UTN.BA

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Introducción

En el presente documento se verán algunos conceptos básicos de programación en JavaScript como el manejo de las operaciones que van a demorar (asincronicas), la nueva función arrow, y funciones para el manejo de vectores.

Vectores

La definición de vectores se realiza de la siguiente forma

Ejemplo

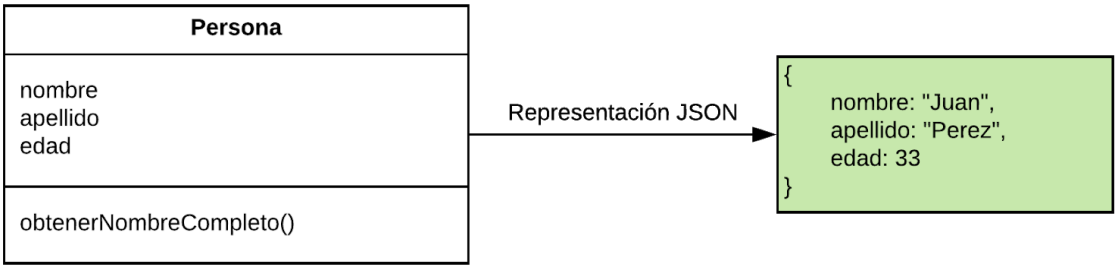
```
var vectorEdades = [10,20,18,33,84];
```

Los índices de los vectores comienzan en 0, por lo que el primer elemento (valor 10) tiene como índice 0 y para acceder al mismo debemos usar vectorEdades[0].

De esta misma forma, el último elemento del vector tiene la posición 4, y para acceder al mismo debemos utilizar vectorEdades[4];

JSON

Es una forma en la cual se representan los objetos en JavaScript. Si tomamos como ejemplo la siguiente clase Persona. Podemos ver la representación JSON de los datos de la persona.



En el JSON solo se representan las propiedades (variables/valores) del objeto, no los métodos. El JSON de un objeto comienza con { y finaliza con }, y todas las propiedades siguen el formato clave: valor. Si el valor es una cadena, debe ir con comillas. Para los valores booleanos, y numéricos, no se utilizan comillas.

La representación de un vector de Personas sería la siguiente

```
[
    {
        nombre: "Juan",
        apellido: "Perez",
        edad: 33
    },
    {
        nombre: "Maria",
        apellido: "Gonzalez",
        edad: 28
    }
]
```

Nótese que ahora el JSON comienza con [y finaliza con], indicando que el contenido es un vector.

Recorrer vectores

Disponemos de diferentes alternativas para recorrer vectores en JavaScript, una de las más utilizadas es forEach. La operación forEach es un método de todos los vectores, y por lo tanto podemos utilizarlo para cualquier vector que utilicemos. El método forEach recibe como parámetro una función que será llamada una vez por cada elemento del vector (pasándole como parámetro el valor del elemento).

Ejemplo

```
var edades = [10,33,12,74,22,84,44];
edades.forEach(function(unElemento) {
    console.log(unElemento);
});
```

La salida por pantalla del código será
10

33
12
74
22
84
44

Llamándose a la función function(unElemento) {} una vez por cada elemento que exista en el vector. Para el ejemplo mostrado, la función será llamada 7 veces.

Función arrow

En JavaScript es muy habitual trabajar con funciones, y hasta pasar una función como parámetro a otra función. Un ejemplo de ellos es la función setInterval, la cual ejecuta una función cada X milisegundos.

La función setInterval tiene el siguiente prototipo

setInterval(<función a llamar cada X milisegundos>, <X milisegundos);

Ejemplo

```
function mostrarFechaHora() {
      console.log(new Date());
}

setInterval(mostrarFechaHora, 1000);
```

Esto llamará a la función mostrarFechaHora cada 1 segundo (1000 milisegundos)

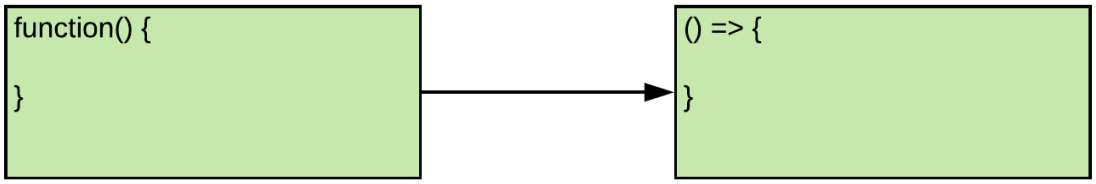
Ejemplo

Otra forma de escribir el mismo código es la siguiente:

```
setInterval(function() {
      console.log(new Date());
}, 1000);
```

En vez de definir una función mostarFechaHora, se incluye el código de la función directamente en el llamado de setInterval(). Al pasar la función, estamos realizando el mismo proceso que el definir una función con un nombre y pasarla como parámetro. Es importante destacar que al pasar una función directamente como parámetro, no se incluye nombre de la misma setInterval(function()...). Este tipo de funciones, son llamadas funciones anónimas.

A partir de la versión 6 de EcmaScript (definición de JavaScript) se incluye una nueva forma de pasar como parámetro estas funciones anónimas. La llamada arrow function.



Se sustituye function() {} por ()=>{}

Este tipo de funciones tiene algunas ventajas que iremos viendo a lo largo del curso.

Encontrar la posición de un elemento en el vector (findIndex)

Muchas veces deseamos encontrar la posición de un elemento dentro de un vector solo teniendo una propiedad del elemento que estamos buscando (no todo el elemento completo). Por ejemplo, si tenemos un listado de personas, puede ser que solo tengamos el ID de la persona que deseamos encontrar, pero no todos los datos de la misma. En estos casos, podemos utilizar el método findIndex() cuyo prototipo es el siguiente

```
<vector>.findIndex(item => {
      return <condición que se debe cumplir para haber encontrado el elemento>
})
```

Supongamos que tenemos el siguiente vector

```
var tareas = [
                {
                    "id": 1,
```

```
            "title": "delectus aut autem",
        },
        {
            "id": 2,
            "title": "quis ut nam facilis et officia qui",
        },
        {
            "id": 3,
            "title": "fugiat veniam minus",
        }
    ]
```

Y deseamos encontrar la posición de la tarea cuyo id es igual a 2. Utilizando findIndex llegaríamos al siguiente código

```
var posicion = respuesta.findIndex(item => {
    return item.id==2
})
console.log(posicion);
```

Para este caso, posición tiene el valor 1, ya que es el segundo elemento del vector.

Eliminar elementos del vector (splice)

Cuando deseamos reducir un vector a menos cantidad de elementos, por ejemplo, si tenemos 10 elementos, pero sólo deseamos quedarnos con los primeros 5 elementos, podemos usar el método splice. El cual tiene el siguiente prototipo

```
<vector>.splice(<desde>);
<vector>.splice(<desde>, <cantidad>);
```

Para el primer caso, splice elimina del vector los elementos desde la posición indicada <desde> hasta la finalización del vector.

Para el segundo caso, splice elimina los elementos desde la posición <desde> y desde esa posición eliminará <cantidad> elementos.

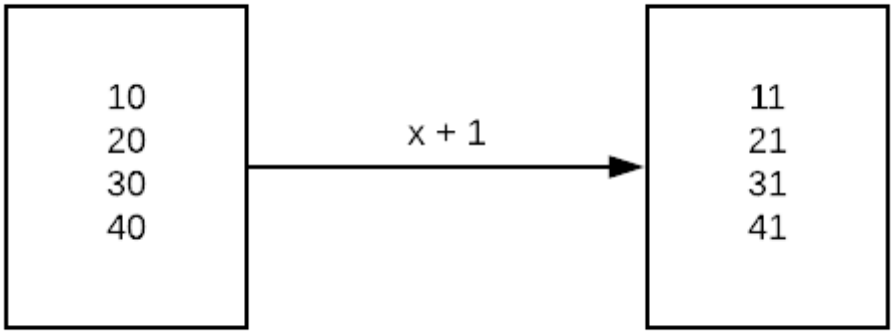
Adicionalmente el método splice puede ser utilizado para agregar un elemento al vector, en una posición determinada. Para ello, utilizamos la siguiente sintaxis

```
<vector>.splice(<posición>, 0, <nuevo elemento>);
```

En la posición <posición> se agrega <nuevo elemento>

Convertir un vector (map)

Si deseamos convertir los elementos de un vector, aplicarle alguna transformación, para ello disponemos del método **map**. El cual permite convertir una estructura A al formato de la estructura B. Si queremos sumar 1 a todos los valores que tenemos en un vector origen, podemos realizar una función map que realice la transformación.



Ejemplo

```
var edades = [10,33,12,74,22,84,44];
edades = edades.map(unItem => {
```

```
	return unItem+1;
})

console.log(edades);
```

La salida será

```
[ 11, 34, 13, 75, 23, 85, 45 ]
```

La función map, es especialmente útil cuando deseamos modificar el contenido de un vector JSON de objetos (JSON) a otro formato. Si deseamos convertir el vector de personas al formato {nombre_completo: ‘’, edad: 0} debemos aplicarle una transformación al vector original

Ejemplo

```
var personas = [
	{ nombre: 'Juan', apellido: 'Perez', edad: 40},
	{ nombre: 'Maria', apellido: 'Gonzalez', edad: 33 }
]

personas = personas.map(unItem => {
	return {
		nombre_completo: unItem.nombre + ' '+unItem.apellido,
		edad: unItem.edad

	}

})

console.log(personas);
```

Para este caso, la salida por pantalla será

```
[
	{ nombre_completo: 'Juan Perez', edad: 40 },
```

```
      { nombre_completo: 'Maria Gonzalez', edad: 33 }
]
```

Operaciones asincrónicas

Las operaciones asincrónicas son aquellas que al llamarlas, la respuesta demora y/o no puede ser procesada en el momento. Las operaciones más comunes que son tratadas de esta forma son:

- Operaciones sobre archivos (solo para JavaScript del lado del servidor)
- Operaciones sobre bases de datos (solo para JavaScript del lado del servidor)
- Conexión y consulta a servidores externos

Las operaciones asincrónicas se pueden tratar de diferentes formas, en el presente documento abordaremos la estructura de **async/await** que veremos a continuación.

Supongamos que tenemos una función que realiza una consulta a un servidor externo, esta operación es del tipo asincrónica, ya que el servidor puede tardar en responder (JavaScript obliga que todas estas operaciones sean asincrónicas), y debemos mostrar la respuesta del servidor por consola.

Ejemplo

```
function async realizarPeticionAServidorExterno() {
    var respuesta = await http.get('http://unServer.com');
    return respuesta;
}
```

La variable respuesta, no será asignada hasta que el servidor no retorne la respuesta. Para ello, la palabra **await**, hace que no continúe la ejecución del programa hasta no recibir la respuesta del servidor.

Al utilizar **await** dentro de una función, debemos "marcar" la misma como una función asincrónica, por ello es que incorporamos, en la definición de la función, antes del nombre la palabra **async**.