**ΘΕΜΑ ΔΙΠΛΩΜΑΤΙΚΗΣ**

“Αυτοματοποιημένη αναδόμηση Javascript κώδικα για εισαγωγή ES6 κλάσεων και ενοτήτων”

# 

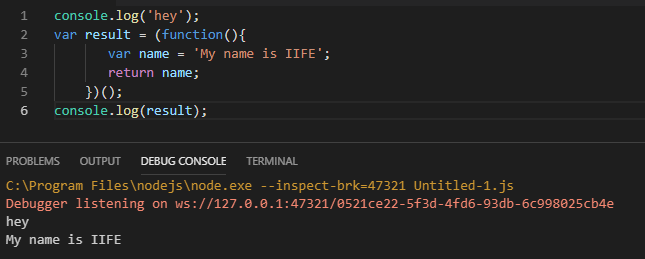
# ***Ορισμός IIFE***

Οι Immediately-Invoked Function Expressions (**IIFE**) είναι εκφράσεις που μπορούν να **εκτελεστούν** **αμέσως** **μόλις** **δημιουργηθούν**.

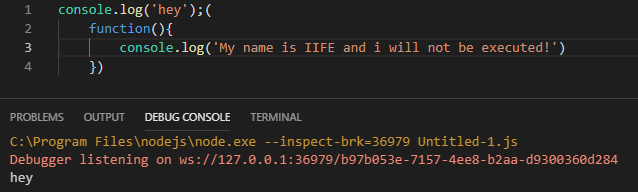
***Δήλωση***

|  |
| --- |
| ;(function() { /\* do some stuff here \*/ })() |
| Δηλαδή δηλώνεται μία συνάρτηση μέσα σε παρενθέσεις και **εκτελείται** προσθέτοντας παρενθέσεις στο τέλος → (/\* function \*/)(). Πολλές φορές για το σώμα των IIFE συναρτήσεων χρησιμοποιείται στην βιβλιογραφία η ορολογία “wrapper” για να υποδηλώσει το περιεχόμενο τους. |

**Επιτυχημένη κλήση**

****

**Αποτυχημένη κλήση λόγω παράλειψης των “execute” παρενθέσεων**

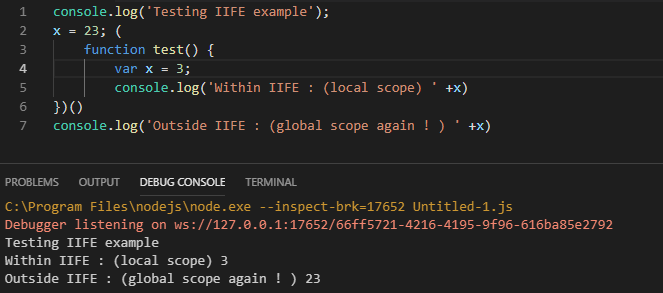
****

# ***Χρησιμότητα***

Είναι πολύ χρήσιμες καθώς [δεν επιβαρύνουν το global object](https://www.freecodecamp.org/forum/t/how-iife-cuts-link-to-global-environment/151851) – χρησιμοποιούν το δικό τους περιβάλλον που παρέχει μία εύκολη και απομονωμένη διαχείριση μεταβλητών. Ας εξετάσουμε ένα παράδειγμα για να γίνει πιο κατανοητό.

|  |
| --- |
| console.log('Testing IIFE example');  x = 23; (  function test() {  var x = 3;  console.log('Within IIFE : (local scope) ' +x)  })()  console.log('Outside IIFE : (global scope again ! ) ' +x) |
| Ας το σκεφτούμε σαν έναν καθρέφτη. Οι εκφράσεις IIFE μπορούν να “δουν” το global scope αλλά όχι το αντίστροφο. Με αυτόν τον τρόπο μπορείς να δημιουργήσεις συναρτήσεις στην καρδία του κώδικα χωρίς να υπάρχει ο κίνδυνος να υπάρχουν συγκρούσεις με άλλα κομμάτια κώδικα ή βιβλιοθήκες γεγονός που εισάγει την έννοια του local function scope καθιστώντας τον κώδικα πιο συντηρήσιμο. Στο συγκεκριμένο παράδειγμα ορίζεται εκτός IIFE και εντός η ίδια μεταβλητή x. Όσο βρίσκεται εντός της IIFE έχει την τιμή 3 που έχει ορίσει σαν τοπική μεταβλητή η IIFE και ύστερα επανέρχεται στην αρχική της τιμή 23.  **Key word : ενθυλάκωση, block scope** |

Αποτελέσματα



# 

# **Namespaces in Node.js**

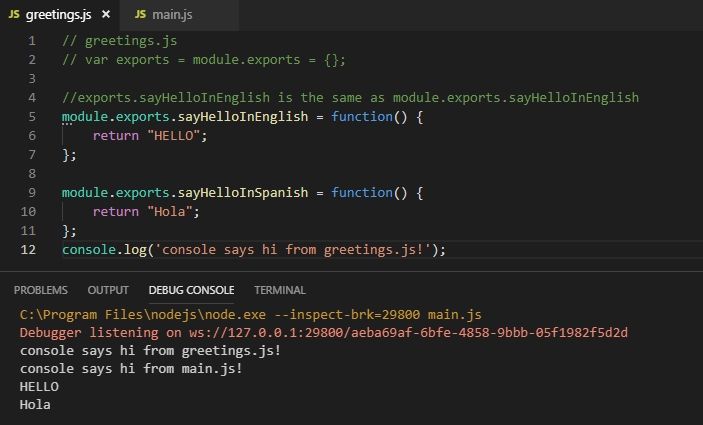
“***Namespaces are used to organize/encapsulate your code. External modules are used to organize/encapsulate your code AND to locate your code at runtime*.**” δηλαδή πρακτικά τα namespaces εξυπηρετούν τον ίδιο σκοπό με τα modules με την διαφορά πως στην περίπτωση των δεύτερων ο κώδικας θα ψάξει κατά την εκτέλεση να εντοπίσει όλα τα modules. (βιβλ [9])

Παρακάτω ακολουθεί ένα πολύ απλοποιημένο παράδειγμα (με χρήση των module export και require) σε Node.js προκειμένου να πάρουμε την γενική ιδέα σχετικά με τα modules (ορισμός 1).

|  |
| --- |
| // main.js  var greetings = require("./greetings.js");  console.log('console says hi from main.js!');  // "Hello"  console.log(greetings.sayHelloInEnglish());  // "Hola"  console.log(greetings.sayHelloInSpanish()); |
| Εδώ με την χρήση της ειδική συνάρτησης που έχει η Node.js **require**("./greetings.js"); μπορεί να επιτευχθεί η εισαγωγή οποιαδήποτε module στον κώδικά μας. Πιο συγκεκριμένα στην μεταβλητή greetings εισάγεται ένα αντικείμενο που έχει πρόσβαση σε όλες τις συναρτήσεις του greetings.js. Όπως βλέπουμε μέσω του συγκεκριμένου αντικειμένου μπορούμε να καλέσουμε τις συναρτήσεις sayHelloInEnglish()και sayHelloInSpanish() και ας βρίσκονται σε διαφορετικό αρχείο από το main.js. Αυτό καθιστά τον κώδικα πιο οργανωμένο και επαναχρησιμοποιήσιμο. (βασική προϋπόθεση για να λειτουργήσει η χρήση των συναρτήσεων είναι να έχουν γίνει ‘export’ στο greetings.js όπως θα δούμε παρακάτω!) |

|  |
| --- |
| // greetings.js  // var exports = module.exports = {};  //exports.sayHelloInEnglish  module.exports.sayHelloInEnglish = function() {  return "HELLO";  };  module.exports.sayHelloInSpanish = function() {  return "Hola";  };  console.log('console says hi from greetings.js!'); |
| Εδώ ορίζεται το module που θα χρησιμοποιηθεί σαν επαναχρησιμοποιήσιμος κώδικας σε οποιοδήποτε αρχείο το κάνει import. Πιο συγκεκριμένα η συνάρτηση sayHelloInSpanish γίνεται export προκειμένου να μπορεί να χρησιμοποιηθεί σε οποιοδήποτε js κάνει require το greetings.js.  Το export μπορεί να γίνει με δύο τρόπους 1.module.exports.sayHelloInEnglish ή εν συντομία 2.exports.sayHelloInEnglish (η φιλοσοφία μας θυμίζει αρκετά δομημένο προγραμματισμό). |

Όπως παρατηρούμε και από την θεωρία λόγω της require πριν τρέξει ο κώδικας θα πρέπει να βρει όλα τα απαραίτητα αρχεία που έχουν οριστεί από την require. Παρακάτω τα αποτελέσματα. (βιβλ [10])



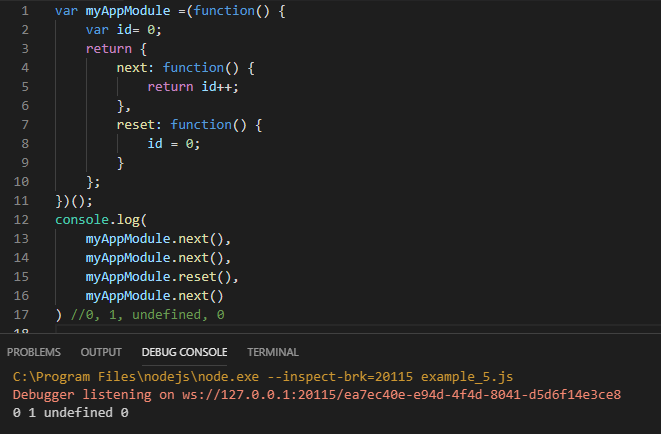
# **IIFE για προσομοίωση Modules / Namespaces**

Προκειμένου να γίνει κατανοητό όμως το πως συνδέονται τα namespaces και τα modules με τις εκφράσεις IIFE θα πρέπει να περιγράψουμε την χρησιμότητα του namespace.

“*JavaScript is designed in such a way that it is very easy to create global variables that have the potential to interact in negative ways. The practice of namespacing is usually to create an object literal encapsulating your own functions and variables, so as not to collide with those created by other libraries*”

Συνεπώς τα namespaces εξυπηρετούν το να μπορούν να δημιουργούνται αντικείμενα με δικά τους functions και μεταβλητές προκειμένου να μην δημιουργούνται collisions είτε με τα global variables είτε με άλλες συναρτήσεις (ότι δεν γίνεται return είναι local scope). Αυτό μας θυμίζει όμως αρκετά και τις IIFE εκφράσεις όπως περιγράψαμε την χρησιμότητά τους παραπάνω. Ας δούμε ένα παράδειγμα παρακάτω προκειμένου να λυθούν όλες οι σχετικές απορίες σχετικά με το πώς συνδυάζονται αυτά τα δύο.

|  |
| --- |
| var myAppModule =(function() {  var id= 0;  return {  next: function() {  return id++;  },  reset: function() {  id = 0;  }  };  })();  console.log(  myAppModule.next(),  myAppModule.next(),  myAppModule.reset(),  myAppModule.next()  ) //0, 1, undefined, 0 |
| “Self Invoking function wrapper returns an object and assigning the result to a namespace variable. Αny variables not included in the return value will remain forever private, visible only to the public functions that reference them.”  δηλαδή η συνάρτηση θα τρέξει με την λογική των IIFE - ακολουθεί άλλωστε και την σύνταξη τους και το αποτέλεσμα αυτής μέσω της return θα είναι ένα αντικείμενο που θα αποθηκευτεί στην μεταβλητή **myAppModule.**Μέσω του αντικειμένου μπορούμε να καλέσουμε τις 2 συναρτήσεις που επιστρέφονται από την IIFE μέσω της return (next,reset). Η ιδιαιτερότητα του συγκεκριμένου παραδείγματος είναι πως σε μία IIFE που τρέχει επιστρέφονται συναρτήσεις που έχουν οριστεί εμφωλευμένα - συναρτήσεις που δεν ακολουθούν το συγκεκριμένο πρότυπο.  \*\*Αξίζει να σημειωθεί πως το συγκεκριμένο pattern var abc = “IIFE function” είναι ιδιαίτερα διαδεδομένο όπως θα παρατηρήσουμε και στις παραλλαγές των IIFE. |

****

# **IIFE & Namespace Injection**

Namespace injection είναι μία παραλλαγή των IIFE σύμφωνα με την οποία εισάγεται δυναμικά σε ένα namespace επιπρόσθετη λειτουργικότητα (μέθοδοι και μεταβλητές) μέσω ενός IIFE wrapper.

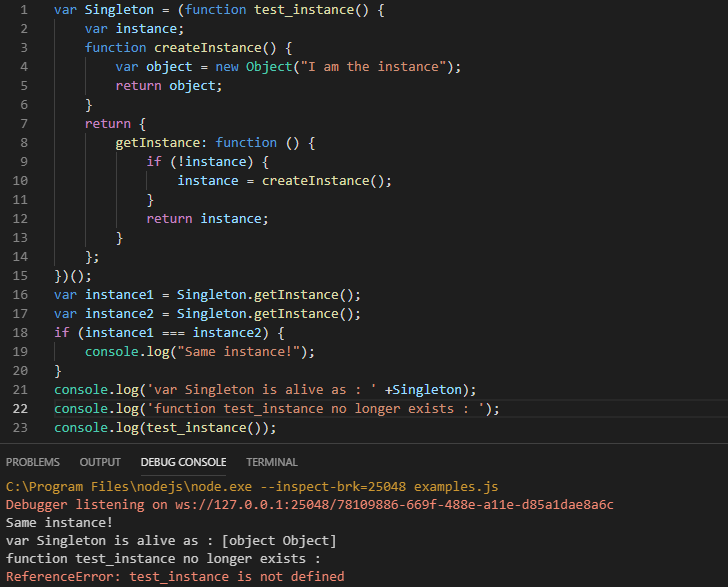
|  |
| --- |
| **var myApp = myApp || {};** **myApp.utils = {};** **(function () {**  var val = 5;  this.getValue = function () { return val; };  this.setValue = function( newVal ) { val = newVal;}  // also introduce a new sub-namespace  **this.tools = {};** **}).apply( myApp.utils );**  /\* inject new behaviour into the tools namespace  which we defined via the utilities module \*/ **(function () {**  this.diagnose = function(){ return "diagnosis"; } **})**.**apply( myApp.utils.tools );**  /\* note, this same approach to extension could be applied to a regular IIFE, by just passing in the context as an argument and modifying the context rather than just "this" \*/ // Usage: Outputs our populated namespace **console.log( myApp );** // Outputs: 5 **console.log( myApp.utils.getValue() );** // Sets the value of `val` and returns it **myApp.utils.setValue( 25 );**  **console.log( myApp.utils.getValue() );** // Testing another level down **console.log( myApp.utils.tools.diagnose() );** |
| Στο παραπάνω παράδειγμα φαίνεται το πώς μπορείς δυναμικά όχι μόνο να δημιουργήσεις namespaces αλλά και να τους δώσεις ιδιότητες. Αρχικά ορίζεται μία μεταβλητή myApp σαν ένα πρώτο namespace και μέσω αυτού ορίζεται και ένα νέο sub-namespace utils με έναν πλέον κλασικό τρόπο “myApp.**utils** = {}”. Ύστερα ορίζεται μια IIFE η οποία δέχεται σαν όρισμα μέσω του .apply το namespace myApp.utils. Η συγκεκριμένη IIFE συνάρτηση - wrapper μέσω του this.tools εισάγει ένα νέο namespace - tools (“this.tools = {} ”). Ύστερα αυτό το νέο namespace που δημιουργήθηκε δυναμικά περνάει με την ίδια λογική σαν όρισμα σε έναν νέο IIFE wrapper ο οποίος δημιουργεί την συνάρτηση this.diagnose(). Σχετικά με τα αποτελέσματα στην κονσόλα :   * console.log( myApp ) → είναι ένα αντικείμενο που εμπεριέχει το αντικείμενο myApp.**utils** που ορίστηκε στην δεύτερη γραμμή μέσω του “myApp.utils = {};” * console.log( myApp.utils.getValue() ); → μέσω του πρώτου wrapper / IIFE συνάρτησης το myApp.utils περνάει σαν όρισμα που δημιουργεί την μεταβλητή val = 5 και την επιστρέφει μέσω της συνάρτησης function () { return val; }; * myApp.utils.setValue( 25 ); → η τιμή 25 περνάει σαν όρισμα στο myApp.utils που μέσω του apply περνάει στο this.setValue που καλεί την συνάρτηση function( newVal ) { val = newVal;} περνώντας στο newVal την τιμή 25 για να την κάνει τελικά assign στην μεταβλητή val που είχε αρχικά την τιμή 5. * console.log( myApp.utils.getValue() ); → ύστερα καλεί πάλι την συνάρτηση getValue() για να φανεί η νέα τιμή της getValue() * console.log( myApp.utils.tools.diagnose() ); → έχοντας δημιουργηθεί δυναμικά μέσα στον πρώτο wrapper το αντικείμενο this.tools = {}; (δηλαδή λόγω this το myApp.utils.tools) μέσω του apply( myApp.utils.tools ); περνάει σαν όρισμα στον δεύτερο wrapper και πάλι μέσω του this αρχικοποιεί μία νέα συνάρτηση που απλά επιστρέφει την τιμή "diagnosis"; που τελικά καταλήγει στο console.log και εμφανίζεται στην κονσόλα! |

# 

# **IIFE για προσομοίωση κλάσεων**

**Παράδειγμα 1 (namespace variable as singleton)**

|  |
| --- |
| var Singleton = (function test\_instance() {  var instance;  function createInstance() {  var object = new Object("I am the instance");  return object;  }  return {  getInstance: function () {  if (!instance) { instance = createInstance(); }  return instance;  }  })();  var instance1 = Singleton.getInstance();  var instance2 = Singleton.getInstance();  if (instance1 === instance2) {  console.log("Same instance!");  }  console.log(Singleton.test\_instance); |
| Εδώ ορίζεται μία IIFE συνάρτηση με namespace Singleton που αρχικοποιείται με ένα sigleton αντικείμενο (μοναδικό) με την βοήθεια ενός IIFE wrapper. Πιο συγκεκριμένα μέσω του var instance1 = Singleton.getInstance() καλείται η συνάρτηση test\_instance η οποία επιστρέφει μέσω του αντικειμένου getInstance ένα αντικείμενο που αρχικοποιείται μέσω της συνάρτησης createInstance() - εφόσον η if(!instance){} θα επιστρέψει true καθώς δημιουργείται για πρώτη φορά - που απλά δημιουργεί το αντικείμενο αυτό μέσω του var object = new Object που γίνεται return στο αντικείμενο getInstance και τελικά αρχικοποιεί την μεταβλητή - αντικείμενο instance 1. Εν συνεχεία με την ίδια λογική το instance 2 θα αρχικοποιηθεί πάλι με το ίδιο αντικείμενο καθώς αυτή την φορά το αντικείμενο Singleton έχει ήδη δημιουργηθεί από την προηγούμενη κλήση συνεπώς η if(!instance){} θα επιστρέψει false και θα επιστραφεί το ίδιο αντικείμενο χωρίς πάλι να γίνει η κλήση της συνάρτησης createInstance() (Βιβ. [1] ) |

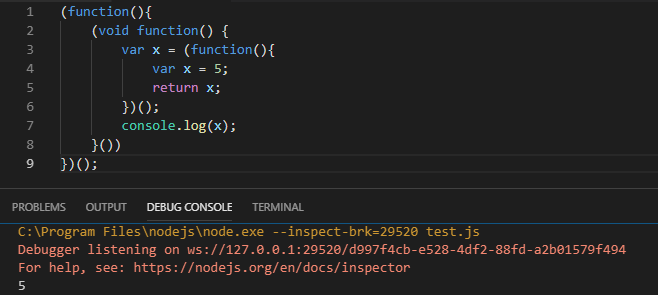
****

# 

# **Παραλλαγές IIFE**

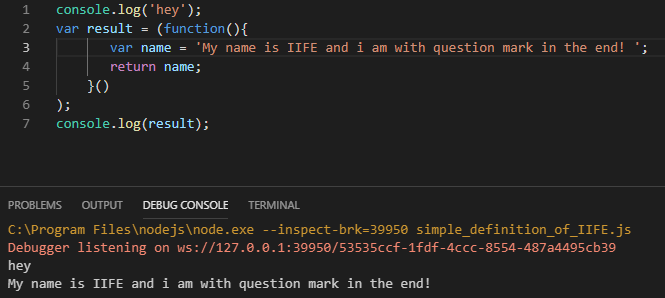
## Παράδειγμα 1 (void)

|  |
| --- |
| (function(){  (void function() {  var x = (function(){  var x = 5;  return x;  })();  console.log(x);  }())  })(); |
| Στο συγκεκριμένο παράδειγμα έχουμε την παραλλαγή της κλασικής σύνταξης μίας IIFE συνάρτησης λόγω της λέξης κλειδί void. Πιο συγκεκριμένα μία εξωτερική IIFE αρχικοποιεί μία νέα void IIFE (η οποία δεν επιστρέφει τίποτα) αλλά καλεί με την σειρά της μία τελευταία IIFE η οποία τελικά θα αρχικοποιήσει την τιμή της μεταβλητής x για να το εμφανίσει η void IIFE μέσω κονσόλας. |

****

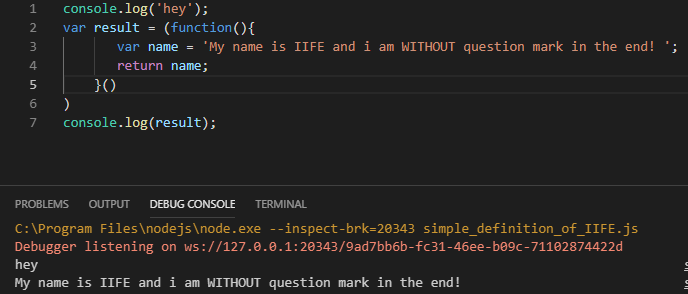
## Παράδειγμα 2 (transition parenthesis)

|  |
| --- |
| console.log('hey');  var result = ( function() {  var name = ‘My name is IIFE and i am with..’;  return name;  }()  );  console.log(result); |
| Αντί για το κλασσικό πρότυπο που οι παρενθέσεις που υποδηλώνουν την εκτέλεση της συνάρτησης μπαίνουν στο τέλος στην συγκεκριμένη παραλλαγή ακολουθούνται από ένα κλείσιμο παρένθεσης, Πριν → (function(){ /\*some stuff\*/ })() και μετά → (function(){ /\*some stuff\*/ }()) |

****

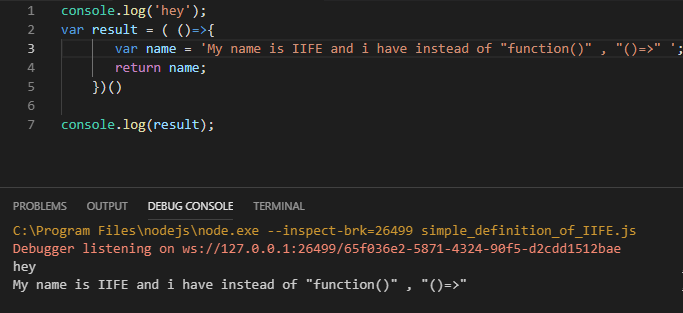
## Παράδειγμα 3 (no question mark)

|  |
| --- |
| console.log('hey');  var result = ( function() {  var name = ‘My name is IIFE and i am with..’;  return name;  }()  )  console.log(result); |
| Μετά από τις παρενθέσεις της εκτέλεσης μπορεί να αφαιρεθεί το ερωτηματικό Πριν → (function(){ /\*some stuff\*/ }()); και μετά → (function(){ /\*some stuff\*/ }()) |

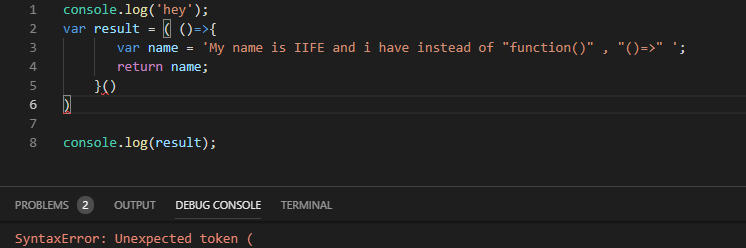
****

## Παράδειγμα 4 (no “function” in declaration)

|  |
| --- |
| console.log('hey'); var result = ( ()=> {   var name = ‘My name is IIFE and i am with..’;   return name; }() ) console.log(result); |
| Αντί για function() οι παρενθέσεις έρχονται μπροστά και η λέξη function αλλάζει με => Πριν → (function(){ /\*some stuff\*/ }()); και μετά → ( ()=>{ /\*some stuff\*/ }()); |

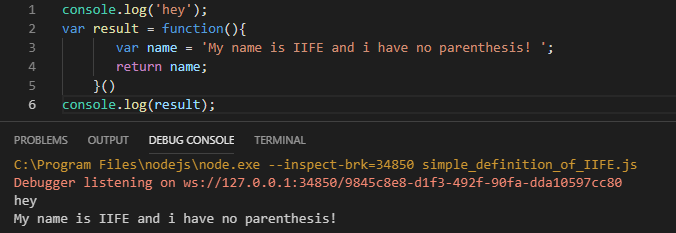
****

**\*\* Αξίζει να σημειωθεί πως σε μία προσπάθεια παραλλαγής με τις παρενθέσεις (κοίτα παραλλαγή 2) θα υπήρχε συντακτικό λάθος καθώς εδώ ακολουθείται μία πιο αυστηρή δομη.**

****

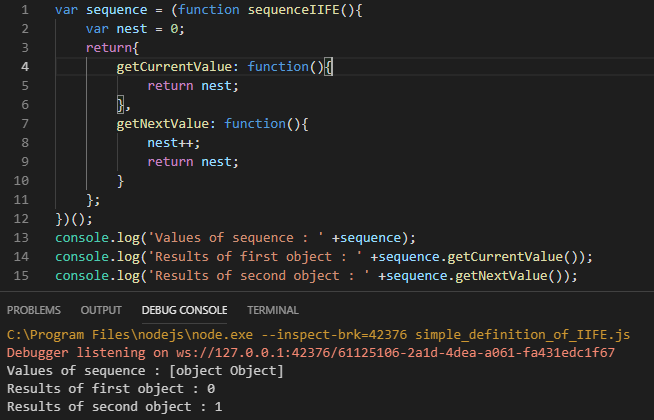
## Παράδειγμα 5 (no parenthesis)

|  |
| --- |
| console.log('hey');  var result = function() {  var name = ‘My name is IIFE and i am with..’;  return name;  }()  console.log(result); |
| Δεν υπάρχουν παρενθέσεις στον ορισμό της συνάρτησης. Πριν → (function(){ /\*some stuff\*/ }()); και μετά → function(){ /\*some stuff\*/ }(); |

****

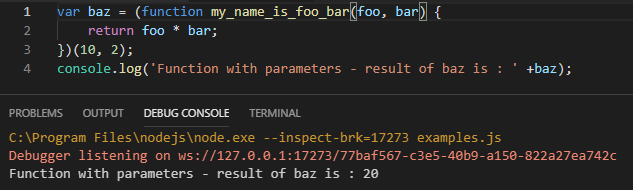
## Παράδειγμα 6 (multiple & nested IIFEs)

|  |
| --- |
| var sequence = (function sequenceIIFE(){  var nest = 0;  return{  getCurrentValue: function(){  return nest;  },  getNextValue: function(){  nest++;  return nest;  }  };  })();  console.log('Values of sequence : ' +sequence);  console.log('Results of first object : ' +sequence.getCurrentValue());  console.log('Results of second object : ' +sequence.getNextValue()); |
| Πολλαπλές και εμφωλευμένες εκφράσεις με την χρήση κόμματος.  Πριν →(function(){ /\*some stuff\*/ }());  και μετά → (function(){  return{  resulted\_object : (function(){/\*\*/}), resulted\_object : (function(){/\*\*/})  }; })() |

****

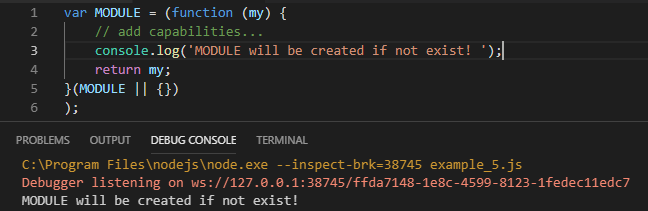
## Παράδειγμα 7 (An anonymous function with parameters that returns a value)

|  |
| --- |
| var baz = (function my\_name\_is\_foo\_bar(foo, bar) {  return foo \* bar; })(10, 2); console.log('Function with parameters - result of baz is : ' +baz); |
| IIFE συνάρτηση με όνομα και ορίσματα! (Βιβ. [2] , σελ. 35) |

****

## Παράδειγμα 8 (Loose Augmentation)

|  |
| --- |
| var MODULE = (function (my) {  // add capabilities...  console.log('MODULE will be created if not exist! ');  return my;  }(MODULE || {})  ); |
| Η ιδιομορφία βρίσκεται κατά την εκτέλεση του namespace-module καθώς δημιουργείται εάν δεν υπάρχει ήδη. |

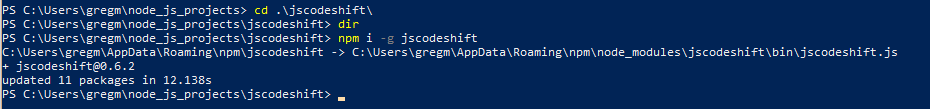
****

# **Jscodeshift & Codemods**

## Τί είναι το jscodeshift?

The jscodeshift toolkit allows you to pump a bunch of source files through a transform and replace them with what comes out the other end. Inside the transform, you parse the source into an abstract syntax tree (AST), poke around to make your changes, then regenerate the source from the altered AST.  
  
The interface that jscodeshift provides is a wrapper around recast and ast-types packages. recast handles the conversion from source to AST and back while ast-types handles the low-level interaction with the AST nodes.

## Installation , Nodes , Node-paths , Codemod

Για την εγκατάσταση του jscodeshift μέσω του npm αρκεί η παρακάτω εντολή :  
  
npm i -g jscodeshift  
  
  
**Nodes**

Nodes are the basic building blocks of the AST, often referred to as “AST nodes.” These are what you see when exploring your code with AST Explorer. They are simple objects and do not provide any methods.

**Node-paths**

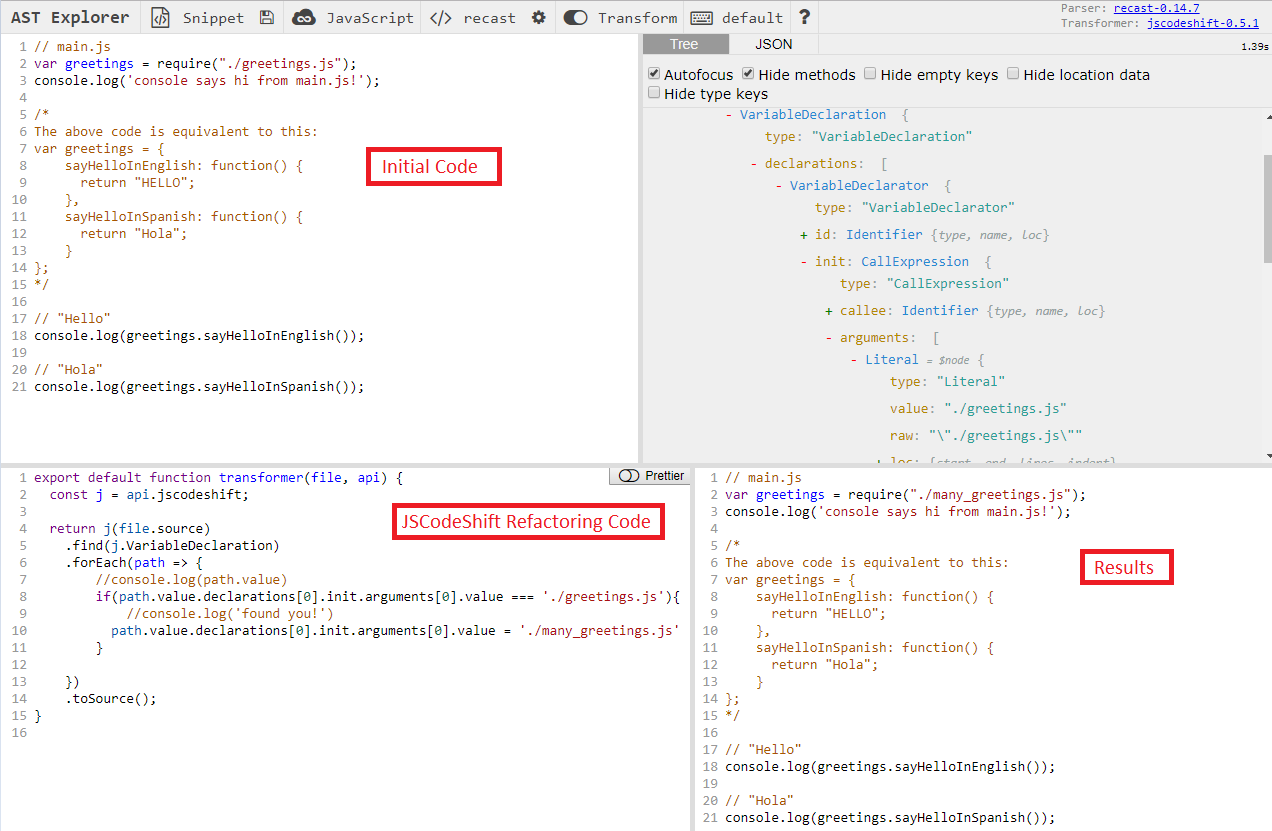
Node-paths are wrappers around an AST node provided by ast-types as a way to traverse the abstract syntax tree. In isolation, nodes do not have any information about their parent or scope, so node-paths take care of that. You can access the wrapped node via the node property and there are several methods available to change the underlying node. node-paths are often referred to as just “paths.”

**Codemod**

Codemod is a tool/library to assist you with large-scale codebase refactors that can be partially automated but still require human oversight and occasional intervention.

## JSCodeshift Codemods & AST Explorer

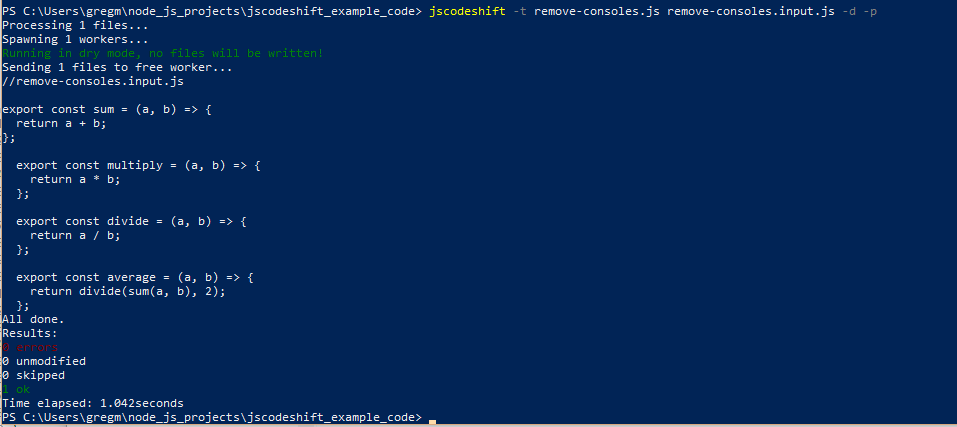
Παρακάτω απεικονίζεται το πώς θα μπορούσαμε με την βοήθεια του AST Explorer (<https://astexplorer.net/>) να δούμε τα αποτελέσματα που θα είχε ένα codemod που έχουμε γράψει.



# Εφαρμογές με χρήση του Codemod

## Παράδειγμα 1 - Αφαίρεση calls σε console

|  |  |
| --- | --- |
| //remove-consoles.js  export default (fileInfo, api) => {  const j = api.jscodeshift;  const root = j(fileInfo.source)  const callExpressions = root.find(j.CallExpression, {  callee: {  type: 'MemberExpression',  object: { type: 'Identifier', name: 'console' },  },  }  );  callExpressions.remove();  return root.toSource();  }; | //remove-consoles.input.js  export const sum = (a, b) => {  console.log('calling sum with', arguments);  return a + b;  };    export const multiply = (a, b) => {  console.warn('calling multiply with',  arguments);  return a \* b;  };  export const divide = (a, b) => {  console.error(`calling divide with ${ arguments }`);  return a / b;  };  export const average = (a, b) => {  console.log('calling average with ' + arguments);  return divide(sum(a, b), 2);  }; |
| // console command  jscodeshift **-t** remove-consoles.js remove-consoles.input.js **-d -p**  Ενδεικτικά αναφέρουμε πως ο παραπάνω κώδικας θα τρέξει το remove-consoles.input.js μέσω του remove-consoles.js (**-t**) και θα εμφανίσει εφόσον δεν υπάρχουν σφάλματα τα αποτελέσματα στην οθόνη(**-p**) χωρίς να κάνει την οποιαδήποτε αλλαγή στο αρχικό αρχείο (**-d**).  (για λεπτομέρειες σχετικά με τα ορίσματα → ‘[JSCodeshift Options - Πίνακας CLI’](#_8agn4h205u1z)) - βιβλ [13] | |

Παρακάτω φαίνονται τα αποτελέσματα της κονσόλας :

# Βιβλιογραφία

**[1]**<https://medium.com/@vvkchandra/essential-javascript-mastering-immediately-invoked-function-expressions-67791338ddc6> ( Article → Essential JavaScript: Mastering Immediately-invoked Function Expressions , Chandra Gundamaraju , 2017)

**[2]** Pro JavaScript™ Design - The essentials of object-oriented JavaScript™ programming Patterns , Ross Harmes and Dustin Diaz , 2008

**[3]** JavaScript Patterns , Stoyan Stefanov , 2010

**[4]** Simplifying JavaScript Writing Modern JavaScript with ES5 ES6 and Beyond , Joe Morgan , 2018

**[5]** Node.js Design Patterns , Mario Casciaro and Luciano Mammino , 2016

**[6]**<https://softwareengineering.stackexchange.com/questions/128734/self-executing-anonymous-function-vs-prototype> ([Self-Executing Anonymous Function vs Prototype](https://softwareengineering.stackexchange.com/questions/128734/self-executing-anonymous-function-vs-prototype))

**[7]**<https://javascriptweblog.wordpress.com/2010/12/07/namespacing-in-javascript/> (Namespacing in JavaScript , Angus Croll , 2010)

**[8]**<https://hackernoon.com/a-definitive-guide-to-javascript-prototypes-2c263788021e>(A Definitive Guide to JavaScript Prototypes , Kevin Lee)

**[9]**<https://stackoverflow.com/questions/38582352/module-vs-namespace-import-vs-require-typescript> ([Module vs Namespace - Import vs Require Typescript](https://stackoverflow.com/questions/38582352/module-vs-namespace-import-vs-require-typescript))

**[10]** https://www.sitepoint.com/understanding-module-exports-exports-node-js/

**[11]** <http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html>

**[12]** <https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch13s15.html>

**[13]** <https://www.toptal.com/javascript/write-code-to-rewrite-your-code> (“Write Code to Rewrite Your Code: jscodeshift”Jeremy Greer , 2016)

**[14]** <https://www.sitepoint.com/getting-started-with-codemods/> (Refactor Code in Your Lunch Break: Getting Started with Codemods , Chris Laughlin , 2017)

**[15]** <https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e> - (A Simple Guide to ES6 Iterators in JavaScript with Examples, Arfat Salman , 2018)

# 

# GITHUB & NPM Projects

[1] <https://github.com/facebook/codemod>

[2] <https://github.com/sejoker/awesome-jscodeshift>

[3] <https://github.com/gmavridakis/jscodeshift_diplomatiki>

[4] <https://www.npmjs.com/package/gulp-iife> (A Gulp plugin for wrapping JavaScript code within immediately invoked function expressions )

[5] <https://github.com/5to6/5to6-codemod> (A collection of codemods that allow you to transform your JavaScript code from ES5 to ES6 using jscodeshift.)

[6] <https://www.npmjs.com/package/jscodeshift-helper> (This little helper will log what you're looking at, give you an API overview, and point you to references.)

# JSCodeshift Options - Πίνακας CLI

|  |  |
| --- | --- |
| $ jscodeshift --help  Usage: jscodeshift [OPTION]... PATH...  or: jscodeshift [OPTION]... -t TRANSFORM\_PATH PATH...  or: jscodeshift [OPTION]... -t URL PATH...  or: jscodeshift [OPTION]... --stdin < file\_list.txt | |
| -c, --cpus=N | start at most N child processes to process source files (default: max(all - 1, 1)) |
| -d, --(no-)dry  --extensions=EXT | dry run (no changes are made to files)  (default: false)  transform files with these file extensions (comma separated list) (default: js) |
| -h, --help | print this help and exit |
| -p, --(no-)print | print transformed files to stdout, useful for development (default: false) |
| -s, --(no-)silent | do not write to stdout or stderr (default: false) |
| -t, --transform=FILE | path to the transform file. Can be either a local path or url (default: ./transform.js) |
| -v, --verbose=0|1|2  --version | show more information about the transform process (default: 0)  print version and exit |
| (projects [2]) | |