

# Laboratori 0 – Introducció a JavaScript

Tecnologia dels Sistemes d'Informació en la Xarxa



# Índex

---

1. Introducció
2. Objectius
3. Eines
4. El llenguatge JavaScript
5. Tipus de dades primitius
6. Tipus estructurats
7. Funcions
8. Àmbit
9. Context d'execució
10. Errors



# Índex d'exercicis

Objectiu	Pàgina
Tipus dinàmics	<a href="#"><u>18</u></a>
Tipificació feble	<a href="#"><u>21</u></a>
El tipus <b>undefined</b>	<a href="#"><u>24</u></a>
Problemes de tipificació	<a href="#"><u>29</u></a>
Coerció de tipus	<a href="#"><u>32</u></a> , <a href="#"><u>34</u></a> , <a href="#"><u>36</u></a> , <a href="#"><u>37</u></a>
Objectes	<a href="#"><u>48</u></a>
Propietats d'objectes	<a href="#"><u>52</u></a>
Ús de funcions	<a href="#"><u>58</u></a>
Implantació de funcions	<a href="#"><u>66</u></a> , <a href="#"><u>67</u></a>
La notació fletxa (=>)	<a href="#"><u>69</u></a>
Let / var	<a href="#"><u>75</u></a>
Let	<a href="#"><u>81</u></a>
Clausures	<a href="#"><u>84</u></a>



# I. Introducció

---

- ▶ JavaScript és un llenguatge de programació àmpliament utilitzat en sistemes distribuïts.
- ▶ Està basat en l'especificació ECMAScript.
  - ▶ Utilitzarem ECMAScript 6 en aquest curs.
- ▶ En TSR, JavaScript és una eina apropiada per a aconseguir diversos objectius pedagògics.
  - ▶ Però no estem interessats en un aprenentatge en profunditat d'aquest llenguatge ni en totes les seues biblioteques.



## 2. Objectius

---

- ▶ Explicar un conjunt bàsic de característiques de JavaScript que permeten abordar fàcilment el Tema 2...
  - ▶ ...i començar la Pràctica I.
- ▶ Entendre que JavaScript és un llenguatge de programació estrany.
  - ▶ No és similar a Java.
  - ▶ Necessita un intèrpret (en comptes d'un compilador).
  - ▶ És difícil d'aprendre en poc temps si alguns dels conceptes bàsics no han sigut explicats amb deteniment.
- ▶ Presentar el conjunt d'eines a utilitzar en els laboratoris.
  - ▶ Aquestes dues setmanes inicials haurien de ser dedicades a l'aprenentatge d'aquestes eines.



## 2. Objectius

---

- ▶ Entendre correctament els missatges d'error.
  - ▶ Alguns missatges resulten de vegades imprecisos.
    - ▶ Però hauríem de revisar-los amb cura perquè proporcionen pistes sobre les causes de cada error.
    - ▶ Aquestes pistes haurien de ser enteses adequadament.
    - ▶ Tractarem de proporcionar una primera guia en aquesta presentació.



## 3. Eines

- ▶ Necessitem, almenys, aquestes eines:
  - ▶ Un editor de text
    - ▶ Per a escriure els nostres programes i modificar-los.
    - ▶ Hi ha una gran varietat d'editors.
      - Cadascun pot proporcionar algunes característiques útils:
        - Suport en la depuració
        - Revisió sintàctica
        - Personalització
        - Col·leccions de complements (*plugins*)
        - Documentació sobre l'API
        - Control de versions
  - ▶ Un intèrpret
    - ▶ Per a executar els nostres programes
    - ▶ Utilitzarem NodeJS amb la seua ordre “node”



## 3.1. Editor de text

- ▶ Utilitzarem *Visual Studio Code* als laboratoris
  - ▶ És un editor multiplataforma amb un conjunt ampli de característiques útils
  - ▶ Pot descarregar-se des de:  
<https://code.visualstudio.com/download>
  - ▶ La seua documentació està disponible en  
<https://code.visualstudio.com/docs>
  - ▶ Està instal·lat en l'escriptori virtual del DSIC (EVIR)  
(<http://www.upv.es/entidades/dsic/infoweb/dsic/info/1043006normali.html>)
    - ▶ EVIR és l'escriptori virtual remot proporcionat pel DSIC per a usar un entorn informàtic similar a l'utilitzat en els laboratoris.
      - Des d'EVIR podem accedir a les nostres màquines virtuals, on els projectes de laboratori haurien de ser desenvolupats i executats.









## 3.2. Intèrpret

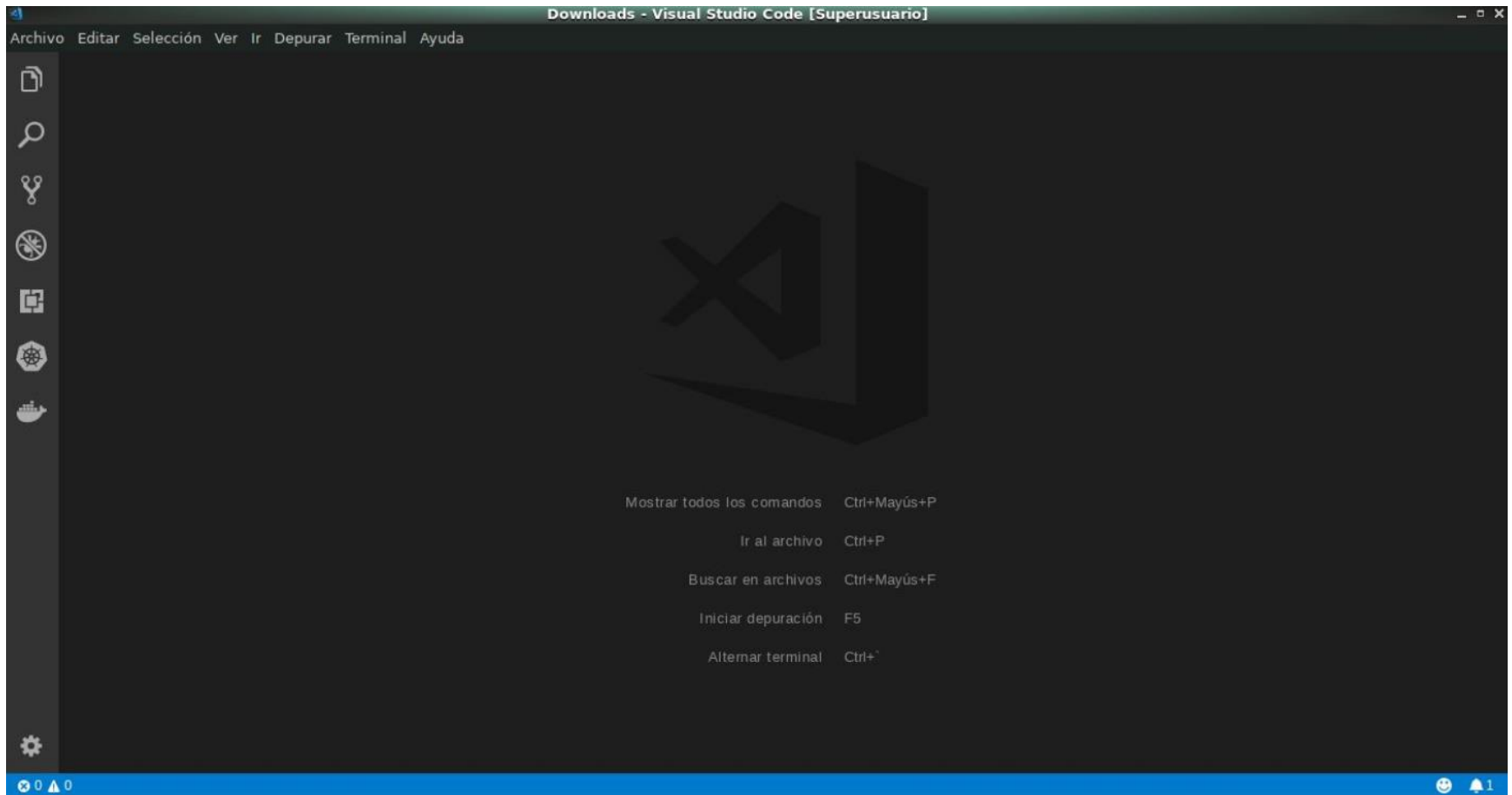
- ▶ Utilitzarem *NodeJS*
  - ▶ És l'interpret que serà usat en l'aula i en els laboratoris
  - ▶ Pot descarregar-se des de <https://nodejs.org/en/download/>
    - ▶ La versió instal·lada als laboratoris és la 10.16.0
  - ▶ La documentació està disponible en <https://nodejs.org/en/docs/>
    - ▶ En cas de l'API, cal utilitzar la seua versió 10.x.y més recent (<https://nodejs.org/dist/latest-v10.x/docs/api/>)
  - ▶ Està també instal·lat en l'EVIR del DSIC.

## 3.3. Utilitzant VS Code

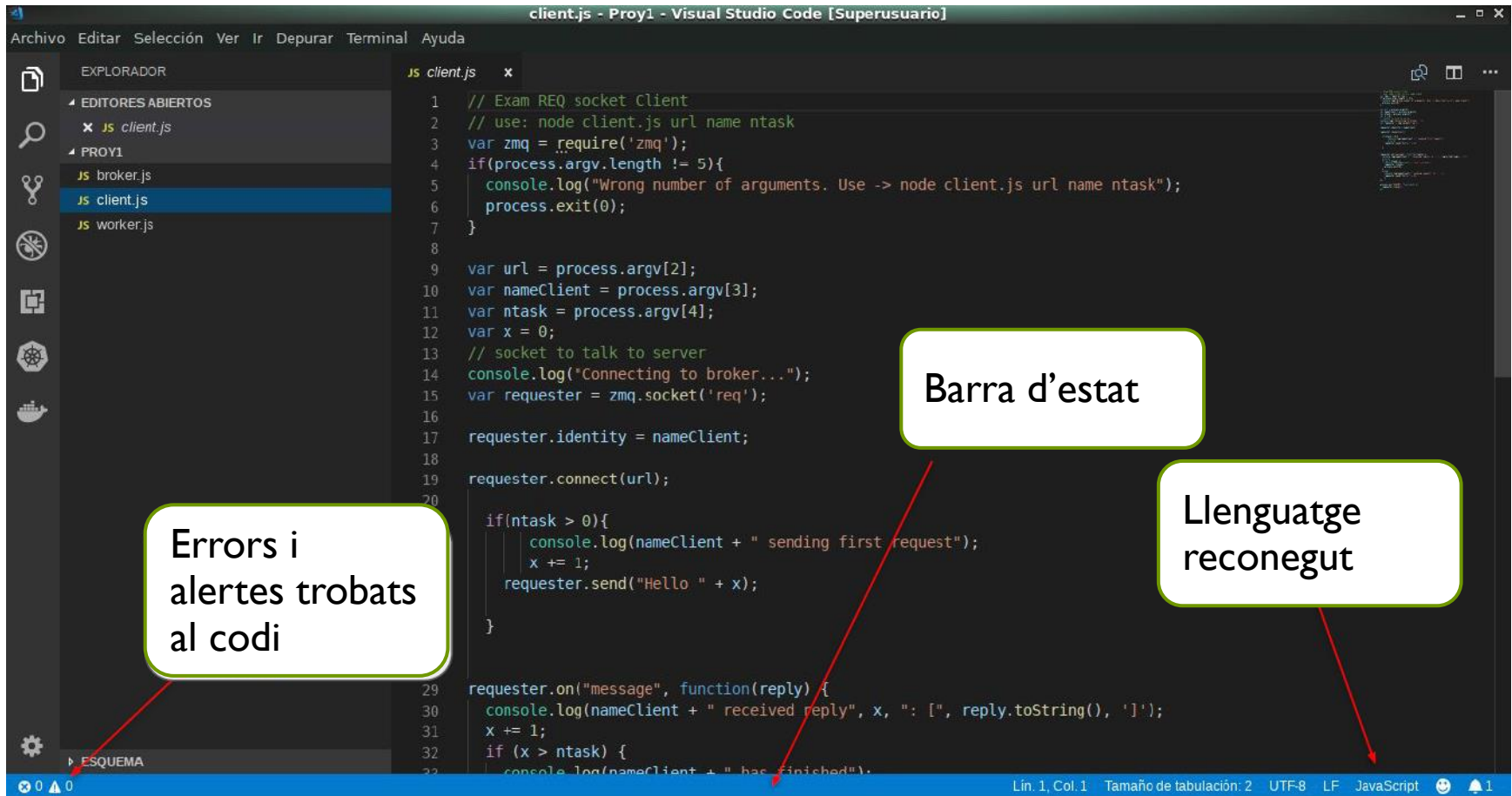
- ▶ Visual Studio Code (VS Code) és un editor de text que gestiona fitxers, carpetes o projectes.
- ▶ Estructura els seus elements d'una manera senzilla.
  - ▶ Hi ha diverses icones en un panell xicotet a l'esquerra:
    - ▶ Explorador de fitxers (  ): Per a accedir a fitxers o carpetes.
    - ▶ Cerca (  ): cerca qualsevol text en el fitxer actual.
    - ▶ Control de versions (  ): Per a integrar el nostre projecte en un sistema de control de versions.
    - ▶ Depuració (  ): facilita la depuració del nostre programa en utilitzar punts de ruptura o explorant els valors actuals de les variables.
  - ▶ Cadascuna d'aquestes icones mostra un menú amb diverses accions relacionades.



## 3.3. Utilitzant VS Code



## 3.3. Utilitzant VS Code





## 3.3. Utilitzant VS Code

---

- ▶ Cada fitxer amb l'extensió “.js” s'identifica com a programa JavaScript
  - ▶ La seua sintaxi es destaca adequadament
- ▶ Per a executar un programa en JavaScript, podem utilitzar...
  - ▶ L'interpret **node** en una terminal separada, o
  - ▶ La terminal inclosa en VS Code (Ctrl + `)
    - ▶ Com es mostra en la pròxima pàgina



## 3.3. Utilitzant VS Code

```
client.js - Proy1 - Visual Studio Code [Superusuariu]
Archivo  Editar  Selecció  Ver  Ir  Depurar  Terminal  Ayuda

EXPLORADOR
  EDITORES ABIERTOS
    JS client.js
  PROY1
    .vscode
    JS broker.js
    JS client.js
    JS prueba.js
    JS worker.js

ESQUEMA
  Filtro
    nameClient
    ntask
    requester
    url
    x

JS client.js
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);

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
1: bash
[root@TSR-mgonzale-1819 Proy1]# node client.js
Wrong number of arguments. Use -> node client.js url name ntask
[root@TSR-mgonzale-1819 Proy1]#
```



## 3.3. Utilitzant VS Code

---

- ▶ Algunes operacions disponibles en VS Code:
  - ▶ Dividir les finestres en dues meitats verticals, per a comparar dos fitxers.
  - ▶ Mantenir un històric dels fitxers editats, recordant l'última posició editada en cada fitxer.
    - ▶ L'històric pot recórrer-se cap...
      - Arrere (Ctrl + Alt + -)
      - Avant (Ctrl + Majs + -)



## 3.4. Utilitzant l'interpret

- ▶ Una vegada instal·lat, podem utilitzar l'interpret de NodeJS amb aquesta ordre:

```
node program.js [llista d'arguments]
```

- ▶ ...on:
  - ▶ L'extensió “.js” no és obligatòria.
  - ▶ Els arguments del programa, si n'hi ha cap, hauran d'escriure's després del seu nom.



## 4. El llenguatge JavaScript

- ▶ Algunes característiques d'aquest llenguatge són:
  - ▶ Tipus dinàmics
  - ▶ Tipificació feble
  - ▶ Tipus primitius
  - ▶ Coerció de tipus
- ▶ Assumirem aquest fragment de codi per a avaluar aquestes característiques:

```
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 llenguatge JavaScript

---

- ▶ Algunes qüestions sobre el programa de la pàgina anterior:
  - ▶ Hi haurà algun error en l'execució del programa?
  - ▶ En què es diferencia d'un programa escrit en Java?
  - ▶ Podem utilitzar una variable que no haja sigut definida amb **“let”** o **“var”**?



## 4.1. Tipus dinàmics

- ▶ JavaScript utilitza tipus dinàmics
  - ▶ No necessita especificar el tipus de les variables
    - ▶ El seu tipus depèn dels valors assignats
  - ▶ Hi ha certa llibertat en accedir a les components d'un vector
    - ▶ Podria ocórrer que no tingueren cap valor assignat
      - Però això no genera cap error!
      - Retornaran “**undefined**”!
- ▶ Els tipus dinàmics poden ser útils quan s'utilitzen adequadament
  - ▶ Però també poden ser una font d'errors!!
- ▶ **Tipus dinàmics:** les variables canvien el seu tipus segons el valor assignat i aquest valor pot canviar mentre el programa s'executa.



## 4.2. Tipificació feble

- ▶ JavaScript és un llenguatge de programació amb tipificació feble
  - ▶ Això significa que les seues expressions no comproven semànticament si els seus operadors estan aplicats a variables amb un tipus adequat.
  - ▶ Això és molt flexible...
    - ▶ ...però és també propens a errors, com mostra aquest exemple:

```
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. Tipificació feble

---

- ▶ L'exemple mostrat en la pàgina anterior...
  - ▶ ...il·lustra que es poden escriure expressions que combinen valors de tipus diferents.
    - ▶ JavaScript té algunes regles d'avaluació per a determinar el tipus resultant.
- ▶ Preguntes sobre l'exemple:
  - ▶ Hi ha algun resultat inesperat?
  - ▶ Hi ha alguna expressió (aparentment) incorrecta?
    - ▶ Prova-les totes i comprova els resultats.
  - ▶ Convé escriure expressions similars en els nostres programes?



## 5. Tipus de dades primitius

- ▶ En JavaScript, els tipus de dades primitius són:
  - ▶ **Number**
  - ▶ **Boolean**
  - ▶ **String**
  - ▶ **undefined**
  - ▶ **null**
    - ▶ A pesar que puga ser considerat un valor, ECMAScript el considera un tipus
  - ▶ **Symbol**
    - ▶ No el considerarem en aquesta presentació
- ▶ Un tipus de dada és **primitiu** quan és senzill (és a dir, no estructurat) i hi ha una manera per a obtenir el tipus d'una variable que pertany a ell (operador *typeof*)

## 5.1. Tipus de dades primitius: *undefined*

- ▶ **undefined** és un tipus de dades que correspon a totes aquelles variables a les quals no s'ha assignat encara un valor

- ▶ És a dir, correspon a variables no inicialitzades

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

- ▶ **undefined** també s'usa quan un paràmetre de funció no ha rebut cap valor en una invocació.

- ▶ **Exemple:**

- ▶ Quan una funció amb tres paràmetres es crida utilitzant un únic argument, el segon i tercer paràmetres reben *undefined*.

## 5.1. Tipus de dades primitius: *undefined*

- ▶ **undefined** també hauria d'usar-se per a comprovar si una variable ja té algun valor assignat.
- ▶ Per a fer això hauríem d'utilitzar l'operador **typeof**, com il·lustra aquest exemple:

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

- ▶ Exercici:
  - ▶ Cercar altres maneres de comprovar si una variable és *undefined* o no.





## 5.2. Tipus de dades primitius: *null*

---

- ▶ **null** és un valor mantingut per aquelles variables Object a les quals encara no se'ls haja assignat un valor.
- ▶ ECMAScript considera **null** com un tipus de dades primitiu, a pesar que ha sigut tradicionalment utilitzat com a valor literal en JavaScript.
  - ▶ no hi ha cap conflicte entre aquestes dues interpretacions
- ▶ Algunes funcions retornen **null** per a indicar que no han trobat cap objecte apropiat.



## 5.3. Tipus de dades primitius: *Number*

- ▶ **Number** és un tipus que correspon tant a nombres enters com reals.
  - ▶ Els reals tenen una precisió limitada a causa del seu format de coma flotant
  - ▶ Exemple:

```
1  let x=0.2
2  let y=0.29999999999999999999
3
4  if (x+y==0.5)
5      console.log("The result is inaccurate.")
```



## 5.3. Tipus de dades primitius: *Number*

- ▶ **NaN** és un resultat que indica que alguna operació numèrica no ha tingut sentit.
  - ▶ Exemples d'operacions d'aquesta classe:
    - ▶ **0/0**
      - Així i tot, les operacions **valor/0** no retornen NaN, sinó **Infinity**.
    - ▶ **Infinity – Infinity**
    - ▶ Operacions matemàtiques on **undefined** s'ha utilitzat com un operand.
    - ▶ Operacions matemàtiques que usen un tipus d'operand inapropiat
      - P. ex., “El meu nom” \* 3
    - ▶ Funcions que esperen un Number com a argument i no reben un valor d'aquest tipus
      - P. ex., parseInt(“una cadena”)



## 5.4. Tipus de dades primitius: *String*

- ▶ Els objectes **String** tenen per ommissió una propietat: *length*
  - ▶ Manté el nombre de caràcters en la cadena
- ▶ Els literals o objectes **String** poden concatenar-se utilitzant l'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. Tipus de dades primitius: Errors i tipus

- ▶ JavaScript té una gestió de tipus feble. A vegades aquesta característica pot ser l'origen d'errors.

- ▶ Exemple:

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

- ▶ Qüestions:

- ▶ Quin és el resultat en aquest exemple?
  - ▶ Per què s'obté un valor inesperat?

- ▶ **NaN** i **undefined** poden ser l'origen de molts errors de programació.

- ▶ Hauríem d'entendre per què aquests valors es generen en algunes declaracions



## 5.6. Tipus de dades primitius: coerció de tipus

- ▶ Què passa si una expressió mescla tipus de dades diferents?
  - ▶ Com JavaScript és un llenguatge de programació amb gestió feble de tipus, no genera errors.
    - ▶ En canvi, aplica algunes regles per a transformar aquesta expressió en alguna cosa que tinga sentit.
    - ▶ A aquest efecte, alguns operands es converteixen a valors dels tipus de dades esperats.
- ▶ Definició de **Coerció de tipus** (segons la Reial Acadèmia d'Enginyeria):
  - ▶ “Característica dels llenguatges de programació que permet, implícitament o explícita, convertir un element d'un tipus de dades en un altre, sense tenir en compte la comprovació de tipus.”

## 5.6. Tipus de dades primitius: coerció de tipus

- ▶ Revisem un exemple 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]) // ??
```

- ▶ Qüestions:
  - ▶ Es pot deduir quines regles dirigeixen les coercions de tipus aplicades en aquest exemple?
  - ▶ Ofereixen totes un resultat “correcte”?
  - ▶ Són útils?



## 5.6. Tipus de dades primitius: coerció de tipus

- ▶ Les regles de la coerció de tipus s'apliquen també a expressions lògiques.
- ▶ **Exercici:** Comprova alguns exemples:
  - ▶ És la cadena literal “5” major que 3?
  - ▶ És una variable amb valor “6” igual a 6?
  - ▶ És la cadena literal “user” igual a false ?
  - ▶ És la cadena buida (“”) igual a false ?
  - ▶ Verifica quin valor booleà correspon a **undefined** i **NaN**.
    - ▶ A causa d'això, què passa quan comparem el valor d'una variable amb **undefined**?
      - Això explica per què **undefined** està considerat un tipus en comptes d'un valor literal.



## 5.6. Tipus de dades primitius: coerció de tipus

- ▶ De vegades, podem controlar la coerció de tipus usant alguns operadors que converteixen un tipus en un altre.
  - ▶ Exemples: `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. Tipus de dades primitius: coerció de tipus

---

### ▶ Exercici:

#### ▶ Determinar el resultat d'aquestes operacions:

- ▶ Boolean("false")
- ▶ Boolean(NaN)
- ▶ Boolean(undefined)
- ▶ Boolean("undefined")

## 5.6. Tipus de dades primitius: coerció de tipus

- ▶ La coerció de tipus pot evitar-se si utilitzem l'operador de comparació estricte (“===”) en comptes de l'operador de comparació normal (“==”).

- ▶ **Exemples:**

```
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. Tipus de dades primitius: coerció de tipus

- ▶ La coerció de tipus pot utilitzar-se per a simplificar condicions.

- ▶ **Exemples:**

- ▶ Si és una cadena buida:

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

- ▶ Si una variable ha sigut definida o no:

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

- Quan *person* siga **undefined** o **null** aquestes instruccions funcionaran com s'espera.
    - Però... què passarà si *person* és 0 o cadena buida?

- ▶ **Exercici:**

- ▶ Com pot comprovar-se si *person* ha sigut definida, considerant també el valor 0 i la cadena buida?

## 5.6. Tipus de dades primitius: coerció de tipus

### ► Solucions a l'exercici de la pàgina anterior:

#### 1. Utilitzant comparació estricta:

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

#### 2. Sense coerció de tipus:

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

### ► Qüestions:

- Què passa si eliminem la línia 1?
- En la segona solució, podria usar-se `!=` en comptes de `!==` ?



## 6. Tipus estructurats

---

- ▶ JavaScript proporciona molts tipus estructurats que mantenen diversos elements de tipus primitius:
  - ▶ Arrays: Seqüències de valors que poden ser accedides utilitzant índexs.
  - ▶ Objectes: Seqüències de parells clau/valor.
  - ▶ Col·leccions: Aquesta classe de tipus estructurat no la utilitzarem en l'assignatura.

## 6.1. Tipus estructurats: *Array*

- ▶ *Array* és un objecte JavaScript similar a una llista...
  - ▶ ...amb una propietat **length**
    - ▶ que retorna quants elements hi ha en l'*Array*
  - ▶ ...i alguns mètodes
    - ▶ `indexOf()`, `pop()`, `push()`, `shift()`, `map()`, `slice()`...
- ▶ Hi ha documentació sobre *Array* en:
  - ▶ [Mozilla Developer Network](#), [w3schools.com](#),...
- ▶ Exemple:

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



## 6.1. Tipus estructurats: *Array*

- ▶ A causa de les característiques de JavaScript, la inserció d'elements i l'accés a elements de l'*Array* encara no definits són tasques que han de gestionar-se amb cura.

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



## 6.1. Tipus estructurats: *Array*

- ▶ No podem copiar un *Array* assignant el seu “valor” a una altra variable:

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

- ▶ En aquest cas, estem copiant una referència a l'objecte *Array*.
- ▶ Per això, ara tindrem dues referències al mateix *Array*.

## 6.1. Tipus estructurats: *Array*

- ▶ No podem copiar un *Array* assignant el seu “valor” a una altra variable:

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

- ▶ En comptes d'això, hauríem d'utilitzar el mètode slice().
  - ▶ Si no s'utilitzen arguments, slice() retorna una còpia del vector.
  - ▶ Hi ha dos paràmetres opcionals en slice():
    1. L'índex del primer element a copiar. Si aquest argument és **undefined**, la còpia comença en l'índex 0.
    2. Un valor una unitat major que l'índex de l'últim element a copiar. Per omissió, s'assumeix la longitud del vector.

## 6.1. Tipus estructurats: *Array*

- ▶ Per a inserir elements en un *Array*, hem d'utilitzar les seues posicions...
  - ▶ Però això sobreescriu el contingut previ d'aquestes components.
  - ▶ Hi ha altres operacions que afegen nous elements al principi o al final del vector, desplaçant o mantenint el seu contingut previ, respectivament.
  - ▶ De manera similar, hi ha altres operacions per a eliminar elements:

	AFEGIR	ELIMINAR
Al principi	<u>unshift</u> (elem1,...)	<u>shift</u> ()
Al final	<u>push</u> (elem1,...)	<u>pop</u> ()

## 6.1. Tipus estructurats: *Array*

- ▶ Hi ha alguns “pseudoarrays” que a vegades haurem de convertir en objectes de tipus *Array*.
- ▶ Per a fer això podem utilitzar el mètode `Array.from()`
- ▶ Exemple 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 edicions anteriors de l'estàndard ECMAScript l'operació `Array.from()` no existia.
  - ▶ Aleshores s'utilitzava:  
**`Array.prototype.slice.call(arguments)`**

## 6.2. Tipus estructurats: *Object*

- ▶ Un objecte és un conjunt no ordenat de parells clau / valor (on “clau” és equivalent a una “propietat”).
  - ▶ El valor de les claus o propietats pot ser qualsevol literal de tipus primitius, funció o un altre objecte.
  - ▶ Exemple:

```
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. Tipus estructurats: *Object*

- ▶ Els objectes també poden ser creats de manera dinàmica.

- ▶ Exemple:

```
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)
```

- ▶ Així i tot, la forma estàtica mostrada en la pàgina anterior és més ràpida i comuna.
- ▶ **Alerta!!** Com la declaració/modificació dinàmica és possible, si escrivim incorrectament el nom de qualsevol propietat i li assignem un valor...
  - ❑ No hi haurà errors...
  - ❑ ...però aquest nom incorrecte crearà una altra propietat en l'objecte!!

## 6.2. Tipus estructurats: *Object*

- ▶ Els objectes també poden ser creats en una

- ▶ Exemple:

```
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)
```

Hi ha una tercera sintaxi per a declarar les propietats d'un objecte: poden definir-se entre claudàtors! (Com en els Arrays)

- ▶ Per tant, aquestes línies són equivalents a les de l'exemple :  
**let person={}; let property="street"**  
**person['name']="Peter"; person['age']=25**  
**person['address']={}; person['address']['city']="València"**  
**person['address'][property]="Tres Cruces"**  
**person['address'].number=12**  
**console.log(person)**



## 6.2. Tipus estructurats: *Object*

- ▶ Els objectes poden crear-se o modificar-se dinàmicament, però...
- ▶ Exercici:
  - ▶ Explicar què passa quan accedim a una propietat no definida.
  - ▶ Provar aquests exemples per a respondre:

```
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. Tipus estructurats: *Object*. JSON

---

- ▶ JSON (JavaScript Object Notation) és un format de text utilitzat per a “serialitzar” objectes quan hagen de transmetre's per la xarxa.
  - ▶ Cada identificador de propietat està tancat entre cometes dobles.
  - ▶ Per a gestionar el format JSON, podem utilitzar...
    - ▶ `JSON.stringify(Object)` converteix un objecte de JavaScript en una cadena JSON.
    - ▶ `JSON.parse(String)` converteix una cadena JSON en un objecte JavaScript.

## 6.2.1. Tipus estructurats: *Object*. JSON

### ► Exemple:

#### ► Considerem aquest 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))
```

#### ► La seua eixida, en format JSON, és...

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

## 6.2.2. Tipus estructurats: *Object*. Bucles

- ▶ Podem utilitzar un bucle **for(variable in objecte)** per a processar cada propietat en un objecte donat.
- ▶ Exemple:

```
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” rep el nom de cada propietat en cada iteració d'aquest bucle.
- ▶ Amb aquest nom, podem accedir al valor de cada propietat
  - ▶ Així, s'intueix que **un objecte és similar a un Array i les seues propietats són els índexs d'aquest Array**.
  - ▶ Aquesta característica pot usar-se quan els identificadors de les propietats es mantenen en altres variables.



## 6.2.2. Tipus estructurats: *Object*. Bucles

- ▶ En l'exemple anterior, l'eixida obtinguda era...

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

- ▶ Exercici:
  - ▶ Estendre l'exemple anterior, mostrant les propietats i valors de la propietat “address”.
  - ▶ Una solució general per a aquest exercici necessita utilitzar funcions i aquestes s'expliquen en la pròxima secció.



## 7. Funcions

- ▶ El concepte de “funció” és similar a l'utilitzat en qualsevol altre llenguatge de programació.
- ▶ Una seqüència d'instruccions que es pot cridar des d'altres parts del programa.
- ▶ Una funció defineix una interfície clara per a utilitzar aquest fragment de codi.

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

## 7. Funcions

- ▶ Si una funció no utilitza la instrucció “**return**”, llavors la seua execució retorna el valor **undefined**.

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

## 7. Funcions

- ▶ Els paràmetres de les funcions es comporten com a variables l'àmbit de les quals està limitat al codi de la seua funció.
- ▶ Executa l'exemple següent i observa que...

```
1  function add(x,y,z) {  
2      |      return x+y+z  
3      |  
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))
```

- ▶ Quan una funció es crida amb menys arguments dels declarats, llavors els paràmetres no utilitzats reben el valor **undefined**.
- ▶ Quan una funció es crida amb més arguments dels previstos, llavors els sobrants són descartats.
  - ▶ No hi haurà cap error en tots dos casos.

## 7. Funcions

- ▶ Si alguns paràmetres foren opcionals, es pot assignar un valor per omisió en la seua declaració.
- ▶ Així, no necessitem comprovar si són **undefined** en el cos de la funció.

```
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))
```

- ▶ Amb això, les línies 6 i 7 no mostren **NaN** perquè ara tots els arguments a processar seran nombres enters.



## 7. Funcions

- ▶ Les funcions que tinguen un nombre desconegut d'arguments poden utilitzar el paràmetre “...”
  - ▶ Per a fer això, l'identificador de l'últim paràmetre hauria d'anar precedit per punts suspensius, és a dir, “...id”
  - ▶ Aquest paràmetre és *un Array* que arreplega els arguments restants.
  - ▶ Exemple:

```
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. Funcions

### ► Exercici:

- Si assumim el programa mostrat en la pàgina anterior, quin serà el resultat de la línia següent?

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

## 7. Funcions

- ▶ Els arguments es passen...
  - ▶ Per valor si pertanyen a un tipus primitiu
  - ▶ Per referència quan són objectes
    - ▶ Observa que els *Arrays* són també objectes
    - ▶ Podrem canviar el contingut de l'objecte, però no podrem canviar la referència rebuda.

```
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. Funcions

- ▶ JavaScript gestiona les funcions com a objectes comuns, per això poden ser:
  - ▶ Utilitzades com a valors, i assignades a variables
  - ▶ Utilitzades com a arguments en crides a altres funcions
  - ▶ Retornades com a resultat d'altres funcions

```
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)
```

- ▶ Convindria distingir aquests usos de les funcions:
  - ▶ La seua definició inicial.
  - ▶ La seua utilització en expressions. Opcions:
    - ▶ Com a referència, quan només s'use el seu identificador.
    - ▶ El resultat de la seua invocació, en usar parèntesi i donar una llista d'arguments (possiblement buida).

## 7. Functions

### ► Exemple:

```
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))
```



## 7. Funcions

- ▶ Les funcions poden ser definides anònimament, és a dir, sense donar-los cap nom.
- ▶ El programa següent és equivalent al mostrat en la pàgina anterior:

```
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. Funcions

- ▶ Les funcions anònimes són àmpliament utilitzades com a arguments en la invocació d'altres funcions.

```
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)})
```

### ► Notació de fletxa

- Les funcions anònimes són àmpliament utilitzades. Tindria sentit una sintaxi més concisa → La “fletxa” ( => )
  - No es necessita la paraula **function**
  - Es manté la llista de paràmetres
    - Els parèntesis són opcionals quan només hi haja un paràmetre
  - L'operador => segueix a la llista
  - Després d'ell, una sentència que calcule el resultat a retornar
    - O un bloc de sentències entre claus.

```
1  function computeTable(n,fn) {  
2      |      for (let c=1; c<11; c++)  
3      |          fn(n*c)  
4      |  
5      |  
6      |  
7      |  
8      |  
9      |  
10     |  
11     |  
12     |  
13     |  
14     |  
15     |  
16     |  
17     |  
18     |  
19     |  
20     |  
21     |  
22     |  
23     |  
24     |  
25     |  
26     |  
27     |  
28     |  
29     |  
30     |  
31     |  
32     |  
33     |  
34     |  
35     |  
36     |  
37     |  
38     |  
39     |  
40     |  
41     |  
42     |  
43     |  
44     |  
45     |  
46     |  
47     |  
48     |  
49     |  
50     |  
51     |  
52     |  
53     |  
54     |  
55     |  
56     |  
57     |  
58     |  
59     |  
60     |  
61     |  
62     |  
63     |  
64     |  
65     |  
66     |  
67     |  
68     |  
69     |  
70     |  
71     |  
72     |  
73     |  
74     |  
75     |  
76     |  
77     |  
78     |  
79     |  
80     |  
81     |  
82     |  
83     |  
84     |  
85     |  
86     |  
87     |  
88     |  
89     |  
90     |  
91     |  
92     |  
93     |  
94     |  
95     |  
96     |  
97     |  
98     |  
99     |  
100    |  
101    |  
102    |  
103    |  
104    |  
105    |  
106    |  
107    |  
108    |  
109    |  
110    |  
111    |  
112    |  
113    |  
114    |  
115    |  
116    |  
117    |  
118    |  
119    |  
120    |  
121    |  
122    |  
123    |  
124    |  
125    |  
126    |  
127    |  
128    |  
129    |  
130    |  
131    |  
132    |  
133    |  
134    |  
135    |  
136    |  
137    |  
138    |  
139    |  
140    |  
141    |  
142    |  
143    |  
144    |  
145    |  
146    |  
147    |  
148    |  
149    |  
150    |  
151    |  
152    |  
153    |  
154    |  
155    |  
156    |  
157    |  
158    |  
159    |  
160    |  
161    |  
162    |  
163    |  
164    |  
165    |  
166    |  
167    |  
168    |  
169    |  
170    |  
171    |  
172    |  
173    |  
174    |  
175    |  
176    |  
177    |  
178    |  
179    |  
180    |  
181    |  
182    |  
183    |  
184    |  
185    |  
186    |  
187    |  
188    |  
189    |  
190    |  
191    |  
192    |  
193    |  
194    |  
195    |  
196    |  
197    |  
198    |  
199    |  
200    |  
201    |  
202    |  
203    |  
204    |  
205    |  
206    |  
207    |  
208    |  
209    |  
210    |  
211    |  
212    |  
213    |  
214    |  
215    |  
216    |  
217    |  
218    |  
219    |  
220    |  
221    |  
222    |  
223    |  
224    |  
225    |  
226    |  
227    |  
228    |  
229    |  
230    |  
231    |  
232    |  
233    |  
234    |  
235    |  
236    |  
237    |  
238    |  
239    |  
240    |  
241    |  
242    |  
243    |  
244    |  
245    |  
246    |  
247    |  
248    |  
249    |  
250    |  
251    |  
252    |  
253    |  
254    |  
255    |  
256    |  
257    |  
258    |  
259    |  
260    |  
261    |  
262    |  
263    |  
264    |  
265    |  
266    |  
267    |  
268    |  
269    |  
270    |  
271    |  
272    |  
273    |  
274    |  
275    |  
276    |  
277    |  
278    |  
279    |  
280    |  
281    |  
282    |  
283    |  
284    |  
285    |  
286    |  
287    |  
288    |  
289    |  
290    |  
291    |  
292    |  
293    |  
294    |  
295    |  
296    |  
297    |  
298    |  
299    |  
300    |  
301    |  
302    |  
303    |  
304    |  
305    |  
306    |  
307    |  
308    |  
309    |  
310    |  
311    |  
312    |  
313    |  
314    |  
315    |  
316    |  
317    |  
318    |  
319    |  
320    |  
321    |  
322    |  
323    |  
324    |  
325    |  
326    |  
327    |  
328    |  
329    |  
330    |  
331    |  
332    |  
333    |  
334    |  
335    |  
336    |  
337    |  
338    |  
339    |  
340    |  
341    |  
342    |  
343    |  
344    |  
345    |  
346    |  
347    |  
348    |  
349    |  
350    |  
351    |  
352    |  
353    |  
354    |  
355    |  
356    |  
357    |  
358    |  
359    |  
360    |  
361    |  
362    |  
363    |  
364    |  
365    |  
366    |  
367    |  
368    |  
369    |  
370    |  
371    |  
372    |  
373    |  
374    |  
375    |  
376    |  
377    |  
378    |  
379    |  
380    |  
381    |  
382    |  
383    |  
384    |  
385    |  
386    |  
387    |  
388    |  
389    |  
390    |  
391    |  
392    |  
393    |  
394    |  
395    |  
396    |  
397    |  
398    |  
399    |  
400    |  
401    |  
402    |  
403    |  
404    |  
405    |  
406    |  
407    |  
408    |  
409    |  
410    |  
411    |  
412    |  
413    |  
414    |  
415    |  
416    |  
417    |  
418    |  
419    |  
420    |  
421    |  
422    |  
423    |  
424    |  
425    |  
426    |  
427    |  
428    |  
429    |  
430    |  
431    |  
432    |  
433    |  
434    |  
435    |  
436    |  
437    |  
438    |  
439    |  
440    |  
441    |  
442    |  
443    |  
444    |  
445    |  
446    |  
447    |  
448    |  
449    |  
450    |  
451    |  
452    |  
453    |  
454    |  
455    |  
456    |  
457    |  
458    |  
459    |  
460    |  
461    |  
462    |  
463    |  
464    |  
465    |  
466    |  
467    |  
468    |  
469    |  
470    |  
471    |  
472    |  
473    |  
474    |  
475    |  
476    |  
477    |  
478    |  
479    |  
480    |  
481    |  
482    |  
483    |  
484    |  
485    |  
486    |  
487    |  
488    |  
489    |  
490    |  
491    |  
492    |  
493    |  
494    |  
495    |  
496    |  
497    |  
498    |  
499    |  
500    |  
501    |  
502    |  
503    |  
504    |  
505    |  
506    |  
507    |  
508    |  
509    |  
510    |  
511    |  
512    |  
513    |  
514    |  
515    |  
516    |  
517    |  
518    |  
519    |  
520    |  
521    |  
522    |  
523    |  
524    |  
525    |  
526    |  
527    |  
528    |  
529    |  
530    |  
531    |  
532    |  
533    |  
534    |  
535    |  
536    |  
537    |  
538    |  
539    |  
540    |  
541    |  
542    |  
543    |  
544    |  
545    |  
546    |  
547    |  
548    |  
549    |  
550    |  
551    |  
552    |  
553    |  
554    |  
555    |  
556    |  
557    |  
558    |  
559    |  
560    |  
561    |  
562    |  
563    |  
564    |  
565    |  
566    |  
567    |  
568    |  
569    |  
570    |  
571    |  
572    |  
573    |  
574    |  
575    |  
576    |  
577    |  
578    |  
579    |  
580    |  
581    |  
582    |  
583    |  
584    |  
585    |  
586    |  
587    |  
588    |  
589    |  
590    |  
591    |  
592    |  
593    |  
594    |  
595    |  
596    |  
597    |  
598    |  
599    |  
600    |  
601    |  
602    |  
603    |  
604    |  
605    |  
606    |  
607    |  
608    |  
609    |  
610    |  
611    |  
612    |  
613    |  
614    |  
615    |  
616    |  
617    |  
618    |  
619    |  
620    |  
621    |  
622    |  
623    |  
624    |  
625    |  
626    |  
627    |  
628    |  
629    |  
630    |  
631    |  
632    |  
633    |  
634    |  
635    |  
636    |  
637    |  
638    |  
639    |  
640    |  
641    |  
642    |  
643    |  
644    |  
645    |  
646    |  
647    |  
648    |  
649    |  
650    |  
651    |  
652    |  
653    |  
654    |  
655    |  
656    |  
657    |  
658    |  
659    |  
660    |  
661    |  
662    |  
663    |  
664    |  
665    |  
666    |  
667    |  
668    |  
669    |  
670    |  
671    |  
672    |  
673    |  
674    |  
675    |  
676    |  
677    |  
678    |  
679    |  
680    |  
681    |  
682    |  
683    |  
684    |  
685    |  
686    |  
687    |  
688    |  
689    |  
690    |  
691    |  
692    |  
693    |  
694    |  
695    |  
696    |  
697    |  
698    |  
699    |  
700    |  
701    |  
702    |  
703    |  
704    |  
705    |  
706    |  
707    |  
708    |  
709    |  
710    |  
711    |  
712    |  
713    |  
714    |  
715    |  
716    |  
717    |  
718    |  
719    |  
720    |  
721    |  
722    |  
723    |  
724    |  
725    |  
726    |  
727    |  
728    |  
729    |  
730    |  
731    |  
732    |  
733    |  
734    |  
735    |  
736    |  
737    |  
738    |  
739    |  
740    |  
741    |  
742    |  
743    |  
744    |  
745    |  
746    |  
747    |  
748    |  
749    |  
750    |  
751    |  
752    |  
753    |  
754    |  
755    |  
756    |  
757    |  
758    |  
759    |  
760    |  
761    |  
762    |  
763    |  
764    |  
765    |  
766    |  
767    |  
768    |  
769    |  
770    |  
771    |  
772    |  
773    |  
774    |  
775    |  
776    |  
777    |  
778    |  
779    |  
780    |  
781    |  
782    |  
783    |  
784    |  
785    |  
786    |  
787    |  
788    |  
789    |  
790    |  
791    |  
792    |  
793    |  
794    |  
795    |  
796    |  
797    |  
798    |  
799    |  
800    |  
801    |  
802    |  
803    |  
804    |  
805    |  
806    |  
807    |  
808    |  
809    |  
810    |  
811    |  
812    |  
813    |  
814    |  
815    |  
816    |  
817    |  
818    |  
819    |  
820    |  
821    |  
822    |  
823    |  
824    |  
825    |  
826    |  
827    |  
828    |  
829    |  
830    |  
831    |  
832    |  
833    |  
834    |  
835    |  
836    |  
837    |  
838    |  
839    |  
840    |  
841    |  
842    |  
843    |  
844    |  
845    |  
846    |  
847    |  
848    |  
849    |  
850    |  
851    |  
852    |  
853    |  
854    |  
855    |  
856    |  
857    |  
858    |  
859    |  
860    |  
861    |  
862    |  
863    |  
864    |  
865    |  
866    |  
867    |  
868    |  
869    |  
870    |  
871    |  
872    |  
873    |  
874    |  
875    |  
876    |  
877    |  
878    |  
879    |  
880    |  
881    |  
882    |  
883    |  
884    |  
885    |  
886    |  
887    |  
888    |  
889    |  
890    |  
891    |  
892    |  
893    |  
894    |  
895    |  
896    |  
897    |  
898    |  
899    |  
900    |  
901    |  
902    |  
903    |  
904    |  
905    |  
906    |  
907    |  
908    |  
909    |  
910    |  
911    |  
912    |  
913    |  
914    |  
915    |  
916    |  
917    |  
918    |  
919    |  
920    |  
921    |  
922    |  
923    |  
924    |  
925    |  
926    |  
927    |  
928    |  
929    |  
930    |  
931    |  
932    |  
933    |  
934    |  
935    |  
936    |  
937    |  
938    |  
939    |  
940    |  
941    |  
942    |  
943    |  
944    |  
945    |  
946    |  
947    |  
948    |  
949    |  
950    |  
951    |  
952    |  
953    |  
954    |  
955    |  
956    |  
957    |  
958    |  
959    |  
960    |  
961    |  
962    |  
963    |  
964    |  
965    |  
966    |  
967    |  
968    |  
969    |  
970    |  
971    |  
972    |  
973    |  
974    |  
975    |  
976    |  
977    |  
978    |  
979    |  
980    |  
981    |  
982    |  
983    |  
984    |  
985    |  
986    |  
987    |  
988    |  
989    |  
990    |  
991    |  
992    |  
993    |  
994    |  
995    |  
996    |  
997    |  
998    |  
999    |  
1000   |  
1001   |  
1002   |  
1003   |  
1004   |  
1005   |  
1006   |  
1007   |  
1008   |  
1009   |  
1010   |  
1011   |  
1012   |  
1013   |  
1014   |  
1015   |  
1016   |  
1017   |  
1018   |  
1019   |  
1020   |  
1021   |  
1022   |  
1023   |  
1024   |  
1025   |  
1026   |  
1027   |  
1028   |  
1029   |  
1030   |  
1031   |  
1032   |  
1033   |  
1034   |  
1035   |  
1036   |  
1037   |  
1038   |  
1039   |  
1040   |  
1041   |  
1042   |  
1043   |  
1044   |  
1045   |  
1046   |  
1047   |  
1048   |  
1049   |  
1050   |  
1051   |  
1052   |  
1053   |  
1054   |  
1055   |  
1056   |  
1057   |  
1058   |  
1059   |  
1060   |  
1061   |  
1062   |  
1063   |  
1064   |  
1065   |  
1066   |  
1067   |  
1068   |  
1069   |  
1070   |  
1071   |  
1072   |  
1073   |  
1074   |  
1075   |  
1076   |  
1077   |  
1078   |  
1079   |  
1080   |  
1081   |  
1082   |  
1083   |  
1084   |  
1085   |  
1086   |  
1087   |  
1088   |  
1089   |  
1090   |  
1091   |  
1092   |  
1093   |  
1094   |  
1095   |  
1096   |  
1097   |  
1098   |  
1099   |  
1100   |  
1101   |  
1102   |  
1103   |  
1104   |  
1105   |  
1106   |  
1107   |  
1108   |  
1109   |  
1110   |  
1111   |  
1112   |  
1113   |  
1114   |  
1115   |  
1116   |  
1117   |  
1118   |  
1119   |  
1120   |  
1121   |  
1122   |  
1123   |  
1124   |  
1125   |  
1126   |  
1127   |  
1128   |  
1129   |  
1130   |  
1131   |  
1132   |  
1133   |  
1134   |  
1135   |  
1136   |  
1137   |  
1138   |  
1139   |  
1140   |  
1141   |  
1142   |  
1143   |  
1144   |  
1145   |  
1146   |  
1147   |  
1148   |  
1149   |  
1150   |  
1151   |  
1152   |  
1153   |  
1154   |  
1155   |  
1156   |  
1157   |  
1158   |  
1159   |  
1160   |  
1161   |  
1162   |  
1163   |  
1164   |  
1165   |  
1166   |  
1167   |  
1168   |  
1169   |  
1170   |  
1171   |  
1172   |  
1173   |  
1174   |  
1175   |  
1176   |  
1177   |  
1178   |  
1179   |  
1180   |  
1181   |  
1182   |  
1183   |  
1184   |  
1185   |  
1186   |  
1187   |  
1188   |  
1189   |  
1190   |  
1191   |  
1192   |  
1193   |  
1194   |  
1195   |  
1196   |  
1197   |  
1198   |  
1199   |  
1200   |  
1201   |  
1202   |  
1203   |  
1204   |  
1205   |  
1206   |  
1207   |  
1208   |  
1209   |  
1210   |  
1211   |  
1212   |  
1213   |  
1214   |  
1215   |  
1216   |  
1217   |  
1218   |  
1219   |  
1220   |  
1221   |  
1222   |  
1223   |  
1224   |  
1225   |  
1226   |  
1227   |  
1228   |  
1229   |  
1230   |  
1231   |  
1232   |  
1233   |  
1234   |  
1235   |  
1236   |  
1237   |  
1238   |  
1239   |  
1240   |  
1241   |  
1242   |  
1243   |  
1244   |  
1245   |  
1246   |  
1247   |  
1248   |  
1249   |  
1250   |  
1251   |  
1252   |  
1253   |  
1254   |  
1255   |  
1256   |  
1257   |  
1258   |  
1259   |  
1260   |  
1261   |  
1262   |  
1263   |  
1264   |  
1265   |  
1266   |  
1267   |  
1268   |  
1269   |  
1270   |  
1271   |  
1272   |  
1273   |  
1274   |  
1275   |  
1276   |  
1277   |  
1278   |  
1279   |  
1280   |  
1281   |  
1282   |  
1283   |  
1284   |  
1285   |  
1286   |  
1287   |  
1288   |  
1289   |  
1290   |  
1291   |  
1292   |  
1293   |  
1294   |  
1295   |  
1296   |  
1297   |  
1298   |  
1299   |  
1300   |  
1301   |  
1302   |  
1303   |  
1304   |  
1305   |  
1306   |  
1307   |  
1308   |  
1309   |  
1310   |  
1311   |  
1312   |  
1313   |  
1314   |  
1315   |  
1316   |  
1317   |  
1318   |  
1319   |  
1320   |  
1321   |  
1322   |  
1323   |  
1324   |  
1325   |  
1326   |  
1327   |  
1328   |  
1329   |  
1330   |  
1331   |  
1332   |  
1333   |  
1334   |  
1335   |  
1336   |  
1337   |  
1338   |  
1339   |  
1340   |  
1341   |  
1342   |  
1343   |  
1344   |  
1345   |  
1346   |  
1347   |  
1348   |  
1349   |  
1350   |  
1351   |  
1352   |  
1353   |  
1354   |  
1355   |  
1356   |  
1357   |  
1358   |  
1359   |  
1360   |  
1361   |  
1362   |  
1363   |  
1364   |  
1365   |  
1366   |  
1367   |  
1368   |  
1369   |  
1370   |  
1371   |  
1372   |  
1373   |  
1374   |  
1375   |  
1376   |  
1377   |  
1378   |  
1379   |  
1380   |  
1381   |  
1382   |  
1383   |  
1384   |  
1385   |  
1386   |  
1387   |  
1388   |  
1389   |  
1390   |  
1391   |  
1392   |  
1393   |  
1394   |  
1395   |  
1396   |  
1397   |  
1398   |  
1399   |  
1400   |  
1401   |  
1402   |  
1403   |  
1404   |  
1405   |  
1406   |  
1407   |  
1408   |  
1409   |  
1410   |  
1411   |  
1412   |  
1413   |  
1414   |  
1415   |  
1416   |  
1417   |  
1418   |  
1419   |  
1420   |  
1421   |  
1422   |  
1423   |  
1424   |  
1425   |  
1426   |  
1427   |  
1428   |  
1429   |  
1430   |  
1431   |  
1432   |  
1433   |  
1434   |  
1435   |  
1436   |  
1437   |  
1438   |  
1439   |  
1440   |  
1441   |  
1442   |  
1443   |  
1444   |  
1445   |  
1446   |  
1447   |  
1448   |  
1449   |  
1450   |  
1451   |  
1452   |  
1453   |  
1454   |  
1455
```



## 7. Funcions

- ▶ Per tant, aquesta declaració

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

- ▶ ...és equivalent a...

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

### ▶ Exercicis:

- ▶ Escriure una funció `doCheckPasswd()` amb tres paràmetres:
  - `input`
  - `correctPassword`
  - `func`
- ▶ Aquesta funció:
  - ▶ Compara les cadenes passades en els primers dos arguments.
  - ▶ Si són iguals, llavors es crida la funció passada com a tercer argument.
- ▶ Prova-la amb les crides següents:

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

### ► Exercicis:

- Estendre el programa mostrat en la pàgina 62, escrivint una altra funció `doWithNFirstNumbers()` amb 3 paràmetres:
  - `n`: L'últim nombre natural a utilitzar
  - `op`: Funció a ser aplicada en cada nombre natural processat
  - `op2`: Funció a ser aplicada en el resultat d'`op(i)` per a acumular tots els resultats
    - `op2` ha de ser alguna de les funcions de l'`Array arithmeticOperations`
- `doWithNFirstNumbers()` aplica `op()` a tots els nombres naturals de l'interval `1..n`, i acumula els seus resultats utilitzant `op2`.
- Exemples d'invocacions:

```
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. Funcions

- ▶ Hi ha moltes funcions que usen altres funcions com a arguments. Per exemple:

- ▶ Array.map()

- ▶ Crea un Array nou amb els resultats de la funció usada com a primer argument aplicada sobre cada element de l'Array original.
- ▶ map() crida a aquesta funció amb tres arguments:
  - L'element en què la funció hauria de ser aplicada
  - El seu índex
  - L'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. Funcions

---

### ▶ Exercici:

- ▶ Modificar l'exemple de la pàgina anterior, utilitzant notació tradicional per a escriure la funció facilitada com a argument de `map()`.



## 8. Àmbit

- ▶ **NOTA: Aquesta part de la presentació serà explicada a fons en el Tema 2.**
- ▶ L'àmbit dels elements (variables, funcions...) d'un programa està determinat per la ubicació de les seues definicions.
- ▶ Hi ha dos àmbits tradicionals en JavaScript:
  - ▶ **Global**
  - ▶ **Funció (també conegut com a local)**
- ▶ Els elements de l'àmbit **global** (és a dir, els que no hagen sigut definits dins d'alguna funció) poden ser accedits des de qualsevol línia del programa.
- ▶ D'altra banda, cada funció defineix el seu propi àmbit **local**.



## 8. Àmbit

- ▶ Quan un programa s'executa, els elements definits en un àmbit local poden accedir-se des de:
  - ▶ Aquest àmbit local
  - ▶ O des de l'àmbit d'altres funcions col·locades en aquest àmbit local → **“children scope”**
- ▶ Això defineix una jerarquia d'àmbits.
  - ▶ Quan un programa executa una seqüència de crides a funció, aquesta seqüència defineix una **cadena d'àmbits**
    - ▶ Determina quins elements en altres àmbits externs poden ser accedits des de l'actual.
    - ▶ Si una funció o variable està definida en qualsevol element d'aquesta cadena i el seu nom coincideix amb algun de l'àmbit global, llavors s'utilitzarà l'element “local” (és a dir, d'algun àmbit més pròxim).

- ▶ Així, en un programa com...

```
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(g1)  
10 }  
11 a()  
12 let g1=0
```

- ▶ ...a1 no pot ser accedit en b(). Es generarà un error!!
- ▶ Qüestió:
  - ▶ Es pot permetre que b() utilitze a1 de dues maneres. Quines són?



## 8. Àmbit

### Solució 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(g11)  
10 }  
11 a()  
12 var g11=0
```

### Solució 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(g11)  
9     }  
10 }  
11 a()  
12 var g11=0
```

- ▶ A passa una còpia d'a1 a b(). Per tant, b() pot llegir a1, però no la pot modificar.
- ▶ B defineix b() com a funció interna a a(). Així, b() pot veure a1. Per tant, pot tant llegir com escriure en a1.

## 8. Àmbit

- ▶ La paraula **let** té el seu propi àmbit:
  - ▶ Quan **let** s'utilitza en l'àmbit global...
    - ▶ No gestiona les variables o funcions com a propietats de l'objecte global.
      - La paraula clau **var** gestiona aquests elements com a propietats d'aquest objecte global.
        - Per tant, són visibles fins i tot abans d'executar la declaració que els defineix.
    - ▶ Per tant, els elements definits amb **let** són només visibles des d'aquest punt en avant.
  - ▶ Quan **let** s'utilitza en l'àmbit no global...
    - ▶ JavaScript considera que l'**àmbit local** abasta la funció sencera que defineix aquest àmbit.
    - ▶ Però **let** no utilitza un àmbit local de funció. En comptes d'això, defineix un “**àmbit de bloc**”.
      - Un “bloc” correspon a un conjunt d'instruccions dins d'un parell de claus.



## 8. Àmbit

### ► Exercicis:

- Executa els programes de la pàgina 73. Comprova la seua eixida. Reemplaça el “**var**” utilitzat en la línia 12 per un “**let**”. Executa una altra vegada els programes. Explica els nous resultats.
- En els programes originals, intercanvia el contingut de les línies 11 i 12. Executa els programes resultants. Pots explicar els nous resultats?
- Llig la documentació de MDN sobre [let](#), executa tots els exemples i explica els seus resultats.



## 9. Context d'execució

---

- ▶ El context d'execució es crea dinàmicament per a proporcionar un context vàlid per al codi que s'executa actualment.
- ▶ El context d'execució està compost per tots els elements de l'àmbit actual.
  - ▶ Conté totes les variables definides en el context actual (tant bloc com funció) i aquelles accessibles a través de la cadena d'àmbits.



## 9. Context d'execució

---

- ▶ Per a definir el context actual, es consideren aquestes etapes:
  - ▶ Quan el programa està començant, les seues funcions i variables globals es van creant i associant a l'objecte **global**. També es crea la referència **this** com a sinònim de **global**.
  - ▶ Cada vegada que s'invoque a una funció, i abans de començar la seua execució, es construeix el context d'aquesta funció, incloent les seues variables locals i paràmetres. Defineixen el seu àmbit local.
    - ▶ El valor de la referència a **this** pot canviar.
    - ▶ Aquest context nou s'afeg a la “pila de contextos d'execució” i a la cadena d'àmbits.

## 9. Context d'execució

### ► Exemple:

```
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 aquest exemple, a pesar que la variable gl1 s'ha definit al final del programa i computeResults() està definit després d'utilitzar-se, tots dos poden ser accedits sense generar errors.
  - A pesar que el valor per a gl1 encara es desconeix.

## 9. Context d'execució

### ► Exemple 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)
```

- Aquest segon exemple utilitza un valor conegut per a gl1.
- A més, ara formatResults() no utilitza paràmetres...
  - Utilitza el paràmetre “x” de la funció que la inclou, i que també està en la “pila de contextos d'execució”.

## 9. Context d'execució

### ▶ Exemple 3:

- ▶ S'escriurà un programa que podrà utilitzar un *Array* de funcions. Cada funció de l'*Array* gestionarà la taula de multiplicar de la seua posició en el vector.
  - ▶ Així, `tables[3]` hauria de ser una funció `f(x)` que retorne `x*3`.
  - ▶ Per tant, `tables[3](2)` hauria de retornar 6.
- ▶ Una primera solució (incorrecta) és:

```
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. Context d'execució

### ▶ Exemple 3 (cont.):

- ▶ Però aquest codi és incorrecte. Vegem per què...
  - ▶ Quan s'afeg un nou context d'execució a la seua pila? Quin és el valor de la variable i en aquest moment?
- ▶ Una primera solució a aquest problema la proporciona l'àmbit de bloc associat a **let**.
  - ▶ La línia 4 defineix el seu propi àmbit de bloc
    - Cada iteració del bucle defineix un nou àmbit de bloc
      - ...que es deixa en la pila de contextos d'execució en iniciar la iteració
      - ...i s'eliminarà de la pila en finalitzar la iteració.
  - ▶ Si se substitueix la línia 3 utilitzant aquest codi:
    - `for(let i=1; i<11; i++)`
  - ▶ **Quin és el resultat del programa en aquest cas? Per què?**
- ▶ La paraula **let** es va definir en ECMAScript 6
  - ▶ Les especificacions anteriors van solucionar aquest problema utilitzant clausures.



## 9. Context d'execució

- ▶ En una **clausura**, una funció interna manté el context d'execució existent quan va ser creada.
- ▶ Analitza aquest exemple i determina com s'utilitza la pila de contextos d'execució:

```
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. Context d'execució

---

- ▶ En l'exemple de la pàgina anterior, `table5` i `table10` són clausures, generades per `createTable()`.
  - ▶ Ambdues *recorden* l'argument rebut per `createTable()` i són una funció el codi de la qual depèn d'aquest argument.
  - ▶ Ambdues comparteixen la mateixa seqüència d'instruccions, però mantenen contextos d'execució diferents.
    - ▶ En el context d'execució de `table5`, `x` és 5
    - ▶ En el context d'execució de `table10`, `x` és 10



## 9. Context d'execució

---

### ▶ Exercici:

- ▶ Reescriu el programa mostrat en la pàgina 80, utilitzant clausures per a proporcionar una solució adequada.
  - ▶ A aquest efecte, s'hauria de reemplaçar la línia 4 original amb altres línies que definiren una clausura i assignar la funció retornada a la component de l'Array “tables”.



## 9. Context d'execució

- ▶ En el context global, hi ha un objecte el nom del qual és **global** (que pot ser accedit també utilitzant aquesta referència).
  - ▶ aquest objecte té diverses propietats.
    - ▶ Algunes d'elles són objectes que proporcionen informació sobre l'entorn d'execució.
- ▶ En NodeJS, una d'aquestes propietats és l'objecte process.
  - ▶ Una de les seues propietats és l'Array **argv**, que manté els arguments utilitzats en la línia d'ordres que va iniciar l'execució del procés.

```
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. Errors

- ▶ JavaScript és un llenguatge de programació en el qual és molt fàcil cometre errors i molt difícil detectar-los i corregir-los.
- ▶ Aquesta secció distingeix diversos tipus d'errors i proporciona algun consell sobre com gestionar-los i/o evitar-los.
  - ▶ Errors sintàctics
  - ▶ Errors semàntics
- ▶ Hi ha diverses referències (p. ex., [la de MDN](#)) que descriuen els missatges d'error més freqüents en JavaScript.



## 10.1. Errors sintàctics

---

- ▶ Els errors sintàctics són molt comuns quan comencem a programar en un nou llenguatge.
- ▶ Algunes de les seues causes habituals són:
  1. Les instruccions han sigut escrites d'una manera incorrecta.
  2. Hem utilitzat un identificador que encara no havia sigut definit.

## 10.1.1. Instruccions incorrectes

- ▶ Molts d'aquests errors seran detectats per l'editor (VS Code, en el nostre cas).

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

- ▶ Així i tot, a vegades, la gestió feble de tipus en aquest llenguatge pot causar que l'editor no detecte un error.

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



## 10.1.1. Instruccions incorrectes

- Un cas típic és que falte tancar algun parell de claus o parèntesis.

```
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 }
```

- En aquest cas, es generarà un missatge d'error. Normalment es refereix a algun *token* inesperat...

```
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. Identificadors no definits

- ▶ Aquesta classe d'errors pot ser causada per un identificador escrit incorrectament.
- ▶ L'interpret és incapaç de trobar la seua definició, i genera un *ReferenceError*
- ▶ Inclou el número de línia

```
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$ |
```

- ▶ Hem de revisar aquesta línia i corregir l'error.



## 10.1.2. Identificadors no definits

- ▶ Cal advertir que JavaScript distingeix entre majúscules i minúscules en escriure els identificadors.

```
1  function computeResult(x) {  
2      |     return x*2  
3      |  
4      |  
5      |  
6      |  
7  }
```

```
5  console.log(computeresult(15))  
6  console.log(ComputeResult(20))
```

- ▶ En aquest exemple, tant les línies 5 com 6 generaran un *ReferenceError*.



## 10.1.2. Identificadors no definits

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

- ▶ Aquest programa no genera cap error
  - ▶ No és obligatori definir un variable precedint-la amb **var** o **let**.
  - ▶ Quan cap d'aquestes paraules clau s'utilitza, llavors la variable té un àmbit global.



## 10.2. Errors semàntics

---

- ▶ **Els errors semàntics** són aquells relacionats amb l'execució del nostre codi.
  - ▶ En aquests errors, els missatges proporcionats per l'interpret són només una pista.
  - ▶ Un subconjunt important d'aquests errors està causat per la invocació incorrecta d'una funció.
    - ▶ Per exemple: la funció necessita un argument d'un tipus donat, però s'ha cridat utilitzant un valor incompatible.

## 10.2. Errors semàntics

```
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 funció `sum()` assumeix que el seu argument serà un *Array*
  - ▶ Així podrà utilitzar el seu mètode `reduce()`
  - ▶ Si es passa una cosa diferent, llavors no tindrà aquest mètode i això generarà un *TypeError* durant l'execució

## 10.2. Errors semàntics

- ▶ En el programa de la pàgina anterior, la línia 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 missatge d'error pot arribar a ser confús...
  - ▶ Declara que “A .reduce” no és una funció, però...
    - Això només significa que A no és un Array.
    - Observa que en la línia 5 es va passar el valor 1 com l'argument de sum()
      - És un enter en comptes d'un Array!!!

## 10.2. Errors semàntics

- ▶ Per a evitar aquests errors, hauríem de comprovar el tipus assumit en els seus paràmetres.
- ▶ A aquest efecte, hauríem d'utilitzar:
  - ▶ `typeof`, per a tipus primitius
    - ▶ Un exemple va aparèixer en la pàgina 24
  - ▶ `instanceof`, per a classes d'objecte
    - ▶ Com es mostra en aquest exemple:

```
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. Errors semàntics

- ▶ Per a evitar aquests errors, hauríem de comprovar el tipus assumit en els seus paràmetres

- ▶ A aquest efecte, hauríem d'utilitzar

- ▶ `typeof`, per a tipus primitius

- ▶ Un exemple va aparèixer en la pàgina

- ▶ `instanceof`, per a classes d'objectes

- ▶ Com es mostra en aquest exemple

La línia 2 comprova si l'argument *A* és *un Array*. Si no ho és, es genera una excepció en la línia 3, indicant que l'argument hauria de ser d'aquest tipus.

```
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))
```