

# Laboratorio 0 – Introducción a JavaScript

Tecnología de los Sistemas de Información en la Red



# Índice

---

1. Introducción
2. Objetivos
3. Herramientas
4. El lenguaje JavaScript
5. Tipos de datos primitivos
6. Tipos estructurados
7. Funciones
8. Ámbito
9. Contexto de ejecución
10. Errores



# Índice de ejercicios

| Objetivo                  | Página  |
|---------------------------|---|
| Tipos dinámicos           | <a href="#"><u>18</u></a>   |
| Tipificación débil        | <a href="#"><u>21</u></a>   |
| El tipo <b>undefined</b>  | <a href="#"><u>24</u></a>   |
| Problemas de tipificación | <a href="#"><u>29</u></a>   |
| Coerción de tipos         | <a href="#"><u>32</u></a> , <a href="#"><u>34</u></a> , <a href="#"><u>36</u></a> , <a href="#"><u>37</u></a> |
| Objetos                   | <a href="#"><u>48</u></a>   |
| Propiedades de objetos    | <a href="#"><u>52</u></a>   |
| Uso de funciones          | <a href="#"><u>58</u></a>   |
| Implantación de funciones | <a href="#"><u>66</u></a> , <a href="#"><u>67</u></a>   |
| La notación flecha (=>)   | <a href="#"><u>69</u></a>   |
| Let / var                 | <a href="#"><u>75</u></a>   |
| Let                       | <a href="#"><u>81</u></a>   |
| Clausuras                 | <a href="#"><u>84</u></a>   |



# I. Introducción

---

- ▶ Javascript es un lenguaje de programación ampliamente utilizado en sistemas distribuidos.
- ▶ Está basado en la especificación ECMAScript.
  - ▶ Utilizaremos ECMAScript 6 en este curso.
- ▶ En TSR, JavaScript es una herramienta apropiada para lograr varios objetivos pedagógicos.
  - ▶ Pero no estamos interesados en un aprendizaje en profundidad de este lenguaje ni en todas sus bibliotecas.



## 2. Objetivos

- ▶ Aprender un conjunto básico de características de JavaScript que permitan abordar fácilmente el Tema 2...
  - ▶ ...y empezar la Práctica I.
- ▶ Entender que Javascript es un lenguaje de programación *extraño*.
  - ▶ No es similar a Java.
  - ▶ Necesita un intérprete (en vez de un compilador).
  - ▶ Es difícil de aprender en poco tiempo si algunos de los conceptos básicos no han sido explicados con detenimiento.
- ▶ Presentar el conjunto de herramientas a utilizar en los laboratorios.
  - ▶ Estas semanas iniciales tendrían que ser dedicadas al aprendizaje de esas herramientas.



## 2. Objetivos

---

- ▶ Entender correctamente los mensajes de error.
  - ▶ Algunos mensajes resultan a veces imprecisos.
    - ▶ Pero deberíamos revisarlos con cuidado pues proporcionan pistas sobre las causas de cada error.
    - ▶ Esas pistas tendrían que ser entendidas adecuadamente.
    - ▶ Trataremos de proporcionar una primera guía en esta presentación.



## 3. Herramientas

- ▶ Todas estas herramientas se encuentran ya instaladas en las virtuales de portal
  - ▶ Consulta introLaboTSR, en PoliformaT
- ▶ Necesitamos, al menos, estas dos herramientas:
  - ▶ Un editor de texto
    - ▶ Para escribir nuestros programas y modificarlos.
    - ▶ Hay una gran variedad de editores con algunas características útiles:
      - ☐ Apoyo en la depuración
      - ☐ Resalte sintáctico
      - ☐ Personalización
      - ☐ Colecciones de complementos (*plugins*)
      - ☐ Documentación sobre la API
      - ☐ Control de versiones
  - ▶ Un intérprete
    - ▶ Para ejecutar nuestros programas
    - ▶ Utilizaremos NodeJS con su orden "node"



## 3.1. Editor de texto

---

- ▶ Utilizaremos *Visual Studio Code* en los laboratorios
  - ▶ Es un editor multiplataforma con un conjunto amplio de características útiles
  - ▶ Puede descargarse desde <https://code.visualstudio.com/download>
  - ▶ Su documentación está disponible en <https://code.visualstudio.com/docs>









## 3.2. Intérprete

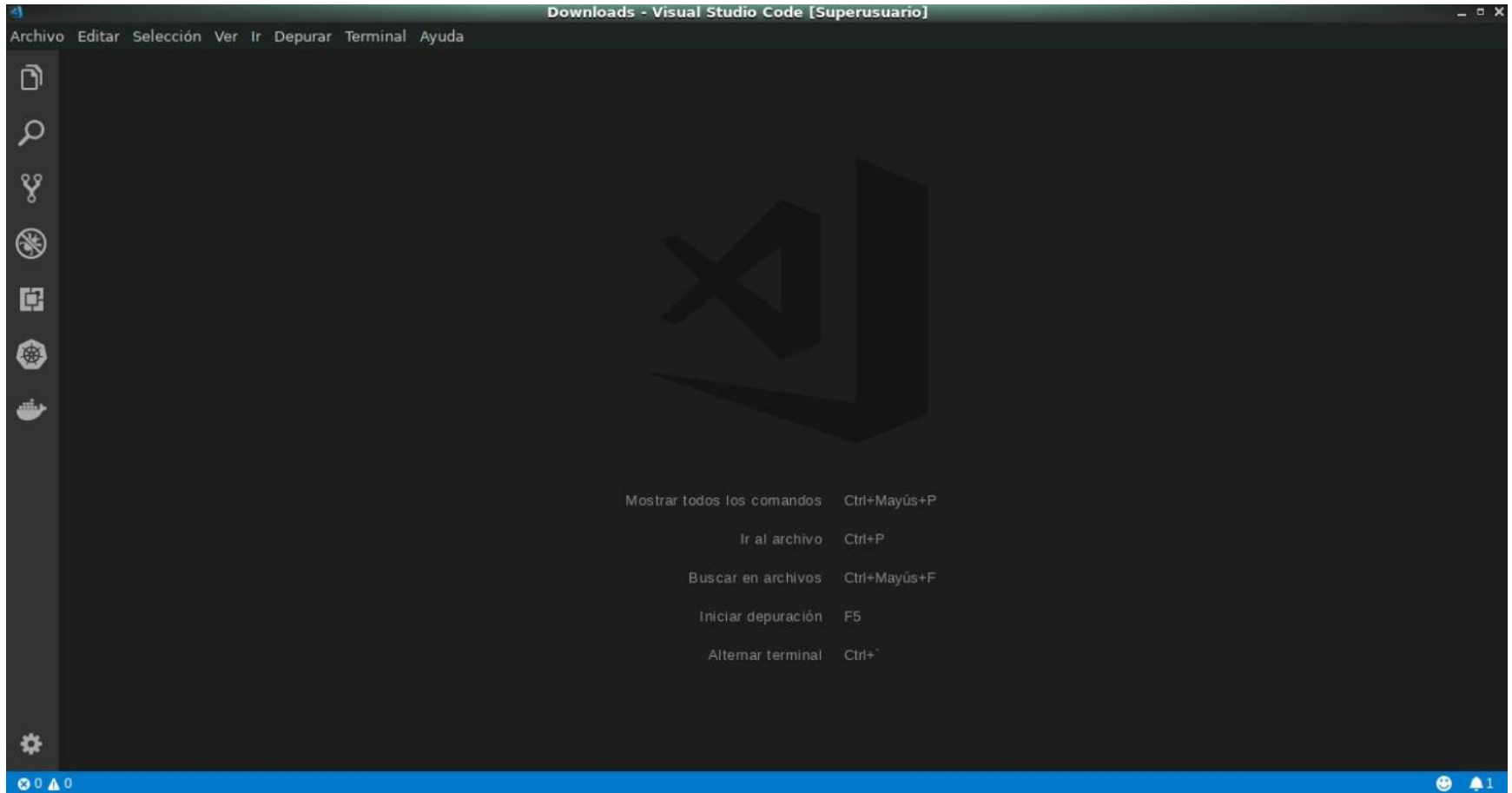
- ▶ Utilizaremos *NodeJS*
  - ▶ Es el intérprete que será utilizado en el aula y en los laboratorios
  - ▶ Puede descargarse de <https://nodejs.org/en/download/>
    - ▶ La versión instalada en los laboratorios es la 20
  - ▶ La documentación está disponible en <https://nodejs.org/en/docs/>
    - ▶ En caso del API, hay que especificar la versión 20 (<https://nodejs.org/docs/latest-v20.x/api/index.html>)
  - ▶ Aunque puedas encontrarlo instalado en otras plataformas, únicamente en las virtuales de portal (y en la imagen para VirtualBox) dispone de toda su funcionalidad.

## 3.3. Utilizando VS Code

- ▶ Visual Studio Code (VS Code) es un editor de texto que gestiona ficheros, carpetas o proyectos.
- ▶ Estructura sus elementos de manera sencilla.
  - ▶ Hay varios iconos en un panel pequeño a la izquierda:
    - ▶ Explorador de ficheros (  ): Para acceder a ficheros o carpetas.
    - ▶ Búsqueda (  ): busca cualquier texto en el fichero actual.
    - ▶ Control de versiones (  ): Para integrar nuestro proyecto en un sistema de control de versiones.
    - ▶ Depuración (  ): facilita la depuración de nuestro programa al utilizar puntos de ruptura o explorando los valores actuales de las variables.
  - ▶ Cada uno de estos iconos muestra un menú con varias acciones relacionadas.

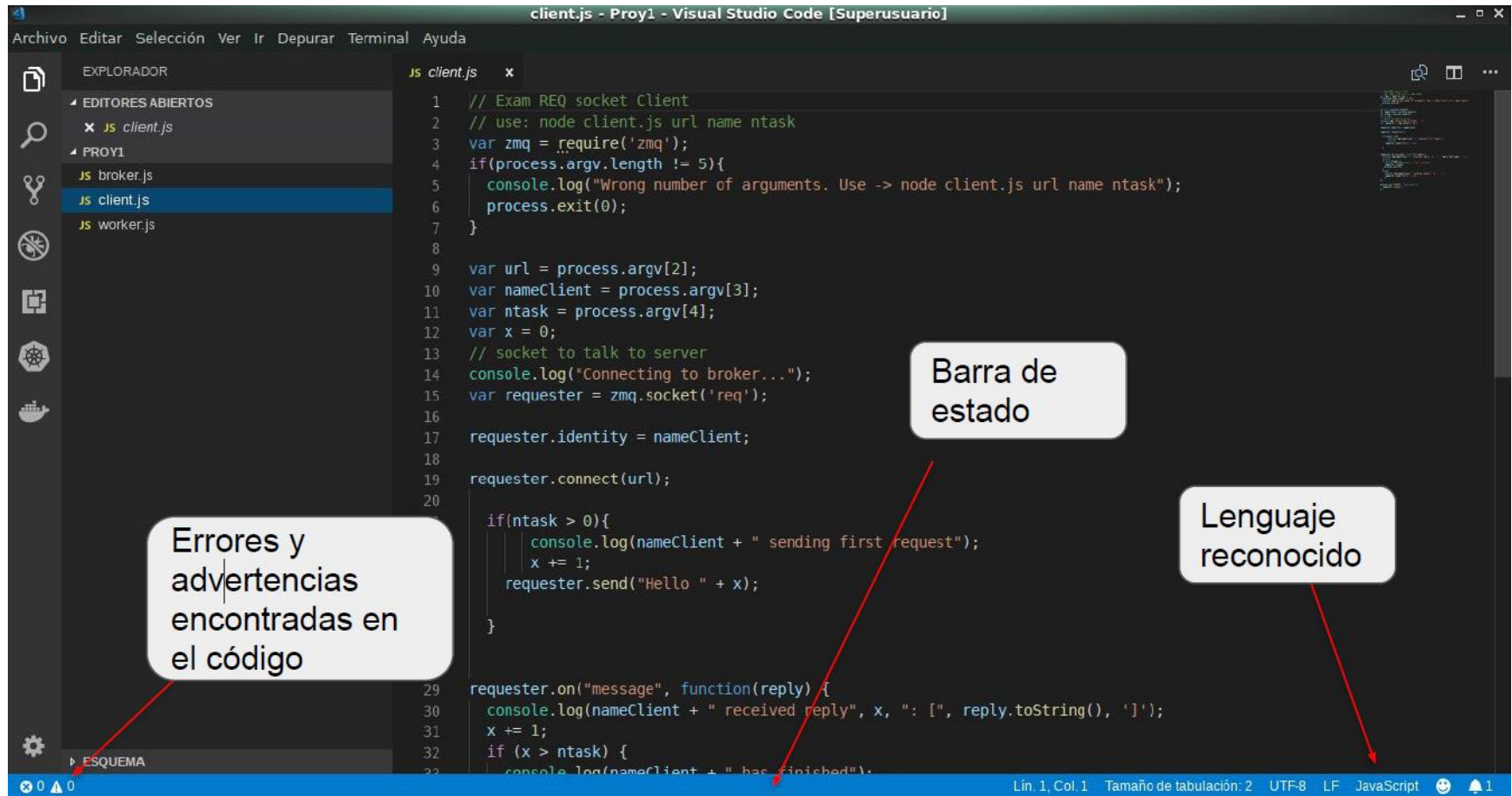


## 3.3. Utilizando VS Code





## 3.3. Utilizando VS Code





## 3.3. Utilizando VS Code

---

- ▶ Cada fichero con la extensión ".js" se identifica como programa JavaScript
  - ▶ Su sintaxis se destaca adecuadamente
- ▶ Para ejecutar un programa en Javascript, podemos utilizar...
  - ▶ El intérprete **node** en una terminal separada, o
  - ▶ La terminal incluida en VS Code (Ctrl + `)
    - ▶ Como se muestra en la próxima página



## 3.3. Utilizando VS Code

The screenshot shows the Visual Studio Code editor interface. The main editor window displays the file `client.js` with the following code:

```
1 // Exam REQ socket Client
2 // use: node client.js url name ntask
3
4 var zmq = require('zmq');
5 if(process.argv.length != 5){
6     console.log("Wrong number of arguments. Use -> node client.js url name ntask");
7     process.exit(0);
8 }
9
10 var url = process.argv[2];
11 var nameClient = process.argv[3];
12 var ntask = process.argv[4];
13 var x = 0;
14 // socket to talk to server
15 console.log("Connecting to broker...");
16 var requester = zmq.socket('req');
17
18 requester.identity = nameClient;
19
20 requester.connect(url);
```

The left sidebar shows the Explorer view with the file `client.js` selected. The bottom panel shows the Terminal view with the following output:

```
[root@TSR-mgonzale-1819 Proy1]# node client.js
Wrong number of arguments. Use -> node client.js url name ntask
[root@TSR-mgonzale-1819 Proy1]#
```

The status bar at the bottom indicates the current line and column (Lin. 12, Col. 5), the tabulation size (Tamaño de tabulación: 2), the encoding (UTF-8), the line ending (LF), and the language (JavaScript).



## 3.3. Utilizando VS Code

---

- ▶ Algunas operaciones disponibles en VS Code:
  - ▶ Dividir las ventanas en dos mitades verticales, para comparar dos ficheros.
  - ▶ Mantener un histórico de los ficheros editados, recordando la última posición editada en cada fichero.
    - ▶ El histórico puede recorrerse hacia...
      - Detrás (Ctrl + Alt + -)
      - Delante (Ctrl + Mays + -)



## 3.4. Utilizando el intérprete

- ▶ Una vez instalado, podemos utilizar el intérprete de NodeJS con esta orden:

```
node program.js [lista de argumentos]
```

- ▶ ...donde:
  - ▶ La extensión ".js" no es obligatoria.
  - ▶ Los argumentos del programa, si existen, deberán escribirse tras su nombre.



## 4. El lenguaje JavaScript

- ▶ Algunas características de este lenguaje son:
  - ▶ Tipos dinámicos
  - ▶ Tipificación débil
  - ▶ Tipos primitivos
  - ▶ Coerción de tipos
- ▶ Asumiremos este fragmento de código para evaluar esas características:

```
1  let x=6  /* Replace 'let' with 'var' and try again the statements in lines 5 or 6. */
2  console.log(x)
3  x = "Hello"
4  console.log(x)
5  // let x  /* Can this be done?      */
6  // var x  /* Can this be done here? */
7  console.log(x)
8  x = []
9  console.log(x)
10 x[1] = 0
11 console.log(x)
12 console.log(x[x[0]])
```



## 4. El lenguaje JavaScript

---

- ▶ Algunas **cuestiones** sobre el programa de la página anterior:
  - ▶ ¿Habrá algún error en la ejecución de ese programa?
  - ▶ ¿En qué se diferencia de un programa escrito en Java?
  - ▶ ¿Podemos utilizar una variable que no haya sido definida con **"let"** o **"var"**?



## 4.1. Tipos dinámicos

- ▶ JavaScript utiliza tipos dinámicos
  - ▶ No necesita especificar el tipo de las variables
    - ▶ Su tipo depende de los valores asignados
  - ▶ Hay cierta libertad al acceder a las componentes de un vector
    - ▶ Podría ocurrir que no tuvieran ningún valor asignado
      - ¡Pero eso no genera ningún error!
      - ¡Devolverán "undefined"!
- ▶ Los tipos dinámicos pueden ser útiles cuando se utilicen adecuadamente
  - ▶ ¡Pero también pueden ser una fuente de errores!!
- ▶ **Tipos dinámicos:** las variables cambian su tipo según el valor asignado y este valor puede cambiar mientras el programa se ejecute.

## 4.2. Tipificación débil

- ▶ JavaScript es un lenguaje de programación con tipificación débil
  - ▶ Esto significa que sus expresiones no comprueban semánticamente si sus operadores están aplicados a variables con un tipo adecuado.
  - ▶ De nuevo, esto es muy flexible...
    - ▶ ...pero es también propenso a errores, como muestra este ejemplo:

```
1 console.log(8*null)    // 0
2 console.log("5" - 1)   // 4
3 console.log("5"+1)     // 51
4 console.log("five"*2)  // NaN
5 console.log("5"*"2")   // 10 ??
6 console.log(5+[1,2,3]) // ??
```



## 4.2. Tipificación débil

- ▶ El ejemplo mostrado en la página anterior...
  - ▶ ...ilustra que se pueden escribir expresiones que combinen valores de tipos diferentes.
    - ▶ JavaScript tiene algunas reglas de evaluación para determinar el tipo resultante.
- ▶ Preguntas sobre ese ejemplo:
  - ▶ ¿Hay algún resultado inesperado?
  - ▶ ¿Hay alguna expresión (aparentemente) incorrecta?
    - ▶ Pruébalas todas y comprueba los resultados.
  - ▶ ¿Conviene escribir expresiones similares en nuestros programas?



## 5. Tipos de datos primitivos

---

- ▶ En JavaScript, los tipos de datos primitivos son:
  - ▶ Number
  - ▶ Boolean
  - ▶ String
  - ▶ undefined
  - ▶ null
    - ▶ A pesar de que pueda ser considerado un valor, ECMAScript lo considera un tipo
  - ▶ Symbol
    - ▶ No lo consideraremos en esta presentación
- ▶ Un tipo de dato es **primitivo** cuando es sencillo (es decir, no estructurado) y hay una manera para obtener el tipo de una variable que pertenece a él (operador typeof)

## 5.1. Tipos de datos primitivos: *undefined*

- ▶ **undefined** es un tipo de dato que corresponde a todas aquellas variables a las que no se ha asignado todavía un valor
  - ▶ Es decir, corresponde a variables no inicializadas

```
1  let result
2
3  console.log(result)
```

- ▶ **undefined** también se usa cuando un parámetro de función no ha recibido ningún valor en una invocación.
  - ▶ Ejemplo:
    - ▶ Cuando una función con tres parámetros se llama utilizando un único argumento, el segundo y tercer parámetros reciben *undefined*.

## 5.1. Tipos de datos primitivos: *undefined*

- ▶ **undefined** también debería usarse para comprobar si una variable ya tiene algún valor asignado.
- ▶ Para ello tendríamos que utilizar el operador **typeof**, como ilustra este ejemplo:

```
1  let result
2
3  if (typeof result !== "undefined")
4      console.log(result)
5  else console.log("The result is not yet defined!")
```

- ▶ **Ejercicio:**
  - ▶ Buscar otras maneras de comprobar si una variable es *undefined* o no.





## 5.2. Tipos de datos primitivos: *null*

---

- ▶ **null** es un valor mantenido por aquellas variables Object a las que todavía no se les haya asignado un valor.
- ▶ ECMAScript considera a **null** como un tipo de datos primitivo, a pesar de que ha sido tradicionalmente utilizado como valor literal en JavaScript.
  - ▶ no hay ningún conflicto entre estas dos interpretaciones
- ▶ Algunas funciones retornan **null** para indicar que no han encontrado ningún objeto apropiado.

## 5.3. Tipos de datos primitivos: *Number*

- ▶ **Number** es un tipo que corresponde tanto a números enteros como reales.
  - ▶ Los reales tienen una precisión limitada debido a su formato de coma flotante
  - ▶ Ejemplo:

```
1  let x=0.2
2  let y=0.29999999999999999
3
4  if (x+y==0.5)
5      console.log("The result is inaccurate.")
```



## 5.3. Tipos de datos primitivos: *Number*

- ▶ **NaN** es un resultado que indica que alguna operación numérica no tuvo sentido.
  - ▶ Ejemplos de operaciones de esta clase:
    - ▶ **0/0**
      - Aun así, las operaciones **valor/0** no devuelven NaN, sino **Infinity**.
    - ▶ **Infinity – Infinity**
    - ▶ Operaciones matemáticas donde **undefined** se ha utilizado como un operando.
    - ▶ Operaciones matemáticas que usan un tipo de operando inapropiado
      - P. ej., "Mi nombre" \* 3
    - ▶ Funciones que esperan un Number como argumento y no reciben un valor de ese tipo
      - P. ej., parseInt("una cadena")



## 5.4. Tipos de datos primitivos: *String*

- ▶ Los objetos **String** tienen por omisión una propiedad: *length*
  - ▶ Mantiene el número de caracteres en la cadena
- ▶ Los literales u objetos **String** pueden concatenarse utilizando el operador **+**

```
1  let s1 = "This is an example."
2  let s2 = "A short sentence. "
3
4  console.log(s1.length)
5  let s3 = s2 + s1
6  console.log(s2.length)
7  console.log(s3.length)
8  console.log(s3)
```

## 5.5. Tipos de datos primitivos: Errores y tipos

- ▶ JavaScript tiene una gestión de tipos débil. A veces esa característica puede ser el origen de errores.

- ▶ Ejemplo:

```
1  let result
2  console.log(result)
3  for(let counter=1; counter<10; counter++)
4      result = result + counter
5  console.log(result)
```

- ▶ Cuestiones:

- ▶ ¿Cuál es el resultado en este ejemplo?
  - ▶ ¿Por qué se obtiene un valor inesperado?

- ▶ **NaN y undefined** pueden ser el origen de muchos errores de programación.

- ▶ Tendríamos que entender por qué esos valores se generan en algunas declaraciones

## 5.6. Tipos de datos primitivos: coerción de tipos

---

- ▶ ¿Qué pasa si una expresión mezcla tipos de datos diferentes?
  - ▶ Como JavaScript es un lenguaje de programación con gestión débil de tipos, no genera errores.
    - ▶ En cambio, aplica algunas reglas para transformar esa expresión en algo que tenga sentido.
    - ▶ Con este fin, algunos operandos se convierten a valores de los tipos de datos esperados.
- ▶ Definición de **Coerción de tipos** (según la Real Academia de Ingeniería):
  - ▶ "Característica de los lenguajes de programación que permite, implícita o explícitamente, convertir un elemento de un tipo de datos en otro, sin tener en cuenta la comprobación de tipos."

## 5.6. Tipos de datos primitivos: coerción de tipos

- Revisemos un ejemplo anterior:

```
1 console.log(8*null)      // 0
2 console.log("5" - 1)     // 4
3 console.log("5"+1)       // 51
4 console.log("five"*2)    // NaN
5 console.log("5"*"2")     // 10 ??
6 console.log(5+[1,2,3])   // ??
```

- **Cuestiones:**

- ¿Se puede deducir qué reglas dirigen las coerciones de tipos aplicadas en este ejemplo?
- ¿Ofrecen todas un resultado "correcto"?
- ¿Son útiles?

## 5.6. Tipos de datos primitivos: coerción de tipos

- ▶ Las reglas de la coerción de tipos se aplican también a expresiones lógicas.
- ▶ Comprueba algunos ejemplos (con sentencias **if**):
  - ▶ ¿Es la cadena literal "5" mayor que 3?
  - ▶ ¿Es una variable con valor "6" igual a 6?
  - ▶ ¿Es la cadena literal "user" igual a **false**?
  - ▶ ¿Es la cadena vacía ("") igual a **false**?
  - ▶ Verifica qué valor booleano corresponde a **undefined** y **NaN**.
    - ▶ Debido a esto, ¿qué pasa cuando comparamos el valor de una variable con **undefined**?
      - Esto explica por qué **undefined** está considerado un tipo en vez de un valor literal.



## 5.6. Tipos de datos primitivos: coerción de tipos

- ▶ A veces, podemos controlar la coerción de tipos usando algunos operadores que convierten un tipo en otro.
  - ▶ Ejemplos: Boolean(), String(), Number, parseInt(), parseFloat()...

```
1  Number(true)           // Returns 1
2  Number(false)          // Returns 0
3  Number("10")           // Returns 10
4  Number(" 10")          // Returns 10
5  Number("10 20")        // Returns NaN
6  Number("John")         // Returns NaN
7  String(10.6)           // Returns "10.6"
8  String(true)           // Returns "true"
9  parseInt("10.33")      // Returns 10
10 parseInt("10 years")   // Returns 10
11 parseFloat("10")       // Returns 10
12 parseFloat("10.33")    // Returns 10.33
```



## 5.6. Tipos de datos primitivos: coerción de tipos

---

### ▶ Ejercicio:

- ▶ Determinar el resultado de estas operaciones:
  - ▶ Boolean("false")
  - ▶ Boolean(NaN)
  - ▶ Boolean(undefined)
  - ▶ Boolean("undefined")

## 5.6. Tipos de datos primitivos: coerción de tipos

- ▶ La coerción de tipos puede evitarse si utilizamos el operador de comparación estricto ("===") en vez del operador de comparación normal ("==").

- ▶ Ejemplos:

```
1 console.log(null == undefined)    // true
2 console.log(null == 0)            // false
3 console.log("5" == 5)             // true
4 console.log(NaN == NaN)           // false ??
5
6 console.log(null === undefined)   // false
7 console.log("5" === 5)            // false
8 console.log(NaN === NaN)          // false ??
```

## 5.6. Tipos de datos primitivos: coerción de tipos

- ▶ La coerción de tipos puede utilizarse para simplificar condiciones.

- ▶ Ejemplos:

- ▶ Si es una cadena vacía:

```
1  if (user)
2    console.log("User is not an empty string.")
```

- ▶ Si una variable ha sido definida o no:

```
1  if (person)
2    console.log("Person exists and it isn't undefined.")
```

- Cuando *person* sea **undefined** o **null** estas instrucciones funcionarán como se espera.
    - Pero... ¿qué pasará si *person* es 0 o cadena vacía?

- ▶ Ejercicio:

- ▶ ¿Cómo puede comprobarse si *person* ha sido definida, considerando también el valor 0 y la cadena vacía?

## 5.6. Tipos de datos primitivos: coerción de tipos

### ► Soluciones al ejercicio de la página anterior:

#### 1. Utilizando comparación estricta:

```
1 let person
2 if (person || person===0 || person==="")
3   console.log("Person exists!")
```

#### 2. Sin coerción de tipos:

```
1 let person
2 if (person!==null && person !== undefined)
3   console.log("Person exists!")
```

### ► Cuestiones:

- ¿Qué pasa si eliminamos la línea 1?
- En la segunda solución, ¿podría usarse `!=` en vez de `!==` ?



## 6. Tipos estructurados

---

- ▶ JavaScript proporciona muchos tipos estructurados que mantienen varios elementos de tipos primitivos:
  - ▶ **Arrays:** Secuencias de valores que pueden ser accedidas utilizando índices.
  - ▶ **Objetos:** Secuencias de pares clave/valor.
  - ▶ **Colecciones:** Esta clase de tipo estructurado no se utiliza en la asignatura.

## 6.1. Tipos estructurados: *Array*

- ▶ *Array* es un objeto JavaScript similar a una lista...
  - ▶ ...con una propiedad **length**
    - ▶ que devuelve cuántos elementos hay en el *Array*
  - ▶ ...y algunos métodos
    - ▶ `indexOf()`, `pop()`, `push()`, `shift()`, `map()`, `slice()`...
- ▶ Hay documentación sobre *Array* en:
  - ▶ [Mozilla Developer Network](#), [w3schools.com](#),...
- ▶ Ejemplo:

```
1  let users = ["Chloe", "Martin", "Adrian", "Danae"]
2
3  for (let c=0; c<users.length; c++)
4  |   console.log(users[c])
```



## 6.1. Tipos estructurados: *Array*

- ▶ Debido a las características de JavaScript, la inserción de elementos y el acceso a elementos del Array todavía no definidos son tareas que deben gestionarse con cuidado.

```
1  let locations=[]
2  locations[1]="Valencia"
3  console.log(locations[0])    // undefined
4  console.log(locations[20])   // undefined
```



## 6.1. Tipos estructurados: *Array*

- ▶ No podemos copiar un *Array* asignando su "valor" a otra variable:

```
1 let users=["Chloe", "Martin", "Adrian", "Danae"]
2 let newUsers=users
3 newUsers[2]="Maria"
4 console.log(users[2])
```

- ▶ En ese caso, estamos copiando una referencia al objeto *Array*.
- ▶ Por ello, ahora tendremos dos referencias al mismo *Array*.

## 6.1. Tipos estructurados: *Array*

- ▶ No podemos copiar un *Array* asignando su "valor" a otra variable:

```
1  let users=["Chloe", "Martin", "Adrian", "Danae"]
2  let newUsers=users.slice()
3  newUsers[2]="Maria"
4  console.log(users[2])
```

- ▶ En vez de eso, tendríamos que utilizar el método slice().
  - ▶ Si no se utilizan argumentos, slice() devuelve una copia del vector.
  - ▶ Hay dos parámetros opcionales en slice():
    1. El índice del primer elemento a copiar. Si ese argumento es **undefined**, la copia empieza en el índice 0.
    2. Un valor una unidad mayor que el índice del último elemento a copiar. Por omisión, se asume la longitud del vector.

## 6.1. Tipos estructurados: *Array*

- ▶ Para insertar elementos en un Array, debemos utilizar sus posiciones...
  - ▶ Pero eso sobrescribe el contenido previo de esas componentes.
  - ▶ Hay otras operaciones que añaden nuevos elementos al principio o al final del vector, desplazando o manteniendo su contenido previo, respectivamente.
  - ▶ De manera similar, hay otras operaciones para eliminar elementos:

|              | AÑADIR                     | ELIMINAR        |
|--------------|----------------------------|-----------------|
| Al principio | <u>unshift</u> (elem1,...) | <u>shift</u> () |
| Al final     | <u>push</u> (elem1,...)    | <u>pop</u> ()   |

## 6.1. Tipos estructurados: *Array*

- ▶ Hay algunos "pseudoarrays" que en ocasiones deberemos convertir en objetos de tipo *Array*.
- ▶ Para ello podemos utilizar el método *Array.from()*
- ▶ Ejemplo que usa el *pseudoarray arguments*:

```
1  function list() {  
2    |    return Array.from(arguments)  
3  }  
4  let list1 = list(1,2,3)    // [1,2,3]  
5  console.log(list1)
```

- ▶ En ediciones anteriores del estándar ECMAScript la operación *Array.from()* no existía.
  - ▶ En su lugar se utilizaba:  
***Array.prototype.slice.call(arguments)***

## 6.2. Tipos estructurados: *Object*

- ▶ Un objeto es un conjunto no ordenado de pares clave / valor (donde "clave" es equivalente a una "propiedad").
  - ▶ El valor de esas claves o propiedades puede ser cualquier literal de tipos primitivos, función u otro objeto.
  - ▶ Ejemplo:

```
1  let person = { name: "Peter",  
2                  age: 25,  
3                  address: {  
4                      city: "Valencia",  
5                      street: "Tres Cruces",  
6                      number: 12  
7                  }  
8              }  
9  console.log(person)
```

## 6.2. Tipos estructurados: *Object*

- ▶ Los objetos también pueden ser creados de manera dinámica.

- ▶ Ejemplo:

```
1  let person={}
2  person.name="Peter"
3  person.age=25
4  person.address={}
5  person.address.city="Valencia"
6  person.address.street="Tres Cruces"
7  person.address.number=12
8  console.log(person)
```

- ▶ Aun así, la forma estática mostrada en la página anterior es más rápida y común.
- ▶ **¡Cuidado!!** Como la declaración/modificación dinámica es posible, si escribimos incorrectamente el nombre de cualquier propiedad y le asignamos un valor...
  - ❑ No habrá errores...
  - ❑ **...¡pero ese nombre incorrecto creará otra propiedad en el objeto!!**

## 6.2. Tipos estructurados: *Object*

- ▶ Los objetos también pueden ser creados en una

- ▶ Ejemplo:

```
1 let person={}
2 person.name="Peter"
3 person.age=25
4 person.address={}
5 person.address.city="Valencia"
6 person.address.street="Tres Cruces"
7 person.address.number=12
8 console.log(person)
```

Hay una tercera sintaxis para declarar las propiedades de un objeto: pueden definirse entre corchetes! (como en los Arrays)

Por tanto, estas líneas son equivalentes a las del ejemplo:

```
let person={}; let property="street"
person['name']="Peter"; person['age']=25
person['address']={}; person['address']['city']="Valencia"
person['address'][property]="Tres Cruces"
person['address'].number=12
console.log(person)
```

## 6.2. Tipos estructurados: *Object*

- ▶ Los objetos pueden crearse o modificarse dinámicamente, pero...
- ▶ **Ejercicio**: explicar qué pasa cuando accedemos a una propiedad no definida.
  - ▶ Probar estos ejemplos para responder:

```
1 let person={}
2 person.name="Peter"
3 person.age=25
4 console.log(person.district)
```

```
1 function printDistrict(who) {
2     console.log("District: "+who.district)
3 }
4 let person={name:"Peter",
5             age:25,
6             address: {
7                 city:"Valencia",
8                 street:"Tres Cruces",
9                 number:12
10            }
11        }
12 printDistrict(person)
```





## 6.2.1. Tipos estructurados: *Object*. JSON

---

- ▶ JSON (JavaScript Object Notation) es un formato de texto utilizado para "serializar" objetos cuando deban transmitirse por la red.
  - ▶ Cada identificador de propiedad está encerrado entre comillas dobles.
  - ▶ Para gestionar el formato JSON, podemos utilizar...
    - ▶ `JSON.stringify(Objeto)` convierte un objeto de JavaScript en una cadena JSON.
    - ▶ `JSON.parse(String)` convierte una cadena JSON en un objeto JavaScript.

## 6.2.1. Tipos estructurados: *Object*. JSON

### ► Ejemplo:

#### ► Consideremos este programa...

```
1  let person = { name: "Peter",  
2                      age: 25,  
3                      address: {  
4                          city: "Valencia",  
5                          street: "Tres Cruces",  
6                          number: 12  
7                      }  
8  }  
9  console.log(JSON.stringify(person))
```

#### ► Su salida, en formato JSON, es...

```
{"name":"Peter","age":25,"address":{"city":"Valencia","street":"Tres Cruces","number":12}}
```

## 6.2.2. Tipos estructurados: *Object*. Bucles

- ▶ Podemos utilizar un bucle **for(variable in objeto)** para procesar cada propiedad en un objeto dado.

- ▶ Ejemplo:

```
1  let person = { name: "Peter",  
2                      age: 25,  
3                      address: {  
4                          city: "Valencia",  
5                          street: "Tres Cruces",  
6                          number: 12  
7                      }  
8  }  
9  for(let i in person)  
10     console.log("Property "+i+": "+ person[i])
```

- ▶ La variable *i* recibe el nombre de cada propiedad en cada iteración de este bucle.
- ▶ Con ese nombre, podemos acceder al valor de cada propiedad
  - ▶ Así, se intuye que **un objeto es similar a un Array y sus propiedades son los índices de ese Array**.
  - ▶ Esta característica puede usarse cuando los identificadores de las propiedades se mantienen en otras variables.



## 6.2.2. Tipos estructurados: *Object*. Bucles

- ▶ En el ejemplo anterior, la salida obtenida era...

```
Property name: Peter  
Property age: 25  
Property address: [object Object]
```

- ▶ **Ejercicio:**
  - ▶ Extender el ejemplo anterior, mostrando las propiedades y valores de la propiedad "address".
    - ▶ Una solución general para este ejercicio necesita utilizar funciones y estas se explican en la próxima sección.

## 7. Funciones

- ▶ El concepto de "función" es similar al utilizado en cualquier otro lenguaje de programación.
- ▶ Una secuencia de instrucciones que se puede llamar desde otras partes del programa.
- ▶ Una función define una interfaz clara para utilizar ese fragmento de código.

```
1  function product(a,b) {  
2      |   return a*b  
3      |  
4      }  
4      let result=product(4,6)  
5      console.log(result)
```

## 7. Funciones

- ▶ Si una función no utiliza la instrucción "**return**", entonces su ejecución devuelve el valor **undefined**.

```
1  function greet(person) {  
2      |    console.log("Hello, "+person+"!!")  
3  }  
4  console.log(greet("Peter"))
```

## 7. Funciones

- ▶ Los parámetros de las funciones se comportan como variables cuyo ámbito está limitado al código de su función.
- ▶ **Ejecuta** el ejemplo siguiente y observa que...

```
1  function add(x,y,z) {  
2      |    return x+y+z  
3  }  
4  
5  console.log(add(1,2,3))  
6  console.log(add(2,7))  
7  console.log(add(null))  
8  console.log(add(1,2,3,4,5))
```

- a) Cuando una función se llama con menos argumentos de los declarados, entonces los parámetros no utilizados reciben el valor **undefined**.
  - b) Cuando una función se llama con más argumentos de los previstos, entonces los sobrantes son descartados.
- ▶ No habrá ningún error en ambos casos.

## 7. Funciones

- ▶ Si algunos parámetros fueran opcionales, se puede asignar un valor por omisión en su declaración.
- ▶ Así, no necesitamos comprobar si son **undefined** en el cuerpo de la función.

```
1 function add(x=0,y=0,z=0) {  
2     return x+y+z  
3 }  
4  
5 console.log(add(1,2,3))  
6 console.log(add(2,7))  
7 console.log(add(null))  
8 console.log(add(1,2,3,4,5))
```

- ▶ Con ello, las líneas 6 y 7 no muestran **NaN** pues ahora todos los argumentos a procesar serán números enteros.



## 7. Funciones

- ▶ Las funciones que tengan un número desconocido de argumentos pueden utilizar el parámetro "..."
- ▶ Para ello, el identificador del último parámetro tendría que ir precedido por puntos suspensivos, es decir, "...id"
- ▶ Ese parámetro es un *Array* que recoge los argumentos restantes.
- ▶ Ejemplo:

```
1  function add(x=0,y=0,...others) {  
2      let sum=0  
3      if (others.length>0) {  
4          for (let c=0;c<others.length;c++)  
5              sum+=others[c]  
6      }  
7      return x+y+sum  
8  }  
9  console.log(add(5,6,7,8,9))
```



## 7. Funciones

### ► Ejercicio:

- Si asumimos el programa mostrado en la página anterior, ¿cuál será el resultado de la línea siguiente?

```
console.log(add({prop1: 12}, 2, 3))
```

## 7. Funciones

- ▶ Los argumentos se pasan...
  - ▶ Por valor si pertenecen a un tipo primitivo
  - ▶ Por referencia cuando son objetos
    - ▶ Observa que los *Arrays* son también objetos
    - ▶ Podremos cambiar el contenido del objeto, pero no podremos cambiar la referencia recibida.

```
1  function changeColour(car, newColour) {  
2    |   return car.colour = newColour  
3  }  
4  function changeCar(car) {  
5    |   car={brand:"Ferrari", colour:"Red"}  
6  }  
7  let myCar={brand:"Volvo", colour:"Grey"}  
8  console.log(changeColour(myCar,"Blue"))  
9  changeCar(myCar)  
10 console.log(myCar)
```

## 7. Funciones

- ▶ JavaScript maneja las funciones como objetos comunes, por ello pueden ser...
  - ▶ Utilizadas como valores, y asignadas a variables
  - ▶ Utilizadas como argumentos en llamadas a otras funciones
  - ▶ Retornadas como resultado de otras funciones

```
1  function square(x) {return x*x}
2  let a = square
3  let b = a(3)
4  let c = a
5
6  console.log(a)
7  console.log(b)
8  console.log(c)
```

- ▶ Convendría distinguir estos usos de las funciones:
  - ▶ Su definición inicial.
  - ▶ Su utilización en expresiones. Opciones:
    - ▶ Como referencia, cuando sólo se use su identificador.
    - ▶ El resultado de su invocación, al usar paréntesis y dar una lista de argumentos (posiblemente vacía).

## 7. Funciones

### ► Ejemplo:

```
1  function product(a,b) {  
2    |   return a*b  
3  }  
4  function add(a,b) {  
5    |   return a+b  
6  }  
7  function subtract(a,b) {  
8    |   return a-b  
9  }  
10 let arithmeticOperations = [product, add, subtract]  
11 console.log(arithmeticOperations[1](2,3))
```



- ```
1 let arithmeticOperations = [function(a,b) {return a*b},
2 | | | | | function(a,b) {return a+b},
3 | | | | | function(a,b) {return a-b}]
4 console.log(arithmeticOperations[1](2,3))
```



## 7. Funciones

- ▶ Las funciones anónimas son ampliamente utilizadas como argumentos en la invocación a otras funciones.

```
1  function computeTable(n,fn) {  
2      for (let c=1; c<11; c++)  
3          fn(n*c)  
4  }  
5  computeTable(2,function(v){console.log(v)})
```

### ► Notación de flecha

- Las funciones anónimas son ampliamente utilizadas. Tendría sentido una sintaxis más concisa: La "flecha" ( `=>` )
  - No se necesita la palabra **function**
  - Se mantiene la lista de parámetros
    - Los paréntesis son opcionales cuando solo haya un parámetro
  - El operador `=>` sigue a la lista
  - Tras él, una sentencia que calcule el resultado a devolver
    - O un bloque de sentencias entre llaves.

```
1  function computeTable(n,fn) {  
2      |   for (let c=1; c<11; c++)  
3      |       fn(n*c)  
4      |  
5      |   }  
6      computeTable(2,v => console.log(v))
```



## 7. Funciones

- ▶ Por tanto, esta declaración

```
1 double = x => x*2
```

- ▶ ...es equivalente a...

```
1 function double(x){  
2   |   return x*2  
3   |  
3   }
```



## 7. Funciones

### ▶ Ejercicios:

#### ▶ Escribir una función doCheckPasswd() con tres parámetros:

- input
- correctPassword
- func

#### ▶ Esa función:

- ▶ Compara las cadenas pasadas en los primeros dos argumentos.
- ▶ Si son iguales, entonces se llama a la función pasada como tercer argumento.

#### ▶ Pruébala con las llamadas siguientes:

```
1 doCheckPasswd("Erroneous", "Correct",  
2   function() {console.log("access granted")})  
3 doCheckPasswd("Correct", "Correct",  
4   function() {console.log("sending data")})
```

## 7. Funciones

- ▶ **Ejercicio:** Extender el programa mostrado en la página 62, escribiendo otra función `doWithNFirstNumbers()` con 3 parámetros:
  - `n`: El último número natural a utilizar
  - `op`: Función a ser aplicada en cada número natural procesado
  - `op2`: Función a ser aplicada en el resultado de `op(i)` para acumular todos los resultados
    - `op2` ha de ser alguna de las funciones del Array `arithmeticOperations`
- ▶ `doWithNFirstNumbers()` aplica `op()` a todos los números naturales del intervalo `1..n`, y acumula sus resultados utilizando `op2`.
- ▶ Ejemplos de invocaciones:

```
10 // Sum the squares of the first four numbers. Result: 30
11 doWithNFirstNumbers(4, x => x*x, arithmeticOperations[1])
12 // Compute how many odd numbers are in the 1..3 range. Result: 2
13 doWithNFirstNumbers(3, x => x%2?1:0, arithmeticOperations[1])
```

## 7. Funciones

- ▶ Hay muchas funciones que usan otras funciones como argumentos. Por ejemplo:

- ▶ Array.map()

- ▶ Crea un *Array* nuevo con los resultados de la función usada como primer argumento aplicada sobre cada elemento del *Array* original.
- ▶ `map()` llama a esa función con tres argumentos:
  - El elemento en que la función tendría que ser aplicada
  - Su índice
  - El *Array* original

```
1  let numbers=[1,5,10,15]
2  let doubles=numbers.map(x=>x*2)
3  // doubles is now [2,10,20,30]
4  // numbers is still [1,5,10,15]
5  console.log(numbers)
6  console.log(doubles)
```



## 7. Funciones

---

### ▶ Ejercicio:

- ▶ Modificar el ejemplo de la página anterior, utilizando notación de flecha para escribir la función facilitada como argumento de `map()`.



## 8. Ámbito

**NOTA: Esta parte de la presentación será explicada a fondo en el Tema 2**

- ▶ El ámbito de los elementos (variables, funciones...) de un programa está determinado por la ubicación de sus definiciones.
- ▶ Hay dos ámbitos tradicionales en JavaScript:
  - ▶ **Global**
  - ▶ **Función** (también conocido como **local**)
- ▶ Los elementos del **ámbito global** (es decir, los que no hayan sido definidos dentro de alguna función) pueden ser accedidos desde cualquier línea del programa.
- ▶ Por su parte, cada función define su propio ámbito **local**.



## 8. Ámbito

- ▶ Cuando un programa se ejecute, los elementos definidos en un ámbito local pueden accederse desde:
  - ▶ Ese ámbito local
  - ▶ O desde el ámbito de otras funciones colocadas en ese ámbito local → "**children scope**"
- ▶ Esto define una jerarquía de ámbitos.
  - ▶ Cuando un programa ejecuta una secuencia de llamadas a función, esa secuencia define una **cadena de ámbitos**
    - ▶ Determina qué elementos en otros ámbitos externos pueden ser accedidos desde el actual.
    - ▶ Si una función o variable está definida en cualquier elemento de esa cadena y su nombre coincide con alguno del ámbito global, entonces se utilizará el elemento "local" (es decir, de algún ámbito más cercano).

## 8. Ámbito

- ▶ Así, en un programa como...

```
1  function a() {  
2      let a1=1  
3      b(a1)  
4  }  
5  function b(p1) {  
6      let b1=2  
7      console.log(a1)  
8      console.log(b1)  
9      console.log(g11)  
10 }  
11 a()  
12 let g11=0
```

- ▶ ...a1 no puede ser accedido en b(). ¡Se generará un error!!
- ▶ **Cuestión:**
  - ▶ Se puede permitir que b() utilice a1 de dos maneras. ¿Cuáles son?



## 8. Ámbito

### Solución A

```
1  function a() {
2      let a1=1
3      b(a1)
4  }
5  function b(p1) {
6      let b1=2
7      console.log(p1)
8      console.log(b1)
9      console.log(gl1)
10 }
11 a()
12 var gl1=0
```

### Solución B

```
1  function a() {
2      let a1=1
3      b()
4      function b(p1) {
5          let b1=2
6          console.log(a1)
7          console.log(b1)
8          console.log(gl1)
9      }
10 }
11 a()
12 var gl1=0
```

- ▶ A pasa una copia de a1 a b(). Por tanto, b() puede leer a1, pero no la puede modificar.
- ▶ B define b() como función interna a a(). Así, b() puede ver a1. Por tanto, puede tanto leer como escribir en a1.

## 8. Ámbito

- ▶ La palabra **let** tiene su propio ámbito:
  - ▶ Cuando **let** se utiliza en el ámbito global...
    - ▶ No gestiona a las variables o funciones como propiedades del objeto global.
      - La palabra clave **var** gestiona esos elementos como propiedades de ese objeto global.
        - Por tanto, son visibles incluso antes de ejecutar la declaración que los define.
    - ▶ Por tanto, los elementos definidos con **let** son solo visibles desde ese punto en adelante.
  - ▶ Cuando **let** se utiliza en el ámbito local...
    - ▶ JavaScript considera que el ámbito local abarca la función entera que define ese ámbito.
    - ▶ Pero **let** no utiliza un ámbito local de función. En su lugar, define un "ámbito de bloque".
      - Un "bloque" corresponde a un conjunto de instrucciones dentro de un par de llaves.



## 8. Àmbito

### ▶ Ejercicios:

- ▶ Ejecute los programas de la página 73. Compruebe su salida. Reemplace el "**var**" utilizado en la línea 12 por un "**let**". Ejecute otra vez los programas. Explique los nuevos resultados.
- ▶ En los programas originales, intercambie el contenido de las líneas 11 y 12. Ejecute los programas resultantes. ¿Puede explicar los nuevos resultados?
- ▶ Lea la documentación de MDN sobre [let](#), ejecute todos los ejemplos y explique sus resultados.



## 9. Contexto de ejecución

---

- ▶ El contexto de ejecución se crea dinámicamente para proporcionar un contexto válido para el código que se ejecuta actualmente.
- ▶ El contexto de ejecución está compuesto por todos los elementos del ámbito actual.
  - ▶ Contiene todas las variables definidas en el contexto actual (tanto bloque como función) y aquellas accesibles a través de la cadena de ámbitos.



## 9. Contexto de ejecución

---

- ▶ Para definir el contexto actual, se consideran estas etapas:
  - a) Cuando el programa está empezando, sus funciones y variables globales se van creando y asociando al objeto **global**.
    - ▶ También se crea la referencia **this** como sinónimo de **global**.
  - b) Cada vez que se invoque a una función, y antes de empezar su ejecución, se construye el contexto de esa función, incluyendo sus variables locales y parámetros. Definen su ámbito local.
    - ▶ El valor de la referencia a **this** puede cambiar.
    - ▶ Este contexto nuevo se añade a la "pila de contextos de ejecución" y a la cadena de ámbitos.

## 9. Contexto de ejecución

### ► Ejemplo I:

```
1  computeResults(10)
2  function computeResults(x) {
3      let y=formatResults(x)
4      console.log(gl1+" "+y)
5      function formatResults(inp) {
6          return String(inp)
7      }
8  }
9  var gl1="GlobalContext1"
```

- En este ejemplo, a pesar de que la variable gl1 se ha definido al final del programa y computeResults() está definido después de utilizarlo, ambos pueden ser accedidos sin generar errores.
  - A pesar de que el valor para gl1 se desconoce todavía.

## 9. Contexto de ejecución

### ► Ejemplo 2:

```
1  function computeResults(x) {  
2      let y=formatResults()  
3      console.log(gl1+" "+y)  
4      function formatResults() {  
5          return String(x)  
6      }  
7  }  
8  var gl1="GlobalContext1"  
9  computeResults(10)
```

- Este segundo ejemplo utiliza un valor conocido para gl1
- Además, ahora formatResults() no utiliza parámetros...
  - Utiliza el parámetro "x" de la función que la incluye, y que también está en la "pila de contextos de ejecución".



## 9. Contexto de ejecución

### ► Ejemplo 3:

- Se va a escribir un programa que podrá utilizar un *Array* de funciones. Cada función del *Array* va a gestionar la tabla de multiplicar de su posición en el vector.
  - Así, `tables[3]` tendría que ser una función  $f(x)$  que retorne  $x*3$ .
  - Por tanto, `tables[3](2)` tendría que devolver 6.
- Una primera solución (incorrecta) es:

```
1  let tables=[]
2
3  for (var i=1; i<11; i++)
4      tables[i]=x=>x*i
5
6  console.log(tables[5](2))
7  console.log(tables[9](2))
```



## 9. Contexto de ejecución

### ▶ Ejemplo 3 (cont.):

- ▶ Pero este código es incorrecto. Veamos por qué...
  - ▶ ¿Cuándo se añade un nuevo contexto de ejecución a su pila? ¿Cuál es el valor de la variable *i* en ese momento?
- ▶ Una primera solución a este problema la proporciona el ámbito de bloque asociado a **let**.
  - ▶ La línea 4 define su propio ámbito de bloque
    - Cada iteración del bucle define un nuevo ámbito de bloque
      - ...que se deja en la pila de contextos de ejecución al iniciar la iteración
      - ...y se eliminará de la pila al finalizar la iteración.
  - ▶ Si se sustituye la línea 3 utilizando este código: `for (let i=1; i<11; i++)`
  - ▶ **¿Cuál es el resultado del programa en este caso? ¿Por qué?**
- ▶ La palabra **let** se definió en ECMAScript 6
  - ▶ Las especificaciones anteriores solucionaron este problema utilizando clausuras.



## 9. Contexto de ejecución

- ▶ En una **clausura**, una función interna mantiene el contexto de ejecución existente cuando fue creada.
- ▶ **Analiza** este ejemplo y determina cómo se utiliza la pila de contextos de ejecución:

```
1  function createTable(x) {  
2    |    return y=>x*y  
3  }  
4  
5  let table5=createTable(5)  
6  let table10=createTable(10)  
7  
8  console.log(table5(2)) // Shows 10  
9  console.log(table10(2)) // Shows 20
```



## 9. Contexto de ejecución

---

- ▶ En el ejemplo de la página anterior, `table5` y `table10` son clausuras, generadas por `createTable()`.
  - ▶ Ambas *recuerdan* el argumento recibido por `createTable()` y son una función cuyo código depende de ese argumento.
  - ▶ Ambas comparten la misma secuencia de instrucciones, pero mantienen contextos de ejecución diferentes.
    - ▶ En el contexto de ejecución de `table5`, `x` es 5
    - ▶ En el contexto de ejecución de `table10`, `x` es 10



## 9. Contexto de ejecución

---

### ▶ Ejercicio:

- ▶ Reescribir el programa mostrado en la página 80, utilizando clausuras para proporcionar una solución adecuada.
- ▶ Con este fin, se debería reemplazar la línea 4 original con otras líneas que definan una clausura y asignar la función devuelta a la componente del Array "tables".



## 9. Contexto de ejecución

- ▶ En el contexto global, hay un objeto cuyo nombre es **global** (que puede ser accedido también utilizando **this**).
  - ▶ Algunas de las propiedades de dicho objeto son otros objetos que proporcionan información sobre el entorno de ejecución.
- ▶ En NodeJS, una de esas propiedades es el objeto **process**.
  - ▶ Una de sus propiedades es el *Array* **argv**, que representa los argumentos utilizados en la línea de órdenes que inició la ejecución del proceso.

```
1 // First two elements are:
2 // + "node": the name of the interpreter
3 // + program-name: the name of this file
4 // They are discarded in this example!
5 let procArgs = process.argv.slice(2)
6
7 console.log(procArgs)
```



## 10. Errores

---

- ▶ JavaScript es un lenguaje de programación en el que es muy fácil cometer errores y muy difícil detectarlos y corregirlos.
- ▶ Esta sección distingue varios tipos de errores y proporciona algún consejo sobre cómo gestionarlos y/o evitarlos.
  - ▶ Errores sintácticos
  - ▶ Errores semánticos
- ▶ Hay varias referencias (p. ej., [la de MDN](#)) que describen los mensajes de error más frecuentes en JavaScript.



## 10.1. Errores sintácticos

---

- ▶ Los errores sintácticos son muy comunes cuando empezamos a programar en un nuevo lenguaje.
- ▶ Algunas de sus causas habituales son:
  1. Las instrucciones han sido escritas de una manera incorrecta.
  2. Hemos utilizado un identificador que todavía no había sido definido.

## 10.1.1. Instrucciones incorrectas

- ▶ Muchos de estos errores serán detectados por el editor (VS Code, en nuestro caso).

```
1 console.log(vector[2,]));
```

- ▶ Aun así, a veces, la gestión débil de tipos en este lenguaje puede causar que el editor no detecte un error.

```
1 false + [1,2,3] / {};
```



## 10.1.1. Instrucciones incorrectas

- Un caso típico es que falte cerrar algún par de llaves o paréntesis.

```
1 function suma(A){  
2   if (!(A instanceof Array) throw "suma: parameter is not an array"  
3   else return A.reduce(function(x,y){  
4       return x+y;  
5   })  
6 }
```

- De ser así, se generará un mensaje de error. Normalmente se refiere a algún *token* inesperado...

```
vagrant@NodeEx:/vagrant/pruebasTSR$ node errores7.js  
/vagrant/pruebasTSR/errores7.js:4  
    if (!(A instanceof Array ) throw "Invalid parameter";  
                                ^^^^^  
SyntaxError: Unexpected token throw  
    at Object.exports.runInThisContext (vm.js:76:16)  
    at Module._compile (module.js:542:28)  
    at Object.Module._extensions..js (module.js:579:10)  
    at Module.load (module.js:487:32)  
    at tryModuleLoad (module.js:446:12)  
    at Function.Module._load (module.js:438:3)  
    at Module.runMain (module.js:604:10)  
    at run (bootstrap_node.js:394:7)  
    at startup (bootstrap_node.js:149:9)  
    at bootstrap_node.js:509:3
```

## 10.1.2. Identificadores no definidos

- ▶ Esta clase de errores puede ser causada por un identificador escrito incorrectamente.
- ▶ El intérprete es incapaz de encontrar su definición, y genera un *ReferenceError*
- ▶ Incluye el número de línea

```
vagrant@NodeEx:/vagrant/pruebasTSR$ node errores1.js
/vagrant/pruebasTSR/errores1.js:1
(function (exports, require, module, __filename, __dirname) { console.log(sinDef
inir());
  ^
ReferenceError: sinDefinir is not defined
    at Object.<anonymous> (/vagrant/pruebasTSR/errores1.js:1:75)
    at Module._compile (module.js:570:32)
    at Object.Module._extensions..js (module.js:579:10)
    at Module.load (module.js:487:32)
    at tryModuleLoad (module.js:446:12)
    at Function.Module._load (module.js:438:3)
    at Module.runMain (module.js:604:10)
    at run (bootstrap_node.js:394:7)
    at startup (bootstrap_node.js:149:9)
    at bootstrap_node.js:509:3
vagrant@NodeEx:/vagrant/pruebasTSR$ |
```

- ▶ Tenemos que revisar esa línea y corregir el error



## 10.1.2. Identificadores no definidos

- ▶ Hay que advertir que JavaScript distingue entre mayúsculas y minúsculas al escribir identificadores.

```
1  function computeResult(x) {  
2      return x*2  
3  }  
4  
5  console.log(computeresult(15))  
6  console.log(ComputeResult(20))
```

- ▶ En este ejemplo, tanto las líneas 5 como 6 generarán un *ReferenceError*.

## 10.1.2. Identificadores no definidos

```
1  function computeResult(x) {  
2      return x*2  
3  }  
4  myResult=15  
5  console.log(myResult)
```

- ▶ Este programa no genera ningún error
  - ▶ No es obligatorio definir una variable precediéndola con **var** o **let**.
  - ▶ Cuando ninguna de estas palabras clave se utiliza, entonces la variable tiene un ámbito global.



## 10.2. Errores semánticos

---

- ▶ **Los errores semánticos** son aquellos relacionados con la ejecución de nuestro código.
  - ▶ En estos errores, los mensajes proporcionados por el intérprete son solo una pista.
  - ▶ Un subconjunto importante de estos errores está causado por la invocación incorrecta de una función.
    - ▶ Por ejemplo: la función necesita un argumento de un tipo dado, pero se ha llamado utilizando un valor incompatible.

## 10.2. Errores semánticos

```
1  function sum(A) {  
2      return A.reduce((x,y)=>x+y)  
3  }  
4  console.log(sum([1,3,5]))  
5  console.log(sum(1))
```

- ▶ La función *sum()* asume que su argumento será un *Array*
  - ▶ Así podrá utilizar su método *reduce()*
  - ▶ Si se pasa algo diferente, entonces no tendrá ese método y eso generará un *TypeError* durante la ejecución

## 10.2. Errores semánticos

- ▶ En el programa de la página anterior, la línea 5 genera un error...

```
return A.reduce((x,y)=>x+y)
      ^
TypeError: A.reduce is not a function
    at sum (C:\Users\fmunyo\Documents\tsr\Lab00\example51.js:2:14)
    at Object.<anonymous> (C:\Users\fmunyo\Documents\tsr\Lab00\example51.js:5:13)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:191:16)
    at bootstrap_node.js:612:3
```

- ▶ El mensaje de error puede llegar a ser confuso...
  - ▶ Declara que "A.reduce" no es una función, pero...
    - Eso sólo significa que A no es un *Array*.
    - Obsérvese que en la línea 5 se pasó el valor 1 como el argumento de sum()
      - ¡Es un entero en vez de un *Array*!!!

## 10.2. Errores semánticos

- ▶ Para evitar esos errores, tendríamos que comprobar el tipo asumido en sus parámetros.
- ▶ Con este fin, tendríamos que utilizar:
  - ▶ **typeof**, para tipos primitivos
    - ▶ Un ejemplo apareció en la página 24
  - ▶ **instanceof**, para clases de objeto
    - ▶ Como el mostrado en este ejemplo:

```
1  function sum(A) {  
2      if (!(A instanceof Array))  
3          throw "sum: The parameter must be an array!"  
4      else return A.reduce((x,y)=>x+y)  
5  }  
6  console.log(sum([1,3,5]))  
7  console.log(sum(1))
```



## 10.2. Errores semánticos

- ▶ Para evitar esos errores, tendríamos que comprobar el tipo asumido en sus parámetros
- ▶ Con este fin, tendríamos que usar
  - ▶ **typeof**, para tipos primitivos
    - ▶ Un ejemplo apareció en la página 24
  - ▶ **instanceof**, para clases de objeto
    - ▶ Como el mostrado en este ejemplo

La línea 2 comprueba si el argumento A es un *Array*. Si no lo es, se genera una excepción en la línea 3, indicando que el argumento tendría que ser de ese tipo.

```
1 function sum(A) {  
2   if (!(A instanceof Array))  
3     throw "sum: The parameter must be an array!"  
4   else return A.reduce((x,y)=>x+y)  
5 }  
6 console.log(sum([1,3,5]))  
7 console.log(sum(1))
```