras funciones como argumentos y/o devuelven funciones como salida. Esto nos permite construir abstracciones sobre otras abstracciones, del mismo modo que "Organizamos una fiesta de cumpleaños" es una abstracción que puede basarse en la abstracción "Hicimos un pastel".

Instrucciones

El uso de la abstracción nos permite escribir más código modular, que es más fácil de leer y depurar. ¡ Haz clic en Siguiente para seguir aprendiendo este nuevo concepto!

FUNCIONES DE ORDEN SUPERIOR

Funciones como datos

Las funciones de JavaScript se comportan como cualquier otro tipo de datos en el lenguaje; Podemos asignar funciones a variables y podemos reasignarlas a nuevas variables.

A continuación, tenemos un nombre de función molestamente largo que perjudica la legibilidad de cualquier código en el que se use. Nota: Si la sintaxis de la función siguiente no le resulta familiar, vuelva a visitar el ejercicio de funciones de flecha para actualizar sus conocimientos sobre la notación de flechas de ES6.

const announceThatIAmDoingImportantWork = () => {

console.log("I’m doing very important work!");

};

Supongamos que esta función realiza un trabajo importante y debe llamarse repetidamente. Para cambiar el nombre de esta función sin sacrificar el código fuente, podemos reasignar la función a una variable con un nombre adecuadamente corto:

const busy = announceThatIAmDoingImportantWork;

busy(); // This function call barely takes any space!

busyes una variable que contiene una referencia a nuestra función original. Si pudiéramos buscar la dirección en la memoria de busyy la dirección en la memoria de announceThatIAmDoingImportantWorkapuntarían al mismo lugar. Nuestra nueva busy()función se puede invocar entre paréntesis como si ese fuera el nombre que le dimos originalmente a nuestra función.

Observe cómo asignamos announceThatIAmDoingImportantWorksin paréntesis como valor a la busyvariable. Queremos asignar el valor de la función en sí, no el valor que devuelve cuando se invoca.

const animals = ['Hen', 'elephant', 'llama', 'leopard', 'ostrich', 'Whale', 'octopus', 'rabbit', 'lion', 'dog'];

// Create the secretMessage array below

const secretMessage = animals.map(animal =>{

  return animal[0];

});

console.log(secretMessage.join(''));

const bigNumbers = [100, 200, 300, 400, 500];

// Create the smallNumbers array below

const smallNumbers = bigNumbers.map(number =>{

  return number/100;

});

console.log(smallNumbers);

const randomNumbers = [375, 200, 3.14, 7, 13, 852];

// Call .filter() on randomNumbers below

const smallNumbers = randomNumbers.filter(

  rNumber => {

    return rNumber < 250;

  }

);

const favoriteWords = ['nostalgia', 'hyperbole', 'fervent', 'esoteric', 'serene'];

// Call .filter() on favoriteWords below

const longFavoriteWords = favoriteWords.filter(

  lWords => {

    return lWords.length > 7;

  }

);

console.log(longFavoriteWords);

En JavaScript, las funciones son objetos de primera clase . Esto significa que, al igual que otros objetos que haya encontrado, las funciones de JavaScript pueden tener propiedades y métodos .

Dado que las funciones son un tipo de objeto, tienen propiedades como .lengthy .namey métodos como .toString(). Puede ver más sobre los métodos y propiedades de las funciones en la documentación .

Las funciones son especiales porque podemos invocarlas, pero aun así podemos tratarlas como cualquier otro tipo de datos. ¡Practiquemos un poco haciendo eso!

Instrucciones

Punto de control 1 superado

1 .

Hemos definido una función con un nombre muy largo: checkThatTwoPlusTwoEqualsFourAMillionTimes(). Esta función tarda mucho en ejecutarse. Comprueba si 2 + 2 = 4, ¡pero lo hace un millón de veces (solo para estar realmente seguro)!

Usando const, declare una variable con un nombre más corto, isTwoPlusTwocon la que será más fácil trabajar. Asigne checkThatTwoPlusTwoEqualsFourAMillionTimescomo su valor.

¿Atascado? Obtén un consejo

Punto de control 2 superado

2 .

Invoca tu isTwoPlusTwo()función.

¿Atascado? Obtén un consejo

Punto de control 3 superado

3 .

Hmmm, si olvidamos el nombre original de nuestra función. ¿Hay alguna manera de que podamos resolverlo?

Utilice isTwoPlusTwola console.log()propiedad namede la función que le asignamos isTwoPlusTwo.

Funciones como parámetros

Como sabes, un parámetro es un marcador de posición para los datos que se pasan a una función. Dado que las funciones pueden comportarse como cualquier otro tipo de datos en JavaScript, puede que no le sorprenda saber que las funciones pueden aceptar otras funciones como parámetros. Una función de orden superior es una función que acepta funciones como parámetros, devuelve una función o ambas cosas. Llamamos funciones de devolución de llamada a las funciones que se pasan como parámetros . Las funciones de devolución de llamada se invocan durante la ejecución de la función de orden superior.

When we invoke a higher-order function, and pass another function in as an argument, we don’t invoke the argument function. Invoking it would evaluate to passing in the return value of that function call. With callback functions, we pass in the function itself by typing the function name without the parentheses:

const higherOrderFunc = param => {

param();

return `I just invoked ${param.name} as a callback function!`

}

const anotherFunc = () => {

return 'I\'m being invoked by the higher-order function!';

}

higherOrderFunc(anotherFunc);

We wrote a higher-order function higherOrderFunc that accepts a single parameter, param. Inside the body, param gets invoked using parentheses. And finally, a string is returned, telling us the name of the callback function that was passed in.

Below the higher-order function, we have another function aptly named anotherFunc. This function aspires to be called inside the higher-order function.

Por último, invocamos a higherOrderFunc(), pasando a ser anotherFuncsu argumento, cumpliendo así sus sueños de ser llamado por la función de orden superior.

higherOrderFunc(() => {

for (let i = 0; i <= 10; i++){

console.log(i);

}

});

En este ejemplo, invocamos higherOrderFunc()con una función anónima (una función sin nombre) que cuenta hasta 10. ¡Las funciones anónimas también pueden ser argumentos!

Practiquemos un poco escribiendo funciones de orden superior.

Instrucciones

Punto de control 1 superado

1 .

Aquí tenemos una función, addTwo()que suma 2 a lo que se le pasa. Debajo de eso, hemos creado lo que será una función de orden superior checkConsistentOutput(). El propósito de la función de orden superior será verificar el trabajo de addTwo(). ¡Empecemos!

Para comenzar, dentro del cuerpo de checkConsistentOutput(), declara dos variables: checkAy checkB:

checkAalmacena la suma valmás 2.

checkBalmacena la invocación de la devolución de funcllamada, con valcomo argumento.

¿Atascado? Obtén un consejo

Punto de control 2 superado

2 .

A continuación, debajo de las variables, escriba una declaración condicional que verifique si el valor de checkAes igual al valor checkB. Si es verdadero, devuelve el resultado de la función de devolución de llamada. Si es falso, devuelve la cadena 'inconsistent results'.

¿Atascado? Obtén un consejo

Punto de control 3 superado

3 .

Finalmente, usando console.log(), registra la invocación de checkConsistentOutput()con dos argumentos: la addTwo()función y cualquier número.

ITERADORES

El método .forEach()

El primer método de iteración que vamos a aprender es .forEach(). Con un nombre adecuado, .forEach()ejecutará el mismo código para cada elemento de una matriz.

Diagrama que describe las partes de un iterador de matriz, incluido el identificador de matriz, la sección que es el iterador y la función de devolución de llamada.

El código anterior registrará una lista de alimentos bien formateada en la consola. Exploremos la sintaxis de invocar .forEach().

groceries.forEach()llama al forEachmétodo en la groceriesmatriz.

.forEach()toma un argumento de la función de devolución de llamada. Recuerde, una función de devolución de llamada es una función que se pasa como argumento a otra función.

.forEach()recorre la matriz y ejecuta la función de devolución de llamada para cada elemento. Durante cada ejecución, el elemento actual se pasa como argumento a la función de devolución de llamada.

El valor de retorno de .forEach()siempre será undefined.

Otra forma de pasar una devolución de llamada .forEach()es utilizar la sintaxis de la función de flecha.

groceries.forEach(groceryItem => console.log(groceryItem));

También podemos definir una función de antemano para usarla como función de devolución de llamada.

function printGrocery(element){

console.log(element);

}

groceries.forEach(printGrocery);

El ejemplo anterior utiliza una declaración de función, pero también puede utilizar una expresión de función o una función de flecha.

Los tres fragmentos de código hacen lo mismo. En cada método de iteración de matriz, podemos usar cualquiera de los tres ejemplos para proporcionar una función de devolución de llamada como argumento para el iterador. Es bueno conocer las diferentes formas de pasar funciones de devolución de llamada como argumentos en iteradores porque los desarrolladores tienen diferentes preferencias estilísticas. No obstante, debido a la fuerte adopción de ES6, usaremos la sintaxis de la función de flecha en los ejercicios posteriores.

Instrucciones

Punto de control 1 habilitado

1 .

Itere sobre la fruitsmatriz para registrar I want to eat a más el nombre de cada fruta en la consola. Por ejemplo, I want to eat a mango.

Puede utilizar cualquier forma de devolución de llamada que prefiera.