Lab 0 – Introduction to JavaScript

Network Information System Technologies

Index

Index of exercises

1. Introduction

- JavaScript is a programming language widely used in distributed systems.
- Its is based on the ECMAScript specifications.
  - We'll use ECMAScript 6 in this course.
- In the scope of NIST, JavaScript is an appropriate tool in order to reach several pedagogical goals.
  - But we are not interested on an in-depth learning of this language nor in all its libraries.

2. Goals

- To explain a basic set of characteristics of JavaScript in order to easily understand Unit 2...
  - ...and be able to start Lab 1.
- To realise that JavaScript is a peculiar programming language.
  - It is not similar to Java.
  - It needs an interpreter (instead of a compiler).
  - It is hard to learn in a short time if some basic concepts haven't been explained and understood.
- To introduce the set of tools to be used in the labs.
  - These two initial weeks should be devoted to learn those tools.

2. Goals

- To take care and correctly interpret the error messages.
  - Those messages are sometimes unclear.
    - But we should focus on them, since they provide hints on the causes of each error.
    - Those hints should be adequately understood.
    - Some guide on this issue is given in this presentation.

3. Tools

- We need, at least, these tools:
  - A text editor
    - In order to write our programs and modifiy them.
    - There is a wide variety of editors.
      - Each one may provide some useful characteristics:
        - Debugging support
        - Syntax highlighting
        - Customisation
        - A collection of complementary plugins
        - API documentation
        - Version control
  - An interpreter
    - In order to run our programs
    - We'll use NodeJS with its "node" command

3.1. Text editor

- We'll use Visual Studio Code in the labs
  - It is a multi-platform editor with a rich set of useful characteristics
  - It may be downloaded from https://code.visualstudio.com/Download
  - Its documentation is available at https://code.visualstudio.com/docs
  - It is already installed in the DSIC EVIR (http://www.upv.es/entidades/DSIC/infoweb/dsic/info/1043006normali.html)
    - EVIR is the remote virtual desktop provided by the DSIC department in order to use a computing environment similar to that used in the labs.
      - From EVIR, we may reach our virtual machines, where the lab projects should be developed and run.

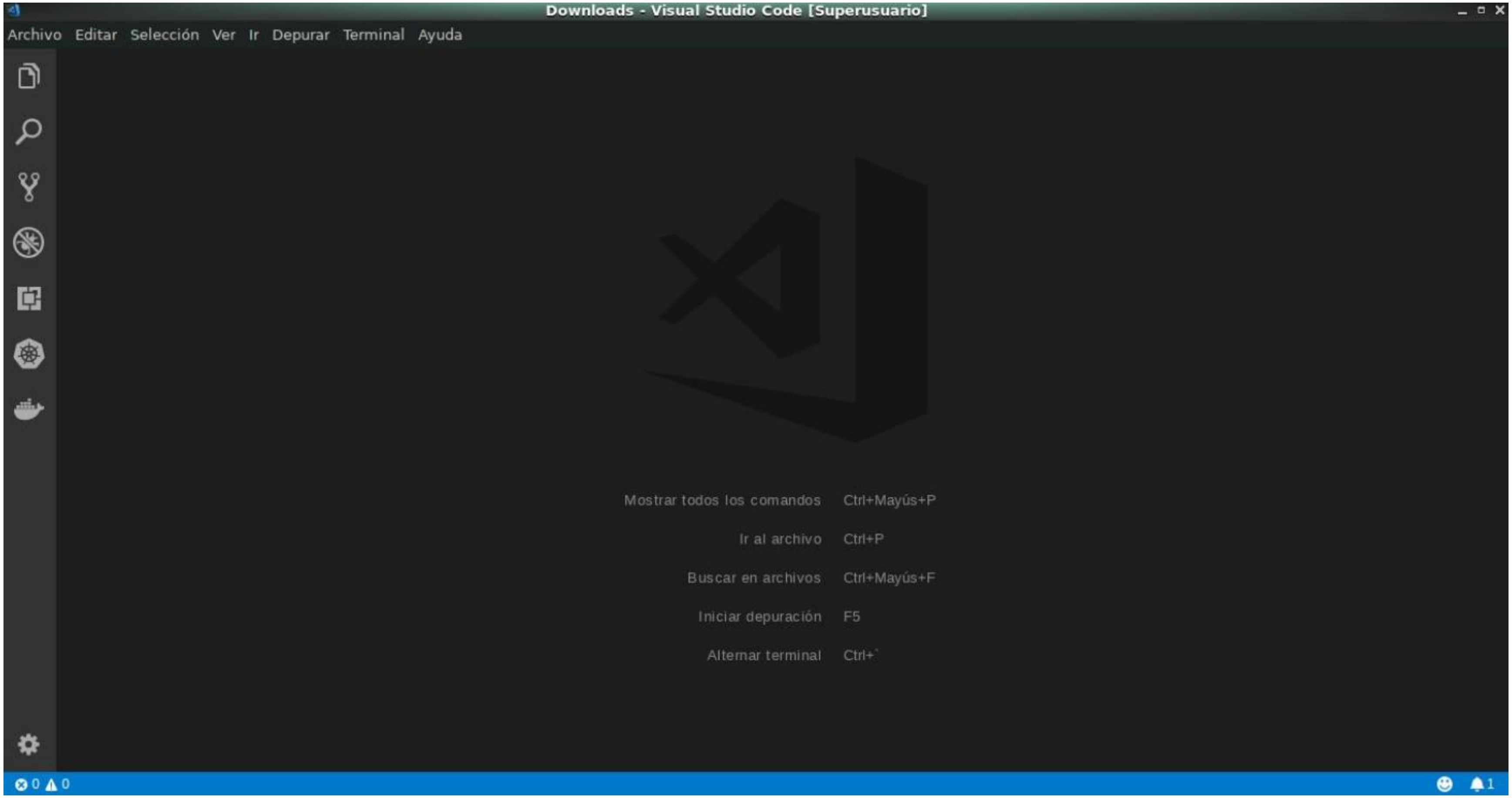
3.2. Interpreter

- We'll use NodeJS
  - It is the interpreter to be used in the classroom and in the labs
  - It may be downloaded from https://nodejs.org/en/download/
    - The installed version in the labs is 10.16.0
  - Documentation available at https://nodejs.org/en/docs/
    - In case of the API, read the latest 10.x.y version (https://nodejs.org/dist/latest-v10.x/docs/api/)
  - It is also installed in the DSIC EVIR.

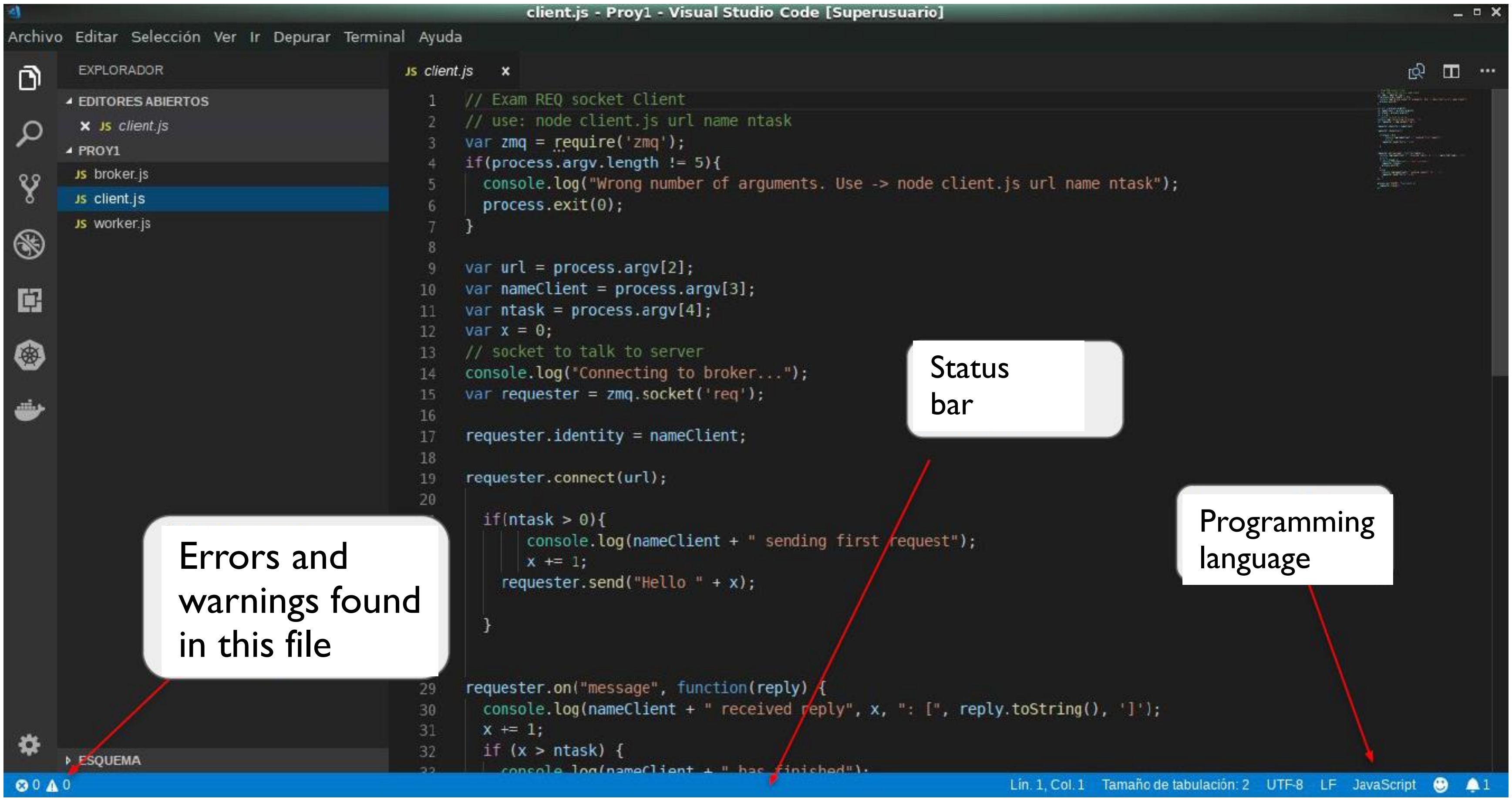
3.3. Using VS Code

- Visual Studio Code (VS Code) is a text editor that manages separate files, folders or projects.
- It arranges its elements in a simple way.
  - There are several icons in a small panel on the left:
    - File explorer ( ): In order to open files or accessing folders.
    - Search ( ): It looks for any text in the current file or folder.
    - Version control ( ): In order to integrate our project into a version control system.
    - Debugging ( ): It facilitates the debugging of our program using break points or exploring the current values in the program variables.
  - Each of those icons shows a menu with several related actions.

3.3. Using VS Code



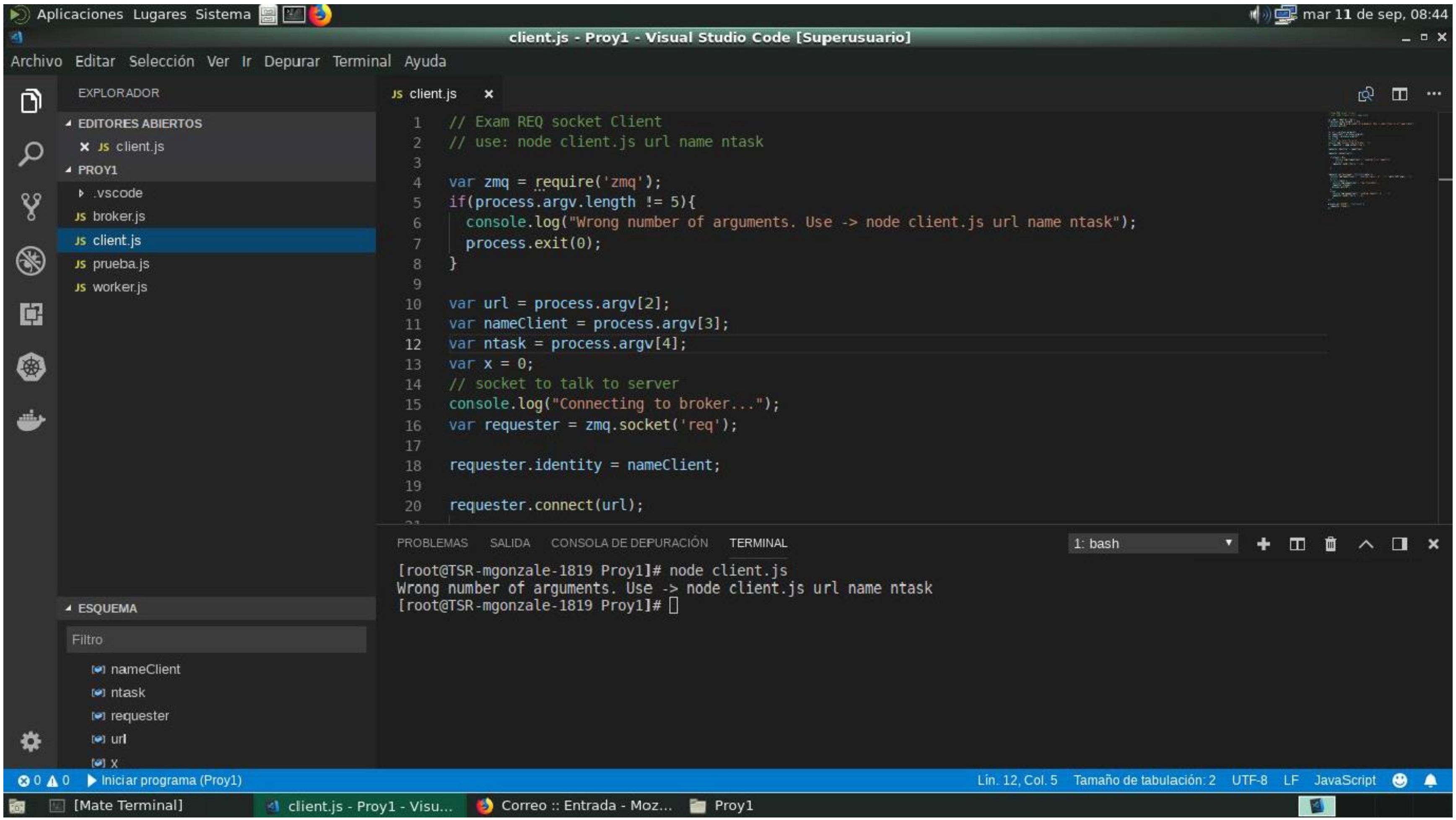
3.3. Using VS Code



3.3. Using VS Code

- Every file with the ".js" extension is identified as a JavaScript program
  - Its syntax is appropriately highlighted
- In order to run any JavaScript program, we may use...
  - The "node" interpreter in a separate terminal, or
  - The terminal embedded in VS Code (Ctrl + `)
    - As it is shown in the next page

3.3. Using VS Code



3.3. Using VS Code

- Some operations allowed by VS Code:
  - To split the windows in two vertical halves, in order to compare two files.
  - To keep a history of the edited files, remembering the last position edited in each file.
    - That history may be scanned...
      - Backwards (Ctrl + Alt + -)
      - Forward (Ctrl + Shift + -)

3.4. Using the interpreter

- Once installed, we may use the NodeJS interpreter running this command:

```
node program.js [list of arguments]
```

- ...where:
  - The ".js" extension is not mandatory.
  - The arguments needed by the program, if any, should follow its name.

4. The JavaScript language

- Some distinguishing characteristics of this language are:
  - Dynamic typing
  - Weakly typed
  - Primitive types
  - Type coercion
- Let us assume this code fragment in order to evaluate those characteristics:

```
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[0])
```

4. The JavaScript language

- Questions about the program shown in the previous page:
  - Is there any error in the execution of that program?
  - Which are the differences when it is compared with a Java program?
  - May we use a variable without any previous definition of it with “let” or “var”?

4.1. Dynamic types

- JavaScript uses dynamic types
  - It does not set the type of any variable
    - Their type depends on the assigned values
  - There is some freedom when array slots are accessed
    - Some times, they have no assigned value
      - But this does not generate any error!
      - "undefined" is returned!
- Dynamic types may be useful when they are appropriately used
  - But they are also a source of errors!!
- **Dynamic typing:** Variables change their type depending on their assigned value, and that value may change while the program is running.

4.2. Weakly typed

- JavaScript is a **weakly typed programming language**.
  - This means that its expressions do not check semantically whether their operators are applied onto variables with their intended types.
  - Again, this is very flexible...
  - ...but it is also error-prone, as shown in this example:

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

4.2. Weakly typed

- The example shown in the previous page…
  - …shows that we may build expressions that combine values from different types.
    - JavaScript has some evaluation rules in order to determine the resulting type.
- Questions on that example:
  - Is there any unexpected result?
  - Is there any (apparently) incorrect expression?
    - Try them all and check the results.
  - May we write similar expressions in our programs?

5. Primitive data types

- The JavaScript primitive data types are:
  - Number
  - Boolean
  - String
  - undefined
  - null
    - Although it may be considered a value, ECMAScript considers it a type
  - Symbol
    - We will not consider it in this presentation
- A data type is **primitive** when it is simple (i.e., not structured) and there is a way for obtaining the type of a variable that belongs to it (operator **typeof**)

5.1. Primitive data types: undefined

- **undefined** is a data type that corresponds to all those variables that have not been assigned yet any value
  - i.e., it corresponds to uninitialised variables

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

- **undefined** is also used when a function parameter has not received any value when that function is called.
  - Example:
    - When a function with three parameters has been called using a single argument, the second and third parameters become **undefined**.

5.1. Primitive data types: undefined

- **undefined** should also be used in order to check whether a variable has already been assigned any value.
  - To this end, we should use the **typeof** operator, as shown in this example:

```
let result

if (typeof result != "undefined")
    console.log(result)
    else console.log("The result is not yet defined!")
```

  - Exercise:
    - Look for other ways of checking whether a variable is **undefined** or not.

5.2. Primitive data types: null

- **null is a value** assigned to Object variables that have not been assigned any value yet.
  - ECMAScript considers **null** a primitive data type, although it has been traditionally used as a literal value in JavaScript.
    - There is no conflict between these two interpretations
  - Some functions return **null** in order to mean that they have not found any appropriate object.

5.3. Primitive data types: Number

- **Number** is a type that corresponds to both integer and floating point numbers.
  - Floating point numbers have a limited precision
  - Example:

```
let x=0.2
let y=0.29999999999999999

if (x+y==0.5)
    console.log("The result is inaccurate.")
```

5.3. Primitive data types: Number

- **NaN** is a result that determines that an operation did not make sense.
  - Examples of operations of this kind:
    - **0/0**
      - However, the operations **value/0** do not return **NaN**. They return **Infinity**, instead.
    - **Infinity – Infinity**
    - Mathematical operations where **undefined** is used as an operand.
    - Mathematical operations that use an inappropriate operand type
      - E.g., "My name" * 3
    - Functions that expect a Number as an argument and do not receive a value of that type
      - E.g., parseInt("a string")

5.4. Primitive data types: String

- **String** objects have a default property: length.
  - It states the amount of characters in that string.
- **String** literals or objects may be concatenated using the operator **+**

```javascript
let s1 = "This is an example."
let s2 = "A short sentence. "

console.log(s1.length)
let s3 = s2 + s1
console.log(s2.length)
console.log(s3.length)
console.log(s3)
```

5.5. Primitive data types: Errors and types

- JavaScript is weakly typed. Sometimes that characteristic may be the origin of errors.
  - Example:

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

  - Questions:
    - What is the result in this example?
    - Why have we obtained that unexpected value?
  - **NaN** and **undefined** may be the source of many programming errors.
    - We should understand why those values are generated in some statements

5.6. Primitive data types: Type coercion

- What happens if an expression mixes different data types?
  - Since JavaScript is a weakly typed programming language, it does not generate any error.
    - Instead, it applies some rules in order to transform that expression into something that makes sense.
    - To this end, some operands are coerced into values of the expected data types.
- **Type coercion** means (according to the 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."*
  - [Translation] "A characteristic of programming languages that allows, implicitly or explicitly, the conversion of an element from one data type to another, without considering any type checking."

5.6. Primitive data types: Type coercion

- Let us consider a previous example:

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

- Questions:
  - Can you understand which rules manage the type coercions applied in that example?
  - Have they been successfully applied?
  - Are they useful?

5.6. Primitive data types: Type coercion

- Type coercion rules are also applied to logical expressions.
- **Exercise**: Let us try some examples (using **if** statements):
  - Check whether the String literal "5" is greater than 3.
  - Check whether a variable with value "6" is equal to 6.
  - Check whether the String literal "user" is **false**.
  - Check whether the empty string ("") is **false**.
  - Check which Boolean value corresponds to **undefined** and **NaN**.
    - Because of this, what happens when we compare the current value of a variable with **undefined**?
      - This explains why **undefined** is considered a type instead of a literal value.

5.6. Primitive data types: Type coercion

- Sometimes, we may control type coercion using some operators that convert one type into another.
  - Examples: Boolean(), String(), Number, parseInt(), parseFloat()...

```
Number(true)          // Returns 1
Number(false)         // Returns 0
Number("10")          // Returns 10
Number("  10")        // Returns 10
Number("10 20")       // Returns NaN
Number("John")        // Returns NaN
String(10.6)          // Returns "10.6"
String(true)          // Returns "true"
parseInt("10.33")     // Returns 10
parseInt("10 years")// Returns 10
parseFloat("10")      // Returns 10
parseFloat("10.33") // Returns 10.33
```

5.6. Primitive data types: Type coercion

- Exercise:
  - Determine the result of these operations:
    - Boolean("false")
    - Boolean(NaN)
    - Boolean(undefined)
    - Boolean("undefined")

5.6. Primitive data types: Type coercion

- Type coercion may avoided when we use the strict comparison operator (“===“) instead of the regular comparison operator (“==“).
  - Examples:

```
console.log(null == undefined)        // true
console.log(null == 0)                // false
console.log("5" == 5)                 // true
console.log(NaN == NaN)               // false ??

console.log(null === undefined)       // false
console.log("5" === 5)                // false
console.log(NaN === NaN)              // false ??
```

5.6. Primitive data types: Type coercion

- Type coercion may be used in order to simplify conditions.
  - Examples of conditions:
    - Empty string checking:

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

    - Whether a variable has been defined or not:

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

      - When *person* is **undefined** or **null** this statement works as expected.
      - However, what happens when *person* is 0 or the empty string?
  - Exercise:
    - How can we check whether variable *person* has been defined, considering also values 0 and empty string?

5.6. Primitive data types: Type coercion

- Solutions to the exercise in the previous page:
  1. Using strict comparison:

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

  2. Without type coercion:

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

- Questions:
  - What happens when we remove line 1?
  - In the second solution, may we use **!=** instead of **!==**?

6. Structured types

- JavaScript provides several structured types that hold multiple elements of primitive types:
  - Arrays: Sequences of values that may be accessed using indexes.
  - Objects: Sequences of key/value pairs.
  - Collections: This kind of structured type isn't used in this subject.

6.1. Structured types: Arrays

- Arrays are list-like built-in JavaScript objects...
  - ...with a **length** property
    - It states how many elements are in the array
  - ...and some methods
    - indexOf(), pop(), push(), shift(), map(), slice()...
- Documentation on arrays may be found at different sites
  - e.g., Mozilla Developer Network, w3schools.com,...
- Example:

```
let users = ["Chloe", "Martin", "Adrian", "Danae"]

for (let c=0; c<users.length; c++)
    console.log(users[c])
```

6.1. Structured types: Arrays

- Because of the JavaScript characteristics, the insertion of elements and the access to array elements that have not been defined yet are tasks that should be done with care.

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

6.1. Structured types: Arrays

- We cannot copy an array assigning its "value" to another variable:

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

  - In that case, we are copying a reference to the array object.
  - Because of this, we have two references (i.e., variables that refer) to the same array.

6.1. Structured types: Arrays

- We cannot copy an array assigning its "value" to another variable:

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

- Instead, we should use the **slice()** method.
  - When no argument is used, slice() returns a copy of the array.
  - There are two optional parameters in slice():
    1. The index of the first element to be copied. If this argument is undefined, the copy starts at index 0.
    2. A value one unit greater than the index of the last element to be copied. By default, it is assumed as the length of the array.

6.1. Structured types: Arrays

- In order to insert elements to an array, we may use their intended indexes...
  - But this overwrites the previous contents in those slots
  - There are other operations that add the new elements at the beginning or at the end of the array, shifting or keeping the previous contents, respectively.
  - Similarly, there are other operations in order to remove elements:

	ADD	REMOVE
At the beginning	unshift(elem1,...)	shift()
At the end	push(elem1,...)	pop()

6.1. Structured types: Arrays

- There are some array-like objects that in some cases should be converted into arrays.
- To this end, we may use the Array.from() method.
- Example that uses the **arguments** default pseudoarray:

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

  - In previous releases of the ECMAScript standard the Array.from() operation did not exist.
    - Instead, we used **Array.prototype.slice.call(arguments)**.

6.2. Structured types: Objects

- An object is an unsorted set of key/value pairs (where "key" is the equivalent to a "property" in traditional object-oriented programming).
  - The values of those keys or properties may be either literals from primitive types, functions or other objects.
  - Example:

```javascript
let person = {  name: "Peter",
                age: 25,
                address: {
                    city: "Valencia",
                    street: "Tres Cruces",
                    number: 12
                }
            }
console.log(person)
```

6.2. Structured types: Objects

- Objects may also be created in a dynamic way.
  - Example:

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

- However, the static way shown in the previous page is faster and more common.
- **Be careful!!** Since dynamic declaration/creation is possible, if we incorrectly type the name of any property and assign a value to it...
  - No error will arise
  - **But such wrong name will be set as another property in our object!!**

6.2. Structured types: Objects

- Objects may also be created in a dynamic w
  - Example:

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

There is a third syntax to state the properties of an object: they may be placed in brackets! (as in arrays)

Therefore, these lines are equivalent to those in the example:

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 // Let us combine both syntaxes!
console.log(person)

6.2. Structured types: Objects

- Objects may be created or modified dynamically, but…
- Exercise:
  - Explain what happens when we access a property that has not been defined.
    - Try these examples in order to answer:

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

```
function printDistrict(who) {
    console.log("District: "+who.district)
}
let person={name:"Peter",
            age:25,
            address: {
                city:"Valencia",
                street:"Tres Cruces",
                number:12
            }
        }
printDistrict(person)
```

6.2.1. Structured types: Objects. JSON

- JSON (JavaScript Object Notation) is a textual format used for object serialisation in order to transfer objects through the network.
  - Each property identifier is enclosed in double quotes.
  - In order to deal with JSON formatting, we may use...
    - JSON.stringify(object) converts a JavaScript object into a JSON string.
    - JSON.parse(string) converts a JSON string into a JavaScript object.

6.2.1. Structured types: Objects. JSON

- Example:
  - Let us consider this program...

```
let person = {  name: "Peter",
                age: 25,
                address: {
                    city: "Valencia",
                    street: "Tres Cruces",
                    number: 12
                }
            }
console.log(JSON.stringify(person))
```

  - Its output, in JSON format, is...

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

6.2.2. Structured types: Objects. Loops

- We may use a **for(***variable* **in** *object***)** loop in order to process every property in a given object.
- Example:

```
let person = {  name: "Peter",
                age: 25,
                address: {
                    city: "Valencia",
                    street: "Tres Cruces",
                    number: 12
                }
            }
for(let i in person)
    console.log("Property "+i+": "+ person[i])
```

  - Variable **i** gets the name of each property in each iteration of this loop.
  - With that name, we may access to the value of each property
    - To this end, we should know that **an object is similar to an array and its properties are indexes in that array**.
    - This characteristic may be used when property names are held in other variables.

6.2.2. Structured types: Objects. Loops

- In the previous example, the obtained output was…

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

- Exercise:
  - Extend the previous example, showing the properties and values in the "address" property.
    - A general solution to this exercise requires the usage of functions, to be explained in the next section!

7. Functions

- The concept of "function" is similar to that used in any other programming language.
  - A sequence of statements that may be called from other parts in the program.
  - A function defines a clear interface in order to interact with that fragment of code.

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

7. Functions

- If a function does not use the **return** statement, then its execution returns the **undefined** value.

```
function greet(person) {
    console.log("Hello, "+person+"!!")
}
console.log(greet("Peter"))
```

7. Functions

- Function parameters behave as variables whose scope is limited to the code of their function.
- Run the following example, and observe that...

```
function add(x,y,z) {
    return x+y+z
}

console.log(add(1,2,3))
console.log(add(2,7))
console.log(add(null))
console.log(add(1,2,3,4,5))
```

  - When a function is called using less arguments than declared parameters, then those unused parameters receive the **undefined** value.
  - When a function is called using more arguments than declared parameters, then those exceeding arguments are ignored.
    - No error is generated in both cases!

- If some function parameters are optional, we may assign default values to them in their declaration.
  - Thus, we do not need to check whether they are **undefined** in the body of the function.

```
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))
```

  - With this, lines 6 and 7 no longer print **NaN** since all add arguments are now integer numbers.

7. Functions

- Functions that have an unknown number of arguments may use the "rest" parameter...
  - To this end, the name of the last specified parameter should be preceded by an ellipsis, i.e., **...name**
  - Such parameter is an array that holds all remaining arguments.
  - Example:

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

7. Functions

- Exercise:
  - Assuming, the program shown in the previous page, what is the result of the following statement?

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

7. Functions

- Arguments are passed...
  - by value if they belong to a primitive type
  - by reference when they are objects
    - Note that arrays are also objects
    - We may change the contents of the object, but we cannot change the received reference.

```javascript
function changeColour(car, newColour) {
    return car.colour = newColour
}
function changeCar(car) {
    car={brand:"Ferrari", colour:"Red"}
}
let myCar={brand:"Volvo", colour:"Grey"}
console.log(changeColour(myCar,"Blue"))
changeCar(myCar)
console.log(myCar)
```

7. Functions

- JavaScript manage functions as common objects, thus they may be...
  - used as values, that can be assigned to variables
  - used as arguments in calls to other functions
  - returned as the result of other functions

```
function square(x) {return x*x}
let a = square
let b = a(3)
let c = a

console.log(a)
console.log(b)
console.log(c)
```

- We should distinguish the following uses of functions:
  - Their initial definition.
  - Their usage in expressions may be...
    - A reference, when only their identifier is used.
    - The result of its invocation, when parentheses (and any required arguments) are used.

7. Functions

- Example:

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

7. Functions

- Functions may be defined anonymously, i.e., without giving them any name.
  - The following program is equivalent to that shown in the previous page:

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

7. Functions

- The anonymous functions are widely used as arguments in the invocation to other functions.

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

7. Functions

- **Arrow notation**
  - Anonymous functions are widely used. Therefore, a more concise syntax makes sense → The "arrow" notation
    - The keyword **function** is dropped
    - The list of arguments is kept
      - Parentheses may be also dropped when there is a single argument
    - Such list is followed by this arrow **=>**
    - Later, a statement that computes the returned value is found
      - Or a list of statements inside curly braces.

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

7. Functions

- Therefore, this statement

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

- ...is equivalent to...

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

7. Functions

- Exercises:
  - Write a function doCheckPasswd() that uses three parameters:
    - input
    - correctPassword
    - fun
  - It compares the strings passed in the first two parameters.
  - If they are equal, then the function passed as the third argument is called.
  - Test it with the following calls:

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

7. Functions

- Exercises:
  - Extend the program shown in page 62, writing another function doWithNFirstNumbers() with 3 parameters:
    - n: The last natural number to be used
    - op: Function to be applied on each processed natural number
    - op2: Function to be applied on the result of op(i) in order to accumulate all those results
      - op2 should be chosen from those functions placed in the arithmeticOperations array
  - doWithNFirstNumbers() applies op() on all natural numbers in the range 1..n, and accumulates the results using op2 to this end.
  - Examples of invocations:

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

7. Functions

- There are many functions that use other functions as their parameters. For instance:
  - Array.map()
    - Creates a new array with the results of the function stated as its first argument applied on each of the original array elements.
    - map() calls that function with three arguments:
      - The element on which the function should be applied
      - Its index
      - The original array

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

7. Functions

- Exercise:
  - Modify the example shown in the previous page, using traditional notation (instead of the arrow one) in order to write the function that is passed as an argument to map().

8. Scope

- **NOTE**: **This part of the presentation will be explained in depth in Unit 2.**
- The scope of the elements (variables, functions…) in a program is determined by the location of their definitions.
- There are two traditional scopes in JavaScript:
  - Global
  - Function (also known as local)
- Elements in the **global scope** (i.e., those that have not been defined inside any function) may be accessed from any location.
- On the other hand, every function defines its own **local scope**.

8. Scope

- When a program is run, elements defined in a local scope may be accessed from:
  - that local scope
  - or in the scope of other functions placed in that local scope → **children scope**
- This defines a hierarchy of scopes.
  - When a program runs a sequence of function calls, such a sequence defines a **scope chain**
    - It determines which elements in other enclosing scopes may be accessed from the current one.
    - If a function or variable is defined in any part of that chain and its name coincides with that of a global element, then the local element is used.

8. Scope

- Thus, in a program like this…

```
function a() {
    let a1=1
    b(a1)
}
function b(p1) {
    let b1=2
    console.log(a1)
    console.log(b1)
    console.log(gl1)
}
a()
let gl1=0
```

- …a1 cannot be accessed in b(). An error is raised!!
- Question:
  - There are two ways in order to allow that b() uses a1. Which are they?

8. Scope

Solution A

```
function a() {
    let a1=1
    b(a1)
}
function b(p1) {
    let b1=2
    console.log(p1)
    console.log(b1)
    console.log(gl1)
}
a()
var gl1=0
```

Solution B

```
function a() {
    let a1=1
    b()
    function b(p1) {
        let b1=2
        console.log(a1)
        console.log(b1)
        console.log(gl1)
    }
}
a()
var gl1=0
```

- A passes a copy of a1 to b(). So, b() may read a1, but it cannot modify it.
- B defines b() as a function internal to a(). So, a1 is visible to b(). Therefore, it may both read from and write to a1.

8. Scope

- The **let** keyword has its own scope:
  - When **let** is used in the global scope...
    - It does not manage variables or functions as properties of the global object.
      - The **var** keyword manages those elements as properties of that global object.
        - So, they are visible even before running the statement that defines them.
    - Therefore, elements defined using **let** are only visible from that point onwards.
  - When **let** is used in a local scope...
    - JavaScript considers that a local scope encompasses the entire function that defines that scope.
    - But **let** does not use such a current function local scope. Instead, it defines a "block scope".
      - A "block" corresponds to a set of instructions inside a pair of curly braces.

8. Scope

- Exercises:
  - Run the programs shown in page 73. Check the printed values. Replace the "var" keyword used in line 12 with a "let". Run again the programs. Explain the new results.
  - In those original programs, exchange the contents of lines 11 and 12. Run the resulting programs. Can you explain the new results?
  - Read the contents of the MDN documentation on **let**, run all the examples and explain the shown results.

9. Execution context

- The execution context is dynamically created in order to provide a valid context for the code that is currently run.
- The execution context is composed of all the elements that are in the current scope.
  - It contains all variables defined in the current context (either block or function) and those accessible through the scope chain.

9. Execution context

- In order to define the current context, these stages are considered:
  - When the program is started, global variables and functions are created and associated to the **global** object. The **this** reference is also created, referring to **global**.
  - Each time a function is called, and before starting its execution, the context for that function is built, including its local variables and parameters. They define its local scope.
    - The value of the **this** reference may change.
    - This new context is pushed on top of the "execution context stack" and is appended to the scope chain.

9. Execution context

- Example:

```
computeResults(10)
function computeResults(x) {
    let y=formatResults(x)
    console.log(gl1+" "+y)
    function formatResults(inp) {
        return String(inp)
    }
}
var gl1="GlobalContext1"
```

- In this example, although variable gl1 is defined at the end of the program and computeResults() is defined after using it, both can be accessed without generating errors.
  - Although the gl1 value is not known yet.

9. Execution context

- Example 2:

```
function computeResults(x) {
    let y=formatResults()
    console.log(gl1+" "+y)
    function formatResults() {
        return String(x)
    }
}
var gl1="GlobalContext1"
computeResults(10)
```

- This second example uses a known value for gl1
- Besides, now formatResults() does not use any parameter…
  - It uses the "x" parameter from its enclosing function, that is also in the "context execution stack".

9. Execution context

- Example 3:
  - Let us write a program that should use an array of functions. Each function in that array should manage the multiplication table associated to its index (i.e., position) in the array.
    - Thus, table[3] should be a function f(x) that returns x*3.
    - Therefore, table[3](2) should return 6.
  - A first (wrong) solution is:

```javascript
let tables=[]

for (var i=1; i<11; i++)
    tables[i]=x=>x*i

console.log(tables[5](2))
console.log(tables[9](2))
```

9. Execution context

- Example 3 (cont.):
  - But this code is incorrect. Let us find why it is incorrect...
    - When is a new execution context appended to the execution context stack? What is the value of variable i at that point?
  - A first solution to this problem is provided by the block scope associated to the **let** keyword.
    - Note that line 4 defines its own block scope
      - So, each iteration in that loop visits a new block scope
        - that is pushed to the execution context stack when the iteration is started
        - and popped from that stack when the iteration ends
    - Replace line 3, using instead this statement:
      - for (let i=1; i<11; i++)
    - **What is the result of the program in this case? Why?**
  - The **let** keyword was introduced in ECMAScript 6
    - Previous specifications solved this same problem using closures.

9. Execution context

- A **closure** is a function that maintains the execution context that was present when it was created.
- Analyse this example and determine how it uses the execution context stack:

```javascript
function createTable(x) {
    return y=>x*y
}

let table5=createTable(5)
let table10=createTable(10)

console.log(table5(2))  // Shows 10
console.log(table10(2)) // Shows 20
```

9. Execution context

- In the example of the previous page, table5 and table10 are closures, generated by createTable().
  - Both *remember* the argument received by createTable() and return a function whose code depends on that argument.
  - Both share the same function body, but they keep different execution contexts.
    - In the execution context of table5, x is 5
    - In the execution context of table10, x is 10

9. Execution context

- Exercise:
  - Rewrite the program shown in page 80, using closures in order to provide an adequate solution.
    - To this end, you should replace the original line 4 with other lines that define a closure and assign the returned function to the tables array slot.

9. Execution context

- In the global context, there is an object whose name is **global** (that may be accessed also using the **this** reference).
  - That object has several properties.
    - Some of those properties are objects that provide information about the execution environment.
- In NodeJS, one of those properties is the **process** object.
  - One of its properties is the **argv** array, that holds the command-line arguments used for starting the execution of that program.

```
// First two elements are:
// + "node": the name of the intepreter
// + program-name: the name of this file
// They are discarded in this example!
let procArgs = process.argv.slice(2)

console.log(procArgs)
```

10. Errors

- JavaScript is a programming language in which it is very easy to cause errors and very difficult to detect and fix them.
- This section distinguishes several types of errors and provides some advice on how to manage and/or avoid them.
  - Syntax errors
  - Semantic errors
- There are several references (e.g., the MDN one) that explain the common JavaScript error messages.

10.1. Syntax errors

- Syntax errors are very common when we start programming in a new language.
  - Some of their usual causes are:
    1. Statements have been written in an incorrect way.
    2. We have used an identifier that has not been defined yet.

10.1.1. Incorrect statements

- Most of them will be detected by the editor (VS Code, in our case).

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

  - However, sometimes, the weakly typed nature of the language may cause that the editor does not detect an error.

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

10.1.1. Incorrect statements

- A typical case is that of a missing closing brace or closing parenthesis…

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

- If so, an error message is generated. It usually refers to an unexpected token…

```
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
```

- This kind of errors may be caused by an incorrectly typed identifier.
  - The interpreter is unable to find its definition, and it generates a ReferenceError
    - It includes the line number

```
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$ |
```

  - We must revise that line and fix the error

10.1.2. Undefined identifiers

- Note that JavaScript is case-sensitive!!

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

- In this example, both lines 5 and 6 would generate a ReferenceError!

10.1.2. Undefined identifiers

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

- This program does not generate any error
  - It is not mandatory to define a variable preceding it with **var** or **let**
  - When none of those keywords is used, then the variable has a global scope

10.2. Semantic errors

- **Semantic errors** are those related with the execution of our code.
  - In these errors, the messages provided by the interpreter are only a hint.
  - An important subset of these errors is caused by an incorrect invocation of a function.
    - For instance: the function needs a parameter of a given type, but it is called using an incompatible element.

10.2. Semantic errors

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

- Function sum() assumes that its argument will be an array
  - So, it uses its reduce() method
  - If something else is passed, it will not have such method and this will cause a TypeError at run time

10.2. Semantic errors

- In the program of the previous page, line 5 generates an error...

```
    return A.reduce((x,y)=>x+y)
             ^

TypeError: A.reduce is not a function
    at sum (C:\Users\fmunyoz\Documents\tsr\Lab00\example51.js:2:14)
    at Object.<anonymous> (C:\Users\fmunyoz\Documents\tsr\Lab00\example51.js:5:1
3)
    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
```

  - The error message may become unclear...
    - It states that "A.reduce" is not a function, but...
      - This only means that A was not an array!!!
      - Note that in line 5, we passed value 1 as the argument to sum()
        - It was an integer instead of an array!!!

10.2. Semantic errors

- In order to avoid those errors, we should check the type of the expected parameters.
- To this end, we should use:
  - **typeof**, for primitive types
    - An example can be shown in page 24
  - **instanceof**, for object classes
    - As it is shown in this example:

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

10.2. Semantic errors

- In order to avoid those errors, we should check the type of the expected parameters.
- To this end, we should use:
  - **typeof**, for primitive types
    - An example can be shown in page 2
  - **instanceof**, for object classes
    - As it is shown in this example:

Line 2 checks whether the A actual parameter is an array. If not, an exception is thrown in line 3, stating that the expected argument should be an array.

```
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))
```