**4-EJERCICIO-TIPO DE DATO**

En este desafío encontrarás una función llamada solution que recibe un parámetro desde la función solution.

Recuerda que el parámetro valor será distinto por cada distinta forma en que ejecutemos la función solution.

Por ejemplo:

Dados los siguientes llamados a la función solution:

solution(1)

solution("Dieguillo")

solution(true)

Debes obtener los siguientes resultados:

"number"

"string"

"boolean"

**7-EJERCICIO-CALCULA LA PROPINA**

En este desafío tendrás que calcular la propina que deben dejar los clientes de un restaurante en función de su consumo.

La función calculateTip recibirá dos parámetros, billAmount que representa el costo total de lo que se haya consumido y tipPercentage que representa el porcentaje de propina a dejar. Ambos valores serán de tipo Number y siempre serán positivos, incluyendo el 0. La función deberá devolver el valor de la propina como un número.

Tendrás inputs y outputs como los siguientes 👇

Ejemplo 1:

Input: calculateTip(100, 10);

Output: 10;

Ejemplo 2:

Input: calculateTip(1524.33, 25);

Output: 3

**11-EJERCICIO-AÑO BISIESTO**

En este desafío, debes crear la lógica de la función isLeapYear, que determina si un año es bisiesto o no. Un año es bisiesto si cumple con las siguientes condiciones:

* Es divisible por 4, pero no por 100.
* Si es divisible por 100 debe serlo por 400 también.

La función isLeapYear recibe un único parámetro: el año a evaluar. Debe devolver true si el año es bisiesto o false en caso contrario.

Toma en cuenta que la función debe ser capaz de manejar valores no enteros o negativos.

Ejemplo 1:

*Input:* 2000;

*Output:* true;

Ejemplo 2:

Input: -2024;

Output: false;

Ejemplo 3:

*Input:* 1984.25;

*Output:* false;

**13-EJERCICIO- Obtén información de mascotas según su tipo**

En este desafío recibirás una serie de tipos de mascotas junto con su edad.

Dependiendo de estos 2 factores tendrás que construir la función llamada getPetExerciseInfo la cual retornará una cadena de texto con la información necesaria acerca de cuanto ejercicio necesita hacer cada tipo de mascota.

La función recibirá las siguientes mascotas:

* perro
* gato
* ave

En caso de pasar una mascota la cual no sea de la lista deberá retornar "Tipo de mascota inválida"

Para cada tipo de mascota, la función retornará diferente información basada en la edad.

* Perros
  + Si la edad es menor al año, deberá retornar "Los cachorros necesitan pequeñas y frecuentes sesiones de juego"
  + Si la edad es entre 1 y 7 años, deberá retornar "Los perros a esta edad necesitan caminatas diarias y sesiones de juego"
  + Si la edad es mayor a 7 años, deberá retornar "Los perros viejos necesitan caminatas más cortas"
* gatos
  + Si la edad es menor al año, deberá retornar "Los gatitos necesitan frecuentes sesiones de juego"
  + Si la edad es entre 1 y 7 años, deberá retornar "Los gatos a esta edad necesitan jugar diariamente"
  + Si la edad es mayor a 7 años, deberá retornar "Los gatos viejos necesitan sesiones de juego más cortas"
* aves
  + Si la edad es menor al año, deberá retornar "Las aves jóvenes necesitan mucho espacio para volar"
  + Si la edad es entre 1 y 7 años, deberá retornar "Las aves necesitan jugar diariamente y un lugar para volar"
  + Si la edad es mayor a 7 años, deberá retornar "Las aves mayores necesitan descansar más, pero siguen ocupando un lugar para volar"

Tendrás inputs y outputs como los siguientes 👇

Ejemplo 1:

Input: getPetExerciseInfo("perro", 3)

Output: "Los perros a esta edad necesitan caminatas diarias y sesiones de juego"

Ejemplo 2:

Input: getPetExerciseInfo("gato", 8)

Output: "Los gatos viejos necesitan sesiones de juego más cortas"

Ejemplo 3:

Input: getPetExerciseInfo("Mamut", 25)

Output: "Tipo de mascota invalida"

**15-EJERCICIO-DIBUJA UN TRIANGULO**

En este desafío, debes dibujar un triángulo isósceles usando bucles.

Recibirás dos parámetros: size y character, que definen el tamaño y el carácter con el que se debe construir el triángulo, respectivamente. Además, el triángulo debe estar alineado a la derecha, lo que significa que la columna más derecha del triángulo debe estar en el borde derecho de la consola.

Recuerda que para hacer el salto de línea debes usar "\n", no olvides removerla de la última parte.

Tendrás inputs y outputs como los siguientes 👇

Ejemplo 1:

Input: printTriangle(5, "\*")

Output:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

Ejemplo 2:

Input: printTriangle(6, "$")

Output:

$

$$

$$$

$$$$

$$$$$

$$$$$$

**ANALISIS**

1. Objetivo
2. Dibujar un triángulo rectángulo donde de los dos parámetros uno será un numero que indicará cuantas líneas tendrá y otro el símbolo que será las piezas del triángulo, utilizando bucles.
3. En base a lo que se pide - (datos de entrada)
4. De entrada, un número que debe relacionarse con el carácter ya que la primera impresión debe ser espacios en blanco más un carácter al final de la fila (pegado al lado derecho) e ir disminuyendo los espacios en blanco y aumentando los caracteres sucesivamente.
5. En base a lo que se pide - (procedimiento platzi)

**17-EJERCICIO-ENCUENTRA MICHI FAMOSO**

En este desafío, debes encontrar al gatito más famoso con base en su número de seguidores.

Recibirás un array de objetos que incluirán las siguientes propiedades:

* name: nombre del gatito.
* followers: un array de números, donde cada uno representa los seguidores de cada red social.

Tu tarea es devolver un array con los nombres de los gatos que tienen solo el mayor número de seguidores. Si hay dos o más gatos con el mismo número máximo de seguidores, deberás incluirlos en el array de resultado, manteniendo el orden en el que aparecen en el array de entrada.

Tendrás inputs y outputs como los siguientes 👇

Ejemplo 1:

Input: findFamousCats([

{

name: "Luna",

followers: [500, 200, 300]

},

{

name: "Michi",

followers: [100, 300]

},

])

Output: ["Luna"]

Ejemplo 2:

Input: findFamousCats([

{

name: "Mimi",

followers: [320, 120, 70]

},

{

name: "Milo",

followers: [400, 300, 100, 200]

},

{

name: "Gizmo",

followers: [250, 750]

}

])

Output: ["Milo", "Gizmo"]

**19-EJERCICIO-OBTEN EL PROMEDIO DE LOS ESTUDIANTES**

En este desafío, deberás calcular el promedio general de una clase, así como el promedio individual de cada estudiante.

Para ello, se te proporcionará un array de objetos, cada uno de los cuales representará a un estudiante y tendrá las siguientes propiedades:

* name: El nombre del estudiante
* grades: Las notas de cada materia del estudiante

A partir de esta información, debes retornar un nuevo objeto que tenga la propiedad classAverage con el promedio de la clase y un array de students con los estudiantes y sus promedios individuales.

Es importante mencionar que los promedios deben ser calculados con precisión y se deben redondear a dos decimales para que los test pasen sin problema alguno. Puedes usar el método toFixed() el cual se usa de la siguiente manera 👇

const number = 100.32433;

number.toFixed(2); // "100.32"

👀 Ten en cuenta que este método regresa el número como un string y se espera que sea de tipo numérico.

Ejemplo:

Input: getStudentAverage([

{

name: "Pedro",

grades: [90, 87, 88, 90],

},

{

name: "Jose",

grades: [99, 71, 88, 96],

},

{

name: "Maria",

grades: [92, 81, 80, 96],

},

])

Output: {

classAverage: 88.17,

students: [

{

name: "Pedro",

average: 88.75

},

{

name: "Jose",

average: 88.5

},

{

name: "Maria",

average: 87.25

}

]

}

**ANALISIS:**

1. objetivo
2. calcular el promedio de una clase y promedio de cada estudiante.
3. Retornar un objeto que tenga la propiedad ClassAverage con el promedio de la clase, y un array students con el nombre del alumno y su promedio.
4. En base a lo que me pide - (datos de entrada)
5. Se da un array con objetos, dentro de estos objetos 2 propiedades: una el valor nombre del estudiante, dos con el valor lista de notas.
6. En base a lo que se pide – (procedimiento)
7. Paso 1: como me proporciona una lista de objetos tengo que utilizar el método “index” para acceder a cada elemento, utilizare el bucle “for” para utilizarlo como acceso a cada elemento utilizando la variable inicializada dentro del “for”.
8. Paso 2: para acceder a cada objeto debo usar la notación de punto, que me permitirá entrar

## 20-EJERCICIO-Encuentra el mayor palíndromo

En este desafío, debes crear una función que encuentre el palíndromo más largo en una lista de palabras.

Recibirás un único parámetro: un array de palabras. Si no hay ningún palíndromo en la lista, la función debe devolver null. Si hay más de un palíndromo con la misma longitud máxima, debes devolver el primer palíndromo encontrado en la lista.

Un palíndromo es una palabra que se puede leer de la misma manera tanto hacia adelante como hacia atrás.

Ejemplo 1:

Input: findLargestPalindrome(["racecar", "level", "world", "hello"])

Output: "racecar"

Ejemplo 2:

Input: findLargestPalindrome(["Platzi", "javascript", "html", "css"])

Output: null

## 26-EJERCICIO-Calculadora con closures

En este desafío tendrás que crear una calculadora mediante el uso de closures.

La calculadora debe contar con los siguientes métodos:

* add: recibe un número, lo suma al total y devuelve el resultado
* subtract: recibe un número, lo resta al total y devuelve el resultado
* multiply: recibe un número, lo multiplica al total y devuelve el resultado
* divide: recibe un número, lo divide al total y devuelve el resultado
* clear: reinicia el total a 0 y devuelve el resultado
* getTotal: devuelve el total actual.

Ejemplo 1:

Input:

const calculator = createCalculator()

calculator.add(10)

Output: 10

Ejemplo 2:

const calculator = createCalculator()

calculator.add(10)

calculator.subtract(-10)

Output: 20

Ejemplo 3:

const calculator = createCalculator()

calculator.add(10)

calculator.subtract(-10)

calculator.clear()

Output: 0

## 28-EJERCICIO-Crea tu propio método map

En este desafío debes desarrollar una implementación personalizada del método map utilizando funciones de orden superior.

Recibirás como parámetros un array y una función (func). El array contendrá un conjunto de elementos (números, objetos, strings, etc.) y la función se utilizará para aplicar una acción sobre cada elemento del array. Tu objetivo es devolver un nuevo array con los resultados de la función tal y como lo haría el método map.

Ejemplo 1:

Input: myMap([1,2,3,4], (num) => num \* 2)

Output: [2,4,6,8]

Ejemplo 2:

Input: myMap([

{name: "michi", age: 2},

{name: "firulais", age: 6}],

(pet) => pet.name)

Output: ["michi", "firulais"]

## 33-EJERCICIO-Crea un planificador de tareas

En este desafío, debes implementar la lógica de un planificador de tareas que permita agregar, eliminar y marcar como completadas las tareas, así como también mostrar un registro de las mismas. Para ello, debes construir la lógica de la función closure llamada createTaskPlanner para que devuelva los siguientes métodos:

* addTask(task): recibe un objeto que contiene la tarea y la agrega al array de tareas. La tarea debe estar conformada por las siguientes propiedades: id, name, priority, tags y completed, donde el estado completed se agrega automáticamente como falso al momento de agregar una tarea.
* removeTask(value): recibe el id o nombre de la tarea y la elimina del array de tareas.
* getTasks(): Devuelve el array de tareas.
* getPendingTasks(): Devuelve solo las tareas pendientes.
* getCompletedTasks(): Devuelve solo las tareas completadas.
* markTaskAsCompleted(value): Recibe el id o nombre de la tarea y la marca como completada.
* getSortedTasksByPriority(): Devuelve una copia de las tareas ordenadas según su prioridad (3: poco urgente, 2: urgente, 1: muy urgente), sin modificar la lista de tareas original.
* filterTasksByTag(tag): Filtra las tareas por una etiqueta específica.
* updateTask(taskId, updates): Buscar la tarea correspondiente con el id especificado y actualizar sus propiedades con las especificadas en el objeto updates.

Ejemplo 1:

Input:

const planner = createTaskPlanner();

planner.addTask({

id: 1,

name: "Comprar leche",

priority: 1,

tags: ["shopping", "home"]

});

planner.addTask({

id: 2,

name: "Llamar a Juan",

priority: 3,

tags: ["personal"]

});

planner.markTaskAsCompleted("Llamar a Juan")

Output:

planner.getCompletedTasks()

[{

id: 2,

name: "Llamar a Juan",

completed: true,

priority: 3,

tags: ["personal"]

}]

Ejemplo 2:

Input:

const planner = createTaskPlanner();

planner.addTask({

id: 1,

name: "Comprar leche",

priority: 1,

tags: ["shopping", "home"]

});

planner.addTask({

id: 2,

name: "Llamar a Juan",

priority: 3,

tags: ["personal"]

});

Output:

planner.filterTasksByTag("shopping")

[{

id: 1,

name: "Comprar leche",

completed: false,

priority: 3,

tags: ["shopping", "home"]

}]

## 35-EJERCICIO-Crea una promesa para mandar emails

En este desafío debes utilizar promesas para enviar un correo electrónico.

La función sendEmail recibe tres parámetros: email, subject y body, los cuales son necesarios para enviar un correo. Deberás implementar la lógica necesaria para usar promesas y enviar el correo después de 2 segundos.

En caso de faltar algún dato, deberás lanzar un error con el mensaje indicando que faltan campos para enviar el correo. Recuerda utilizar la siguiente sintaxis:

reject(new Error(message));

También recuerda que para usar setInterval o setTimeout debes usar el namespace de window de la siguiente manera para que las pruebas pasen correctamente.

window.setTimeout(() => {

// Código aquí

}, 1000);

Ejemplo 1:

Input:

sendEmail(

"test@mail.com",

"Nuevo reto",

"Únete a los 30 días de JS"

)

.then(result => console.log(result))

Output:

// Después de 2 segundos

{

email: "test@mail.com"

subject: "Nuevo reto",

body: "Únete a los 30 días de JS",

}

Input:

Ejemplo 2:

sendEmail(

"test@mail.com",

"",

"Únete a los 30 días de JS"

)

.then(result => console.log(result))

.catch(error => console.log(error))

Output:

// Después de 2 segundos

"Error: Hacen falta campos para enviar el email"

## 36-EJERCICIO-Evita el callback hell haciendo uso de Promises

En este desafío tienes un código base el cual se aplica una mala práctica de programación que conocemos como el **callback hell** y tu reto es evitarlo usando promesas.

Nota: No uses async/await, ya que este reto se trata de resolverlo mediante promesas con la función then.

Para solucionarlo vas a encontrar una función llamada runCode que no recibe parámetros de entrada que tiene el código base que tienes que refactorizar, además el archivo tasks.js con las funciones que tienes que ejecutar.

Dentro del cuerpo de la función runCode debes escribir tu solución y además pasar y deberías pasar las funciones del archivo tasks.js a promesas.

Ejemplo:

Input:

runCode()

.then(response => console.log(response));

Output:

["Task 1", "Task 2", "Task 3"]

En este desafío tienes un código base el cual se aplica una mala práctica de programación que conocemos como el **callback hell** y tu reto es evitarlo usando promesas.

Nota: No uses async/await, ya que este reto se trata de resolverlo mediante promesas con la función then.

Para solucionarlo vas a encontrar una función llamada runCode que no recibe parámetros de entrada que tiene el código base que tienes que refactorizar, además el archivo tasks.js con las funciones que tienes que ejecutar.

Dentro del cuerpo de la función runCode debes escribir tu solución y además pasar y deberías pasar las funciones del archivo tasks.js a promesas.

Ejemplo:

Input:

runCode()

.then(response => console.log(response));

Output:

["Task 1", "Task 2", "Task 3"]

## 38-EJERCICIO-Evita el callback hell haciendo uso de await

En este desafío tienes un código base el cual se aplica una mala práctica de programación que conocemos como el **callback hell** y tu reto es evitarlo usando async/await.

Para solucionarlo vas a encontrar una función llamada runCode que no recibe parámetros de entrada que tiene el código base que tienes que refactorizar, además el archivo tasks.js con las funciones que tienes que ejecutar.

Dentro del cuerpo de la función runCode debes escribir tu solución y además pasar y deberías pasar las funciones del archivo tasks.js a promesas.

Ejemplo:

Input:

await runCode();

Output:

["Task 1", "Task 2", "Task 3"]

## 41-ejercicio-Válida un formulario de registro

En este desafío deberás validar un formulario de registro de usuario.

Tu tarea es implementar la lógica de la función validateForm la cual recibirá como parámetro un objeto con los datos del formulario al igual que una lista de usurios registrados.

La función debe verificar que todos los campos requeridos del formulario (name, lastname, email y password) estén completos, si falta algún campo, debe lanzar un error especificando los campos faltantes.

Para lanzar dicho error debes usar la siguiente sintaxis

throw new Error("Faltan los siguientes campos: name, email, etc...");

Además, la función debe verificar si el email ingresado ya existe en la lista de usuarios registrados. Si el email ya está en uso, debe retornar un error especificando el email duplicado.

Si todo está correcto, se debe agregar el usuario a la lista de usuarios registrados con todos los datos excepto la contraseña y retornar un mensaje indicando que el registro fue exitoso junto con el nombre y apellido del usuario.

Ejemplo 1

Input:

const formData = {

name: "Juan",

lastname: "Perez",

email: "juan@example.com",

password: "123456"

}

const registeredUsers = [

{ name: "Pedro", lastname: "Gomez", email: "pedro@example.com" },

{ name: "Maria", lastname: "Garcia", email: "maria@example.com" },

]

validateForm(formData, registeredUsers)

Output:

"Tu registro fue exitoso Juan Perez"

Ejemplo 2

Input:

const formData = {

name: "Juan",

password: "123456",

};

const registeredUsers = [

{ name: "Pedro", lastname: "Gomez", email: "pedro@example.com" },

{ name: "Maria", lastname: "Garcia", email: "maria@example.com" },

]

validateForm(formData, registeredUsers)

Output:

"Faltan los siguientes campos requeridos: lastname, email"