Resumen de JavaScript. Curso 22-23

Introducción

Tipos de datos

Los tipos de datos en JavaScript son dinámicos, es decir, no es necesario declarar el tipo de dato que vamos a utilizar, ya que el propio lenguaje lo infiere.

```
let nombre = "Pepe"; // String
let edad = 34; // Number
let isAlumno = true; // Boolean
let aDatos = [1, 2, 3]; // Array
let oDatos = {nombre: "Pepe", edad: 34}; // Object
// null
// undefined
// Funcion
var suma = function(a, b) {
    return a + b;
}
```

Variables y constantes

En JavaScript tenemos dos tipos de variables, las variables y las constantes. Las variables son aquellas que pueden cambiar de valor, mientras que las constantes no pueden cambiar de valor.

- const: las variables constantes no pueden ser modificadas (no tienen set)
- let: pueden ser modificadas (tienen set)
- var: no tienen scope (¡No usar nunca!)

```
const nombre = 'Pepe'
const apellido = 'Perez'
let edad = 45
// nombre = 'Peter' // error porque nombre es una constante
edad = 46 // ok porque edad es una variable
```

Template literals

Template literals son una forma de escribir strings con interpolación de variables. Dentro de la interpolación se pueden utilizar llaves para llamar a funciones.

```
const nombre = "Juan";
const apellido = "Perez";
const nombreCompleto = `${nombre} ${apellido}`;
```

```
console.log(nombreCompleto)
console.log(`Resultado: ${1 + 1}`)
```

Operadores ternarios y operadores condicionales

Ternarios son una forma de escribir condicionales en una sola línea.

```
const edad = 18
const edadMinima = 18
const esMayorDeEdad = edad >= edadMinima ? 'Es mayor de edad' : 'Es menor de edad'
console.log(esMayorDeEdad)
```

Igualdad o Identidad

En JavaScript tenemos dos tipos de igualdad, la igualdad estricta y la igualdad débil. La igualdad estricta compara el valor y el tipo de dato, mientras que la igualdad débil solo compara el valor.

```
console.log(1 == '1') // true
console.log(1 === '1') // false
```

Salida de datos

Para mostrar datos por consola podemos usar `console.log()` o similares.

```
console.log('Hola mundo')
console.info('Hola mundo')
console.warn('Hola mundo')
console.error('Hola mundo')
console.table([{nombre: 'Pepe', edad: 34}, {nombre: 'Juan', edad: 23}])
```

Programación estructurada

JavaScript es un lenguaje de programación estructurada, es decir, que permite la ejecución secuencial de instrucciones, la selección condicional y la repetición.

```
// Secuencial
let a = 1
let b = 2
let c = a + b
console.log(c)

// Condicional
if (a > b) {
    console.log('a es mayor que b')
} else {
    console.log('b es mayor que a')
}

// switch
let i = 2
```

```
switch (i) {
    case 0:
        console.log('i es 0')
        break;
    case 1:
        console.log('i es 1')
        break;
    default:
        console.log('i es mayor que 1')
        break;
}

// Bucle for
for (let i = 0; i < 10; i++) {
    console.log(i)
}

// Bucle while
let i = 0
while (i < 10) {
    console.log(i)
    i++
}

// Bucle do while
let i = 0
do {
    console.log(i)
    i++
} while (i < 10)
```

Arrays

Los arrays son una forma de definir una lista de elementos cuyo acceso queda referenciado por un índice.

```
const numeros = [1, 2, 3, 4, 5] // o new Array(1, 2, 3, 4, 5)
numeros.push(6)
console.log(numeros)
console.log(numeros[0])
// eliminamos
numeros.pop()
console.log(numeros)
// añadimos en una posicion concreta
numeros.splice(2, 0, 3.5)
console.log(numeros)
// eliminamos en una posicion concreta
numeros.splice(2, 1)

// Operaciones con colecciones
// for
for (let i = 0; i < numeros.length; i++) {
  console.log(numeros[i])
}
// for in
for (const key in numeros) {
  console.log(numeros[key])
```

```
}
```

Programación Funcional

La programación funcional es un paradigma de programación que nos permite trabajar con funciones de primera clase, es decir, que podemos pasar funciones como parámetros de otras funciones, devolver funciones como resultado de otras funciones, etc.

Funciones

Las funciones nos permiten definir fragmentos de código que podemos reutilizar.

En JS las funciones son ciudadanos de primera clase, es decir, **son un tipo**.

Las funciones flecha (arrow functions) Una expresión de función flecha es una alternativa compacta a una expresión de función tradicional. No vinculan su this con el del contexto en el que se invocan, , el objeto al que hace referencia el this en una función flecha siempre va a ser el mismo independientemente del lugar desde el que se invoque.

```
// Funciones
function saludar(nombre) {
  return `Hola ${nombre}`
}
console.log(saludar('Pepe'))

// funciones flecha (=>)
const saludar2 = (nombre) => `Hola ${nombre}`
console.log(saludar2('Pepe'))
```

Parametros por defecto

En JavaScript podemos definir valores por defecto para los parámetros de las funciones.

```
function saludar(nombre = 'Pepe') {
  return `Hola ${nombre}`
}

console.log(saludar()) // Hola Pepe
console.log(saludar('Juan')) // Hola Juan
```

Parámetros variables rest

En JavaScript podemos definir parámetros rest (número variable) para las funciones. Los parámetros rest nos permiten pasar un conjunto de parámetros y recuperarlos como una lista.

```
function sumar(...numeros) {
  let resultado = 0
  for (let i = 0; i < numeros.length; i++) {
    resultado += numeros[i]
  }
  return resultado
}

console.log(sumar(1, 2, 3, 4, 5)) // 15
```

Parámetros variables spread

En JavaScript podemos definir parámetros spread (número variable) para las funciones. Los parámetros spread nos permiten pasar una lista de parámetros a la función y recuperarlos en variables separadas.

```
function sumar(a, b, c, d, e) {
  return a + b + c + d + e
}

const numeros = [1, 2, 3, 4, 5]
console.log(sumar(...numeros)) // 15
```

API Funcional para el manejo de colecciones

Las funciones de orden superior nos permiten trabajar con colecciones de datos de forma más sencilla.

```
// foreach - recorre el array y ejecuta una función por cada elemento
numeros.forEach(numero => console.log(numero))
// find - busca el primer elemento que cumpla la condicion
const numero = numeros.find(numero => numero === 3)
console.log(numero)
// Copiar un array
const numeros2 = [...numeros]
// map - transforma el array
const numeros3 = numeros.map(numero => numero * 2)
// filter - filtra el array
const numeros4 = numeros.filter(numero => numero > 2)
// reduce - reduce el array a un valor
const numeros5 = numeros.reduce((acumulador, numero) => acumulador + numero,
0)
// some - busca algun elemento que cumpla la condicion
const numeros6 = numeros.some(numero => numero > 2)
// every - busca que todos los elementos cumplan la condicion
const numeros7 = numeros.every(numero => numero > 2)
// findIndex - busca el indice del primer elemento que cumpla la condicion
const numeros8 = numeros.findIndex(numero => numero > 2)
```

Objetos

Los objetos son una forma de definir una colección de propiedades y métodos.

Objetos literales

Los objetos literales son una forma de definir un objeto en base a una expresión. Los objetos se pasan por referencia y para clonarlos

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  direccion: {
    ciudad: 'Madrid',
    pais: 'España'
  }
}
console.log(persona)
console.log(persona.direccion.ciudad)

// Clonando con spread, para copiar un objeto, ¡pero no clona en profundidad!
// Intenta cambiar las cosas y verás que no funciona
let persona2 = { ...persona }
persona2.nombre = 'Ana'
console.log(persona2)
console.log(persona)

// Clonando objetos con profundidad
persona2 = structuredClone(persona)
persona2.nombre = 'Alberto'
console.log(persona2)
console.log(persona)
```

Objetos con constructor

Los objetos con constructor son una forma de definir un objeto en base a una función constructora.

```
function Persona(nombre, apellido, edad) {
  this.nombre = nombre
  this.apellido = apellido
  this.edad = edad
}
const persona = new Persona('Pepe', 'Perez', 45)
console.log(persona)
```

Objetos con clases

Las clases son una forma de definir un objeto en base a una clase.

```
class Persona {
```

```
  constructor(nombre, apellido, edad) {
    this.nombre = nombre
    this.apellido = apellido
    this.edad = edad
  }
}

const persona = new Persona('Pepe', 'Perez', 45)
console.log(persona)
```

Métodos

Los métodos son funciones que se definen dentro de un objeto.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  saludar: function () {
    console.log(`Hola soy ${this.nombre}`)
  }
}
persona.saludar()
```

Herencia

La herencia es una forma de reutilizar código.

```
class Persona {
  constructor(nombre, apellido, edad) {
    this.nombre = nombre
    this.apellido = apellido
    this.edad = edad
  }
  saludar() {
    console.log(`Hola soy ${this.nombre}`)
  }
}

class Alumno extends Persona {
  constructor(nombre, apellido, edad, curso) {
    super(nombre, apellido, edad)
    this.curso = curso
  }
  saludar() {
    super.saludar()
    console.log(`Estudio ${this.curso}`)
  }
}

const alumno = new Alumno('Pepe', 'Perez', 45, 'Angular')
alumno.saludar()
```

Getters y Setters

Los getters y setters son funciones que se definen dentro de un objeto para obtener y establecer valores.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  get nombreCompleto() {
    return `${this.nombre} ${this.apellido}`
  },
  set nombreCompleto(nombreCompleto) {
    const partes = nombreCompleto.split(' ')
    this.nombre = partes[0]
    this.apellido = partes[1]
  }
}

console.log(persona.nombreCompleto)
persona.nombreCompleto = 'Juan Lopez'
console.log(persona.nombre)
console.log(persona.apellido)
```

Null checking

Null checking es una forma de verificar si una variable es nula y es muy util para los campos de objetos.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
}
console.log(persona.power ? persona.power : "No tiene")
```

API Funcional para el manejo de objetos

Las funciones de orden superior nos permiten trabajar con objetos de forma más sencilla.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  direccion: {
    ciudad: 'Madrid',
    pais: 'España'
  }
}

// Object.keys - devuelve las claves de un objeto
console.log(Object.keys(persona))
// Object.values - devuelve los valores de un objeto
console.log(Object.values(persona))
// Object.entries - devuelve las claves y valores de un objeto
console.log(Object.entries(persona))
// Object.assign - asigna propiedades de un objeto a otro
```

```
const persona2 = Object.assign({}, persona)
persona2.nombre = 'Ana'
console.log(persona2)
console.log(persona)
// Object.freeze - congela un objeto
const persona3 = Object.freeze(persona)
persona3.nombre = 'Alberto'
console.log(persona3)
console.log(persona)
```

Desestructuración

La desestructuración es una forma de extraer valores de un objeto o array.

Desestructurar arrays

Desestructurar arrays es una forma de extraer valores de un array.

```
const numeros = [1, 2, 3, 4, 5]
const [numero1, numero2, ...resto] = numeros
console.log(numero1)
console.log(numero2)
console.log(resto)
```

Desestructurar objetos

Desestructurar objetos es una forma de extraer valores de un objeto.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  direccion: {
    ciudad: 'Madrid',
    pais: 'España'
  }
}
const { nombre, apellido, edad, telefono= 'No tiene' } = persona
console.log(nombre)
console.log(apellido)
```

Rest y spread

Rest y spread es una forma de extraer valores de un objeto o array.

```
// Arrays
const numeros = [1, 2, 3, 4, 5]
const [numero1, numero2, ...resto] = numeros
console.log(numero1)
console.log(numero2)
console.log(resto)
```

```
// Objetos
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  direccion: {
    ciudad: 'Madrid',
    pais: 'España'
  }
}
const { nombre, apellido, ...resto } = persona
console.log(nombre)
console.log(apellido)
console.log(resto)
```

Importaciones y exportaciones

Importaciones y exportaciones son una forma de importar y exportar funciones, clases, variables, etc. Podemos exportar varias funciones o clases en un solo archivo con export y usar import para importar lo que queremos. Podemos tener una exportación por defecto con export default.

```
// Exportar
const nombre = 'Pepe'
const apellido = 'Perez'
const lista = [1, 2, 3, 4, 5]
export { nombre, apellido }
export default lista

// Importaciones
import { nombre, apellido } from './persona.js'
import lista from './persona.js'
console.log(nombre)
console.log(apellido)
console.log(lista)
```

Asincronía

La asincronía es una forma de ejecutar código de forma asíncrona.

Promesas

Las promesas son una forma de definir una tarea que se va a realizar en el futuro (y nos sirven para gestionar código asíncrono).

```
const promesa = new Promise((resolve, reject) => {
  // Tarea asíncrona
  if (true) {
    resolve('Tarea finalizada')
  } else {
    reject('Tarea fallida')
  }
```

```
})
promesa.then(resultado => console.log(resultado))
promesa.catch(error => console.log(error))
```

Async / Await

Async / Await es una forma de escribir código asíncrono en una función. Es azúcar sintáctico para manejar promesas.

```
async function obtenerPersonajes() {
  try {
    const response = await
axios.get('https://rickandmortyapi.com/api/character/')
    console.log('Mis datos de retorno')
    console.log(response.data)
  } catch (error) {
    console.log(error)
  }
}
obtenerPersonajes()
```

Ejemplo de asincronia

```
const miPromesa = () => {
    return new Promise(( resolve, reject )=> {
        setTimeout(() => {
            // resolve('Tenemos un valor en la promesa')
            reject('REJECT en miPromesa')
        }, 1000);
    })
}

const medirTiempoAsync = async() => {

    try {
        console.log('Inicio')

        const respuesta = await miPromesa()
        console.log(respuesta)

        console.log('Fin')

        return 'fin de medir tiempo async'

    } catch (error) {
        // return 'catch en medirTiempoAsync'
        throw 'Error en medirTiempoAsync'
    }

}


medirTiempoAsync()
    .then( valor => console.log( 'THEN Exitoso:', valor  ) )
    .catch( err => console.log( 'Error:', err ))
```

API REST

API REST es una forma de comunicar dos aplicaciones a través de HTTP.

JSON

JSON (acrónimo de JavaScript Object Notation, 'notación de objeto de JavaScript') es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript,

```
{
  "nombre": "Pepe",
  "apellido": "Perez",
  "edad": 45,
  "direccion": {
    "ciudad": "Madrid",
    "pais": "España"
  }
}
```

Podemos usar JSON para enviar y recibir datos a través de una API REST y pasar objetos de JavaScript a JSON y viceversa.

```
const persona = {
  nombre: 'Pepe',
  apellido: 'Perez',
  edad: 45,
  direccion: {
    ciudad: 'Madrid',
    pais: 'España'
  }
}

// Pasar un objeto de JavaScript a JSON
const personaJSON = JSON.stringify(persona)
console.log(personaJSON)

// Pasar un JSON a un objeto de JavaScript
const personaObjeto = JSON.parse(personaJSON)
console.log(personaObjeto)
```

Métodos

Las API REST tienen métodos para realizar acciones en los recursos.

- GET: Obtener un recurso.
- POST: Crear un recurso.
- PUT: Actualizar un recurso.
- PATCH: Actualizar un recurso (solo unos campos)
- DELETE: Eliminar un recurso.

Códigos de estado

Las API REST devuelven códigos de estado para indicar el estado de la petición.

- 200: OK
- 201: Created
- 204: No Content
- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden
- 404: Not Found
- 500: Internal Server Error

Api Fetch

Api fetch es una librería que nos permite hacer peticiones a una api web.

```
fetch('https://rickandmortyapi.com/api/character/')
    // https://developer.mozilla.org/es/docs/Web/API/Response
    .then((response) => response.json()) // Obtenemos la respuesta, pero esto
es una promesa en sí, por eso hay otro then
    .then((data) => console.log(data)) // finalmente obtenemos los datos de
la promesa anterior
    .catch((error) => console.log(error))
```

Axios

Axios es una librería que nos permite hacer peticiones a una api web. Se debe instalar el paquete axios en el proyecto.

```
import axios from 'axios'
axios.get('https://rickandmortyapi.com/api/character/')
    .then((response) => console.log(response.data))
    .catch((error) => console.log(error))
```

API REST con axios

```
import axios from 'axios'

// GET
axios.get('https://rickandmortyapi.com/api/character/')
    .then((response) => console.log(response.data))
    .catch((error) => console.log(error))

// POST
axios.post('https://rickandmortyapi.com/api/character/', {
    name: 'Rick',
    status: 'Alive',
    species: 'Human'
```

```
})
    .then((response) => console.log(response.data))
    .catch((error) => console.log(error))

    // PUT
axios.put('https://rickandmortyapi.com/api/character/1', {
    name: 'Rick',
    status: 'Alive',
    species: 'Human'
})
    .then((response) => console.log(response.data))
    .catch((error) => console.log(error))

    // DELETE
axios.delete('https://rickandmortyapi.com/api/character/1')
    .then((response) => console.log(response.data))
    .catch((error) => console.log(error))
```

Codificado con 💔 por José Luis González Sánchez