NIST: Lab 1. Session 1
Running JavaScript programs

Introduction

During this first session you must execute, understand and be able to reason about different features of the JavaScript language, especially if we compare this language with a language that we know more about, such as Java.

We are going to work on the following aspects:

1. Hello world in JavaScript.
2. Variable scope
3. Variable types
4. Function arguments.
5. JavaScript classes
6. Closures
7. Asynchronous programming using callbacks.

Note : You have support material with various documents and examples that can help you become familiar with the JavaScript language. All of this is contained in **Practice 0: non-guided, non-face-to-face practice**, which we believe every student should do as a step prior to developing the practices. Practice 0 should allow you to work with the next examples in a much easier way. There are multiple examples and explanations of various concepts that we continue to work on and expand on in this practice session. **If you have not done practice 0,** you should review the documents of that practice as they contain material that will facilitate your work in this practice and the following ones.

1. - Hello world in JavaScript

With this example we review the minimum elements of a JavaScript program.

1.1 Read and execute the program “ej1_HelloWorld.js”

- Notice the use of the "use strict" directive at the beginning of the program.
- Note that there is no "main" function. All the code we provide is executed directly line by line.
- Notice the difference between declaring a function and executing it.
- Reason about all the steps that are executed and what you observe that is printed on the screen.

2.- Scope of the variables

We have to know the differences between using "let" and using "var" to declare variables. We must know when we can use each of them and what implications it has. We should also note what happens if we don't use "let" or "var" without using JavaScript strict mode.

2.1 Read and run the program “ej2_AmbitoVariables.js” and **answer** the following questions

Question 1. Notice that the program modifies the value of a global variable within the function “f1”. Can we modify the value of the other global variables? What is the difference between declaring global variables with "let" and with "var"? Yes, we can modify other global variables.

When declaring variables with let, they are only available inside the block where they are defined, but with var, the variable is available in the entire function where it is defined.

Question 2. Notice that on line 24 the value of the variable “var_i” is printed. Can we print the value of the variable “let_i” ? Try to do it and explain what you observe. Explain the differences between using “let” and using “var”. We can't, because it only exists inside the for loop, outside it it's out of scope (unlike with var, which is available in the entire function).

Question 3. This program does not use the "use strict" directive at the beginning of the program. Add it and try running the program again. What do you observe? Explain what the “use strict” directive is for.

With 'use strict' there is an error, because it prevents us from creating global variables by using `var_name = value`.

Question 4. Modify the program so that it works correctly using the “use strict” directive. This fixes the mistake made by the programmer for wrongly referencing the variable “local1_let”. Discuss what advantages the use of the "use strict" directive has for the programmer.

It makes it harder to create variables that don't exist when you are just trying to reassign the value of one that exists.

3. Variable types

JavaScript is an untyped language. This fact can be quite confusing in many situations and it is necessary to program with awareness of variable types and their conversions.

3.1 Read and execute the program “ej3_TiposDeVariables.js”

3.2 Reason why subtraction works differently than addition.

3.3 Look up information on the Internet about the “TypeScript” language. Explain the reasons that are leading to its growing implementation today.

It adds static typing and type annotations to JavaScript. It's popular because it simplifies JavaScript code and makes it easier to read.

4. Function arguments

JavaScript is a fairly flexible language in terms of the number of function arguments and their types. This feature is powerful and at the same time can lead to programming errors.

4.1 Read and execute the program “ej4_Argumentos.js” and **answer** the following questions:

Question 1. Explain what the “arguments” pseudo-array does. Can the different arguments be accessed by this pseudo-array? It contains all the arguments used to call the function. The arguments can be accessed by using arguments[i]

Question 2. When printing result3, we see “NaN”. What does NaN mean and why do we see this result?

Because it is performing the addition 1 + undefined + undefined, which is not a number.

Question 3. What do the ellipses (…) mean in the call to print “resultv1” ?

It is the spread operator, it passes the values of the vector as individual values to the function, so a = 1, b = 2, c = 3.

Question 4. Notice what is printed as the result “resultv2”. Why do we get this result?

resultv2: 1,2,3,4undefinedundefined

Because it is adding the vector as a string (the first argument) to the second and third arguments, which are `undefined`.

5. Classes in JavaScript

JavaScript support for object-oriented programming is somewhat primitive, although it has improved a lot since the 2015 versions. Today you can use the newer and more comfortable class syntax or use the "old" way of creating classes and instances. Both forms coexist.

5.1. Read and execute the program ej5-1_Clases.js

Notice how "this" is used, similar to how it is used in other languages.
Notice how the attributes of the classes are declared. Delete the "nombre" attribute and run the program again. Note that the attribute declaration is completely optional.
Notice how constructors are declared and how "super" is used.

This program makes use of the most modern JavaScript syntax. It is still very possible that JavaScript object support will be improved in future versions.

Although the usage of classes should be recommended in JavaScript, it will not be required in this subject. Therefore, no other example related with classes is presented in this lab session.

6.- Closures

Surely one of the most important concepts in JavaScript (and other languages) are closures. This section does not intend to give a complete course of closures, but to give a basic draft to their use.

To simplify the explanation, we can summarize by saying that every function that is executed in JavaScript has an associated closure. The closure of a function consists of the variables (and functions) that are referenced from that function. These symbols that are referenced from the function remain in memory as long as the function is being referenced. The practicality is found when the closure contains **symbols that are not global.**

The simplest case is in a function that does not reference anything. In this case we can say that its closure is empty. More commonly, this function is simply said to be closureless or make no use of closures. The same is true if the closure only references global symbols, since these symbols are always accessible and no “language effort” is required to allow access to these symbols.

6.1.- Read and execute the program “ej6-1_Clausuras.js”

Notice that function f1 has no closure.
Notice that the closure of f2 is the variable x. Since it is a global symbol, it is not really a closure in the strict sense, since the function can access this global variable without any problem.
Look at function f3. In this case we have the **usual pattern** of closures. A function that returns another function. The f3 function is called the **generator function,** and the function we return is often called the **closure function**, or just closure for short. The closure function returned by f3 has as its closure the argument “arg” and the variable “i”. Both symbols are local to “f3” and therefore are visible in the closure function, since they are in its scope. JavaScript's closure support will keep these variables in memory as long as the closure function is referenced. Notice how the closure function is referenced in variable “f”. Observe how the value of variable "i" persists between successive invocations.

6.2.- Read and execute the program “ej6-2_Clausuras.js”

This example illustrates a common use case, particularly in JavaScript software for browsers. Notice how there are 3 segments of code that do practically the same thing. The first segment uses a global variable. The second code segment uses a generator function and a closure and in this way we avoid the use of global variables. However we create a global symbol to hold the function. The third code segment is more complex, but more powerful, because through closures and anonymous functions we can avoid the use of global symbols.

Answer the following question.

Question 1. Why might it be interesting to use a pattern like the one described in this example when we make software that will be used as a library, from a browser or from nodejs?

6.3.- Read and execute the program “ej6-3_Clausuras.js”

This example illustrates a closure that contains variables, arguments, and functions. After running and studying the code, answer the following question:

Question 1. What is the closure returned by the generator function? Detail the variables, arguments, and functions that are part of the closure.

6.4.- Read and execute the program “ej6-4_Clausuras.js”

This example shows a generator function that returns a function between 2 possible ones. After running and studying the code, answer the following questions:

Question 1. What is the closure of function g0?

Question 2. What is the closure of function g1?

Question 3. How many copies in memory are there of the “traza” variable? Remember that closures keep the variables they refer to in memory.

7.- Asynchronous programming through callbacks

Asynchronous programming provides a different approach to concurrency than classic multi-threaded programming. It is an approach that can be seen as complementary, rather than as an alternative.

In the theory classes, the **event loop will be explained** as a central mechanism to the asynchronous programming that we find in JavaScript. Therefore, this lab section does not replace the content of theory, but rather, it can serve as a first approach to certain practical aspects.

For simplicity, we can say that a program in "nodejs" can **register event handlers or listeners**. When these events happen, the listeners will be executed. It is somewhat similar to the interrupt handlers found in operating systems, or the event handlers found in certain graphics libraries. (In fact, several graphics libraries studied at this University are based on an event loop: AWT, Swing, JavaFX, etc).

The easiest event to generate and handle is probably time.

With the "setTimeout" call, we simultaneously do 2 things: we set up a listener for the "timeout" event, and we ask that the "timeout" event occur within a certain number of milliseconds. Later, when the time we have indicated expires, this listener will be executed asynchronously.

7.1.- Read and run the ej7-1_Timeouts.js program

Observe what the program does and answer the following questions:

Question 1 . Why do we see the message "cinco" before "cuatro"

Question 2 . Why do we see “dos” before “tres” or “tres” before “dos”? To explain your answer to this question, run the program several times, modifying the number of iterations of the loop so that it does 100, 1000, 10000, 100000 iterations.

7.2.- Read and execute the program ej7-2_Timeouts.js

The program registers 10 event listeners and reaches the end of the program.

Note that the program does not terminate when printing the final message to the console.

Observe the variable value that is printed.

Answer the following questions:

Question 1. Why does it always print 10?

Question 2. Why does it print a message every second?

Question 3. Why does the program terminate when printing 10 messages to the console? Why didn't it terminate when executing the last line of code in the program?

7.3.- Read and execute the program ej7-3_Timeouts.js

This program combines event management with closures. With this small example we can see that it improves the previous example thanks to the closures.

The code is short, but you need to study it carefully. Give it several minutes. If you understand this program you will take an important step to understand how to program using JavaScript and nodejs.

Answer the following questions:

Question 1. Identify the generating function and the closing function. When and how many times is the generator function called? How many closure functions are created and who calls them?

Question 2. What is the content of the closures?

Question 3. What advantage does the use of closures offer in this example?

7.4.- Read and execute the program ej7-4_Timeouts.js

This example creates closures without using generator functions. We can rather speak of a closure generator block. This example serves as a reminder that generator functions are a common pattern for creating closures, but as this example illustrates, they are not strictly necessary.

Notice that the closure generator block declares a variable via “let”.

Answer the following question.

Question 1. Modify the program so that it declares the variable inside the “do” block using “var”. Explain why now the operation is different.

7.5.- Read and execute the program ej7-5_Timeouts.js

This example contains a slightly more complete program than the rest. We can observe closures, a closure generation loop and it provides a function that will be notified when all closures are done. The way to notify this end is through a callback.

Study and run the code to understand what it does and answer the following question.

Question 1. When will the callback function provided to the forkJoinAsync function be executed, and where will it be executed from?