Seminari 1 – Node.js

Guia d'estudi

1. Introducció

Aquesta guia estén algunes seccions de la presentació sobre Javascript/Node.js inclosa com a material del Seminari 1. JavaScript és un llenguatge de programació amb abundant documentació disponible en la web. S'assumeix que l'alumne ha utilitzat Java en altres assignatures d'aquesta titulació, per la qual cosa no li resultarà difícil llegir codi escrit en JavaScript ja que tant els bucles com les sentències condicionals usen una sintaxi similar a la de Java.

En la bibliografia recomanada en la presentació se citen els següents treballs:

1. Tim Caswell: "Learning JavaScript with Object Graphs". Disponible en: http://howtonode.org/object-graphs, 2011.
2. Tim Caswell: "Learning JavaScript with Object Graphs (Part II)". Disponible en: http://howtonode.org/object-graphs-2, 2011.
3. Patrick Hunlock: "Essential JavaScript – A JavaScript Tutorial". Disponible en: http://www.hunlock.com/blogs/essential_javascript_--_A_Javascript_Tutorial, 2007.
4. David Flanagan: "JavaScript: The Definitive Guide", 5ª ed., O'Reilly Media, 1032 pàgs., agost 2006. ISBN: 978-0-596-10199-2 (ed. impresa), 978-0-596-15819-4 (ebook).

Seria recomanable començar amb la referència 3, que pot ser llegida en poc més de 30 minuts. Després hauria de consultar-se tant la referència 1 com la 2, que tampoc requereixen excessiu temps. La referència 4 és la recomanada en la majoria dels cursos intensius sobre JavaScript però requereix molt més temps del que tindrem disponible. Incorpora també el manual de referència del llenguatge en la Part III del llibre. En el seu lloc, si volem aprofundir en el nostre aprenentatge, podem utilitzar:

- Marijn Haverbeke: "Eloquent Javascript. A Modern Introduction to Programming", juliol 2007, primera edició. Disponible en: http://eloquentjavascript.net/1st_edition/contents.html
  - Inclou 14 capítols i dos apèndixs.
  - Només cal consultar els capítols 1, 2, 3, 4, 5, 6, 8 i 9. Pot fer-se en poc més de sis hores en una primera lectura.
  - Ofereix exercicis (convé parar esment al capítol 1, on explica com resoldre'ls i executar-los des del navegador) amb les seues solucions. Recomanem que s'intente completar tot aquest conjunt d'exercicis.
  - Existeix una versió impresa d'aquesta primera edició, publicada en 2011 per *No Starch Press*, *Inc*. (ISBN 978-1-59327-282-1).
  - Recentment (desembre de 2014) s'ha publicat una segona edició, molt més extensa. Està disponible en format HTML, ePUB i PDF en aquesta adreça: http://eloquentjavascript.net/index.html. És preferible començar amb la primera edició ja que invertint un temps relativament breu s'aconsegueix assimilar el més important del llenguatge. Posteriorment pot consultar-se aquesta segona edició si es volguera, però no ho creiem necessari.
- Mozilla Developer Network: "JavaScript Guide". Disponible en:
  - https://developer.mozilla.org/en-us/docs/web/javascript/guide (en anglès, cobreix fins a Javascript 1.8)

        - És també una bona guia per a aprendre el llenguatge, amb abundants exemples...
        - ...pero no planteja exercicis, com sí feia el llibre anterior.
    - https://developer.mozilla.org/es/docs/gu%C3%ADa_Javascript_1.5 (en castellà; documenta JavaScript 1.5 encara que també cobreix algunes de les extensions posteriors).
- Mozilla Developer Network: "JavaScript Reference". Disponible en:
    - https://developer.mozilla.org/en-us/docs/web/javascript/reference (en anglès).
        - Aquest manual de referència no cal llegir-lo a priori. N'hi ha prou amb assabentar-se del que s'explique en la guia. La referència descriu l'API i serà necessària quan ja hàgem d'utilitzar cadascuna de les operacions.
    - https://developer.mozilla.org/es/docs/javascript/referencia (en castellà).

En les pròximes seccions es descriuran alguns aspectes importants de JavaScript i Node.js que no han sigut suficientment coberts en la presentació.

Convé recordar que JavaScript és un llenguatge interpretat i amb una comprovació de tipus molt relaxada. Per la seua banda, Java és un llenguatge compilat i amb una gestió forta dels tipus. Per això, en els programes Java bona part dels errors que el programador tinga seran detectats pel compilador i obligaran al programador a corregir-los abans de poder executar el programa que estiga desenvolupant. En JavaScript no succeirà el mateix. Bona part dels errors no generaran cap problema aparent. L'intèrpret executarà les sentències sense donar cap advertiment i l'usuari no observarà cap problema. És recomanable que s'estructure adequadament el codi que haja d'escriure's per a resoldre les activitats dels seminaris i les pràctiques del laboratori i que es comprove que cada mòdul generat (objectes, mètodes, funcions...) treballe adequadament. Per a fer això es recomana realitzar tests unitaris. Durant l'etapa de desenvolupament, convé també acostumar-se a utilitzar alguns missatges que permeten seguir la traça de l'execució, mostrant els valors de les variables i atributs més importants per a comprovar que tot funcione com s'espera. Aquests missatges s'eliminaran posteriorment, abans d'entregar els exercicis.

Per exemple, en JavaScript no hi ha cap restricció a l'hora de definir atributs en un objecte. Podrem afegir-ne tants com vulguem en els objectes que utilitzem. Per això, si en algun dels objectes ens equivoquem a l'hora de donar el nom d'un dels seus atributs, no apareixerà cap missatge d'error. Així, si escriguérem:

```
socket.identify="unNom"
```

Quan hauríem d'haver escrit:

```
socket.identity="unNom"
```

No apareixerà cap error durant l'execució (però probablement el programa resultant no arribe a comportar-se com s'espera, però això no sempre serà fàcil d'advertir, llevat que incloguem sentències que ens permeten seguir la traça). No obstant això, si posteriorment utilitzem aquesta sentència:

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

Observarem que la seua eixida és:

```
{ identity: undefined, identify: 'unNom' }
```

En lloc de només mostrar l'atribut esperat observem que ara n'hi ha dos. Això fa recomanable que, de tant en tant, en la traça es bolque l'estat dels objectes que estiguem utilitzant i que comprovem que no han "aparegut" atributs inesperats en ells.

2. Clausures

La fulla 12 de la presentació mostra un exemple de codi en el qual s'il·lustra com es gestiona l'àmbit d'una variable. En alguns casos resulta útil declarar una funció dins d'una altra. En aquestes situacions pot ser que resulte interessant retornar la funció interna. Aquesta funció interna seguirà tenint accés a les variables utilitzades en la funció que l'engloba, així com als seus arguments. Això segueix sent vàlid encara que la invocació que la va crear haja acabat. El fet que es mantinga el context extern es coneix com a "clausura".

Vegem un exemple en el qual podrem utilitzar clausures. La funció "log()" del mòdul Math de JavaScript retorna el logaritme neperià del valor rebut com a argument. Utilitzarem clausures per a implantar una funció que en "construïsca" i retorne una altra capaç de calcular el logaritme en la base que especifiquem. Per a fer això aprofitarem aquesta propietat dels logaritmes:

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

El codi a utilitzar seria:

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

Com pot observar-se, la funció "constructora" es diu "logBase()" i utilitza el paràmetre "b" per a especificar la base a utilitzar. La seua única sentència s'encarrega de retornar una funció. Aquesta funció és una funció anònima que rep com a únic paràmetre un valor "x". El codi de la funció retornada s'encarrega d'aplicar la propietat que hem enunciat a dalt, recordant l'argument rebut en la funció constructora.

Prenent aquest codi com a punt de partida, podrem generar les funcions necessàries per a calcular logaritmes en diferents bases. En el següent fragment utilitzem tant la base 2 com la 8. Observe's que resulta senzill seguir el codi resultant:

```
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
```

Pot ser que necessitem usar alguna variable de la funció que proporcione el context de la clausura, en lloc d'utilitzar únicament algun dels seus paràmetres. En aquest cas ha de recordar-se que l'accés a les variables es fa per referència i això pot ser problemàtic.

Vegem-ne un exemple. En el següent fragment de codi es desitja desenvolupar una funció que retorne un vector que conté tres funcions. Cadascuna d'elles retornarà el nom i població de cadascun dels tres països més poblats del nostre planeta. Aquesta solució no funciona:

```
 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();
```

La crida efectuada en la línia 16 executa el codi de la funció “populations()”. Seria d'esperar que el vector “ps” continguera les tres funcions sol·licitades i que les crides efectuades en les línies 22 a 24 mostraren el nom i població dels tres països més poblats. No obstant això, l'eixida obtinguda és:

```
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
```

Això es justifica perquè quan cridem a les funcions “first()”, “second()” i “third()” el valor de la variable “i” dins de la funció “populations()” és 3. Poc importa que quan es van crear les tres funcions valguera 0, 1 i 2, respectivament. Seguim utilitzant la variable “i” quan s'invoquen les funcions creades (és com si s'estiguera passant per referència) i el seu valor actual és 3. Per això, l'eixida obtinguda no és correcta.

Com podem resoldre-ho? Utilitzant clausures a l'hora d'accedir al valor de “i”. El nostre objectiu és que la funció que està sent declarada entre les línies 7 i 12 recorde el valor que tenia “i” en aquella iteració del bucle. No ens interessa el valor que tindrà la “i” quan invoquem a les funcions que retornarem en el vector sinó el que té mentres itera.

La solució es mostra a continuació:

```
 1: function populations() {
 2:    var pops = [1365590000, 1246670000, 318389000];
 3:    var names = ["Xina", "Índia", "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();
```

Els canvis s'han ressaltat en negreta. Necessitem una altra funció que la tanque i que reba com a únic argument el valor actual de la variable “i”. Per a fer això, la nova funció rep un paràmetre “x” i la funció que retornarem accedirà a “x” en lloc d'accedir a “i”. En cada iteració del bucle s'invoca la funció que engloba a la que anàvem a retornar, passant com a argument el valor actual de la variable “i” (vegeu la línia 14). L'única cosa que fa la funció contenidora és retornar la funció que en la versió anterior estava llistada entre les línies 7 i 12.

Amb aquest “truc” aconseguim que l'eixida proporcionada pel programa siga:

```
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
```

Aquesta era l'eixida a obtenir. Amb això queda demostrat que ha de portar-se certa cura a l'hora d'utilitzar bucles per a generar funcions utilitzant clausures.

Pot obtenir-se més informació sobre les clausures en el capítol 3 de [1], així com en [2].

3. Cua d'esdeveniments

JavaScript té una cua d'esdeveniments (també coneguda com a "cua de torns"). Això es deu al fet que JavaScript és un llenguatge que no suporta múltiples fils d'execució. Per això, quan en alguna part d'un programa es genere alguna altra activitat, aquesta activitat es modela com un nou "esdeveniment" i és afegida a la cua d'esdeveniments o torns. Els torns se serveixen en ordre FIFO. Per a iniciar el servei d'un nou torn haurà d'haver acabat l'execució del torn anterior.

Analitzem l'exemple mostrat en la fulla 23 de la presentació:

```
 1: function fibo(n) {
 2:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1);
 3: }
 4: console.log("Iniciant execució...");
 5: // Esperar 10 ms per a escriure un missatge.
 6: // Implica generar nou esdeveniment.
 7: setTimeout( function() {
 8:   console.log( "M1: Vull escriure açò..." );
 9: }, 10 );
10: // Més de 5 segs. per a executar fibo(40)
11: var j = fibo(40);
12: function altreMissatge(m,o) {
13:   console.log( m + ": El resultat és: " + o );
14: }
15: // M2 s'escriu abans que M1 perquè el
16: // fil "principal" no se suspèn...
17: altreMissatge("M2",j);
18: // M3 ja s'escriu després de M1.
19: // És un altre esdeveniment (a executar després d'1 ms).
20: setTimeout( function() {
21:  altreMissatge("M3",j);
22: }, 1 );
```

La funció setTimeout() s'utilitza en JavaScript per a programar l'execució de la funció rebuda en el seu primer paràmetre al cap del nombre de mil·lisegons especificats en el seu segon paràmetre.

Seguim una traça d'aquest programa:

- L'execució comença a les línies 1 a 3, on es declara una funció recursiva “fibo()” que rebrà un argument enter i calcularà el nombre de Fibonacci associat al seu valor.
- Arribem ara a la línia 4 que escriu el missatge “Iniciant execució...” en la pantalla.
- En les línies 7 a 9 s'utilitza setTimeout() per a programar l'escriptura d'un altre missatge (“M1: Vull escriure açò...”) dins de 10 ms. De moment això no té cap efecte.
- Arribem a la línia 11 on s'invoca la funció “fibo()” amb el valor 40. La seua execució necessitarà alguns segons. Durant aquest termini hauran transcorregut els 10 ms esmentats en el punt anterior. Això implica que la funció que ha d'escriure el missatge M1 per pantalla està ja situada en la cua d'esdeveniments, esperant el seu torn. Aquest torn no començarà mentre no acabe el fil principal.
- El fil principal arribarà poc després a les línies 12 a 14, on es declara la funció altreMissatge() que utilitzarem posteriorment per a escriure el nombre de Fibonacci obtingut.
- Amb això arribem ja a la línia 17. En ella s'escriu per primera vegada el resultat. En pantalla es mostrarà el següent: “M2: El resultat és: 165580141”
- I finalment arribem a les línies 19 a 21, on es programa novament la invocació d'altreMissatge() (missatge M3) després d'una espera d'1 ms. Amb això acaba l'execució del fil principal.
- En aquest moment en la cua d'esdeveniments solament trobem un torn actiu. Iniciem la seua execució. En aquest torn s'imprimirà per pantalla aquest missatge: “M1: Vull escriure açò...” Durant aquesta escriptura haurà finalitzat l'espera d'1 ms iniciada en el punt anterior. Amb això, es diposita un nou context en la cua d'esdeveniments, que vol escriure el missatge M3.
- Una vegada mostrat el missatge M1, el primer torn que havíem creat finalitza. Es revisa la cua d'esdeveniments i s'observa que en ella existeix encara un altre context. S'inicia la seua execució, que mostra per pantalla el missatge: “M3: El resultat és: 165580141”.
- Amb això acaba l'execució del programa. L'eixida completa proporcionada és:

```
Iniciant execució...
M2: El resultat és: 165580141
M1: Vull escriure açò...
M3: El resultat és: 165580141
```

Com pot observar-se, les dues primeres línies han sigut impreses pel fil principal. Per la seua banda, les línies que comencen amb M1 i M3 han sigut impreses utilitzant dos esdeveniments. Encara que el primer dels esdeveniments va ser generat molt prompte (mentre s'estava executant la funció fibo(); prou abans que es volguera escriure el missatge M2) el seu resultat no va poder mostrar-se llavors per pantalla. Ha hagut d'acabar abans el fil principal.

En la fulla 30 de la presentació es mostra un altre programa, basat en la classe EventEmitter, que també utilitza la cua d'esdeveniments. Revisem aquest exemple:

```
1: /*********************************************************/
2: /* Events1.js                                            */
3: /*********************************************************/
```

```
 4: var ev = require('events');
 5: var emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: var i1 = "print";
 8: var i2 = "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(i1, function() {
16:   console.log( "Event " + i1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(i2, function() {
19:   console.log( "Event " + i2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(i1, 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(i1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(i2);}, 3000 );
```

El programa principal instància l'objecte “emitter” en la línia 5 i crea quatre variables entre les línies 7 i 11. Posteriorment associa tres funcions als dos esdeveniments que van a crear-se. Finalment, en les línies 28 i 29 es programa l'esdeveniment “i1” (print) perquè ocórrega cada dos segons i en les línies 31 i 32 es programa l'esdeveniment “i2” (read) perquè ocórrega cada tres segons. Fet això, el fil principal acaba.

Cada vegada que es genere l'esdeveniment “i1” es dipositaran en la cua d'esdeveniments dues funcions, segons s'indica en les línies 15 a 17 (aquesta funció imprimeix el nombre de vegades que ha ocorregut l’esdeveniment) i en les línies 23 i 24 (aquesta funció sempre imprimeix el missatge “Something has been printed!!”). Per la seua banda, cada vegada que es genere l'esdeveniment “i2” s'inserirà en la cua d'esdeveniments la funció especificada en les línies 18 a 20 que imprimeix el nombre de vegades que ha ocorregut l'esdeveniment “read”.
Com els intervals de generació dels esdeveniments són molt amplis, la cua d'esdeveniments romandrà normalment buida. Quan es generen els esdeveniments, es deixen les seues funcions tractadores en la cua d'esdeveniments i passen a executar-se de seguida, buidant-se de nou la cua de torns.

La part inicial de l'eixida proporcionada serà:

```
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.
```

Com a exercici senzill, justifique quina serà l'eixida del següent programa, on s'estén lleugerament l'exemple anterior.

```
 1: /*************************************************************/
 2: /* Events2.js                                                */
 3: /*************************************************************/
 4: var ev = require('events');
 5: var emitter = new ev.EventEmitter;
 6: // Names of the events.
 7: var i1 = "print";
 8: var i2 = "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(i1, function() {
16:   console.log( "Event " + i1 + " has " +
17:     "happened " + ++num1 + " times.")});
18: emitter.on(i2, function() {
19:   console.log( "Event " + i2 + " has " +
20:     "happened " + ++num2 + " times.")});
21: // There might be more than one listener
22: // for the same event.
23: emitter.on(i1, 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(i1);}, 2000 );
30: // Second event generated every 3 seconds.
31: setInterval( function() {
32:    emitter.emit(i2);}, 3000 );
33: // Loop.
34: while (true)
35:    console.log(".");
```

4. Gestió de mòduls

La fulla 28 de la presentació descriu com poden exportar-se les funcions "públiques" que anem a definir en un mòdul, utilitzant "`exports`", i com han d'importar-se des d'uns altres mitjançant "`require()`". Allí es proporcionen alguns exemples d'ús.

El que no es comenta en ella és que "`exports`" és només un àlies para "`module.exports`" i que tant "`module.exports`" com "`module`" són objectes JavaScript. Això implica que, en ser objectes, disposaran d'un conjunt dinàmic de propietats, que podrem ampliar o reduir a voluntat. Cada mòdul té un objecte "module" privat. No és un objecte global comú a tots els fitxers. Amb ell podem especificar quines operacions i quines variables seran exportades pel mòdul.

Per a afegir una propietat o mètode a un objecte n'hi ha prou amb declarar-los i assignar-los un valor. És el que es mostra en els exemples d'aquella fulla. Tant "`area()`" com "`circumference()`" s'utilitzen com a noves propietats de l'objecte "module.exports". A l'ésser de tipus funció, s'aconsegueix així exportar-les com a operacions del mòdul.

Per a eliminar una propietat n'hi ha prou amb utilitzar l'operador "**delete**" seguit del nom de la propietat a eliminar.

Aquest mecanisme permetrà que puguem construir mòduls que importen a uns altres i modifiquen algunes de les seues operacions, mantenint les altres. Per exemple, aquest codi...

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

delete c.circumference;
```

```
c.circumferència= function( r ) {
  return MATH.PI * r * 2;
}

module.exports = c;
```

...és capaç de renombrar l'operació "`circumference()`" del mòdul "Circle.js" presentat en la fulla 28, passant a anomenar-la "`circumferència()`", sense afectar la resta d'operacions del mòdul original. Per a aconseguir-ho només necessita fer el següent:

1. Quan s'importa un mòdul mitjançant "`require()`", assignem l'objecte exportat a una variable del nostre programa. En aquest cas és la variable "`c`", que mantindrà l'objecte exportat en el fitxer "Circle.js".
2. A continuació eliminem una de les operacions exportades: "`circumference()`". L'altra operació, "`area()`", no s'ha vist afectada.
3. Posteriorment afegim una nova operació "`circumferència()`" que en aquest cas manté un codi similar al que tenia l'operació original.
4. Com a última sentència, exportem tot l'objecte "`c`", amb el que s'oferiran les operacions "`area()`" i "`circumferència()`" a qui ho importe.

Si aquest codi es guardara en un fitxer "Cercle.js", ja tindríem un mòdul amb aquest nom que exportaria les dues operacions.

Bibliografia

[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", Disponible en: http://javascript.info/tutorial/closures, 2011.