

# Event programming in JavaScript and Node.js: an introduction

Antonio Natali

Alma Mater Studiorum – University of Bologna  
viale Risorgimento 2, 40136 Bologna, Italy  
`antonio.natali@unibo.it`

# Table of Contents

Event programming in JavaScript and Node.js: an introduction .....	1
<i>Antonio Natali</i>	
1 Introduction.....	2
2 JavaScript and Node.js .....	2
2.1 JavaScript .....	2
2.2 Node.js .....	3
2.2.1 npm. ....	3
2.2.2 Frameworks and altJS. ....	3
2.3 ES6 .....	4
2.4 Strict mode.....	4
3 Basic concepts .....	5
3.1 Variables and scoping .....	5
3.1.1 Immediately Invoked Function Expression (IIFE). ....	5
3.2 Functions and hoisting .....	5
3.3 Define and invoke functions .....	6
3.4 Function as constructor .....	6
3.5 The variable <code>this</code> .....	7
3.6 Objects .....	7
3.7 Methods .....	7
3.8 Constructors.....	8
3.9 Prototypes .....	8
3.10 Inheritance .....	8
3.11 The property <code>__proto__</code> .....	9
3.12 Higher order functions .....	9
3.13 Lexical closures .....	10
3.13.1 Closure .....	10
3.14 Callbacks.....	11
3.15 Continuation passing style (CPS).....	11
4 A deeper view .....	13
4.1 The variable <i>arguments</i> .....	13
4.2 call and apply.....	13
4.3 Function.prototype.bind() .....	13
4.3.1 bind .....	14
4.4 Methods as callbacks .....	14
4.5 Arrow functions .....	15
4.6 About <code>this</code> .....	15
4.7 About <code>this</code> in Node .....	17
4.8 Tail recursion and Trampoline .....	17
5 Asynchronous Operations .....	20
5.1 event loop .....	20
5.2 setTimeout .....	20
5.2.1 setTimeout at work .....	21
5.2.2 setTimeout and scope .....	22
5.2.3 setTimeout and bind .....	23
5.3 Calling a function (pseudo-)asynchronously .....	23

---

5.4	The callback hell .....	24
6	Node.js .....	26
6.1	Standard input and count down .....	26
6.2	Node module.exports and require .....	26
6.3	Browserify .....	27
6.4	Event emitters and Event listeners .....	27
6.5	process.nextTick .....	27
6.6	setImmediate .....	28
6.7	The <code>fs</code> library .....	29
6.8	The <code>net</code> library .....	29
6.8.1	TcpServer. ....	30
6.8.2	TcpClient. ....	30
6.9	The <code>http</code> library .....	31
6.9.1	HTTP server that returns the content of a JSON file. .....	32
6.9.2	Http CRUD Server. .....	32
6.9.3	cURL .....	33
7	Node on RaspberryPi .....	35
7.1	Blink a Led with JavaScript .....	35

---

## 1 Introduction

These days, computers and embedded devices are *event-driven systems*, that continuously wait for the occurrence of some external or internal *event*. After recognizing the event, they react by performing the appropriate computation (event-driven systems are also called *reactive systems*). Once the event handling is complete, the software goes back to a dormant state (an idle loop or a power-saving mode) in anticipation of the next event.

When we begin to work with a event-driven framework, there is no more discernible flow of control - the main routine doesn't do anything except start the framework's *event-loop* (see *Subsection ??*): once the event-loop is started, it is the code hidden inside the framework that drives the action. What's left of the program seems to be little more than a collection of *event-handlers*: the program structure seems to be turned inside out.

So procedural programmers often find that, on first encounter, event-driven programming makes no sense at all. Experience and familiarity will gradually lessen that feeling, but moving from procedural programming to event-driven programming requires a very real *mental paradigm shift*.

In this notes, we will attempt to gradually understand event-driven programming and the paradigm shift.

## 2 JavaScript and Node.js

JavaScript (written and introduced by Brendan Eich in 1995) is - alongside HTML and CSS - one of the three core technologies of World Wide Web content production. All browsers have JavaScript engines: Firefox has an engine called Spidermonkey, Safari has JavaScriptCore, and Chrome has an engine called V8<sup>1</sup>.

Node.js<sup>2</sup> (written and introduced by Ryan Dahl in 2009<sup>3</sup>) is an open-source server side runtime environment built on Chrome's V8. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.

### 2.1 JavaScript

ECMAScript is the official name of JavaScript, but nowadays ECMAScript is the name used by the language specification, while JavaScript denotes the (high-level, dynamic, untyped, and interpreted<sup>4</sup>) programming language.

JavaScript<sup>5</sup> is prototype-based with first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded.

JavaScript design was influenced by programming languages such as Self and Scheme. It is a single-threaded language where asynchronous tasks are handled with events. Thus, JavaScript and Java, are distinct languages that differ greatly in their design.

The culture of JavaScript is already geared towards event programming, since it is designed specifically to be used with an event loop; anonymous functions; closures; only one callback at a time; I/O through DOM event callbacks, etc.

#### JS timeline

- 1995: JavaScript is born as LiveScript
- 1997: ECMAScript standard is established
- 1999: ES3 comes out and IE5 is all the rage
- 2000 - 2005: XMLHttpRequest, a.k.a. AJAX, gains popularity in app such as Outlook Web Access (2000) and Oddpost (2002), Gmail (2004) and Google Maps (2005).
- 2009: ES5 comes out (this is what most of us use now) with `forEach`, `Object.keys`, `Object.create` (specially for Douglas Crockford), and standard JSON
- 2015: ES6 (formally called ECMAScript2015) comes out; its changes are geared toward facilitating the solution of problems that developers actually have. More controversial changes will perhaps introduced in ES7

---

<sup>1</sup> For some experiment, connect to the [|jsfiddle|](http://jsfiddle) site

<sup>2</sup> See [|nodejs.org/api|](http://nodejs.org/api), [|Tutorial node.js|](http://tutorialnodejs) and [|Node University site|](http://nodeuniversity).

<sup>3</sup> Visit Wikipedia [node.js](http://en.wikipedia.org/wiki/Node.js) to know the history of Node.js.

<sup>4</sup> Recent browsers perform just-in-time compilation.

<sup>5</sup> See [|JavaScript wiki|](http://en.wikipedia.org/wiki/JavaScript) and [|w3schools|](http://www.w3schools.com)

---

## 2.2 Node.js

Node.js is not a programming language; it is rather an interpreter and environment for JavaScript which includes a bunch of libraries for using JavaScript as a general-purpose programming language, with an emphasis on asynchronicity and non-blocking operations. Node actually runs the same interpreter as Google Chrome (V8), but provides a different set of libraries and a different run-time environment. It also includes a package management system (**npm**) and a few language extensions (for example modules) that we don't find standard in browsers<sup>6</sup>.

### Node.js

Node.js is an open source project designed to help you write JavaScript programs that talk to networks, file systems or other I/O (input/output, reading/writing) sources. It is just a simple and stable I/O platform that you are encouraged to build modules on top of. Node does I/O in a way that is asynchronous which lets it handle lots of different things simultaneously.

Node.js can be used to build different types of applications such as command line application, web application, real-time chat application, REST API server etc. Every Node application runs on a single thread. What this means is that apart from I/O - at any time, only one task/event is processed by Node's event loop. The event loop can be viewed as a queue of callbacks (see Subsection 3.14) that are processed by Node on every 'tick' (cycle) of the event loop. So, even we are running Node on a multi-core machine, we will not get any parallelism in terms of actual processing - all events will be processed only one at a time.

This is why Node is a great fit for *I/O bound tasks*, and definitely not for *CPU intensive tasks*. For every I/O bound task, you can simply define a callback that will get added to the event queue. The callback will fire when the I/O operation is done, and in the mean time, the application can continue to process other I/O bound requests.

### 2.2.1 npm. Useful npm commands are:

- **npm init**: creates a package.json
- **npm list**: lists all installed packages
- **npm prune**: removes packages not depended on by your project according to package.json
- **npm outdated**: tells you which installed packages are outdated with respect to what is current in the npm registry but allowable by the version definition in package.json

**2.2.2 Frameworks and altJS.** Plain JavaScript and the *jQuery* library<sup>7</sup> has been used for years to build complex web interfaces but with lot of effort and complexity in code development and maintenance. In fact, dealing with complex sets of events in an elegant way is still frontier territory in JavaScript. To overcome the problem, *altJS* languages<sup>8</sup> are aimed specifically at "taming" asynchronous callbacks by allowing them to be written in a more synchronous style.

Moreover, many JavaScript frameworks<sup>9</sup> have been proposed to facilitate the building of interactive web applications. Most of the JavaScript frameworks work on MVC design paradigm and enforce structure to ensure more scalable, reusable, maintainable JavaScript code.

There are also many Node.js frameworks<sup>10</sup> that allow us to build real time end to end web applications without the need of any other third party web server, app server, tool or technology. If we need to create APIs, there are more specialized node.js api framework like LoopBack, actionHero.js and Restify.

---

<sup>6</sup> To use a Node application within a browser, you can use the `|browserify|` utility.

<sup>7</sup> See `|jQuery|`: jQuery makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, *jQuery* has changed the way that millions of people write JavaScript.

<sup>8</sup> See the sites `|altjs|` , `|async|`

<sup>9</sup> See for example `|Javascript frameworks|` and `|Comparison of JavaScript frameworks|`.

<sup>10</sup> See for example `|nodejs frameworks|`.

---

## 2.3 ES6

ECMAScript ([ES6](#), formally called ECMAScript2015) does introduce features that represent the foundation for modern JavaScript applications. Many JavaScript environments (in particular browsers and Node.js) are working on implementing ES6, with possible inconsistencies between implementations (see [ES6 compatibility table](#)). Please consult [Zakas's book: Understanding ECMAScript 6](#).

In these notes we will make reference to ES6 to show how it can help in solving problems that often JavaScript developers actually face.

## 2.4 Strict mode

ECMAScript 5's `strict` mode is a way to opt in to a restricted variant of JavaScript. Strict mode isn't just a subset: it intentionally has different semantics from normal code. In fact, strict mode makes several changes to normal JavaScript semantics:

- strict mode eliminates some JavaScript silent errors by changing them to throw errors;
- strict mode fixes mistakes that make it difficult for JavaScript engines to perform optimizations;
- strict mode prohibits some syntax likely to be defined in future versions of ECMAScript.

Thus, strict mode changes both syntax and runtime behaviour. Strict mode applies to entire scripts or to individual functions. It doesn't apply to block statements enclosed in braces; attempting to apply it to such contexts does nothing.

The normal (non-strict) mode is sometimes called "`sloppy mode`".

For the details, see [JavaScript Strict mode](#).

---

## 3 Basic concepts

Complexity in software can be made manageable by separating it into small compartments that are isolated from each other.

**Functions** are a basic concept and a basic tool in every modern programming language. They allow to structure larger programs, to reduce repetition, to associate names with subprograms, and to isolate these subprograms from each other. Moreover, they are useful to define new vocabulary.

An **Object** hides the complexity inside it and instead offers methods that present an interface through which the object is to be used. The idea is that the interface is relatively simple and all the complex things going on inside the object can be **ignored** when working with it. These ideas were initially worked out in the 1970s and 1980s and, in the 1990s, they lead to the object-oriented programming revolution.

The **Prototype** relationship of JavaScript between two objects is about inheritance: this relationship form a tree-shaped structure, and at the root of this structure sits **Object.prototype**.

### 3.1 Variables and scoping

With reference to *variables*, bindings rules<sup>11</sup> determine how a name is associated to a value inside a scope.

In computer programming, the *scope* of a name *binding* (an association of a name to an entity, such as a variable) is the part of a computer program where the binding is valid: where the name can be used to refer to the entity (see [Scope in computer science]).

In JavaScript, functions (and only functions<sup>12</sup>) introduce new scopes. A simulation of block scoping is usually done with a pattern named **IIFE**<sup>13</sup> (*Immediately Invoked Function Expression*) (see Subsection 7.1).

**3.1.1 Immediately Invoked Function Expression (IIFE).** Immediately-invoked function expressions (IIFE) is the name given to a pattern that defines an anonymous function and calls it immediately without saving a reference. This pattern allows us to create a scope that is shielded from the rest of a program.

```
1  /*
2  * =====
3  * iife.js
4  * =====
5  */
6  let entity = function(name) {
7      return {
8          getName: function() { return name; }
9      };
10 }("book");
11 console.log("iife=", entity.getName()); //iife=book
```

Listing 1.1. iife.js

### 3.2 Functions and hoisting

A JavaScript function is a block of code designed to perform a particular task.

JavaScript **hoists** all function and variable declarations<sup>14</sup>, i.e. it moves them to the beginning of their direct scope, since a variable can be declared after it has been used.

<sup>11</sup> *Binding* is the act of associating properties (values) with names. *Binding time* is the moment in the program's life cycle when this association occurs.

<sup>12</sup> ES6 does introduce block-level, lexical bindings with **let** and **const** declarations.

<sup>13</sup> pronounced 'iffy'

<sup>14</sup> JavaScript only hoists declarations, not initializations.

A variable defined inside a function cannot be seen outside the function. However, if we define a function inside a function, the inner function can see variables in the outer function.

Variables declared outside of any function are called **global**, because they are visible throughout the program. It is possible to access such variables from inside a function, as long as you haven't declared a local variable with the same name.

```
1 var n = 0;
2 var v = 0;
3 var f = function(n){
4   var v = 30;
5   console.log("f: n=" + n + " v=" + v + " k=" + k);
6 }
7 var k = 50; //hoisted
8 f(10);      //f: n=10 v=30 k=50
9 g(20);      //g: n=100 v=0 k=20
10 function g(k){ //function declaration: hoisted
11   var n = 100;
12   console.log("g: n=" + n + " v=" + v + " k=" + k)
13 }
```

A JavaScript function can play several roles:

- **normal function**: it can be invoked directly
- **method**: the function is stored as a property of an object (see Subsection 7.1)
- **constructor**: the function is invoked via the **new** operator (see Subsection 7.1)

### 3.3 Define and invoke functions

The example shows different ways to define a JavaScript function and different ways to invoke it.

```
1 /*
2  * A function that belongs to the global object
3  */
4 function toCelsius(fahrenheit) {
5   return (5/9) * (fahrenheit-32);
6 }
7 //INVOKE as a normal function
8 console.log("toCelsius(75)" , toCelsius(75));
9 /*
10 * A function defined with JavaScript built-in Function constructor
11 * Functions created with the Function constructor do not create closures to their creation contexts;
12 * they always are created in the global scope.
13 */
14 var mul = new Function("a", "b", "return a * b");
15 //INVOKE using Function
16 console.log("mul(2,3)=" , mul(2,3) );
17 /*
18 * A (anonymous) function defined using an expression.
19 * Functions defined using an expression are not hoisted.
20 */
21 var sum = function(a,b){ return a + b; } ;
22 //INVOKE as a normal function
23 console.log("sum(2,3)=" , sum(2,3) );
```

Listing 1.2. FunctionDefIntro.js

### 3.4 Function as constructor

The constructor **Function()** evaluates JavaScript code stored in strings (it works similarly to **eval**, but with a more clear interface):

```
1 var add = new Function('a','b','return a+b');
2 console.log("Using Function : " + add(2,3)); //Using Function : 5
```

However, this way of defining functions is slow: it is much better to introduce functions with a function expression (that produce a function object) or with function declaration.



---

### 3.5 The variable `this`

When you call a function, the variable `this` is always an implicit parameter. The following example does produce a very long output, that displays all the properties of the function:

```
1 function testThis(){ console.log( this ) } ;  
2 testThis();           //a very long output
```

However, the reference to `this` is always `null` in `strict mode` (see Subsection 2.4):

```
1 function testThisStrict(){ 'use strict'; console.log( this ) } ;  
2 testThisStrict(); //undefined
```

In case of methods, the value of `this` is called the *receiver* of the method call (see Subsection 7.1).

```
1 function retrunThisStrict(){ 'use strict'; return this } ;  
2 console.log("about this ...");  
3 console.log( retrunThisStrict() ); //undefined  
4  
5 var obj = { op : retrunThisStrict }  
6 console.log( obj.op() );           //{ op: [Function: retrunThisStrict] }  
7 console.log( obj.op() === obj ); //true
```

### 3.6 Objects

In JavaScript, all non-primitive values are objects. An object can be considered as a set of properties expressed as (`key,value`) pairs. The most common kinds of objects are:

- `plain objects`, created by object literals

```
1 var bob = { name : "Bob", age : 35 };
```

- `arrays`, created by *array literals*

```
1 var a = [ '1', 2, bob ]  
2 console.log( a[0] + " type=" + typeof a[0] ) ; //1 type=string  
3 console.log( a[1] + " type=" + typeof a[1] ) ; //2 type=number  
4 console.log( a[2] + " type=" + typeof a[2] ) ; //[object Object] type=object
```

- `regular expressions`, created by *regular expressions literals*;
- `functions`: each functions has its own methods. Among them the methods `call`, `apply` and `bind` help with calling a function. See Subsection 4.2 and Subsection 4.3.

Besides single objects, JavaScript provides *factories* for objects in form of `constructors`.

### 3.7 Methods

In JavaScript, methods can be introduced as properties that hold function values.

In the following example, we add a method to a literal object (`bob`) (see Subsection 4.6):

```
1 var bob = { name : "Bob", age : 35 };  
2 bob.speak = function ( msg ) { //method  
3   console.log ( "Entity " + this.name + " of age " + this.age + " says:" + msg ) ;  
4 };  
5  
6 bob.speak("hello"); //Entity Bob of age 35 says:hello
```

### 3.8 Constructors

A JavaScript function invoked with the **new** operator plays the role of a constructor of objects.

A basic pattern is to define a functions that sets up the instance data:

```
1 function Point(x,y){
2   this.x = x;
3   this.y = y;
4 }
5 var p = new Point(10,20);
6 console.log("p.x=" + p.x); //p.x=10
7 console.log("p.y=" + p.y); //p.y=20
8 console.log("p distance=" + p.distance() ); //TypeError: p.distance is not a function
```

The next step is to define operations that can be shared among all the instances:

```
1 Point.prototype.distance = function(){
2   return Math.sqrt( this.x*this.x + this.y*this.y );
3 }
4 var q = new Point(2,3);
5 console.log("q distance=" + q.distance() ); //q distance=3.605551275463989
6 console.log("p distance=" + p.distance() ); //p distance=22.360679774997898
```

### 3.9 Prototypes

The prototype relationship between two objects is about inheritance: every object can have another object as its prototype. Then the former object inherits all of its prototype's properties.

An object specifies its prototype via the internal property `[[Prototype]]`. Every object has this property, but it can be null. The chain of objects connected by the `[[Prototype]]` property is called the *prototype chain*.

The internal property `[[Prototype]]` is not accessible from JavaScript. The language provides functions for reading the prototype and for creating new objects with a given prototype.

```
1 var protoPerson = {
2   show : function(msg){ console.log("protoPerson " + this.name + " : " + msg); }
3 }
4
5 var jane = Object.create(protoPerson);
6 jane.name = "Jane";
7 jane.show("hello"); //protoPerson Jane:hello
8
9 var fred = Object.create( protoPerson, {name : {value:'Fred', writable:true} } );
10 fred.show("hello"); //protoPerson Fred:hello
11
12 console.log( Object.getPrototypeOf( fred ) ); //{ show: [Function: show] }
13 console.log( Object.getPrototypeOf( fred ) == protoPerson ); //true
14 console.log( Object.prototype.isPrototypeOf(fred) ); //true
15 console.log( protoPerson.isPrototypeOf(fred) ); //true
```

### 3.10 Inheritance

An object can be created with another object as its direct prototype. The **Prototype** relationship of JavaScript between two objects is about inheritance: this relationship forms a tree-shaped structure, and at the root of this structure sits `Object.prototype`.

```
1 var victor = Object.create(fred); //now fred is the prototype of victor
2
3 console.log( fred.isPrototypeOf(victor) ); //true
4 console.log( Object.prototype.isPrototypeOf(victor) ); //true
```

The object `victor` inherits the method `show` from `fred`, but it has no a specialized `name` property:

```
1 victor.show("hello"); //protoPerson Fred:hello
```

Let us set a specific name for `victor` and try again the method `show`:

```
1 victor.name = "Victor";
2 victor.show("hello"); //protoPerson Victor:hello
```

---

### 3.11 The property `__proto__`

The property `__proto__` is a special property for getting and setting the prototype of an object.

```
1 console.log( fred.__proto__ );    //{ show: [Function: show] }
```

Let us now change the prototype of victor (perhaps a not wise thing to do):

```
1 console.log( victor.__proto__ === fred ); //true
2
3 var protoStudent = {
4   id : 12345 ,
5   show : function(msg){ console.log("protoStudent " + this.id + " " + this.name + ":" + msg); }
6 }
7
8 victor.__proto__ = protoStudent ;
9 victor.show("hello"); //protoStudent 12345 Victor:hello
10 victor.id = 9876;
11 victor.show("hello"); //protoStudent 9876 Victor:hello
```

### 3.12 Higher order functions

A higher-order function is a function that can take another function as an argument, or that returns a function as a result.

Since JavaScript functions are *Objects*, they can be assigned as the value of a variable, and they can be passed and returned just like any other reference variable. Thus, JavaScript supports a very natural approach to functional programming; for example, we have already seen (Subsection 3.14) that a *callback* is a functions that is passed to another (higher-order) function as argument.

In the following example, the function `compose` (line 25) is a higher-order function that encapsulates function composition. In fact, given two functions `f` and `g`, it returns *another function* that computes `f(g(x))`. For example, at line 38 we call `compose(square,sine)(x)` in order to evaluate  $\sin^2(x)$ .

In this first call of `compose`, both the functions given as arguments are pure functions that return a number, given a number. Since JavaScript is not statically typed, we annotate the function type signature with a comment (borrowed from Haskell). For example:

- `square :: Number -> Number` says that the function accepts a number and returns a number;
- `squarePair :: Number -> ( Number, String )` says that the function accepts a number and returns a tuple containing a number and a string;

```
1  /*
2  * =====
3  * hofcompose.js
4  * =====
5  */
6  /*
7  * PURE functions of type signature:
8  *   functionName :: Number -> Number
9  */
10 var square = function(x) { return x * x };
11 var sine = function(x) { return Math.sin(x) };
12
13 /*
14 * PURE functions of type signature:
15 *   functionName :: Number -> ( Number,String )
16 */
17 var squarePair = function(x) { var v = x * x; return [ v, "square" ] };
18 var sinePair = function(x) { var v = Math.sin(x); return [ v, "sin" ] };
19
20 /*
21 * -----
22 * HIGH ORDER function to encapsulate function composition
23 * -----
24 */
25 var compose = function(f, g) {
```

```

26     return function(x) {
27         return f(g(x));
28     };
29 };
30
31 /*
32  * MAIN
33  */
34 console.log("square(0.5)=", square(0.5) );
35 console.log("squarePair(0.5)=", squarePair(0.5) );
36 console.log("sine(Math.PI/4)=", sine(Math.PI/4) );
37 console.log("sinePair(Math.PI/4)=", sinePair(Math.PI/4) );
38 console.log("COMPOSE_1=", compose(square,sine)( Math.PI/4 ) );
39 console.log("COMPOSE_2=", compose(squarePair,sine)( Math.PI/4 ) );
40 console.log("COMPOSE_3=", compose(squarePair,sinePair)( Math.PI/4 ) );

```

Listing 1.3. hofcompose.js

Note that:

- the functions `squarePair` and `sinePair` are pure functions that return also a `String` (the second item of the pair) useful for logging information. They are pure, since do not perform any side-effects (they do not use `console.log`);
- at line 39, we call `compose(squarePair,sine)(...)` in order to evaluate  $\sin^2(\pi/4)$ . This returns a pair with a number at the first place
- at line 40, we call `compose(squarePair,sinePair)(...)` in order to evaluate  $\sin^2(\pi/4)$ . This returns a pair with a `NaN`. A simple composition does not work here because the return type of `sinePair` (a pair) is not the same as the argument type required by `squarePair` (a number). We will return on this point in Section ??.

The results are:

```

1  /*
2  square(0.5)= 0.25
3  squarePair(0.5)= [ 0.25, 'square' ]
4  sine(Math.PI/4)= 0.7071067811865475
5  sinePair(Math.PI/4)= [ 0.7071067811865475, 'sin' ]
6  COMPOSE_1= 0.4999999999999999
7  COMPOSE_2= [ 0.4999999999999999, 'square' ]
8  COMPOSE_3= [ NaN, 'square' ]
9  */

```

Listing 1.4. hofcompose.js

### 3.13 Lexical closures

Lexical closures are usually associated with, languages that can treat functions as data (first class functions), since storing a closure for later use implies an extension of the lifetime of the closed-over lexical environment beyond what one would normally expect.

#### 3.13.1 Closure

Operationally, a closure is a record storing a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.

```

1  /*
2  * =====
3  * ClosureExample.js
4  * =====
5  */
6  function addClosure(x){
7      function incrementBy(y){
8          return x + y
9      }

```

---

```

10     return incrementBy
11 }
12
13 var c1 = addClosure(10) ;
14 var c2 = addClosure(20) ;
15
16 console.log( c1(1) );
17 console.log( c2(1) );

```

**Listing 1.5.** *Execution of ClosureExample.js*

The result is:

```

1  /*
2   11
3   21
4   */

```

**Listing 1.6.** *Execution of ClosureExample.js*

### 3.14 Callbacks

A callback is a function that is passed to another function (let's call this other function "otherFunction") as a parameter, and the callback function is called (or executed) inside the "otherFunction".

#### callback

A callback is a piece of executable code that is passed as an argument to other code, which is expected to call back (execute) the argument at some convenient time. The invocation may be immediate as in a *synchronous* callback or it might happen at later time, as in an *asynchronous* callback.

A callback function is essentially a pattern<sup>15</sup>, and therefore, the use of a callback function is also known as a *callback pattern*.

```

1  /*
2   * =====
3   * SynchCallback.js
4   * =====
5   */
6  var start = new Date;
7
8  function workAndCallback( callback ){
9      var n = 0;
10     while( n < 10 ){ n = new Date - start };
11     callback( n );
12 }
13
14 workAndCallback( function(k){ console.log("hello "+k) } ) //hello 10

```

**Listing 1.7.** *SynchCallback.js*

### 3.15 Continuation passing style (CPS)

The usage of callback leads to a programming style also known as *programming by continuation*. The reason is that the behaviour of a conventional function is split in two parts: an immediate part and a continuation part that will be encapsulated in a callback.

The idea behind continuation passing style (CPS) is:

- no function is allowed to return to it's caller
- each function takes a callback or continuation function as its last argument
- that continuation function is the last thing to be called

---

<sup>15</sup> The term 'pattern' is used here to denote an established solution to a common problem.

CPS works well for systems that are asynchronous by default (like Node) where the event loop constantly runs and the API functions as callbacks. Converting direct style programming to CPS, however, requires some work and some different thinking about how your processing takes place.

Let us consider for example a function (*workNormal*) that first calls another function (*evalValue*) and then computes some new value by using the result of *evalValue* and some local value:

```

1  /*
2  * =====
3  * Continuation.js
4  * =====
5  */
6  function evalValue(){
7      return 10;
8  }
9  function workNormal(){
10     var v0 = 5;
11     console.log( "v0=" + v0 ); //v0 = 5
12     var n = evalValue( );
13     //-----
14     var k = n + v0 ;
15     v0 = v0 -1;
16     console.log( "k normal=" + k + " v0= " + v0 );
17 }
18 workNormal(); //k normal=15 v0= 4

```

**Listing 1.8.** workNormal in Continuation.js

Let us define now a CPS function *workByContinuation* that performs the same work as the function *workNormal*. It first calls *evalValue* and then calls the callback given as input argument that will complete the work.

```

1  function workByContinuation( callback ){
2      var n = evalValue( );
3      callback( n );
4  }
5  continuation = function(n){
6      var k = n + v0 ;
7      v0 = v0 -1;
8      console.log( "k continuation=" + k + " v0= " + v0 );
9  }
10
11  var v0 = 5;
12  console.log( "v0=" + v0 ); //v0=4
13  workByContinuation(
14      function(n){ //callback
15          var k = n + v0 ;
16          v0 = v0 -1;
17          console.log( "k continuation=" + k + " v0= " + v0 ); //k continuation=15 v0= 4
18      }
19  );
20  console.log( "v0=" + v0 ); //v0=4

```

**Listing 1.9.** workByContinuation in Continuation.js

The caller defines the callback as an unnamed lexical closure that makes reference to the (global) variable *v0*.

In section Subsection ?? we will see techniques and constructs to overcome CPS.

---

## 4 A deeper view

### 4.1 The variable *arguments*

When we create a function in JavaScript, it creates a special variable called `arguments`, that is sort of an array that contains **all** the arguments passed in a function call, regardless of how many are defined.

The `arguments` variable has a `length` property, like an array, but it is not an array (it is not an instance of `Array`). If we want to use it as an actual array, we can convert the arguments variable into a real array by using a built-in array method called `Array.prototype.slice`.

```
1 function mul(x,y){
2   for( var i=0; i<arguments.length;i++){
3     console.log("add arg " + i + "=" + arguments[i]);
4   }
5   // arguments.forEach( //WRONG since arguments is not a true array
6   //   function(elem,index){console.log("add arg " + index + ")" + elem);} );
7   var aa = Array.prototype.slice.call(arguments);
8   aa.forEach(
9     function(elem,index){console.log("add arg " + index + ")" + elem);} );
10  return 2*3;
11 }
```

### 4.2 call and apply

The operations `apply` and `call` take a first argument that can be used to simulate method calls.

```
1 function speak( msg ){
2   console.log ("Speak running " + this.name + " says: " + msg );
3 }
4
5 speak("hello"); //Speak running undefined says: hello
6
7 speak.call( bob, "hello from call" ); //Speak running Bob says: hello from call
8 speak.apply( bob, ["hello from apply"]); //Speak running Bob says: hello from apply
```

The first argument to the `call` method defines what the special variable `this` refers to inside the function. Any arguments after this one are passed directly to the function.

The `apply` method is much like `call`, except that instead of passing individual arguments one-by-one, `apply` allows us to pass an *array of arguments* as the second parameter; this is great for **variadic functions**<sup>16</sup>

`call()` and `apply()` are predefined JavaScript `Function` methods.

### 4.3 Function.prototype.bind()

The `bind` method creates a new *bound function* (BF). A BF is an exotic function object (term from ECMAScript 2015) that wraps the original function object. Calling a BF generally results in the execution of its wrapped function.

```
1 speak.bind(bob)("hello from bind"); //Speak running Bob says: hello from bind
```

---

<sup>16</sup> A variadic function takes varying number of arguments.

The `bind()` function creates a new bound function (BF). A BF is an exotic function object (term from ECMAScript 2015) that wraps the original function object. Calling a BF generally results in the execution of its wrapped function. A BF has the following internal properties:

- `[[BoundTargetFunction]]`: the wrapped function object;
- `[[BoundThis]]`: the value that is always passed as the `this` value when calling the wrapped function.
- `[[BoundArguments]]`: a list of values whose elements are used as the first arguments to any call to the wrapped function.
- `[[Call]]`: executes code associated with this object. Invoked via a function call expression. The arguments to the internal method are a `this` value and a list containing the arguments passed to the function by a call expression.

#### 4.3.1 bind

When bound function is called, it calls internal method `[[Call]]`<sup>a</sup> with following arguments `Call(target, boundThis, args)`. Where, `target` is `[[BoundTargetFunction]]`, `boundThis` is `[[BoundThis]]`, `args` is `[[BoundArguments]]`. A bound function may also be constructed using the `new` operator: doing so acts as though the target function had instead been constructed. The provided `this` value is ignored, while prepended arguments are provided to the emulated function.

<sup>a</sup> Internal properties define the behavior of code as it executes but are not accessible via code. ECMAScript defines many internal properties for objects in JavaScript. Internal properties are indicated by double-square-bracket notation `[[prop]]`.

#### 4.4 Methods as callbacks

A common mistake for new JavaScript programmers is to extract a method from an object, then to later call that function (e.g. by using that method in callback-based code) and expect it to use the original object as its `this`. Without special care however, the original object is usually lost. Creating a bound function from the function, using the original object, neatly solves this problem.

Let us show an example:

```
1  /*
2  * =====
3  * bind.js
4  * =====
5  */
6  x = 9;    //GLOBAL VARIABLE
7  /*
8  * Literal object Point
9  */
10 var Point = {
11     x : 81,
12     getX : function() { return x; },
13     getMyX : function() { return this.x; }
14 };
15 //New programmers might confuse the global x with module's property x
16 console.log( "Point.getX()=" + Point.getX() + " Point.getMyX()=" + Point.getMyX() );
17
18 var retrieveX = Point.getMyX;
19 console.log( "retrieveX=" + retrieveX() ); //9: The function gets invoked at the global scope
20
21 // Create a new function with 'this' bound to Point
22 var boundGetX = retrieveX.bind(Point);
23 console.log( "boundGetX=" + boundGetX() ); // 81: call a new function with 'this' bound to Point
24
25 /*
```

Listing 1.10. bind.js

The output is:

```
1  /*
2  Point.getX()=9 Point.getMyX()=81
3  retrieveX=9
4  boundGetX=81
5  */
```

Listing 1.11. bind.js result



---

## 4.5 Arrow functions

Arrow functions are introduced in ES6 (see [exploringjs.com](http://exploringjs.com)) as functions defined with a new syntax that uses an 'arrow' (`=>`) with the following properties:

- Cannot be called with *new*. Thus: No prototype
- No *arguments* object
- The value of *this*, *super*, *arguments*, and *new.target* inside of the function is by the closest containing no-narrow function.
- Can't change *this*

Arrow functions are designed to be "throwaway" functions, and so cannot be used to define new types.

```
1  /*
2  * =====
3  * arrowfun.js
4  * =====
5  */
6  //No argument
7  let noArg = () => "hello from noArg"
8  console.log("noArg:" , noArg() ) //noArg: hello from noArg
9
10 //Single argument
11 let oneArg = v => v
12 console.log("oneArg:" , oneArg(3) ) //oneArg: 3
13
14 //More arguments
15 let moreArgs = (v1, v2) => v1 + v2
16 console.log("moreArgs:" , moreArgs(2,3) ) //moreArgs: 5
17
18 //With body
19 let withBody = (v1, v2) => { return v1 - v2; } //return must explicit
20 console.log("withBody:" , withBody(4,3) ) //withBody: 1
21
22 //IIFE
23 let thing = ( (name) => { //note the sourrounding ( )
24   return {
25     getName: function() { return name; }
26   };
27 } )("book");
28 console.log("iife=", thing.getName()); //iife= book
```

Listing 1.12. Examples of arrow functions

## 4.6 About this

The `this` keyword behaves differently in JavaScript compared to Object Oriented languages, where `this` refers to the current instance of a class. The ECMAScript Standard defines:

<b>this</b>	a keyword that evaluates to the value of the <i>ThisBinding</i> of the current execution context" (§11.1.1). <i>ThisBinding</i> is something that the JavaScript interpreter maintains as it evaluates JavaScript code, like a special CPU register which holds a reference to an object.
-------------	---

In JavaScript the value of `this` is determined mostly by the *invocation context* of function and where it is called. In fact, evaluating a JavaScript function establishes a distinct execution context that appends its local *scope* to the scope chain it was defined within. JavaScript resolves identifiers within a particular context by climbing up the scope chain, moving locally to globally

Thus, in JavaScript, `this` is set by how a function is called, not by where it is defined. More specifically, from [JavaScript Reference] we read:

**this**

- In the global execution context (outside of any function), **this** refers to the global object, whether in *strict* mode or not.
- Inside a function, the value of **this** depends on how the function is called.
  - Simple call
  - With **call** and **apply** (see Subsection 4.2)
  - The **bind** method (see Subsection 4.3)
  - ES6 Arrow functions (see Subsection 4.5)

In particular:

- When a function is called as a *method* of an object, its **this** is set to the object the method is called on (see Subsection 3.7).
- When a function is used as a *constructor* (with the **new** keyword), its **this** is bound to the new object being constructed.
- When a function is used as an *event handler*, its event handler is set to the element the event fired from (some browsers do not follow this convention for listeners added dynamically with methods other than *addEventListener*).
- When code is called from an *in-line on-event handler*, its in-line *on-event* handler is set to the DOM element on which the listener is placed.

A JavaScript function is a block of code designed to perform a particular task; it is executed when "something" invokes it. JavaScript function can be invoked in different ways.

- Simple call
- With **call** and **apply** (see Subsection 4.2)
- With the **bind** method (see Subsection 4.3)
- ES6 Arrow functions (see )

The value of the variable **this** is determined mostly by the *invocation context* of function and where it is called.

Let us consider the case of a method of an object (literal):

```
1 age = 52;
2 person = {
3     name : "Bob",
4     age : 35,
5     getSomeAge : function(){
6         return age;    //refers to the global age
7     },
8     getName : function(){
9         console.log("person name=" + this.name)
10        return this.name;
11    },
12    getAge : function(){
13        console.log("person age=" + this.age)
14        return this.age;
15    }
16 }
17 console.log("person.age=" + person.age);           //35
18 console.log("person.getName()" + person.getName()); //Bob
19 console.log("person.getSomeAge()" + person.getSomeAge()); //52
```

**Listing 1.13.** **this** in a object

When `person.getName()` is executed, JavaScript establishes an execution context for the function call, setting **this** to the object referenced by whatever came before the last `.`, in this case: **person**. The method can then look in the mirror via **this** to examine its own properties, returning the value stored in `this.name`.

As another example, let us use *this* within a constructor function:

---

```

1 function student( name, age ){
2     this.name = name;
3     this.age = age;
4     this.getName = function(){ return this.name; }
5 }
6 alice = new student( "alice", 22 );
7 console.log("method call=" + alice.getName()); //method call=alice

```

Listing 1.14. `this` in a constructor

## 4.7 About `this` in Node

From [meaning-of-this-in-node-js-modules-and-functions](#) we read:

**this**

- In the top-level code in a Node module, `this` is equivalent to `module.exports`.
- When you use `this` inside of a function, the value of `this` is determined anew before each and every execution of the function, and its value is determined by how the function is executed. This means that two invocations of the exact same function object could have different `this` values if the invocation mechanisms are different (e.g. `aFunction()` vs. `aFunction.call(newThis)` vs. `emitter.addListener("someEvent", aFunction)`), etc.).
- When JavaScript files are required as Node modules, the Node engine runs the module code inside of a wrapper function. That module-wrapping function is invoked with a `this` set to `module.exports`. (Recall, above, a function may be run with an arbitrary `this` value.)

## 4.8 Tail recursion and Trampoline

JavaScript (before ES6) does not implement tail call optimization<sup>17</sup>. Thus, one obvious way is to get rid of recursion, and rewrite the code to be iterative. An alternative is to figure out a way to turn regular recursion into an optimized version that will execute without growing the stack.

The *trampoline* is a technique to optimize recursion and prevent stack-overflow exceptions in languages that don't support tail call optimization.

**trampoline**

A trampoline is a loop that iteratively invokes thunk-returning functions (continuation-passing style). A single trampoline is sufficient to express all control transfers of a program; a program so expressed is trampolined, or in trampolined style; converting a program to trampolined style is trampolining. Trampolined functions can be used to implement tail-recursive function calls in stack-oriented programming languages.

*Trampolining* is common in functional programming and provides us a way to call a function in tail position without growing the stack. Instead of executing directly the recursive steps, we will utilize higher order functions to return a wrapper 'thunk' function (a bit more complex code is required here) instead of executing the recursive step directly, and let another function control the execution.

In the following example we introduce:

- A conventional recursive implementation of the factorial (function *fact* at line 8)
- A tail-recursive implementation of the factorial (function *factTail* at line 30)
- An implementation of the *trampoline* function (line 48)
- A trampolined implementation of the factorial (function *factTrampolined!* at line 57)

```

1  /*
2  * =====
3  * Trampoline.js
4  * =====
5  */
6  /*
7  * Fact recursive
8  */

```

<sup>17</sup> ES6 will probably have support for tail call optimization.

```

9  fact = function( n ){
10     if( n==0) return 1;
11     return n * fact(n-1);
12 }
13 console.log("fact(3)=" , fact(3) );           //6
14 /*
15 (fact 3)
16 (* 3 (fact 2))
17 (* 3 (* 2 (fact 1)))
18 (* 3 (* 2 (* 1 (fact 0))))
19 (* 3 (* 2 (* 1 1)))
20 (* 3 (* 2 1))
21 (* 3 2)
22 6
23 Here is a visualization of the stack where each vertical dash is a stack frame:
24     ---|---
25     ---|   |---
26     ---|       |---
27     ---|           ---
28 */
29 //console.log("fact(15711)=" , fact(15711) ); //RangeError: Maximum call stack size exceeded
30
31 factTail = function( n ){
32     var _factTail = function myself (acc, n) {
33         return n ? myself(acc * n, n - 1) : acc
34     };
35     return _factTail(1, n);
36 }
37 console.log("factTail(3)=" , factTail(3) );     //6
38 /*
39 factTail(3)
40 _factTail(1, 3)
41 _factTail(3, 2)
42 _factTail(6, 1)
43 _factTail(6, 0)
44 6
45
46 The interpreter could reuse the activation record of _factTail
47 */
48 //console.log("factTail(15711)=" , factTail(15711) ); //RangeError: Maximum call stack size exceeded
49
50 function trampoline(fn){
51     var args = [].slice.call(arguments, 1);
52     var res = fn.apply(this, args);
53     while(res instanceof Function){
54         res = res();
55     }
56     return res;
57 }
58
59 function factTrapolined(n) {
60     var _fact = function myself (acc, n) {
61         return n ? function () { return _fact(acc*n, n-1); } : acc
62     };
63     return trampoline( _fact, 1, n );
64 }
65 console.log("factTrapolined(3)=" , factTrapolined(3) ); //6
66 /*
67 You can visualize the stack like a bouncing trampoline:
68     ---|--- ---|--- ---|---
69     ---    ---    ---
70
71 */
72 console.log("factTrapolined(15711)=" , factTrapolined(15711) ); //Infinity

```

Listing 1.15. Trampoline.js

Note that the call `factTrapolined(15711)` returns a value (*Infinity*) while a similar call for `fact` (line 29) and `factTail` (line 48) raises the exception: *Maximum call stack size exceeded*.

---

In the following example, we use the module `big-integer`<sup>18</sup> in order to avoid the Infinity result. Moreover, we use (line 21) the *bind* operation (see Subsection 4.3) to return a function properly bound to the correct context.

```
1  /*
2  * =====
3  * TrampolineBigdata.js
4  * =====
5  */
6  //npm install big-integer
7  var bigInt = require("big-integer");
8
9  function trampoline(fn){
10     var args = [].slice.call(arguments, 1);
11     var res = fn.apply(this, args);
12     while(res instanceof Function){
13         res = res();
14     }
15     return res;
16 }
17
18 function factorial(n) {
19     function _factorial (n,acc) {
20         acc || (acc=bigInt(1) ) //pattern to set the optional argument acc
21         return n.greater(0) ? _factorial.bind(this, n.minus(1), acc.times(n) ) : acc
22     };
23     return trampoline( _factorial, bigInt(n) );
24 }
25 console.log("factorial(5)=" , factorial(5) ); //{ [Number: 120] value: 120, sign: false, isSmall: true }
26
27 console.log("factorial(15711)=" , factorial(15711).toString(base=10) ); //a very big number ...
```

**Listing 1.16.** TrampolineBigdata.js

---

<sup>18</sup> The function `require` is defined by NodeJs only: see Subsection 6.2

---

## 5 Asynchronous Operations

A JavaScript function is a block of code designed to perform a particular task. When we execute a task *synchronously*, we wait for it to finish before moving on to another task. When we execute something *asynchronously*, we can move on to another task before it finishes.

Many applications require to call functions asynchronously because this enables the application to continue doing useful work while the function runs. Technically, the concept of synchronous/asynchronous does not have anything to do with *threads*. The concept of synchronous/asynchronous has to do solely with whether or not a second or subsequent task can be initiated before the other (first) task has completed, or whether it must wait. Although, in general, it would be unusual to find asynchronous tasks running on the same thread, it is possible to find two or more tasks executing synchronously on separate threads.

However, JavaScript executes user-defined functions by using a **single-threaded event loop**.<sup>19</sup>

### 5.1 event loop

- The *event loop* is an activity within the JavaScript engine (VM) that monitors code execution and manages the so called event-queue.
- The *job-scheduling* is the action that adds a new task/job to job-queue.
- The *event-queue* (or better *job-queue*) is data structure in which the engine stores the code ready to run.
- The code ready to run is usually represented as a *callback* (see Subsection 3.14) that can represent the response to an *event*.
- Events are a part of the *Document Object Model* (DOM) Level 3 and every HTML element contains a set of events which can trigger JavaScript Code. In this context, an *event* is a signal from the browser that something has happened.

For example, when a user clicks a button, an event (`onClick`) is triggered and a new job is put into the job-queue. This code is not executed immediately, but when all the other jobs ahead of it in the queue are complete. That's why web pages that use JavaScript imprudently tend to become unresponsive.

This is the most basic source of asynchronous programming in JavaScript and each JavaScript environment comes with its own set of *asynchronous functions*, that fall into two main categories: I/O and timing. But JavaScript (unlike for example Erlang) has no syntactic way of dealing with asynchronous code and the software designer must use callbacks.

Moreover, JavaScript (before ES6, ES7) does not allow us to define truly custom asynchronous function<sup>20</sup>; we have to leverage on a technology provided natively, such as `setTimeout` or `setInterval` that can be found in any JavaScript environment,

### 5.2 setTimeout

The `setTimeout` operation calls a function or evaluates an expression after a specified number of milliseconds, by returning immediately the control to caller. Thus, `setTimeout` allows us to specify a delay before a task/job is added to the *event-queue*.

**setTimeout**

The function  
`long setTimeout(function f, unsigned long timeout, any args...)`  
registers `f` to be invoked after `timeout` milliseconds have been elapsed and returns a number that can later be passed to `clearTimeout()` to cancel the pending invocation. When the specified time has passed, `f` will be invoked and will be passed any specified args. If `f` is a string rather than a function, it will be executed after `timeout` milliseconds as if it were a `<script>`.

<sup>19</sup> WebWorkers do introduce multi-threading in JavaScript but without shared state and with message-based interaction based on callbacks that are run from the event queue, like a conventional I/O.

<sup>20</sup> ES6 does introduce the *async function* declaration, which returns an *AsyncFunction* object

When we call `setTimeout`, a *timeout event* is queued. Then execution continues until there current task terminates. At this point, the JavaScript virtual machine looks at the event-queue. If there's at least one event on the queue that's eligible to "fire", the JavaScript engine VM will pick one and call its handler. When the handler returns, the VM goes back to the queue.

**5.2.1 setTimeout at work** In the example that follows, we introduce an undefined global variable (`data`) and two functions: *i)* `setData`, to set (after some delay) a value to `data` and *ii)* `showData`, to show the value of `data`. The function *sequence* is introduced to call the previous functions in sequence and to show that the result is `undefined` since the `setData` function has not been executed when `showData` runs.

```

1  /*
2  * =====
3  * SetTimeoutBasic.js
4  * =====
5  */
6  var data; // global variable
7
8  showData = function() {
9      console.log(data);
10 };
11 var setData = function() {
12     setTimeout(function() {
13         data = 10;
14     }, 200);
15 };
16 function sequence() {
17     setData();
18     showData(); // here data is still undefined
19 }

```

**Listing 1.17.** A sequence call in *SetTimeoutBasic.js*

Let us introduce now two other asynchronous functions in CPS style (see Subsection 3.15): *i)* `setDataVal` to set a value to `data` and *ii)* `getDataVal`, to get the value of `data`. The function *sequenceCPS* calls the previous functions in CPS sequence.

```

1  /*
2  * CPS style
3  */
4  setDataVal = function(v,dt,callback) {
5      setTimeout(function() {
6          data = v;
7          callback(data);
8      }, dt);
9  };
10
11 getDataVal = function(dt, callback) {
12     setTimeout(function() {
13         callback("getVal done=", data)
14     }, dt);
15 }
16
17 function sequenceCPS() {
18     var dt = 100;
19     setDataVal(10, dt, showData );
20     setDataVal(20, dt, function(){ showData(); } );
21     setDataVal(30, dt, function(){ getDataVal( 0, console.log); } );
22 }

```

**Listing 1.18.** A sequence CPS call in *SetTimeoutBasic.js*

Finally we introduce a *main* function<sup>21</sup>.

<sup>21</sup> We use here the Node `process` object - see [|process.html|](#) - to avoid the call of *main* if the current program is not our file.

---

```

1 function main(){
2   console.log("data1=", data);
3   sequence(); // shows undefined
4   console.log("data2=", data);
5   sequenceCPS(); // will shows 10 20 30
6   console.log("data3=", data);
7 }
8
9 //Conditional call to main
10 if( process.argv[1].toString().includes("SetTimeoutBasic") ) main();

```

**Listing 1.19.** A main for *SetTimeoutBasic.js*

The global result is:

```

1 /*
2 data1= undefined
3 undefined
4 data2= undefined
5 data3= undefined
6 10
7 20
8 getVal done= 30
9 */

```

**Listing 1.20.** Execution of *SetTimeoutBasic.js*

The function *sequenceCPS* produces the lines 10, 20 getVal done= 30 according to the sequence of activation (i.e. insertion in the event-queue) since we use the same *dt*.

**5.2.2 setTimeout and scope** In the next example, we delay the execution of functions that make reference to variables with different scope:

- at line 8, the function body makes reference to the loop variable *i* and not the actual value at the moment inside each loop. Thus, the result of the function, when called, is always 4
- at line 13, we pass the actual value of the for loop variable *i* at the moment of each loop execution
- at lines 18-21, we adopt the IIFE pattern to obtain the same result as the previous case

The loop at line 32 is introduced to show that event handlers don't run until the single thread is free.

```

1 /*
2 * =====
3 * SetTimeoutExample.js
4 * =====
5 */
6 //here we pass the reference to the variable i, and not the actual value at the moment inside each loop
7 for (var i = 1; i <= 3; i++) {
8   setTimeout(function(){ console.log("i should be 4:",i); }, 5*i);
9 };
10
11 //here we pass the actual value of i at the moment of each loop execution in the for statement
12 for (var i = 1; i <= 3; i++) {
13   setTimeout(function(x) { return function() { console.log("x=", x); }; }(i), 20*i);
14 };
15
16 //here we adopt the Immediately Invoked Function Expression (IIFE) pattern
17 for (var i = 1; i <= 3; i++) {
18   (function(){
19     var k = i;
20     setTimeout( function(){ console.log("IIFE k=",k); }, 100*i);
21   })();
22 };
23
24
25 var start = new Date;
26 setTimeout(function(){
27   var end = new Date;

```



---

```

28     console.log('Time elapsed:', end - start, 'ms');
29   }, 500);
30
31   //here we keep the control for 800 msec
32   while( new Date - start < 800) { };

```

**Listing 1.21.** *SetTimeoutExample.js*

The result is:

```

1  /*
2  i should be 4: 4
3  i should be 4: 4
4  i should be 4: 4
5  x= 1
6  x= 2
7  x= 3
8  IIFE k= 1
9  IIFE k= 2
10 IIFE k= 3
11 Time elapsed: 3031 ms
12 */

```

**Listing 1.22.** *Execution of SetTimeoutExample.js*

**5.2.3 setTimeout and bind** Another example using the *bind* operation (see Subsection 4.3) for a Counter object:

```

1  /*
2  * =====
3  * setTimeoutBind.js
4  * =====
5  */
6  var val = 100;
7
8  function Counter() {
9     this.val = 0;
10    this.inc = function(){ this.val += 1; }
11  }
12
13  Counter.prototype.getVal = function() {
14     setTimeout(this.show.bind(this), 500);
15  };
16
17  Counter.prototype.show = function() {
18     console.log('I am a counter with val= ' + this.val );
19     this.inc();
20  };
21
22  var count = new Counter();
23  count.getVal();
24  count.getVal();

```

**Listing 1.23.** *SetTimeoutBind.js*

### 5.3 Calling a function (pseudo-)asynchronously

The *setTimeout* operation allows us to start a function and continue on our way without waiting for that function to return. A callback describes what to do after the asynchronous function call has completed.

In the following example, we introduce the function *factIterAsynch* that computes the factorial of a given number without monopolising the control of the single JavaScript Thread.

```

1  /*
2  * =====
3  * FactAsynch.js

```

```

4  * =====
5  */
6  factAsynch = function( n, callback ){
7      factIterAsynch(n,n,1,callback);
8  }
9  factIterAsynch = function( n, n0, v, callback ){
10     var res = n*v; //ACCUMULATOR
11     console.log( "factIterAsynch n0=" + n0 + " n=" + n, " v=" + v + " res=" + res);
12     if( n == 1 ) callback( "factIterAsynch(" + n0 + ") RESULT="+res );
13     else setTimeout( function(){ factIterAsynch(n-1, n0, res, callback) ; } , 0 );
14 }
15
16 console.log("START");
17 console.log("CALL= ", factAsynch(4, console.log) );
18 factAsynch(6,console.log);
19 console.log("END");

```

**Listing 1.24.** *FactAsynch.js*

The output shows the interleaved behaviour of two calls to the helper function *factAsynch* for *n*=4 and *n*=6.

```

1  /*
2  START
3  factIterAsynch n0=4 n=4 v=1 res=4
4  CALL= undefined
5  factIterAsynch n0=6 n=6 v=1 res=6
6  END
7  factIterAsynch n0=4 n=3 v=4 res=12
8  factIterAsynch n0=6 n=5 v=6 res=30
9  factIterAsynch n0=4 n=2 v=12 res=24
10 factIterAsynch n0=6 n=4 v=30 res=120
11 factIterAsynch n0=4 n=1 v=24 res=24
12 factIterAsynch(4) RESULT=24
13 factIterAsynch n0=6 n=3 v=120 res=360
14 factIterAsynch n0=6 n=2 v=360 res=720
15 factIterAsynch n0=6 n=1 v=720 res=720
16 factIterAsynch(6) RESULT=720
17 */

```

**Listing 1.25.** *Execution of FactAsynch.js*

## 5.4 The callback hell

*Callback Hell* is the name given to what happens when we want to do a bunch of sequential things using callbacks and asynchronous functions (see [|callbackhell.com|](http://callbackhell.com)).

In the following example, we define a function (A) that gives a value to a (undefined) global variable in asynchronous way. Our intent is to increment that value with another function (B) that must be called only when A is terminated. The functions C and **show** are introduced to show the current value as a possible last action (C) or as an action that should performed between other actions (**show**). Finally we introduce a *main* function, using the Node **process** object<sup>22</sup> to avoid the call of *main* if the current program is not our file.

```

1  /*
2  * =====
3  * sequencingHell.js
4  * =====
5  */
6  var value; //GLOBAL
7  /*
8  * A sets value asynchronously
9  */
10 var A = function(callback) {
11     setTimeout(function() {

```

<sup>22</sup> See [|process.html|](http://process.html).

---

```

12     console.log("A (before) value=", value );
13     value = 10;
14     callback();
15     }, 200);
16 };
17 /*
18  * B should run after A
19  */
20 var B = function(callback) {
21     value = value + 100;
22     console.log("B value="+value );
23     callback();
24 };
25 /*
26  * C shows value without continuations
27  */
28 var C = function() {
29     console.log("C value=" , value );
30 };
31 /*
32  * show shows value with continuations
33  */
34 var show = function(callback) {
35     console.log("show value=", value );
36     callback();
37 };
38
39 /*
40  * MAIN
41  */
42 function main(){
43     A( function(){
44         B( function(){ C();}
45     );
46 });
47
48     A( function(){
49         show( function(){
50             B( function(){ C();}
51         );
52     );
53 })
54 }
55
56 //Conditional call to main
57 if( process.argv[1].toString().includes("sequencingHell") ) main();

```

**Listing 1.26.** sequencingHell.js

The result produce by *main* is:

```

1  /*
2  A (before) value= undefined
3  B value=110
4  C value= 110
5  A (before) value= 110
6  show value= 10
7  B value=110
8  C value= 110
9  */

```

**Listing 1.27.** sequencingHell.js

The main problem is how to return to a conventional design process, without entering in the hell of "programming by continuation" (see Subsection 3.15) <sup>23</sup>

---

<sup>23</sup> Proper libraries, like *Async.js* (<https://github.com/caolan/async>), are introduced to avoid most "callback hell" scenarios in a Node code.

---

## 6 Node.js

This section presents some example and exercise to better understand the virtues of callback/continuation/asynchronous programming and the difficulties related to a proper design and understanding of the code.

### 6.1 Standard input and count down

Our first example is a simple application in which a count-down counter is stopped before it reaches the value 0, as soon as some information is read from the standard input device.

```
1  /*
2  * =====
3  * stdinExample.js
4  * =====
5  */
6  // prepare for input from terminal
7  process.stdin.resume();
8  // when receive data do ...
9  process.stdin.on('data', function (data) {
10     //console.log("input=" + data);
11     goon = false;
12 });
13
14 /*
15 * Count down
16 */
17 var v0 = 10;
18 var goon = true;
19
20 var count = function(){
21     if( v0 > 0 && goon ){
22         v0 = v0 - 1;
23         console.log("v0=" + v0);
24         setTimeout( count, 1000 );
25     }else if( ! goon ) console.log("counte down stopped");
26 };
27 //main
28 console.log("START with v0=" + v0 );
29 setTimeout( count, 1000 );
30 console.log("END" );
```

Listing 1.28. stdinExample.js

The count function is repeated every second, while input data are acquire in 'reactive way' when the user press CR. Since there is only one execution thread, the input is 'perceived' only if no other 'task' is running.

### 6.2 Node module.exports and require

The module.exports or exports is a special object which is included in every js file in the Node.js application by default. module is a variable that represents current module and exports is an object that will be exposed as a module. So, whatever you assign to module.exports or exports, will be exposed as a module.

```
1  /*
2  * =====
3  * NodeModuleIntro.js
4  * =====
5  */
6  //var exports = module.exports = {}; //default
7
8  console.log("initial this=" , this);
9  //console.log("initial module" , module); //shows all module info
```

```

10 console.log("initial exports=" , exports);
11 console.log("initial module.exports=" , module.exports);
12 exports.SimpleMessage = 'Hello world';
13 console.log("current exports=" , exports);
14
15 /*
16 initial this= {}
17 initial exports= {}
18 initial module.exports= {}
19 current exports= { SimpleMessage: 'Hello world' }
20 */

```

**Listing 1.29.** `NodeModuleIntro.js`

The function `require` is a synchronous operation<sup>24</sup> that locate a module, given its path, and loads its code into the program.

### 6.3 Browserify

Browserify allows us to use NodeJs style modules in the browser. We define dependencies and then Browserify bundles it all up into a single neat and tidy JavaScript file. You include your required JavaScript files using `require('./yourfancyJSfile.js')` statements and can also import publicly available modules from `npm`. It's also quite simple for Browserify to generate source maps, so that we can debug each js file individually, despite the fact it's all joined into one.

By default, browserify doesn't let us access the modules from outside of the browserified code. If we want to call code in a browserified module, we have to browserify our code together with the module. However, we can explicitly make our operation accessible from outside like this:

```

1 window.op =function(){
2   ...;
3 };

```

Then we can call `op()` in the page.

### 6.4 Event emitters and Event listeners

Much of the *Node.js* core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") periodically emit named events that cause Function objects ("listeners") to be called. Thus, underneath the surface of many of the Node core objects there is the `EventEmitter` object.

In fact, all objects that emit events are instances of the `EventEmitter` class. These objects expose an `eventEmitter.on()` function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

We have already introduced an exemple of user-defined event emitter

### 6.5 `process.nextTick`

The operation `process.nextTick()` defers the execution of an action till the next pass around the event loop.

```

1 /*
2  * =====
3  * nextTickExample.js
4  * =====
5  */

```

<sup>24</sup> It should be used with care in asynchronous applications, like an HTTP server (see Subsection 6.9).

```

6 f = function (data, callback) {
7   process.nextTick( function(){
8     callback(data) ;
9   } );
10 };
11 //main
12 console.log("START" );
13 f(3, function(v){console.log('OUTPUT=' + v);} );
14 f(5, function(v){console.log('OUTPUT=' + v);} );
15 console.log("END" );
16
17 /*
18 START
19 END
20 OUTPUT=3
21 OUTPUT=5
22 */

```

Listing 1.30. `nextTickExample.js`

### nextTick

The method `process.nextTick` attaches a callback function that's fired during the next tick (loop) in the Node event loop. You would use `process.nextTick` if you wanted to delay a function for some reason, but you wanted to delay it asynchronously. A good example would be if you're creating a new function that has a callback function as a parameter and you want to ensure that the callback is truly asynchronous.

Callbacks passed to `process.nextTick` will usually be called at the end of the current flow of execution, and are thus approximately as fast as calling a function synchronously. Use `process.nextTick` to effectively queue the function at the head of the event queue so that it executes immediately after the current function completes.

## 6.6 setImmediate

Callbacks passed to `setImmediate`s are queued in the order created, and are popped off the queue once per loop iteration.

### setImmediate

`setImmediate(callback, [arg], [...])` can be used to schedule the "immediate" execution of callback after I/O events callbacks and before `setTimeout` and `setInterval`. Returns an immediateObject for possible use with `clearImmediate()`. Optionally you can also pass arguments to the callback. Immediate objects are queued in the order created, and are popped off the queue once per loop iteration. This is different from `process.nextTick` which will execute `process.maxTickDepth` queued callbacks per iteration. `setImmediate` will yield to the event loop after firing a queued callback to make sure I/O is not being starved. While order is preserved for execution, other I/O events may fire between any two scheduled immediate callbacks.

```

1  /*
2  * =====
3  * setImmediateExample.js
4  * =====
5  */
6  f = function (data, callback) {
7    process.nextTick( function(){
8      callback(data) ;
9    } );
10 };
11 //main
12 console.log("START" );
13 setImmediate( function(){console.log('IMMEDIATE1' );} );
14 f(3, function(v){console.log('OUTPUT=' + v);} );
15 setImmediate( function(){console.log('IMMEDIATE2' );} );
16 f(5, function(v){console.log('OUTPUT=' + v);} );
17 setImmediate( function(){console.log('IMMEDIATE3' );} );
18 console.log("END" );
19
20 /*
21 START
22 END
23 OUTPUT=3
24 OUTPUT=5

```

---

```

25  IMMEDIATE1
26  IMMEDIATE2
27  IMMEDIATE3
28  */

```

**Listing 1.31.** `setImmediateExample.js`

When we are trying to break up a long running, CPU-bound job using recursion, we should use *setImmediate* rather than *process.nextTick* to queue the next iteration as otherwise any I/O event callbacks wouldn't get the chance to run between iterations.

## 6.7 The fs library

Accessing a file is a time-consuming operation, and a single-threaded application accessed by multiple clients that blocked on file access would soon bog down and be unusable. Let us suppose to have the following Json file:

```

1  { "name": "John", "age": 30, "city": "New York" }
2  { "name": "Alice", "age": 24, "city": "Los Angeles" }

```

**Listing 1.32.** `a.json`

The `fs` library of Node provides *asynchronous* read/write operations.

```

1  /*
2  * =====
3  * ReaderWithNode.js
4  * =====
5  */
6  var fs = require('fs');
7
8  readFileCallback = function(err, data){
9      if (err) console.log("error " + err);
10     else console.log( data );
11 }
12
13 try{
14     console.log("START" );
15     fs.readFile('./a.json', 'utf8', readFileCallback);
16     console.log("END" );
17 }catch(err){
18     console.log("error " + err);
19 }
20
21 /*
22 START
23 END
24 { "name": "John", "age": 30, "city": "New York" }
25 { "name": "Alice", "age": 24, "city": "Los Angeles" }
26 */

```

**Listing 1.33.** `ReaderWithNode.js`

The output shows that the reading is asynchronous. Once completed, the *readFile* operation calls the given *readFileCallback* that will continue the application with all the file content available in the `data` argument of the callback. If we want to continue the application logic after each line, we must build our library function, by using JavaScript `setTimeout` or other Node.js operators like `process.nextTick` and `setImmediate`.

## 6.8 The net library

Much of the Node core API has to do with creating services that listen to specific types of communications.

## Sockets

A socket is an endpoint in a communication, and a network socket is an endpoint in a communication between applications running on two different computers on the network. The data flows between the sockets in what's known as a stream. The data in the stream can be transmitted as binary data in a buffer, or in Unicode as a string. Both types of data are transmitted as packets: parts of the data split off into specifically sized pieces. There is a special kind of packet, a finish packet (FIN), that is sent by a socket to signal that the transmission is done. How the communication is managed, and how reliable the stream is, depends on the type of socket created.

### 6.8.1 TcpServer. Let us introduce a TCP server that performs the echo of the received data.

```
1  /*
2  * =====
3  * TcpServerNode.js
4  * =====
5  */
6  var net = require('net');
7  var port = 23;
8  var server = net.createServer(
9    function(conn) {
10     console.log('connected');
11     conn.write('TCP node server READY' );
12     conn.on('data', function (data) {
13       console.log(data + ' from ADDR=' + conn.remoteAddress + ' PORT=' + conn.remotePort);
14       conn.write('Echo:' + data );
15     });
16     conn.on('close', function() {
17       console.log('client closed connection');
18     });
19   }).listen(port, function(){ console.log('bound to port ' + port); } );
20
21 internalWork = function( ){
22   console.trace( );
23   setImmediate( function(){ internalWork( ) ; } );
24 }
25 //main
26 console.log('TcpServerNode START ' );
27 internalWork( );
28
29 /*
30 USAGE:
31 netcat 192.168.251.1 8050
32 telnet 127.0.0.1 (on port 23)
33 */
```

Listing 1.34. TcpServerNode.js

A callback function is attached to the two events via the `on` method. Many objects in Node that emit events provide a way to attach a function as an event listener by using the `on` method. This method takes the name of the event as first parameter, and the function listener as the second.

## allowHalfOpen

There is an optional parameter for `createServer`: *allowHalfOpen*. Setting this parameter to `true` instructs the socket not to send a FIN when it receives a FIN packet from the client. Doing this keeps the socket open for writing (not reading). To close the socket, you'd then need to explicitly use the `end` method. By default, *allowHalfOpen* is false.

### 6.8.2 TcpClient. Let us introduce a TCP client that sends some data to the server and waits for the answer.

```
1  /*
2  * =====
3  * TcpClientNode.js
4  * =====
5  */
6  var net = require('net');
7  var client = new net.Socket();
8  client.setEncoding('utf8');
9
```



---

```

10 // connect to server
11 client.connect ('8050','localhost', function () {
12     console.log('connected to server');
13     client.write('Line1');
14 });
15
16 // prepare for input from terminal
17 process.stdin.resume();
18 // when receive data, send to server
19 process.stdin.on('data', function (data) {
20     client.write(data);
21 });
22 // when receive data back, print to console
23 client.on('data',function(data) {
24     console.log(data);
25 });
26 // when server closed
27 //client.on('close',function() {
28 //console.log('connection is closed');
29 //});

```

**Listing 1.35.** *TcpClientNode.js*

## 6.9 The http library

Let us create a Node HTTP server that uses a callback to define response logic:

```

1  /*
2  * =====
3  * HttpServerBase.js
4  * =====
5  */
6  var http = require("http");
7
8  http.createServer(function(request, response) {
9      //The request object is an instance of IncomingMessage (a ReadableStream and it's also an EventEmitter)
10     var method = request.method;
11     var url = request.url;
12     console.log("Server request method=" + method + " url="+ url);
13     if (request.method === 'GET' && request.url === '/') {
14         response.writeHead(200, {"Content-Type": "text/plain"});
15         response.write("Hello World from the server");
16         response.end();
17     }
18 }).listen( 8080, function(){ console.log('bound to port 8080');} );
19
20 console.log('Server running on 8080');

```

**Listing 1.36.** *HttpServerBase.js*

The *listen* method tells the HTTP server object to begin listening for connections on the given port. Node doesn't block, waiting for the connection to be established. When the connection is established, a listening event is emitted, which then invokes the callback function, outputting a message to the console.

When processing a web request from a browser, that browser may send more than one request. For instance, a browser may also send a second request, looking for a *favicon.ico*.

The result of *http://localhost:8080/* is **Hello World from the server** in the browser page, while the output of the program is:

```

1  /*
2  Server running on 8080
3  bound to port 8080
4  Server answers to GET url=/
5  Server answers to GET url=/favicon.ico
6  */

```

**Listing 1.37.** *Execution of HttpServerBase.js*

### 6.9.1 HTTP server that returns the content of a JSON file.

Let us create now a Node HTTP server that uses a callback to return the content of a Json file, for example the following one:

```
1 { "name":"John", "age":30, "city":"New York"}
2 { "name":"Alice", "age":24, "city":"Los Angeles"}
```

Listing 1.38. a.json

#### JSON

Json (*JavaScript Object Notation*) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

```
1 /*
2  * =====
3  * HttpServerFile.js
4  * =====
5  */
6 var http = require('http');
7 var fs = require('fs');
8
9 http.createServer(function (request, response) {
10     var method = request.method;
11     var url = request.url;
12     console.log("Server request method=" + method + " url=" + url);
13     if (request.method === 'GET' && request.url === '/') {
14         response.writeHead(200, {'Content-Type': 'text/plain'});
15
16         //Read JSON file and use callback to define what to do with its contents
17         fs.readFile('./a.json', 'utf8', function(err, data) {
18             if (err){
19                 response.write('Could not find or open file for reading\n');
20             }else{
21                 console.log("data=" + data );
22                 response.write(data);
23             }
24             response.end();
25         });
26     }
27 }).listen(8123, function() { console.log('bound to port 8123');});
28
29 console.log('Server running on 8123');
```

Listing 1.39. HttpServerFile.js

The result of `http://localhost:8123/` in the browser page is the content of the file `a.json`, while the output of the program is:

```
1 /*
2 Server running on 8123/
3 bound to port 8123
4 Server request method=GET url=/favicon.ico
5 Server request method=GET url=/
6 data={ "name":"John", "age":30, "city":"New York"}
7 { "name":"Alice", "age":24, "city":"Los Angeles"}
8
9 Server request method=GET url=/favicon.ico
10 */
```

Listing 1.40. Execution of HttpServerFile.js

### 6.9.2 Http CRUD Server.

Let us suppose now to create a Node HTTP server that executes typical create, read, update and delete (*CRDD*) actions. In the example that follows we implement create and read actions: the server stores data sent with a POST request and returns the list of stored data by answering to a GET request.

```

1  /*
2  * =====
3  * HttpServer.js
4  * =====
5  */
6  var http = require("http");
7  var datastore = []; //Array of Buffers
8
9  http.createServer(function(request, response) {
10     //The request object is an instance of IncomingMessage (a ReadableStream; it's also an EventEmitter)
11     var headers = request.headers;
12     var method = request.method;
13     var url = request.url;
14     if( method == 'GET'){
15         datastore.forEach( function(v,i){
16             response.write( i + " " + v + "\n")
17         });
18         response.end();
19     }//if GET
20     if( method == 'POST'){
21         var item = '';
22         request.setEncoding("utf8"); //a chunk is a utf8 string instead of a Buffer
23         request.on('error', function(err) {
24             console.error(err);
25         });
26         request.on('data', function(chunk) { //a chunk is a byte array
27             item = item + chunk;
28         });
29         request.on('end', function() {
30             //dataStore = Buffer.concat(dataStore).toString();
31             datastore.push(item);
32             console.log("dataStore=" + datastore);
33             response.on('error', function(err) { console.error(err); });
34             response.statusCode = 200;
35             response.setHeader('Content-Type', 'application/json');
36             //response.writeHead(200, {'Content-Type': 'application/json'}) //compact form
37             var responseBody = {
38                 //headers: headers, //comment, so to reduce output
39                 method: method,
40                 url: url,
41                 datastore: datastore
42             };
43             response.write( JSON.stringify(responseBody) );
44             response.end();
45             //response.end(JSON.stringify(responseBody)) //compact form
46         });
47     }//if POST
48 }).listen( 8080, function(){ console.log('bound to port 8080');} );
49
50 console.log('Server running on 8080');
51
52
53 /*
54 Server running on 8080
55 bound to port 8080
56 Server answers to GET url=/
57 Server answers to GET url=/favicon.ico
58 */

```

**Listing 1.41.** Execution of HttpServer.js

### 6.9.3 cURL

To test our server, we can use a powerful command-line HTTP-client that can be used to send requests in place of a web browser. Thus, we have to download curl (curl donwload win64) and then execute some command; for example:

```

1  curl -d d1 http://localhost:8080 --d sets the request method to POST
2  {"method":"POST","url":"/","dataStore":["d1"]}
3  curl -d d2 http://localhost:8080

```

---

```
4  {"method":"POST","url":"/","dataStore":["d1","d2"]}
5  curl -d d3 http://localhost:8080
6  {"method":"POST","url":"/","dataStore":["d1","d2","d3"]}
7
8  curl -d data1 localhost:8080 //GET
9      0)d1
10     1)d2
11     2)d3
```

---

## 7 Node on RaspberryPi

As of the November 2015 version of Raspbian Jessie, Node-RED comes preinstalled on the SD card image that can be downloaded from *RaspberryPi.org*.

From [www.npmjs.com/package/pi-gpio](http://www.npmjs.com/package/pi-gpio) we read: Raspbian has node installed, but it's quite old. To get to a more recent version:

```
pi@raspberrypi:~ $ node -v
v0.10.29
pi@raspberrypi:~ $ sudo su -
root@raspberrypi:~ # apt-get remove nodered -y
root@raspberrypi:~ # apt-get remove nodejs nodejs-legacy -y
root@raspberrypi:~ # apt-get remove npm -y # if you installed npm
root@raspberrypi:~ # curl -sL https://deb.nodesource.com/setup_5.x | sudo bash -
root@raspberrypi:~ # apt-get install nodejs -y
root@raspberrypi:~ # node -v
v5.12.0
root@raspberrypi:~ # npm -v
3.8.6
exit
npm install onoff --save
```

### 7.1 Blink a Led with JavaScript

From <http://webofthings.org/2016/10/23/node-gpio-and-the-raspberry-pi/> we read: You can find a dozen Node.js GPIO libraries for the Pi, offering different abstraction layers and functionality. We decided to use one called `onoff`.

```
npm install onoff --save
```

The 'Hello World' equivalent of the IoT is to make a real LED blink:

```
1  /*
2  * =====
3  * ledGpio.js
4  * =====
5  */
6
7  var onoff = require('onoff'); //A
8
9  var Gpio = onoff.Gpio,
10  led = new Gpio(4, 'out'), //B
11  interval;
12
13  interval = setInterval(function () { //C
14    var value = (led.readSync() + 1) % 2; //D
15    led.write(value, function () { //E
16      console.log("Changed LED state to: " + value);
17    });
18  }, 2000);
19
20  process.on('SIGINT', function () { //F
21    clearInterval(interval);
22    led.writeSync(0); //G
23    led.unexport();
24    console.log('Bye, bye!');
25    process.exit();
26  });
27
28  // #A Import the onoff library
```

---

```
29 // #B Initialize pin 4 to be an output pin
30 // #C This interval will be called every 2 seconds
31 // #D Synchronously read the value of pin 4 and transform 1 to 0 or 0 to 1
32 // #E Asynchronously write the new value to pin 4
33 // #F Listen to the event triggered on CTRL+C
34 // #G Cleanly close the GPIO pin before exiting
```

**Listing 1.42.** Execution of `ledGpio.js`

More on this can be found in `blsJavaScript.pdf`.