

Introducción a ECMA Script (Javascript)

Técnicas de Visualización de la Información

Máster en Ingeniería del Software: Cloud, Datos y Gestión TI

Departamento de Lenguajes y Sistemas Informáticos



Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

ECMAScript es lenguaje de programación publicado por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular lenguaje JavaScript propuesto por Netscape. Aceptado como estándar ISO/IEC 22275-2018, su última versión es ECMAScript 2022 (Javascript ES13):

- El lenguaje original era ejecutado por el navegador al cargar una página web, que contiene HTML, CSS y Javascript:
 - Permitía añadir cierto dinamismo a las páginas web y acceder al DOM (Document Object Model) que es la estructura interna en memoria de la página web cargada.
 - A lo largo de los años, surgen diferencias en las versiones que ejecutan cada uno de los navegadores, aunque por regla general eran compatibles.
- Javascript se considera un dialecto de ECMAScript, es un lenguaje débilmente tipado y dinámico, clave en el desarrollo de frameworks de frontends / backends que lo utilizan como su lenguaje de referencia.

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

```
const greetingMessage = 'Hello world'  
console.log(greetingMessage)
```

<https://github.com/jorgeBIGS/TVI-D3-exercices.git>

Utiliza la plantilla para crear tu propio repositorio y clónalo.

Correr `npm install` para importar librerías.

Cada ejercicio puede ser ejecutado con:

```
node nombreFichero.js
```

Las variables son contenedores de valores, que pueden ser datos y también funciones:

- **const** se asigna un valor en la declaración que ya no podrá cambiarse. Se recomienda usar este tipo de variables.
- **let** se utiliza cuando sabemos que los valores asignados van a cambiar.
- **var** esta opción quedó obsoleta y no se recomienda su uso.

```
let x, y      // Statement 1
x = 5         // Statement 2
y = 6         // Statement 3
let z = x + y // Statement 4
```

Off-topic:

No es necesario usar los `;` para separar las sentencias del código
Uso habitual de `//` para poner comentarios de una línea

Variables: Identificadores

El nombre de una variable se denomina **identificador** y debe ser único.

Las reglas generales son para los identificadores son:

- Se distinguen mayúsculas y minúsculas.
- No se pueden utilizar palabras reservadas del lenguaje.
- Cadenas que contienen letras, dígitos, además de `_` y `$`
- Se recomienda:
 - Empezar con una letra.
 - Usar *Lower camel case* para nombres de variables y funciones (por ejemplo, `nombreUsuario`)
 - Usar *Camel case superior* para nombres de tipos y clases (por ejemplo: `RegExp`)

Javascript es un lenguaje débilmente tipado y dinámico, lo que significa que:

- A una variable le podríamos asignar sucesivamente valores correspondientes a diferentes tipos, incluso funciones.
- Hay una **conversión de tipos automática** para evaluar expresiones, aunque no siempre obtendremos el valor esperado y hay que tener cierta precaución.

Entre los tipos más frecuentes:

- Numéricos.
- Cadenas.
- Booleanos.
- Objetos {}.
- Arrays [].

```
const isEmpty = true
const message = 'Hello!'
const size = 3
const height = 1.75
```

```
// Object
const person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  }
}
```

```
// Array
const cars = [ 'Saab', 'Volvo', 'BMW' ]
```


Variables: Comparaciones

Javascript permite dos maneras de comparar:

- Una comparación doble `==` devuelve cierto cuando los valores son iguales:
 - Puede implicar una conversión implícita de ambos operandos.
- Una comparación triple `===` devuelve cierto cuando los valores son iguales y también sus tipos.

Off-topic:

Nótense los caracteres de escape para añadir una comilla simple en una cadena definida con comillas simples

```
const height = 1.75
// Comparison for equal values
if (height == 1.75) {
  console.log('height == 1.75 is true')
}
if (height == '1.75') {
  console.log('height == \'1.75\' is true')
}
// Comparison for equal values and types
if (height === 1.75) {
  console.log('height === 1.75 is true')
}
if (height === '1.75') {
  console.log('height === \'1.75\' is true')
} else {
  console.log('height === \'1.75\' is false')
}
```

Variables: Alcance (ámbito)

```
const x = 5      // Global scope

{
  const x = 6    // Block scope, not accesible from outside
  const y = 7
  console.log(x) // Prints 6
}

console.log(x)   // Prints 5
console.log(y)   // Error: Out-of-scope variable
```

Off-topic:

Nótese que bloques y objetos se definen con llaves {}

El alcance o ámbito de una variable (**scope**) define su visibilidad:

- Las variables globales son visibles en la extensión completa de un módulo:
 - Pueden ser exportadas.
- Las variables de bloque (definidas entre {}) sólo serán visible dentro del bloque donde se declaran:
 - Pueden re-declarar una variable de mayor alcance.
- Se recomienda que todas las variables sean declaradas al inicio del módulo o del bloque donde sean necesarias.

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Cadenas: Entrecomillados e Interpolación

Javascript permite valores de tipo cadena con comillas simples (preferible) y dobles, que pueden combinarse:

```
const str1 = 'My dog is "Haru"'
const str2 = "My dog is 'Haru'"
const str3 = "It's my dog!"
```

Las cadenas pueden concatenarse con el operador +

Hay un tipo especial de entrecomillado usando comillas simples de acento grave (`) que permite la **interpolación de cadenas** para incluir valores de variable:

```
const firstName = 'John'
const lastName = 'Doe'
// Using the concat operator +
console.log('Welcome ' + firstName + ', ' + lastName + '!')
// Using string interpolation (preferred)
console.log(`Welcome ${firstName}, ${lastName}!`)
```

Cadenas: Propiedades y Métodos

Hay propiedades y métodos de cadenas, aunque sólo vamos a destacar unos cuantos:

- `length`
- `indexOf()`
- `slice()`
- `charAt()` o `[]`
- `split()`
- `replace()`

```
const str = 'Apple,Banana,Kiwi'
const start = 6           // Starting from zero
const end = 12            // Non-inclusive end
console.log(str.length)   // Prints 17
console.log(str.indexOf('Banana')) // Prints 6
console.log(str.indexOf('Orange')) // Prints -1
console.log(str.slice(start, end)) // Prints Banana
console.log(str.slice(start))      // Prints Banana,Kiwi
console.log(str.charAt(6))         // Prints B
console.log(str[0])               // Prints A
console.log(str.split(','))        // Prints ['Apple', 'Banana', 'Kiwi']
console.log(str.replace('Kiwi','Orange')) // Prints Apple,Banana,Orange
```

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Los objetos contienen pares nombre: valor que se denominan **propiedades**, separados por comas:

- Una propiedad también puede tener asignada una función (**métodos**)
- Un objeto declarado como constante puede cambiar los valores asignados a sus propiedades.
- La variable que referencia al objeto es lo que no puede cambiar cuando está declarada con **const**.
- **this** es una palabra clave para hacer referencia al propio objeto dentro del código de un método definido en el objeto.

```
// Object
const person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  }
}

person.firstName = 'Paul'
console.log(person.fullName()) // Prints Paul Due
person = { firstName: 'Paul' } // Error: assignment to constant variable
```

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Un array es una lista de elementos almacenados en una variable:

- Un array declarado como constante puede cambiar los elementos que contiene.
- La variable que referencia al array es lo que no puede cambiar cuando está declarada con **const**.
- Los elementos se acceden mediante el operador `[]` que recibe un índice (el primer elemento es el 0).

```
// Array
const cars = [ 'Saab', 'Volvo', 'BMW' ]

const firstCarBrand = cars[0]
console.log(firstCarBrand) // Prints Saab

cars[0] = 'Mercedes' // Notice that cars were declared as const,
console.log(cars)    // but its elements can be changed!
cars = ['Toyota']    // Error: assignment to constant variable
```

Arrays: Propiedades y Métodos

```
// Array
const cars = [ 'Saab', 'Volvo', 'BMW' ]

console.log(cars.length) // Prints 3

cars.push('Honda')      // Appends Honda as last element
console.log(cars)        // Prints ['Saab','Volvo','BMW','Honda']

const poppedCar = cars.pop() // Removes last element (Honda)
console.log(`popped car ${poppedCar}`) // Prints Honda
console.log(cars)         // Prints ['Saab','Volvo','BMW']

cars.shift()            // Removes first element (Saab)
console.log(cars)        // Prints ['Volvo','BMW']

cars.unshift('Lexus')    // Adds Lexus as first element
console.log(cars)        // Prints ['Lexus','Volvo','BMW']
```

Javascript ofrece una serie de propiedades y métodos para manipular los arrays. Entre ellos:

- **length**
- **push()**
- **pop()**
- **shift()**
- **unshift()**

- splice()
- concat()
- slice()

Arrays: Iteradores (forEach y filter)

```
// Array
const cars = ['Volvo', 'Audi', 'Mercedes', 'Lexus', 'BMW', 'Seat']

cars.forEach(printElement) // Array iteration.
// The function will be called for each item of the array cars
function printElement (item) {
  console.log(item)
}

cars.forEach(item => console.log(item)) // Lambda function
// It is the equivalent inline arrow function to printElement

const longNamedBrands = cars.filter(brandName => brandName.length > 4)
console.log(longNamedBrands)
// The lambda function filters the cars to be printed
```

Javascript permite la iteración sobre los elementos de un array:

- **forEach()**
- **filter()**

Estos iteradores admiten como parámetro una **función lambda**, que se ejecuta sobre cada elemento del array.

Arrays: Iteradores (map y reduce)

Otros iteradores interesantes con arrays:

- `map()`
- `reduce()`

```
const array1 = [1, 2, 3, 4]

// total (the reduced or aggregated value so far)
// value (the value of the current element)
const reducer = (total, value) => total + value
console.log(array1.reduce(reducer)) // Prints 10
// Runs a function on each element to produce (reduce it to) a single value
// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5)) // Prints 15
// Reduce may accept an initial value (5)
// 5 + 1 + 2 + 3 + 4

const array2 = [1, 4, 9, 16]
const map1 = array2.map(x => x * 2)
// Creates a new array by computing a value for each element
// To do so, a function to map is passed
console.log(map1) // Prints [2,8,18,32]
```

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Estructuras: If / else if / else

```
const customer1 = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'customer1@customer.com',
  birthDate: new Date('9/15/1990'), // september 15th
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  },
  age: function () {
    const ageDifMs = Date.now() - this.birthDate.getTime()
    const ageDate = new Date(ageDifMs) // milliseconds from epoch
    return Math.abs(ageDate.getUTCFullYear() - 1970)
  }
}
```

Off-topic:

Nótese el objeto **Math** y uno de sus métodos: **abs()**

Off-topic:

Nótese el objeto **Date** y cómo se puede crear una variable fecha, obtener la fecha actual y otros métodos relacionados.

```
const age = customer1.age()
console.log(`${customer1.fullName()} is ${age} years old`)

if (age > 18) {
  console.log('Adult')
} else if (age > 13) {
  console.log('Teenager. Not a valid customer')
} else {
  console.log('Child. Not a valid customer')
}
```

Estructuras: If / else if / else

```
const customer1 = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'customer1@customer.com',
  birthDate: new Date('9/15/1990'), // september 15th
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  },
  age: function () {
    const ageDifMs = Date.now() - this.birthDate.getTime()
    const ageDate = new Date(ageDifMs) // milliseconds from epoch
    return Math.abs(ageDate.getUTCFullYear() - 1970)
  }
}
```

Off-topic:

Nótese que la expresividad de Javascript permite que la cantidad de código escrito sea mucho menor.

```
function isCustomerValidV1 (customer) {
  const age = customer.age()
  if (age > 18) {
    return true
  } else {
    return false
  }
}

console.log(isCustomerValidV1(customer1))

function isCustomerValidV2 (customer) {
  return customer.age() > 18
}

console.log(isCustomerValidV2(customer1))
```


Estructuras: Operador condicional ternario

```
const customer1 = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'customer1@customer.com',
  birthDate: new Date('9/15/1990'), // september 15th
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  },
  age: function () {
    const ageDifMs = Date.now() - this.birthDate.getTime()
    const ageDate = new Date(ageDifMs) // milliseconds from epoch
    return Math.abs(ageDate.getUTCFullYear() - 1970)
  }
}
```

Nótese que el operador ternario no necesariamente tiene que devolver el mismo tipo en el segundo y tercer operandos.

Esta expresión sería válida:

return customer ? customer.age() : false

```
// This is a conditional ternary operator
// Here it is used for argument checking
function isCustomerValidV3 (customer) {
  return customer ? customer.age() > 18 : false
}
```

```
console.log(isCustomerValidV3(customer1)) // Prints true
console.log(isCustomerValidV3())           // Prints false
```

Estructuras: Parámetro sin definir (undefined)

```
const customer4 = {  
  firstName: 'Alan',  
  lastName: 'Turing',  
  id: 5567,  
  fullName: function () {  
    return this.firstName + ' ' + this.lastName  
  },  
}
```

Off-topic:

El método **test()** comprueba si una variable satisface una condición de validez expresada como una **expresión regular**.

Una introducción a las expresiones puede encontrarse en la bibliografía reseñada al final de la presentación.

```
function isCustomerIdValidV1 (customer) {  
  const re = /^[0-9]{4}$/  
  // operator '?' to check undefined. Notice that is diferent to ternary operator  
  return customer?.id ? re.test(customer.id) : false  
}  
  
console.log(`${customer4.fullName()} id is ${customer4.id} and`  
  + ' is valid = ${isCustomerIdValidV1(customer4)}`)  
console.log(`${customer4.fullName()} id is ${customer4.id} and`  
  + ' is valid = ${isCustomerIdValidV1()}`)
```

Estructuras: Otras definiciones de Funciones

```
// Assign a variable to a function
const isValidCustomerV2 = function (customer) {
  const re = /^[0-9]{4}$/
  return customer?.id ? re.test(customer.id) : false
}
console.log('Using variable: ' + isValidCustomerV2(customer4))
```

Asignando una función
a una variable

Usando el operador flecha

```
// Using arrow functions
const isValidCustomerIdV3 = (customer) => {
  const re = /^[0-9]{4}$/
  return customer?.id ? re.test(customer.id) : false
}
console.log('Using arrow function: ' + isValidCustomerIdV3(customer4))
```

```
// Using arrow functions without parenthesis
// if there is only one parameter
const isValidCustomerIdV4 = customer => {
  const re = /^[0-9]{4}$/
  return customer?.id ? re.test(customer.id) : false
}
console.log('Using arrow function, no parenthesis: ' + isValidCustomerIdV4(customer4))
```

Usando el operador flecha
sin paréntesis

Una sola línea

```
// if there is only a return statement this syntax is valid:
const fullNameV2 = customer => customer.firstName + ' ' + customer.lastName

console.log('Using function that only includes a return statement: '
  + fullNameV2(customer4))
```

Estructuras: Bucles con For

```
// Loops and iterations
const customer1 = { }
const customer2 = { }
const customer3 = { }
const customers = [customer1, customer2, customer3]

// Classic
for (let index = 0, len = customers.length; index < len; ++index) {
  console.log(customers[index].fullName())
}

// Extended
for (const customer of customers) {
  console.log(customer.fullName())
}

// Functional
customers.forEach(customer => console.log(customer.fullName()))
```

Javascript permite los esquemas clásicos de iteración de arrays:

- El **for clásico** que usa una variable de índice (*index*) de manera que (1) se inicializa ésta y otras variables necesarias, (2) hay una condición de fin, y (3) el paso en cada iteración.
- El **for extendido** define un iterador (primer operando) sobre el array (segundo parámetro) denotado por la palabra clave **of**.
- El esquema **funcional** ya se ha presentado anteriormente, pasa como parámetro una *función lambda* cuyo parámetro es un elemento del array.

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Programación asíncrona: funciones callback

Una **función de *callback*** se pasa como parámetro a otra función, siendo llamada cuando ésta finaliza:

- En este caso, ***setTimeout()*** espera 3 segundos y cuando finaliza la espera, llama a ***greetCustomer(customer1)***.

```
const customer1 = {
  firstName: 'John',
  lastName: 'Doe',
  email: 'customer1@customer.com',
  birthDate: new Date('9/15/1990'), // september 15th
  fullName: function () {
    return this.firstName + ' ' + this.lastName
  },
  age: function () {
    const ageDifMs = Date.now() - this.birthDate.getTime()
    const ageDate = new Date(ageDifMs) // milliseconds from epoch
    return Math.abs(ageDate.getUTCFullYear() - 1970)
  }
}
```

```
function greetCustomer (customer) {
  console.log(`Welcome1 ${customer.fullName()}`)
}

console.log('Before timeout1')
// Params to be passed to the callback function:
// 1. callback function,
// 2. time in milliseconds,
// 3+. optional params to be passed to the callback f.
setTimeout(greetCustomer, 3000, customer1)
console.log('After timeout1')

console.log('Before timeout2')
// Passing an arrow anonymous function
setTimeout(customer => console.log(`Welcome2 ${customer.fullName()}`), 5000, customer1)
console.log('After timeout2')
```

Programación asíncrona: Reading I/O

Una función puede ser síncrona o asíncrona:

- Una función síncrona como ***readFileSync()*** **bloquea** la ejecución hasta que obtiene una respuesta.
- Una función asíncrona como ***readFile()*** **no bloquea el flujo de ejecución**, por lo que el mensaje se mostrará antes de obtener el resultado. Nótese que el segundo parámetro es una función de *callback* que recibe dos parámetros: si hay error y los datos. Esta función se ejecuta cuando ***readFile*** termine de leer los datos.

```
const fs = require('fs') // importing module

// Block I/O.
const data = fs.readFileSync('file.txt') // blocks here until file is read
console.log(data)
console.log('After blocking I/O. This will run AFTER reading file')

// Async I/O (non blocking)
fs.readFile('file.txt', (err, data) => {
  if (err) throw err
  console.log(data)
})
console.log('After async I/O. This will run BEFORE reading file')
```

Off-topic:

Nótese la importación del módulo **fs** (“*File System*”) para utilizar métodos relacionados con la entrada / salida de ficheros.

La variable fs contiene la referencia que se ha importado, de manera que los métodos se llaman a través de dicha variable.

Programación asíncrona: Promesas

Una promesa (1) se declara creando un objeto de tipo **Promise**, pasando como parámetro la función que ejecuta de manera asíncrona, (2) dicha función a su vez tiene dos parámetros:

- El primer parámetro, **resolve** se invoca en caso de éxito.
- El segundo, **reject** se invoca en caso contrario, lanzando un error.

Al ejecutar la promesa, se espera un resultado de manera asíncrona:

- En **then** se ejecuta una función que recibe como parámetro el resultado del **resolve**.
- En **catch** se captura el error que lanza **reject**.

```
// Promises
const promiseOfEvenInteger = new Promise((resolve, reject) => {
  const n = Math.floor(Math.random() * 10)
  console.log(n)
  if (n % 2 === 0) {
    resolve(`Even number found: ${n}`)
  } else {
    reject(new Error(`Odd number found: ${n}`))
  }
})

promiseOfEvenInteger
  .then(message => console.log('Then: ' + message))
  .catch(error => console.error('Catch: ' + error))
```


Programación asíncrona: Async/await functions

Una manera alternativa para invocar funciones asíncronas es mediante **async/await**:

- Si una función va a llamar otra función asíncrona, se precede su declaración con la palabra clave **async**.
- La propia llamada a la función asíncrona se precede con la palabra clave **await** en el contexto de un bloque **try-catch**.
- De esta manera, el flujo de ejecución no continúa hasta resolverse la promesa, imitando el estilo de programación síncrono.
 - Si la respuesta es exitosa (*resolve*), se recoge el resultado y se continúa la ejecución del bloque **try**.
 - Si la respuesta es errónea (*reject*), entonces se ejecuta el bloque **catch**.

```
// Promises
const promiseOfEvenInteger = new Promise()

// Async/await solution
async function noPromise () {
  // This code is easier to read.
  try {
    const message = await promiseOfEvenInteger
    console.log(message)
  } catch (err) {
    console.log('Catch: ' + err)
  }
}

noPromise()
```

Off-topic:

Recordamos el uso de las cláusulas *try-catch* en cualquier contexto en el que puedan resultar excepciones.

Programación asíncrona: Ejemplo con Axios

La librería Axios se utiliza para llamadas a servicios con API RESTful:

- Las funciones de la librería son asíncronas.
- En este caso, invocamos un método GET sobre dicha URL para obtener una lista de usuarios haciendo uso de la resolución de promesas con **then()** y **catch()**:
 - En caso de éxito, el resultado está en **response**
 - En caso de fallo, los detalles del error están en **error**

```
// Other example
const axios = require('axios')
const usersEndpointUrl = 'https://jsonplaceholder.typicode.com/users'

// First version with promises
axios.get(usersEndpointUrl)
  .then(response => {
    console.log(response) // Prints whole response object
  })
  .catch(error => {
    console.log(error)
  })
```

Programación asíncrona: Ejemplo con Axios

La librería Axios se utiliza para llamadas a API RESTful:

- En esta segunda versión, se invoca mediante una función **async/await**.

```
// Other example
const axios = require('axios')
const usersEndpointUrl = 'https://jsonplaceholder.typicode.com/users'

// async/await solution
async function processUsers (url) {
  try {
    const response = await axios.get(url)
    // Notice this is not a promise (no await needed)
    const usersJsonString = JSON.stringify(response.data)
    console.log(`Users: ${usersJsonString}`)
  } catch (error) {
    console.log(error)
  }
}

processUsers(usersEndpointUrl)
```

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Exportación de módulos: Objetos

```
// Loading from node modules folder by its identifier
const axios = require('axios')
const baseUrl = 'https://jsonplaceholder.typicode.com'
const usersEndpointPath = '/users'

// Common JS format is used in Node environments
// ('module.exports' to export, and 'require' to import)
module.exports = {
  getUsers: async () => {
    try {
      const response = await axios.get(`${baseUrl}${usersEndpointPath}`)
      return response.data
    } catch (error) {
      console.log(error)
    }
  },
  getUserById: async (id) => {
    try {
      const response = await axios.get(`${baseUrl}${usersEndpointPath}\\${id}`)
      return response.data
    } catch (error) {
      console.log(error)
    }
  }
}
```

En el entorno Node, se utiliza el formato CommonJS para declarar módulos.

En el ejemplo, se importa el módulo de Axios con **require** y se codifica un **objeto** que contiene las llamadas a una API, que exportamos mediante **module.export**.

Nótese que cada propiedad del objeto contiene una función asíncrona para realizar una llamada a la API.

Exportación de módulos: Funciones

```
// Loading from node modules folder by its identifier
const axios = require('axios')
const baseUrl = 'https://jsonplaceholder.typicode.com'
const usersEndpointPath = '/users'

// Equivalent to module.exports
exports.getUsers = async function (endpoint) {
  try {
    const response = await axios.get(`${baseUrl}${usersEndpointPath}`)
    return response.data
  } catch (error) {
    console.log(error)
  }
}

exports.getUserById = async function (id) {
  try {
    const response = await axios.get(`${baseUrl}${usersEndpointPath}\\${id}`)
    return response.data
  } catch (error) {
    console.log(error)
  }
}
```

En este caso, en vez de exportar un objeto que contiene las funciones de llamada a la API, **exportamos una a una cada una de las funciones.**

Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Exportación de módulos

Importación de módulos

Conclusiones

Importación de módulos

Podemos importar módulos de tres maneras:

```
const ApiModule = require('./8.apiModule')  
const myFunctions = require('./9.exportFunctions')  
const { getUsers, getUserById } = require('./9.exportFunctions')
```

- Si se importa un objeto de otro módulo:

```
async function processUsers1 () {  
  try {  
    const users = await ApiModule.getUsers()  
    console.log(`Users: ${JSON.stringify(users)}`)  
  } catch (error) {  
    console.log(error)  
  }  
}
```

- Si se importan todas las funciones de otro módulo:

```
const users = await myFunctions.getUsers()
```

- Si solo se importan algunas funciones de otro módulo:

```
const users = await getUsers()
```


Objetivo

Variables y tipos de datos

Cadenas

Objetos

Arrays

Programación estructurada

Programación asíncrona

Módulos de API

Exportación de funciones

Importación de módulos y funciones

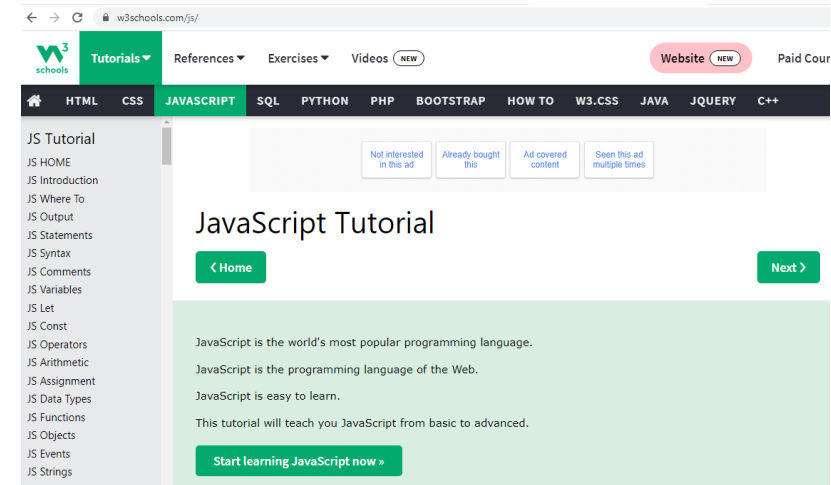
Conclusiones

- Se ha realizado una introducción a Javascript donde se han resaltado los aspectos principales que hacen falta para la asignatura:
 - Importación y exportación de módulos propios.
 - Importación de librerías de terceros:
 - Llamadas a API Restful mediante Axios.
 - Llamadas asíncronas:
 - Promesas
 - Async/await
 - Otros aspectos habituales de un lenguaje de programación.

Referencia de Javascript

- En **w3schools** podemos encontrar una completa referencia del lenguaje y herramientas para hacer pruebas:

- <https://www.w3schools.com/js/>



- Referencia oficial del lenguaje ECMA Script 2022

- <https://tc39.es/ecma262/multipage/>



Introducción a ECMA Script (Javascript)

Técnicas de Visualización de la Información

Máster en Ingeniería del Software: Cloud, Datos y Gestión TI

Departamento de Lenguajes y Sistemas Informáticos



Visualización mediante librerías javascript

Técnicas de Visualización de la Información

Máster en Ingeniería del Software: Cloud, Datos y Gestión TI

Departamento de Lenguajes y Sistemas Informáticos





Recomendaciones para visualización de información

Estándares web para la visualización online






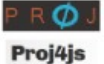
Bibliotecas para la visualización web

Ejemplo práctico con D3.js




HERRAMIENTAS DE VISUALIZACIÓN DE DATOS

	KIBANA	GRAFANA	TABLEAU PUBLIC	SAPAGOBI
LOGO				
VERSIONES EJECUTABLES	Escritorio/Nube			Escritorio
EXTENSIONES/ LIBRERÍAS	Si (Mathlion, Markdown Doc View, Wazh, entre otras)	Si (Zabbix, 3D Globe Panel, PostgreSQL, entre otras)	Si (Dynamic Parameters, 3-D Scatterplot, D3 Projections, entre otras)	No
PRECIO	Gratuita / diferentes versiones de pago		Gratuita (*) / diferentes versiones de pago	Gratuita
(*) Solo visualizaciones públicas				

HERRAMIENTAS DE VISUALIZACIÓN GEOESPACIAL

	CARTO	OPEN LAYERS	OPEN STREET MAP	GEOCODER Y GEOPY	GDAL	PROJ.4 Y PROJ4.JS
LOGO		 OpenLayers	 OpenStreetMap	 GeoPy	 GDAL	 Proj4js
TIPO DE HERRAMIENTA	Visualización			Librerías de geocodificación	Librerías de traslación	Librerías de transformación de coordenadas
EXTENSIONES	GDAL, PostgreSQL, Deck.gl, Python, entre otras	TextPath, AnimatedCluster, Canvas, GeoRSS, entre otras	Geocoder, Kartograph, Atlas, GDAL, entre otras	Se pueden invocar desde: Python, PostGIS, JavaScript, OpenStreetMap, entre otros	Se pueden invocar desde: PostGIS, Carto, ArcGIS, R, entre otros	Se pueden invocar desde: JavaScript, Ruby, MySQL, Excel, entre otros
PRECIO	Gratuita/diferentes versiones de pago	Gratuita				

HERRAMIENTAS Y APIS DE VISUALIZACIÓN DE DATOS

	GOOGLE CHART TOOLS	JAVASCRIPT INFOVIS TOOLKIT	D3.JS	MATPLOTLIB	BOKEH
LOGO	Google Charts	JavaScript InfoVis Toolkit			
TIPO	API	Librería			
LENGUAJE DE PROGRAMACIÓN	Javascript			Python	
EXTENSIONES	Java, ChartWeapper o ChartEditor	RGraph, Sunburst o ForceDirected	c3, d3-timeseries, plotly.js, d3-carto-map, entre otras	Basemap, brokenaxes, animatplot, mpl_interactions, entre otras	node.js, html_button.py, jsmol-bokeh- extension, entre otras

Recomendaciones para visualización de información

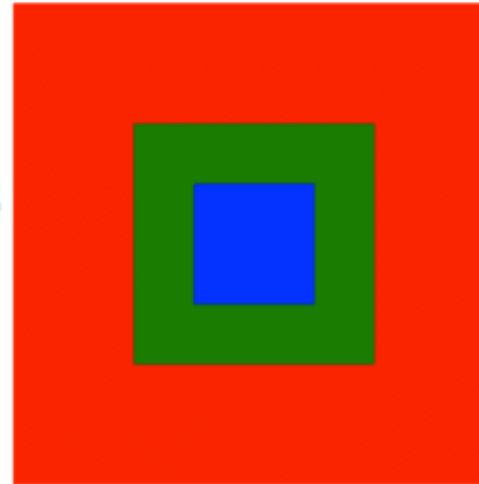
Estándares web para la visualización online

Bibliotecas para la visualización web

Ejemplo práctico con D3.js

Estándares para la visualización web: canvas

```
<html>
<body>
  <canvas id="myCanvas" width="200" height="200" />
  <script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");
    ctx.fillStyle = "red";
    ctx.fillRect(0, 0, 200, 200);
    ctx.fillStyle = "green";
    ctx.fillRect(50, 50, 100, 100);
    ctx.fillStyle = "blue";
    ctx.fillRect(75, 75, 50, 50);
  </script>
</body>
</html>
```



Elementos

- Fill: color fondo

En el HTML:

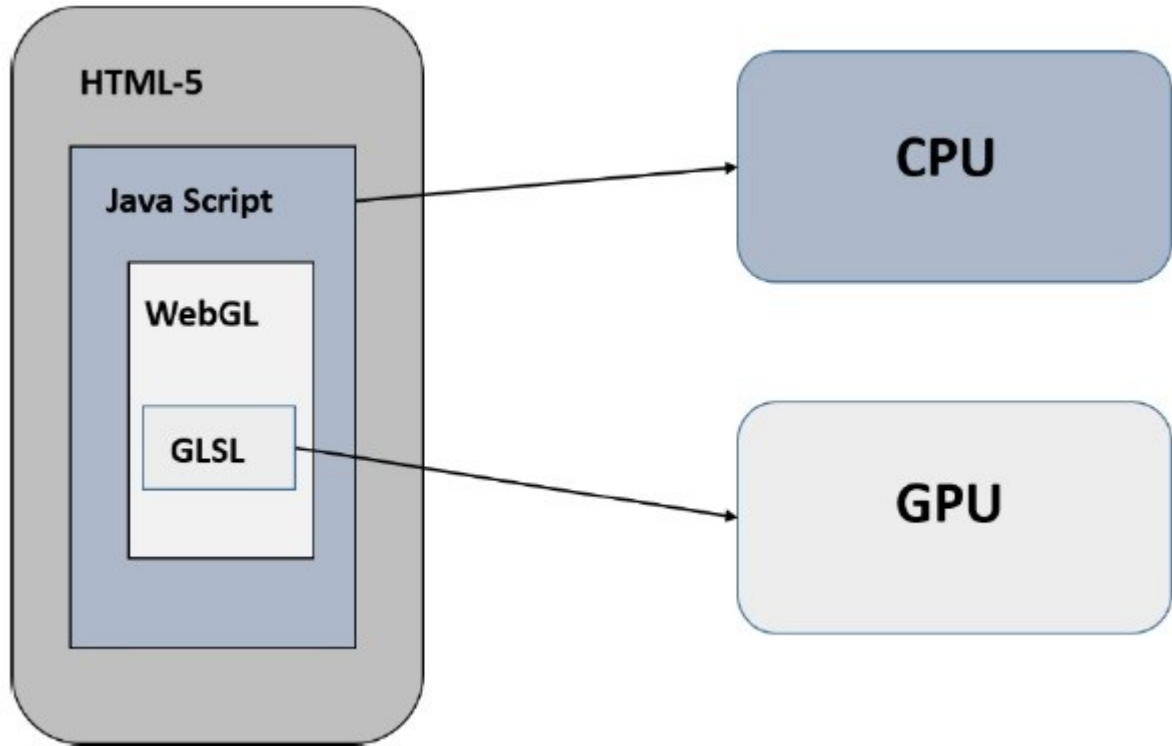
```
<svg class="lienzo">  
<circle cx="50" cy="50" r="40" class="estilo"/>  
</svg>
```

En el CSS:

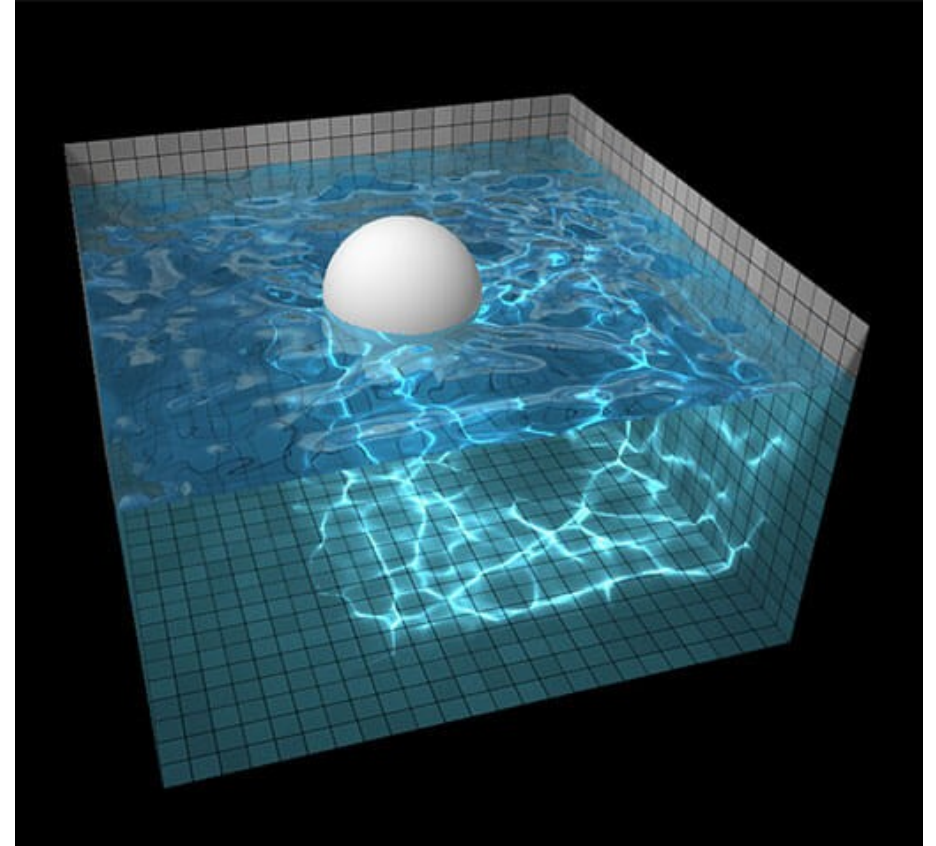
```
.estilo{  
  fill: blue;  
}
```



Estándares para la visualización web: webgl



https://www.tutorialspoint.com/webgl/webgl_sample_application.htm



<https://davidwalsh.name/webgl-demo>

Estándares para la visualización web: webgl

```
<!doctype html>
<html>
<body>
  <canvas width = "300" height = "300" id = "my_Canvas"></canvas>

  <script>
    /* Step1: Prepare the canvas and get WebGL context */

    var canvas = document.getElementById('my_Canvas');
    var gl = canvas.getContext('experimental-webgl');

    /* Step2: Define the geometry and store it in buffer objects */

    var vertices = [-0.5, 0.5, -0.5, -0.5, 0.0, -0.5,];

    // Create a new buffer object
    var vertex_buffer = gl.createBuffer();

    // Bind an empty array buffer to it
    gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);

    // Pass the vertices data to the buffer
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

    // Unbind the buffer
    gl.bindBuffer(gl.ARRAY_BUFFER, null);

    /* Step3: Create and compile Shader programs */

    // Vertex shader source code
    var vertCode =
      'attribute vec2 coordinates;' +
      'void main(void) { ' + ' gl_Position = vec4(coordinates,0.0, 1.0);' + ' }';

    //Create a vertex shader object
    var vertShader = gl.createShader(gl.VERTEX_SHADER);

    //Attach vertex shader source code
    gl.shaderSource(vertShader, vertCode);

    //Compile the vertex shader
    gl.compileShader(vertShader);

    //Fragment shader source code
    var fragCode = 'void main(void) { ' + 'gl_FragColor = vec4(0.0, 0.0, 0.0, 0.1);' + ' }';

    // Create fragment shader object
    var fragShader = gl.createShader(gl.FRAGMENT_SHADER);

    // Attach fragment shader source code
    gl.shaderSource(fragShader, fragCode);

    // Compile the fragment shader
    gl.compileShader(fragShader);

    // Create a shader program object to store combined shader program
    var shaderProgram = gl.createProgram();
```

```
    // Link both programs
    gl.linkProgram(shaderProgram);

    // Use the combined shader program object
    gl.useProgram(shaderProgram);

    /* Step 4: Associate the shader programs to buffer objects */

    //Bind vertex buffer object
    gl.bindBuffer(gl.ARRAY_BUFFER, vertex_buffer);

    //Get the attribute location
    var coord = gl.getAttribLocation(shaderProgram, "coordinates");

    //point an attribute to the currently bound VBO
    gl.vertexAttribPointer(coord, 2, gl.FLOAT, false, 0, 0);

    //Enable the attribute
    gl.enableVertexAttribArray(coord);

    /* Step5: Drawing the required object (triangle) */

    // Clear the canvas
    gl.clearColor(0.5, 0.5, 0.5, 0.9);

    // Enable the depth test
    gl.enable(gl.DEPTH_TEST);

    // Clear the color buffer bit
    gl.clear(gl.COLOR_BUFFER_BIT);

    // Set the view port
    gl.viewport(0,0,canvas.width,canvas.height);

    // Draw the triangle
    gl.drawArrays(gl.TRIANGLES, 0, 3);
  </script>
</body>
</html>
```



Recomendaciones para visualización de información

Estándares web para la visualización online

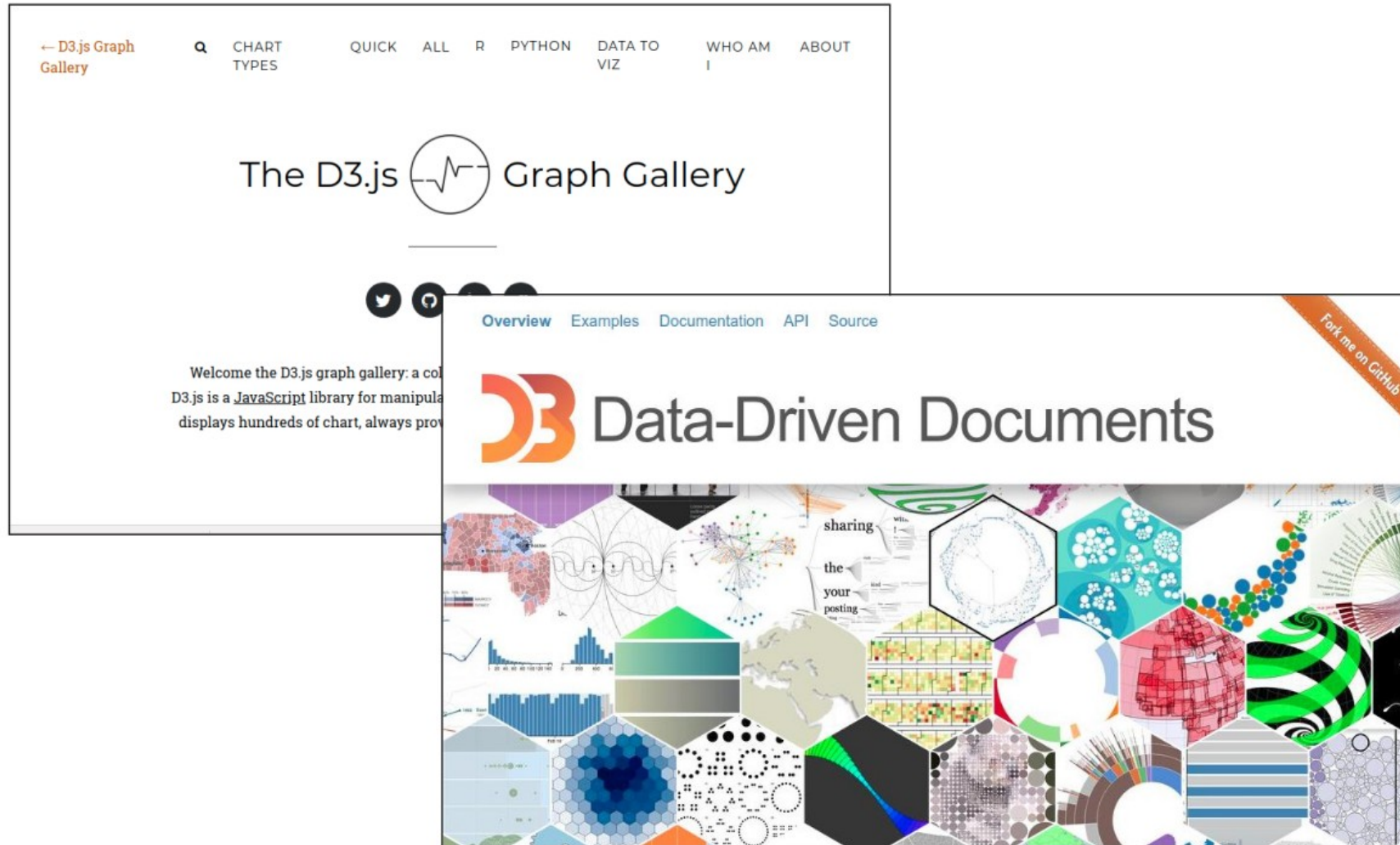
Bibliotecas para la visualización web

Ejemplo práctico con D3.js

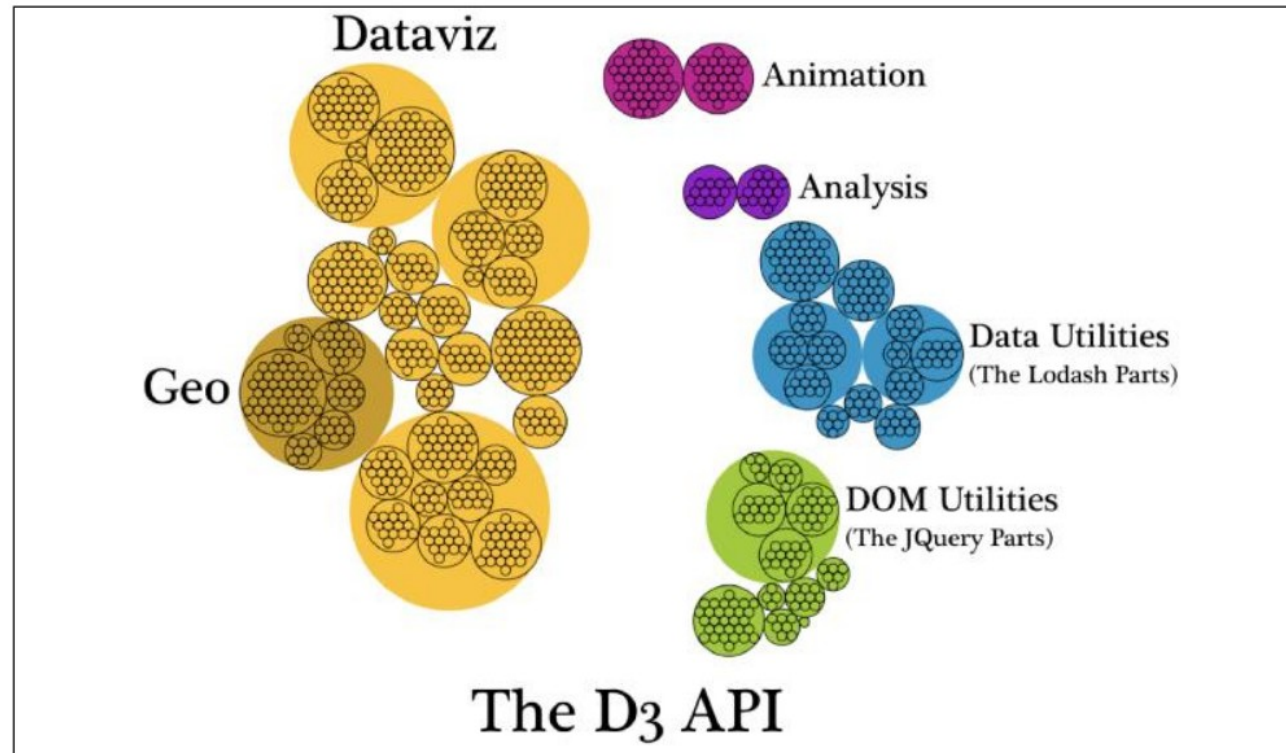
Bibliotecas para visualización web: Gráficos 2D

	Estándar	Paleta de gráfico	Acabado final	Facilidad de uso	Extensible	Modalidad de uso
D3.js	SVG	Altamente extensible	Bueno	Difícil	Si	Gratuita
Google Charts	SVG	Muy amplia	Bueno	Sencillo	No	Gratuita
Chart.JS	Canvas	Líneas, Barras, Radar y Tarta	Muy bueno	Sencillo	No	Gratuita
HighchartsJS	Canvas	Muy amplia	Muy bueno	Sencillo	No	Pago/Gratuita
JavaScript InfoVis Toolkit	WebGL	Bastante reducida	Regular	Difícil	No	Gratuita

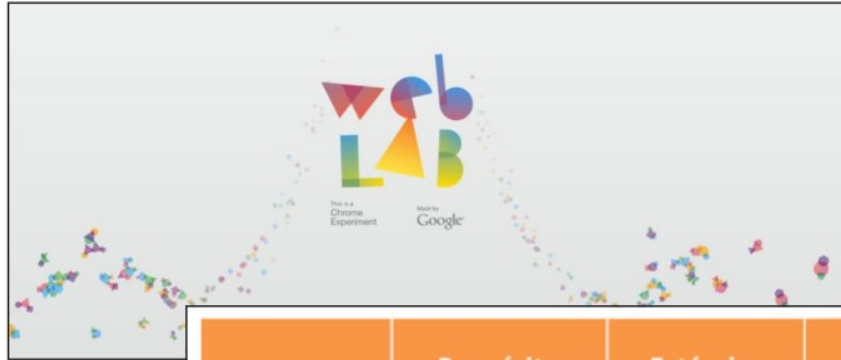
Bibliotecas para visualización web: Gráficos 2D



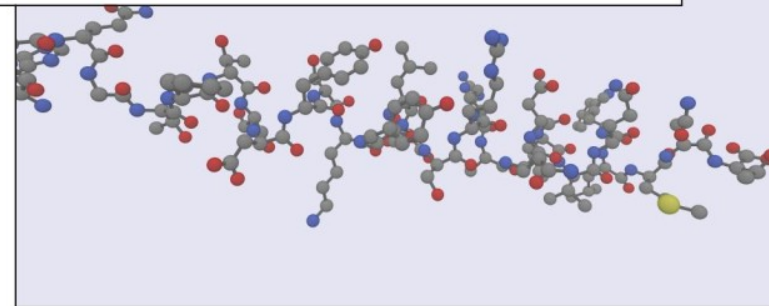
Bibliotecas para visualización web: Gráficos 2D



Bibliotecas para visualización web: Gráficos 3D



	Propósito	Estándar	Gráficas y Animaciones 3D	Dependencias
Three.js	General	WebGL	Sí	No
BabylonJS	Animaciones, juegos	HTML5, WebGL	Sí	Hand.js



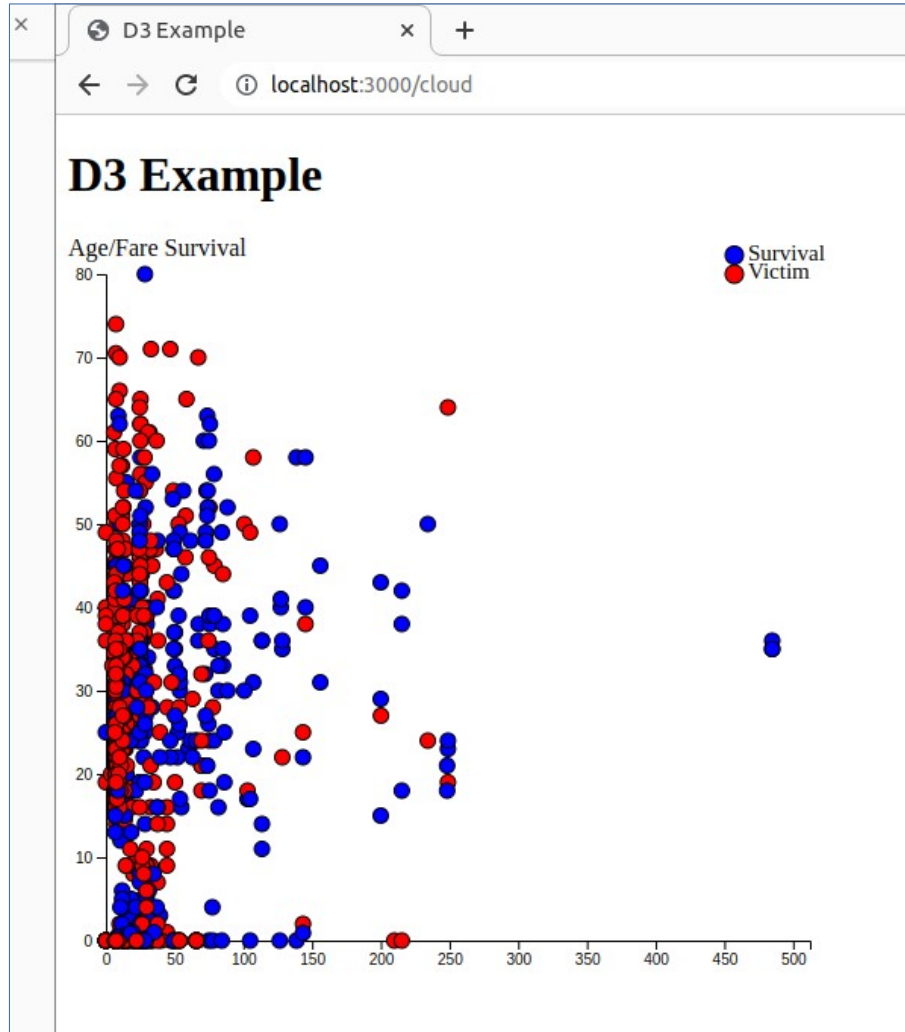
Recomendaciones para visualización de información

Estándares web para la visualización online

Bibliotecas para la visualización web

Ejemplo práctico con D3.js

Ejemplo práctico con D3: Supervivientes Titanic



```
d3.csv("data/titanic.csv", row=>jsonify(row)).then(data=>updateSVG(data))
```

Ejemplo práctico con D3: Supervivientes Titanic

```
import * as d3 from "https://cdn.skypack.dev/d3@7";

const height = 465
const marginY = 25
const width = 465
const marginX = 25

function jsonify(row, i){
  return {
    name : row.Name,
    survived : (row.Survived == 1) ? "Yes" : "No",
    sex : row.Sex,
    age : +row.Age,
    fare : +row.Fare,
    key : i
  }
}
```


Ejemplo práctico con D3: Supervivientes Titanic

```
function updateSVG(rows) {
  var ages = rows.map(row=>row.age);
  var fares = rows.map(row=>row.fare);
  var maxAge = d3.max(ages);
  var minAge = d3.min(ages);
  var maxFare = d3.max(fares);
  var minFare = d3.min(fares);

  // Set x, y and colors
  var xScale = d3.scaleLinear()
    .domain([0, maxFare]) // input
    .range([0, width]); // output

  var yScale = d3.scaleLinear()
    .domain([0, maxAge])
    .range([height, marginY]);

  d3.select("svg").append("g")
    .attr("class", "y axis")
    .attr("transform", "translate(" + marginX + "," + 0 + ")")
    .call(d3.axisLeft(yScale)); // Create an axis component with d3.axisLeft

  d3.select("svg").append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(" + marginX + "," + height + ")")
    .call(d3.axisBottom(xScale)); // Create an axis component with
    // d3.axisBottom
}
```

Ejemplo práctico con D3: Supervivientes Titanic

```
rows.forEach(row=>d3.select("svg").
  append("circle").
  attr("stroke", "black").
  attr("cy", marginY + (height-marginY)*(maxAge-row.age)/(maxAge-minAge)).
  attr("cx", marginX + (width-marginX)*(row.fare-minFare)/(maxFare-minFare)).
  attr("r", 5).
  attr("fill", (row.survived=="Yes")?"blue":"red").
  append("title").text(row.name + "\nFare: " + row.fare + "\nAge: " + row.age ));

d3.select("svg").append("text")
  .attr("x", 0)
  .attr("y", marginY/2)
  .style("font-size", "16px")
  .style("text-decoration", "bold")
  .text("Age/Fare Survival");

d3.select("svg").append("circle").attr("cx",width-marginX).
  attr("cy",marginY/2).attr("r", 6).style("fill", "blue").style("stroke", "black")
d3.select("svg").append("circle").attr("cx",width-marginX).
  attr("cy",marginY).attr("r", 6).style("fill", "red").style("stroke", "black")
d3.select("svg").append("text").attr("x", width-marginX + 9).
  attr("y", marginY/2).text("Survival").style("font-size", "15px").
  attr("alignment-baseline","middle")
d3.select("svg").append("text").attr("x", width-marginX + 9).
  attr("y", marginY).text("Victim").style("font-size", "15px").
  attr("alignment-baseline","middle")
```


- En la página oficial de D3 podemos encontrar ejemplos de uso de la librería bastante bien documentados.
- <https://d3js.org/>
- Referencia oficial del Ministerio de Asuntos Económicos y Transformación Digital
 - <https://datos.gob.es/es/documentacion/herramientas-de-procesado-y-visualizacion-de-datos>

[Overview](#) [Examples](#) [Documentation](#) [API](#) [Source](#)



- Modificar la visualización para que en lugar de tener una nube de puntos, visualicemos un diagrama de barras con el porcentaje de supervivientes para hombres y mujeres, ordenados por edad (década)
- Realizar la misma visualización pero cambiando la edad por el coste del pasaje.

NOTA: Use como base el código que se le suministra en d3-titanic-bars.js que se visualiza desde 'http://localhost:3000/bars'. Cambie el nombre a d3-titanic-bars-UVUS.js, donde UVUS es su usuario en blackboard y suba exclusivamente el fichero a la actividad.

Visualización mediante librerías javascript

Técnicas de Visualización de la Información

Máster en Ingeniería del Software: Cloud, Datos y Gestión TI

Departamento de Lenguajes y Sistemas Informáticos

