CLASS(E)

Módulo 1. Fundamentos

Antes de empezar

Qué necesitas

Un editor de texto.

Ejecutar código JavaScript:

- Node (recomendado)
- Chrome

let vs. const

Variables

En JavaScript declaramos variables utilizando `let`.

```
let name = "Homer"
```

Variables

Podemos reasignar el valor de una variable.

```
let name = "Homer"
name = "Fry"

console.log(name) // "Fry"
```

Variables

En Javascript declaramos constantes utilizando const.

```
const name = "Homer"
```

Variables

NO podemos reasignar constantes.

```
const name = "Homer"
name = "Fry"

console.log(name) // Error!
```

Errores

Los errores bloquean la ejecución del código.

- Excepto en bloques try/catch
- Excepto en promesas

Constantes

Utilizaremos const siempre que no planeemos reasignar valores.

var

```
var name = "Homer"
```

Inicialmente, JavaScript solo permitía declarar variables con var.

var introduce el problema del hoisting.

Hoisting

Hoisting

¿Qué imprime esto?

```
function valueLogger() {
  console.log('primer log:', value) // ?
  var value = 12
  console.log('segundo log:', value)
}

valueLogger()
```

Hoisting

El hoisting mueve la declaración de las variables a la parte superior del scope, y las declara sin valor, por lo que su valor será undefined

Función equivalente →

```
function valueLogger() {
  var value
  console.log('valor: ', value) // undefined
  value = 12
  console.log('valor: ', value)
}

valueLogger()
```

Hoisting

Las variables let no tienen hoisting.

```
function valueLogger() {
  console.log('valor: ', value) // ?
  let value = 12
  console.log('valor: ', value)
}

valueLogger()
```

var

¿Qué devuelve esta función?

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

var

¡Las variables var tienen ámbito de función!

```
function myLoop() {
  for (var i = 0; i <= 10; i++) {
    // no-op
  }
  return i // 11
}
```

let

¡Las variables let tienen ámbito de bloque!

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

Ejercicio variables

Esta función genera una lista de 10 funciones, que llamando a cada una, debería proporcionarnos en consola el valor de `i` cuando se generó.

Pero no lo está haciendo, encuentra y arregla el bug.

```
function createIndexLoggers() {
  let list = []

  for (var i = 0; i< 10; i++) {
    list.push(function() {
     console.log(i)
    })
  }

  return list
}

const loggers = createIndexLoggers()
const firstLogger = loggers[0]
const secondLogger = loggers[1]


firstLogger() // ?
secondLogger() // ?
```

Ejercicio variables II

Esta función genera un número random en base a un valor que proporcionamos y cierto compartimiento interno, pero al ejecutarla, recibimos un error.

Encuentra y arregla el bug.

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

randomNumber(2) // Error!
```

¿Qué es el scope?

Es el acceso que tenemos a variables y funciones en una zona específica de nuestro código.

- Las variables definidas en un scope comparten su tiempo de vida.
- **Global scope:**
  Las variables aquí definidas son accesibles desde cualquier parte de nuestro código.
- **Local scope:**
  Son los scopes generados por debajo del scope global, se pueden anidar, y las variables definidas en un scope local tienen prioridad sobre los scopes superiores.

```
Global scope

const MAX_NUMBER = 5

Local scope
function globalFunction(number) {

  Local scope
  if (number > MAX_NUMBER) {
      return false
  }

  return true
}
```

Tipos de scopes

- **Global scope:**
  Todas las variables declaradas aquí son visibles desde cualquier parte de nuestro código.

- **Function scope:**
  Un tipo de scope local, que se genera al lanzar una función.

- **Block scope:**
  Otro tipo de scope local, que se genera al ejecutarse un if, un for, y cualquier otro código encerrado entre corchetes { y }.

Global scope

```
const MAX_NUMBER = 5

Function scope
function globalFunction(number) {

  Block scope
  if (number > MAX_NUMBER) {
      return false
  }

  return true
}
```

Ejemplo I

```
let name = 'Lorem'
function logName(){
    let name = 'Ipsum'
    console.log(name)
}
logName()
console.log(name)
// ?
// ?
```

Ejemplo I

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

Ejemplo II

```
let name = 'Lorem'
function logName(){
    name = 'Ipsum'
    console.log(name)
}
logName()
console.log(name)
// ?
// ?
```

Ejemplo II

```
let name = 'Lorem'
function logName(){
    name = 'Ipsum'
    console.log(name)
}
logName()
console.log(name)
// Ipsum
// Ipsum
```

Scope bloques if, for, etc.

Los bloques if, for y switch también crean scopes locales si usamos `let`.

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

console.log(name) // Error!
```

Scope bloques if, for, etc.

Accesible por hoisting.

Aunque parezca que hemos “arreglado” el error, en realidad hemos hecho nuestro código un poco más anárquico. Estamos accediendo a un valor declarado en un scope ajeno, y este comportamiento escalado en cualquier base de código puede traer comportamientos indeseados.

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

console.log(name) // Lorem
```

¿Qué devuelve esta función?

```
function myFunc() {
  let a = 1
  let b = 0

  for (let i = 4; i--;) {
    let b = a + 1
  }

  return b
}
```

¿Y ahora?

```
function myFunc() {
  let a = 1

  for (let i = 4; i--;) {
    let b = a + 1
  }

  return b
}
```

¿Y ahora?

```
function myFunc() {
  let a = 1

  for (let i = 4; i--;) {
    let a = a + 1
  }

  return a
}
```

¿Por qué JS limita el scope?

Es una buena práctica tener acceso sólo 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 crean antes o después, siempre que estén en el scope adecuado. Una variable declarada con `function` como si fuera una variable declarada con `var`.

Conclusión

- Utiliza const siempre que puedas
- Utiliza let para los demás casos
- var es problemático y está obsoleto

Este es un “proceso” que nos ayuda a que nuestras variables mantengan un comportamiento de acceso seguro.

Estos tipos de procesos se llaman **patrones de implementación**, nos ayudan a automatizar las decisiones más pequeñas, para centrarnos en los problemas importantes.

Tipos de datos

Tipos de datos primitivos

JavaScript ofrece 6 tipos de datos primitivos:

- Boolean
- Number
- String
- Symbol
- Null
- Undefined

typeof

El operador typeof nos informa del tipo de dato.

```
console.log(typeof 5) // "number"
console.log(typeof "5") // "string"
```

typeof

¿En este caso?

```
console.log(typeof undefined) // ?
```

typeof

¿En este caso?

```
console.log(typeof undefined) // undefined
```

typeof

¿En este caso?

```
console.log(typeof null) // ?
```

typeof

¿En este caso?

```
console.log(typeof null) // ?
```

Igualdad

Igualdad

En los valores primitivos, la igualdad se compara por valor.

```
let num1 = 5
let num2 = 5

console.log(num1 === num2) // true
```

En los tipos compuestos comparamos por referencia

Truthiness

Todos los valores equivalen a `true` excepto:

- `false`
- `null`
- `undefined`
- `0`
- `NaN`
- `“”`

Ejercicio truthiness

Determina si las siguientes expresiones son truthy o falsy:

- “Abc”
- 20
- “20”
- “0”
- 0
- “”
- “undefined”
- undefined
- null

Igualdad

Loose equality (==)

- Intenta **type coercion**, “convierte” los valores usados a un estado donde se puede encontrar coincidencias entre ellos para determinar esa igualdad.
- Si comparamos un valor tipo Number, con otro tipo String, determinará que son iguales si el texto contiene el mismo valor que el número.
- Si comparamos dos valores falsy, determinará que la igualdad es correcta.

```
77 == "77" // true
0 == false // true
undefined == false // true
```

Igualdad

Strict equality (===)

- Compara igualdad en **valor** y en **tipo**
- Se olvida del type coercion.
- Loose equality comprobamos “que se parezcan”, mientras que con Strict equality comprobamos que sean el mismo valor.
- Preferimos Strict equality, ya que nos permite tener control sobre algo tan sensible como son los datos. Esta preferencia es otro **patrón de implementación**.

```
77 === "77" // false
0 === false // false
undefined === false // false
```

Arrow functions

Arrow functions

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

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

Equivale a:

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

Arrow functions

Especialmente útiles cuando pasamos una función por parámetro:

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

Equivale a:

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

Arrow functions

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

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

Arrow functions

Si la función devuelve un objeto, usamos paréntesis.

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

Hoisting y funciones

- Funciones declaradas con `function` sufren hoisting.

```
const four = double(2) // 4

function double(n) {
     return n * 2
}
```

- Funciones declaradas con `const`, no sufren de hoisting.

```
const four = double(2) // Uncaught
ReferenceError: double is not defined

const double = (n) => n * 2
```