JavaScript I

(Semana 3)

MigraCode

Alexandra Yamaui

Propiedades de arreglos

Objetos:

- Un **Objeto** es una colección de propiedades
- Una **Propiedad** es una asociación entre una clave y un valor

example.js

```
// person Object
let person = {
  first_name: "Alexandra",
  last_name: "Yamaui",
  greeting: function (msg) {
    return `${msg}, ${this.first_name}`
  }
}
// Example Array
let arr = [1, 2, 3];
console.log(arr.length);
```

Propiedades de arreglos

Objetos:

- Un **Objeto** es una colección de propiedades
- Una **Propiedad** es una asociación entre una clave y un valor

example.js

```
// person Object
let person = {
  first_name: "Alexandra",
  last_name: "Yamaui",
  greeting: function (msg) {
    return `${msg}, ${this.first_name}`
  }
}
// Example Array
let arr = [1, 2, 3];
console.log(arr.length);
```

Propiedades de arreglos

Objetos:

- Un **Objeto** es una colección de propiedades
- Una **Propiedad** es una asociación entre una clave y un valor

example.js

```
// person Object
let person = {
  first_name: "Alexandra",
  last_name: "Yamaui",
  greeting: function (msg) {
    return `${msg}, ${this.first_name}`
  }
}
// Example Array
let arr = [1, 2, 3];
console.log(arr.length);
```

Métodos de arreglos

Método:

- Es una **Propiedad** que tiene como valor una **función**

example.js

```
// person Object
let person = {
  ...
  greeting: function (msg) {
    return `${msg}, ${this.first_name}`   <- Method
  }
}
console.log(person.name); // "Alexandra"
console.log(person.last_name); // "Yamaui"
console.log(person.greeting("Hello!")); //
"Hello!, Alexandra"
```

Métodos de arreglos

Método:

- Es una **Propiedad** que tiene como valor una **función**

example.js

```
// person Object
let person = {
  ...
  greeting: function (msg) {
    return `${msg}, ${this.first_name}`
  }
}

console.log(person.greeting("Hello!")); //
"Hello!, Alexandra"

function greeting(msg, name) {...}
console.log(greeting("Hello!, ",
"Alexandra")); // "Hello!, Alexandra"
```

Métodos de arreglos

.sort():

- Ordena los elementos de un arreglo y retorna el arreglo ordenado (método mutable)

example.js

```
const unorderedLetters = ["z", "v", "b", "f", "g"];
const orderedLetters = unorderedLetters.sort();
const unorderedNumbers = [8, 5, 1, 4, 2];
const orderedNumbers = unorderedNumbers.sort();

console.log(orderedLetters); // logs [ 'b', 'f', 'g', 'v', 'z' ]
console.log(unorderedLetters); // ?

console.log(orderedNumbers); // logs [ 1, 2, 4, 5, 8 ]
console.log(unorderedNumbers); // ?
```

Métodos de arreglos

.sort():

- Ordena los elementos de un arreglo y retorna el arreglo ordenado (método mutable)

example.js

```
const unorderedLetters = ["z", "v", "b", "f", "g"];
const orderedLetters = unorderedLetters.sort();
const unorderedNumbers = [8, 5, 1, 4, 2];
const orderedNumbers = unorderedNumbers.sort();

console.log(orderedLetters); // logs [ 'b', 'f', 'g', 'v', 'z' ]
console.log(unorderedLetters); // logs [ 'b', 'f', 'g', 'v', 'z' ]
console.log(orderedNumbers); // logs [ 1, 2, 4, 5, 8 ]
console.log(unorderedNumbers); // logs [ 1, 2, 4, 5, 8 ]
```

Métodos de arreglos

.concat(array2, array3, ...):

- Concatena dos o más arreglos
- **NO** modifica el arreglo original, sino que retorna uno nuevo

example.js

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3); // ["a", "b", "c", "d",
"e", "f"]
console.log(array1); // ?
console.log(array2); // ?
```

Métodos de arreglos

.concat(array2, array3, ...):

- Concatena dos o más arreglos
- **NO** modifica el arreglo original, sino que retorna uno nuevo

example.js

```
const array1 = ['a', 'b', 'c'];
const array2 = ['d', 'e', 'f'];
const array3 = array1.concat(array2);

console.log(array3); // ["a", "b", "c", "d",
"e", "f"]
console.log(array1); // ['a', 'b', 'c']
console.log(array2); // ['d', 'e', 'f']
```

Métodos de arreglos

.slice(inicio, fin):

- Devuelve un **nuevo** arreglo, que es una porción del arreglo original incluyendo los elementos desde la posición *inicio* hasta una posición **antes** de *fin* (el elemento en la posición *fin* no se incluye)

example.js

```
let arr = [0, 1, 2, 3, 4]
arr.slice(0, 2) // The element at index 2 is NOT included -> [0, 1]

["a", "b", "c", "d"].slice(1, 2) // ?
```

Métodos de arreglos

.slice(inicio, fin):

- Devuelve un **nuevo** arreglo, que es una porción del arreglo original incluyendo los elementos desde la posición *inicio* hasta una posición **antes** de *fin* (el elemento en la posición *fin* no se incluye)

example.js

```
let arr = [0, 1, 2, 3, 4]
arr.slice(0, 2) // The element at index 2 is
NOT included -> [0, 1]

["a", "b", "c", "d"].slice(1, 2) // ['b']
```

Métodos de arreglos

.slice(inicio, fin):

- Si no se pasa ningún parámetro, se devuelve una **copia real** del arreglo original

example.js

```
let arr = [0, 1, 2, 3, 4]
arr.slice(0, 2) // The element at index 2 is
NOT included -> [0, 1]

["a", "b", "c", "d"].slice(1, 2) // ['b']

["a", "b", "c", "d"].slice(1) // ?

["a", "b", "c", "d"].slice() // ?

["a", "b", "c", "d"].slice(1, 1) // ?
```

Métodos de arreglos

.slice(inicio, fin):

- Si no se pasa ningún parámetro, se devuelve una **copia real** del arreglo original

example.js

```
let arr = [0, 1, 2, 3, 4]
arr.slice(0, 2) // The element at index 2 is
NOT included -> [0, 1]

["a", "b", "c", "d"].slice(1, 2) // ['b']

["a", "b", "c", "d"].slice(1) // ['b', 'c',
'd']

["a", "b", "c", "d"].slice() // ['a', 'b',
'c', 'd']
["a", "b", "c", "d"].slice(1, 1) // []
```

Copia real vs superficial

- Aplica sólo a objetos y arreglos (no a tipos de datos primitivos como los números y strings)

Copia real vs superficial

Copia superficial (shallow):

- Crea una **referencia** al objeto
- **No** copia los valores
- Si modificamos la copia (referencia), modificamos el objeto original

example.js

```
let arr = [0, 1, 2, 3, 4]
let shallow_copy_arr = arr // Shallow copy

shallow_copy_arr[0] = -1

console.log(shallow_copy_arr) // [-1, 1, 2, 3, 4]
console.log(arr) // [-1, 1, 2, 3, 4]
```

Copia real vs superficial

Copia real o profunda (deep):

- Crea una copia **real**
- **Sí** copia los valores
- Si modificamos la copia, el objeto original no se modifica

example.js

```
let arr = [0, 1, 2, 3, 4]
let deep_copy_arr = [...arr] // Deep copy
let deep_copy_arr2 = arr.slice() // Deep copy

deep_copy_arr[0] = -1
deep_copy_arr2[0] = -2

console.log(deep_copy_arr) // [-1, 1, 2, 3, 4]
console.log(deep_copy_arr2) // [-2, 1, 2, 3,
4]
console.log(arr) // [0, 1, 2, 3, 4]
```

Métodos de arreglos

.includes(elem):

- Devuelve true si el valor pasado como parámetro está en el arreglo, de lo contrario devuelve false

example.js

```
const mentors = ["Daniel", "Irini",
"Ashleigh", "Rob", "Etzali"];

function isAMentor(name) {
return mentors.includes(name); // ?
}

console.log("Is Rukmuni a mentor?");
console.log(isAMentor("Rukmini")); // ?
```

Métodos de arreglos

.includes(elem):

- Devuelve true si el valor pasado como parámetro está en el arreglo, de lo contrario devuelve false

example.js

```
const mentors = ["Daniel", "Irini",
"Ashleigh", "Rob", "Etzali"];

function isAMentor(name) {
return mentors.includes(name); // mentors is
global
}

console.log("Is Rukmuni a mentor?");
console.log(isAMentor("Rukmini")); // false
```

Métodos de arreglos

.join(separador):

- Une todos los elementos de un arreglo para formar un string usando el caracter pasado como parámetro como separador
- Si no se pasa ningún parámetro se utiliza la coma (,) como separador

example.js

```
["H", "e", "l", "l", "o"].join(); //
'H,e,l,l,o'
["H", "e", "l", "l", "o"].join("--"); //
'H--e--l--l--o'
[1, true, "l", "l", "o"].join(); //
'1,true,l,l,o'
```

Métodos de Strings

.split(patrón):

- Divide un string usando como divisor el patrón pasado como parámetro y retorna un **arreglo** con los sub-strings creados

example.js

```
'H,e,l,l,o'.split(","); // ["H", "e", "l",
"l", "o"]

'Hi! My name is Alexandra'.split("!"); //
["Hi",    " My name is Alexandra"]

'Hi! My name is Alexandra'.split(" "); // ?


'Hi! My name is Alexandra'.split("Alexandra");
// ?
```

Métodos de Strings

.split(patrón):

- Divide un string usando como divisor el patrón pasado como parámetro y retorna un **arreglo** con los sub-strings creados

example.js

```
'H,e,l,l,o'.split(","); // ["H", "e", "l",
"l", "o"]

'Hi! My name is Alexandra'.split("!"); //
["Hi",    " My name is Alexandra"]

'Hi! My name is Alexandra'.split(" "); // [
'Hi!', 'My', 'name', 'is', 'Alexandra' ]

'Hi! My name is Alexandra'.split("Alexandra");
// [ 'Hi! My name is ', '' ]
```

Ejercicios

Ejercicio B (10 minutos)

- Crea un arreglo con los nombres de tus compañeros
- Combínalo con otro arreglo que contenga el nombre de los estudiantes de otra clase
- Imprime (`console.log`) los nombres en orden alfabético
- Crea una función que reciba un `nombre` y un arreglo de nombres y retorne un string. Si el nombre no está en el arreglo, debe retornar `<nombre> no está en la clase con <nombres en el arreglo>`. Si el nombre está en el arreglo, debe retornar `<nombre> está en la clase con <nombres en el arreglo>`.

Concatenación de Métodos

- Invocar múltiples métodos uno después de otro sobre el mismo objeto o elementos
- El valor de retorno del primer método se usa como parámetro de entrada del siguiente y así sucesivamente...

example.js

```
const name = "   Alexandra";
console.log(name.trim().toLowerCase()); //
"alexandra"
```

Métodos de Arreglos

.map(funcion):

example.js

```
const mentors = ["Daniel ", "irina ", "
Gordon", "ashleigh "];

function tidy(name) {
 // trim() method removes blank spaces at the
beginning and at the end of a string
 return name.trim().toLowerCase();
}

let tidy_mentors = [];

for (let i = 0; i < mentors.length; i++) {
 const tidy_mentor = tidy(mentors[i]);
 tidy_mentors.push(tidy_mentor);
}
```

Métodos de Arreglos

.map(funcion):

- Aplica la función a cada elemento del arreglo, es decir, invoca la función usando como parámetro cada elemento del arreglo y retorna un nuevo arreglo con el resultado.

example.js

```
const mentors = ["Daniel ", "irina ", "
Gordon", "ashleigh "];

function tidy(name) {...}

let tidy_mentors = [];

for (let i = 0; i < 100; i++) {
 const tidy_mentor = tidy(mentors[i]);
 tidy_mentors.push(tidy_mentor);
}

// Equivalent
console.log(mentors.map(tidy)); // ?
console.log(mentors); // ?
```

Métodos de Arreglos

.map(funcion):

- Aplica la función a cada elemento del arreglo, es decir, invoca la función usando como parámetro cada elemento del arreglo y retorna un nuevo arreglo con el resultado.

example.js

```
const mentors = ["Daniel ", "irina ", "
Gordon", "ashleigh "];

function tidy(name) {...}

let tidy_mentors = [];

for (let i = 0; i < 100; i++) {
 const tidy_mentor = tidy(mentors[i]);
 tidy_mentors.push(tidy_mentor);
}

// Equivalent (tidy is passed as parameter)
console.log(mentors.map(tidy)); // [ 'daniel',
'irina', 'gordon', 'ashleigh' ]
console.log(mentors); // ?
```

Métodos de Arreglos

.map(funcion):

- Aplica la función a cada elemento del arreglo, es decir, invoca la función usando como parámetro cada elemento del arreglo y retorna un nuevo arreglo con el resultado.

example.js

```
const mentors = ["Daniel ", "irina ", "
Gordon", "ashleigh "];

function tidy(name) {...}

let tidy_mentors = [];

for (let i = 0; i < 100; i++) {
 const tidy_mentor = tidy(mentors[i]);
 tidy_mentors.push(tidy_mentor);
}

// Equivalent (tidy is passed as parameter)
console.log(mentors.map(tidy)); // [ 'daniel',
'irina', 'gordon', 'ashleigh' ]
console.log(mentors); // ["Daniel ", "irina ",
" Gordon", "ashleigh "]
```

Función Callback

Una función se llama callback cuando se pasa como parámetro a otra función

example.js

```
const mentors = ["Daniel ", "irina ", "
Gordon", "ashleigh "];

function tidy(name) {...}

let tidy_mentors = [];

for (let i = 0; i < 100; i++) {
 const tidy_mentor = tidy(mentors[i]);
 tidy_mentors.push(tidy_mentor);
}

// Equivalent (tidy is passed as parameter)
console.log(mentors.map(tidy));
console.log(mentors);
```

Función Callback

Una función se llama callback cuando se pasa como parámetro a otra función

example.js

```
function tidy(name) {
  // trim() method removes blank spaces at the beginning and at the end of a string
  return name.trim().toLowerCase();
}

console.log(mentors.map(tidy));

// We can pass the definition of the function directly as a parameter
console.log(mentors.map(function tidy(name) {
  return name.trim().toLowerCase();
}));
```

Función Anónima

Una función sin nombre

example.js

```
function tidy(name) {
  // trim() method removes blank spaces at the beginning and at the end of a string
  return name.trim().toLowerCase();
}
console.log(mentors.map(tidy));

// We can pass the definition of the function directly
console.log(mentors.map(function tidy(name) {
  return name.trim().toLowerCase();
}));
```

Funciones Flecha
(Arrow functions)

- Es una forma compacta de definir funciones
- Permite definir una función sin usar la palabra **function**

example.js

```
function tidy(name) {
  // trim() method removes blank spaces at the
beginning and at the end of a string
  return name.trim().toLowerCase();
}

console.log(mentors.map(tidy));

console.log(mentors.map(function (name) {
  return name.trim().toLowerCase();
}));

console.log(mentors.map((name) => {
  return name.trim().toLowerCase();
}));
```

Funciones Flecha
(Arrow functions)

- Si la función tiene una sola instrucción, podemos quitar las llaves (`{}`) de la definición y el resultado de la instrucción será retornado sin necesidad de escribir `return` (return implícito)

example.js

```
function tidy(name) {
  // trim() method removes blank spaces at the beginning and at the end of a string
  return name.trim().toLowerCase();
}

console.log(mentors.map(tidy));

console.log(mentors.map((name) => {
  return name.trim().toLowerCase();
}));

console.log(mentors.map((name) =>
name.trim().toLowerCase()));
```

Ejercicios

Ejercicio C (10 minutos)

1. Dada la función de la derecha, pasa una función *callback* como parámetro que modifique el arreglo `abracaArray` tal que:
- Transforme a mayúscula todas las letras de los elementos del arreglo

```
function abracaFunction(yourFunc) {
  console.log(
    "I am abracaFunction! Watch as I mutate an array of strings to your heart's content!"
  );
   const abracaArray = [
    "James",
    "Elamin",
    "Ismael",
    "Sanyia",
    "Chris",
    "Antigoni",
  ];
   const abracaOutput = yourFunc(abracaArray);
   return abracaOutput;
}
```

Ejercicios

Ejercicio D

Modifica tu función callback del ejercicio anterior para que:

- Ordene el arreglo `abracaArray` en orden alfabético

```
function abracaFunction(yourFunc) {
  console.log(
    "I am abracaFunction! Watch as I mutate an array of strings to your heart's content!"
  );
   const abracaArray = [
    "James",
    "Elamin",
    "Ismael",
    "Sanyia",
    "Chris",
    "Antigoni",
  ];
   const abracaOutput = yourFunc(abracaArray);
   return abracaOutput;
}
```

Métodos de Arreglos

.forEach(funcion):

- Permite aplicar una función a cada elemento del arreglo
- Similar a `.map()` excepto que **no** retorna un arreglo nuevo
- A diferencia de .map() la función callback podría modificar el arreglo (**efecto-secundario/side-effect**)

example.js

```
const names = ["Daniel", "mozafar", "irina"];

names.forEach((name) => {
  console.log(name);
});

// Output

Daniel
mozafar
irina
```

Métodos de Arreglos

.forEach(funcion):

- Permite aplicar una función a cada elemento del arreglo
- Similar a `.map()` excepto que **no** retorna un arreglo nuevo
- A diferencia de .map() la función callback podría modificar el arreglo (**efecto-secundario/side-effect**)

example.js

```
const names = ["Daniel", "mozafar", "irina"];

names.forEach((name, index) => {
  console.log(index + ": " + name);
});

// Output

0: Daniel
1: mozafar
2: irina
```

Efecto Secundario (Side Effect)

Idealmente, las funciones deberían recibir parámetros de entrada y retornar un valor sin tener efectos secundarios (*funciones puras*).

Función Pura

Una función pura no debe:

- Acceder a ninguna variable externa que no sea pasada como parámetro
- Cambiar variables definidas fuera de la función
- Interactuar con nada fuera de la función (ej. Imprimir un mensaje en la consola, mostrar un mensaje un sitio web, guardar data en disco)

Ejercicios

Ejercicio E (10 minutos)

- Crear una función que tome como parámetro un año de nacimiento `birthYear`, y retorne la edad en años
- Dado un arreglo con los años de nacimiento de 7 personas `[1964, 2008, 1999, 2005, 1978, 1985, 1919]`, crea otro arreglo que contenga las edades de esas personas.
- Imprimir el arreglo resultando

Ejercicios

Ejercicio F (5 minutos)

La edad mínima de manejo en el Reino Unido es de 17 años.

- Escribe otra función que reciba el año de nacimiento `birthYear` y retorne el string Los `nacidos en el año {birthYear} pueden manejar` o `Los nacidos en el año {birthYear} pueden manejar en {x} años`
- Usa el arreglo de los años de nacimiento, `[ 1964, 2008, 1999, 2005, 1978, 1985, 1919 ]`, para obtener un arreglo de strings que indiquen si estas personas pueden manejar
- Imprime el arreglo resultante

Métodos de arreglos

`.filter(function)`:

- Crea un **nuevo** arreglo con todos los elementos del arreglo original que cumplan la condición, es decir, al aplicar la función dada como parámetro el valor de retorno es `true`

example.js

```
const testScores = [90, 50, 100, 66, 25, 80,
81];

function isHighScore(score) {
return score > 80;
}

const highTestScores =
testScores.filter(isHighScore);

console.log(highTestScores); // [90, 100, 81]
```

Ejercicios

Ejercicio G (10 mins)

Crea una función que:

- Reciba como parámetro un arreglo con años de nacimiento `birthYears`
- Imprima el mensaje `Estos son los años de nacimiento de las personas que pueden manejar: <años de nacimiento filtrados>`
- Retorne un arreglo con los años de nacimientos de las personas que pueden conducir

Métodos de arreglos

.find(funcion):

- Returns the **first element** in the provided array that satisfies the condition

example.js

```
const names = ["Daniel", "James", "Irina",
"Mozafar", "Ashleigh"];

function isLongName(name) {
 return name.length > 6;
}

const longName = names.find(isLongName);

console.log(longName); // Mozafar
```

Ejercicios

Ejercicio H (10 mins)

Crea una función tal que:

- Reciba un arreglo de nombres y nombre como parámetro
- Revise si el nombre está en el arreglo
- Si está en el arreglo, que retorne `Found me!`; sino, retorne `Haven't found me :(`

EXERCISES

Ejercicio I (15 minutos)

Crea una función que acepte un arreglo de strings "sucios". Ejemplo a la derecha ->

Esta función debe:

- Quitar todos los elementos que no sean strings
- Cambiar a mayúscula todos los string y agregar un signo de exclamación al final

En el ejemplo, el arreglo resultante contiene 2x `ELAMIN!`, 1x `SANYIA!`, 2x `ISMAEL!` and 1x `JAMES!`.

```
[100, "iSMael", 55, 45, "sANyiA", 66,
"1JaMEs", "eLAmIn", 23, "IsMael", 67, 19,
"ElaMIN"]
```

Concatenación de Métodos

```
function formatName(name) {
 return name.split("")[0].toUpperCase() +
name.slice(1);
}

function log(name, index) {
 console.log(index + ": " + name);
}

const namesFormatted = names.map(formatName);
namesFormatted.forEach(log);
```

```
names.map(formatName).forEach(log);
```

Concatenación de Métodos

```
function formatName(name) {
 return name.split("")[0].toUpperCase() +
name.slice(1);
}

function log(name, index) {
 console.log(index + ": " + name);
}

const namesFormatted = names.map(formatName);
namesFormatted.forEach(log);
```

```
names.map(formatName).forEach(log);


names.forEach(log).map(formatName); // ERROR
```