Javascript intermedio

Fechas en Javascript

Fechas

- En javascript, las fechas se guardan en **objetos Date**

```
let date = new Date('2019-07-21')
```

Fechas

- Date acepta strings con **otros formatos**

```
let date = new Date('01 Jan 1970 00:00:00 GMT')
```

Fechas

- Si no pasamos string, crea un objeto Date **con fecha de hoy**

```
let date = new Date()
```

Objeto Date

- Los objetos Date tienen varios métodos útiles:

```
let date = new Date('01 Jan 1970 00:00:00 GMT')

date.getDay() // 0-6 dependiendo del dia de la semana
date.getMonth() // 0
date.getFullYear() // 1970
date.getHour() // // 0
date.getTime() // 156223453453
```

Objeto Date

- Podemos comparar los objetos Date para saber si una fecha es anterior a otra:

```
let date = new Date('01 Jan 1970 00:00:00 GMT')
let date2 = new Date('01 Jan 2019 00:00:00 GMT')

if(date > date2) …
if(date >= date2) …
if(date < date2) …
if(date <= date2) …
```

Objeto Date

- **No podemos igualar fechas** para saber si son iguales

```
let date = new Date('01 Jan 1970 00:00:00 GMT')
let dateCopy = new Date('01 Jan 1970 00:00:00 GMT')

date === dateCopy // false (wrong!)
```

Objeto Date

- **Usamos getTime para comprobar igualdad**

```
let date = new Date('01 Jan 1970 00:00:00 GMT')
let dateCopy = new Date('01 Jan 1970 00:00:00 GMT')

date.getTime() === dateCopy.getTime() // true
```

ES6

ES6

Javascript es un lenguaje que **evoluciona** continuamente

- Hay varias versiones: ES1, ES2, ES3, ES4, ES5, ES6, ES7…
- Durante mucho tiempo se ha utilizado **ES5** y todavía se utiliza por temas de compatibilidad (Internet Explorer principalmente)

ES6

En 2015 se publica la versión **ES6** que revoluciona el lenguaje con nuevas funcionalidades y **syntax sugar**

Vamos a ver algunos de estos cambios

Template strings

Template strings

Permiten insertar variables dentro de una string fácilmente

- Se declaran usando **backticks** (``)
- Podemos meter **expresiones** dentro de ${} (variables, valores, etc.)

```
let error = 404
let message = "Page not found"

let str= `Error ${error}: ${message}`
console.log(str)      // Error 404: Page not found
```

Ejercicio templates strings

- Crea una arrow function que reciba un tag name y devuelva un nodo HTML
- Ejemplo:
  - **Input**: “div”
  - **Output** : “ ”

Ejercicio

- Utiliza **string templates** para…
  - crear un programa que muestra la hora (*HH:MM:SS*) por la consola cada segundo

Variables

ES6

Se añaden **let** y **const** para reemplazar a las antiguas **var**

- **var** tenía comportamientos extraños en cuanto al **scope**
  - Ejemplo: **hoisting**

Hoisting

```
function myFunc() {
  console.log('valor: ', x)
  var x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myFunc() {
  var x
  console.log('valor: ', x)
  x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myFunc() {
  console.log('valor: ', x)
  let x = 12
  console.log('valor: ', x)
}

myFunc()
```

```
function myLoop() {
  for (var i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

```
function myLetLoop() {
  for (let i=0; i <= 10; i++) {
    // no-op
  }
  return i
}
```

Ejercicio

- Encuentra y arregla el bug

```
function randomNumber(n) {
  if (Math.random() > .5) {
    let base = 1
  } else {
    let base = -1
  }
  return base * n * Math.random()
}
```

Ejercicio templates strings

- Crea un bucle que imprima los números del 1 al 10 con **300ms** de espacio entre cada impresión

Arrow functions

Arrow functions

Las **arrow functions** son un tipo de **syntax sugar** que permite crear funciones con menos código

```
const suma = (a, b) => a + b
```

Equivale a:

```
function suma(a, b){
     return  a + b
}
```

Arrow functions

Especialmente útiles cuando pasamos una función por parámetro:

```
setTimeout(() => console.log("Hola"), 1000)
```

Equivale a:

```
setTimeout(function(){ console.log("Hola") }, 1000)
```

Arrow functions

Si la arrow function **tiene más de una línea** debemos usar **llaves** y **return**

```
const suma2 = () => {
     result = 2 + 2
     return result
}
```

Arrow functions

Si la función **devuelve un objeto**, usamos **paréntesis**

```
const func = () => ({a: 1, b: 2})
```

Destructuring

Destructuring

Permite descomponer un array en varias variables:

```
[a, b] = [10, 20]
console.log(a) // 10
console.log(b) // 20
```

Destructuring

Permite descomponer un array en varias variables:

```
[a, b, ...rest] = [10, 20, 30, 40, 50]
console.log(a) // 10
console.log(b) // 20
console.log(rest) // [30, 40, 50]
```

Rest operator: …

- Permite capturar el resto de variables en forma de array

Destructuring

Podemos descomponer cualquier número de variables

```
[a] = [10, 20, 30, 40, 50]
[a, b] = [10, 20, 30, 40, 50]
```

```
// only first
// only first and second
```

En cualquier posición

```
[, , a] = [10, 20, 30, 40, 50]
[, , a, b] = [10, 20, 30, 40, 50]
```

```
// only third
// only third and fourth
```

Ejercicio rest

Cuanto vale tail en cada caso

```
const [head, tail] = [ 1, 2, 3]

const [head, ...tail] = [1, 2]

const [head, ...tail] = [1]

const [head, , ...tail] = [1, 2, 3]
```

Destructuring

Permite descomponer un objeto en varias variables:

```
const { x, y } = { x: 10, y: 20 }
console.log(a) // 10
console.log(b) // 20
```

Ejercicio

- Desestrucutura el objeto `{ uno: 1, dos: 2 }` en dos variables: **uno** y **dos**

Ejercicio

- Utiliza desestructuración para **intercambiar el valor** de las variables **a** y **b** *(sin crear ninguna otra variable!)*

```
let a = 1
let b = 2
// ???

console.log(a, b) // “2 1”
```

Destructuring

Podemos cambiar el nombre de las variables al desestructurarlas

```
const { x: equis, y: igriega } = { x: 10, y: 20 }
```

Podemos desestructurar de forma anidada

```
const { x: { y } } = { x: { y: 10 } }

// ?
```

Destructuring

Podemos desestructurar de forma anidada

```
const { x: { y } } = { x: { y: 10 } }
```

```
{ y }  = { y: 10 }
```

```
y = 10
```

Ejercicio

- Desestructura el siguiente objeto en las variables **uno**, **dos**, **tres**, **cuatro** y **cinco**

```
{ uno: 1, lista: [2, 3], cuatro: 4, x: { cinco: 5 } }
```

Ejercicio

- Desestructura el siguiente objeto en las variables **a**, **b**, **c**, **d** y **e**

```
{ uno: 1, lista: [2, 3], cuatro: 4, x: { cinco: 5 } }
```

Destructuring

El **rest operator** también permite recibir los argumentos de una función en forma de array

```
function consoleLogEverything(...things){
     for(let thing of things){
           console.log(thing)
     }
}

consoleLogEverything(1, ´dos´, true)
// 1
// dos
// true
```

Destructuring

El operador opuesto a **rest** es **spread**

```
function suma(a, b){
     return a + b
}

let nums = [1, 2]
suma(...nums)
```

Representa todos los elementos de un array, pero **de uno en uno**

Se puede utilizar para convertir un array en parámetros

Ejercicio destructuring

- Crea una arrow function que reciba un número cualquiera de argumentos numéricos y devuelva el más pequeño

Ejercicio destructuring

- Crea una arrow function que reciba un número cualquiera de argumentos numéricos y devuelva el más pequeño y el más grande
- Llama a la función y captura los dos valores en variables separadas utilizando destructuring

Scope

Qué es el scope

- El acceso que tenemos a variables y funciones en una zona específica de nuestro código

Scope en Javascript

```
// Global Scope
function someFunction() {
    // Local Scope #1
    function someOtherFunction() {
        // Local Scope #2
    }
}

// Global Scope
function anotherFunction() {
    // Local Scope #3
}
// Global Scope
```

Ejemplo I

```
let name = 'Hammad'
function logName(){
     let name = 'Sajid'
     console.log(name)
}
console.log(name)

// ?
// ?
```

Ejemplo I

```
let name = 'Hammad'
function logName(){
    let name = 'Sajid'    <- El scope local tiene prioridad
    console.log(name)
}
console.log(name)

// Sajid
// Hammad
```

Ejemplo II

```
let name = 'Hammad'
function logName(){
    name = 'Sajid'
     console.log(name)
}
console.log(name)

// ?
// ?
```

Ejemplo II

```
let name = 'Hammad'
function logName(){
    name = 'Sajid'
     console.log(name)
}
console.log(name)

// Sajid
// Sajid
```

Scope bloques if, for, etc.

```
if (true) {
    let name = 'Hammad' // name está en el global scope
}


console.log(name) // logs 'Hammad'
```

- Los bloques if, for y switch no crean scopes locales.

Por qué limitar el scope?

- Es una buena práctica **tener acceso solo a lo que se necesita**: evitamos confusiones y errores innecesarios.
- **Evitamos choques** entre nombres de variables que usamos muchas veces en nuestro código: i, index, name, result, etc.

Scope: variables vs. funciones

- Las variables se tienen que crear antes de ser usadas y estar en el scope adecuado
- Las funciones son accesibles se creen antes o después, siempre que estén en el scope adecuado

Scope: variables vs. funciones

```
alfa()
function alfa(){
      beta()
      function beta(){
            //…
      }
      beta()
}
alfa()
beta()
```

Scope: variables vs. funciones

```
alfa()     // ok
function alfa(){
     beta()
     function beta(){
          //…
     }
     beta()
}
alfa()
beta()
```

Scope: variables vs. funciones

```
alfa()
function alfa(){
     beta()
     function beta(){
          //…
     }
     beta()
}
alfa()
beta()     //error!
```

Clausuras

Closure

```
function CrearNombreySaludo() {
    name = 'Maria'
    function saludar() {
        console.log('Hola ' + name)
    }
}

saludar() // ?
```

Closure

```
function CrearNombreySaludo() {
    name = 'Maria'
    return function () {
        console.log('Hola ' + name)
    }
}

let saludar = CrearNombreYSaludo() // Que hay aqui?
saludar()
```

Closure

```
function CrearNombreySaludo() {
    name = 'Maria'
    return function () {
        console.log('Hola ' + name)
    }
}

let saludar = CrearNombreYSaludo()
saludar() // Que imprime esto?
```

Closure

```
function CrearNombreySaludo() {
    name = 'Maria'
    return function () {
        console.log('Hola ' + name)
    }
}

let saludar = CrearNombreYSaludo()
saludar() // Hola Maria
```

Closure

```
function CrearNombreySaludo() {
    name = 'Maria'
    return function () {
        console.log('Hola ' + name)
    }
}

let saludar = CrearNombreYSaludo()
saludar()
```

Ejercicio clausuras

- Crea una función que reciba un número y devuelva una función que sume ese número a un nuevo número que la segunda función recibe por parámetro

Clausuras II

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i = i++;
  };
}
```

Clausuras

```
const c1 = counter();
```

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i = i++;
  };
}
```

```
const c1 = counter();
console.log(c1());
```

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i = i++;
  };
}
```

```
const c1 = counter();
console.log(c1()); // 0
```

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i = i++;
  };
}
```

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Clausuras

```
function counter() {
  return () => {
    let i = 0;
    return i = i++;
  };
}
```

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 0
console.log(c1()); // 0
```

Clausuras

```
const c1 = counter();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

Clausuras

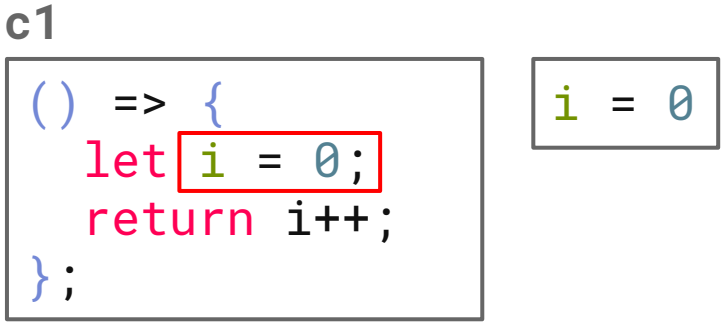
```
const c1 = counter();
c1();
```

c1

```
() => {
  let i = 0;
  return i++;
};
```

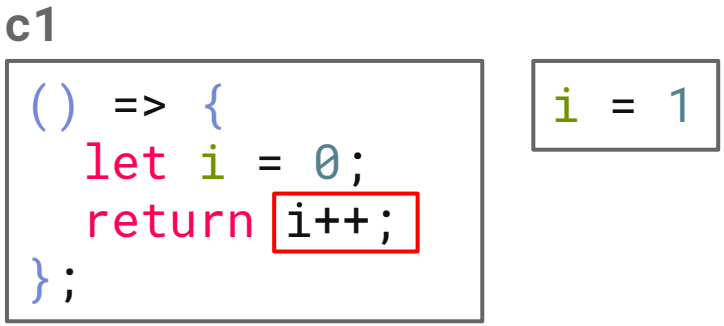
Clausuras

```
const c1 = counter();
c1();
```



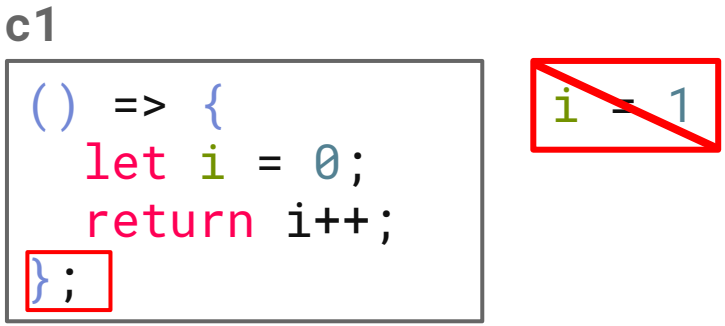
Clausuras

```
const c1 = counter();
c1(); // 0
```



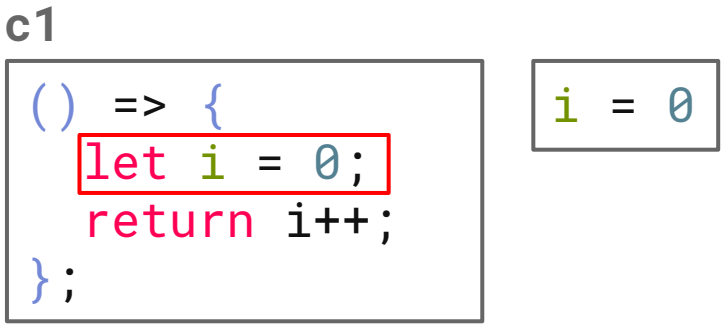
Clausuras

```
const c1 = counter();
c1(); // 0
```



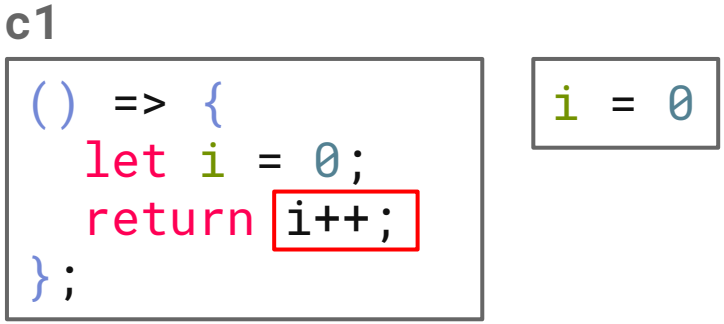
Clausuras

```
const c1 = counter();
c1(); // 0
c1();
```



Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 0
```



Clausuras

```
function counter() {
  let i = 0;
   return () => i++;
}
```

Variable libre

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Clausuras

```
const c1 = counter();
console.log(c1());
```

Clausuras

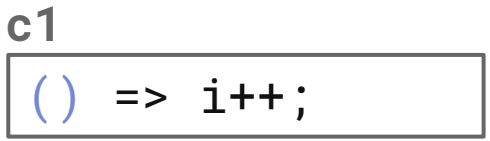
```
const c1 = counter();
console.log(c1()); // 0
console.log(c1());
```

Clausuras

```
const c1 = counter();
console.log(c1()); // 0
console.log(c1()); // 1
console.log(c1()); // 2
```

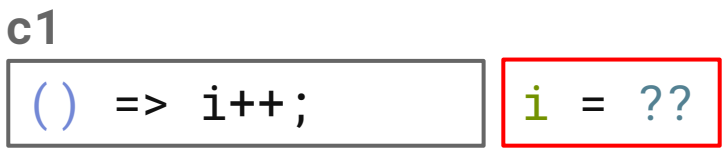
Clausuras

```
const c1 = counter();
let i = 10;
console.log(c1()); // ???
console.log(i); // ???
```



Clausuras

```
const c1 = counter();
let i = 10;
c1();              // 0
console.log(i); // 10
```



Clausuras

```
const c1 = counter();
c1(); // 0
c1(); // 1

const c2 = counter();
c2(); // ???
```

Clausuras

```
function counter() {
  let i = 0;
  return () => i++;
}
```

Funciones sobre listas

Funciones sobre listas

Tres funciones fundamentales

```
// map
let array = [1, 2, 3]
array.map(x => x + 1) // [2, 3, 4]
```

```
// filter
let array = ["lorem", "foo", "bar", "ipsum", "at" "dolor"]
array.filter(x => x.length > 3) //["lorem", "ipsum", "dolor"]
```

```
// reduce
let array = [1, 2, 2, 2]
array.reduce((acc, x) => acc + x, [])
```

Demo

Ejercicio map

Realiza las siguientes transformaciones utilizando map

- [8, 12, 20] -> [4, 6, 10]
- [1, 7, 50] -> [“1”, “7”, “50”]
- [-2, 5, 15, -7, -8] -> [“-”, “+”, “+”, “-”, “-”]
- [“lorem ipsum dolor”, “hello world”] -> [“lid”, “hw”]
- [[1, 2], [34, 20], [11, 11]] -> [3, 54, 22]
- [{name: “Alberto”}, {name: “Fran”}] -> [“Alberto”, ”Fran”]
- [[1, 2], [34, 20, 5], [11], [2, 4]] -> [3, 59, 11, 6]

Ejercicio filter

Realiza las siguientes transformaciones utilizando filter

Dado un array de números

- Conserva los números impares

Dado un array de objetos

- Conserva los objetos que tengan una propiedad **important:true**

Ejercicio reduce

Utiliza reduce para

- Calcular la resta de todos los números de un array
- Concatenar todas las strings de un array
- Calcular la suma de todos los números de un array (excepto los negativos)
- Encontrar el máximo de un array de números

Ejercicio reduce

Utiliza reduce para

- Calcular la resta de todos los números de un array
- Concatenar todas las strings de un array
- Calcular la suma de todos los números de un array (excepto los negativos)
- Encontrar el máximo de un array de números

Ejercicio listas avanzado

Implementa tu propia función map

Ejercicio listas avanzado II

Implementa tu propia función reduce

Ejercicio listas avanzado III

Utiliza una función reduce para

- Implementar tu propia función map
- Implementar tu propia función filter

Ficheros JSON

JSON

JSON: Javascript Object Notation

- Ficheros que guardan datos con el formato Object de Javascript
- Es un estándar y cualquier lenguaje de programación tiene herramientas para leerlos
- Ejemplo: pendiente

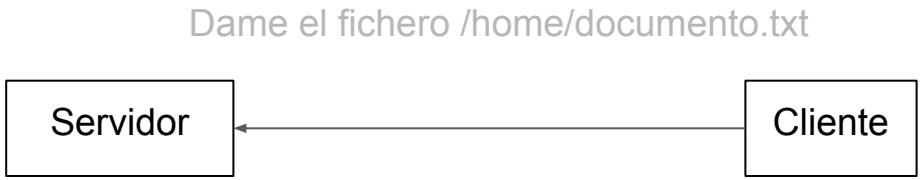
Arquitectura cliente/servidor

Cliente / servidor

- Modelo de comunicación entre dos programas

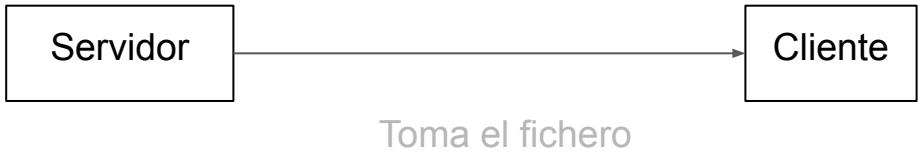
Cliente / servidor

- El cliente realiza una petición (request)



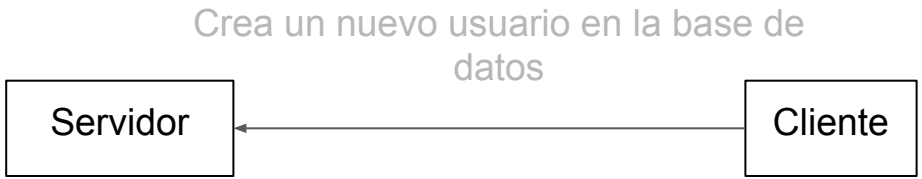
Cliente / servidor

- El cliente devuelve una respuesta (response)



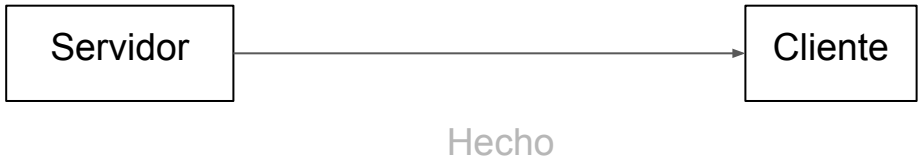
Cliente / servidor

- El cliente realiza una petición (request)



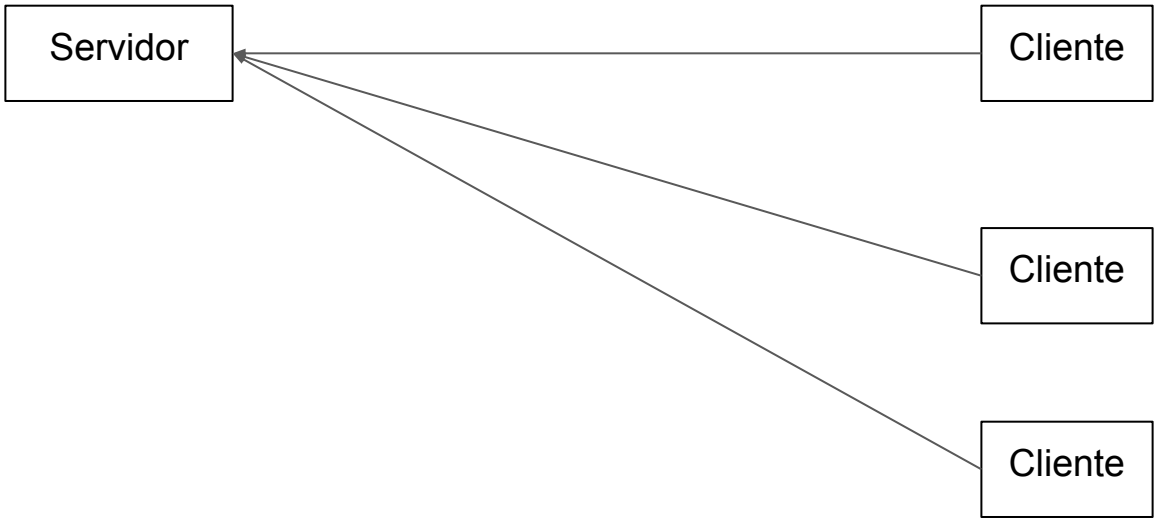
Cliente / servidor

- El cliente devuelve una respuesta (response)



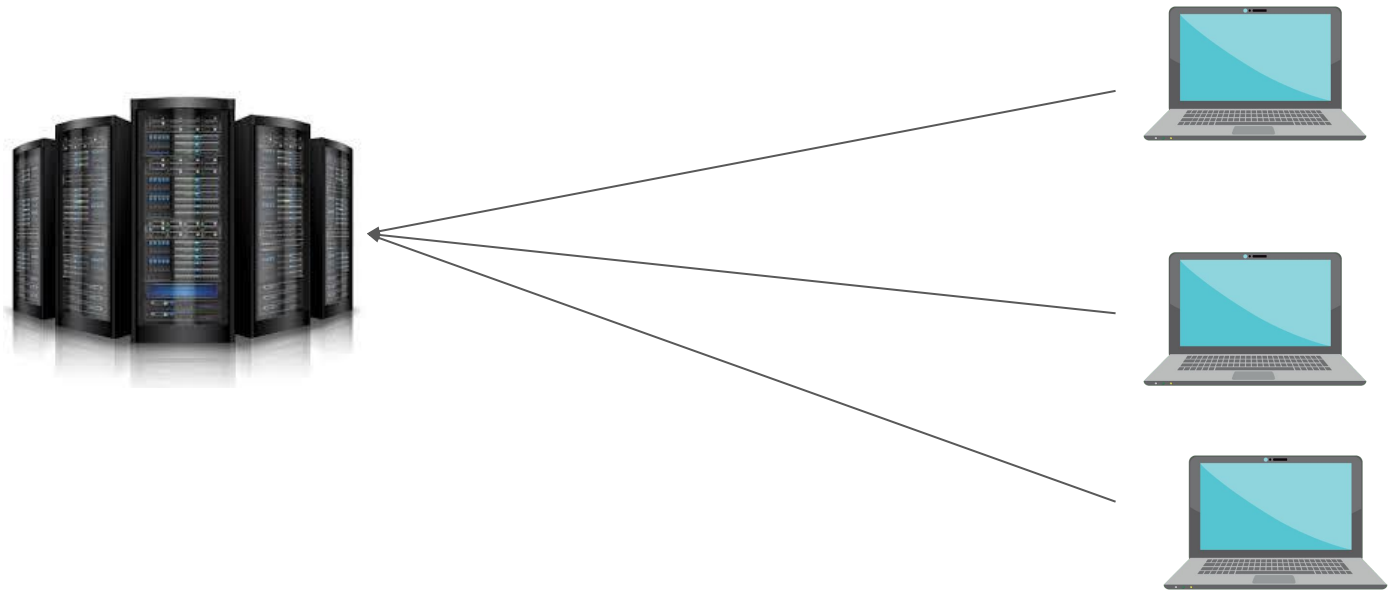
Cliente / servidor

- Un servidor puede recibir peticiones de múltiples clientes



Cliente / servidor

- El cliente y el servidor, **pueden** estar en máquinas separadas



Cliente / servidor

- Así es como funciona internet

Servidor

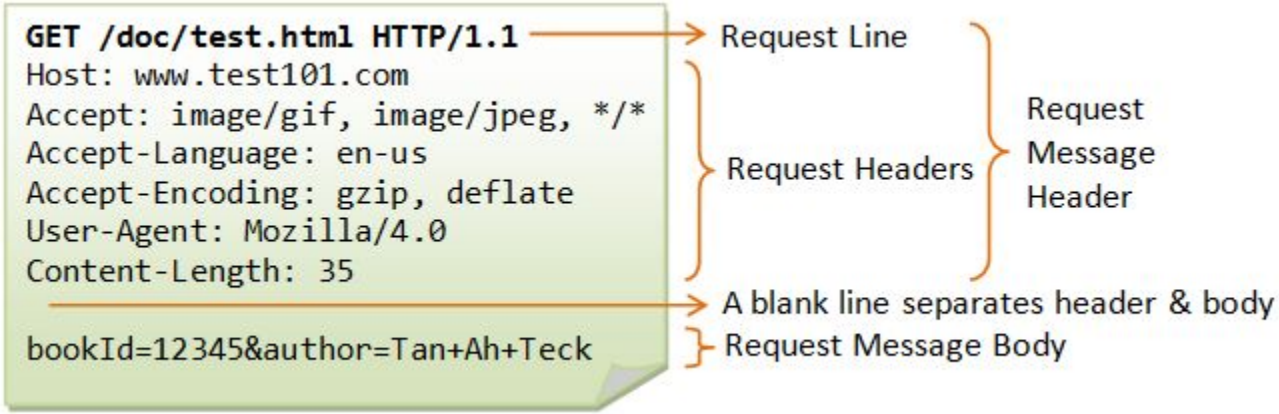
Un servidor (**hardware**) es un tipo de ordenador especial:

- Encendido las 24h para recibir peticiones
- Mucho más espacio (para guardar BBDD, ficheros, etc.)

Peticiones

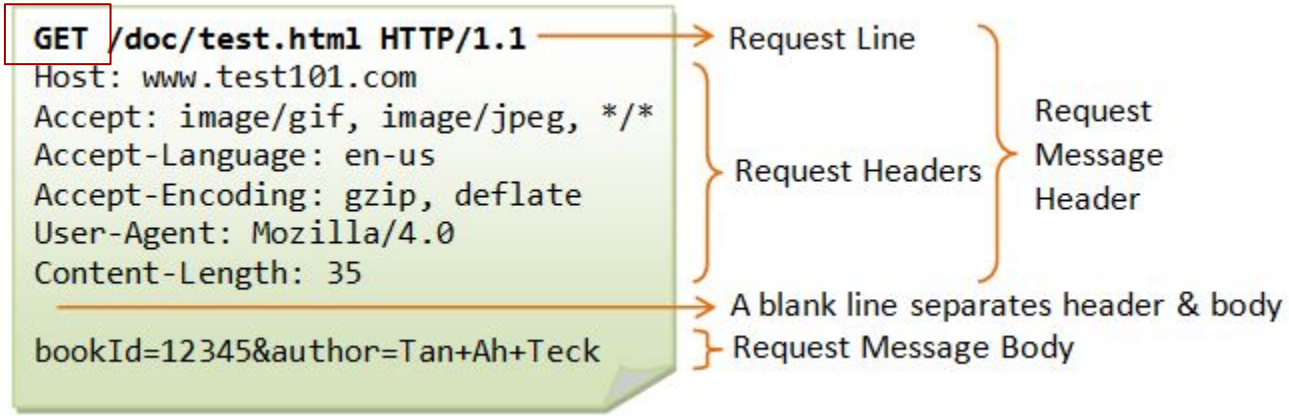
Get vs post

Requests



- Es un mensaje con un formato estándar

Requests



- Tipo de request

Tipos de peticiones

Tipos de requests

- GET
- POST
- PUT
- DELETE
- CONNECT
- OPTION
- TRACE

Tipos de requests

- GET
- POST
- PUT
- DELETE
- CONNECT
- OPTION
- TRACE

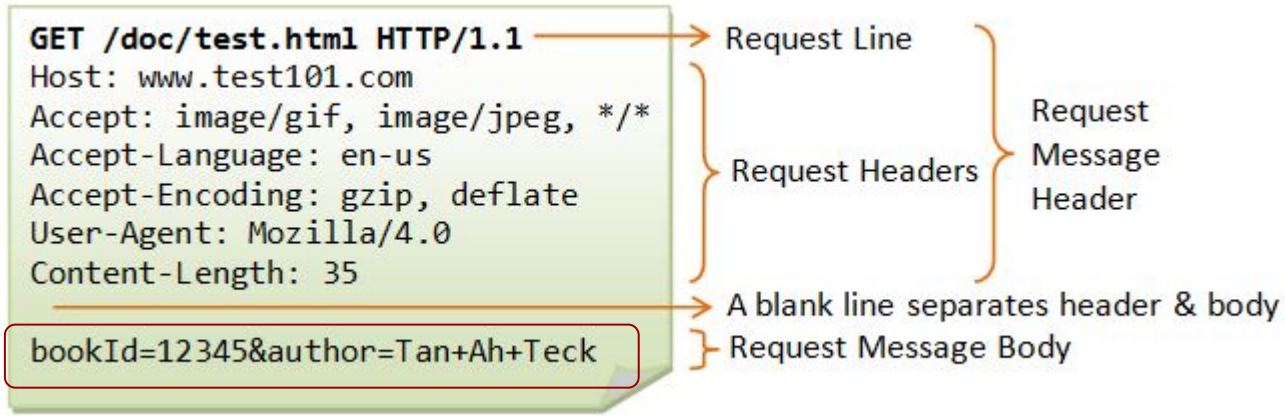
GET request

- Pide datos (eg: una página, un fichero) a un servidor a través de una URL
- La URL determina qué queremos
- La URL puede contener parámetros

    www.elperiodico.com/economia/1179
    www.google.com/search?query=gatitos

POST request

- Envía información al servidor (eg: al rellenar un formulario)
- La información va en el body de la request, ¿por qué?



Respuestas

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

Políticas de seguridad

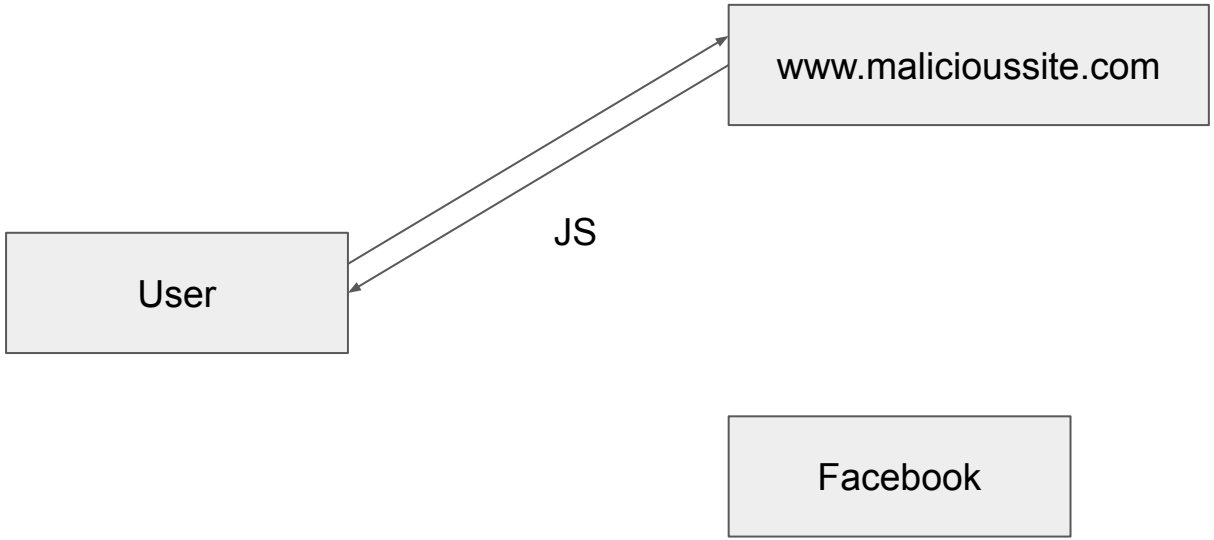
Políticas de seguridad

Las páginas web a menudo acceden a recursos (imágenes, enlaces, etc.) de **otros servidores**

Estos accesos se regulan mediante **políticas de seguridad**

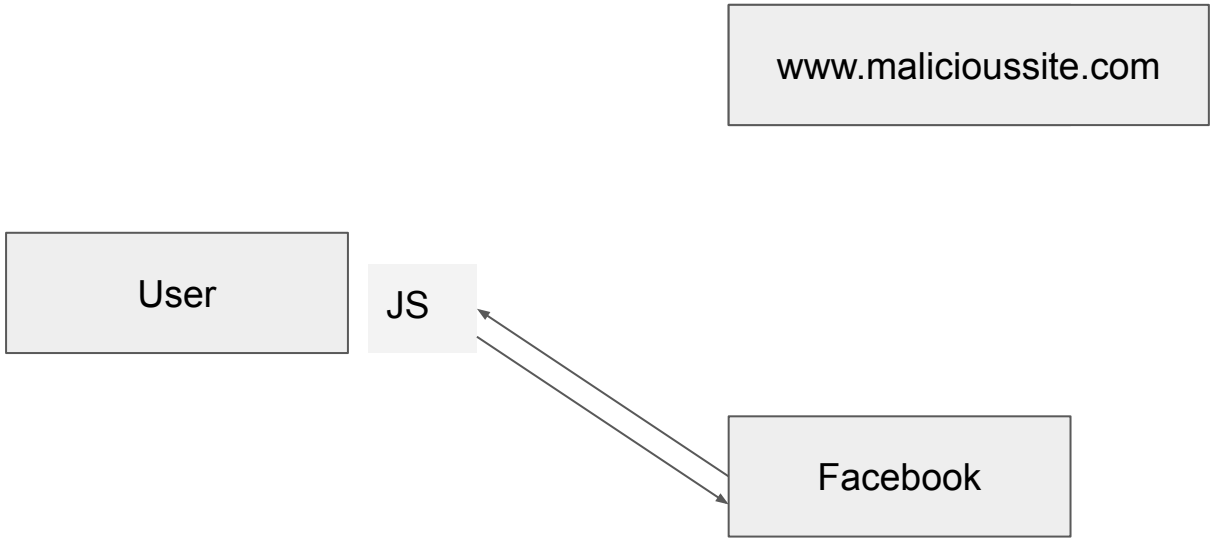
Políticas de seguridad

Acceder a recursos de otros servidores puede abrir vectores de ataque



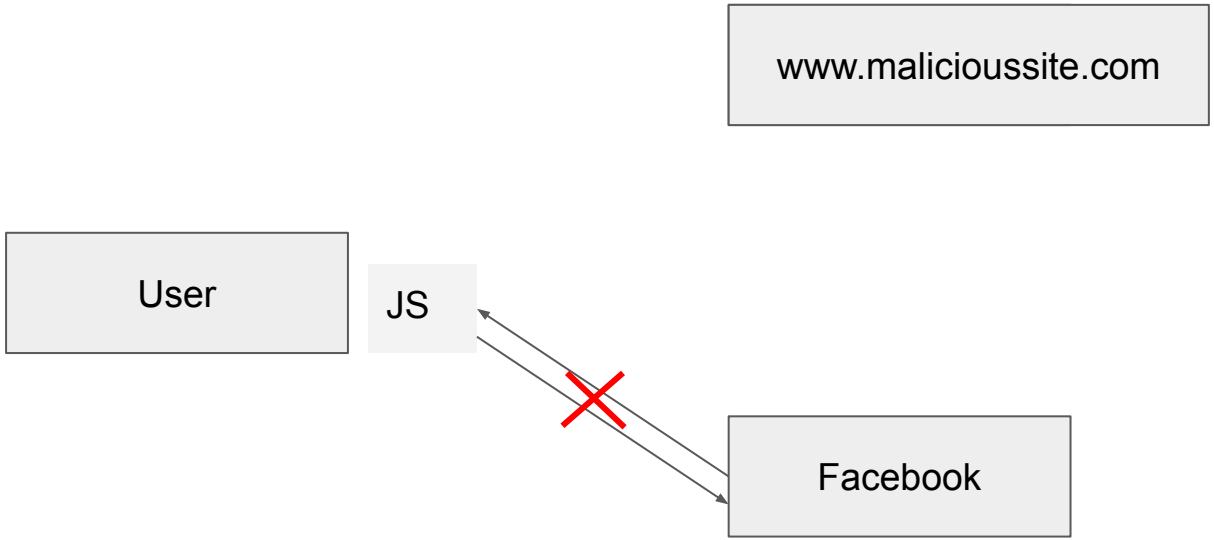
Políticas de seguridad

Acceder a recursos de otros servidores puede abrir vectores de ataque



same-origin

same-origin solo permite que los recursos de un servidor sean accesibles desde páginas de ese mismo servidor



same-origin

same-origin es el comportamiento por defecto en un servidor

No obstante, es una política de seguridad **muy restrictiva**

CORS

CORS

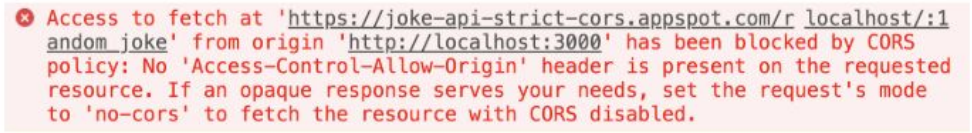
Cross origin resource sharing: permite recibir peticiones de otros **origins**

Especifica **CÓMO** se puede acceder a los recursos

- **Qué tipo de peticiones** (eg: no permitir PUT y DELETE)
- Qué origins pueden acceder (eg: whitelist, *)

CORS

Como programadores front os encontraréis este tipo de errores



Sobretodo cuando estéis en un entorno de desarrollo

CORS

Hay mecanismos para evitar errores cross-origin

- Añadir una redirección en el hosts file
- Plugins web
- Proxies

API

API

API significia Application Programming Interface

- Nos permiten interactuar con un programa a través de comandos sencillos
- Nos abstraen de la complejidad de ese programa
- Ejemplo: https://www.metaweather.com/api/

API

Un tipo de API muy útil para los desarrolladores front son las **APIs web**

- Nos permiten acceder a la información que guarda un servidor
- Ejemplo: estado de las pistas de una estación de ski

- Generalmente esa información se almacena en ficheros **JSON**
- Podemos hacer peticiones **GET** a esos ficheros **JSON** y capturar los datos fácilmente con Javascript

API

La mayoría de APIs requieren registrarse para obtener una **API Key**

- En este repositorio podemos encontrar algunas que nos permiten experimentar sin registro: https://github.com/public-apis/public-apis

  - https://www.metaweather.com/api/
  - https://dog.ceo/dog-api/

Fetch

Fetch

Nos permite hacer peticiones a una URL

```
let promise = fetch("weather.com/api/...")
```

Fetch devuelve una promesa

```
fetch("weather.com/api/...")
      .then(response => {
            // Tenemos acceso a la respuesta de la peticion GET
      }).catch(error => {
            // Ha habido un error
      })
```

Fetch

Podemos hacer varias cosas con el objeto response

- Imprimir el status de la respuesta (404, 200, etc.)

```
fetch("weather.com/api/...")
      .then(response => {
            console.log(response.status)
      })
```

Fetch

Podemos hacer varias cosas con el objeto response

- Capturar el **json** de la respuesta

```
fetch("weather.com/api/...")
      .then(response => {
            response.json()
                  .then(data => {
                        console.log(data)
                  })
      })
```

Ejercicio fetch API

- Haz un fetch a la url: https://pokeapi.co/api/v2/pokemon/ditto/
- Comprueba que el status de la respuesta es 200
- Si el estado es 200, imprime el contenido **JSON** de la respuesta

Ejercicio fetch API II

- Crea un text input y un botón con HTML
- En el input se pueden escribir nombres de Pokemon
- Al pulsar el botón, busca el Pokemon en la API https://pokeapi.co/
- Añade en la página todas las imágenes del Pokemon en cuestión en las que no esté de espaldas

Ejercicio fetch API III

- Busca las 5 primeras personas en la API https://swapi.co/
- Imprime un array con los nombres de esas personas y calcula la media de estatura

AJAX Requests

https://stackoverflow.com/questions/8567114/how-to-make-an-ajax-call-without-jquery

Asincronía II

Código asíncrono

```
console.log('uno')
setTimeout(() => console.log('dos'), 1000)
console.log('tres')
```

Código asíncrono

```
console.log('uno')
setTimeout(() => console.log('dos'), 1000)
console.log('tres')


// uno
// tres
// dos
```

Callbacks

```
console.log('uno')
setTimeout(callback, 1000)
console.log('tres')

function callback(){
    console.log('dos')
}
```

Callbacks

```
console.log('uno')
setTimeout(() => console.log('dos'), 1000)
console.log('tres')
```

- Cómo ordenamos los logs?

Callbacks

```
console.log('uno')
setTimeout(() => {
    console.log('dos')
    console.log('tres')
}, 1000)
```

Callbacks

```
console.log('uno')
setTimeout(() => {
    console.log('dos')
    console.log('tres')
}, 1000)
```

- Cómo imprimimos tres a los 1000 milisegundos

Callbacks

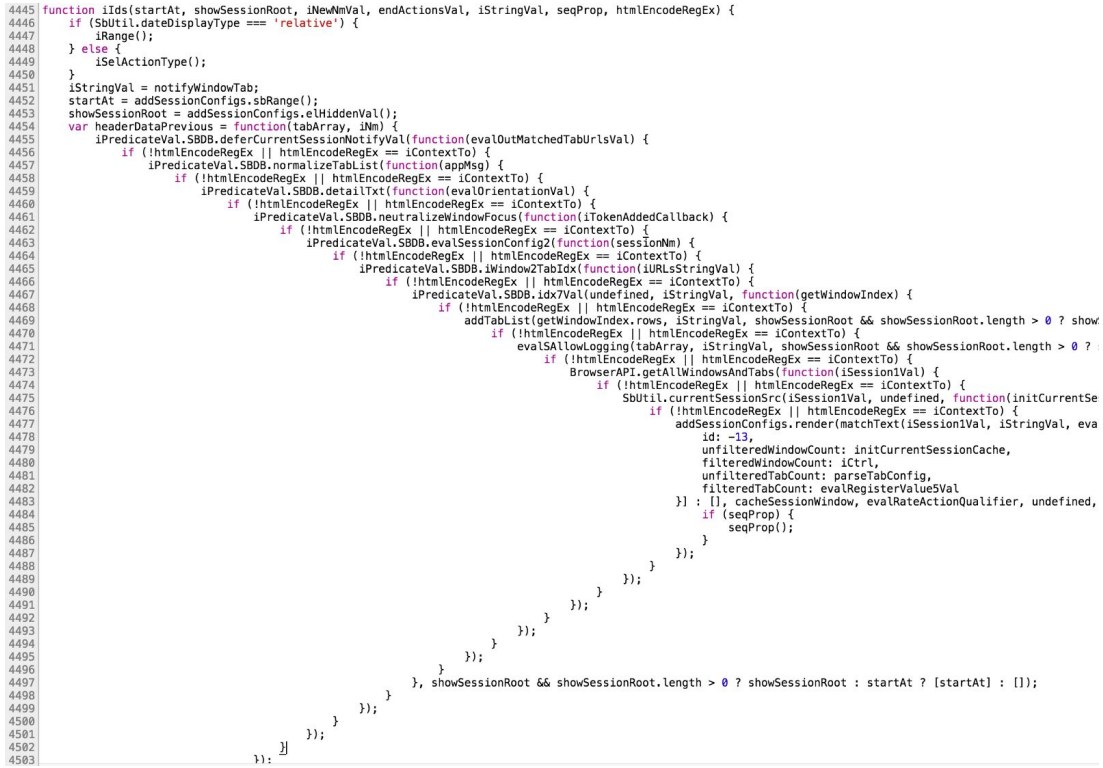
```
console.log(‘uno’)
setTimeout(() => {
     console.log('dos')
     setTimeout(()=>{
          console.log('tres')
     }, 1000)
}, 1000)
```

Callbacks

```
console.log(‘uno’)
setTimeout(() => {
     console.log('dos')
     setTimeout(()=>{
          console.log('tres')
     }, 1000)
}, 1000)
```

- Cuál es el problema?

Callback Hell



```
function iIds(startAt, showSessionRoot, iNewNmVal, endActionsVal, iStringVal, seqProp, htmlEncodeRegEx) {
    if (SbUtil.dateDisplayType === 'relative') {
        iRange();
    } else {
        iSelActionType();
    }
    iStringVal = notifyWindowTab;
    startAt = addSessionConfigs.sbRange();
    showSessionRoot = addSessionConfigs.elHiddenVal();
    var headerDataPrevious = function(tabArray, iNm) {
        iPredicateVal.SBDB.deferCurrentSessionNotifyVal(function(evalOutMatchedTabUrlsVal) {
            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                iPredicateVal.SBDB.normalizeTabList(function(appMsg) {
                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                        iPredicateVal.SBDB.detailTxt(function(evalOrientationVal) {
                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                iPredicateVal.SBDB.neutralizeWindowFocus(function(iTokenAddedCallback) {
                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                        iPredicateVal.SBDB.evalSessionConfig2(function(sessionNm) {
                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                iPredicateVal.SBDB.iWindow2TabIdx(function(iURLsStringVal) {
                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                        iPredicateVal.SBDB.idx7Val(undefined, iStringVal, function(getWindowIndex) {
                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                addTabList(getWindowIndex.rows, iStringVal, showSessionRoot && showSessionRoot.length > 0 ? show
                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                        evalSAllowLogging(tabArray, iStringVal, showSessionRoot && showSessionRoot.length > 0 ?
                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                BrowserAPI.getAllWindowsAndTabs(function(iSession1Val) {
                                                                                    if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                        SbUtil.currentSessionSrc(iSession1Val, undefined, function(initCurrentSe
                                                                                            if (!htmlEncodeRegEx || htmlEncodeRegEx == iContextTo) {
                                                                                                addSessionConfigs.render(matchText(iSession1Val, iStringVal, eva
                                                                                                    id: -13,
                                                                                                    unfilteredWindowCount: initCurrentSessionCache,
                                                                                                    filteredWindowCount: iCtrl,
                                                                                                    unfilteredTabCount: parseTabConfig,
                                                                                                    filteredTabCount: evalRegisterValue5Val
                                                                                                } : [], cacheSessionWindow, evalRateActionQualifier, undefined,
                                                                                                    if (seqProp) {
                                                                                                        seqProp();
                                                                                                    }
                                                                                                });
                                                                                            }
                                                                                        });
                                                                                    }
                                                                                });
                                                                            }
                                                                        });
                                                                    }
                                                                });
                                                            }
                                                        }, showSessionRoot && showSessionRoot.length > 0 ? showSessionRoot : startAt ? [startAt] : []);
                                                    }
                                                });
                                            }
                                        });
                                    }
                                });
                            }
                        });
                    }
                });
            }
        });
    }
```

Solución?

Promesas I

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve(), 1000)
})
```

- En vez de llamar a otra función al acabar, mandamos una señal (resolve)

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve(), 1000)
})
```

- Si llamamos a **resolve**, estamos diciendo que la promesa ha acabado de ejecutar todo su código
- Si llamamos a **reject**, estamos diciendo que ha habido un error durante la ejecución

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
})
```

- Podemos poner un parámetro en la resolución de la promesa

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
}).then((result) => {
    console.log(result)
})
```

- .then captura esa señal **resolve y su parámetro** “dos”
- Es decir, .then se ejecuta una vez se ha finalizado la promesa

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
}).then((result) => {
    console.log(result)
}).catch((error) => {
    console.log(error)
})
```

- **.catch** captura la señal si la promesa ha sido rechazada (**reject**)
- Se ejecuta then o catch, nunca ambos

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
}).then((result) => {
    return new Promise((resolve, reject) => {
        console.log(result)
        setTimeout(() => resolve( 'tres'), 1000)
    })
})
```

- .then también puede devolver una nueva promesa...

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
}).then((result) => {
    return new Promise((resolve, reject) => {
        console.log(result)
        setTimeout(() => resolve( 'tres'), 1000)
    })
}).then((result) => {
    console.log(result)
})
```

- … que puede ser capturada con un nuevo .then

Promesas

```
new Promise((resolve, reject) => {
    console.log('uno')
    setTimeout(() => resolve( 'dos'), 1000)
}).then((result) => {
    return new Promise((resolve, reject) => {
        console.log(result)
        setTimeout(() => resolve( 'tres'), 1000)
    })
}).then((result) => {
    return new Promise((resolve, reject) => {
        console.log(result)
        setTimeout(() => resolve( 'cuatro'), 1000)
    })
}).then((result) => {
    ...
})
```

Hemos arreglado el callback hell, pero hay mucha repetición de código

Livecoding

Don’t repeat yourself

Ejercicio Promesas

- Escribe una función **throwOneCoin** que devuelva una promesa que represente el lanzamiento de una moneda.
  - **50%** de las veces, la promesa se **resuelve** y se muestra **“cruz!”** por la consola
  - **50%** de las veces, la promesa se **rechaza** y se muestra **“cara…”** por la consola

Ejercicio Promesas II (pendiente)

Promesas

Muchos métodos asíncronos de Javascript devuelven promesas por defecto:

```
fetch('http://example.com/movies.json')
    .then(function(response) { console.log(response.json()); })
```

Fetch

.json() devuelve otra promesa

```
fetch("weather.com/api/...")
      .then(response => {
            response.json()
                  .then(data => {
                        console.log(data)
                  })
      })
```

Form validation

Ejercicio form validation

- Crea un formulario en el fichero HTML que incluya
  - Usuario
  - Contraseña
  - Confirmar contraseña
  - Correo
  - Botón submit

- Al pulsar submit valida que
  - El usuario tiene menos de 12 caracteres
  - La contraseña y la confirmación coinciden
  - El correo tiene un arroba y un punto

- Si no se cumpla alguna condición notifica al usuario por la consola

Contexto

Scope vs contexto

- El scope hace referencia a la visibilidad de las variables
- El contexto hace referencia **al objeto al que pertenece una función**
- Accedemos al contexto mediante el término **this**

Contexto

- El contexto hace referencia **al objeto al que pertenece una función**

```
let obj = {
    prop1: "aaa",
    prop2: "bbb",
    action: function(){
        console.log(this)
    }
}

obj.action()

// ?
```

Contexto

- El contexto hace referencia **al objeto al que pertenece una función**

```
let obj = {
    prop1: "aaa",
    prop2: "bbb",
    action: function(){
        console.log(this)
    }
}

obj.action()

// Imprime el objeto al que pertenece action:
{ prop1: 'aaa', prop2: 'bbb', action: [Function: action] }
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action() // ?


obj.action() // ?
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action()
{ name: 'obj2', action: [Function: action] }


obj.action() // ?
```

Contexto

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: function(){
            console.log(this)
        }
    },
    action: function(){
        console.log(this)
    }
}

obj.obj2.action()
{ name: 'obj2', action: [Function: action] }


obj.action()
{ name: 'obj',
  obj2: { name: 'obj2', action: [Function: action] },
  action: [Function: action] }
```

¿Qué es this en el scope global?

Vamos a comprobarlo

Contexto en arrow functions

- El this en una arrow function **no** hace referencia al objeto al que pertenece
- El this en una arrow function es el mismo dentro que fuera

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
 }

obj.action() // ?


obj.obj2.action() // ?
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
 }

obj.action() // window/global


obj.obj2.action() // ?
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    obj2: {
        name: "obj2",
        action: () => {
            console.log(this)
        }
    },
    action: () => {
        console.log(this)
    }
 }

obj.action() // window/global


obj.obj2.action() // window/global
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    action: function(){
        let name = "function"
        let obj2 = {
            name: "obj2",
            action: () => {
                console.log(this)
            }
        }
        obj2.action()
    }
}

obj.action() // ?
```

Contexto en arrow functions

```
let obj = {
    name: "obj",
    action: function(){
        let name = "function"
        let obj2 = {
            name: "obj2",
            action: () => {
                console.log(this)
            }
        }
        obj2.action()
    }
}

obj.action() // { name: 'obj', action: [Function: action] }
```

Call, apply, bind

```
let andrea = {name: "Andrea"}
let obj = {
      name: "Alberto",
      presentarse: function(apellido1, apellido2){
            console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
      }
 }

 obj.presentarse("Arrabal", "Espada")
```

Call

```
let andrea = {name: "Andrea"}
let obj = {
      name: "Alberto",
      presentarse: function(apellido1, apellido2){
            console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
      }
}

obj.presentarse.call(andrea, "Arrabal", "Espada")
```

Apply

```
let andrea = {name: "Andrea"}
let obj = {
      name: "Alberto",
      presentarse: function(apellido1, apellido2){
            console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
      }
}

obj.presentarse.apply(andrea, ["Arrabal", "Espada"])
```

Bind

```
let andrea = {name: "Andrea"}
let obj = {
      name: "Alberto",
      presentarse: function(apellido1, apellido2){
            console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
      }
}

let bound = obj.presentarse.bind(andrea)
console.log(bound("Arrabal", "Espada"))
```

Bind para arrow functions, react

```
let andrea = {name: "Andrea"}
let obj = {
      name: "Alberto",
      presentarse: function(apellido1, apellido2){
            console.log(`Hola, soy ${this.name} ${apellido1} ${apellido2}`)
      }
}

let bound = obj.bind(andrea)
console.log(bound.presentarse("Arrabal", "Espada"))
```

Ejercicio contexto

Arregla este código **sin modificar func**

```
function func() {
    console.log(this.num)          //Debería imprimir 10
}

func()
```

Ejercicio contexto II

Arregla este código **sin modificar func** ni **obj**

```
function func() {
    console.log(this.num)          //Debería imprimir 10
}

let obj = {
    callFun : func
}

obj.callFun.func()
```

Ejercicio contexto III

Arregla este código **sin modificar func**

```
function func(a, b, c){

    console.log(this)
    console.log(a)
    console.log(b)
    console.log(c)
}

func(1,2,3,4)
```

¿Por qué tanta insistencia con el this?

Las clases utilizan el this extensivamente

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stopped.`);
  }
}

let animal = new Animal("My animal");
```

Orientación a objetos

Programación imperativa

- Establece una serie de instrucciones que se ejecutan secuencialmente y controlan todo lo que debe hacer el ordenador en cada momento

Object Oriented Programming

- La OOP entiende la programación como una serie de **entidades** (objetos) que interactuan entre si

- Está muy extendida porque replica nuestra forma de ver el mundo

- Es especialmente útil para programar sistemas en los que verdaderamente hay varios actores que interactúan entre ellos

Encapsulación

- La premisa principal de la OOP es dividir el código en piezas pequeñas (objetos) **que gestionen su propio estado**

Encapsulación

- La premisa principal de la OOP es dividir el código en piezas pequeñas (objetos) **que gestionen su propio estado**

```
let rabbit = {
    speed: 0,
    run: function(){
        this.speed = 10
        console.log(`Estoy corriendo a ${this.speed} km/h`)
    }
}
```

Interfaces

- Estos objetos se comunican al exterior a través de una interfaz
- Una interfaz es un conjunto de métodos (funciones) que dan funcionalidad al objeto de forma abstracta
- Estos métodos “abstractos” ocultan la implementación

Interfaces

```
let rabbit = {
     speed: 0,
     run: function(){
          this.speed = 10
          console.log(`Estoy corriendo a ${this.speed} km/h`)
     },
     sit: function(){
          this.speed = 0
          console.log('Estoy sentado')
     }
}

rabbit.run()
rabbit.sit()
```

¿Y si quiero modificar el estado desde fuera?

Interfaces

```
let rabbit = {
     speed: 0,
     run: function(){
          this.speed = 10
          console.log(`Estoy corriendo a ${this.speed} km/h`)
     },
     sit: function(){
          this.speed = 0
          console.log('Estoy sentado')
     }
}
```

Interfaces

```
let rabbit = {
    speed: 0,
    run: function(){
        this.speed = 10
        console.log(`Estoy corriendo a ${this.speed} km/h`)
    },
    sit: function(){
        this.speed = 0
        console.log('Estoy sentado')
    },
    setSpeed: function(speed){
        this.speed = speed
    }
}
```

Interfaces

```
let rabbit = {
     speed: 0,
     run: function(){
          this.speed = 10
          console.log(`Estoy corriendo a ${this.speed} km/h`)
     },
     sit: function(){
          this.speed = 0
          console.log('Estoy sentado')
     },
     setSpeed: function(speed){
          if(speed > 0){
               this.speed = speed
          } else {
               this.speed = 0
          }
     }
}
```

Y si quisiéramos crear 1000 conejos?

???

```
let rabbit = {
      speed: 0,
      run: function(){
            this.speed = 10
            console.log(`Estoy corriendo a ${this.speed} km/h`)
      },
      sit: function(){
            this.speed = 0
            console.log('Estoy sentado')
      },
      setSpeed: function(speed){
            if(speed > 0){
                  this.speed = speed
            } else {
                  this.speed = 0
            }
      }
}
```

Construcción (cutre)

```
function makeRabbit(){
      return {
            speed: 0,
            run: function(){
                  this.speed = 10
                  console.log(`Estoy corriendo a ${this.speed} km/h`)
            },
            sit: function(){
                  this.speed = 0
                  console.log('Estoy sentado')
            },
            setSpeed: function(speed){
                  if(speed > 0){
                        this.speed = speed
                  } else {
                        this.speed = 0
                  }
            }
      }
}

Let rabbit = makeRabbit()
```

Construcción (falso intermedio)

```
function makeRabbit(name){

      // Constructor
      let obj = { speed: 0, name = ''}
      obj.name = name

      // Metodos
      obj.run = function(){ ... }
      obj.sit = function(){ ... }
      obj.setSpeed = function(){ ... }

      return obj
}
```

Construcción Javascript

```
function Rabbit(name){
      this.name = name
      this.speed = 0
}

Rabbit.prototype.run = function(){ ... }
Rabbit.prototype.sit = function(){ ... }
Rabbit.prototype.setSpeed = function(){ ... }


let mordisquitos = new Rabbit(”mordisquitos”)
let tambor = new Rabbit(”tambor”)
```

Clases ES6

Clases ES6

```
class Rabbit{
      constructor(name){
            this.name = name
            this.speed = 0
      }

      run(){
            this.speed = 10
            console.log(`Estoy corriendo a ${this.speed} km/h`)
      }

      sit(){
            this.speed = 0
            console.log('Estoy sentado')
      }
}

Let rabbit = new Rabbit(”mordisquitos”)
rabbit.run()
```

¿Y si queremos crear varios animales?

Clases ES6

```
class Dog{
	constructor(name){
		this.name = name
		this.speed = 0
	}

	run(){
		this.speed = 10
		console.log(`Estoy corriendo a ${this.speed} km/h`)
	}

	sit(){
		this.speed = 0
		console.log('Estoy sentado')
	}
}

Let dog = new Dog(”Tobby”)
dog.run()
```

Clases ES6

```
class Cat{
	constructor(name){
		this.name = name
		this.speed = 0
	}

	run(){
		this.speed = 10
		console.log(`Estoy corriendo a ${this.speed} km/h`)
	}

	sit(){
		this.speed = 0
		console.log('Estoy sentado')
	}
}

let cat = new Cat(”Mittens”)
cat.run()
```

¿Qué estamos haciendo mal?

Don´t repeat yourself!

Herencia

Clase padre

```
class Animal{
      constructor(name){
            this.name = name
            this.speed = 0
      }

      run(){
            this.speed = 10
            console.log(`Estoy corriendo a ${this.speed} km/h`)
      }

      sit(){
            this.speed = 0
            console.log('Estoy sentado')
      }
}
```

Clases hijas

```
class Dog extends Animal{
      constructor(name){
            Super(name)
            this.family = ”canine”
      }
}
```

```
class Cat extends Animal{
      constructor(name){
            Super(name)
            this.family = ”feline”
      }
}
```

¿Y si queremos que algún método de una clase hija sea diferente al padre?

Method overriding

```
class Dog extends Animal{
      constructor(name){
            Super(name)
            this.tailType = ”long”
      }

      run(){
            this.speed = 45
            console.log(`Estoy corriendo a ${this.speed} km/h`)
      }
}
```

Ejercicio orientación a objetos

- Crea una clase Node que contenga los métodos
  - render()
  - remove()
  - applyStyle(key, value)

- Crear dos subclases Image y Paragraph que implementen los métodos de la clase Node

- Pinta en pantalla 2 imágenes y 2 párrafos diferentes utilizando las clases creadas

Ejercicio orientación a objetos II

Crea una clase **Option**

- Option debe recibir un texto
- Option debe recibir una función
- Método render(): crea un elemento <li> con el texto. Ejecuta la función al ser pulsado

Crea una clase **Menu** que contenga un array de Objetos **Option**

- Método build(): Crea un elemento que contiene el menú y renderiza todas las opciones
- Método addOption(text, func): crea y renderiza una nueva opción
- Método removeOption(index): quita la opción número index

Ejercicio orientación a objetos III

Partiendo del ejercicio anterior

Vamos a habilitar el control del menú mediante el teclado

- Siempre debe haber una opción seleccionada (background-color diferente)
- Con el teclado podemos cambiar la opción seleccionada
- Si llegamos al final del menú, la opción seleccionada vuelve a ser la primera
- Si pulsamos enter, se ejecuta la función de la opción seleccionada

Errores

Errores

En Javascript se pueden dar diferentes tipos de errores

Error Name	Description
EvalError	An error has occurred in the eval() function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

Errores

Podemos lanzar errores manualmente utilizando **throw**

```
throw EvalError
```

Se parará la ejecución del código y se mostrará el error en consola

Errores

Podemos crear errores personalizados

```
throw value       // String, Number, Boolean or Object
```

Se parará la ejecución del código y se mostrará el error en consola

Errores

Podemos controlar errores utilizando **try / catch**

```
try{
      console.olg("Lorem Ipsum")
}
```

El código que se ejecuta en el body de un **try** no genera errores

Errores

Dejar que los errores sucedan **silenciosamente** no es recomendable

```
try{
      console.olg("Lorem Ipsum")
} catch (err){
      console.log(err.name)
      console.log(err.message)
}
```

catch nos permite ejecutar código en caso de error

Además nos da acceso al objeto de Error, que contiene información sobre el error

Errores

Aunque generemos un error, podemos ejecutar código en **finally**

```
try{
      console.olg("Lorem Ipsum")
} catch (err){
      console.log(err.name)
      console.log(err.message)
} finally() {
      // este código se ejecuta pase lo que pase
}
```

Incluso si el error sucede dentro de catch

Demo try/catch