Seminar 1 – Node.js

Student Guide

1. Introduction

This guide extends several sections of the presentation about JavaScript/Node.js that provides the basis for Seminar 2. JavaScript is a programming language with abundant documentation available on the web. It is assumed that the student has learned Java in other subjects. As a result, it will not be difficult to read JavaScript programmes since many syntactical structures of the JavaScript language are similar to those of Java.

The presentation provides the following references about JavaScript:

1. Tim Caswell: "Learning JavaScript with Object Graphs". Available at: http://howtonode.org/object-graphs, 2011.
2. Tim Caswell: "Learning JavaScript with Object Graphs (Part II)". Available at: http://howtonode.org/object-graphs-2, 2011.
3. Patrick Hunlock: "Essential JavaScript – A JavaScript Tutorial". Available at: http://www.hunlock.com/blogs/Essential_Javascript_--_A_Javascript_Tutorial, 2007.
4. David Flanagan: "JavaScript: The Definitive Guide", 5th ed., O'Reilly Media, 1032 pgs., August 2006. ISBN: 978-0-596-10199-2 (printed edition), 978-0-596-15819-4 (eBook).

We recommend starting with reference 3. It may be read in 30 minutes. Later, the student could read references 1 and 2, since they do not require a long time. Reference 4 is the one recommended in all intensive courses about JavaScript, but it requires more time than the one available in a 6 ECTS subject. That book includes a reference about JavaScript in its Part III. As an alternative, the student could read any of these references in order to delve into JavaScript knowledge:

- Marijn Haverbeke: "Eloquent JavaScript. A Modern Introduction to Programming", July 2007, first edition. Available at: http://eloquentjavascript.net/1st_edition/contents.html
  - There is a printed edition, published in 2011 by *No Starch Press*, *Inc*. (ISBN 978-1-59327-282-1).
  - It consists of 14 chapters and 2 appendices.
  - You could read only chapters 1, 2, 3, 4, 5, 6, 8 and 9. This could be done in 6 or 7 hours.
  - It provides a lot of examples and exercises, with their solutions. Chapter 1 explains how to read and execute the solutions using your web browser.
  - Recently (December 2014) a second edition has been published; it is quite longer. It is available in HTML, ePUB and PDF formats at this address: http://eloquentjavascript.net/index.html. We recommend that you start with the first edition, since in a short interval you could grasp the essential parts of this programming language. Later on, this second edition may be read.
- Mozilla Developer Network: "JavaScript Guide". Available at:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide (covers up to JavaScript 1.8)
- Mozilla Developer Network: "JavaScript Reference". Available at:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

The following sections describe some important aspects of JavaScript that have not been analysed in the presentation.

It is worth noting that JavaScript is an interpreted programming language with a very relaxed management of types. In turn, Java is a compiled programming language with a strong type management. Because of this, when a Java programme is being developed most of the errors are detected by the compiler. In JavaScript there is no compiler. As a result, we cannot rely on it for implementing correct programmes. When we have a non-syntactical bug in a JavaScript programme, the JavaScript interpreter will not generate any warning and the user might not perceive any problem. Due to this, it is recommendable that we adequately structure our programmes in small modules, thoroughly checking each of their functions before delivering the solutions for the lab practices or the seminar activities. In the developing stage we could add printing sentences showing the contents of the main programme objects in order to adequately follow a trace of the programme execution. Those sentences will be removed before delivering the programmes.

For instance, in JavaScript there is no constraint for defining attributes in an object. Those attributes can be added at any time. Because of this, if we incorrectly type the name of an object attribute in any of the programme sentences, there will be no error message when it is executed. Thus, if we write:

```
socket.identify=”myName”
```

Instead of this:

```
socket.identity=”myName”
```

No error or warning will be shown in the programme execution (but the programme may not behave as expected, although unfortunately this might be difficult to observe, so we should take care of printing the contents of each object in order to follow a trace of our programmes while they are being developed). So, if we add a sentence as this:

```
console.log(socket);
```

We will see that...

```
{ identity: undefined, identify: ‘myName’, ... }
```

...; i.e., there is an unexpected attribute in the printed contents. Let us recommend again to add sentences in the functions that we are developing in order to carefully check that all the objects being managed in our programme have their expected contents.

2. Closures

Slide 12 in the presentation of this seminar shows a programme fragment that presents how variable scopes are managed in JavaScript. In some cases it is useful to define a function inside another, returning the internal function. This internal function may still use the variables or parameters declared in its encompassing function. This is still valid once the encompassing

function has completed its execution. The fact that this external scope is maintained and remains valid is known as a “function closure”.

Let us describe an example of use of closures. The “log()” function in the Math standard JavaScript module returns the natural logarithm of its received argument. We may use closures for implementing a function that “builds” and returns another that computes logarithms in any specified base. To this end, the following logarithm property is considered:

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}.$$

And the needed fragment of JavaScript code is:

```
function logBase(b) {
   return function(x) {
          return Math.log(x)/Math.log(b);
   }
}
```

In this example, the builder function is “logBase()” and it uses a parameter “b” for specifying the base to be used. Its single sentence returns an internal function. Such function is anonymous and receives a single parameter: “x”. The code of such returned function applies the logarithm property that has been presented above, remembering the argument received in the builder function.

Using these functions we may generate new functions that compute logarithms in any given base. In the following code fragment, two functions are generated for using base 2 and base 8. The resulting code is quite intuitive:

```
log2 = logBase(2);
log8 = logBase(8);

console.log("Logarithm with base 2 of 1024 is: " + log2(1024));  // 10
console.log("Logarithm with base 2 of 1048576 is: " + log2(1048576)); // 20
console.log("Logarithm with base 8 of 4096 is: " + log8(4096));  // 4
```

In some cases we may use some of the local objects in the function that provides the closure scope, instead of using only one of its parameters. In that case, it should be noted that those objects are accessed by reference and this might have unexpected results.

Let us see an example. In the following sample we want to develop a function that returns an array holding three functions. Each function returns the name and population of each one of the three most populated countries in the world. Unfortunately, this version does not work:

```
1: function populations() {
2:    var pops = [1365590000, 1246670000, 318389000];
3:    var names = ["China", "India", "USA"];
4:    var placeholder = ["1st", "2nd", "3rd", "4th"];
```

```
 5:    var array = [];
 6:    for (i=0; i<pops.length; i++)
 7:          array[i] = function() {
 8:             console.log("The " + placeholder[i] +
 9:                          " most populated country is " +
10:                          names[i] + " and its population is " +
11:                          pops[i]);
12:          };
13:    return array;
14: }
15:
16: var ps = populations();
17:
18: first = ps[0];
19: second = ps[1];
20: third = ps[2];
21:
22: first();
23: second();
24: third();
```

The sentence being used in line 16 calls the “populations()” function. We expect that the “ps” array gets the three requested functions. In that case, the sentences shown in lines 22 to 24 would print the requested information. However, their output is the following:

```
The 4th most populated country is undefined and its population is undefined
The 4th most populated country is undefined and its population is undefined
The 4th most populated country is undefined and its population is undefined
```

This may be explained because when the functions “first()”, “second()” and “third()” are called, the “i” variable in the “populations()” function holds a value of 3. The access to such variable is made by reference. So, we are accessing to “names[3]” (“4th”) and “pops[3]” (undefined). It does not matter that when such functions were created the value of “i” was 0, 1 and 2, respectively. Currently, its value is 3 and this is the used value. Due to this, the output being printed is not what we tried to print.

We should use function closures in order to access to “i”. Our aim is that the function being built in lines 7 to 12 remembers the value of “i” in that iteration of the loop. We are not interested in the value of “i” when we call the built function but in the value of “i” when the function is built.

A correct solution is shown here:

```
 1: function populations() {
 2:    var pops = [1365590000, 1246670000, 318389000];
 3:    var names = ["China", "India", "USA"];
 4:    var placeholder = ["1st", "2nd", "3rd", "4th"];
 5:    var array = [];
 6:    for (i=0; i<pops.length; i++)
```

```
 7:            array[i] = function(x) {
 8:              return function() {
 9:                      console.log("The " + placeholder[x] +
10:                                  " most populated country is " +
11:                                  names[x] + " and its population is " +
12:                                  pops[x]);
13:              };
14:            }(i);
15:    return array;
16:  }
17:
18:  var ps = populations();
19:
20:  first = ps[0];
21:  second = ps[1];
22:  third = ps[2];
23:
24:  first();
25:  second();
26:  third();
```

The changes are highlighted in boldface. We need another encompassing function that receives as its single argument the current value of "i". To this end, this new function receives an "x" parameter and the function being returned uses "x" instead of "i". So, in each iteration of the loop the encompassing function is invoked using "i" as its argument (this is the aim of the current line 14).

Using this hint, the new programme output is:

```
The 1st most populated country is China and its population is 1365590000
The 2nd most populated country is India and its population is 1246670000
The 3rd most populated country is USA and its population is 318389000
```

This is the expected output.

Additional information about closures may be found in Chapter 3 of [1], and in [2].

3. Event Queue

JavaScript maintains an event queue (also known as "turn queue"). This is needed because JavaScript does not support multiple threads of execution. Because of this, when a new activity is generated in a programme, that new activity is implemented as a new "event" and it is inserted at the end of the turn queue. Those turns are served in FIFO order. This means that the service of a new turn demands that the previous turn was completed.

Let us analyse the example shown in the slide 23 of the seminar presentation:

```
1:  function fibo(n) {
2:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1);
```

```
 3: }
 4: console.log(“Starting...");
 5: // Wait for 10 ms to write the message.
 6: // A new event is generated.
 7: setTimeout( function() {
 8:   console.log( "M1: First message..." );
 9: }, 10 );
10: // More than 5 seconds needed.
11: var j = fibo(40);
12: function anotherMessage(m,u) {
13:    console.log( m + ": Result is: " + u );
14: }
15: // M2 is written before M1 since the main
16: // thread has not been suspended...
17: anotherMessage("M2",j);
18: // M3 is written after M1.
19: // It is scheduled in 1 ms.
20: setTimeout( function() {
21:  anotherMessage("M3",j);
22: }, 1 );
```

Function setTimeout() is used in JavaScript for scheduling the execution of the function received in its first parameter after the number of milliseconds specified in its second parameter.

Let us follow a trace of this programme:

- Execution is started in lines 1 to 3, where the recursive function “fibo()” is defined. It receives an integer argument and computes the Fibonacci number associated to that value.
- Line 4 prints the message “Starting...” on screen.
- Lines 7 to 9 use setTimeout() to schedule the printing of another message (“M1: First message...”) after 10 ms. At the moment, this has no effect.
- Line 11 invokes the “fibo()” function with argument 40. Its execution lasts several seconds. In that term, the interval of 10 ms mentioned in the previous paragraph is concluded. This means that the function that prints the M1 message is already placed in the event queue, waiting for its turn. Such turn will not be started until the main thread is terminated.
- This main thread eventually arrives to lines 12 to 14, where the function anotherMessage() is defined. Such function will be used later for printing the obtained Fibonacci number.
- Execution is now at line 17. It prints the result. The message being printed is “M2: Result is: 165580141”.
- Finally, lines 20 to 22 are executed, where the programme schedules the execution of anotherMessage() (M3 message) after 1 ms. This terminates the execution of the main thread.

- At this time, there is only a single turn in the event queue. Its execution is started. Such turn prints message “M1: First message...” on screen. While this message is being printed, the 1 ms interval started in the previous point terminates. As a result, a new context is inserted in the turn queue. Its aim is to print the M3 message.
- Once M1 has been printed, such first turn is concluded. The event queue is revised and another turn is found. Its execution is started, printing this message “M3: Result is: 165580141”.
- This terminates the programme execution. Its full output is:

```
Starting...
M2: Result is: 165580141
M1: First message...
M3: Result is: 165580141
```

As it can been seen, the two first lines were printed by the main thread. In turn, the M1 and M3 messages were printed using two events. Although the first of those events was generated very soon (while the function fibo() was being executed; before printing message M2) its result could not be shown at that time on screen. The main thread of the programme need to terminate before those additional events started their execution.

Slide 30 of the seminar presentation contains another programme, based on the EventEmitter class, that also uses the event queue. Let us revise that example:

```
 1: /*************************************************************/
 2: /* Events1.js                                                */
 3: /*************************************************************/
 4: var ev = require('events');
 5: var emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: var e1 = "print";
 8: var e2 = "read";
 9: // Auxiliary variables.
10: var num1 = 0;
11: var num2 = 0;
12:
13: // Listener functions are registered in
14: // the event emitter.
15: emitter.on(e1, function() {
16:   console.log( "Event " + e1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(e2, function() {
19:   console.log( "Event " + e2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(e1, function() {console.log(
24:   "Something has been printed!!");});
25:
26: // Generate the events periodically...
```

```
27: // First event generated every 2 seconds.
28: setInterval( function() {
29:    emitter.emit(e1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(e2);}, 3000 );
```

The “emitter” object is created in line 5 and four other variables are defined in lines 7 to 11. Later three functions are associated to two events to be generated. Later, in lines 28 and 29 the event “e1” (print) is scheduled every 2 seconds and in lines 31 and 32 the event “e2” (read) is scheduled every 3 seconds. After this, the main thread is terminated.

Each time the “e1” event is generated, two functions are inserted in the event queue, as it is programmed in lines 15 to 17 (this function prints the number of times such event has happened) and in lines 23 and 24 (this function prints the message “Something has been printed!!”). In turn, each time the “e2” event is generated, the function defined in lines 18 to 20 is inserted in the event queue. Such function prints the number of times that the “read” event has happened.
Since the raising intervals for these events are long, the event queue usually remains empty. When an event is generated, its handling functions (i.e., listeners) are inserted in the event queue and they are started almost immediately, emptying again the event queue.

An initial fragment of the output is...

```
Event print has happened 1 times.
Something has been printed!!
Event read has happened 1 times.
Event print has happened 2 times.
Something has been printed!!
Event read has happened 2 times.
Event print has happened 3 times.
Something has been printed!!
Event print has happened 4 times.
Something has been printed!!
Event read has happened 3 times.
Event print has happened 5 times.
Something has been printed!!
Event read has happened 4 times.
Event print has happened 6 times.
Something has been printed!!
Event print has happened 7 times.
Something has been printed!!
Event read has happened 5 times.
Event print has happened 8 times.
Something has been printed!!
Event read has happened 6 times.
Event print has happened 9 times.
Something has been printed!!
Event print has happened 10 times.
Something has been printed!!
```

```
Event read has happened 7 times.
Event print has happened 11 times.
Something has been printed!!
Event read has happened 8 times.
Event print has happened 12 times.
Something has been printed!!
Event print has happened 13 times.
Something has been printed!!
Event read has happened 9 times.
```

As an easy exercise, justify which is the output of the following programme, where the previous programme is slightly extended.

```
 1: /*************************************************************/
 2: /* Events2.js                                                */
 3: /*************************************************************/
 4: var ev = require('events');
 5: var emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: var e1 = "print";
 8: var e2 = "read";
 9: // Auxiliary variables.
10: var num1 = 0;
11: var num2 = 0;
12:
13: // Listener functions are registered in
14: // the event emitter.
15: emitter.on(e1, function() {
16:   console.log( "Event " + e1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(e2, function() {
19:   console.log( "Event " + e2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(e1, function() {console.log(
24:   "Something has been printed!!");});
25:
26: // Generate the events periodically...
27: // First event generated every 2 seconds.
28: setInterval( function() {
29:    emitter.emit(e1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(e2);}, 3000 );
33: // Loop.
34: while (true)
35:    console.log(".");
```

4. Module Management

Slide 28 in the presentation describes how to export "public" functions in a module, using "`exports`", and how they are imported using "`require()`". That slide also presents some examples of their use.

But "`exports`" is only an alias for "`module.exports`". Moreover, both "`module.exports`" and "`module`" are regular JavaScript objects. This means that they hold a dynamic set of properties (i.e., attributes) that we could enlarge or decrease at our own. Each module has a private "`module`" object. It is not a global object being shared by all source files in a given application. Using this object we may specify which operations and objects are exported in a module.

In order to add a property or method to an object, we only need to define them, assigning an initial value, if needed. This is already shown in slide 28. Both "`area()`" and "`circumference()`" are used as new properties of the "`module.exports`" object. Since they are functions, they are exported as public operations of that module.

For removing a property we only need the "**delete**" keyword, followed by the name of the property to be removed.

This mechanism allows building modules that import other modules, modifying some of their operations but maintaining the others. For instance, this sample of code...

```
var c = require('./Circle');

delete c.circumference;

c.circunferencia= function( r ) {
  return Math.PI * r * 2;
}

module.exports = c;
```

...renames the "`circumference()`" operation of module "Circle.js". Its new name is "`circunferencia()`", and this does not modify any of other operations in that module.

A detailed analysis of the actions taken in this code fragment is:

1. When the module "`Circle.js`" is imported using "`require()`", we assign its exported object to variable "`c`" in our programme.
2. Later, its "`circumference()`" operation is removed. The other operations or properties are not affected. In this example, there is only another operation: "`area()`".
3. Later we add a new operation: "`circunferencia()`".
4. As a last sentence, we export the entire "`c`" object. This provides the "`area()`" and "`circunferencia()`" operations to the programmes that import it.

If we store this code fragment into a "`Círculo.js`" file, we will have a new module with that name exporting those two operations.

References

[1] M. Haverbeke, "Eloquent JavaScript. A Modern Introduction to Programming", ISBN 978-1-59327-282-1: No Starch Press, Inc., 2011.

[2] I. Kantor, "JavaScript: From the Ground to Closures", Available at: http://javascript.info/tutorial/closures, 2011.